PROFESSOR: You guys have been talking about graphs in lecture, right? So what are graphs? So graphs are a kind of formalism that have vertices and edges. A set of vertices is-- you can think of it-- it's like a set of things. And then the edges are the relationships between those things.

So the set of all your friendships and your friends' friendships could be considered a graph. So if this is me and this is all two of my friends, then an edge between us would indicate a relationship of friendship. But there's no edge here, so there's no friendship between these two.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes. So you have vertices, which are also called nodes, and then you have edges, and they could also be called arcs. So if you see any of these names, these two are the same, and these two are the same. So what kinds of graphs do we have?

AUDIENCE: [INAUDIBLE]

PROFESSOR: So there's a directed graph, and there's an undirected graph. So up on the screen here, is a directed graph, right? And so-- I'm not a sports fan.

I think those are the Bruins, and I don't know what the other team is. Do you know? OK. Yes, I think it's a hockey team somewhere up in Montreal.

So this is a representation or a graph representation of cities, which are the nodes and vertices and then the roads that connect the cities. And, obviously, it's not to scale or accurate, but it's an abstraction, so we're OK with that. And the question here is what's the path to get from Boston to Montreal. So in this case, it's a directed

graph, so that means that we can only go this direction from node to node, in the direction of the arrow. So, really, there's only one way to go, right?-- two hops.

So that's a directed graph, and then an undirected graph is basically the same thing, except we can go either direction on the edges, right? So on the directed graph in the previous slide, you could only go in this direction, so you had to make two hops in order to get to Montreal. In this undirected graph, you can just make one hop to get to Montreal-- so really conceptually very easy.

AUDIENCE: Can you have a directed graph in a case where there's one going from Boston to New York and another one going from New York to Boston?

PROFESSOR: Yes. So-- and actually we'll see that in the code. So the question was, can I have New York here and then Boston here? Can I have this sort of relationship? And the answer is yes. This is actually just equivalent to an undirected graph.

AUDIENCE: Probably you can have a directed graph in some cases like that.

PROFESSOR: Yes. So you could have-- and we actually have-- actually, I think this might have an example of that, so, yes. So in this case, you have-- Hartford has a path to Albany and back in a path to New York City, but there's no path directly back to Hartford. That work for you?

AUDIENCE: [INAUDIBLE]

PROFESSOR: What's that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Well, then I won't insult Hartford then. I didn't do these graphs. If I had done them, I certainly wouldn't have used sports teams because I know next to nothing about sports. So I don't know the rationale behind picking the names, other than I'm supposed to say that the reason why the Bruins want to go to Montreal is because they want to kick Canada's butt.

AUDIENCE: They did.

PROFESSOR: Did they?

AUDIENCE: They already have.

PROFESSOR: Oh. Well, then this is a very apropos slide then. So what we have up here is a weighted graph, right? So the undirected and directed graphs have been really easy so far because they're just defining the fact that there is a relationship that exists between two vertices, right? So what a weighted graph does, though, is it says that not only is there a relationship between these two entities-- these vertices-- but it also has maybe a cost associated with it.

So if these represent the road networks, then these represent, kind of, the total costs-- the weights on these edges. So in saying that in order to get the Hartford, I need to pay $1.00; and then to get to Albany, I need to pay $3.00; and then to get to Montreal, to pay $6.00. So a common question-- on weighted graphs in general-- is what's the least cost path from here to here. So can anyone tell me?

AUDIENCE: From Boston to Montreal?

PROFESSOR: Yes-- what the least cost is from Boston to Montreal.

AUDIENCE: It's-- Hartford is your only choice.

PROFESSOR: Right.

AUDIENCE: Then New York and then Hartford.

PROFESSOR: Right. So-- and the cost for that is $9.00, right?-- because you got $1.00, $7.00, $1.00-- sum $9.00. There are actually two paths to get from Boston to Montreal. The other path is from Boston, Hartford, Albany, Montreal, but the cost of that path is $10.00. So the question is how do you figure out which path is shortest, right?

So did he talk about breadth-first search and depth-first search at all?

AUDIENCE: He only talked about depth-first.

PROFESSOR: He only talked about depth-first. OK. So we need to do breadth-first. So before we do breadth-first, can someone define depth-first for me and maybe walk me through it a little bit?

So let's take this off the screen, and let's assume that I have a very simple graph. I'm going to start here. I want to end here. And I have-- So I'm starting at v0, and I want to get to v8.

Let's call this a directed graph, so you can only go one direction. It also doesn't have any cycles, so that makes it a little easier. So if I'm starting here in depth-first search, what do I do?

AUDIENCE: Pick a daughter-- Pick a daughter?

PROFESSOR: So you pick one of your children, so you're going to pick v1 or v4. Now, you go to this node. What do you do now?

AUDIENCE: Pick a daughter.

PROFESSOR: Same thing again. So you pick another daughter and because this is a really trivial example, we just walk down the line until you find the node that you're looking for. Or if you don't find it, and you have no more-- there are no more children to look at, then there's no path that exists. But we can get to our goal node here, right? So what do we do once we find this node?

AUDIENCE: [INAUDIBLE]

PROFESSOR: We save it off somewhere. It becomes a path with just itself, and then we return that to who-- And then we say, OK, well, where did I come from? Well, I came from v7. So, now, I know that my shortest path from v7 to v8 is going to be v7-v8.

And then I'm back here, I'm going to add wherever I came from here. So the idea is that I grow my shortest paths backwards. Right? And, actually, the shortest path is the top one here, so--

AUDIENCE: If you hit a branch in v7, can you go back to v7 would you jump to the next branch?

PROFESSOR: So, yes. So if I had something like this, and the answer would be-- let's say that it loops around-- so now we have a cycle. So it becomes interesting. So let's say that I've reached my goal here, and I know that if I'm at my goal, then my shortest path is just my goal, right?

So now I return here and I say, well, what would be my shortest path in this case? Does this child have the shortest path? When I get here first to v7, I asked the question, what's the shortest path to v8 from either of my two children.

So I'm going to look at all the children of v7, and I'm going to find what's the shortest path from v8-- or from this node to the end node-- and then from this node to the end node. And, obviously, this one's the shortest because it is the end node. So now, I know that my shortest path is this plus myself. And so that means that the shortest path from v6 to the end node is going to be this path, plus this. And I don't know if that's getting any clearer.

So, really, if we start out at the beginning here, we're looking at this first node-- we ask the question, what's the shortest path from v1 to the end, and what's the shortest path from v4 to the end? And we choose the shortest of those two paths as our answer, then we just add ourselves to the beginning. And that's all we do for each of these nodes. We ask, of the children at each of these nodes, what's the shortest path, and then we add ourselves to the beginning of that path and return that as our answer.

So if we are to look at that in code-- so you guys have all seen the graph object in class with the node and edges? Yes or no? OK So here is shortest path depth-first. So there's a lot of debugging code here, but-- and some administrative stuff-- so all this is doing is just making sure that the nodes we're looking for are in the graph.

And, actually, let me backtrack. So when we first call shortest path, we're going to call it with a graph object. We're going to start in an end node. And we're also going to have this parameter visited, which keeps track of the nodes that we've already seen. And we'll get to that in a second.

So one of the first things that we do that's of any importance is, we check to make sure that the start and end nodes are actually in the graph, because you can't get from one to the other if they don't exist. And now we're going to construct a path or a list that just contains the start node as its element. And then we're going to check to see if start is equal to end. So if we're already at our goal, then the shortest path is just us, right? We don't have to go anywhere.

AUDIENCE: [INAUDIBLE]

PROFESSOR: For comparison purposes. I mean, if you look at the definition of node, it's pretty sparse-- pretty spartan. If we wanted to make it so that we just added the node object itself, would have to add an underbar equal method-- stuff like that-- and in this case, we don't want to bother with kind of complicating it like that.

So if we're not at the end, though, now, we need to figure out what the shortest path is from each of our children to the goal node, right? So we have a variable that we call shortest, and that's going to keep track of what our shortest path so far is, and then we're going to iterate through all the children in this node. And if the node is not in visited-- and that's where this parameter comes in-- we're going to say, OK, well, let's take a look at it. And then we're going to say-- we're going to add it to visited so we know that we visited this node.

It sounds kind of cyclic, which is funny because we have visited so that we avoid cycles. Because if we've already visited a node and we figured out what its shortest path is, why would we want to visit it again? If we're on a path, and we're saying-- let's say I have-- I'm trying to figure out the shortest path from v1 to v4-- and I'm using depth-first-- and so I decide depth-first first goes to v2, then to v3.

Now, it's got two choices on which nodes to get the shortest path for. Let's say, for some reason, when it gets the list of children, it's going to get v1 and v4, and if it's looking at v1 before it looks at v4, then, if we didn't have that check in there, just to make sure that we are not visiting-- or looking at other nodes that we've already visited-- then the algorithm would just go here, and it would repeat itself in a cycle, like that. So that's what that visited parameter is doing, is it's preventing that cyclic

check.

So now we ask the question-- or we make a recursive call to shortest path, right? And the only parameter that changes is the start node. And we're going to ask it, what's the shortest path from this node to the end. And it's going to call itself again and return an answer. And if it doesn't return anything, then we're just going to ignore it and continue.

But if it returns something, and either we haven't found shortest path yet, or the length of this new path that it's found is shorter than the shortest path that we've already found, then we're going to record it, and say that this is our new shortest path here. And then we're just going to keep iterating through all the children of the node until we've exhausted all possibilities. And then once we finish going through all the children, we're going to say-- if we found the shortest path-- that means that there exists a path from one of its children to the goal node-- then we're going to add it to our existing path, which is just ourself.

So we're adding ourselves to the beginning of the shortest path that it found. And then we will return it. So it's kind of growing the shortest path from the back to the front, right?

Breadth-first search works in the opposite direction. So the way that-- well, first, is anyone confused by depth-first search?

AUDIENCE: [INAUDIBLE]

PROFESSOR: So this line here-- this is-- well, this if statement first is checking to see that we've got-- that one of our children has a shortest path. It's possible that none of our possible children leads to the goal node, so let's say that I have another kind of subgraph on here. When v2 is my start node, I'm still going to check these two children. And let's say that my goal node is to get to v8, right? Well, I'm still going to check to see what the shortest path is for both of these children.

Well, once I use this is my start node, there's obviously no path to the actual goal node, so the depth-first search call, or the call to the shortest path, is going to return

none in this case. And we need to check that, so that's what that bit of code is doing there. It's saying if there is a shortest path, then we're going to just add ourselves to the front of that shortest path, and return that as our answer. But if there is no shortest path from one of our children-- from any of the children on the start node-- to the goal node, then we're just going to return none as our answer because we can't get to the goal node from where we are. Did that work for you?

So why don't we take a look at how this is working. So we're going to try DFS on undirected graph. And the code that does this is called Test 2 here, and all it does is, it creates a graph with 10 nodes.

And, in this case, it's going to be an undirected graph. And we're going to create a bunch of edges. So-- is that diagram you sent out, is that the representation of it?

AUDIENCE: This is the code from lecture, Professor [INAUDIBLE] code. So it only uses 5 nodes.

PROFESSOR: So we have this graph, and what we're going to do is use depth-first search to compute the shortest path from here to here. So this is showing the depth of the recursion, right? So we start off on node 0, and then it starts looking for the shortest path from 1 to 4.

And at the same depth, it's going to try and find the shortest path from 2 to 4. So it starts out here, and then it asks what's the shortest path from 1 to 4, and then what's the shortest path from 2 to 4. And so when it's looking at 1, now it's going to ask what's the shortest path from-- I want it to do that-- Hey, Sarri? Is there a bug in your code?

AUDIENCE: Is there?

PROFESSOR: So is this the lecture code?

AUDIENCE: None of this is mine. I did the breadth-first search. This is the depth-first?

PROFESSOR: Yes, because it seems like it's checking node 0 twice.

AUDIENCE: It didn't do that on mine.

PROFESSOR: So it's going from--

AUDIENCE: Oh, no-- because there's a directed-- is from 1 to 0, right? Yes. So what it's doing-- what the code does is, it says-- it does a depth-first, so first, it looks at node 0, and then it goes for child in-- for all the children nodes. What's the first child of node 0? It's node 1.

PROFESSOR: Oh, because 0 hasn't been added to the visited list.

AUDIENCE: Right. And then--

PROFESSOR: And then it asks, what are all the children of--

AUDIENCE: Well, no. The print statement comes before it discovers that checking node 0 is an invalid path. I forgot to add another print statement. If you go to the code--

PROFESSOR: OK. So where are we at?

AUDIENCE: Yes. So see how I have the very first at the top of the function? See how there's the if to print? I say that, but then there's this check here if it's not in visited. If it is in visited, there's no print statement that says--

PROFESSOR: --that says that-- you know-- it's--

AUDIENCE: That-- yes. So, basically, what's happened is, when we do that second check where we end up finding the second path from 0 to 4, it ends up hitting that test that says, we've already visited node 0, so we don't continue that.

PROFESSOR: I got it. I got it. That was just a little--

AUDIENCE: There was no print statements in the code before, and I was having a really hard time figuring it out, and I did this, and this made it a lot more clear-- I thought-- to try and figure out how the depth-first search was working.

PROFESSOR: No, I think so, too. It's just maybe we should add a couple extra--

AUDIENCE: Yeah, I agree.

PROFESSOR: Sorry. I was perplexed-- because I'm like, uh-oh. So it's working correctly. Is there any confusion on depth-first search, in spite of that? OK.

So let's start our breadth-first search. So you guys covered depth-first search in lecture, so we need to do breadth-first search. OK. So the idea behind breadth-first search is that instead of asking the question, what's the shortest path from v1 to v4 and adding ourselves to the front of that, to get the shortest path, we're going to build the paths from the start outward-- or from the start forward-- so, in this case, we're building the paths from the goal backward.

In this case, we're going to say, I'm at v0, so my current partial path is v0. Then I'm going to look at v1 and v4. And so I'm going to have-- I'm going to say, now, this is my list of partial paths to the goal.

And then, for v1, I'm going to say-- I'm going to ask what its neighbors are, and we have v2. And then, for v4, we're going to ask the same question. So now we're building our partial paths. So, conceptually, what we're doing is, we're just maintaining a list of all the paths that we-- or all-- the history of nodes-- or paths that we've been looking at and just kind of going out one by one by one.

So if we look at this in code, shortest path-- BFS-- has a slightly different call signature than the DFS method. So we still get a graph, and then we get this variable called paths. And what paths contains is the partial path, or list of the partial paths of tuples.

So the format of this is, each element has a list of the nodes in the path and then the length of that particular path. So we also have the goal node. And what we do-- the first thing that happens is, this pass gets shorted-- or sorted. So it's sorted by the length of the path. So is anyone puzzled by this lambda here?

AUDIENCE: Yes.

PROFESSOR: OK. So when you call the sorted function, you can pass it a list. You can also pass it this key parameter. And this key parameter is a function that takes an object, and

it'll return, kind of, the value associated with that object. So, in this case, each of these paths contain-- each of the elements and paths contains a tuple. And what this key function does is, it says, each element in this path has got a value that is equal to the length of that path.

And this length parameter-- that's just the second element in the tuple, right? So when sorted does its work, it's going to call this key function on each element in path. And what this function has to do is return the value associated with that particular element.

AUDIENCE: Is lambda just a key word that generates an effect?

PROFESSOR: Lambda is what's known as an anonymous function. And he covered that in a lecture at one point.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: Did he just use lambda?

AUDIENCE: He blew past it, so-- this is something called lambda. Ask your TAs.

PROFESSOR: Well, since I'm your TA and I'm here-- so real briefly-- lambda is a way of, kind of, doing really simple-- not really simple functions-- but it's a way of creating anonymous function. So let's say that I want to create a function that squares a number. This is exactly-- well, not exactly-- but, it's equivalent to this. I can use g as a function just as easily as I can use h. That's all.

And it's useful for situations where you have elements that don't have an order to find. So like, sorted needs to know a value of an element in order to put it in order-- in order to do the sorting, right? So that's what this function does, is it gives each element in that list a value so that the sorting can do its job. So the key that we want to sort on, or the item that we want to sort on, is the length the path.

And the reason why we want to do that is, we're going to take the first partial path that exists in our paths, and for every node in the shortest-- or for every node in the-- for every child of the last node in this path, we want to check to see if it's the goal,

and if it's not, then we're going to append a new partial path, which is the path that we're looking at, plus one of the children nodes. And then the length of the path to this value called new paths.

So it's saying-- this node that I'm looking at-- let's say that this is my partial path. What it's doing is, it's looking at this last node in this path, and it's saying what are all the children of this node? And if none of the children are the goal node, then it's going to create a new set of paths that are composed of this path, plus all the children.

AUDIENCE: So if v2 also went to v10, would the new path be easier [INAUDIBLE]?

PROFESSOR: No. So we should probably get rid of this. The reason why breadth-first search has its name, is that it doesn't try to go towards the end immediately. It grows gradually. So if you can kind of envision a graph like this.

Depth-first works by going-- by proceeding down until it finds the goal node for each of the possible paths. Right? What breadth-first search does is, it starts at a node, and then it builds all the paths from that node. And then for each of these, it builds the paths. So that's what I'm saying when I say partial paths.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes. It's a new partial path, so--

AUDIENCE: So it would be v0, v1, v2, v4-- the next one. And v0, v2, v [INAUDIBLE].

PROFESSOR: So let's say this is v0, v1, v2. Let's say that I have this partial path already. All right? When I build a new partial-- when I'm looking at this path, which is what I'm doing when I pop it off the front of the list of paths that I already have-- I'm going to look at all its children. So I'm going to look at these two nodes right here.

And then I'm going to say-- let's say that my goal node is here, and then I'm going to say that these aren't my goal nodes. So I'm going to create new partial paths, and one of them is going to be v0, v1, v2, and v4. So I already have that. And the

other partial path is going to be v0, v1, v2, and v5.

AUDIENCE: You just find every single possible path to get the tuples [INAUDIBLE].

PROFESSOR: What's that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Until you reach the goal node. So with depth-first search, you do have to-- well, no, not even with depth-first search. There's some interesting properties about the different searches, but we just want you to be familiar with how they go about doing their task.

So the idea is that we start off with a partial path-- just this guy-- and then we find all the other partial paths. So it'll start off with just this guy as its list of paths-- so one element-- and then it's going to pop this off the front. It's going to say, OK, I'm looking at this partial path. What are the additional paths that I can build off of this? And I can build four additional paths off of this.

So now I have a set of four paths that I want to look at. And then it takes a look at, say, this partial path of v0, v1, and it says, what are the paths that I can build off of this, and then adds it to the end. And then it says, what are the paths that I can build off of this guy-- adds those to the end-- what are the paths that I get from this guy-- adds those to the end, and so on and so forth.

AUDIENCE: So you keep track of all the paths that you [INAUDIBLE]?

PROFESSOR: Right. Oh, OK. Is that what you meant?

AUDIENCE: You hit a node-- and there's another possible thing-- but if you don't hit a node-- if there's some paths somewhere else in the graph, then it doesn't ever reach into that. It just doesn't account for it.

PROFESSOR: Well, I mean, it is possible for breadth-first search to go find nodes that don't reach the goal, but what's going to happen in those cases is, they're going to-- so let's say that there's such a path at the front of this paths list, and it doesn't have any

children. So you can't go anywhere after you've gotten to this node. Then what's going to happen is this for loop's not going to execute, right?-- because it has no children. And so it's just going to be discarded as a possibility.

So I mean, the key thing is that you're generating these new partial paths for every node that you're looking at. And you're adding those to a list or a queue of partial paths that you need to examine in the future if you don't find your goal. If you do find a partial path that has a child that is the goal, then you can just immediately return that path because you know it's going to be the shortest path, in this case. So-- I don't know.

AUDIENCE: So what happens when you find two partial paths that both reach?

PROFESSOR: Well, then it depends on which one appears first in your list. So you're asking-- let's do something really simple-- if I have--

AUDIENCE: Oh, OK. So that makes sense.

PROFESSOR: So at some point, you're going to have a partial path right here, or a list of partial paths that consist of v0, v1, and then v0 and v2. Whichever path you ultimately wind up returning is going to be dependent on whether or not this one comes first in your list of partial paths to check, or this one comes first in your list of partial paths to check. Did that kind of answer it?

So is everyone good on breadth-first search, conceptually? Is the code flummoxing anyone?

AUDIENCE: How do you find which one is the shortest?

PROFESSOR: If you're building your partial path-- so you've popped off this path that you're examining-- and you look at all its children, and one of the children is a goal node, then you know that you've found the shortest path, right?-- because you've been building this incrementally, one by one by one. And then if you've found the goal node, then you know to-- you know, you already built up the shortest path possible. Because if the goal node had been on a shorter path before, you would have found

it already.

AUDIENCE: That would be a different scenario if that path were weighted.

PROFESSOR: Well, if the paths are weighted, then it's a little different because here, we're just doing shortest path. What you're talking about is doing a least cost path, in which case the main difference would be how it sorts its list of candidate paths, right?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Hmm?

AUDIENCE: When it does sort it, it would say, keep this [UNINTELLIGIBLE] path [UNINTELLIGIBLE PHRASE]?

PROFESSOR: Yes. So in this case, our lambda function is saying that we're going to sort these lists based on their length. But if I wanted to say, sort them based on the sum of their weights, then I would have a function that sums up the weights on that path and uses that in the sorting in order to select the next partial path to search.

AUDIENCE: Although, in that case, you'd actually have to ennumerate every single possible path, to show you found the lowest cost path.

PROFESSOR: No.

AUDIENCE: Because doesn't-- I might understand this-- is that you find all the paths of length one through a certain point and all the paths of length 2.

PROFESSOR: Right.

AUDIENCE: And so if you're worried about cost, it's possible-- in principal-- that your longest path is your lowest cost path.

AUDIENCE: Wait. You're not sorting it by length then.

PROFESSOR: Right.

[INTERPOSING VOICES]

PROFESSOR: But what he's saying is that because you're growing it by one each time, then it's possible that-- let's say that I have partial paths of length 2, but my least cost path is actually of length 5, then even if one of these nodes reaches-- like, say that v2 reaches the goal before the actual least cost path, then you can't stop. You do have to integrate through all of them.

AUDIENCE: Well, you just sort it by cost. I think that it's a better example to say that if you have one partial path that has cost 7 ...

PROFESSOR: Well, actually, wait. No. You're actually right. Because if you sort it by cost, then your least cost path is going to be in front of your queue all the time.

AUDIENCE: No, I don't think that's true. Because imagine this-- imagine you had a graph that was like-- this is your start node. This is your end node, and you go like this, this, this, this. And so you make a breadth-first, where you go to this node and this node, and the cost of this is 1 and the cost of this is 8. The cost of this is 20 and the cost of this is 1.

This is your shortest path, or this is your cheapest path, but when you add them-- they're both of length 1-- so you explore this one first because it's the cheapest at a cost of 1, but the total cost is 21 to continue from this node. But the total cost is [UNINTELLIGIBLE] that node. So I think that he's right. Right?

I thought you had to expand all nodes when you're doing breadth-first. That's why you do, like, [UNINTELLIGIBLE]. But, yes. For weights you have to expand all of them.

PROFESSOR: So--

AUDIENCE: [INAUDIBLE]

PROFESSOR: Yes?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Possibly, yes. I'm blanking on the answer. But in this case, we don't have to enumerate all paths.

AUDIENCE: We're just looking for the shortest path-- we can just find the shortest path.

PROFESSOR: Right. But if you're incorporating costs, then you would use a different algorithm, so it'd be, like, actual shortest path algorithm or something like that. So why don't we just run this.

AUDIENCE: So is one better than the other, when you [INAUDIBLE]?

PROFESSOR: They have different running characteristics. It really depends on your application. The nice thing about depth-first search is that it's fairly memory efficient because with breadth-first search, you're storing all the possible paths-- all the possible partial paths.

With depth-first search, you only have one path and memory at a given point, right? You have the path that you've already found from one of your children to the end node, plus 1. But with the breadth-first search, you're going to have a list of all the partial paths that you have to search through.

So let's see. This doesn't make sense.

AUDIENCE: I don't think it matches the picture.

AUDIENCE: [0, 4] is a path, isn't it? From the graph?

PROFESSOR: Right. Yes. No, it's not matching the picture.

AUDIENCE: [INAUDIBLE]

PROFESSOR: What's that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Is what?

AUDIENCE: It was undirected.

PROFESSOR: Yes, it was an undirected graph-- that's why. So if we run it directed, let's see what happens.

AUDIENCE: 0 to 4 is still in the graph.

PROFESSOR: Yes. You are absolutely right. Oh, you know what's going on-- it's running both. It's running the depth-first search.

AUDIENCE: No, it's running both the directed and the undirected.

PROFESSOR: Well, it's also running the depth-first search as well. That was my question, because I was looking at this, and I was, like, that's not breadth-first search, that's depth-first search. This, on the other hand, is breadth-first search. So there we go.

So if we have a graph-- and let's see-- so it starts off with node 0, and then it builds a new list of partial paths with [0, 1], [0, 2], so it's got, in this case, this partial path and then this partial path. And then it expands one of them. So, in this case, I guess [0, 1] was the path that was first on the list, right? So it's going to expand the children of 1, which is just 2 and itself. But it's going to avoid cycles.

AUDIENCE: [INAUDIBLE]

PROFESSOR: What's that?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh, yes. And then it's going to expand this partial path-- the [0, 2] -- because it's the shortest one, and it's going to add that new partial path to the end. And then it's going to expand this guy, and add the expansion here, and then it's going to expand this guy. And it turns out that one of the children of 3 is 4. So it's found the shortest path from 0 to 4. Make sense?

AUDIENCE: So, can you go back to [INAUDIBLE]? Then why does it say [0, 2, 3] then?

PROFESSOR: That's the representation in the pass list, so each element in the path's list is represented as a tuple. The first element is the list of nodes on that path, and then

the second element is the length of that path. And that's why we had that lambda function in the first place, right? So if you read the specification here, you have path length, and then you have the nodes that are on that path. And when we sort the path list-- because we need to make sure that we're looking at the shortest path-- then we use this lambda function in order to get that length for each tuple.