[SQUEAKING] [RUSTLING] [CLICKING]

JUSTIN SOLOMON: OK, team. Let's get started for the day. It's a pleasure to see all of you guys. In case you don't remember, I'm Justin. I'm the third instructor of 006 that you probably forgot about, but you're going to see a lot more of me in the graph theory part of our course because that's the part of algorithms that I like. If I were reincarnated as a theoretical computer scientist, I would probably go into this area.

Hey, guys. OK. We have our PhD admit visit days coming up for the next couple of days I'm working on my camp counselor cheerleader voice. So don't make me wake all of you guys up for the day. You're not going to like it.

But in any event, so in 6.006 if you look back at the course outline, we're officially starting part two of this class. There are a few corollaries to that fact. So unless there are any questions about that, we'll get started with our new unit in 6.006 which is a graph theory.

If you're wondering, there's a graph on the screen here. But of course, we'll fill in a little bit more information today throughout our lecture. When I was learning how to teach, which I'm still doing, actually my PhD advisor told me if you want somebody to learn something, you have to write it as big as possible. And so I'm really leaning into that approach today in our slides.

So in any event, so today we're going to have our first lecture on graphs which I think will somewhat be a review for many of you guys. And if it's not, that's cool too. Because we'll start from the beginning and kind of build up all the notions that we need to understand and process graphs and hopefully by the end of lecture, have some style of algorithm for computing the shortest path from one vertex to all the other ones.

So in case we forgot a little bit of terminology, a graph-- some people call this network, but sometimes that term is overloaded with a few different kind of variations on the theme-- is a collection of two things. That's what this parentheses notation means. There's a set of vertices and a set of edges.

And the edges, like you can see in the sort of third point on our screen here, are a subset of v cross v. Now this is fancy notation for something really, really simple. Because what is this telling me? This is telling me that an edge, like in the picture that we see on the screen here. it just just something that connects to vertices together.

So if I think of there being a pair of vertices, like the from and the to, then that is a subset of the cross product of v and itself. So hopefully the notation in that third line on the screen makes some sense. This is just fancy notation for edges are pairs of vertices. But of course, inside of that notation there are two special cases that we care about in this class. One is when you have a directed graph, and one is when you have an undirected graph-- because I said them in opposite order from what's on the screen.

So in an undirected graph, I guess we still think of an edge like a pair of vertices, but really I should have notated this slightly differently-- in fact, maybe I'll revise it in the slides before they go into OCW-- where instead of writing e equals w comma v, I should write in fact equals v comma w. And notice that there's a slight difference between the notation on the slide and what I've written on the board, which is the set notation here.

What's the difference between parentheses and squiggly lines is that this guy is unordered. This is a set of things. And what's on the board is ordered-- or what's on the screen rather. And of course, in an undirected edge there's no such thing as an edge from w to v being distinct from an edge from v to w. Those are the same thing. They're undirected. It just is a notion of connectivity. Whereas in a directed graph, now we're going to use that parenthetical notation to say that the edge from w to v is different than the edge from v to w.

That's going to make a big difference. So for example in the graph on the right-- let's maybe redraw it on the board here. So we have four vertices. I drew this last night, and I'm hoping that this example actually works. Like that-- can I get from the upper right vertex to the lower left vertex following edges in this graph? I heard one person. Everybody on three-- 1, 2, 3.

AUDIENCE: No.

JUSTIN SOLOMON: No, right. Because if I wanted to-- I mean maybe I think of drawing this path here-- but of course, if I would go from the upper right to the lower left-- this is like the ugliest thing I've ever done, I'm so sorry-- you can notice that the edges are pointing in the up direction here.

So I'd have to go against the stream of the water, but that's not allowable in the directed graph case. Of course, I'm already anticipating the notion of a path which we haven't really defined yet. But I think intuitively, that's sort of the big difference between a directed and undirected graph.

Does that distinction makes sense to all of you all or have I managed to lose you in four minutes or less? Excellent. So I flipped things a tiny, tiny bit from the course notes because I figured we'd define what a graph is first before telling you what the implications are.

But in any event, I think it's really not a big stretch of the imagination to say that graphs are literally everywhere in our everyday life, right. Any time that we come up with a network of stuff connected together, implicitly the right abstraction often in the back of our heads is to think about a graph.

So some simple examples that I think would all come to mind for us would be like computer networks-- so the nodes or the vertices of your graph in that case, maybe are computers, and then the edges are roughly the cables connecting them together in my very coarse understanding of how networks work-- or maybe at a social network-- the nodes are people on your social network, and the edges are friend relationships or frenemy relationships or whatever. In fact, I think you could think of both directed and undirected versions of that particular network.

In road networks, maybe I'm working for Google and I want to tell you the shortest path between your house and MIT. Of course, in order to do that and essentially behind the scenes, we're solving some version of computing the shortest path between two vertices in a graph.

That's a tiny bit of a lie in the sense that there's a lot of structure in that problem that we're not going to leverage in this course. A road network is a very special type of graph, and if you take an advanced course maybe you'll say, well, if I know a little more about my graph I can do better than the general case we'll talk about here.

But the basic algorithms that we'll talk about in 6.006 are certainly relevant in that case and are really the building blocks for what goes on in the tools that are used every day on your phone when you open Google Maps or Ways or whatever. And of course, there's many others. So for instance, an example that maybe is a little bit more subtle would be the set of states and transitions of a discrete thing.

So think about like a Rubik's cube. So I could make a graph where the node is every configuration of my Rubik's cube, like every rotation. And then the edges are like can I get from this configuration to that one by making one simple transition, like one flip.

I don't actually know the terminology in Rubik's cube, I have a feeling you do, for one rotation. Twist-- thank you. And of course, there are many other places. So for instance, in my day job here at MIT I typically teach computer graphics courses. And actually graph theory, although we talk about it very differently, appears in that world constantly. Because of course, with sitting behind any 3D model on your computer is a giant network of triangles.

This is called a triangulated surface-- like this torus we see here. And this is nothing more than a graph. And in fact, if you squint at the algorithms that we cover in six eight three eight, you'll see they're roughly just graph algorithms in disguise.

In fact, if you take my graduate course one thing we'll do is we'll spend a lot of time doing differential geometry. And then we'll step back 10 feet and notice that exactly the algorithms we are using for computing curvature and bendiness on triangle meshes, just looks like a graph algorithm and can be applied to networks in exactly the same way. So it will be a nice kind of fun reveal there.

And of course, there's one last kind of fun application. I actually was gone the last couple of days at a conference on political redistricting. And the funny thing is most of the discussion at that conference was about graph theory. And the reason for that is sort of a theme that shows up a lot in geometry world, which is if I take my state, in this case I think these are the voting precincts in some state or another, and I look at adjacency relationships, then maybe I put a node for every precinct and an edge any time that they share a boundary with one another. Well now I have a network. And maybe a region on my graph is like a connected piece of this network.

And so anyway, this is one of these examples where graphs and networks and connectivity and so on just show up literally no matter where you go. They're totally unavoidable. And so that's what we'll be spending quite a bit of time on in this class here. Now you could easily take, I would argue, at least three entire courses on graph theory here at MIT, and you could easily build a PhD dissertation doing nothing more than really simple problems on graphs. Of course, in this class we're limited to just a few lectures out of many. So we're going to make a couple of assumptions both on the problems we want to solve, as well as in the graphs that we care about.

So in particular, one simplifying assumption, which actually really doesn't affect many of the algorithms we'll talk about here but it's worth noting explicitly, is that we'll mostly be thinking about a particular type of graph which is a simple graph. And in fact often, depending on how you define your graph, you kind of accidentally made your graph simple even if you didn't intend to.

So for example, we wrote that our edges were a subset of v cross v. Which maybe means that I can't have multiple edges that sort of traverse the same pair of vertices. So let's see an example of a graph that is not simple. So sorry, I haven't actually defined it. A simple graph is a graph that has no self loops, so it can't go from a vertex to itself, and every edge is distinct.

So let's make the most non simple graph we can think of. Like let's say I have two vertices. So maybe if I want to make my-- so there's a graph, right, two vertices and one edge. This is simple. If I wanted to be annoying and make it not simple, maybe I take this edge and I'd duplicate it three times just for fun. That violates the second assumption. And now to make it even worse, I could violate the first one by adding an edge that goes from this vertex to itself. This is not simple.

I don't know what you would call it actually-- general graph, I guess-- complicated because it's not simple. I don't know-- a multigraph. I always thought of that-- anyway, it doesn't matter. But in any event, in this class we're not going to worry about this particular circumstance. And of course, in many applications of graph theory that's a totally reasonable assumption to make. Any questions about the definition of a simple graph?

OK, so from now on whenever we think about a graph, in the back of our head we're going to think of our graph as simple. There's one nice property that a simple graph has, which I've written in really big text on the screen here, which is that the edges are big O of v squared. And in fact, let's expand that formula just a tiny bit.

So there's sort of two cases, one is when my graph is undirected, the other is when my graph is directed. So if I have a directed graph-- well, let's think about how many edges we could possibly have. So an edge is a pair of a from vertex and a to vertex, and I can never repeat it twice. That's sort of like the second assumption here.

So in particular, what do we know? We know that mod E-- or rather the number of edges in our graph is upper bounded by what? Well, I can take any pair of vertices-- like that-- but I have to be a little bit careful because my graph is directed-- so from and to matter here. So this is v choose 2 is saying that I can take any unique pair of vertices, but I have to put a factor of 2 in front of it to account for the fact that the source and the target can be flip back and forth. And of course, if I want to do the undirected I don't have to worry about that.

We'll get E here is less than or equal to just mod v choose 2. So this is just a fancy way of saying that every edge consists of two vertices, and my edges are unique. And one thing, if you just write down the formula for our binomial coefficient here, we'll see that both of these things-- oops, oh, yeah, sorry-- are at worse mod v squared here. And that makes perfect sense, because of course, an edge is a pair of vertices. You kind of expect there to be a square there. Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: I'm so sorry. I can't hear you.

AUDIENCE: So the 2 comes from the fact that it's from the source.

JUSTIN SOLOMON: Yes, exactly. So the 2 for the director case, comes from the fact that an edge from v to w is different than an edge from w to v. So remember that the binomial coefficient here, it's just counting the number of ways that I can choose two things from a set of size v, but it doesn't care about ordering. Yeah, any other questions? Fabulous.

So why is this going to matter? Well, these sorts of bounds, I mean they might seem a little bit obvious to you, but we're going to write down graph algorithms. And now when we analyze the runtime and the space that they take, we now have sort of two different numbers that we can think about-- the number of vertices and the number of edges.

And so for instance, if I write down an algorithm whose runtime is proportional to the number of edges, maybe then genErikally I could also think of the algorithm as having a runtime that looks like the number of vertices squared unless I put some additional assumptions on my graph. And so there's some connection between all of these different constants, and it's useful to kind of keep that at the back of our head. That sometimes you'll see a bunch of different expressions that really are encoding roughly the same relationship just in different language

Of course, that also means that we can be more precise. So sometimes a graph is what we would call sparse. So in my universe, almost all graphs that I deal with in my day to day life are extremely sparse. This is a consequence of topology. And because of that, an algorithm that scales like the number of edges might actually be much preferable to an algorithm that scales like the number of vertices squared because, in practice, often there are fewer edges than like every single possible pair. And so that's the sort of reason why it's we're thinking about these numbers.

OK, so let's continue making boring definitions here. So some other ones that we should think about involve the topology or the connectivity of our graph-- in particular, thinking about neighbors. So in general we kind think about pairs of vertices as being neighbors of one another if there's an edge between them. We have to be a little bit careful because, of course, when we have a directed edge, we have to be careful who's on the sort of giving and the receiving end of this neighbor relationship.

Yeah, so let's draw a really, really simple graph. So here's vertex 0, here's vertex 1, here's vertex 2. And maybe we'll have an edge going up, an edge going down, and then a cycle here. OK. Now we can define a lot of different notions of neighbors-- like the outgoing neighbor set, the incoming neighbor set. And the basic idea here is that we want to keep track of edges going from a vertex and edges pointing into one.

Yeah, so for instance, the outgoing neighbor set, which we're going to notate as Adj plus here-- what is the outgoing neighbor set of node 0 here? Well, if we take a look, notice that there's one edge going out of node 0, and it points to node 2. So of course, this is a set which just contains one other node. And similarly, the incoming neighbor set of node 0, well notice that there's one incoming neighbor from vertex 1, so that is a set like that.

Now of course, in an undirected graph the sort of distinction between these two things doesn't matter. So if you look at our final bullet point here, often in the undirected case we just drop that plus or minus superscript because it sort of doesn't matter. In any event, there's one additional piece of terminology that matters quite a bit, which is degree. And this is nothing more than just counting the size of this set. So the out degree is the number of edges that point out of a vertex. And the in degree is the number of edges that point in.

So notice in this case, both of those numbers are 1. Let's see an example where they're not. So in node 1, notice there's two edges that come out. So the out degree of node 1 is 2. There's one edge that points in, so the in degree is 1. OK, so often why are we going to do this? Well, we're going to get a lot of graph algorithms that like have a FOR loop over the neighbors of a given vertex. And then this degree number is going to come into play. It's worth bounding these things just a tiny bit.

So in particular, one thing we could think about-- I write too big, and I'm going to run out of space really quickly here-- is the following. So let's take a look at all of the possible nodes inside of my graph, and now let's sum up all of their degrees. So I'm going to-- let's see, if I look at this graph notice there's three edges adjacent to this vertex here, three edges adjacent to that one, two adjacent to this.

So we sum them all together. So it's just a convenient bound to have around-- is to sum these things, because we're going to have algorithms that look like for every vertex, for every neighbor do something. So we might as well know roughly how much time that's going to take.

Let's think about this. So what do we know? In an undirected graph every edge is adjacent to two vertices. So if we think about how we account for degree what do we know? Well we know that an edge sort of contributes to the degree of two different vertices. So if we think about it carefully here, what we're going to see is that if our graph is undirected-- oh, sorry-- is that right, wait I'm backward again.

So if I have a graph with two vertices and one edge and it is undirected, notice that is the number of edges here is 1. What is the sum of the degree? Well, it's 1 plus 1 equals 2. Yeah, so there's a 2 here if my graph is undirected, and E if my graph is directed, if what I'm counting is just the outgoing degree. Does that makes sense?

I think I managed to totally botch that sentence, so maybe let's try that again. So if I'm counting just the number of edges pointing out of every vertex, and I count that over all of the possible vertices, then there's two cases-- one is directed and one is undirected. So in the undirected case you get a 2 here because essentially every edge is simultaneously in going and outgoing. Whereas you get a 1 as the coefficient in the directed case. Does that makes sense? I'm sorry I botched that for a second. OK, excellent.

OK, that's going to be a useful bound for us later on. Now we think about graphs, of course, we just spent the last couple of weeks thinking about data structures. We should think about how to store a graph on a computer, and there's many different options. In fact, really one thing that you can do is sort of pair-- just like when we talked about sets.

There are many different ways to store sets. And one way to think about it was depending on how we're going to interact with that set we might choose one data structure or another to sort optimize the types of interactions we're going to have with that set and make them as fast as possible. This is exactly the same story for a graph.

So for instance, the world's dumbest representation of a graph would be to just have a long list of edges. So for example, for this graph up here maybe I have 0, 1, that's an edge, and then 0, 2, that's another edge, and then 1, 2, and then 2, 1. There's a big list of edges. It's really a set. I don't care about the order.

AUDIENCE: The first one's 1, 2.

JUSTIN SOLOMON: 1-- oh, you're right. I'm sorry. Yeah, the edge points up-- thanks Erik, or not Erik-- Jason. OK, so let's say that I have a graph algorithm, and I'm going to have to do something like check whether there exists an edge from v to w a bunch of times. How long is that going to take in this data structure?

Well, if I just have like a hot mess disorganized list of edges and I want to know does there exist an edge from v to w, all I can do is write a FOR loop that just goes along this and says, like this the edge I'm looking for. No. Is that the edge I'm looking for? No. So every single time I want to find an edge, it's going to take me time proportional to the number of edges of my graph which could potentially be up to v squared. Yeah, so this is not such a great representation of a graph on my computer.

So if we're thinking back to our data structure we may say, OK, so an edge list is probably not the way to go. Although notice that the way we notated what is a graph kind of looks like an edge list. But in any event, the more common thing to do is to source something like an adjacency list.

So the basic idea of an adjacency list is that what I'm going to store is a set that maps a vertex u to everything adjacent to u. So in other words, I'm just going to keep track of all the outgoing edges from every vertex. And now I have to decide, how am I going to store this object. And oftentimes, we're going to have to answer queries like does there exist an edge from v to w.

So how could I do that? First, I would look up v, and I get back sort of a list or a set of all the things that are adjacent to v. And I have to query that thing. And I want it to be pretty fast. So maybe what I do is I store the set of adjacent stuff as something like a direct access array or a hash table to make that look up fast.

So for example, how long would it take-- I see, I'm going to finish the sentence here-- how long would it take me to check if an edge existed in my graph? Well, what would I do? I would first pull out this object, and then I'd look inside of here. So if I stored this as a hash table, then the expected time I would have order one look up, because this is order one and then you have another order one look up there. So we went from v squared to one with one simple trick. Yes?

AUDIENCE: Does it matter what direction [INAUDIBLE]

JUSTIN SOLOMON: That's a great question. So this is a design decision here. I'm sorry, in my head I think a lot about undirected graphs, and I'm going to make this mistake a lot. And I'm glad that you caught me. There's a totally reasonable thing to do, which is maybe just to keep track of the outgoing edges for every vertex. This is a design decision. For an algorithm maybe I want to keep track of the incoming edges. Whatever, I just have to make sure that it aligns with what I want to do with my graph later. Excellent point.

Sorry, as a geometry person we rarely encounter directed graphs. But it's important to keep remembering that not everybody works on the same problems that I do. OK, now if I wanted to be totally extreme about it-- as just a third example of representation, which actually, in some sense, you could think of like an adjacency list-- we need an adjacency matrix where now I just keep a giant v by v array of like does this exist, does that edge exist.

Now it's really, really easy to check if an edge exists. But now let's say that I make a graph algorithm that's going to have a FOR loop over all the neighbors of some vertex. So here, if I wanted to loop over all the neighbors of u, I could do that in time proportional to the number of neighbors of u. But if I just have a big adjacency matrix, just a bunch of binary values-- like for every pair of vertices are these vertices adjacent-- yea or nay.

If I want to iterate over all my neighbors, now I have to iterate over all the vertices and check is that number one and then do something. So actually that can incur some additional time and additional space. Does that makes sense?

So in any event, that's a sort of a lazy man's graph representation. I use it a lot when I'm coding because adjacency matrices are easy to work with. But it does incur a lot of additional space, and it's not always the most efficient thing even if you have the space because iterating over neighbors, it actually can take quite a bit of time.

OK, so the real point of our lecture today is to start introducing sort of the canonical problem that we all worry about on graphs which is computing paths, in particular shortest paths. So the first thing we should do is, of course, define what a path is on a graph. So we're going to talk about our graph like a road network. Let's think of maybe every node here as an intersection. So this is a roughly Kendall Square. See it's a square.

But in any event, let's say that I want to find-- maybe a question one would be does there exist a way to get from vertex 1 to vertex 3. And then a better question to ask would be does there exists a short way to get from vertex 1 to vertex 3. Then of course, the first thing I have to do is to define my enemy. I have define what I'm looking for, which is a path.

So a path is nothing more than a sequence of vertices in a graph where every pair of adjacent vertices in that sequence is an edge. I think this all aligns with our intuition of what a path is in a graph. So for instance, here's a path p equals v1, v2, v3. So notice that there's an edge from v1 to v2 and also an edge from v2 to v3. So it satisfies the assumptions set forth in our definition.

What would not be a path in our graph-- would be like v1 comma v3, because there's no edge there. OK, so if we talk about paths, then there's a very natural notion which is the length. Length, I guess you could think of like the number of vertices in your path minus 1, or the number of edges that your path traverses. Those are the same thing.

So for instance, the length of the path p here is 2. Does everybody see that? A very common coding bug that I encounter a lot is adding 1 to that number by accident. Because of course, there's one more vertex in your path than there are edges. OK, and there are many different-- there could be potentially more than one path between any pair of vertices.

So let's say that I have an undirected graph that looks like the following. So it's just a square plus a diagonal. So here are nodes. So then a perfectly valid path from the lower left to the upper right would be to go one over and one up, but of course, there's a more efficient way to get from the lower left to the upper right, which is to go across the diagonal. And so when we talk about the shortest path, it's nothing more than the length of the path that has the fewest number of edges or vertices between any pair of vertices in my graph.

OK, so this is our enemy. This is what we're after. It's computing the shortest path between vertices in a graph. And this is the thing that we'll be talking about quite a bit in this course. Because of course, it's a very practical matter. Like when I want to solve routing problems, I want to move packets out of my network, I'd prefer not to-- well, unless I'm doing Tor-- I would prefer them not to hit too many computers in between. Then maybe I want a computer shortest path. Or on a surface maybe I want to move information in a way that's not too far away.

But of course, there's sort of many variations on that theme when we talk about shortest path or even just existence of a path. So these are three sort of model problems that we might solve on a graph. So the first one, which in this of course we're calling the single pair reachability, would be the idea that I take two vertices s and t on my graph g, and I ask you does there exists a path between s and t.

So what would be the sort of extreme example where this problem may not always give back the answer yes? Somehow in our head, I think we think of all graphs as being connected. But a perfectly valid graph the way we've defined it would be like 10 vertices and no edges. This function would be very easy to code if that were the only graph you ever cared about. But any event, the existence of a path is already a query that takes a little bit of algorithmic thinking. We haven't figured out how to do that yet.

Now another problem we can solve would be the shortest path. Given a graph and two vertices, we might say, well, how far apart are these vertices of my graph if I want to use the shortest possible distance from one to the other. Notice that I can use the second problem to solve the first one. Because what's the length of the shortest path between two vertices that don't have a path between them? Infinity or a shrug-- that's actually a totally valid answer. Yeah, that's right.

So how could I implement the reachability code? Well, I could call my shortest path code, and it gives me infinity. Then I return no, it's not reachable. And if it gives me not infinity, I return yes. So remember that a key idea in an algorithms class is this idea of reduction. That I can use one function to solve another. So in case, if we can solve shortest path, then we can certainly solve the reachability problem by calling that piece of code.

And then finally we could talk about single source shortest path. So notice now that there's only one input node here s-- so what this problem is saying is give me the length of the shortest path from s to every single other vertex in my graph. Does that makes sense? Like maybe I return a big array with all the information, every single shortest distance. So can we solve single pair shortest path using single source shortest path? Absolutely.

I could take s in my single pair shortest path problem, compute the shortest path from s to literally everything else, and then throw away all of that information except the shortest path to t, and now I'm good. Now I haven't justified that this is the fastest possible way to solve that second problem, but at least it shows that if I can solve problem three I can also solve problem two. If I can solve from two I can also solve problem one.

So in today's lecture, we're just going to worry about problem three. In other words, these things are sort of listed in increasing order of their difficulty. OK, so in order to think about the single source shortest path problem, we're going to make one additional construction. And this is an idea called the shortest path tree. I got lazy drawing PowerPoint slides at 2:00 AM yesterday, and instead thought I'd draw a picture on the board.

So let's draw a graph. So here we have a, b-- I'm going to use letters instead of numbers to refer to nodes from now on because I don't want to confuse the length of the shortest path with the index of my node. So here's a, b, c-- I'm going to match my notes here-- d, e, f. Here's a graph-- again undirected because your instructor likes to think about undirected graphs. But I know I'm going to get feedback that I shouldn't have done that later.

But in any event, let's say that I want to compute the shortest path from a to everything else-- or the length rather. So first of all, even without talking about an algorithm, I think it's pretty easy to guess what it is. So clearly the shortest path from a to a has length 0. The shortest length from a to b is 1, from a to c is 2-- because I can follow these guys. Now it gets complicated. It branched. So the next shortest path is length 3, and then 4 like that.

Does everybody agree with me that the numbers I've decorated here are the length of the shortest path from a to everything else? But what have I not done? I haven't told you how to actually compute the path, I've just given you the length of the path. So I may want a piece of code that in addition to doing single source shortest path length, also gives me a single source shortest path.

So initially when I think about that, I might think about, well, how do I even write down a data structure that can store all of those paths. Well every path could have like v vertices in it, right. It could be that for whatever reason, there's a lot of branching in my graph. And all the paths are super long. Actually, I guess I have to think about whether branching would make them longer or shorter.

But in any event, I could have a really boring data structure that just for every single vertex keeps track of the shortest path from a to that vertex. How big would that data structure be? Well, if the only bound I have on the length of a path is that-- it certainly at most it takes all the vertices in my graph-- then any one path will take v space. So that would take v squared space total. That wouldn't be so good. Because somehow I have an amount of information on my graph currently that's linear. It's just the length of the path. If I want to actually reconstruct that path, initially sort of spiritually feels like I need way more space to do that.

But the answer is that we actually don't. That we're going to only need linear space, and the idea for that is to store an object called the shortest path tree. Yes?

AUDIENCE: Just for [INAUDIBLE] previous [INAUDIBLE].

JUSTIN SOLOMON: So the question was about recursion. We haven't actually written down any graph algorithms. So we're going to defer on that until we actually recurse. And then we'll think about it more carefully. Yeah, but it's a totally reasonable question. There are plenty of recursive graph algorithms out there. And then we'll have to do our counting very carefully for sure.

Right, so instead, we're going to define an object called the shortest path tree. And the basic trick here is to say, well, how did I get from a to c? Well, there's always a vertex, which is its predecessor, on the shortest path. And shortest path have this really beautiful property, which is that the shortest path from a to c, if I truncate it-- right, so it goes a to b to c-- then the truncated one is also the shortest path to that previous vertex.

So let's think about that a little bit, because that sentence was, as usual, poorly phrased by your instructor. So let's say that I have the shortest path from a to d, which is very clearly a, b, c, d. I think we can all agree. And now I take like this sublist. I just look from a to c. Is there ever a circumstance when this is not the shortest path or a shortest path from a to c?

No, right because if there existed a shorter path from a to c, I could splice it in here and find the shortest path from a to d. Do you see that? So based on that reasoning, rather than string like this giant set of shortest paths, sort of actually applying, in some senses, recursive suggestion, instead I can just think of the one vertex that's before me in my shortest path. I'm going to trace backwards.

So let's take a look at our graph here. Essentially, the object I'm going to keep track of is like a predecessor, right. So what is the predecessor of f on the shortest path? It's actually either d or e. It doesn't matter in this case. Maybe the predecessor is e for fun, right. What's the predecessor of e? Well, clearly the previous vertex on the shortest path is c. Similarly for d-- now we have b and a and a bunch of arrows that point this way.

So for every vertex I'm just going to start an arrow pointing toward the previous vertex on the shortest path. I'm not going to store the whole shortest path, just the very last edge. So first of all, how much storage does this take? It takes v space. Do you see that? Or the size of the vertices space. Because every vertex just has to store one thing, which is the previous vertex on the shortest path.

Now what does my algorithm for tracing shortest path? It's really simple. I just start walking along these edges all the way until I get back to a. Now this object is called the shortest path tree. Notice I snuck in one additional word which is tree. Why is that? Can I ever have a cycle in this graph? It wouldn't really make any sense, right. These are shortest path. You should be able to kind of follow the gradient back to the original vertex.

OK, so in other words, I'm going to basically decorate my graph with one additional thing. We'll call it p of v which is the previous vertex on the shortest path from my source point to my vertex v. And what I think I've tried to argue to you guys today is that if I have this information, that's actually enough to reconstruct the shortest path. I just keep taking p of v, and then p of p of v, and then p of p of p of v, and so on, which sounds more complicated than it is, until I trace back to my original vertex. And this object conceptually is called the shortest path tree. Any questions about that? Yes?

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: If I had an edge that connected a to d, OK.

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: Oh, OK so the question was, let's say that our colleague here added an edge-- this is a great question. You know somebody was evil, my adversarial neural network, stuck an edge here because it was adversarial, and it wanted my shortest path code to fail. And now somehow the tree that I gave you is no longer correct. And my answer to that is yes. Why is that?

Well, by adding this edge here, the length of my shortest path changed. The shortest path from a to d is now 1. So this tree is no longer valid. I need a new tree. So now what would be the previous p of d here? Well, rather than being c, it would be a. Yes, that's absolutely right. And it actually is reflective of a really annoying property of shortest path, which is if I add one edge to my graph, the length of the shortest path to every vertex can change. Well, I guess with the exception of the source vertex.

Yeah, and that's actually a really big headache in certain applications. So for instance-- and then I'll shut up about applications and do math again-- I work a lot with 3D models. And there's a big data set of 3D models of like ballerinas. And ballerinas are really annoying because sometimes they put their hands together like that. And then suddenly the shortest path between your fingers goes from your entire body to like 0. And so incremental algorithms for computing shortest path can fail here, right. Because I have to update like everything if I accidentally glued together fingers like that.

So anyway, I'll let you think about how you might fix that problem. If you want to know more, you should take 6.838. Yes?

AUDIENCE: [INAUDIBLE].

JUSTIN SOLOMON: If you change your source node, the shortest possible change again. Yeah, so this is going to be one of these really boring things where I'm going to keep answering like any time I change anything about my problem-- I change my source, I change my edges-- I have to just recompute all the shortest paths. There are obviously algorithms out there that don't do that. But we're not going to think about them yet. OK.

So as usual, I've talked too much and left myself about 10 minutes to do the actual algorithm that's interesting in the lecture here-- although actually, it's really not so complicated, so I think we'll do OK-- which is how do I actually compute shortest paths? Yeah, and the basic thing we're going to do is sort of build on this tree analogy here.

We are going to define one more object, which I really like-- actually I enjoy this from Jason's notes because it looks like calculus, and I enjoy that-- and that's an idea of the level set. And so this is a whole set of things L sub k. And these are all the vertices that are distance k away from my source. So for instance, if my source vertex in this example is the vertex all the way on the left, then L0 obviously contains just that vertex, right. L1 is the next one. L2 is the third one. But now L3 is a set of three vertices because those are all the things that are distance 3 away from the source. That's what I've labeled in pink here.

OK, so that's all that this notation here means. Oh, I've made a slight typo because in this class distance is delta and not d, but whatever.

AUDIENCE: [INAUDIBLE]

JUSTIN SOLOMON: The shortest distance-- that's absolutely right. So for instance, I could have a very long distance from L0 to L2, right. I could just flip back and forth between L0 and L1, maybe go over to L4 and then go back. But that wouldn't be a terribly helpful thing to compute. That's absolutely right. Yes?

AUDIENCE: [INAUDIBLE].

JUSTIN SOLOMON: Oh, the red background is the set L. So for example, L3 contains these three vertices because they're all the things that are distance 3 away from the left. I got a little too slick drawing my diagram late last night. I'm kind of proud of it. OK, so essentially if I wanted to compute the length of the shortest path from all the way on the left to all the other vertices, one way to do that would be to compute all these level sets and then just sort of check what level set I'm in, right.

So we're going to introduce an algorithm called Breadth-First search which does roughly that. So Breadth-First search, the way we'll introduce it today is going to be an algorithm for computing all of those level sets, L sub i, and then from that, we can construct the length and even the shape of the shortest path. And I'm going to move to my handwritten notes.

OK, and here's what our algorithm is going to do. I'm going to write it in a slightly different way than what's in the notes and on the screen, but only slightly. So first of all, one thing I think we can all agree on is that level set 0-- oh, that's-- this chalk bifurcated-- it contains one node. What should that node be? The source because the only thing that's distance is 0 away from the source, is the source node.

OK, and in addition to that, we can initialize the distance from the source to itself. Everybody on three, what is the distance from the source to itself-- 1, 2, 3.

AUDIENCE: 0.

JUSTIN SOLOMON: Thank you. See you're waking up now, it's almost 11:00-- 12:00. What time is it? Almost 12:00-- OK, and then finally-- well maybe initially we don't really know anything about the array p, so we just make it empty. Because p of the source, it somehow doesn't matter. Because once I've made it back to the source, I'm done computing shortest path.

So we're going to write an algorithm that computes all the level sets and fills in this array p and fills in the distances all in one big shot. We're going to call it Breadth-First search. OK, so let's do that.

So we can use the notation here. And notice that there's basically an induction going on, which is I'm going to compute level set 1 from level set 0, level set 2 from level set 1, and so on, until I fill in all my level sets. Does that makes sense? So here's a slightly different way to notate the same thing.

I'm going to use a WHILE loop, which I know is like slightly non-kosher, but that's OK. So I'm going to initialize a number i to be 1. This is going to be like our counter. I'm going to say WHILE the previous level set is not empty, meaning that potentially there's a path that goes through the previous level set into the next one. Because as soon as one of my levels is empty, notice that like the Li for even bigger i are also going to be empty. There's like never a case when there's something not distance i but then distance i plus 5.

OK, so now what am I going to do? Well, let's think back to our graph. So like now I know that this guy is distance 0 away. That's what I started with. So now I'm going to look at all the neighbors of this vertex. And I'm going to make them distance 1 away. Does that makes sense? And similarly here, this guy is distance 2.

And eventually I'm going to get in trouble because maybe-- well, what's a good example here. I won't even try to draw. I could run into trouble if I don't want to add a vertex twice to two different level sets. Once I've put it in Li then I don't want to put it in Li plus 5 because I already know that it's distance i away. Does that makes sense?

OK, so what I'm going to do is I'm going to iterate over all the vertices in my previous level set. And now I'm going to look at every vertex that is adjacent to u. Because what do I know? I know that if I can get to u in i minus 1 steps, how many steps should it take me to get to any neighbor of u? i steps because I can go through the path, which is the length of i minus 1, add one additional edge, and I'll get to that new guy.

So what can I do? I can iterate over all of v, which is in the adjacent set of u. But I have to be a little bit careful because what if I have an edge backwards? So like for instance, here I have an edge back to the source. I guess this is-- yeah, that's a valid example. I wouldn't want to add the source to the third level set because I already added it in the previous guy. So I want to get rid of the union of all of the previous level sets. Does that make sense?

So in other words, I'm only going to look at the adjacent vertices that I haven't visited yet in my level set computational algorithm. And all I have to do is update my arrays, right. So in particular, I'm going to add vertex v to level set i because I haven't seen v yet. I'm going to set the distance from s to v equal to i because I'm currently filling in my level set i.

And then finally what is p of v? What is the previous vertex to v in my shortest path from my source? It's u, right. Because that's the guy in the previous level set that I'm building my path from, right. I'm going to set that to u. And then-- sorry, I ran out of space-- but I also have to increment i.

OK, so what does this algorithm do? It's just building one level set at a time. If we go back to our picture, so it starts by initializing L0 to just be the source vertex, then it looks at all the edges coming out of that-- in that case just one-- it makes that length 1-- and so on. And so this is just incrementally building up all these level sets.

Now there's a pretty straightforward proof by induction that this algorithm correctly computes the L's the p's and the deltas which is all the information that we need to compute the shortest path. I think you guys can do that in your recitation if you still need a little bit of induction proof practice here.

And the final thing that we should check is what is the runtime of this algorithm. I'm going to squeeze it in there just at the last second here. So let's take a look. So first of all, I did something a little-- oh, no it's OK-- in my algorithm actually in step zero I had to make an array which was the size equal to the number of vertices. Remember that in 6.006 how much time does it take to allocate memory?

Yeah, it takes the amount of time proportional to the amount of memory that I allocate. So already-- Steph, I see your hand but we're low on time. So we're to make it to the end. Already we've incurred v time because our shortest pathway array takes v space. But in addition to that, we have this kind of funny FOR loop where for every node I have to visit all of its neighbors. But first of all, do I ever see a node of twice here?

No, because I'm going in order of distance. And the second that I've seen a node in one level set, it can't be in another. That's our basic construction here. Well, conveniently for you guys, you already proved exactly the formula that we need. And if I'm lucky, I didn't trace it. Yeah, here we are.

So if we take a look here, this is exactly the scenario that we're in. Because what did we do? We iterated over all the nodes in our graph, and then we iterated over all the neighbors of those nodes. And that's the basic computational time in our algorithm. So that FOR loop, or that WHILE loop rather, in my code is incurring time proportional to the number of edges.

So what is the total run time for Breadth-First search? Well, we need to construct that array. So just at step zero, we've incurred v time. And then we have to iterate over something that takes up most the number of edges. So overall our algorithm takes big O of mod v plus mod e time.

Now, notice that this is-- you might view this as kind of redundant. By the way this-- I have a little bit of a quibble with Jason. But in this class we will call this a linear time algorithm because it's linear in the space that you're using to store your graph. I think that's a little fishy personally because this scale could scale quadratically in v, but I digress.

In any event, why do we need both of these terms here? Well, notice that if I had no edges in my graph, now this term is going to dominate. But as I add edges to my graph, this thing could go up to v squared. So this is somehow a more informative expression than just saying, well at worst this is v squared time. Does that makes sense? It's a slightly better formula to have.

OK, so with that we just squeaked into the finish line. We have an algorithm for computing shortest paths. And I will see you guys again I guess on Tuesday.