[CREAKING] [CLICKING] [SQUEAKING] [RUSTLING] [CLICKING]

JUSTIN SOLOMON: We just started a new unit on graph theory, which is going to be sort of our focus for the next couple of lectures in 6006. And so I thought we'd give it a little bit of review at the beginning the lecture because, as usual, I've muddled together a lot of notions in our previous lecture, and then start with some new ideas.

So basically, in our previous lecture, we talked about an algorithm called breadth-first search. And then almost always you see that paired with a second algorithm called depth-first search. And following tradition, and basically logic, we'll do the same thing in 006 today. But in any event, for today we'll stick to the technical material.

So as a little bit of review, I guess actually, the one thing I didn't do on this slide was actually draw a graph. So we should probably start with that. So if you recall, graph is a collection of nodes or vertices depending-- I don't know, is it like a European American thing or something-- and edges.

So here's an example, which as usual, I'm not managing to draw particularly clearly. So this graph is kind of like a cycle. So I have directed edges here, here, here, and here. And of course, there are many kind of variations on the theme, right?

So our basic sort of definition of a graph is that we have some set V, which is like the set of vertices. And then we have a set E, which is set of edges. And this was a subset of V cross V. And this is nothing more than fancy notation for saying that an edge is a pair of vertices, like a from and a to vertex.

Of course, there are many variations on this theme. You could have a directed versus an undirected graph. So this one is directed, meaning the edges look like arrows. If they didn't have arrowheads, they'd be undirected.

We define something called a simple graph where you have essentially no repeated edges. So for instance, you can't do something like this where you have the same edge twice.

And then there are a couple of different definitions that were kind of useful. So in particular-- I'm going to erase this, whoops-- useless edge here. Maybe make my graph slightly more interesting. So add another edge going in the reverse direction.

So maybe I have-- I'm going to give my vertices labels. x, y, z, and w. Then we talked about the neighbors of a given vertex, which are the vertices that you can reach by following edges in or out of your vertex. So in particular, the outgoing neighbors, which we sort of implicitly defined in our previous lecture but didn't call it out, we're going to notate with Adj+.

And these are all of the things that you can reach by going out of a vertex into the next one. So for example, Adj+ of w is going to be the set of vertices. We'll notice I can get from w to y and also from w to z. Yeah. So. Nope, nope. y comma z. OK.

So to continue just our tiny amount of review for the day, remember that a graph-- there are many different ways to represent a graph. The sort of brain dead one would be just like a big long list of edges. But of course, for our algorithms it's not a particularly efficient way to check things like, does this edge exist in my graph.

So the basic representation that I think we're mostly working from in this course is to think of a graph like a set of vertices, each of which maps to another set of vertices. So roughly every vertex maybe stores its outgoing set of edges. And so this is kind of nice because, of course, very quickly we can answer questions like, is this edge inside of our graph? Or we can iterate over the neighbors of a vertex and so on, which are the kind of typical things that we do in a lot of graph algorithms.

And then finally, in our previous lecture, we started talking about paths. So a path is like a chain of vertices that can get me from one vertex to the other only following edges of my graph. There is a term that I think I forgot to define last time because it didn't really matter a ton, which is a simple path, which is just a path that doesn't have the same vertex more than once.

And then, of course, there are many different questions you could ask about a graph that are basically different problems involving computing paths. So for instance, the shortest path between two vertices is sort of our canonical one in graph theory. Or you could ask questions about reachability and so on.

So there's our basic review from our previous lecture. Does our course staff have any questions about things so far? Excellent.

OK. And there's one additional piece of terminology that I fudged a little bit last time-- or rather, my co-instructor suggested a bit of an attitude adjustment. So I thought I'd better clarify really quick.

There's this interesting phrase, linear time, which we all know and love in computer science theory. And this sort of implicit thing, especially in this course, is that when we say linear time, we mean in the size of the input. Right? And so if we have a linear time graph algorithm, well, how much space does it take to store a graph? Well, we need a list of vertices and a list of edges, if nothing else.

So a reasonable way to interpret this phrase linear time is that it's an algorithm that looks like what we've shown on the screen. The times proportional to maybe the sum of the number of vertices and the number of edges. If that makes you uncomfortable like it does for me because one of these can kind of scale on the other, I think it's always fine to add more detail. Right? So if you want to say, linear in the sum of the number of vertices and edges, that's perfectly fine.

But if you see this phrase, that's how you should interpret it. Hopefully that's a fair way to put it. Excellent.

OK. So last time, we talked about an algorithm called breadth-first search-- BFS, for those in the know. Breadth-first search is an algorithm. And the reason we use the word breadth is because it's kind of, remember, we talked about level sets last time because we talked about breadth-first search in the context of computing shortest paths.

And in particular, we have our source node all the way on the left-hand side. And then breadth-first search constructed all the nodes that were distance 1 away. Right. That's the first level set, and then all the distance 2 away, and then all the distance 3 away, and so on. So in particular, the level set L3 isn't visited until we're completely done with level set L2.

Today, we're going to define another algorithm, which is called depth-first search, which doesn't do that, but rather, starts with its first vertex and just starts walking all the way out until it can't do that anymore. And then it kind of backtracks. That's one way to think about it.

And so somehow, in breadth-first search, we're like, drawing concentric circles. In depth-first search, we're doing the opposite. We're like, shooting outward until we reach the outer boundary, and then exploring the graph that way.

OK. And these are sort of the two extremes in terms of graph search kind of techniques that are typically used under the basic building blocks for algorithms in graph theory.

So in order to motivate and think about depth-first search, we're going to define a second problem, which is closely related to shortest path, but not exactly the same. And that's the reachability problem.

So here I have the world's simplest directed graph. So the black things are the edges. And the circles are the nodes or the vertices. And I've marked one special node in blue. And his name is the source node.

And now the question I want to ask is, what are all of the other nodes in my graph that I can reach by following edges-- directed edges-- starting with the source? So obviously, I can get to the node in the lower right, no problem. And of course once I get there, I can traverse and edge upward to get to that second green vertex.

Notice that I was really sneaky and evil, and I drew edges in this graph that might make you think that the red node is reachable. The red one being on the upper left. I'm realizing now that for colorblind people, this isn't a great slide.

But of course, because all the edges from the red vertex on the left here point out, I can't actually reach it from the blue source node. So the reachability problem is just asking, which nodes can I reach from a given source? Pretty straightforward, I think.

Of course, there are many ways to solve this. Right? In fact, one way we could do it would be to use our previous lecture. We could compute the shortest path distance from the source to all the other nodes. And then what would the length of the shortest path from the source to an unreachable node be? Any thoughts from our audience here?

Infinity. Thank you, Professor Demaine. Right.

So in addition to this, of course, a totally reasonable question, thinking back to our shortest path lecture, there are sort of two queries we might make. Right? One is just what is the length of the shortest path? The other is like, what is the actual shortest path from the source to a given vertex?

We can ask a very similar thing here, which is like, OK. You tell me that the green guy is reachable, but how? Give me a path as evidence or a certificate, if you want to be fancy about it.

So in order to do that, just like last time, remember, we defined a particular data structure that was the shortest path tree. We can do something very similar here. In particular, this is like the extent of my PowerPoint skills here. If I have a reachability problem, I can additionally store-- I can decorate every node in my graph with one other piece of information, which is the previous node along some path from my source to that thing. Right?

And just like last time, if I want to get an actual path from the source to w, what could I do? I can start with w and then just keep following those parent relationships until I get back to the source. Then if I flip the order of that list of vertices, I get a path from the source to the target that's valid.

So this object is called a path tree, just like we talked-- or a parent tree, rather. Just like we talked about in our last lecture, there's no reason why this thing should ever have a cycle in it. It's certainly a tree.

Right. So that's the basic reachability problem. And in addition to that, we can compute this object P, which is going to give me sort of information about how any given node was reachable. There's a slight difference between the parent tree that I've defined here and the shortest path tree, which I defined last time, which is, I'm not going to require that the shortest path I get-- oh, man-- the path I get when I backtrack along my tree P is the shortest path, it's just a path because for the reachability problem, I actually don't care.

Like, I could have a weird, circuitous, crazy long path. And it still tells me that a node is reachable. Right.

So that's our basic set up and our data structure. And now we can introduce a problem to solve reachability. Again, we already have an algorithm for doing that, which is to compute shortest paths. And remember that our shortest path algorithm from previous lecture took linear time and the size of the input. It took v plus e time.

Now the question is, can we do a little better? The answer, obviously, is yes, because I just asked it, and I gave you this problem. OK.

And here's a technique for doing that, which unsurprisingly, is a recursive algorithm. I'm going to swap my notes for my handwritten notes. And this algorithm is called depth-first search. And here's the basic strategy. I'm going to choose a source node and label that Node 1 here. I suppose it actually would have made sense for me to actually 0 index this. Maybe in the slides I'll fix it later.

But in any event, I'm going to mark my source node. And now I'm going to look at every node, every edge coming out of that node, and I'm going to visit it recursively. So that's our sort of for loop inside of this function visit.

And then for each neighboring node, if I haven't visited it before, in other words, I currently haven't given it a parent. That's our if statement here. Then I'm going to say, well, now they do have a parent. And that parent is me. And I'm going to recurse.

You guys see what this is doing? It's kind of crawling outward inside of our graph. So let's do the example on the screen.

And I purposefully designed this experiment-- or this example-- to look a little bit different from breadth-first search, at least if you choose to do the ordering that I did. So here's our graph. 1, 2, 5, 3, 4. OK.

And let's think about the traversal order that the depth-first search is going to do. Right. So here's our source. And now what does the source do? It rec-- so let's think about our recursion tree. So we have the source all the way up in here. And now he's going to start calling the visit function recursively.

So. And I'll go ahead and number these the same as on the screen. Well, he has one outgoing neighbor, and it hasn't been visited yet. So of course, the very first recursive call that I'll make is to that neighbor 2.

Now the neighbor 2 also recurses. Hopefully this kind of schematic picture makes some sense, what I'm trying to draw here. And well now, the 2 has two neighbors, a 3 and a 5. So let's say that we choose 3 first.

Well, the 3 now recurses and calls 4. And then the recursion tree is kind of done. So now it goes back out.

And then finally, well, now, the 3-- or, oh, boy. Yeah. The 2 looks at his next neighbor, which is the 5 and visits that recursively.

Notice that this is not following the level sets. Right? The depth-first search algorithm got all the way to the end of my tree in the recursive calls and then kind of backed its way out to the 2 before calling the 5. These are not the same technique. One goes all the way to the end and then kind of backtracks. When I say backtrack, what I mean is the recursion is kind of unraveling. Whereas in breadth-first search, I visit everything in one level set before I work my way out.

That distinction make sense? OK. So of course, we need to prove that this algorithm does something useful. So let's do that now.

So in particular, we need a correctness proof. So our claim is going to be that-- let's see here-- the depth-first search algorithm visits all, I guess reachable v, and that it correctly sets the parent in the process.

OK. So in order to prove this, of course, as with almost everything in this course, we're going to use induction. And in particular, what we're going to do is do induction on the distance from the source. So we're going to say that, like, for all vertices in distance k from the source, this statement is true. And then we're going to prove this inductively on k. OK?

So we want to do induction on k, which is the distance to the source vertex. So as with all of our inductive proofs, we have to do our base case and then our inductive step. So in the base case, k equals 0. This is a hella easy case because, of course, what is the thing that is distance 0 from the source? It's the source! Yeah.

And take a look at our strategy all the way at the top of the slide. We explicitly set the correct parent for the source, and in some sense, visit it because the very first thing we do is call visit of s. So there's kind of nothing to say here. Yeah? Or there's plenty to say if you write it on your homework. But your lazy instructor is going to write a check mark here. OK.

So now we have to do our inductive step. So what does that mean? We're going to assume that our statement is true for all nodes within a distance k. And then we're going to prove that the same thing is true for all nodes within a distance k plus 1. OK.

So let's do that. Let's consider a vertex v that's distance k plus 1 away. So in other words, the distance from the source to v is equal to k plus 1. And what's our goal? Our goal is to show that the parent of v is set correctly. Yeah?

What was that?

AUDIENCE: [INAUDIBLE].

JUSTIN SOLOMON: Oh, sorry. I forgot that the distances in this class are in order. Yeah. That's absolutely right. So it should be the distance from s to v. Yeah. Sorry. I'm really not used to thinking about directed graphs. But that's a good fix. OK.

So now what can we do? Well, there's this number is distance here. So in particular, the shortest path from s to v. So remember our argument last time that essentially, when we look at shortest path and we kind of truncate by 1, it's still shortest path? That property doesn't matter so much here. But at least we know that there's another vertex on the path, which is 1 distance from one less away.

So let's take u, which is also a vertex, to be the previous node on the shortest path from s to v. Right. And so in particular, we know that the distance from s to u is equal to k. And conveniently, of course, by our inductive hypothesis here, we know that our property is true for this guy. OK.

So now our algorithm, what do we know? Well, because our property is true, the visit function at some point in its life is called on this vertex u. That's sort of what our induction assumes.

So we have two cases. Right. So when we visit u, we know that when we call this visit function, well, remember that v kind of by definition is in Adj+ of u. Right. So in particular, DGS is going to consider v when it gets called. OK.

And now there's two cases. Right? So either when this happens, P of v does not equal None. Right.

Well, what does that mean? Well, it means that we already kind of found a suitable parent for v. And we're in good shape. Otherwise, p of v does equal None. Well, in this case, the very next line of code correctly sets the parent. And we're all set.

So in both of these two cases, we show that the parent of u was set correctly either by that line of code right here or just previously. And so in either case, our induction is done. All right. I guess given the feedback I received from our previous lecture, we now can end our LaTeX suitably.

OK. So what did we just show? We showed that the depth-first search algorithm can dig around in a graph and tell me all of the things that are searchable, or rather, are reachable from a given source, just basically by calling visit on that source and then expanding outward recursively. OK. So I think this is certainly straightforward from an intuitive perspective. It's easy to get lost when you write these kind of formal induction proofs because they always feel a tiny bit like tautology. So you should go home and kind of convince yourself that it's not.

OK. So of course, what do we do in this class? We always follow the same kind of boring pattern. The first thing we do, define an algorithm. Second thing we do, make sure that it's the right algorithm. What's the third thing we need to do?

AUDIENCE: Analyze it.

JUSTIN SOLOMON: Analyze it. That's right. In particular, make sure that it finishes before the heat death of the universe. And indeed, depth-first research doesn't really take all that long, which is a good thing.

So let's think about this a bit. So what's going to end up happening in depth-first search, well, we're going to visit every vertex at most once, kind of by definition here. And in each case, we're going to just visit its neighboring edges.

Can we ever traverse an edge more than one time? No. Right. Because the visit function only ever gets called one time per vertex. And our edges are directed. Right. So kind think about the from of every edge, the from vertex is only ever visited one time. And hence, every edge is only visited one time.

Do we ever visit-- ah, yes.

AUDIENCE: Does DFS work for an undirected graph?

JUSTIN SOLOMON: An undirected graph. Absolutely. So there's sort of different ways to think about it. One is to think of an undirected graph like a directed graph with two edges pointed either way, which I think is in this class how we actually kind of notated it in the previous lecture.

Yeah. Actually, that's probably a reasonable way to reduce it. So let's stick with that.

Right. Now, does DFS ever visit a vertex that is not reachable from the source? Well, the answer is no because all I ever do is recursively call on my neighbors. And so kind of by definition, if I'm not reachable, DFS will never see it. So if I think about my runtime carefully, it's not quite the same as breadth-first search.

Remember that breadth-first search took v plus e time. In depth-first search, it just takes order e time because I'm expanding outward from the source vertex, hitting every edge adjacent to every vertex that I've seen so far. But I never reach a vertex that I haven't-- that isn't reachable. Right?

And so because this only ever touches every edge one time, we're in good shape. And I see a question here. Yeah.

AUDIENCE: Does BFS reach vertices that are not reachable?

JUSTIN SOLOMON: Does BFS reach vertices that are not reachable? I guess not, now that you mention it. But at least in my boring proof of order v time last time, our very first step of BFS, reserve space proportional to v, which is enough to already make that runtime correct. Good question. Yeah.

So I guess the way that we've talked about it where you can stretch one little set after a time, if you think of that as reachability, then no. It doesn't reach it in the for loop. But just by construction, when we started we already took the time that we're talking about here.

So notice these run times aren't exactly the same. So for example, if my graph has no edges, BFS still is going to take time because it still has to take order v time, at least in the sort of brain-dead way that we've implemented it last time. Obviously, in that case, we could probably do something better. Whereas the way that we've defined the DFS algorithm, it only takes edge time.

I see confusion on my instructor's face. No? OK. Good.

The one thing to notice is that these are algorithms for slightly different tasks in some sense. The way that we wrote down breadth-first search last time, conveniently, it gives us the shortest path. There are breadth-first search algorithms that doesn't. I think in this class we kind of think of breadth-first search-- we motivate it in terms of the shortest path problem. But it's just kind of a strategy of working outwards from a vertex.

Whereas here, the way we've written down depth-first search, there's no reason why the path that we get should be the shortest. Right? So to think of a really extreme example, let's say that I have a cycle graph. So I get a big loop like this. Let's say that I do depth-first search starting from this vertex.

Well, what will happen? Well, this guy will call its neighbor recursively, who will then call its neighbor recursively, who will then call his neighbor recursively, and so on. So of course, when I do depth-first search, when I get to this vertex, there's a chain of 1, 2, 3, 4 vertices behind it.

Is that the shortest path from the source to the target here? Well, clearly not. Right? I could have traversed that edge. I just chose not to.

OK. So that's the depth-first search algorithm. It's just essentially a recursive strategy where I traverse all my neighbors, and each of my neighbors traverses their neighbors, and so on.

OK. So why might we want to use this algorithm? Well, we've already solved the reachability problem. So let's solve a few more things using the same basic strategy here.

So there's some notions that we've sort of-- actually, in some sense, already used in the lecture here. But we might as well call them out for what they are, which is this idea of connectivity. So a graph is connected if there's a path getting from every vertex to every other vertex. Right.

Now connectivity in a directed graph is kind of a weird object. Like, for instance, think of a directed graph with just two edges. And one edge goes from u to v. Then I can get from v to u, but not vise versa. That's kind of a weird notion.

So here in 6006 we'll mostly worry about connectivity only for undirected graphs because they're-- the vertices just basically come in like, big connected clumps. Or the more technical term for a big connected clump is a connected component. Yeah?

So let's see an example. So let's say that I have a graph, which has an edge and then a triangle. This is one graph. Do you see that? There's a collection of vertices, and there's a collection of edges.

But it has two connected components-- the guy on the right and the guy on the left, meaning that each vertex here is reachable from every other vertex here. Each vertex here is reachable from every vertex here. But there's no edge that goes from the triangle to the line segment. Yeah?

And so in the connected components problem, we're given a graph like this guy. And initially, we don't, you know-- OK. When I draw it like this, it's pretty clear that my graph has two connected components. Maybe my graph-embedding algorithm failed and it drew an edge like that. Well, then maybe-- I don't know-- it's still pretty obvious that there's two connected components.

But you can imagine a universe where you don't know that a priori. And the problem you're trying to solve is just to enumerate all these clumps of vertices that are reachable from one another in an undirected graph. And conveniently, we can use depth-first search to solve this problem pretty easily. Right?

So how could we do it? Well, in some sense how can we find one connected component? So let's say that I just choose a vertex in my graph. Well, what do I know about everything in its connected component?

Well, it's reachable from that vertex. Remember, we just solved the reachability problem, which says, if I have a vertex, I can now tell you all the other vertices that are reachable from this guy. So I could call DFS on, well, any vertex of this cycle here. Call the reachability thing.

And I know that for every vertex there's one of two things. Either the vertex has a parent in that object P, or it's the source. So I can very easily find the connected component corresponding to that vertex. Does that makes sense?

Have I found all the connected components? No. I found one. I found the one corresponding to the arbitrary vertex that I just chose. So how could I fix this?

Well, it's super simple. I could put a for loop on the outside, which just loops over all the vertices, maybe. And if that vertex is not part of a connected component yet, then I need to make a new one.

So then I call DFS on that vertex. I collect all the vertices that I got. And I iterate.

So this is the algorithm that in this class we're going to call full DFS. By the way, you could do the same thing with full breadth-first search. That's perfectly fine. Just kind of by analogy here.

Right. So what is full D-- oh, this chalk is easier. Well, I'm going to iterate over all my vertices. Where I stands for for loop. Of-- right. So if v is unvisited, then I'm going to do to DFS starting at v. I guess we used visit to refer to this in the previous slide.

And that's going to kind of flood fill that whole connected component. And then I can collect that connected component and continue. We have to be a little bit careful because, of course, we don't want like, checking things-- something to be visited to somehow take a bunch of time and make my algorithm slower than it needs to be. But of course, we have a set data structure that we know can do that and order one time at least in expectation.

OK. So this is the full DFS algorithm. It's really simple. Of DFS because I called DGS on every vertex. And it's full because I looped over all the vertices. Right.

And so if we think about it, how much time does this algorithm take? It's little bit sneaky because somehow I have a for loop over all the vertices. Then I could imagine a universe where I get, like, vertices times some other number because there's a for loop, and then there's something inside of it. I think that's how we're used to thinking about runtime of for loops.

But in this case, that actually doesn't happen because there's never a case where an edge gets traversed more than one time. Because if I'm in one connected component, then by definition, I can't be in another connected component. Right?

And so what happens is, in some sense, this innocent looking call to DFS-- I suppose if you were like a LISP or a programmer, you somehow wouldn't like this. It has a side effect, which is that I marked all the vertices in that connected component as "don't touch me again." Right. And so implicitly I kind of removed edges in this process.

So if you think through it carefully, the runtime of this full DFS algorithm is v plus e time because every edge is touched no more than one time. Kind of amortized over all the different calls to DGS here. And there's this for loop over vertices. So there's clearly an order v that you need here.

Does that argument make sense? So again, we call that linear in the size of the input. I'm going to say it as many times to get it in my own head correctly. OK.

Right. So this is the basic problem. This comes up all the time, by the way. Like, it seems like somehow a totally brain dead weird algorithm. Like, somehow, why would you want an algorithm that finds connected components. Like, why would you even have a graph that's disconnected or something?

But of course, that can happen a lot. So for instance, maybe you work at a social media company, and people have friends. But like, Eric and I are friends. And we're not friends with anybody else. We have a-- there's like, a blood oath kind of thing.

Then that might be not so easy to find in the graph because, of course, we're just two among a sea of students in this classroom, all of which have different interconnections that are just enumerated based on the list of edges. And so even though like, pictorially, it's kind of hard to draw a connecting component algorithm in a way that doesn't make it sound kind of like a useless technique from the start, because it's very clear there are two connected components there. Of course, we still have to be able to write code to solve this sort of thing.

OK. So for once, I think I'm almost on time in lecture today.

So we have one additional application of depth-first search in our class today, which is sort of on the opposite end of the spectrum. So we just talked about graphs that are undirected and thinking about cycles. Now, on the opposite end we might think of a DAG. So a DAG is a Directed Acyclic Graph.

Can anyone think of a special case of a DAG? I suppose I should define it first. And then we'll come back to that question, which means exactly what it sounds like. So it's a graph that has directed edges now and doesn't have any cycles in it.

So actually, the graph I gave you all the way at the beginning of lecture I think secretly was an example of one of these. So let's say that I have directed edges. Maybe if I make the head a triangle, it's a little easier to see. I'm not so sure. In any event, so I'm going to have an edge up and an edge to the right, and similarly, an edge down and an edge to the right.

This graph looks like a cycle. But it's not because the only direction that I can move is from the left-hand side to the right-hand side. So this is a directed graph. And it doesn't contain any cycles, meaning there's no path that it can take from a vertex that gets back to itself along the directed edges.

OK. And DAGs show up all the time. Now that I've defined what a DAG is, can somebody give me an example of a DAG that we've already seen in 6006?

AUDIENCE: A tree.

JUSTIN SOLOMON: A tree. At least if we orient all all the edges kind of pointing downward in the tree. Yeah. Otherwise, it gets kind of debatable over whether it's a DAG or not. If there's no direction to the edges, then somehow the definition just doesn't apply.

OK. So in processing directed acyclic graphs, there's a really useful thing that you can do that's going to show up in this class apparently quite a bit, which is kind of interesting to me, I'm curious to see what that looks like, which is to compute a topological order on the graph. We're at topologies here. So as a geometry professor in my day job, I get all excited.

But in this case, a topological order is a fairly straightforward thing. Actually, it's defined on the screen, and I have bad handwriting anyway. So let's just stick with that.

So topological ordering. So we think of f as a function that assigns maybe every node an index in array. I guess I shouldn't use the word array here. But just like an index, an ordering. So like, this is the first vertex. And this is the second vertex. And so on.

Then a topological order is one that has the properties shown here, which is that if I have a directed edge from u to v, then f of u is less than f of v. So in other words, if I look at the ordering that I get on my topological order, u has to appear before v. Yeah?

Let's look at our example again. So let's give our nodes names. So here's A, B, C, D. Well, what clearly has to be the first node in my topological order?

A. Right. It goes all the way to the left-hand side. Yeah.

Well, after that it's a bit of a toss-up. What do we know? We know that B and C have to appear before D. So maybe just to be annoying, I do A, C, B-- that's a B-- and then D. So it's a topological order. Notice that things that are on the left appear in my graph before things that are on the right, where the word "before" here means that there's an edge that points from one to the other. OK.

By the way, are topological orderings unique? No. So if we look at our graph example here, ABCD is also a topological order. And what that means is somehow very liberating. It means that when we design an algorithm for finding a topological order, so there's some design decisions that we can make. And we just have to find one among many.

But in any event, we're going to define a slightly different notion of order. And then we're going to show that they're closely linked to each other. And that is the finishing order.

So in the finishing order, we're going to call full DFS on our graph. Remember, that means we iterate over all our nodes. And if we haven't seen that node yet, we call DFS on it. And now we're going to make an order in which as soon as the call to a node in that visit function is complete, meaning I've already iterated over all my neighbors, then I add my node to the ordering.

That make sense? It's like a little bit backward from what we're used to thinking about. So it's the order in which full DFS finishes visiting each vertex. Yeah?

And now here's the claim, is that if we have a reverse finishing order, meaning that we take the finishing order and then we flip it backward. That's exactly going to give us a topological ordering of the vertices in our graph. Right.

So let's do that really quickly. So in other words, our claim here-- I think, yeah, let's see-- is that if I have a directed graph. So G is a DAG. Then let's see here. Then the-- oops. My notes are backwards. So I should switch to my-- Jason's notes, which of course, are correct.

Right. So if I have a graph that's a DAG, then the reverse of the finishing order is a topological order.

By the way, we're not going to prove the converse that if I have a topological order, that somehow that thing is the reverse of DFS, at least the way that maybe I coded it. There's a slightly different statement, which is, does there exist a DFS that has that ordering? But that's one that we'll worry about another time around piazza or whatever.

OK. So let's see here. So we need to prove this thing. So what are we going to do?

Well, what do we need to check is the topological order is that if I look at any edge of my graph, it obeys the relationship that I have on the screen here. So in particularly, we're going to take uv inside of my set of edges. And then what I need is that u is ordered before v using the reverse of the finishing order that we've defined here. OK.

So let's think back to our call to the DFS algorithm, where call this visit function. Right. So we have two cases. Either u is visited before v. Or it ain't. Yeah.

So let's do those two cases. So case Number 1 is, u is visited before v. OK.

All right. So what does that mean? Well, remember that there's an edge. Like, pictorially, what's going on?

Well, there's all kinds of graph stuff going on. And then there's u. And we know that there's a directed edge from u to v. That's our picture. Right? And maybe there's other stuff going on outside of us.

So in particular, well, just by the way that we've defined that visit function, what do we know? We know that when we call visit on u, well, v is one of its outgoing neighbors. So in particular, a visit on u is going to call visit v. And we know that because well, u is visited before v.

So currently, v's parent is l when I get to you That make sense?

Now, here's where reverse ordering, we're going have to keep it in our head because now, well, visit of u calls visit of v. So notice that visit of v has to complete before visit of u. Right?

V completes before visit of u. Well. So in reverse, sorting-- in reverse finishing order here, what does that mean? Well, if this completes before the other guy, then they get flipped backward in the list, which is exactly what I want because there's an edge from u to v.

OK. So Case 1 is done. Now we have Case 2, which is that v is visited before u.

OK. So now I'm going to make one additional observation. OK. So now I'm going to go back to my other notes because I like my schematic better.

Right. So what's our basic picture here? Oh, no. I-- Oh, you know what it was? I printed out another copy of this. That's OK. I can do it off the top of my head.

OK. So here's my source vertex. His name is S. Now, there's a bunch of edges and whatever. There's a long path. And now eventually, what happens?

Well. I have a node v. And somewhere out there in the universe is another node u. And what do I know?

I know that by assumption, I know that there's an edge from u to v. That make sense? So that's our sort of picture so far.

OK. So what do we know? We know that our graph is acyclic. Yeah? Kind of by definition, that's our assumption.

So can we reach u from v? In other words, does there exist a path from v to u? So that would look like this.

No because our graph is acyclic, and I just drew a cycle. So this is a big X. There's a frowny face here. Can't do it. He has hair, unlike your instructor.

OK. So right. So what does this mean? Well, OK. So by this picture, I suppose, we know that u cannot be reached from v.

Yeah. So what does that mean? Well, it means that the visit to v is going to complete and never see u because remember, the visit to v only ever call things that are kind of descendants of v. So in other words, visit of v completes without seeing u.

Well, that's exactly the same thing that we showed in our first case. Right? So by the same reasoning, what does that mean? In our reverse finishing order, the ordering from u to v is preserved.

OK. So that sort of completes our proof here that reverse finishing order gives me a topological order, which is kind of nice. And so this is a nice convenient way of taking all of the nodes in a directed acyclic graph and ordering them in a way that respects the topology or the connectivity of that graph.

So we're going to conclude with one final problem, which I don't have a slide on. But that's OK. And that's cycle detection.

So there's a bit of an exercise left to the reader here. So the problem that we're looking for now is that we're given a directed graph. There's a G in graph, in case you're wondering. But now, we don't know if it's a DAG or not.

And so the question that we're trying to ask is, does there exist a cycle in our directed acyclic graph? So we're just given our graph, and we want to know, can we do this?

Let's think through the logic of this a bit. So what do we know? We know that if our graph were a DAG, then I could call DGS, get the ordering out, and then I guess flip its ordering backwards. So I could compute the reverse finishing order. And it would give me a topological order of my graph.

So if I were a DAG, I would get a topological order when I call DFS. So let's say that I ran DFS. I got whatever ordering I got. And now I found an edge the points in the wrong direction.

I can just double check my list of edges, and I find one that does not respect the relationship that I see in the second bullet point here. Can my graph be a DAG?

No. Because if my graph were a DAG, the algorithm would work. I just proved it to you. Right?

So if my graph were a DAG, then I could do reverse finishing order. And what I would get back is a topological order. So if I found a certificate that my order wasn't topological, something went wrong, and the only thing that could go wrong is that my graph isn't a DAG. Yeah. Isn't a DAG.

In fact, sort of an exercise left to the reader and/or to your section-- do we still have section? I think we do, as of now-- is that this is an if and only if, meaning that the only time that you even have a topological ordering in your graph is if your graph is a DAG. This is a really easy fact to sanity check, by the way. This is not like, a particularly challenging problem.

But you should think through it because it's a good exercise to make sure you understand the definitions, which is to say that if you have a topological order, your graph is a DAG. If you don't have a topological order, your graph isn't a DAG.

So in other words, we secretly gave you an algorithm for checking if a graph is a DAG at all. Right? What could I do? I could compute reverse finishing order. Check if it obeys the relationship on the second bullet point here for every edge. And if it does, then we're in good shape. My graph is a DAG.

If it doesn't, something went wrong. And the only thing that could have gone wrong is not being a DAG.

OK. So in other words, secretly we just solved-- well, I guess the way that I've written it here, we've solved the cycle detection problem here, which is to say that, well, I have a cycle if and only if I'm not a DAG, which I can check using this technique. Of course, the word "detection" here probably means that I actually want to find that cycle, and I haven't told you how to do that yet.

All we know how to do so far is say, like, somewhere in this graph there's a cycle. And that's not so good. So we can do one additional piece of information in the two minutes we have remaining to sort of complete our story here, which is to modify our algorithm ever so slightly to not only say thumbs up, thumbs down, is there a cycle in this graph, but also to actually return the vertices as a cycle.

And here's the property that we're going to do that, which is following, which is that if G contains a cycle, right, then full DFS will traverse an edge from a vertex v to some ancestor of v. I guess we haven't carefully defined the term "ancestor" here. Essentially, if you think of the sort of the running of the DFS algorithm, then an ancestor is like something that appears in the recursive call tree before I got to v.

OK. So how could we prove that? Well, let's take a cycle. And we'll give it a name. In particular, we'll say that it's a cycle from v0 v1 to vk. And then it's a cycle, so it goes back to v0. OK.

And I can order this cycle any way I want. Notice that if I permute the vertices in this list in a cyclical way, meaning that I take the last few of them and stick them at the beginning of the list, it's still a cycle. That's the nice thing about cycles.

So in particular, without loss of generality, we're going to assume that v0 is the first vertex visited by DFS.

What does that mean? That means, like, when I do my DFS algorithm making all these recursive calls, the very first vertex to be touched by this technique is v0. OK.

Well, now what's going to end up happening? Well, think about the recursive call tree starting at v0. By the time that completes, anything that's reachable from v0 is also going to be complete. Do you see that?

So for instance, v0 somewhere in its call tree might call v2. And notice that v2 was not already visited. So in fact, it will. For v1 I got to call v2 and so on.

And in particular, we're going to get all the way to vertex k. Right? So in other words, we're going to visit a vertex vk. And notice what's going to happen. So remember our algorithm. In fact, we should probably just put it up on the screen would be easier than talking about it a bunch.

Well, vk is now going to iterate over every one of the neighbors of vk. And in particular, it's going to see vertex v0. Right? So we're going to see the edge from vk to v0, which is an edge kind of by definition because we took this to be a cycle here.

But notice that's exactly the thing we set out to prove, namely that full DFS traverses an edge from a vertex to one of its ancestors. Here's a vertex k. Here's the ancestor v0. Why do we know that it's an ancestor? Well, because v0 was called in our call tree before any of these other guys. Right?

So we wanted an algorithm that not only did cycle detection, but also actually gave me the cycle. What could I do? Well, it's essentially a small modification of what we already have. Right.

So-- whoops. Right. If I want to compute topological order or whatever, I can just do DFS. And that'll tell me like, yay or nay, does there exist a cycle.

If I want to actually find that cycle, all I have to do is check that topological order property at the same time that it traversed the graph during DFS. And the second that I find an edge that loops back, I'm done. And so that's our basic algorithm here. And this is a technique for actually just finding the cycle in a graph using the DFS algorithm.