[SQUEAKING]

[RUSTLING]

[CLICKING]

ERIK DEMAINE: All right. Welcome back to graphs. This will be our last lecture that's fully about graphs. And we're going to move away from what we've been talking about for the last three lectures, simple undirected graphs, and move to the other kind of graph we talk about in this class, which is directed graphs. Which we sometimes shorten to digraph.

So from now on, we're going to have to be explicit. Every time we talk about a graph, we should say directed or undirected. Simple is implicit when we say undirected. But we should distinguish.

So the motivation is what if your connections only go in one direction? Maybe you have one way roads, like all of Boston, or you have one way follow relationships instead of friendships, or you have network connections you can only send in one direction. Those do exist, though they're less common. And so directed graphs are going to model that for us. So things are going to look very similar to the old definition. Just the edges are different.

So you may recall we had this cross product notation. I'm going to remind you so we can compare. This is a set of ordered pairs of vertices, in this case. So a very similar property here. Edges were unordered sets of two vertices. Here they are ordered pairs of two vertices.

The other difference is here, we forbade having a loop. Here we allow it. So this was not allowed over here, but allowed over here. Still over here we forbade two edges connecting the same two vertices. Here we allow two edges going in opposite directions. So this was a no. We still forbid in this picture having two edges going in the same direction between the same two vertices. That's a consequence because e is a set. You're only allowed to have one of each.

So that's the definition. Very simple. Here's a little example of on six vertices. There's a floating vertex out here f. So you see sometimes you can have an edge that goes in both directions. Sometimes you have an edge that only goes in one direction. Maybe sometimes you add a loop. I think I wanted one here. And I should have written it over here. That just looks like d comma d. And there you go.

Now, the first part of today is just going to be porting over all the definitions from last class into this model. Most of the things we talked about behave similarly. Some of them behave different in interesting ways. So bear with me while we define a few things. The very first thing is what about degrees? So that's way back in the beginning of graphs. We talked about the degree of a vertex was the number of incident edges. But now a vertex can have two types of incident edges. It can have some incoming edges and it can have some outgoing edges.

So in general, if a vertex v is going to have some incoming guys and some outgoing, we call this the in degree and we call this the out degree. We don't define degree, because it's confusing and ambiguous. So in degree, for example, of a vertex is the-- sorry, not the sum. The cardinality of the set of edges UV that are in e, where u is a vertex. So this is just counting the number of edges going into v and the out degree is symmetric. This would be like v comma u.

Before we only had one type of edge, so there was only one notion. Sometimes these are called degree plus and degree minus or degree sub in, sub out. I can never remember which is plus and which is minus. So I always write in degree and out degree. You can choose your own adventure. A related thing to degree is the handshaking lemma. Who remembers what the handshaking lemma was for undirected graphs? I know it was ages ago, before spring break. No one? Yeah.

STUDENT: The sum of the degrees of the vertices is the number [INAUDIBLE].

ERIK DEMAINE: Right. The sum of the degrees of the vertices is twice the number of edges. I think you said over two, but close. So in this world, this was, in some sense, because every edge was getting counted twice. If you sum the degrees of this vertex and of this vertex, you count this edge once from this side and once from this side.

Now we have distinguished in some sense each end of each edge. There's the in edge and the out edge. Or I guess you call this in edge and this is the-- it comes out of b and goes into d. So if we write the sum of, say, in degrees, so this was just remembering undirected. In the directed case, we have a sum over all vertices of in degree equals exactly the number of edges, and also equals the sum of the out degrees.

Because now our edges are directed. I have an arrow. And so if we sum in degrees, we will only count this edge from this side when we're looking at this vertex. If we sum the out degrees, we'll only count this edge once from this vertex. So these two sums are the same and they both give you the number of edges. So in some sense, a little cleaner than what we had from undirected graphs, because we split each edge or we distinguished the two sides of the edge, I should say.

All right. Let's catch up to last lecture, which is all about walks. So this definition is actually it's pretty much exactly the same. I'll just repeat it so we remember. We had a sequence of vertices where each vi to vi plus 1 is an edge. But before we had curly braces here saying the order didn't matter because edges were undirected. Now we have parentheses saying this is an ordered pair. You have to go forward along each edge. So that's a crucial distinction. Must go forward along the edge. In other words, you have to respect the arrows.

So for example, in this graph, we could start at a and go to c and then go to e. That would be an example of not just a walk, but a path which has no repeated vertices or edges. Or we could go from a to b to a to b to a to b to d to d to d to d to e. That would be an example of a walk. A little more exciting.

I should say these structures should look familiar. If you can remember even farther back to lecture four, which was about state machines, these are exactly state machines. I think the only difference is in directed graphs, we usually think about finite graphs. And with state machines, we're often thinking about infinite machines. But this is the same setup. This definition mirrors exactly state machines and walks.

The only difference, another difference is state machine had an initial state. There's no notion of initial state here. But a walk is just like an execution in a state machine. So unifying all that terminology. We didn't have a notion of path or anything. We just had walks.

As before, a walk like this is from v0 to vk. And k is the length. Length always counts the number of edges, not vertices. And then we had the closed notion. There was a closed walk. And in particular, a cycle. Closed meant the v0 equals vk. And cycle was essentially the closed version of a path. So this was closed walk of length greater than 0 with no other repeated vertices or edges other than the first equaling the last.

The difference is in an undirected graph, the shortest cycle had length 3. We could make a cycle like this, a triangle. We weren't allowed to make a cycle with a single edge, because then you'd necessarily repeat that edge. Where's my red? In an undirected graph, this was forbidden. But in a directed graph, it's possible. If we have the edge uv and we have the edge vu, then there is a two cycle like that. It goes uvu.

There's even a one cycle, because we can have loops. So here, for example. d to d is a length one cycle in this notation. So a little different than what we had before. A little more general. But it turns out this notion of cycle is still interesting for directed graphs. All right, let's move on to connectivity.

Before we had a nice notion of u and v being connected to each other, and it was symmetric. u is connected to v if and only if v is connected to u. Now it's no longer symmetric and we're going to use a different term for it to make that clear. So we'll say v is reachable from u or u can reach v if there exists a walk from u to v.

So this looks like the same definition of uv connected in the undirected case. But now because walks can't be reversed, because if you reverse a walk, if I go from vk back to v0, then I'm using all the edges in the opposite direction. And that's not in general allowed. Unless you happen to have a graph where every edge is doubled, you won't be able to reverse your walk.

So this is no longer equivalent to their existing a walk from v to u, but it is still equivalent to the existing path from v to u. From u to v, sorry. That part is still true. The proof is actually exactly the same. You take a shortest walk from u to v. And if it's not a path, then it has some loop in it, some cycle essentially. And then you emit that cycle and you get a shorter walk contradiction. So exactly the same proof as before. Still respects the edge directions, which is nice.

So for the purpose of reachability now, walks and paths are the same. So we can assume whichever one is more convenient. This relation is still this reachability property is still reflexive and transitive. Remind you what those mean. Reflexive means v can reach v for any vertex v. You can always get to yourself. You just don't do anything. And it's still transitive.

Meaning if you can get from a to b and b can reach-- if a can reach b and b can reach c, then a can reach c. You can still concatenate paths together and that's fine. But no longer symmetric. I've already said this a couple times, but if a can reach b, we don't necessarily have b can reach a. So that makes things more interesting.

In particular, what does it mean for the whole graph to be connected? For that, it's useful to tal about. There's a very strong notion of connectivity. First let's say two vertices u and v are strongly connected in a directed graph if they're mutually reachable. In other words, u can reach v and v can reach u.

This behaves symmetrically. If u and v are strongly connected, then v and u are also strongly connected by definition. u can reach v and v can reach u. And in particular, we call the entire graph strongly connected if everything can reach everything. If for all vertices u and v, u can reach v. Or in other words, for all u and v, u and v are strongly connected. So again, the pairwise relation.

These graphs are nice, because you can start anywhere and get anywhere else. You never get stuck. But in general, in a directed graph, you could walk and not be able to return. So for example, in this graph, maybe you walk around. You take one of the walks that I drew, but once you end up in e, you're stuck. Or if you start in e, you can't get anywhere except e itself. Start in f, you also can't go anywhere, but that's a little less exciting. So that graph obviously is not strongly connected. Maybe I'll draw an example of one that is.

OK, this graph is strongly connected. It's also Eulerian. So remember an Euler tour was a closed walk. It visits every edge exactly once. And visits every vertex at least once. So this graph is Eulerian. I drew it in an Eulerian way, but I will also redraw a path here. You can go around the outside. Notice now I'm respecting the edge directions. That's a little different from the example we saw last class. I think I got all the edges exactly once.

So again, you might ask which graphs are Eulerian. We did the really long proof last class. And it turns out there's a very similar characterization. So Euler tour exists if and only if your graph is strongly connected. And for every vertex, the in degree equals the out degree.

And there's a similar characterization for Euler walks instead of Euler tours. It's just a little uglier, so I won't go into it. It's just you allow these to differ by 1 for a couple of vertices in a particular way.

So in particular, if you look at this example, it should be every vertex either has two in, two out. This one has two coming in, two going out, or one in, one out. And in general, if you have k incoming edges, you should have also k outgoing edges in order to have an Euler tour.

Why? Because what goes in must come out. All the proofs here are similar to what we did for undirected graphs. Before we were saying the number of edges at each vertex had to be even, because every time we enter the vertex we also had to leave. But this is the directed version of that where every time you come in, you must also exit if you have a closed loop. If you have a closed walk that visits all these edges.

So certainly this is necessary. Strong connectivity is also necessary because the Euler tour gives you a way to get from anywhere to anywhere. It's like a giant subway line. You can get on whenever you want, and follow it until you get to whatever vertex you want. Because we said the Euler tour visits every vertex, that's a way to get from anywhere to anywhere. It might take a while, but you'll get there.

And what else? Yeah, the proof again is similar. If you want to show that these properties are sufficient, you take the longest possible trail that visits every edge at most once and argue that it's closed for the same reasons as before, and that it actually has to visit every edge, because otherwise you could make it longer.

I think we're pretty much through the main repetition from before. There's one more related notion to connectivity, which is connected components. We talked about connected components a little bit for undirected graphs. They're a little more interesting for directed graphs. In particular, they mainly make sense for strong connectivity. We don't define any other types. So it's not surprising. So strong connected component or SCC of a vertex v is the graph induced. This is similar to how we set it up before. By all the vertices that are strongly connected to v.

Let's look at an example. This example is a little bit interesting. So for example, what is the strongly connected component of a? I haven't made this too ugly. Redraw this a little bit. What vertices are strongly connected to a? Shout them out. b? I can definitely get to b and b can get back to a. c. I can get from a-- oh no, I can get from a to c this way and c can get to a via b. Anything else? No.

Turns out, if you follow any of these edges, any of these two edges, you're never getting back. And also f is neither reachable, is not reachable from a in either direction. So this chunk here is a strongly connected component. And again, when I say the graph induced by these three vertices, so this is the graph induced by ABC, I mean, that's my vertex set. I take those three vertices, none of the others.

And I take all the edges that were in my original graph among those three vertices, and throw away everything outside. So that boxed region is the strongly connected component of a. It's also the strongly connected component of b and the strongly connected component of c. What's the strongly connected component of d? Just d. So this is in a strongly connected component by itself. And e similarly and f similarly.

In general, we get a bunch of strongly connected components. And before it was very clean. Anything you could possibly touch was in one strongly connected-- was in one connected component. And every vertex and every edge fell into one connected component. That was the situation for undirected graphs. We said that connected components partitioned the graph. Everything appeared exactly once in one of the pieces.

Now it's not so clean. The vertices get partitioned. There's some vertices in this component, some in this, some in this, some in this. Every vertex is in exactly one strongly connected component. But for directed graphs, there are also some edges between components. That's a little more interesting. So let's say every vertex is in exactly one strongly connected component. But we have some edges between.

And there's a nice way to organize this, which we call a condensation graph. A condensation graph of a graph g is another graph h, vertex set c, and edge set f. These are just some made up letters where c is-- I forgot some notation. So the strongly connected component of vertex v I'm going to call square bracket v probably just for this lecture. But it's convenient to have some name for the strongly connected component that contains v. And c is going to be the set of all strongly connected components. So now I can write it as square bracket v for all vertices v and v.

So in this example, I draw the graph over here. So this was g. h is going to have one vertex for this strongly connected component, one for this one, one for this one, one for this one. So four vertices. I'm going to label them with the vertices in the original graph that they came from. Got d up here, e roughly over here, and f over here.

Now, what are the edges? Well, the edges are just going to correspond to whenever-- I want to read this from right to left. So let me write it first. So whenever I have an edge uv in the original graph, I want to draw an edge between the corresponding strongly connected components, the strongly connected component containing u to the strongly connected component containing v, except I don't want edges from a strongly connected component to itself, just because it's cleaner that way.

So this will represent exactly the edges between components. There are some edges within components which I kind of want to ignore, because I can get anywhere within this component. So from a connectivity standpoint, from a reachability standpoint, I don't really care about the distinction between a, b, and c. They're all essentially the same. If I'm at any of them, I can go to any other.

But then there's these edges between components that are interesting. Highlight them here in bold. So we've got a, b c, to d. We've got a, b, c, to e. We've got d to e and that's it. So this is the condensation graph. And so if you want to know reachability between two vertices in the original graph, you can figure it out by looking at reachability within h. So if everything's strongly connected, you're in one big component, that's easy. The answer is always yes. Otherwise you have to look at walks in the condensation graph.

Next topic. And actually the last topic, which we'll spend a bunch of time on, it's much more interesting than last time, is acyclic graphs. So what would you call an undirected acyclic graph? Anyone remember? I forget if we mentioned it.

Tree is what you were going to say. Actually, trees are connected undirected acyclic graphs. So almost. So we may not have even mentioned it here, but we use the word forest for the general case when you're not necessarily connected. So in the undirected case, a forest is just any acyclic graph. Each of its connected components is a tree, but it might have more than one.

Today we're going to talk about directed acyclic graphs. I feel like there should be some nature based pun for directed acyclic graphs, but instead they're just commonly called DAGs for Directed Acyclic Graph. So it's kind of boring. Sometimes I've seen DAWG, like D-A-W-G. That's another type of graph, which we won't get into. So that's nature based, I guess.

All right. So what's an example of a directed acyclic graph? Have we seen any in this class? Yeah.

STUDENT: [INAUDIBLE]

ERIK DEMAINE: A terminating state machine. Yeah. Maybe not all terminating state machines, but the ones that we-- the method we use to prove them. So a state machine. Yeah, maybe all terminating state machines, actually. But in particular, those with strictly decreasing derived variable. If it's strictly decreasing, then you can't make any cycles. I think even conversely, if you made a cycle, then that would be-- repeating that cycle over and over would be a non-terminating execution. So indeed. Or terminating state machines.

Good. Another one is this graph, the condensation graph we just defined. Maybe not obvious. These are just examples. Of course, there are many directed acyclic graphs, but there's no definition here. I mean, the definition is directed acyclic graph. Or I guess you might say acyclic digraph, given the terms we've used. But it's always written DAG. So there you go.

Condensation graph is also always acyclic. Because if there was a cycle like if I could get from a, b, c to d to e back to a, b, c, well then these three should actually all be in one strongly connected component. So there can't be any cycles in this graph. Otherwise I would have collapsed further. So there we go. What makes the condensation graph interesting in some sense is it becomes acyclic. You contract all the cycles into one vertex or each cycle into one vertex.

I have a, quote, "practical" example here. You see why I'm laughing in a second. So directed acyclic graphs come up naturally a lot. I mean, you can make any graph acyclic by constructing the condensation graph, but where they're quite common is in a set of constraints. Suppose you have a bunch of tasks you need to do, and there's some precedence constraints. This has to happen before that. This has to happen before that. You could imagine project management or whatever. I'm going to give a very real world example you do every single day, I hope, which is getting dressed.

So suppose you have some socks you want to put on, and you may know that socks come before shoes. I've tried it the other way. It doesn't work so well. And usually I put one sock on at a time. So there's your left sock and your right sock. Apologies for people who have a different number of feet, like dogs. It's even more annoying.

But these are examples of precedence relations. You want to put the left sock on before you put your left shoe on. You want to put your right sock on before you put your right shoe on. But you don't really care about the ordering between these two sides. I could do both of these and then do both of these. Or I could do both socks and then both shoes. I could do left, right, right, left. There are lots of possible orderings that are valid here.

You can add more fun things like pants. Those have to happen before your shoes. And I don't know, you probably want to put a shirt on. And maybe you have a jacket on top of that, and you have a belt, which you should put on after your pants, let's say. And maybe a winter coat. It's still kind of winter maybe. We'll see.

And if you're really cold, maybe you want a scarf. And I've heard debates about which of these goes first, but let's say scarf before coat. Certainly after your jacket. And maybe you want a hat on, but that could happen pretty much any time after your jacket. So something like this. I think those are all the annotations I had here.

And now you'd like to find a valid ordering. If you want to find a valid ordering, there better not be any cycles here, because then you could never resolve that cycle. If you have a cycle of precedence constraints and a has to happen before b, b has to happen before c, and c has to happen before a, then none of a, b, and c could happen first. So that would be bad.

So in any kind of precedence constraint graph like this, you want to have no cycles. So this is practical motivation for directed acyclic graphs. And now we'd like to-- let's define a notion of topological ordering. The DAG is an ordering of all the vertices such that each vertex appears before all vertices it can reach. I guess I gave it a name, so I'll call it v here.

So for example, pants has to happen before belt, but also pants has to happen before your winter coat because of these two connections. And pants has to happen before shoes. So pants should be listed earlier in your sequence before your right shoe, before your left shoe, before your winter coat, before your belt, before-- that's it for pants. And if we can find an ordering that satisfies that property, then that would be a legitimate way to execute these set of tasks.

And the next thing I'd like to do is prove that topological orderings always exist. It seems reasonable in a DAG. As I said before, if you have a cyclic graph, these don't exist. That's a little less exciting. But as long as your graph is acyclic, you can always find a topological order, and there's a nice way to do it. Let me first define a couple of useful terms.

One is source. This is for any directed graph. A source is a vertex of in degree 0. This is going to act kind of like a leaf for us that we had for trees. Remember, directed acyclic graph here are our analogy of trees or forests more generally. And degree 1 vertices. Well, there's no notion of degree here anymore. We just have in degree and out degree. So instead of degree 1 vertices, we're going to look at in degree 0. So for example, left sock is a source, right sock is a source, pants is a source, and shirt is a source in this graph.

And the complementary notion now is sink. And this is a vertex of out degree 0. So in this graph, we have a few of them also. Left shoe is a sink. Right shoe is a sink. I have no outgoing edges. Winter coat is a sink and hat is a sink. So these are sinks. And then there's some vertices that are neither. Probably in general, most of them are neither.

But what we do know, like the fact that every tree with at least two vertices has at least two leaves, we know that every DAG has at least one source and at least one sink. So if you're wondering why two leaves for a tree, this is kind of why. Maybe this is distinguishing the two ends. And the proof is actually very similar. So let's do lemma. Every DAG has a source and a sink.

Proof. I don't know if you remember the proof that every tree has at least two leaves. It was take a longest path or walk. Maybe it's worth mentioning for DAGs, walks and paths are the same thing. Because the thing we were worried about with paths is do you ever cycle? Do you ever repeat a vertex? But if there's no cycles, you can't. So for DAGs, walks equal paths, which is nice. So for the rest of today, I don't have to remember which is which, but I'll try to be a little bit careful if I need something.

So let's just take the longest or a longest path in the DAG. So let's say it starts at v0, v1, and goes up to vk, as usual. Then the claim is v0 is a sink. And vk-- sorry. Backwards v0 is a source and vk is a sink. And so then we found it. OK, why? Let me do a proof by picture. So here is our hypothetical longest path.

And I want to claim, for example, that v0 is a source, meaning there is no incoming edge into v0. Its in degree is 0. Well, let's prove it by contradiction. Suppose there was some vertex here u and there was an edge from u to v0. This is for contradiction. Imagine that this happens. Well, then that looks like a longer path in the graph. How could it not be a longer path? Well, maybe it's cycles. Maybe this vertex u was actually one of these other vi's.

But it can't be, because our graph is acyclic. So either we get a cycle, which contradicts that we're a DAG. Or we get a longer path in the graph, which contradicts that we were longest. So either way, we get a contradiction. And that proves that v0 is a source and exactly symmetric argument proves that vk is a sink. Cool. So these things exist.

Now let's use them to construct an order to getting dressed. I know you've been wondering the answer to this gripping question. How should I get dressed every morning? Finally, in 61200 I learned how to do it for any graph.

The idea algorithmically is very simple. If you want a topological order for your graph, what should I start with? A source seems good. Source has nothing that has to come before it. So any one of these would be a valid first step. So the algorithm is just take any source, do that first, then conceptually remove that vertex to represent that it's been finished and then find another source. And because from this lemma there's always a source, you never get stuck. So you can just keep going. And eventually, you'll have executed every task.

So let's do it briefly on this example. Say we do right sock first. And we see if there are any new sources. I think there are not. This is still not a source because it has an incoming edge. One left. Then maybe we do our pants, put those on. Seems like a good idea. Now we can put on our right shoe finally. I've been waiting for that right shoe. Because there's no more incoming edges, because we've eliminated, we've finished the tasks from before. So if I write down the ordering, it would be right sock, then pants. Now I'm going to do right shoe. Then I don't know. I'm lazy. Let's do left sock and left shoe next.

Then let's see. At this point, I've eliminated the left half of the graph. The only source is shirt. So I have to do shirt next. I had a lot of choices up till now, but that choice is forced. After I've eliminated these two, this becomes a source and this becomes a source. So I can choose belt or jacket first. Let's do belt. Still can't do winter coat, because I need to do scarf before it. So jacket is the only source at this point. So I'll do that. I think you get the idea. Now I could do hat, for example, both a source and a sink. I can do scarf and then finally, I can do winter coat. So that was a valid execution order or a valid topological ordering. Cool.

Let's prove that this always works a little more carefully using induction instead of an algorithm, because this is more of a math class than a algorithms class. Though of course, we talk about both. But I think it's good to get more induction practice.

So in particular, induction with graphs is still a pretty new thing. But again, induction about graphs is always induction or some regular induction over natural numbers. So we're going to do induction on the number of vertices in the graph, which is how we normally do it. So our p of n is that every DAG with n vertices has a topological order. I'm going to assume by induction that all smaller n have this property and prove it for n, or prove it for v, size of v. So what is the smallest that size of v can be?

Well, I didn't put it in the definition, but I meant to say v has to be at least 1, as before. We don't allow the empty graph with no vertices. You have to have at least one vertex. So the smallest graph has one vertex. So here it is. I guess, yeah, I mean, in a directed graph, you could have a loop here. But that's a cycle. So we're not allowed. In a DAG, there's only one graph of one vertex. It's that one. Let's call this a. The ordering is a. It satisfies all the properties. That's their topological order. So this is easy.

More interesting is the induction step. So now we can assume that v is at least two, bigger than the base case. So what should we do? Well, we have this nice lemma that tells us there's a source. So whereas the algorithm took a source and then had to do more work to continue, we can just take one source and then use induction. So by the lemma, there exists a source. Let's call it s. And the intuition is I want to do s first.

But to figure out how to do the rest of the graph, I can use induction. So let's let g prime be g minus s. Remember, this was the graph-- g minus s was the graph induced by the subset of vertices that's everything except s. So in other words, erase the vertex s and all of its incident edges, which in this case will only be outgoing edges. So here, we have s. And there's some outgoing edges into g prime and g is this whole graph.

So the point is number of vertices in g prime is 1 less than the number of vertices in g. We just removed s. Everyone else is there. But in particular, g prime is smaller than g was, so we can use induction. By the induction hypothesis, g prime satisfies this. I guess we also need to check g prime is a DAG. But if you remove a vertex, you're not going to add any cycles. So that's cool.

So g prime is a smaller DAG according to the measure of number of vertices. So by induction hypothesis, it has a topological order. g prime has a topological order. Let's call it v1, v2, up to v sub I guess number of vertices minus 1. That's how many vertices there are. So by induction, I get an explicit ordering of all those vertices. And now I just put s first before all of those. So we get s comma v1, v2 and so on. v sub v minus 1. This is a topological order of g. And this is where I guess you have to actually check something.

So I claim it's OK to put s before all these other vertices, and that will be a valid ordering. Intuitively, because s has nothing that has to happen before it. So if you check the constraint of a topological ordering, every vertex listed must appear before all vertices that it can reach. Well, s definitely appears before all vertices it can reach, because it appears before all vertices. And the other vertices are also OK. Like v1, let's say, appears before all the vertices it can reach, because all the vertices it can reach, it's inside g prime.

Once you're inside g prime, you can't escape. You can't get back to s. It's in a different strongly connected component. Any vertex down here, which is all of the v1 through v minus 1, cannot reach s because it only has outgoing edges. And so they also appear before all the vertices they can reach, because all the vertices they can reach are within g prime. So great.

Now, given a set of tasks with precedence constraints, you can find a valid ordering. So you can get dressed in the morning. But most of us are computer scientists. And so often we're not just one computer or one agent executing a bunch of tasks. But maybe we have many computers to execute our tasks.

What if we allow parallelism? Can we do this any faster? You imagine you want to get dressed, but you have a crazy Rube Goldberg style machine that can put on many garments all at once. So maybe then you can do left and right sock in parallel. If I had four arms, I would do that. Maybe even three arms, I don't know, that's a good question.

So let's think about that version where we can do parallel task execution. Then I'd like to do it as fast as possible. If I'm just a measly single processor and I want to execute the tasks in this graph, well, the number of steps it takes me is exactly the number of vertices, no matter how I do it. It doesn't really matter which ordering I choose in terms of speed. Assuming, I don't know, assuming instantaneous switching time between different tasks. But with parallel task scheduling, you can do more.

So let me first define the problem a little bit, and then we'll prove a nice theorem. So I want to define a schedule to say basically what order you should do the tasks in. But now I'm allowing multiple tasks to be scheduled at the same time. So a schedule is going to assign what time t of v, it's going to be a natural number, to each vertex v. And it needs to satisfy the property that if u can reach v and u does not equal v, then the time that we schedule u has to be strictly less than the time we schedule v.

And this is the analog of topological ordering. Except now I can assign the same time to multiple tasks. So maybe I do a bunch of tasks at time 0, as long as none of them has to happen before another. Then I could do a bunch of tasks at time 1, a bunch of tasks at time 2, and so on. And I'll call it the span of the schedule is the number of distinct times t of v.

So I can look at all the different times assigned to the vertices. And if I use 17 different times, then my span is 17. Span is a measure of how long it takes to execute the schedule. Here I'm assuming an unlimited number of processors. I can do all the tasks at once if I need if I'm allowed to. If I have no edges in my graph, I'll just do everything in time 0 and I'll be done. And that will be a span of 1, because there's only one time value that I needed.

But in general, we're going to need more. I can't do left sock and left shoe at the same time, because left sock has to happen strictly before left shoe. That's what this says. If one vertex can reach another vertex, that's different than the time assigned to the earlier precedent has to be strictly less than the later one. So that's the definition of the problem. And now that's a nice theorem. It says the minimum possible span of any schedule.

So certainly a schedule like this exists, right? We could do the one given by the topological order. We could do right sock in time 0, pants in time 1, right shoe in time 2. We could do this sequentially, one task at a time. But if I want to minimize the span, I want to do lots of things in parallel. This is equal to the number of vertices in a longest path in the graph. So a very simple characterization of how long this takes.

In this example, I think the longest path is shirt to jacket to scarf to coat. That's a path of length 1, 2, 3. But what I wrote there is not the length of the path. I wrote the number of vertices in the path, and that's 4. So this is the one time when we want to count vertices, not edges. So I write number of vertices in the longest path with shirt, jacket, scarf, winter coat. So that's four. And so I claim this set of tasks can be executed in just four steps. Four rounds of parallel execution. Namely, I think I need another color. Let me see what I brought. Oh, purple.

So how I do it is actually pretty intuitive. I just want to take all the sources at each step. So in the beginning, I had four sources. I'm going to do all of those at time 0. Then what's the source? This becomes a source because both left sock and pants were done in the previous step. This becomes a source because right sock and pants were done in the previous step. This becomes a source and this becomes a source. I think that's it.

So then in the second step, I'm going to do all these. Everything above that purple line. And then hat becomes a source. Scarf becomes a source. But winter coat is still being annoying. So we do everything above that line. And then in the fourth round, we just do winter coat. So that's the algorithm. But let's prove. It's not obvious that this algorithm takes exactly the longest path time. And there's a nice, elegant proof of it. So let's go back here.

So in general, when we want to prove that two numbers are equal, usually we do it in two parts. We prove that one number is greater than or equal to the other, and one number is less than or equal to the other. We've seen that a couple of times, I think. And this is definitely one of those situations.

So first let's think about why is the minimum span of a schedule at least the number of vertices in the longest path? Well, that makes sense, because if I just think about this longest path, it's some sequence of vertices. And let's say it has length l. So we start at number one or it has l different vertices in it.

Well, in the first round, just looking at this path, the first round, the only thing I can do is the first vertex in the path. In the second round, the only thing I can do is the second one. To execute this path clearly requires at least l time steps. So this is just to schedule the longest path. You certainly need the number of vertices of that path. That's a bit hand-wavy, but pretty intuitive.

The interesting part is the other direction. So let's suppose I found the longest path and it has l vertices. I claim that I can achieve, I can find a schedule that does that, that is that fast. And here it is. Given a vertex, we're going to define its depth to be the length of the longest path ending at v. Or I'll say I guess 2v. So it can start anywhere. But I want the longest path that ends at v. And I'm going to-- here I'll use length measuring the number of edges, as usual, because it doesn't really matter.

So I claim that depth is a schedule and that it has the right span. Because depth assigns a natural number 0, 1, 2, whatever to every vertex, and I claim that it is a valid schedule. So let's prove that.

So what does it mean to be a valid schedule? It says that if I have two vertices u and v, where u can reach v and u is different from v, then I have to prove that depth of u is less than depth of v. So suppose u can reach v and suppose v does not equal u. We want to show depth of u is strictly less than depth of v. This is what it means to be a valid schedule.

What does it mean that u can reach v? Probably still have the definition over here. v is reachable from u or you can reach v if there exists a walk from u to v. Or for DAGs it's the same, in general, it's the same as there exists a path from u to v. So we know that there exists a path, let's call it puv, from u to v. That's the meaning of u can reach v.

And we also know that it has length greater than 0. Why? Because these vertices are different and there are no cycles. I guess the only way to have a length 0 path is if you start and end at the same vertex. But if v is different from u, that path has to have a non-zero length. This will be important in a moment.

OK, let's see, what else do we know? We also know or we know that there's a path. There's a longest path. pu to u. My words are starting to sound like numbers. This tou. That's the definition of depth. Depth is the length of the longest path that ends at that vertex. So let's just take that longest path. So this one has length exactly depth of u. That's the definition of depth. It might be 0. Maybe the longest path that ends at u, if u is the source, it's just the path that starts at u and doesn't do anything. But there is this path.

All right. Now, my big idea is concatenate these two paths. Because here we have a path that ends at u. And here, we have a path that starts at u and goes to v. So obvious thing to do is glue them together. So let's concatenate. And I'll call this p sub v is going to be p sub u comma p sub uv. So follow the path pu. Follow the path puv.

And the length of this path, so first of all, it's a path because this one ended at u and this one starts at u. So you can actually concatenate them. The length is just going to be the sum of these two lengths. So it starts with a depth of u number of edges. And then it has at least one more. So the length is strictly greater than depth of u, because it's just the sum of these two numbers.

Now, what do I know about this path? It is a path that ends at v. pv ends at v. Because if you look at the last vertex it visits, it's v. So this is a path that ends at v and its length is this, greater than depth of u.

Now, depth of v is the length of the longest path that ends at v. And the longest path, is it greater than or equal to any particular path? So I found a path here. That means that the longest path is greater than or equal to this one. And that's strictly greater than depth of u. And the length of the longest path here is equal to depth of v.

So I've proved that depth of v is strictly greater than depth of u. Because I found a path that was longer than depth of u. Therefore, depth of v is greater than depth of u. So that's cool.

There's one last thing to check here. So I proved that this is a schedule, that it processes things in a valid order. But we also need to check how many time steps does it take? Why is it the number of vertices in the longest path and not the number of edges, for example. The span of depth I claim as 1 plus the number of-- 1 plus the length of longest path.

This is what I wanted to prove, because length counts edges. So if I add one more, that's the number of vertices in the path. And that's exactly what this theorem says. Why? Because if you look at all the possible depths, they start at 0. All the sources are going to get depth 0. And they go 1, 2, and so on. And I claim they end at the length of the longest path.

Why? Because what are the depths? The depths are the length of the longest path ending at a particular vertex. And we do this for all the vertices. And so the maximum value this is going to take on is just whatever the globally longest path is, it ends anywhere. So that's why we max out here. And it's plus 1 because we started at 0 and we end it here. So we have 1 through length of longest path. That's length of longest path. But then we have a plus 1 for 0.

So that proves this theorem and tells us how to optimally schedule if we have an infinite number of processors or not infinite, but enough that we could do all the tasks at once if we wanted to. Cool. Let me recast these results in a slightly different form that's interesting and will connect to the next lecture.

Let me give you a couple of terms that are concept called partial order, which we'll talk about next time. And they let us think about these kinds of precedence relations in a different way. So in particular, there's this idea that, OK, left sock and left shoe, they're related. Left sock has to happen before left shoe. Right shoe and pants are also related. Right shoe has to happen after pants. Also winter coat has to happen after pants.

But for example, pants and shirt are unrelated. They can happen in either order. There's no path from pants to shirt. There's no path from shirt to pants. So let's define a notion of comparable, meaning there is a precedence relation between them in one direction or the other. So either u is reachable from v or v is reachable from u. In either case, we call u and v comparable. And otherwise, they're not comparable. These are vertices.

All right. So there's two complementary notions I want to define. A chain is a set of pairwise or mutually comparable vertices. And an anti-chain is a set of pairwise incomparable vertices.

OK. So we're thinking about sets of vertices, subsets of the whole graph, that are either all mutually related, they have a defined sequence for them, or not. So for example, where's my purple? Need even more colors. Hopefully this will still be clear.

So for example, if I take shirt and jacket and winter coat, these are three vertices and I claim they form a chain because these two are related, these two are related, these two are related. All pairs of them are related. And in fact, they have a fixed order. Shirt has to happen before jacket. Jacket has to happen before winter coat. I happen to skip scarf, but I could also put scarf in. That would be a longer chain.

Chains basically correspond to paths, except you can throw some of the vertices away. So that's why I left a blank line here. This is basically a subset of a path. Every chain is going to be a subset of a path.

An anti-chain, though, is the interesting new notion here. That's going to be a set of vertices that are mutually incomparable. For example, all the sources. By definition, you can't reach any of them for any other, because if you start at a source, you can't get to any-- sorry, if you start outside, you can never get to a source. That's what I want to say. Unless you started at that exact source. So all the circled red vertices for the sources are an anti-chain. So are all the squared ones at the bottom. Those are the sinks. There are four of them. Those form an anti-chain.

Now, I claim anti-chains are really what's going on in the scheduling problem. In the sense that all the tasks that are done at the same time must be an anti-chain. So if you look at all the things that happen at time 0, those are the sources. Those have to be an anti-chain. All the things done at time 53 also have to be an anti-chain.

Because if there's any relation between them, if one of them had to proceed another, if there was a path between two of the vertices done at time i, then that would be invalid. You're not allowed to do that. If u and v have a reachability relationship, they must be done at different times. So the things you do at a particular time form an anti-chain. And so by that exact theorem, we get this corollary. I should say fewest.

Another way to look at a schedule is it's partitioning the vertices. The vertices are all the tasks I have to do. And into the fewest number of anti-chains. Anti-chain is a set of vertices I can do at the same time. So the number of steps I need, the span of that schedule, is exactly the number of anti-chains I need to cover all the vertices, to execute all the tasks. The theorem is this has the same size as the size of the partition. The number of anti-chains you need is the longest chain.

The proof is exactly you apply the schedule and then your anti-chains are all the things I do at time 0. That's one anti-chain. All the things I do at time 1, that's another anti-chain. And so this is kind of a neat connection between-- these are kind of dual notions. One, you want lots of comparisons possible. And the anti-chain, you want no comparisons possible.

And this is saying that they're related in that if I try to split into anti-chains, the number of anti-chains I need to cover everything is equal to the longest chain. It turns out the reverse is also true. If I want to partition into the fewest chains, the number of chains I need is the same as the longest anti-chain. It's kind of cool. I guess this should technically be largest, because it's a set.

What else do I want to say? You can use this to prove some fun facts like there's always a large chain or a large anti-chain. So for example, for any t, either there exists a chain of length or of size greater than t or there exists an anti-chain of size at least n over t or I guess size of vertices over t.

This just follows from that thinking about the partitions. If you know that all the chains are small, suppose all the chains have size at most t, then you know that the partition into anti-chains, or sorry, one way or the other. If I know all the chains are small, then I know I can partition into anti-chains with that number. So I can partition into t anti-chains if all the chains have length at most t. And that means that the longest anti-chain has to be at least v over t, because all the vertices get covered by the partition. And so the biggest one is going to be bigger than the average.

So you can use it to do that. You can use it to do other fun things. Like have you ever wondered-- I don't know what your calendar looks like, but for me, I have lots of conflicting events. I have to be in many places at the same time. It's pretty annoying. And partly because I have many calendars all displayed in the same place. And so have you ever wondered, how do you display a calendar like this?

Well, for one event that occurs strictly before another, I'll draw an edge. So here's an edge there. There's an edge there. Some of these have edges, some don't. And then in particular, if I look at two conflicting events, they will be incomparable according to this definition. There will be no path from this vertex to this vertex or this vertex to that vertex.

And so what I'm looking at here is a partition into chains. Each of these chains has to be nicely ordered. And I want to know how many different chains do I need? Well, it turns out the answer is exactly the size of the largest anti-chain. In other words, the size of the largest set of events that are pairwise conflicting, which is the best you could hope for. I need at least one column for each of these guys if they're all conflicting with each other. And it turns out you can always achieve that. And that's how calendars draw their events. All right, that's it.