LESSON: GR1: Strongly Connected Components
***
(Slide 1) Graph Algorithms
The next section is about graph algorithms. I'm sure you've all seen the basic graph algorithms before: DFS for depth-first search, BFS for breadth-first search and Dijkstra's single source shortest path algorithm. We'll do a quick review of how DFS is used to find the connected components of an undirected graph, and then we'll build on that to look at connectivity in directed graphs.

We'll use DFS to find the strongly connected components of directed graphs. These are the analog of connected components in directed graphs. And then we'll see an application of our strongly connected component algorithm to solve the 2-SAT problem.

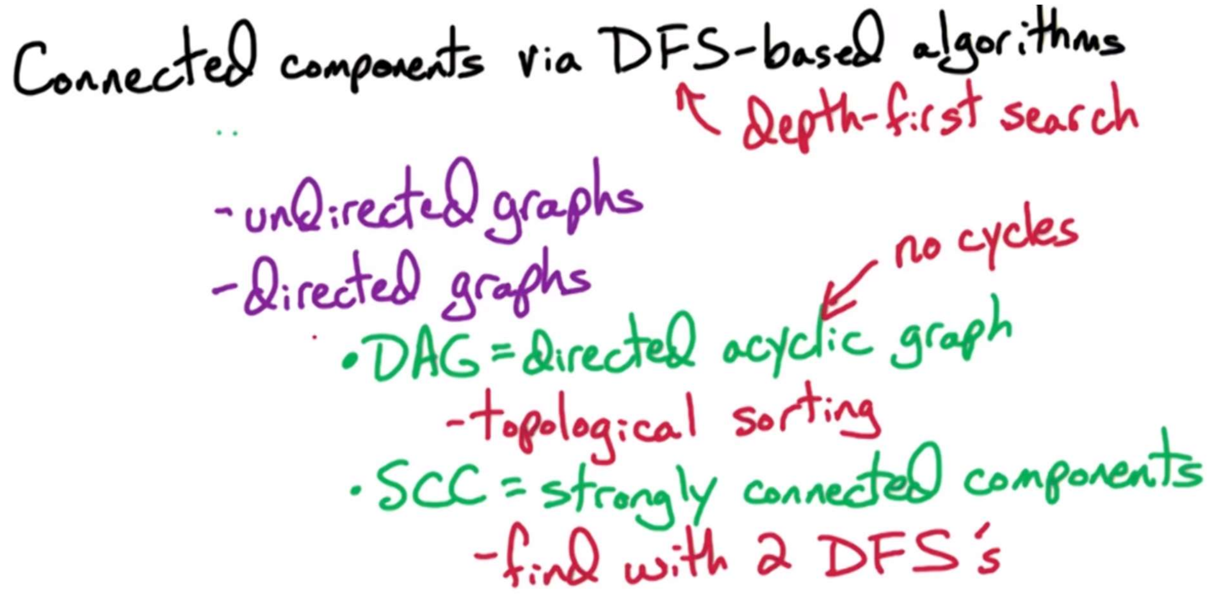
Next, we'll look at the Minimum Spanning Tree problem: MSTs. You've likely seen Kruskal's and Prim's algorithm before for finding an MST. We'll look at the correctness behind these algorithms.

Finally, we'll look at the PageRank algorithm. This is the algorithm that looks at the Webgraph and assigns weights to vertices or webpages. It's a measure of their importance. This algorithm was devised by Brin and Page, and it's at the heart of Google's search engine.

To understand the PageRank algorithm, I'll first give you a quick primer on Markov chains, and then you'll see how it relates to strongly connected components.

***

(Slide 2) Outline



Connected components via DFS-based algorithms

↖ Depth-first search

- undirected graphs
- directed graphs
  - DAG = directed acyclic graph ← no cycles
    - topological sorting
  - SCC = strongly connected components
    - find with 2 DFS's

There is a footnote on this slide: See [DPV] Chapter 3 (Decompositions of graphs) and Eric's notes: https://cs6505.wordpress.com/schedule/scc/

In this lesson, we're going to look at connectivity algorithms using algorithms based on DFS. DFS of course, refers to Depth-First Search. We'll start off by reviewing DFS for undirected graphs and look at the algorithm for determining connected components in undirected graphs. The algorithm is probably familiar to many of you. After that, we'll look at DFS for directed graphs and our goal is to determine the analog of connected components for directed graphs.

We begin by looking at DAGs. DAGs are Directed Acyclic Graphs. Acyclic means that it has no cycles. We'll see how to topologically sort DAGs. What this means is that we can order the vertices, say from left to right, so that all edges go left to right.

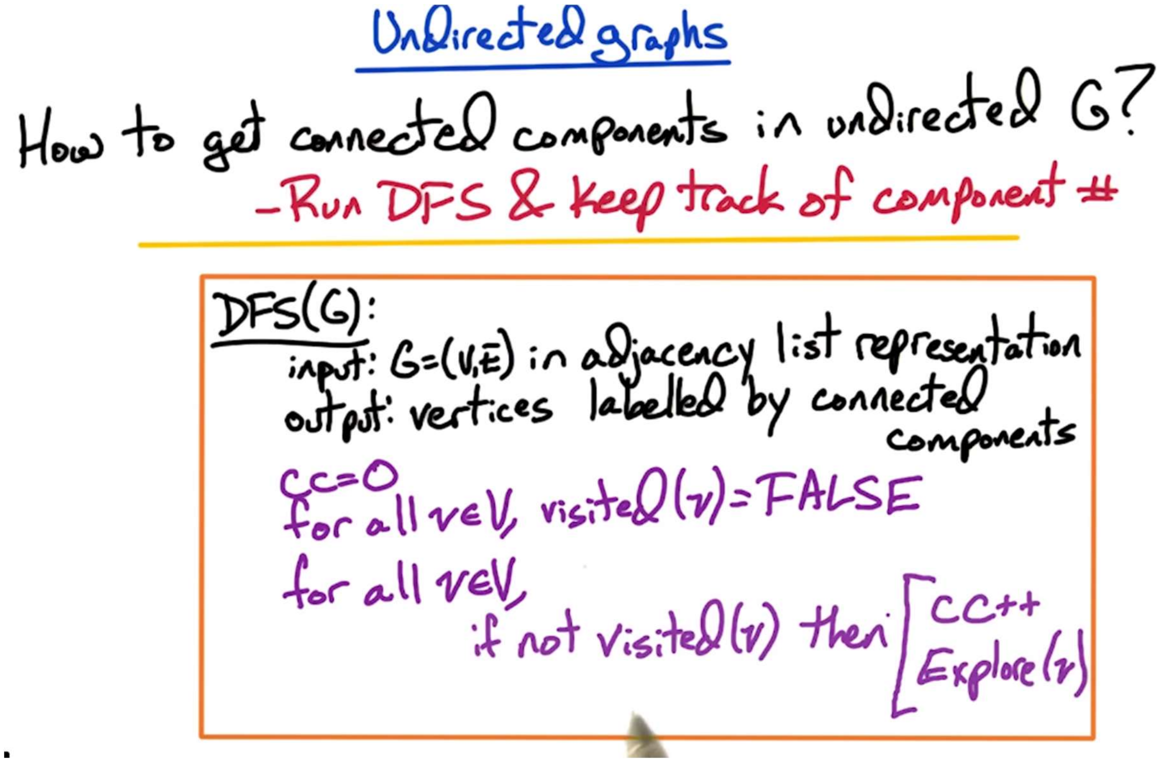
Now, this algorithm may be familiar to many of you, but we'll use it to derive some intuition for our more sophisticated algorithms for general directed graphs. For general directed graphs, we are going to be looking to find the SCCs. These are the Strongly Connected Components. This is the analog of connected components for directed graphs.

The algorithm for finding SCCs is really sweet. It's just two DFS's. Actually, it's the same DFS algorithm we found before for undirected graphs and we just run it two times and we'll find thestrongly connected components of any directed graph. Now, it's a very simple algorithm but we're going to get there a bit slowly. We're going to go through all of these steps in order to derive some intuition before we get to this more general algorithm for general directed graphs.

So let's start with undirected graphs and let me remind you about the DFS algorithm that you've probably seen a million times.

***

(Slide 3) Undirected graphs



Undirected graphs

How to get connected components in undirected G?

- Run DFS & keep track of component #

```
DFS(G):
    input: G=(V,E) in adjacency list representation
    output: vertices labelled by connected components
    cc=0
    for all v∈V, visited(v)=FALSE
    for all v∈V,
        if not visited(v) then
            cc++
            Explore(v)
```

For a given undirected graph, how do we find it's connected components? Well, we simply run DFS and we keep track of the component number. So, each vertex is going to have its component number stored.

Let me remind you of this pseudocode for DFS, because we're going to make some tweaks to it during the course of this lecture:

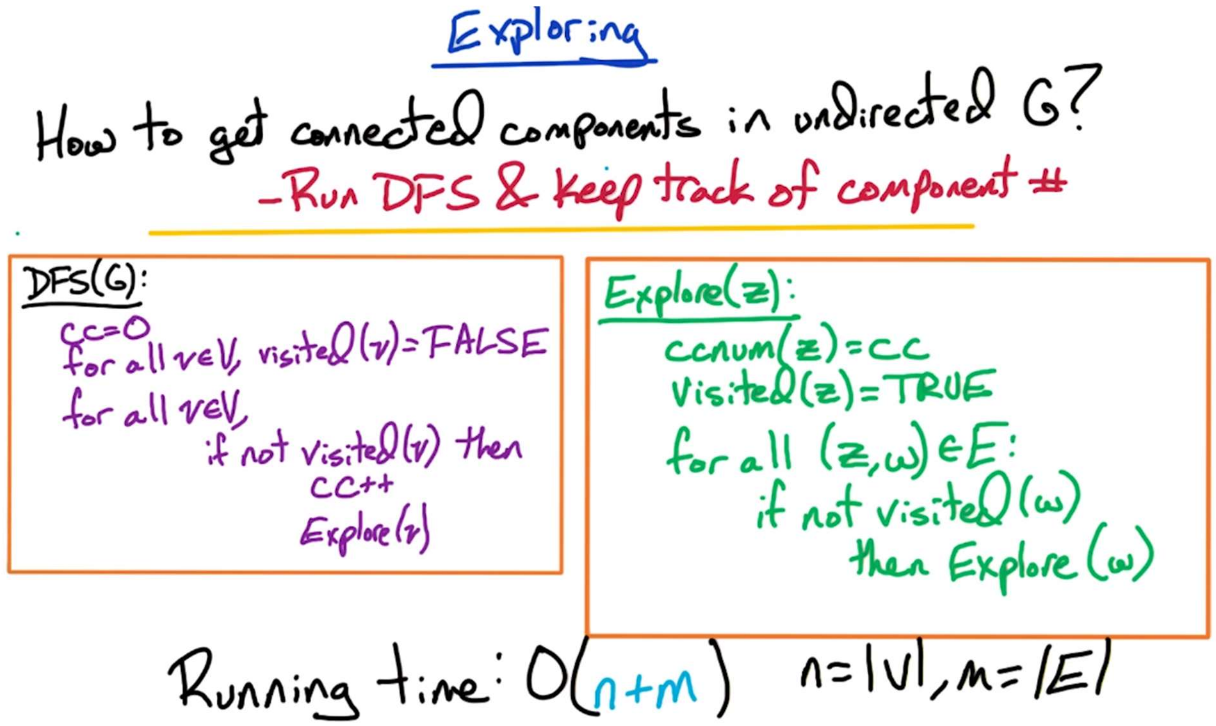
- So, the DFS algorithm takes as input a graph, G. For now, let's think of G as an undirected graph. But later, we're going to run the same algorithm on a directed graph - it's going to be identical pseudocode. And we assume that the graph is given to us in adjacency list representation.
- And for now, we're looking at connected components of an undirected graph. So, the vertices are going to be labeled by a connected component number. We're going to have a counter which is the current connected component number.
- We're going to have an array which keeps track of whether we visited a vertex yet or not, and we're going to start off by initializing the visited array to false for all vertices of the graph.

Now, we go through the vertices in an arbitrary order. If we get to a vertex that we haven't visited yet, what do we do? - well, this means that we found a new connected component. So, we increment the connected component number and then we start exploring from this vertex.

And let's now look at the subroutine for explore.

***

(Slide 4) Exploring Undirected Graphs



Exploring

How to get connected components in undirected G?

-Run DFS & keep track of component #

```
DFS(G):
    cc=0
    for all v∈V, visited(v)=FALSE
    for all v∈V,
            if not visited(v) then
                cc++
                Explore(v)
```

```
Explore(z):
    ccnum(z)=cc
    visited(z)=TRUE
    for all (z,w)∈E:
        if not visited(w)
            then Explore(w)
```

Running time: $O(n+m)$ $n=|V|, m=|E|$

Let's look at the pseudocode for the Explore procedure, and let's say we're running Explore from a vertex Z.

This is our first time visiting x. So we have to store its connected component number as the current count for the connected components, and we have to set z to be visited (visited(z) = True). Now, we want to explore all edges out of z (Recall that G was given to us an adjacency list representation, so now we can look through the linked list of neighbors of z). For a particular neighbor w, if w hasn't been visited yet then we recursively explore from w and we repeat this procedure.

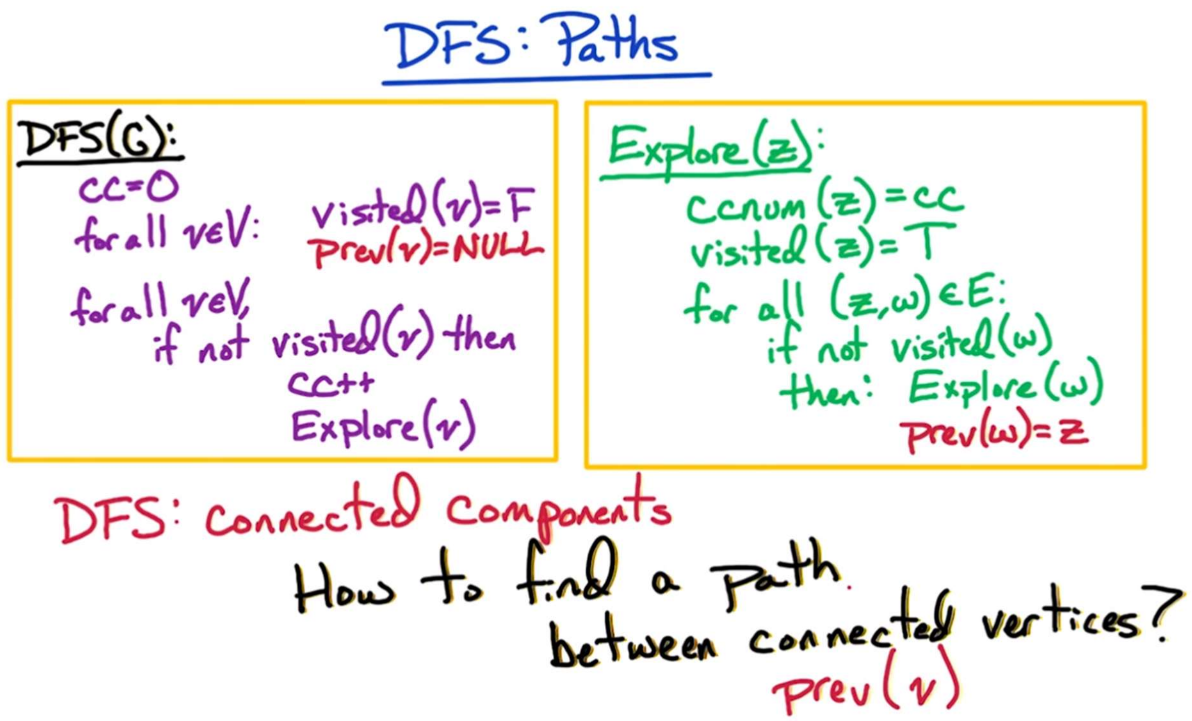
Now what's the running time of this algorithm? Hopefully, you recall that DFS is a linear time algorithm. The running time is O(n+m).

For undirected graphs, this is it. This gets all the information that we're trying to glean … the connected components of the graph.

Now we're going to turn to directed graphs and we're going to need more information from DFS in order to obtain connectivity information.

***

(Slide 5) DFS: Paths



DFS: Paths

```
DFS(G):
    cc=0
    for all v∈V:  visited(v)=F
                  prev(v)=NULL
    for all v∈V,
        if not visited(v) then
            cc++
            Explore(v)
```

```
Explore(z):
    ccnum(z)=cc
    visited(z)=T
    for all (z,w)∈E:
        if not visited(w)
        then: Explore(w)
              prev(w)=z
```

DFS: connected components

How to find a path between connected vertices?

prev(v)

We saw in the last slide how to use the DFS algorithm to find the connected components of an undirected graph. Before we move on to directed graph, let's glean a little bit more information from the DFS algorithm.

In particular, suppose I have a pair of vertices, v and w which are in the same connected component. I want to find a path between this pair of vertices. To do that, we simply have to keep track of the predecessor vertex (prev(w) = z) when we first visit a vertex.

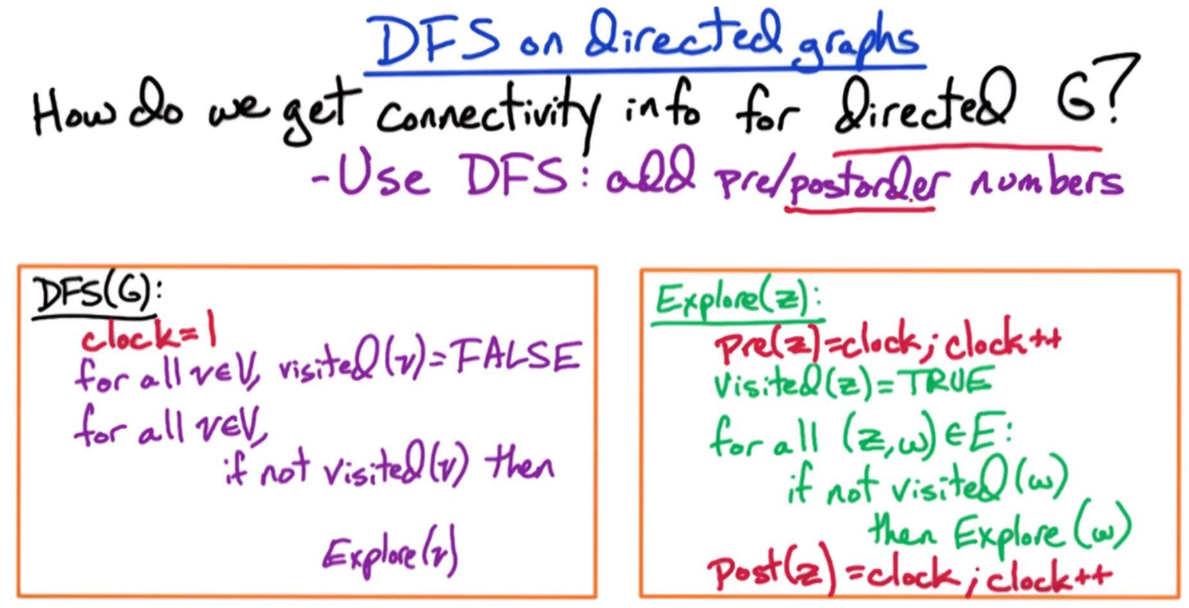
Here's the DFS algorithm once again for finding connected components of an undirected graph. We're going to use this previous array as is used in Dijkstra's algorithm to keep track of the predecessor vertex. We initialize this previous array to null for every vertex. And when we first visit a vertex - so, at this point in the algorithm - we should set its previous array to its predecessor vertex, which is vertex z. So, we set prev(w) = z.

Now, after running DFS algorithm, given this previous array, we can use this previous array to backtrack. So, for a pair of vertices which are in the same connected component, we can use the previous array to find a path between this pair of connected vertices.

That completes our discussion of DFS algorithms for undirected graphs. Let's move on now to directed graphs.

***

(Slide 6) DFS on Directed Graphs



DFS on directed graphs

How do we get connectivity info for directed G?

-Use DFS: add pre/postorder numbers

```
DFS(G):
    clock=1
    for all v∈V, visited(v)=FALSE
    for all v∈V,
        if not visited(v) then
            Explore(v)
```

```
Explore(z):
    pre(z)=clock; clock++
    visited(z)=TRUE
    for all (z,w)∈E:
        if not visited(w)
            then Explore(w)
    post(z)=clock; clock++
```

We saw how to determine the connected components for an undirected graph. Now, let's take a look at directed graphs.

How do we determine the connectivity properties for a directed graph? Once again, we're going to use a DFS based approach; but, for directed graphs, we're going to need additional information from the DFS.

The additional info that we use are the preorder or postorder numbers for the tree or forest of explore the edges. The algorithm is going to be a slight variant of DFS that we saw just before. So, let's look at that previous algorithm and just modify it a little bit.

Here's a DFS algorithm for figuring out the connected components of an undirected graph. Our basic algorithm is going to be the same, but we no longer need to keep track of the connected component number, so let's remove those lines.

So. I've removed those three lines which talked about the connected component number. Now, I want to add in lines which take care of the preorder and postorder numbers. In order to keep track of the preorder and postorder numbers, we're going to add in a clock.

The preorder number for a vertex z - it's going to be the value of the clock when we first visit vertex Z - and the postorder number - it's going to be the value of the clock at the time when we finished exploring vertex z.

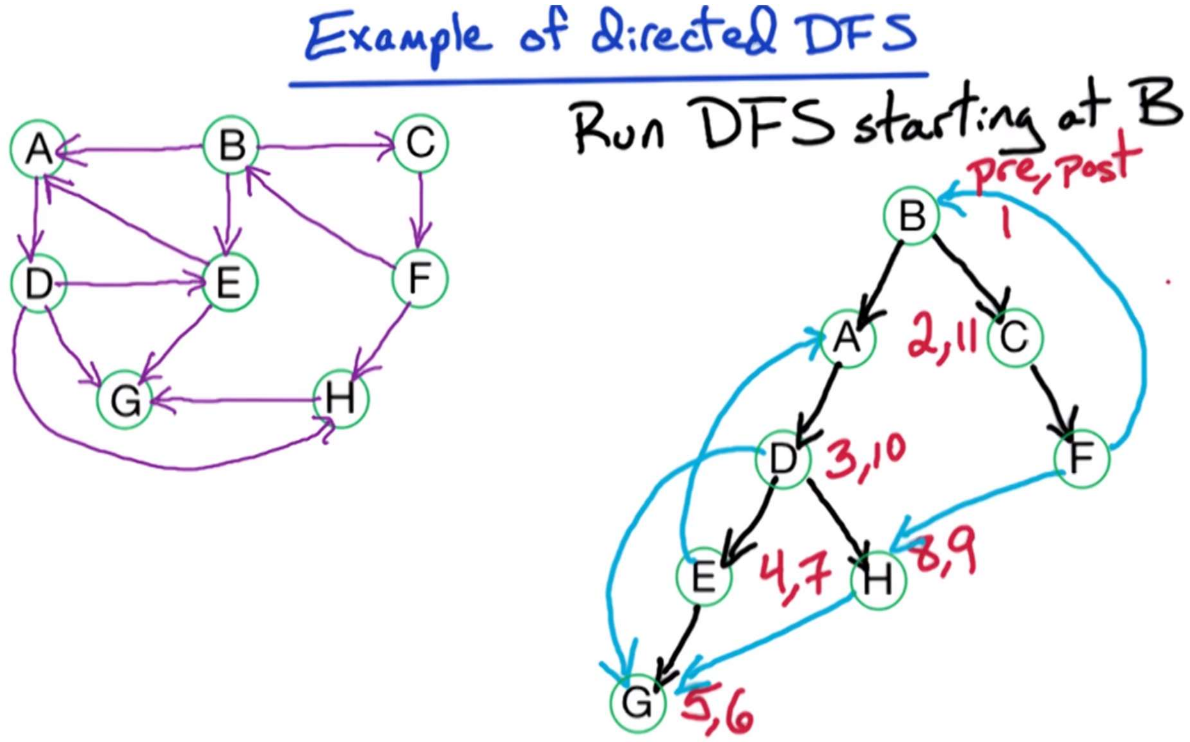
So we looked at all edges out of z. First, we need to initialize the clock to 1. When we first visit a vertex z, we can store the preorder number, which is the current value of the clock. After we do this, we need to increment the clock.

Finally, when we finish exploring vertex z, then we can set its postorder number to be the current value of the clock. And then once again we have to increment the value of the clock.

That gives us DFS for undirected graphs. Now we want to see the properties of these preorder and postorder numbers. And actually for our connectivity algorithms, we're simply going to use the postorder numbers. And to be perfectly honest, I'm not even sure where preorder numbers come into play. But, just in case. I kept the preorder number in here in the algorithm.

***

(Slide 7) Directed DFS: Example



Here's an example of a specific directed graph on eight vertices.

Let’s run our DFS algorithm for directed graphs on this specific example starting at the vertex B. And, for concreteness, we'll assume that the linked lists are stored in alphabetical order.

So, when we look at the edges out of B for example, we're going to see A and C then E.   And let's look at the tree of explored edges in our DFS run. We're going to start at the vertex B - this is going to be the root of the tree. And let's keep track of the preorder and postorder numbers of the vertices as they are stored.

We start exploring from B, so it gets a preorder number. We first see neighbor A of vertex B. So the next edge that we explore is the edge B to A. We then assign a preorder number to A. We then see vertex D and we give it its preorder number. Then exploring from vertex D we see vertex E and then from vertex E, we see vertex G.   G gets preorder number five. But notice from G, there's nothing to explore. So, we're going to pop back up from G - back up to E. So we're going to finish exploring from G. So, we can give it postorder number 6.

Now notice from Vertex E … when we do Explore from E … it was an edge from E to A. A has already been explored - has already been visited at that time. So, we're not going to rerun explore from A - this is not an edge in the DFS tree since A has already been explored at this time. We're not going to rerun Explore from A but let's keep track of this edge and mark it as a blue edge in order to distinguish it from the explored edges which are marked as black edges.

After we've explored the two edges out of E - to A and to G - then we're going to pop back up to Vertex D. But first we're going to assign a postorder number to E.

Now from D we're going to see this edge to H, so we're going to explore from H. From H, we see this neighbor G. G has already been visited, so let's mark this edge from H to G as a blue edge. Then we're going to finish exploring from H and we're going to pop back up to D. Lets assign its pre or postorder numbers.

From D there was an additional edge to G. Then we can assign its postorder number and pop back up to A. A is done and then we can pop back up to B.

From vertex B, we're going to see vertex C and then F, so we're going to get this right sub-tree.

Now, from F, we're going to see this edge back to B - that's a blue edge - and we're going to see this edge across to H and we can save the preorder and postorder numbers.

And that completes the DFS run for this example.

***

(Slide 8) Types of Edges

Types of edges

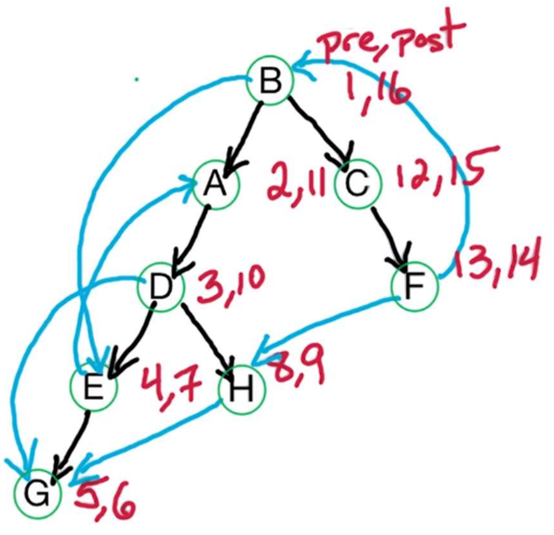
Edge $z \to w$:

Tree edge: $B \to A$, $A \to D$, …
$post(z) > post(w)$

Back: $E \to A$, $F \to B$
$post(z) < post(w)$

Forward: $D \to G$, $B \to E$
$post(z) > post(w)$

Cross: $F \to H$, $H \to G$
$post(z) > post(w)$



Now let's look at a specific edge of the graph. Let's say it goes from vertex z to vertex w. Now, that edge is going to appear in this graph over here of the DFS tree. It's either going to appear as a black edge - an explored edge - or as a blue edge.

Let's look at the properties of this edge based on whether it's a black edge or it's a blue edge and what type of blue edge.

Now, the black edges correspond to explored edges or edges of this tree - this DFS tree. Notice that, in this example, the DFS tree happened to be a tree. Every vertex happens to be reachable from the vertex B. That doesn't have to be the case. It could be a forest. We could have multiple components here. So, actually this should be called “forest edges”, but let's ignore that. We’ll call this a DFS tree, even though it might be a forest.

Some examples of these tree edges are B to A, A to D, and so on. All of the black edges are tree edges. Now, let's look at the properties of the postorder numbers for these tree edges.

Let's take the example of A to D. So we started exploring from A. Then we saw this neighbor D. So, we recursively started exploring from D. We're going to finish off D, and assign a postorder number, and then we're going to pop back to A. So, A is going to finish after D.

So, the postorder number of A is going to be bigger than the postorder number of D, because it's going to finish later. And in general, the postorder number of the head of this edge is going to be bigger than the postorder number of the tail of this edge if it's a tree edge.

Now, let's look at the blue edges. There's going to be three types of blue edges: Back edges, forward edges, and cross edges. The examples of back edges in this graph are E -> A and F -> B. The edge goes from a descendant to an ancestor. These are edges that go back up the tree.

Now, let's take a specific example. Let's say E -> A. Notice we're going to finish off E before we finish off A. In this case, the postorder number of E is going to be smaller than the postorder number of A. And, in general, for back edges, the postorder number of the head of the edge is going to be smaller than the postorder number of the tail of the edge. This is opposite of the case of the tree edges.

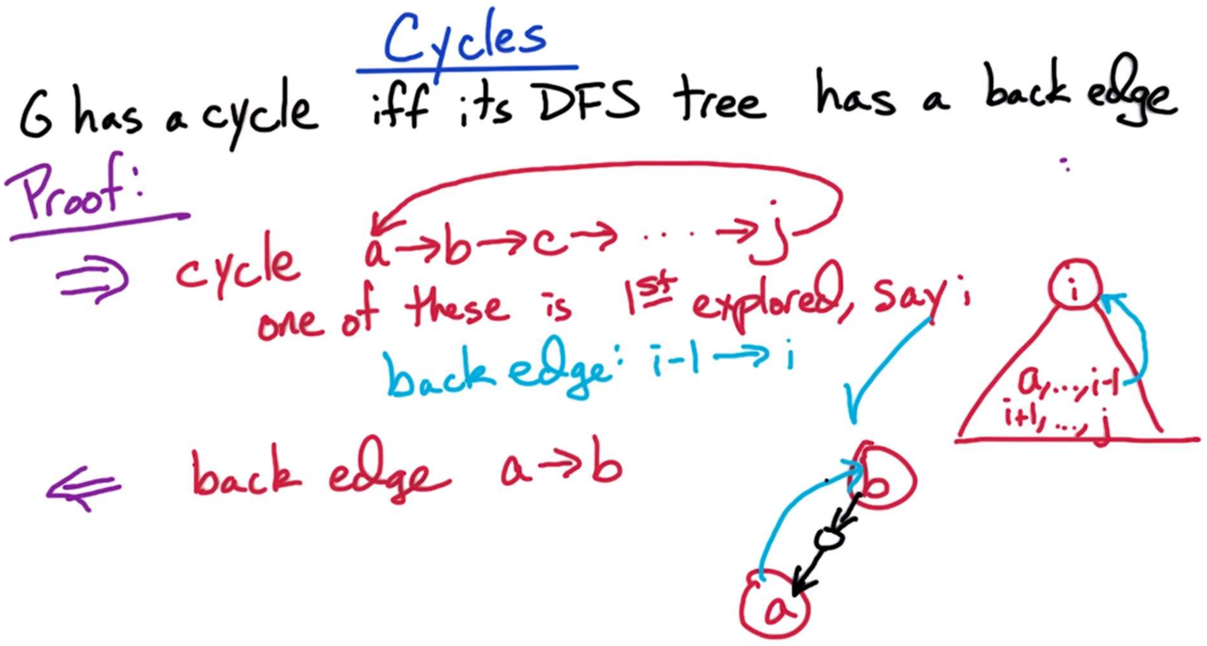
Let's look at forward edges. These are going down the tree. For example, D to G. Another example of a forward edge is this edge, B -> E - I forgot to notice this edge earlier when we were running DFS on our earlier example. Now, what do you notice about the postorder numbers for forward edges? Well, these behave just like tree edges. Tree edges are going down one depth. Forward edges are going down multiple depths. But they have the same key property - the postorder number goes down.

Finally, we have cross edges. For example, F -> H and H -> G. These are pairs of vertices that have no ancestor descendant relation to each other. Look at this edge from F -> H. H must have been explored before F. Otherwise, H would be in the subtree of F. Since this is an unexplored edge, that means that H was finished first - the postorder number of H is going to be smaller than the postorder number of F. Once again, the postorder number goes down with the edge.

Now, the key property for back edges is that the postorder number goes up. For all other edges, the other three types of edges - tree edges, forward edges, and cross edges, the postorder number goes down. Now, that is the key property that we need for postorder numbers. Back edges behave differently than the other three types of edges.

***

(Slide 9) Cycles



Cycles

G has a cycle iff its DFS tree has a back edge

Proof:

$\Rightarrow$ cycle $a \to b \to c \to \cdots \to j$

one of these is 1st explored, say i

back edge: $i-1 \to i$

$\Leftarrow$ back edge $a \to b$

Now, let's look at properties of the graph and how these properties manifest themselves in the DFS tree.

Let's look now at cycles. How does the cycle manifest itself in the DFS tree? The key property is that graph G has a cycle if and only if its DFS tree has a back edge.

Now, it doesn't matter where we start the DFS. What is the starting vertex? It doesn't matter how the vertices are ordered in the adjacency list representation also. For any start vertex, for any ordering on the vertices … for any ordering on the neighbors of every vertex, the DFS tree will contain a back edge if and only if G has a cycle. So, if there is a cycle, there will be a back edge that will appear in our DFS tree. And if the DFS tree contains a back edge, then there is a cycle in the graph. Let's see why this property holds.

This is an equivalence relation. So let's look at the two implications:

- Let's look first at the forward implication. Let's suppose that G has a cycle and let's see how a back edge appears. Let's suppose the graph G has a cycle and let's label the vertices of that cycle as a, b, c, up to j.

  So, there's an edge from a -> b, b -> c, up to j, and then back to a. Now, one of these vertices has to be explored first. There always has to be somebody first. So, let's say the first vertex is vertex i. So, what do we know then about our DFS tree? We know we have this vertex i. And now if we look at the sub-tree of i, we know that all these other vertices of the cycle are reachable from i. So, they're all going to lie in the sub-tree rooted at i. So, all the other vertices of the cycle are going to be contained in this sub-tree rooted at i. We don't know anything else about the structure of this subtree, we just know that this subtree contains all the other vertices of this cycle because they're all reachable from vertex i. Now at least one of these vertices has an edge to i. In this case, we know that i - 1 has an edge to i. This edge is going to appear as a back edge because it goes from a descendant to an ancestor. So we're going to have a back edge from i-1 to i.

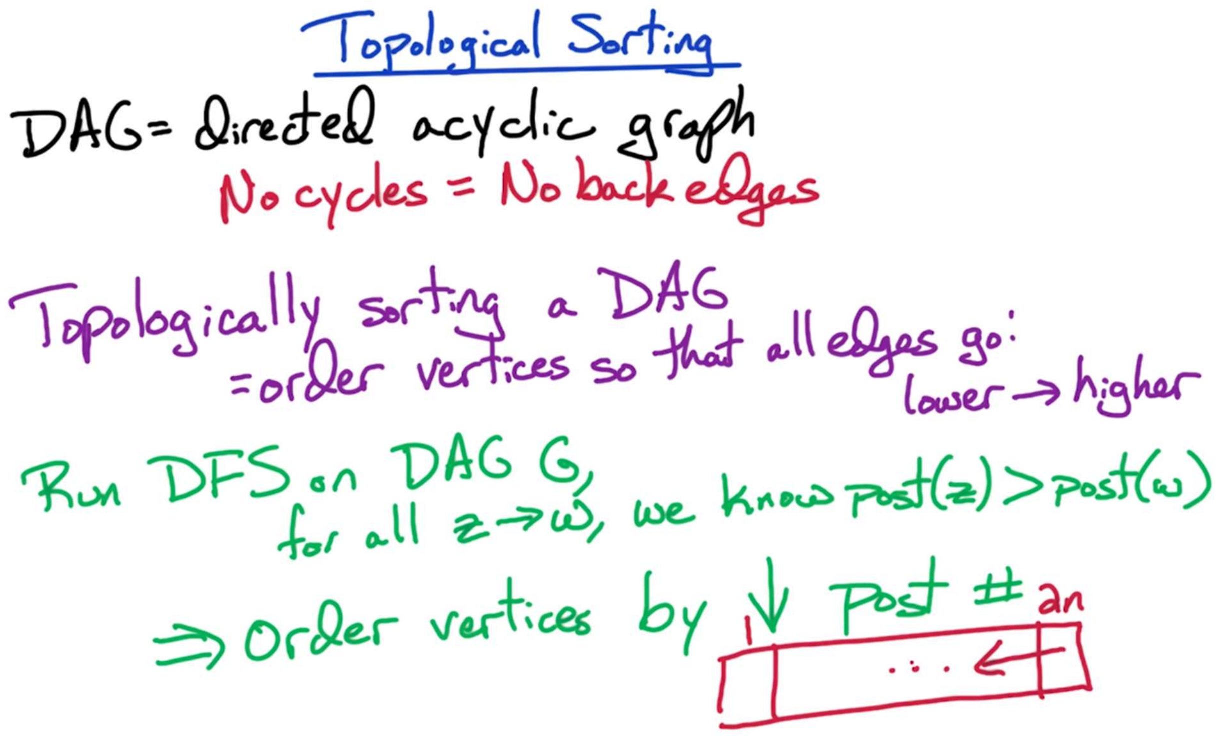
- Now, let's look at the reverse implication. Let's suppose that our DFS tree has a back edge and let's prove that the graph then must contain a cycle.

  Let's say there is a back edge from vertex a to vertex b. So, what do we know? We know that vertex a is a descendant of vertex b, and there's this back edge from a to b. But since a is a descendant of b in this DFS tree, we know there's a sequence of tree edges which go from b down to a. And notice we now have our cycle.

  These tree edges are edges in the graph and then this back edge is also the edge of the graph. So, we have our cycle from b down to a and then back to b. That shows that there is a cycle. For every back edge, there is a cycle. That proves this property.

***

(Slide 10) Topological Sorting



Topological Sorting

DAG = directed acyclic graph

No cycles = No back edges

Topologically sorting a DAG
= order vertices so that all edges go:
lower $\rightarrow$ higher

Run DFS on DAG G,
for all $z \rightarrow w$, we know $post(z) > post(w)$

$\Rightarrow$ Order vertices by $\downarrow$ Post #

$1 \quad \ldots \leftarrow n$

Now let's take a look at DAGs. These are directed acyclic graphs. Acyclic means that there are no cycles in the graph.

What we just saw is that the graph has a cycle if and only if the DFS tree has a back edge. Since there are no cycles, there's going to be no back edges in our DFS tree.

Now what we're going to try to do is, we're going to try to topologically sort that DAG. What does that mean? - we're going to order the vertices so that all edges go from lower order number vertex to a higher order number of vertex.

So for instance, if we write down the vertices in order from lowest to highest, then all edges are going to go from left to right. We're not going to have any edges going backwards.

To topologically sort this DAG, what we're going to do is … we're just going to run DFS on this DAG.

What is the key property we know for this DAG? We know it has no back edges. What do we know about the postorder numbers for all other types of edges?

- We know that for back edges, the postorder number increases along the edge.

- For all other types of edges, the postorder number goes down along the edge.

Now we want to order the vertices from lowest to highest so that all edges go left to right. So which vertex do we want to put first? We want to put the vertex with highest postorder number first and therefore all edges are going to go from higher postorder number to lower postorder number, because there are no back edges.

Just to summarize, we know for every edge of the graph – so, for instance, this edge from z to w - we know that the postorder number of z is greater than the postorder number of w.

So in order to topologically sort the graph, we order the vertices by decreasing postorder number. The highest postorder number comes first and the lowest post or number comes last.

So in order to topologically sort a DAG, we just have to do one run of DFS and then sort by decreasing postorder number.

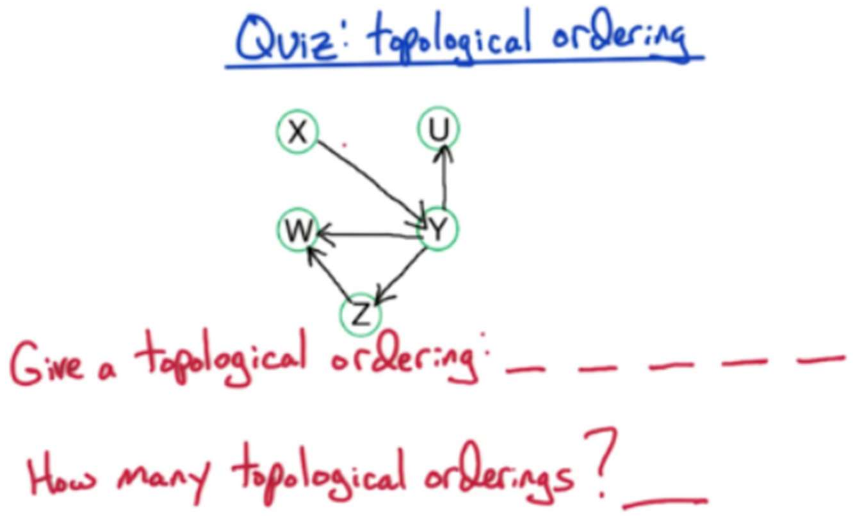
Now, how long does it take us to sort by decreasing postorder number? You might think this is O(n log n) time, because we have to do sorting. But, what is the range of these postorder numbers? When the clock starts at 1, all the postorder numbers are at least 1. How large can the postorder number be? - the maximum imposed order number is going to be 2n so all the postorder numbers range between 1 and 2n. So, what can you do?

You can make an array of size 2n. We can go through the vertices, take their postorder number, and insert them into the appropriate place in the array based on their postorder number. We go through this array from highest to smallest and we just output the vertices as we see them. This gives us the vertices in decreasing order of their postorder number.

How long does this take to construct? - just takes linear time. It takes O(n) time to sort the vertices by decreasing postorder numbers and takes linear O(n+m) time to run DFS. So the total algorithm takes O(n+m) time.

***

(1st Slide 11) Topological Ordering Quiz

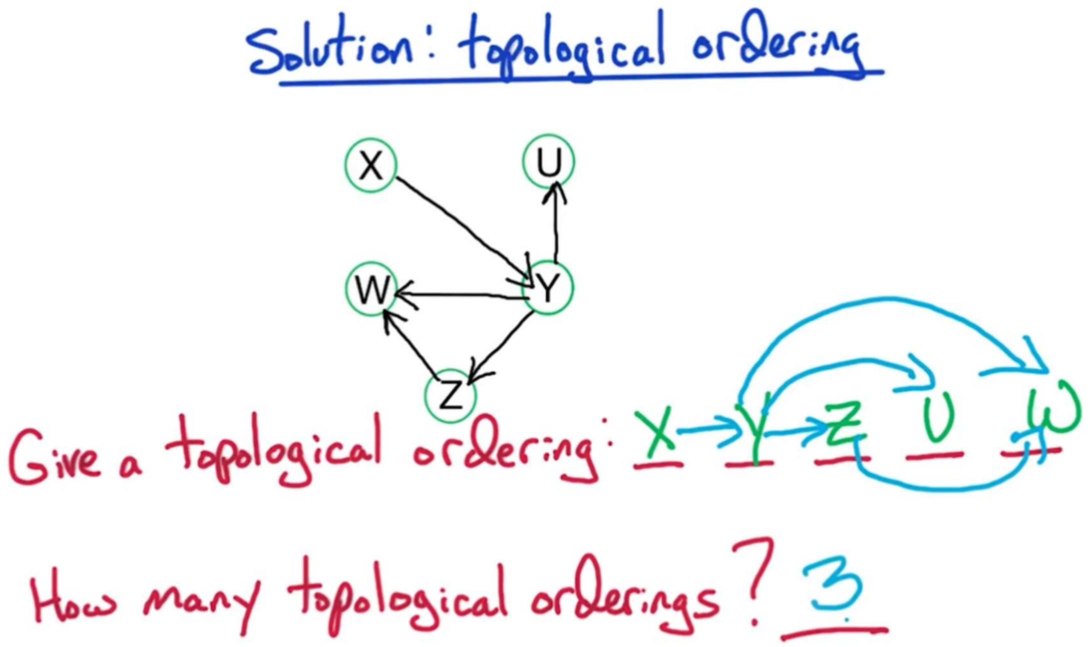

Quiz: topological ordering
X
U
W
Y
Z
Give a topological ordering: _ _ _ _ _
How many topological orderings? __

Here's an example of a graph on five vertices. Why don't you go ahead and give a topological ordering for this graph. There are five vertices, so let's write down the vertices in order so that all edges go from left to right.

This graph happens to have multiple topological orderings. How many topological orderings does it in fact have? To make sure you understand topological orderings, why don't you go ahead and specify the number of topological orderings which are valid for this graph.

***

(2nd Slide 11) Topological Ordering



Solution: topological ordering

Give a topological ordering: X → Y → Z U W

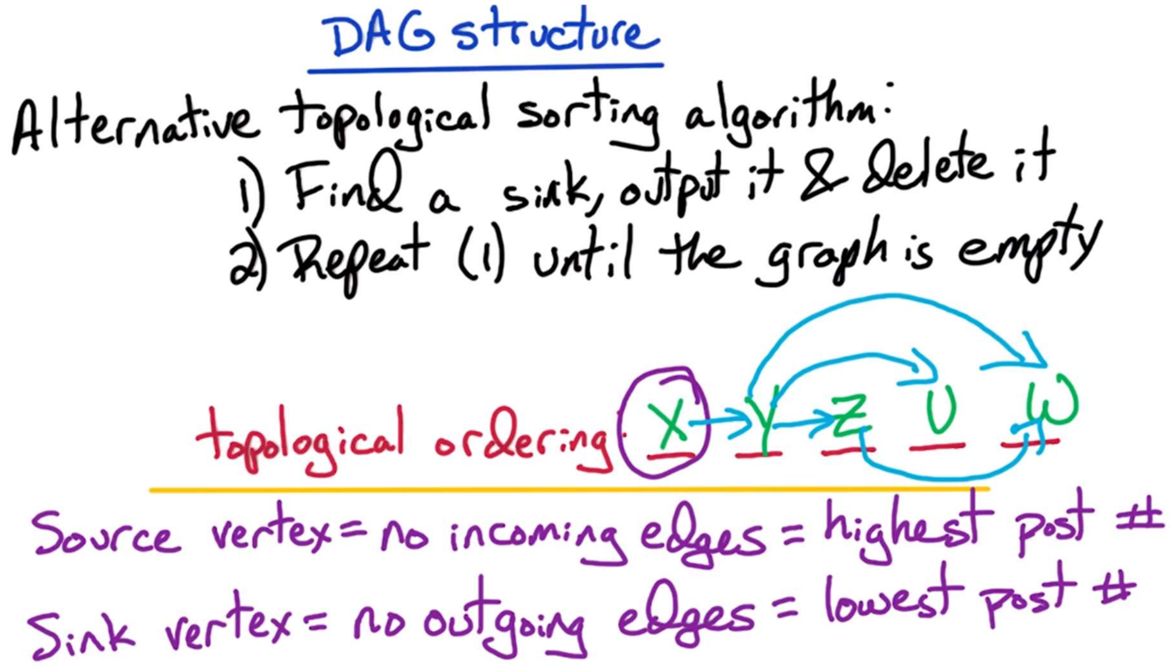
How many topological orderings? 3

Here is one valid topological ordering, X, then Y, then Z, then U, then W. Notice that all the edges go left to right.

How many valid topological orderings are there? Well, I can move U to any of these last three positions. So there are three choices for the position of U. After I specify the position for U, then the position for Z and W are forced because Z has to come before W. So this graph has 3 topological orderings.

***

(Slide 12) DAG Structure



DAG structure

Alternative topological sorting algorithm:

1) Find a sink, output it & delete it
2) Repeat (1) until the graph is empty

topological ordering: X Y Z U W

Source vertex = no incoming edges = highest post #

Sink vertex = no outgoing edges = lowest post #

Let's take a look at some properties of a DAG that we can derive from the topological ordering.

There are two types of vertices we want to distinguish: source vertices and sink vertices. A source vertex has no incoming edges. So everything goes out of a source. For a sink vertex, it's the opposite - nothing comes out of sink. Things only come into a sink.

Now a DAG always has at least one source and at least one sink. There may be multiple sources. There may be multiple sinks. But we're always guaranteed there's at least one source and at least one sink.

How do we know that there's a source vertex in every DAG? - well, take any topological ordering and look at the first vertex, in this case, x. What do we know about x?

We know all edges in the topological ordering go from left to right, from earlier in the topological ordering to later. So, edges can only come out of the first vertex in the topological ordering. No edges can come in, because then they would come from right to left. So every DAG has a topological ordering, and the first vertex in every topological ordering must be a source vertex.

So, that guarantees that every DAG has at least one source vertex. And there might be multiple source vertices because there might be multiple topological orderings and they might have different vertices at the beginning.

Now which vertex is first in our topological ordering? It's the vertex with the highest postorder number. Therefore, the vertex with the highest postorder number is guaranteed to be a source vertex.

Similarly, the last vertex in our topological ordering must be a sink vertex, because edges can come into it but nothing can come out, otherwise it would be going right to left again. Therefore, the vertex with the lowest postorder number is guaranteed to be a sink vertex. And once again, there might be multiple sink vertices. There might be multiple topological orderings with different vertices at the end.

In this example, u and w are both sink vertices and only x is a source vertex.

Now, let's look at an alternative topological sorting algorithm. Now this algorithm is not going to be very useful for DAGs, but it's going to be very useful when we look at general directed graphs.

We know that, in a topological ordering, the last vertex in the ordering is a sink vertex. So what can we do? We can find a sink vertex in some way. We can put it at the end of our list and then we can repeat on the remainder of the graph.

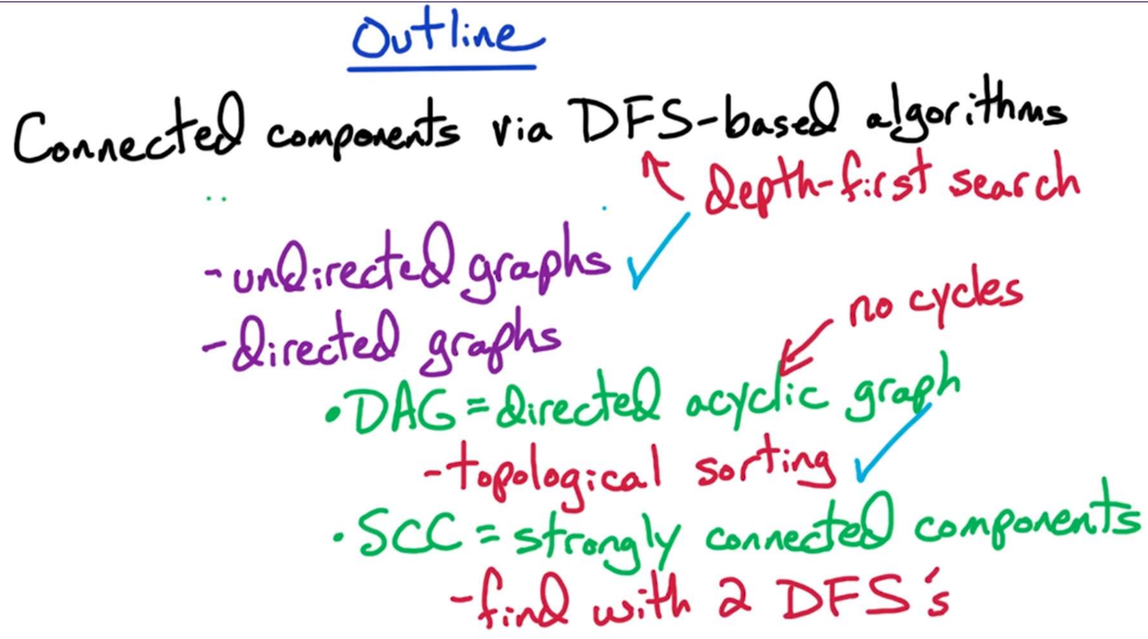
So we're going to find a sink vertex, output it and then we're going to delete it from the graph and then we're going to repeat. Now we repeat this first step, find a sink, in this case maybe it's u, and then we find z, and we find y and we find x. Notice when x is the only vertex remaining, it's the sink vertex in that graph of size one. Finally, we're left with the empty graph and then we stop.

What we've done is we outputed the vertices from the end to the beginning. And this gives us a valid topological sorting. Now how we actually find a sink vertex is another question. But this algorithm, this basic approach is valid. And we're going to use this basic approach when we consider general directed graphs.

And that's what we're going to turn our attention to now, general directed graphs, and we're going to look at the general question of what does this kind of activity mean in general directed graphs?

***

(Slide 13) Outline Review



Outline

Connected components via DFS-based algorithms

- undirected graphs ✓
- directed graphs
  - DAG = directed acyclic graph
    - topological sorting ✓
  - SCC = strongly connected components
    - find with 2 DFS's

Depth-first search
no cycles

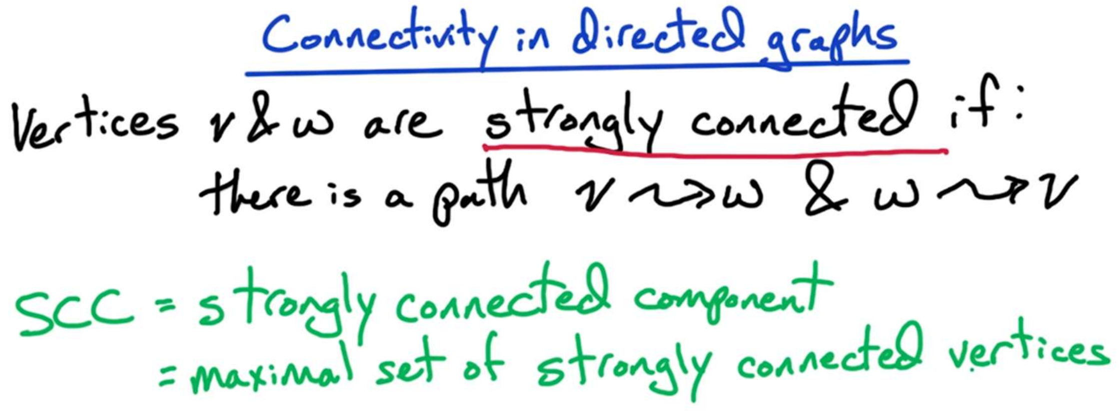
We've seen now how to find connected components in undirected graphs and how to topologically sort a DAG. Both of these algorithms involved one run of a vanilla version of DFS. Now let's look at general directed graphs.

First, we have to talk about what is the right analog of connected components for directed graphs. It turns out to be strongly connected components. And then we'll see an algorithm to find the strongly connected components of a general directed graph. Now the amazing aspect is that we're going to find these strongly connected components with just two runs of the vanilla version of DFS.

Let's dive into this algorithm and let's start by defining strongly connected components.

***

(Slide 14) Connectivity in Directed Graphs



Connectivity in directed graphs

Vertices v & w are strongly connected if:
there is a path $v \rightsquigarrow w$ & $w \rightsquigarrow v$

SCC = strongly connected component
= maximal set of strongly connected vertices

We're looking now at connectivity in directed graphs. So, the remainder of the lecture we're going to be talking about directed graphs.

First off, we have to talk about the correct notion of connectivity between a pair of vertices. So, let us take a pair of vertices, v and w and we will say they're strongly connected if there is a path from v to w and from w to v. I have this vertex v and this vertex w and there's a path from v to w, it may pass through many vertices on the way and there's a path from w to v that may pass through many vertices along the way and these paths may intersect. So these vertices v and w are strongly connected.

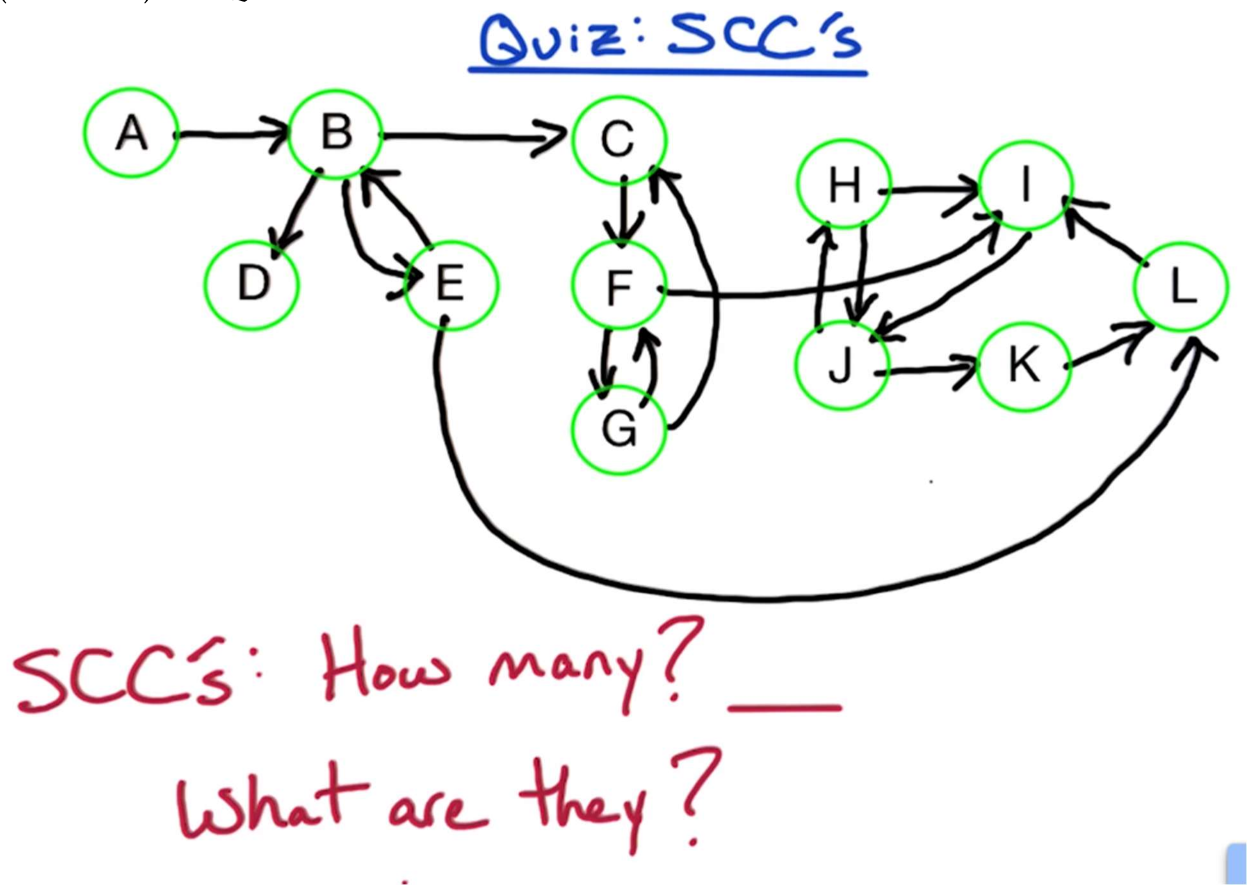
So instead of connected components in undirected graphs, the analog for directed graphs is going to be strongly connected components which we'll denote as SCC.

Now, in undirected graphs, the connected component is a maximal set of connected vertices. We keep adding in connected vertices as long as we can. For directed graphs, strongly connected components are the maximal set of strongly connected vertices. So we keep adding in strongly connected vertices as long as we can.

Let's take a look at a specific example and mark the strongly connected components in the example to make sure everybody understands it.

***

(1st Slide 15) SCC Quiz



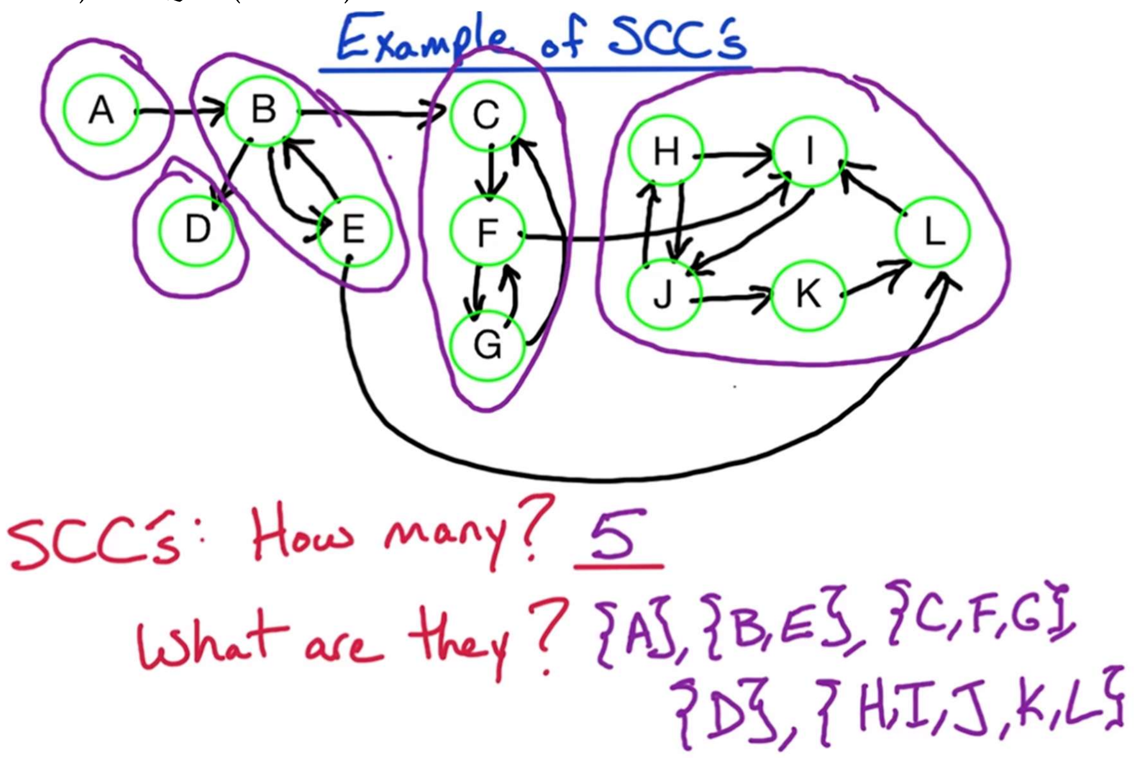
Here's a directed graph on 12 vertices. Let's go ahead and mark the strongly connected components in this graph, to make sure everybody understands the definition of SCCs. And then we can dive into some interesting properties of strongly connected points.

First off, how many strongly connected components does this graph have? And now can you mark what are the strongly connected components in this graph?

GR1: Strongly Connected Components: SCC Quiz
Question 1: How many Strongly Connected Components (SCC's) does the graph have?
Question 2: What are the SCCs? Enter them as sets of nodes in alphabetical order, for example {A,C}, {B,D,E}, etc.

***

(2nd Slide 15) SCC Quiz (Answer)



In this graph, there are five strongly connected components. Let's go ahead and mark them.

- An easy one to notice is A by itself. Notice that from vertex A, I can reach many other vertices but no other vertices can reach A, so A is not strongly connected to any other vertices, therefore A is a strongly connected component by itself.
- These five vertices ({H,I,J,K,L}) form a strongly connected component. Notice that all five vertices are strongly connected with each other, in particular, from vertex J, I can reach all of the other vertices and all the other vertices can reach vertex A via I.
- Similarly, these three vertices ({C,F,G}) form a strongly connected component.
- These two vertices B and E form a strongly connected component
- and D is by itself because other vertices can reach D but D can't reach anybody else.

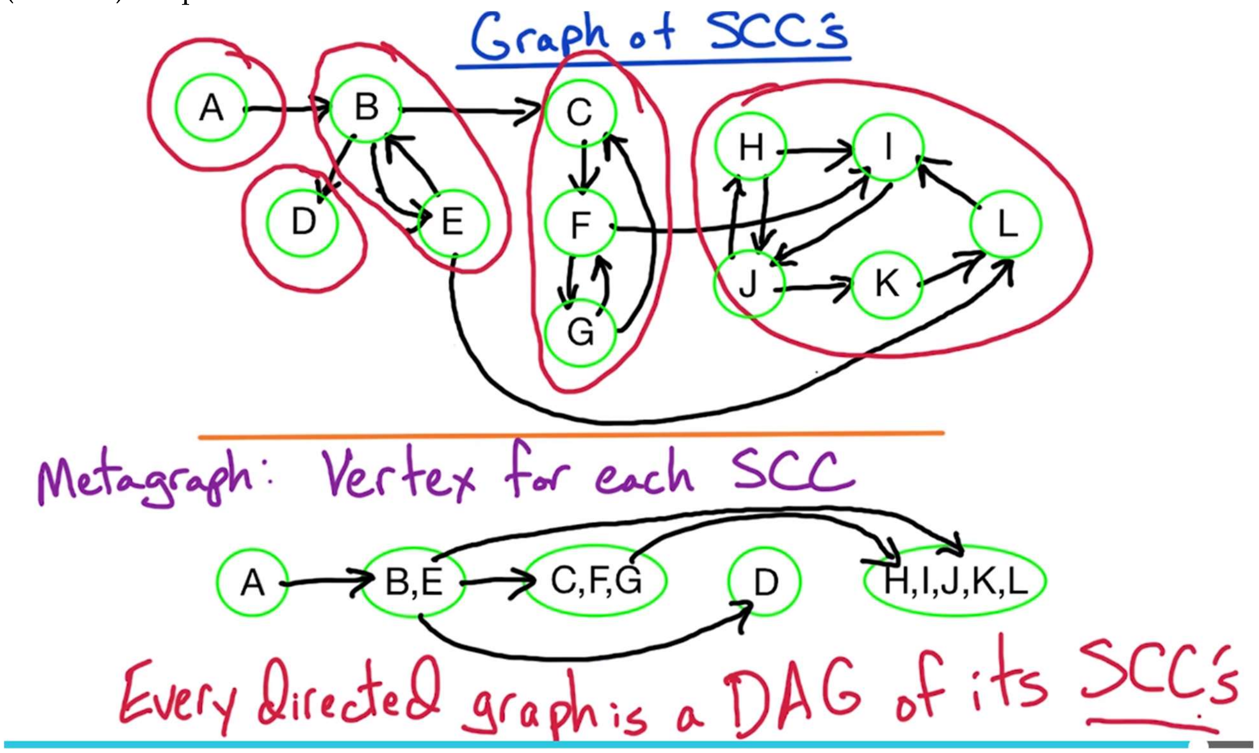
So, I have these five strongly connected components, {A}, {B,E}, {C,F,G}, {D}, and {H,I,J,K,L}. I want to point out some interesting properties of strongly connected components so I want to look at this graph on five vertices.

I'm going to make a metavertex for each strongly connected component - I have a metavertex for A by itself; B and E together; C, F, G; D; and H through L. So I'm going to think of compressing each of these purple blobs into a vertex - a metavertex - and then there's going to be an edge from this metavertex to this metavertex because there is an edge from some vertex in C, F, G to some vertex in this strongly connected component - namely from F to I; and, similarly, there will be an edge between these meta vertices and these meta vertices and so on.

So when we look at this graph on the meta vertices of strongly connected components we're going to see some interesting properties.

***

(Slide 16) Graph of SCC


Graph of SCC's
A
B
C
D
E
F
G
H
I
J
K
L
Metagraph: Vertex for each SCC
A
B,E
C,F,G
D
H,I,J,K,L
Every Directed graph is a DAG of its SCC's

I want to look at the metagraph on strongly connected components. So we're going to have a vertex for each strongly connected component.

So let's go ahead and mark the five strongly connected components, the five vertices in this metagraph.

Here are the five strongly connected components, {A}, {B,E},{D},{C,F,G},{H,I,J,K,L}. So I've marked those five vertices down here, A, BE, CFG, D, and HIJKL. Now, what are the edges in this metagraph?

Well, some vertex in this component, BE, has an edge to some vertex in this component, CFG, namely B to C. So, we'll put an edge from this component to this component. Similarly, there's an edge from this component, BE, to D, from A to B, F to I, and E to L.

What do you notice about this graph? … this metagraph on strongly connected components? - this metagraph on strongly connected components may, in fact, be a multigraph. For example, I may have an edge from G to J and then, I'll have another edge from this component to this component. So, I have a pair of edges from this component, CFG, to HIJKL.

Now, the multiplicity doesn't matter. So, we can keep those multiple edges or we can drop them. It doesn't matter. So let's go ahead and drop those multiple edges for simplicity.

Either way, what is the key property of this metagraph on strongly connect components? What do you notice about this metagraph? Notice that there are no cycles. So, this metagraph is a DAG.

And that's always the case. Every metagraph on strongly connected components for every directed graph is a DAG.

Why is that? Why are there no cycles in this metagraph? - well, suppose there's two strongly connected components which are involved in a cycle. Then, that means there's a path from somebody in this component to somebody in that component, and from somebody in this component, there's a path back to that component.

Now, we know everybody in this component S is connected to each other because it's strongly connected. Everybody over here in S′ is strongly connected to each other. So, therefore if there are these paths from S to S′ and S′ to S, that means everybody in S can reach everybody in S′ and everybody in S′ can reach everybody in S. Therefore, $S \cup S'$ is a strongly connected component.

Now these components are defined to the maximal sets of strongly connected vertices. Therefore, we have a contradiction because a strongly connected component should be $S \cup S'$ - it shouldn't be S separated from S′.

So, if there is a cycle in this metagraph, then we can merge strongly connected components together to get a larger strongly connected component and therefore we get a contradiction.

Therefore, there can't be any cycles in this metagraph and thus it must be a DAG. We now have this amazing property: Every directed graph is a DAG of it's strongly connected components.

So, we can break up a directed graph into strongly connected components and then we can order these strongly connected components into topological ordering because it's a DAG.

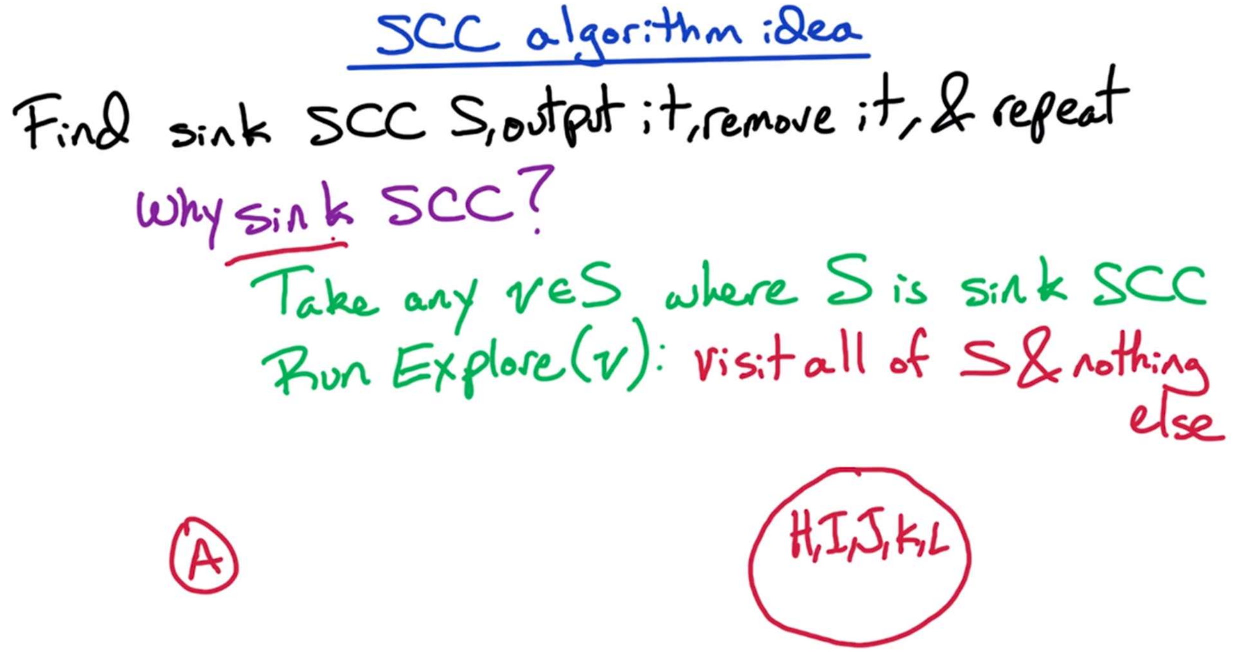
So, you can take an arbitrary directed graph, which may be very complicated, and you can find this beautiful structure hidden in it. You can break it up into strongly connected components and then you can topologically sort these strongly connected components so that all edges go left to right. That's what we're going to do now.

We're going to find an algorithm which is going to find the strongly connected components and it's going to find the strongly connected components in some order and the order is going to be topological ordering of these strongly connected components. So, we're going to topologically sort these strongly connected components as we find them.

Now, the amazing thing is that we're going to find these strongly connected components and this topological ordering with just two runs of DFS.   Now, let's dive into the algorithm to see how we're going to find these strongly connected components.

***

(Slide 17) SCC Algorithm Idea



SCC algorithm idea

Find sink SCC S, output it, remove it, & repeat

Why sink SCC?

Take any $v \in S$ where S is sink SCC

Run Explore(v): visit all of S & nothing else

Now, let's look at the main idea for our strongly connected component algorithm. We're going to find these strongly connected components in topological ordering.

So let's go back and look at our topological ordering algorithm for DAGs. Let's look at the topological ordering of a DAG where vertex v happens to be first and vertex w happens to be at the end.

Now, what do we know about vertex w? We know it has to be a sink vertex. It might have some edges in but it can't have any edges out because those edges would go backwards in the ordering. Similarly, vertex v must be a source vertex. It can have edges out but it can't have any edges in.

We have this alternative approach for topologically sorting a DAG. We could find a sink vertex, output it, rip it out, and repeat. Find a new sink in the resulting graph and repeat. Or, we could find a source vertex, put it at the beginning, rip it out of the graph, and repeat. Find a new source vertex in the resulting graph and so on. We can either work left to right, or right to left. Finding sink vertices and moving on, or finding source vertices and moving on.

We're going to do a similar idea here but instead of finding a single vertex, we're going to find a sink strongly connected component. This is a component which is a sink vertex in a metagraph on strongly connected components. We're going to find a sink strongly connected component then we're going to output it. That's going to be at the end of our ordering. We're going to remove it from the graph, so we're going to remove all vertices from this strongly connected component from the graph and then we're going to repeat. We're going to find a sink component in this resulting graph, output it, remove it, and repeat until the graph is empty.

Now, why do we do “sink strongly connected components”? Why not do “source strongly connected components”? - for the topological ordering of the DAG, it didn't matter whether we started with sinks and worked that way backwards, or if we started with source and work forward. But, for SCC, it matters.

Sinks are easier to work with. Why are sinks easier to deal with? Well, take any vertex which lies in a sink SCC. So S is a sink and strongly connected component, V is the vertex lying in that component. Now run Explore from V. This is the basic procedure in the DFS algorithm. Suppose this is the first vertex that you run Explore from. Which vertices do you visit when you explore from V?

Well, take our earlier example where we had this sink strongly connected component which was H through L … Say we run Explore from any of these vertices. What's going to happen when we run Explore? We're going to visit all the vertices in this sink component but we're not going to visit any other components because we can't reach any other components from this component because it's a sink strongly connected component. So we visit all of this component and we visit nothing else. We don't see any other vertices, we just see this component itself.

So if we can find a vertex which is guaranteed to be in a sink strongly connected component, then we can run Explore from that vertex, and we're going to visit and we're going to find exactly that sink component. That's the key property about sink components. We just need to find a vertex which lies in that component, then when we Explore from it, we're going to find the component itself and we're going to see nothing else.

Therefore, we can mark all the vertices that we visited from this Explore as lying in this sink component, then we can rip out those visited vertices and we can repeat the algorithm: find a vertex in a sink of the resulting graph, run Explore from it, mark those vertices as being in that component, and repeat.

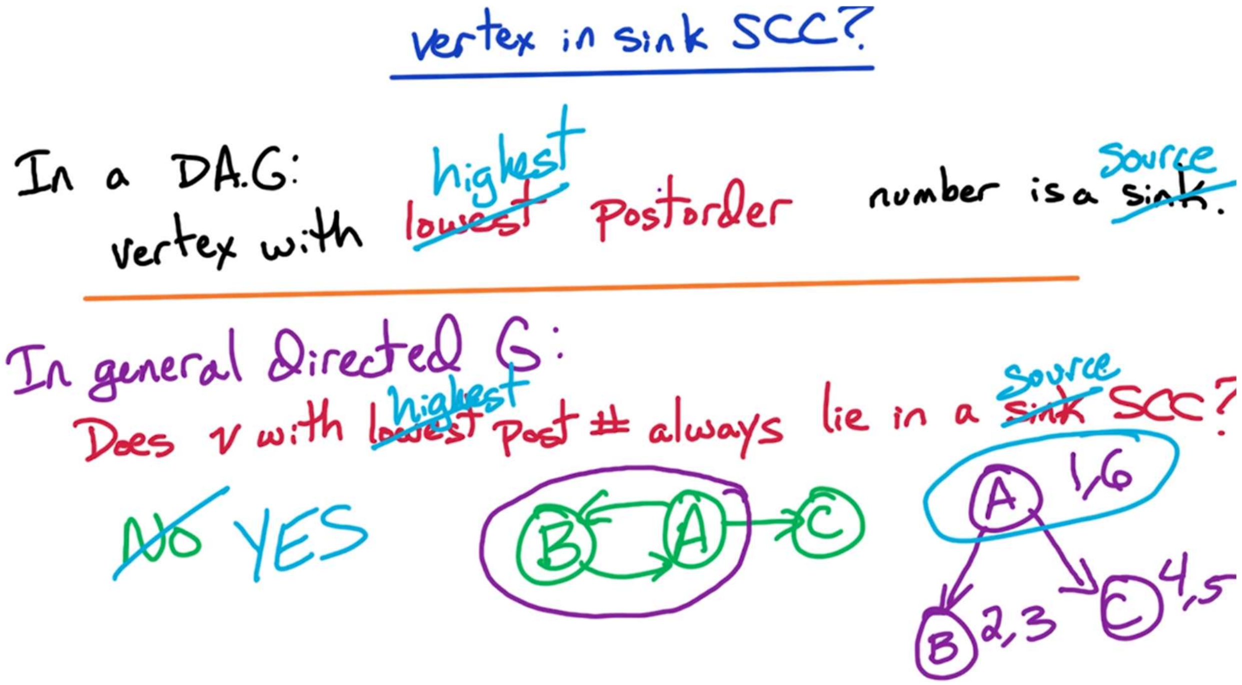
Now, what if we could find a vertex lying in the source component? For example, what if we can find vertex A? And we know that A is guaranteed to be in a source component, while in our earlier example, A happened to be a source component by itself.

But, suppose there are other vertices in this component and we want to figure out who are the other vertices in this component? When we run Explore from vertex A, what happens? All we know is that from A we can reach many vertices. It's a source. So in fact, we can reach the whole graph from A. The whole graph is going to be visited. So we have no way of marking which vertices happen to be in this SCC and which vertices lie in other SCCs. But if we run Explore from a vertex which lies in a sink SCC, we only visit that component and nothing else. That's the key property about sink components.

Now, how can we find a vertex V which is guaranteed to lie in a sink component? That's our key task. Once we can find a vertex which is guaranteed to lie in a sink component, then we can run Explore from that vertex - we'll find that sink component, rip it out, and repeat the algorithm. We'll find a sink component in the resulting graph, rip it out, repeat and so on.

***

(Slide 18) Vertex in sink SCC


vertex in sink SCC?
In a DAG: vertex with ~~lowest~~ highest postorder number is a ~~sink~~ source.
In general Directed G:
Does v with ~~lowest~~ highest post # always lie in a ~~sink~~ source SCC?
~~NO~~ YES
B
A
C
A 1,6
B 2,3
C 4,5

So, in a DAG, the vertex with the lowest postorder number is guaranteed to be a sink vertex. Now, let's look at a general directed graph – so, there may be cycles. Let's run DFS on this general directed graph.

Is there some property of postordering numbers that we can use to find a vertex which is guaranteed to lie in a sink SCC? While drawing inspiration from DAGs, we may say, the vertex with the lowest postorder number. Maybe that happens to be guaranteed to lie in a sink SCC?

Now we might hope for the following property in general directed graphs. We might run DFS on this general directed graph and we might hope that the vertex with the lowest postorder number always is guaranteed to lie in the sink SCC. If that was the case, then in order to find a vertex in a sink SCC, we just run DFS on this general directed graph, take the vertex with the lowest postorder number, and that has the guaranteed property that we're looking for.

This guess follows from our inspiration from the topological sorting algorithm for DAG. Does this guess hold? Is the property true? Unfortunately, it's not true.

Here's an easy example where it's not true. I have a graph on three vertices, A, B, and C. Now, A and B are strongly connected with each other and C is by itself. Now, let's say I run DFS starting from vertex A. So, A's a root and then from A, I visit vertex B, then from B, I pop back to A and then I visit vertex C, and then I pop back to A. Now, what are the preorder and postorder numbers? I start with A. Then I go to B. Then I finished B, pop back to A, go to C. Then I finish C and then I pop back to A. So, which vertex has the lowest postorder number? - it's vertex B. But, B lies in this strongly connected component. Is this a sink SCC? No, in fact, it's a source SCC. So, in this example, the vertex with the lowest postorder number lies in a source SCC. Complete opposite of what we're hoping for.

But what if, instead of finding a sink SCC, I want to find a source SCC. I don't know how to use that, but let's just say I wanted to find a source SCC.

Let's go back to look at our DAG algorithm. So, I want to find a source vertex. Remember, in our topological sorting algorithm, the vertex at the beginning of the topological ordering is the vertex with the highest postorder number. So, the vertex with the highest postorder number is guaranteed to be a source vertex in a DAG. So, in the general directed graph, does the vertex with highest postorder number always lie in a source SCC? Well, in this example, that's actually true. The vertex with highest postorder number is A, which lies in a source SCC.

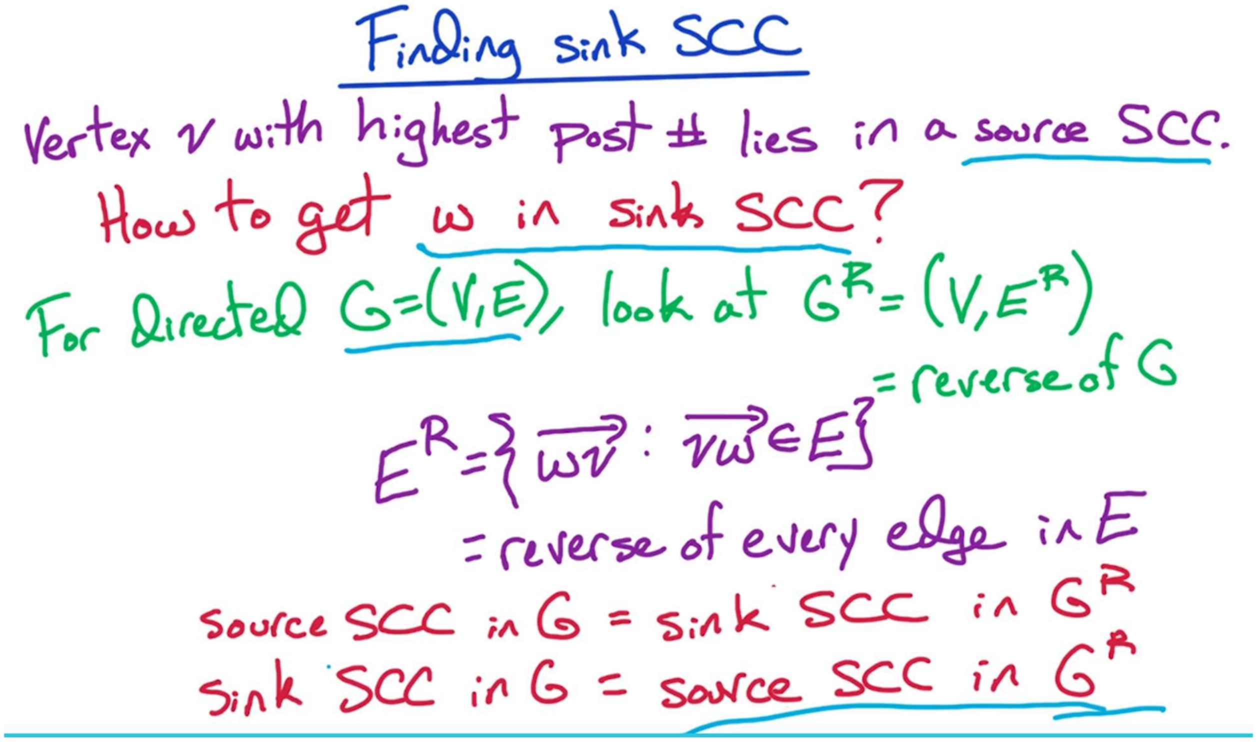
It turns out that this property is true and we're going to prove that it's true.

So, in every directed graph, when we run DFS on that directed graph, it doesn't matter on which vertex we start at and which is the ordering on the neighbors: For every directed graph, for every DFS run on that directed graph, the vertex with the highest postorder number is guaranteed to lie in a source SCC. Now, let's use this property to get an SCC algorithm and then we'll go back and we'll prove that this holds.

First off, how can we use it? We notice before that we need a vertex that lies in a sink SCC. If we have a vertex which lies in the source SCC, that's not useful for us. But all we can guarantee is to find a vertex which lies in a source SCC. We don't know how to find a vertex in a sink SCC.

***

(Slide 19) Finding sink SCC



Finding sink SCC

Vertex v with highest post # lies in a source SCC.

How to get w in sink SCC?

For Directed $G=(V,E)$, look at $G^R=(V,E^R)$ = reverse of G

$E^R=\{\overrightarrow{wv} : \overrightarrow{vw} \in E\}$

= reverse of every edge in E

source SCC in G = sink SCC in G^R

sink SCC in G = source SCC in G^R

This is the property that we just claimed is true. When we run DFS on any directed graph, the vertex with the highest postorder number lies in a source SCC. We'll go back and prove this property momentarily, but let's first use it to get an SCC algorithm.

We need a vertex which lies in a sink SCC. How can we find such a vertex w which is guaranteed to lie in a sink SCC? Well, we know how to find a vertex which lies in a source SCC. I claim we can use this as a subroutine in order to find a vertex which is guaranteed to lie in a sink SCC. And then we're all done; then, we have our SCC algorithm. Why don't you go ahead and think about this? How can we find a vertex which is guaranteed to lie in a sink SCC using this property that the highest postorder number lies in a source SCC?

In some sense, we just want to redefine the terms. We want to redefine a source to be a sink and a sink to be a source. What do we mean by that? Think about our topologically ordering of a DAG. The edges go left to right, and the beginning of the ordering is a source and at the end is a sink. What if we flipped all the edges to go backwards? Then this vertex which used to be a sink would now be a source, and this vertex which used to be a source will now be a sink because all the edges go right to left. So the ordering will be opposite.

That's what we want to do now for our general directed graph. We want to flip the graph. We want to look at the opposite graph or the reverse graph. And then the source component will become a sink component, and the sink component will become a source component.

Now, for a general directed graph G which has G= {V,E}, we're going to look at G^R - this is going to be the reverse of the graph G. The vertex set is going to stay the same. The edge set is going to change from E to E^R. We're just going to reverse all the edges. These are directed edges so we're just going to look at the reverse edges.

What exactly is E^R? For every edge in E, if we have an edge from v -> w, we're going to add an edge from w -> v into E^R, we're going to add in the edge from w -> v. E^R is simply the reverse of every edge in E. Notice if we flip all the edges, we look from G to G^R then all of the sources and sinks get flipped.

Now, how does the strongly connected components of G compare to the strongly connected components of G^R? Notice that if a pair of vertices are strongly connected in G, then they're also strongly connected in G^R. There's a path from v -> w in G, and a path from w -> v; and then, in G^R, there also is a path from v -> w and w -> v. The set of strongly connected components are the same in the two graphs.

Now, what about the metagraphs on the strongly connected components? Look at these two Graphs. It might be different. In particular, the metagraph of strongly connected components in G is a DAG. And now, what does it look like in G^R? Well, we flipped that DAG so all the strongly connected components which were at the beginning of the topological ordering are now at the end. The edges are not going right to left instead of left to right.

So, if we take a component which was a source SCC in G, then it becomes a sink SCC in G^R because the ordering goes backwards. Similarly, if we take a component which was at the end of the ordering for G then it's going to be at the beginning of the ordering for G^R.

Now, how do we address our original problem? We want to find a vertex which is guaranteed to lie in a sink SCC of G. All we can do is find vertices which are guaranteed to lie in a source SCC. Well, the sink SCC of G corresponds to a source SCC of G^R. So we take this input graph G, we construct its reverse graph, and then we run DFS on this reverse graph, and then we take the vertex with highest postorder number as guaranteed to be in a source SCC of this graph G^R.

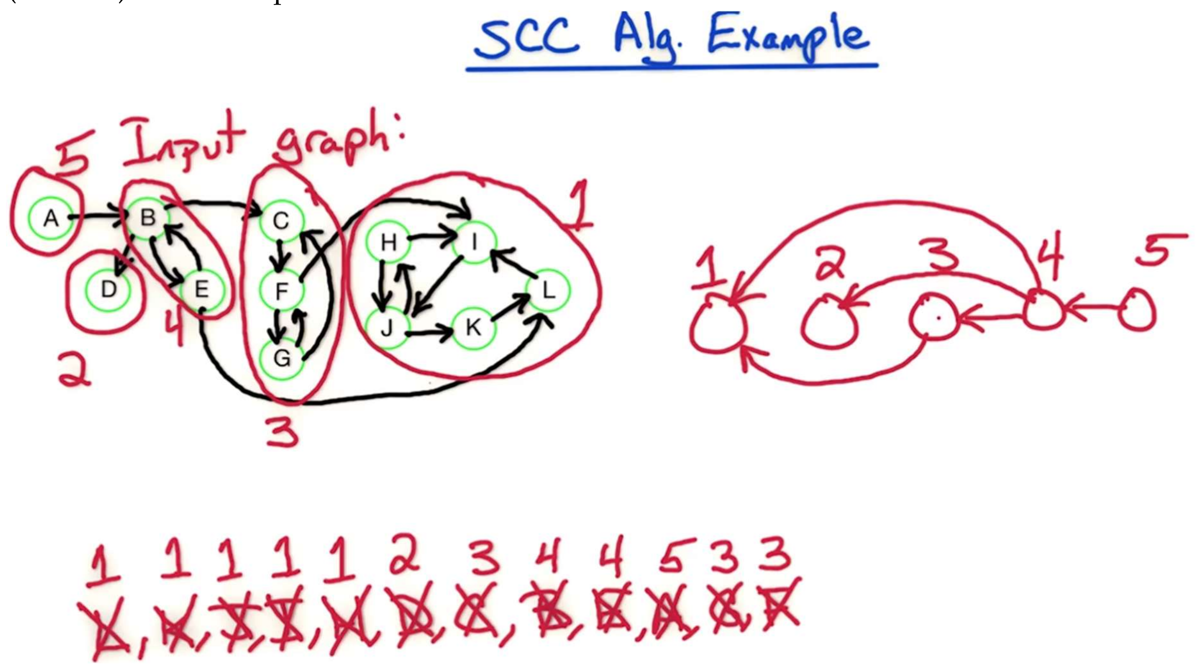
This vertex, which is guaranteed to be in a source SCC of G^R (or that SCC), is a sink in G. So this vertex with highest postorder number for the DFS run on G^R is guaranteed to be in a sink SCC of G, the original graph.

That's it. That's our algorithm for finding a vertex which is guaranteed to lie in a sink SCC of a graph G. You just reverse the graph, run DFS, take the highest postorder number of vertex, and it is guaranteed to be in a source of G^R, and therefore a sink of G.

And now we have our algorithm for finding strongly connected components and it's going to find the strongly connected components in topological ordering. We're going to find a sink SCC and move on, so we're going to find this ordering from right to left.

***

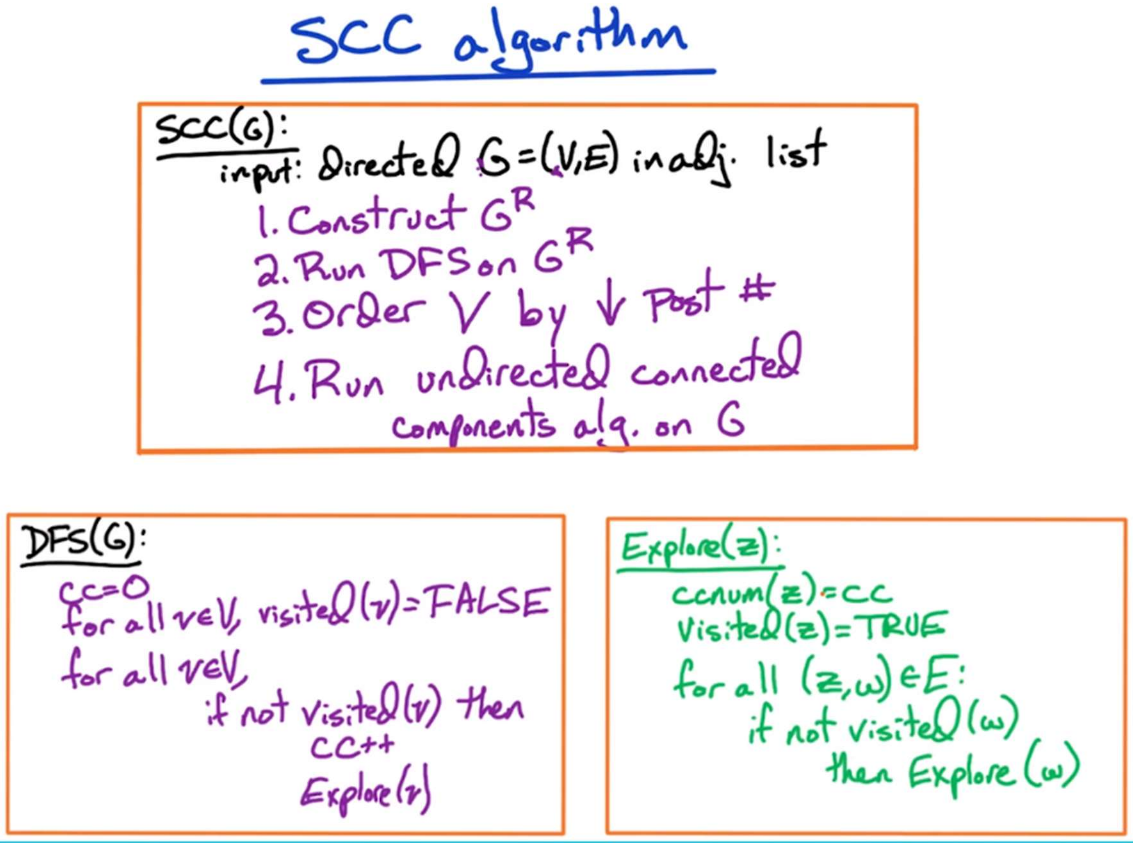
(Slide 20) SCC Example


SCC Alg. Example
Input graph:
A
B
C
D
E
F
G
H
I
J
K
L
1
2
3
4
5
1 1 1 1 1 2 3 4 4 5 3 3
~~L~~, ~~K~~, ~~J~~, ~~I~~, ~~H~~, ~~D~~, ~~C~~, ~~B~~, ~~E~~, ~~A~~, ~~G~~, ~~F~~

There is a footnote on slide: Typo: The preorder number of D and the postorder number of C are both 12. The preorder number of D should be 13 and all preorder/postorder numbers from 13 onwards should be incremented by 1. The resulting order on the postorder numbers does not change.

***

(Slide 21) SCC Algorithm



SCC algorithm

```
SCC(G):
input: Directed G=(V,E) in adj. list
    1. Construct $G^R$
    2. Run DFS on $G^R$
    3. Order V by ↓ Post #
    4. Run undirected connected
           components alg. on G
```

```
DFS(G):
    cc=0
    for all v∈V, visited(v)=FALSE
    for all v∈V,
        if not visited(v) then
            cc++
            Explore(v)
```

```
Explore(z):
    ccnum(z)=cc
    visited(z)=TRUE
    for all (z,w)∈E:
        if not visited(w)
            then Explore(w)
```

Finally, let's detail our strongly connected component algorithm. The input to our algorithm is a directed graph G in adjacency list representation.

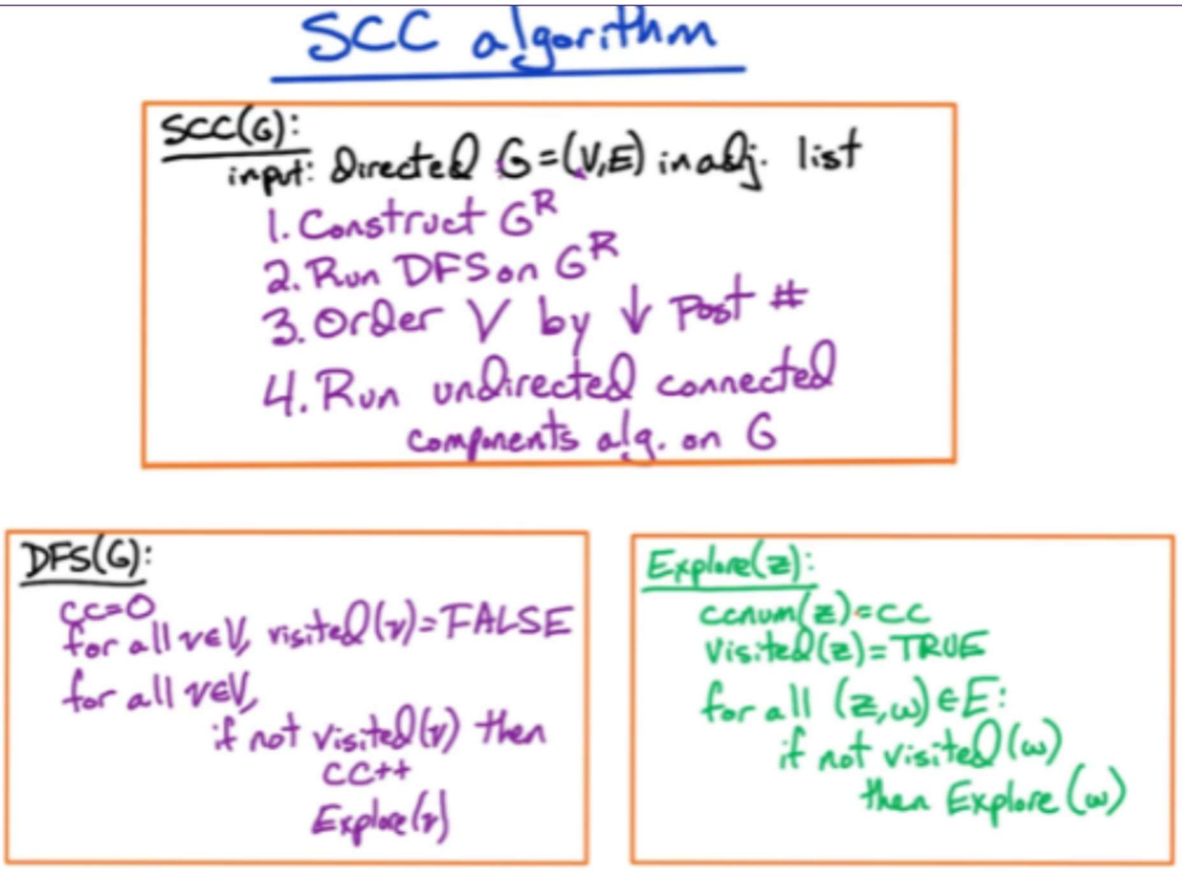
The first step of our algorithm is to construct the reverse of the graph G. Then we run DFS on the reverse graph. What do we know? We know that the vertex with highest postorder number from this DFS run is guaranteed to lie in a source SCC of GR, and therefore is in a sink SCC of G. So, now what we want to do is we want to run Explore from this vertex with highest postorder number from this DFS run.

So now, what we're going to do is we're going to order the vertices. These are the vertices of the original graph G. We're going to order these vertices by decreasing postorder number. This is like we're ordering them by topological ordering, as for our DAG algorithm, and these are the postorder numbers from this DFS run. So, we run DFS on the reverse graph and then we order the vertices in the graph by decreasing postorder number from this DFS run. Now, finally, we run DFS on the original graph, where the vertices are ordered by this decreasing postorder number from this DFS run.

Now, what is the version of DFS we use for this last one? - we're actually going to use the undirected connected components algorithm. If you recall, that runs DFS and it marks the components that we see along the way with a connected component number ccnum. We're going to run that identical pseudocode. Even though this is a directed graph and that was designed for undirected graphs - we run the identical pseudocode and the components, the strongly connected components, in this case, are going to be numbered and they're going to be numbered in topological ordering. So, the first component is going to be at the end of our topological ordering and so on.

Now, just to remind you, this is our pseudocode for our undirected connected component algorithm. So, this was just DFS where we kept track of the connected component number and we marked the vertices with their connected component number as we visited them. So, for step four of this directed SCC algorithm, we're going to run this identical pseudocode and this ccnum - it’s going to give us the ordering, the topological ordering on the strongly connected components.

GR1: Strongly Connect Components: SCC Algorithm



SCC algorithm

```
SCC(G):
  input: Directed G=(V,E) in adj. list
    1. Construct G^R
    2. Run DFS on G^R
    3. Order V by ↓ post #
    4. Run undirected connected
         components alg. on G
```

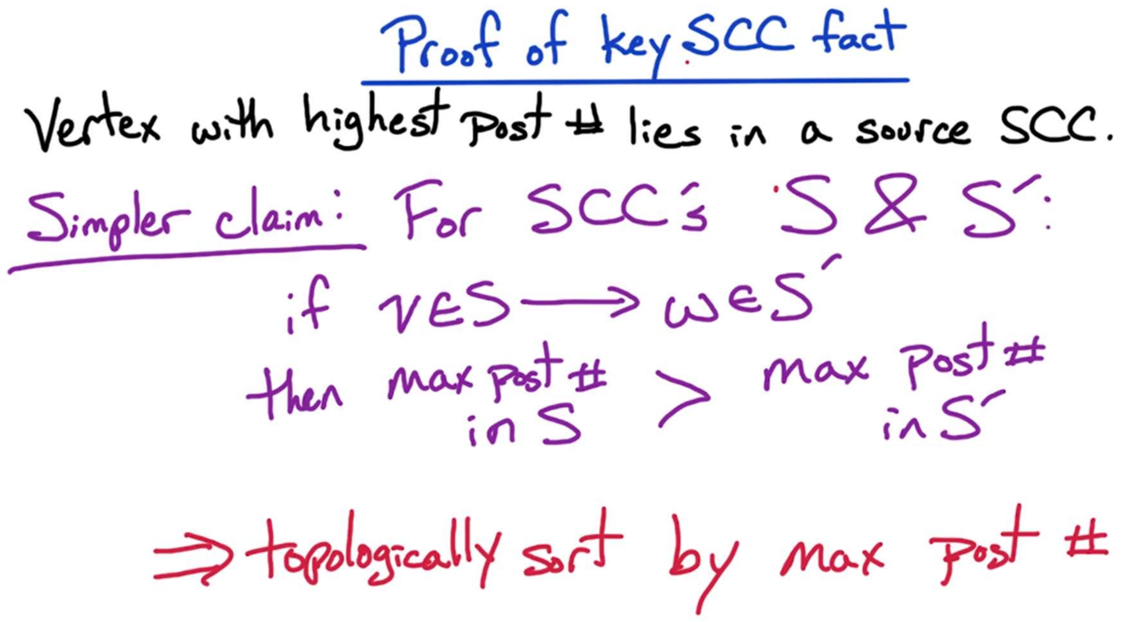
```
DFS(G):
  cc=0
  for all v∈V, visited(v)=FALSE
  for all v∈V,
        if not visited(v) then
              cc++
              Explore(v)
```

```
Explore(z):
  ccnum(z)=cc
  visited(z)=TRUE
  for all (z,w)∈E:
        if not visited(w)
              then Explore(w)
```

What is the running time of the SCC Algorithm?

***

(Slide 22) Proof of Key SCC Fact



Proof of key SCC fact

Vertex with highest post # lies in a source SCC.

Simpler claim: For SCC's S & S′:

if $v \in S \longrightarrow w \in S'$

then max post # in S $>$ max post # in S′

$\Rightarrow$ topologically sort by max post #

Now, we took the following fact for granted in the design of our algorithm: The vertex with the highest postorder number lies in a source SCC. Now, let's prove that fact so that we complete the proof of correctness of our algorithm.

Let's look at the following simpler claim. Let's take a pair of strongly connected components S and S′. Now, if some vertex in S has an edge to some vertex in S′, then what can we say about the postorder numbers for S versus S′? What we show is that the maximum postorder number in S is greater, strictly greater than the maximum postorder number in S′. Now, what does this simpler claim give us?

Well, this simpler claim gives us a way to topologically sort the strongly connected components. How do we topologically sort them? We sort them by the maximum postorder number in that component. So, for each strongly connected component, we're going to look at the max postorder number of the vertices lying in that component. So we can think of the postorder number for this component to be the max over the vertices in that component of the postorder numbers.

Now, we sort these strongly connected components by their postorder numbers and we sort them in decreasing postorder number. And this claim, this simpler claim, tells us that all edges will go from larger postorder number to smaller postorder number. So, all edges will go left to right in the ordering of these strongly connected components.

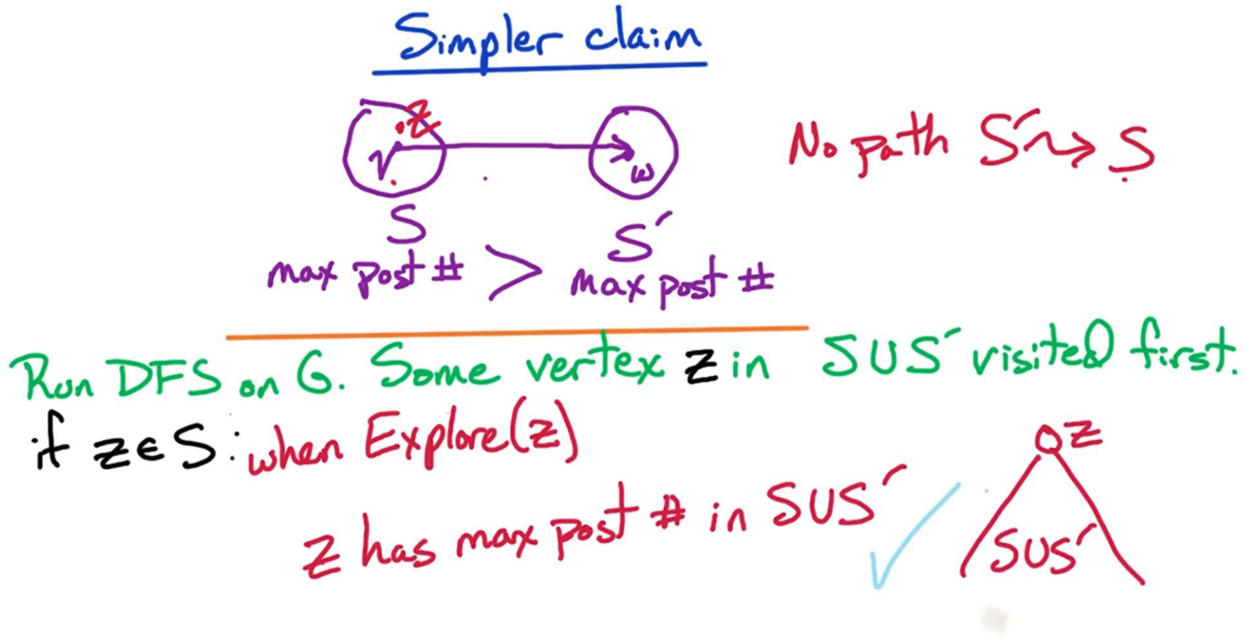
Now, what do we know about the vertex with the highest postorder number? Well, it's strongly connected component is going to have the maximum - the largest of these max postorder numbers. So, its component is going to be at the beginning of this topological ordering and it's guaranteed to be a source SCC since it's at the beginning of the topological ordering.

So, therefore, the vertex with the highest postorder number will be in the component which is at the beginning of the topological ordering and therefore it's a source SCC. So, if we prove this simpler claim, then this gives us a way of topologically sorting the components by the max postorder number. And this implies that the vertex with the highest postorder number lies in a source SCC.

So, if we prove this simpler claim, it implies the fact that we used in our SCC algorithm. So all we need to prove is this simpler claim and then we're done.

***

(Slide 23) Simpler Claim



Here's a simpler claim once again. We have two components - two strongly connected components - S and S′. And there's some vertex in S, let's call it v. And there's a vertex in S′, let's call it w. And there's an edge from v -> w. And the claim is that if we look at the maximum postorder number of all vertices lying in S compared to the maximum postorder number of all vertices lying in S′, then the maximum postorder number in S is bigger than the max postorder number in S′. So let's go ahead and prove that fact.

Let's make one simple observation about this graph before we proceed. Notice we have an edge from this strongly connected component S to the strongly connected component S′. We know there's no cycles in this graph on strongly connected components, so there can't be a path from S′ to S. Otherwise, these two strongly connected components would be a strongly connected component by itself. S ∪ S′ would be a strongly connected component and that would contradict the maximality of these SCCs.

We know that there is no path from S′ back to S because there's this edge that we assumed from v in S to w in S′. We're going to use this simple fact that there is no path from S′ to S in the proof of this simpler claim.

Run DFS on this graph and we're looking at the postorder numbers from this DFS run. Initially, all the vertices are unvisited. At some point some vertex in S ∪ S′ must be visited. Let's take the vertex in S ∪ S′ which is visited first and let's give it a name. Let's call it z. We're going to have two cases: Either Z lies in S′ or Z lies in S.

Let's look at the first case. Let's say that z lies in S′. Now, z visits first so we're going to run Explore on z. And who are we going to see? What vertices are we going to visit when we do Explore on z? Well, which vertices are reachable from z? z lies in S′ so from z we can see all of S′ but we can't see any of S. We see all of S′ and we see none of S. All of S′ is going to be visited and finished exploring, and none of S is going to be visited at all before we finish exploring from z.

z and all of S′ is going to be assigned postorder numbers before we even give a preorder number for any vertex in S. The punchline is that all of the postorder numbers in S′ - for every vertex in S′ - its postorder number is strictly smaller than the postorder number of every vertex in S … because we visited and finished exploring all the vertices in S′, including z, before we even visit S, before we even given a preorder number to any vertex in S. And, therefore, the maximum postorder number in an S′ is strictly smaller than the minimum or the maximum postorder number in S which proves that claim. This completes the proof of the claim in this case where z lies in S′.

Now let's look at this case with a vertex z lies in this component S. All these vertices in S and S′ are unvisited. And now we visit z and we start exploring from z. Who do we reach? What vertices can we reach from z? We can reach all of S because this is a strongly connected component, and we can reach all of S′ because we can go from v over to S′, and then we can reach around S′.

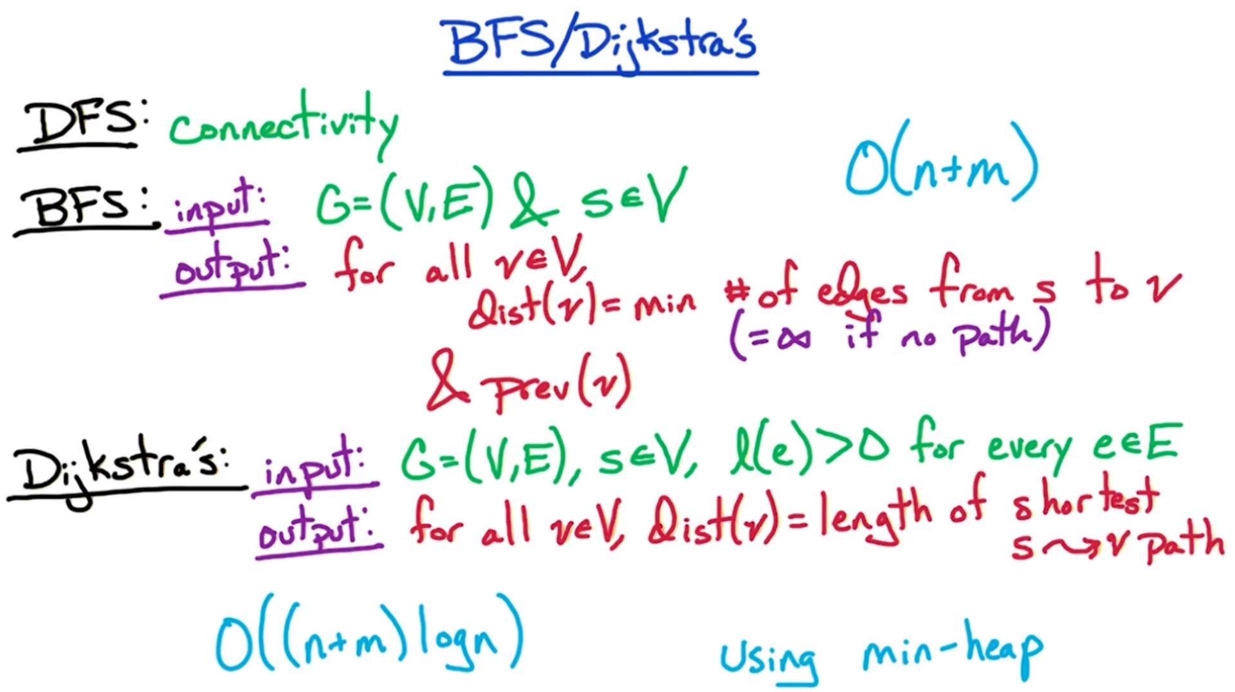
If we looked at the DFS tree, we're going to have this vertex z. And what lies in that sub-tree? The rest of S U S′ lies in its sub-tree, in the DFS tree. Therefore, what do we know about z? We know that z is going to be the last vertex that we finish in $S \cup S'$ because it's the root of this sub-tree. We know that z has the maximum postorder number of any vertex in $S \cup S'$ because it's the root of this sub-tree in the DFS tree. All the descendants have to be finished before we finished z. And, therefore, since z lies in S then we know that the vertex in $S \cup S'$ with maximum postorder number is z, which lies in S. So that proves the claim.

The max postorder number in S, which corresponds to z, is larger than anybody in S′. This completes the proof of this claim in this case where z lies in S, and we also did the case where z lies in S′.

And that completes the proof of the simpler claim which also completes the proof of the key fact that the vertex with the highest postorder number lies in a source SCC. That completes the proof of correctness of our SCC algorithm and that's the end of our SCC algorithm description.

It's quite an amazing fact. We can take any directed graph and we can compute its strongly connected components and we can topologically sort the strongly connected components. And we do it with just two runs of the vanilla DFS.

(Slide 24) BFS/Dijkstra’s
***



BFS/Dijkstra's

DFS: connectivity

BFS: input: $G=(V,E)$ & $s \in V$

$O(n+m)$

output: for all $v \in V$, $dist(v) = \min$ # of edges from s to v ($=\infty$ if no path) & $prev(v)$

Dijkstra's: input: $G=(V,E)$, $s \in V$, $\ell(e)>0$ for every $e \in E$

output: for all $v \in V$, $dist(v) =$ length of shortest $s \rightsquigarrow v$ path

$O((n+m)\log n)$ Using min-heap

We've seen how to use the DFS algorithm to solve connectivity problems in undirected and directed graphs. Let's quickly remind you of some other common algorithms for exploring graphs.

As opposed to DFS, which is Depth First Search, BFS is Breadth First Search. BFS explores the graph in layers. The input to the BFS algorithm is similar to the DFS algorithm. It's an undirected or directed graph G in adjacency list representation. But BFS has an additional input parameter. We specify a start vertex which we denote as little s. BFS returns the distance for every vertex from the start vertex little s. The graph G is unweighted, so the distance is defined as the minimum number edges to get from vertex s to vertex v. Now if there is no path from s to v, then this distance is defined as infinite.

Now how do we get such a path of minimum length? Well, BFS also returns this previous array, which one can use to construct a path of minimum length from s to v.

Now what's the running time of the BFS algorithm? BFS, like DFS, is linear time - the running time is O(n+m), where n is the number of vertices in the graph G, and m is the number of edges in the graph G.

Dijkstra's algorithm is a sort of more sophisticated version of BFS. It solves a similar problem as

BFS, but instead, it considers a weighted version of the graph G. As in the BFS algorithm, the input to Dijkstra's algorithm is a graph G. It could be directed or undirected, and we have a specified start vertex, s. But Dijkstra's algorithm has an additional input parameter. We were given a weight, a length, for every edge, and this length has to be positive. What is the output of Dijkstra's algorithm? Well, it is the weighted analog of the BFS output, so it outputs this array dist and dist(v) is the length of the shortest path from s to v.

One of the key requirements of Dijkstra's algorithm is that these edge lengths are positive. If you want to know how to deal with negative edge lengths, then you should refer to our dynamic programming lecture, DP3.

Dijkstra's algorithm uses the BFS framework with the min-heap data structure. The min-heap data structure is often called the priority queue. Each operation in the min-heap data structure takes O(log n) time, so we get an additional log and factor on the BFS running time, and hence, the total runtime of Dijkstra's algorithm is O((n+m) log n).

Now there are other variants of Dijkstra's algorithm with different data structures that they utilize. We'll always refer to using the min-heap data structure in this class. And for concreteness, in this class we'll say the running time of Dijkstra's algorithm is O((n+m) log n).

I assume that many of you have seen BFS and Dijkstra's many times in the past. If you need a quick review, I suggest you look at chapter 4 of the textbook.