[SQUEAKING]

[RUSTLING]

[CLICKING]

JASON KU: Hi, everyone. Welcome to the 11th lecture of 6.006, our first lecture on weighted shortest paths. Until now, we've only been talking about graphs that-- where we measure distance in terms of the number of edges in a path. Today, we're going to generalize that notion. But I just want to go over what we've talked about in the last two lectures.

In the last two lectures, we've talked about two algorithms, breadth-first search and depth-first search to solve a range of problems. Here's some of the problems that we've been solving. Single-source shortest paths, where distances are measured in number of edges in a path. And we used BFS to solve this problem, starting from a single source, usually a vertex s that we call. And we solve that in linear time. And we solve that in order v plus e. That's what we called linear time for a graph.

For the special case of single-source reachability, here we had to return a shortest path distance for every vertex. And there was, at most, E things reachable from a vertex. So this is the bound we got. But in the special case for single-source reachability, when our output only has to list the vertices that are reachable from me, the number of things reachable in basically a spanning tree of the connected component of my source can almost be of order E. And so for all the little singleton vertices in my graph, I don't really care. So I can get this in order E, but that's kind of a little optimization.

The next thing we did was we talked about connected components. And we didn't just reduce to using a search algorithm like a single-source reachability algorithm like BFS or DFS. We put a for loop around that to explore the entire graph by basically saying, if I've explored one connected component, then I can look at any other vertex I haven't seen and explore the next one. And so that actually with some-- a little analysis, also got linear time, because I'm at most traversing any component of my graph once. That's kind of the idea.

And we can use that using BFS or DFS really, because we're just trying to get a thing that searches an entire connected component. And then this topological sort we did at the end of the last lecture. We used full DFS to give an ordering of the vertices in a DAG-- maybe I'll specify clearly that this is only for a DAG-- where we have an ordering of the vertices so all the edges go forward in that order, for example. And that we also did in linear time.

All right, in this lecture, and in actually the next four lectures, what we're going to do is instead of measuring distance in terms of the number of edges in a path-- so previously, distance equaled number of edges-- we're going to generalize that notion. So instead counting an edge, we're going to count an integer associated with that edge. It's going to be called a weight.

So here's an example of a weighted graph G. And I've labeled, in red, weights for each of these edges. This is a directed graph on eight vertices. And I've got an integer associated with each edge. You'll notice, some of them are positive, some of them are negative. It's OK to be zero as well. It's just any integer edge weight here.

So generally we're going to be-- along with our graph G, we're going to be given a weight function that maps the edges of G to, we're going to say, integers, in this class anyway. In other contexts, in mathematics, you might have these be real numbers. But in this class, we're going to deal with integers.

So each edge, if you have an edge, we're going to say this is the edge weight-- the weight of this edge e, from e. Sometimes, if this edge e is u, v, we might sometimes say the weight from u to v, since we have a simple graph that's unambiguous. All right, so but this is just talking about our notation. So in general, for example, the weight from vertex b to f in this graph is what? Can someone tell me?

AUDIENCE: Minus 4.

JASON KU: Minus 4, right? It's right here. And I'll be consistent with my coloring, because I've got colored chalk today. Minus 4. Happiness.

All right, so why do we care about adding weights to our graph? Well, this comes up a lot in many applications. For example, distances in a road network. if I have a road from here-- so from Mass Ave, front of MIT, to Central Square, we might think of that as one road. Maybe you've got each road is a connection between two intersections in my road network. But an edge, it takes longer to go from, say, Vassar Street to Amherst. That takes a shorter amount of time than it does to go from Memorial Drive across the river to Beacon Street. So we might want to associate a larger distance or a weight associated with that edge.

Latency in a network, for example. Maybe strength of relationships in a social network. And you could imagine that it's possible maybe you're "frenemies" with someone, you don't like them, and so maybe you have a negative weight associated with an edge in a social network. I'm not sure. Maybe not. But there are lots of applications where you might want weights on your edges.

So that comes to the next question of, how do I represent-- how do I give the user, or the algorithm, these weights in my graph? We had a representation for a graph. Our common way to represent a graph was store a set data structure on the vertices mapping to the adjacencies of each vertex, which we stored in what we called an adjacency list, which really could be any data structure. Commonly, it's just an array of the adjacencies. But you could also have that be a set data structure, where you can query in constant time what-- if a particular adjacency exists in that graph.

So there are two common ways to store these weights. One is just, with every adjacency, I'm going to store its weight. Maybe just in a tuple. With each adjacency, also store weight of the edge that it corresponds to, just in any way.

A second way, instead of trying to modify our graph structure that we gave you before, let's just have a dictionary of all the edges mapping to their weights. And we already know how to do that. Just any set data structure-- any separate set data structure mapping edges to their, I guess, weights. Bad notation, but you get the idea.

And it doesn't really matter how we're doing this. The assumption that we're going to rely on here is that, given an edge, given this vertex pair, I can query what the weight of that edge is in constant time. And so if I'm going to do that, I can either store it with maybe a hash table of hash tables-- a hash table mapping the set of vertices to their adjacencies, and then each adjacency list stores its adjacencies in a hash table. And that way, in constant time, I can check what the weight is there.

Or here, I'm just-- I could even have just a single hash table mapping the pair, the edge, the tuple, constant size, to its weight. So either way is fine. We're just going to assume that we can query an edge in constant time-- the weight of an edge in constant time.

OK, so this is that graph example. It's a little busy here. I'm probably going to erase that in just a second. But we're going to move on to what giving these edges weights implies for these problems that we've defined in terms of unweighted graphs. In particular, we are going to be concentrating on single-source shortest paths, again, at least for the next three lectures. We'll generalize that even still in the next lecture-- I mean, in the fourth-- in three lectures from now.

But what we had here was that the distance before in an unweighted graph was the number of edges in the path. Here, we're going to generalize that notion kind of obviously to weighted paths. And the weight of a path, I'm going call it pi. So some weight of path pi is just going to be the sum of the weights in the edges in the path. So edge in the path, I'm going to sum their weights. So that's all the weight of a path means. It's just I'm going to sum all the weights in a path.

So if I took a look at the-- maybe there's a particular path here. The path from a to b to f to g is going to be minus 5, minus 4, 2. It's going to be minus 9 plus 2 is minus 7. So just as an example.

So then what is the shortest path then? Well, kind of obviously among all paths between two vertices, it's going to be one with the minimum weight. Yeah, question.

AUDIENCE: Can I use the same edge more than once?

JASON KU: Can I use the same edge more than once? Right now, you're asking about the distinction in our class which we have between paths and simple paths. So here, a weighted path doesn't really care if we visit an edge more than once. So if an edge appears more than once in pi, we have to count that more than once in the edge weight-- in the weight of the path. OK, great question.

But what we're going to see later on is shortest paths cannot repeat an edge more than once in certain contexts. So we're going to get to the problem there a little later in this lecture. And we're going to solve that in tomorrow's lecture. But if you have-- we're getting a little ahead of ourselves. But when we have negative weights in a graph, it's possible that things go wrong. We're going to get there in about five lines. OK, great.

So a shortest path-- and in this case, I'm going to clarify that this is the weighted shortest path-- is a minimum-- min-i-mum-- sure-- is a minimum weight path from s to t. Nothing too interesting here, but there's actually some subtleties we have to deal with here.

We're going to call-- just like we did with breadth-first search when we talked about shortest paths, we're going to define an expression for what the distance or the shortest path weight is between two vertices. And I'm going to represent that by a delta. A delta from a vertex s to t is going to be-- let's-- I'm going to do the wrong thing first-- the minimum over the weight of all paths for all paths pi from s to t.

OK, so there's a couple things that go wrong here. First thing that goes wrong is the same thing that went wrong with breadth-first search. Anyone remember what could go wrong with breadth-first search for this delta definition?

AUDIENCE: Maybe there's no path.

JASON KU: Maybe there's no path, right. So except if no path. Just by convention, we're setting delta s, t, to equal infinity. But there's one additional problem with weighted shortest paths, and it's a little subtle. It's possible that a finite shortest-- finite length shortest path doesn't exist.

And what do I mean by that? It means I could keep going through edges in my graph and continually getting a shorter path. So if the shortest-- the minimum weight of a path from s to t actually goes through an infinite number of edges, then this isn't really well-defined.

So I'm going to change this minimum here to-- in mathematics we would, just to be specific, we call it an infimum. So if in the case where the weight of a shortest path can approach arbitrarily small, then we'll call this thing minus infinity. So when does that occur? When does that occur? When could we have our shortest path go through lots and lots of vertices?

Well, let's actually take a look at this example here. Can someone tell me what the shortest path is from a to actually any vertex in this graph?

AUDIENCE: b, f, g, c.

JASON KU: Ah, OK. So well, we could look at this path I have to b. Let's just take a look at b. I have a path going from a to b that is minus 5. OK, that's pretty good. That's pretty small. And it seems that if I go around this graph through another way, it might be bigger.

So I go 7 plus 3 plus 8-- that's 15-- minus 1-- that's 14. That's much bigger than minus 5, so it seems like minus 5 should be good, right? Anyone have a problem with this path or a problem with this being the shortest path?

And what your colleague just informed me was that there is something interesting happening here in this graph in particular. We have a cycle from b to f to g to c that has negative total weight back to b. This has minus 4 plus 2 plus 1 minus 1. So that total cycle has a cycle weight of minus 2, this negative weight cycle.

So if I want to get to b, I could go there via this minus 5 weight edge. But every time I circled around this cycle, I incur minus 2 to my path weight. So I just keep going around this cycle over and over and over and over and over and over again, and I don't have any finite length minimum weight path. And in such cases, we just say that delta is minus infinity.

So the problem here is that we could have negative weight cycles-- deserves a capital letter-- Negative weight cycles. It's a problem. In particular, if there exists a path from s to some vertex v that goes through a vertex on a negative weight cycle, then I can take that path to that vertex, circle around the negative weight cycle, and then proceed to v, and I can take that cycle as many times as I want. Then this delta s,v we're going to set to minus infinity.

And in such cases, in our shortest paths algorithm, we don't really care about what the shortest path is. We're not even going to deal with parent pointers here, because there is no finite length shortest path. So I'm just going to kind of throw up my hands in the air and say, you know what, I can't return you a shortest path, but I might want to return to you a negative weight cycle.

If you told me that this thing has bad weight, maybe I want you to tell me what a path is that goes through a negative weight cycle to get back to s. So that's what we're going to talk about next lecture. This lecture, we are not going to talk about that. We are going to talk about weighted shortest paths, though. That's what the remainder of this unit on graphs is really about is weighted shortest paths.

OK, so in weighted shortest paths, we actually know an algorithm already to solve a subset of weighted shortest parts, namely BFS, right? Now, you're like wait, Jason, BFS doesn't solve weighted shortest paths. We didn't even know about weighted graphs then. How does that solve weighted shortest paths?

Well, there's a couple cases where we might be able to reduce to solving shortest paths using BFS. Can anyone think of such a scenario? So let's say, I mean, kind of what we did before was we counted the number of edges. So if we gave a weight of 1 to every edge in my graph, then just that graph, that weighted graph, corresponds to an unweighted graph using the other distance metric. So in that case, BFS just solves our problem.

And in fact, we can generalize further. What if all of our weights were positive, but the same value? If it was all positive and the same value, then we could just divide by that value. Now we have an unweighted graph which we can run BFS, and then multiply shortest path distances by that value later on.

And in fact, there's one further generalization we can make, which is a little bit of a tricky graph transformation problem. But we can also get this linear time algorithm for weighted single-source shortest paths in contexts where the weights aren't that large.

So if I have positive edge weights-- if I have a positive edge weight, let's say-- using my weight color here-- that's, like, weight of 4, that's kind of problematic, because I don't know how to simulate that using an unweighted graph. Or do I? Anyone have an idea of how I could simulate an edge of weight 4 with an unweighted graph? Yeah.

AUDIENCE: Have four edges of weight 1.

JASON KU: Yeah, I can just put four edges of weight 1 in parallel here-- I'm sorry, in series, the opposite of parallel. I can just convert this here into 1, 2, 3, 4 edges. And if I do that for every edge in my graph and we have positive edge weights, then that transformation can hold. Now, that's not necessarily a good transformation to make. Why?

AUDIENCE: The weight might be very big.

JASON KU: Yeah, the weights might be very big compared to the number of vertices and edges in my graph. However, if the sum of all weights in my graph is asymptotically less than v plus e, we can get a linear time algorithm again by reducing to BFS. OK, so that's great.

But in general, that gives us a linear time algorithm in these very special cases. And in general, it's an open problem. We don't know whether we can solve the single-source shortest paths problem in the weighted context for general graphs in linear time. We don't know how to do it.

But what we do know are some algorithms that do pretty well. And that's what we use all the time. But one more special case we're going to go over today is when we have this really nice structure where we have a DAG, a Directed Acyclic Graph, like we were talking about in the last lecture.

For any set of edge weights-- remember, with BFS, we needed to restrict our edge weights to be positive and maybe bounded to get this good running time? For any set of edge weights, if our graph structure is DAG-- it really has nothing to do with the weights-- if the graph structure is a DAG, then we can actually solve this single-source shortest paths problem in linear time, which is pretty awesome.

Now, for general graphs, we're going to show you in the next lecture how to, for any graph-- even with cycles, even with negative weight cycles-- we're going to show you how to solve this single-source shortest paths problem in something like a quadratic running time bound. Now, this isn't the best known, but it's a really practical algorithm and people use it all the time. And we are going to show Bellman-Ford in the context of the DAG algorithm we're going to solve today.

So that's the very general case in terms of restrictions on our graph. But in reality, most problems that come up in applications occur with graphs that have positive edge weights. You can think of a road network. You've got-- or non-negative ones anyway. You're traveling along, and it's not ever useful to go back to where you came from, because you want to make progress to where you're going.

So in the context where you don't have negative weights, you don't have this problem where you have negative weight cycles. We can actually do a lot better by exploiting that property. And we get a bound that's a little bit-- that looks a little bit more like n log n. It's pretty close to linear. You're losing a log factor on the number of vertices. But it's pretty good. This is called Dijkstra, and we'll get to that in two lectures.

OK, so that's the roadmap of what we're going to do for at least the next three lectures. But before we go on to showing you how to solve single-source shortest paths in a DAG using this algorithm that I'm calling DAG relaxation here, I'm going to go back to a thing that we talked about in breadth-first search, where in breadth-first search when we solved single-source shortest paths, we output two things. We output single-source shortest paths, these deltas, for the other definition of distance, the weights-- I mean, not the weights, the distances, the shortest distances. But we also returned parent pointers. We return parent pointers back along paths to the source along shortest paths. We call this the shortest paths tree.

So I'm going to revisit this topic of shortest paths tree-- shortest path trees-- shortest path trees. And in particular, it's kind of going to be annoying to talk about both of these quantities-- distances and parent pointers-- as we go through all three of these algorithms. It's basically going to be bookkeeping to-- distances are actually sufficient for us to reconstruct parent pointers if we need them later.

So what I'm going to show for you-- prove to you now is that, if I give you the shortest path distances for the subset of the graph reachable from s that doesn't go through negative weight cycles, if I'm giving you those distances, I can reconstruct parent pointers along shortest paths in linear time for any graph I might give you if I give you those shortest path distances.

OK, so that's what I'm going to try to show to you now. So here's the algorithm. For weighted-- there's the caveat here I'm going to write down. For weighted shortest paths, only need parent pointers for v with finite shortest path distance-- only finite shortest path distance. We don't care about the infinite ones or the minus infinite ones, just the finite ones.

OK, so here's the algorithm. I can initialize all Pv to equal-- sorry, oh, getting ahead of myself. I'm writing down DAG. Init parent pointer data structure to be empty. At first, I'm not going to sort any parent pointers. But at the beginning, I'm going to set the parent pointer of the source to be none. So that's what we kind of did in breadth-first search as well.

Now, what I've given you is-- I'm trying to show that, given all the shortest path distances, I can construct these parent pointers correctly. So what I'm going to do is, for each vertex u in my graph, where my delta s of u is finite, what am I going to do? I'm going to say, well, let's take a look at all my outgoing neighbors. This is kind of what we do in every graph algorithm.

For each v in the adjacency, the outgoing adjacencies of u, if there is no parent pointer assigned to this v, there's the potential that i-- u-- [CHUCKLES] I, you-- this u, this vertex u, is the parent of v. It's possible. It's some incoming edge to v.

When will it be an incoming edge to v? If v not in P-- I haven't assigned it a parent pointer-- and-- so this means it could be my parent. When is it my parent along the shortest path? Sure.

AUDIENCE: Sum the distance along the edge to the distance of the other.

JASON KU: Yeah, so we have some edge from u to v. It has some weight. If I already know the shortest path distance to u, and I know the shortest path distance to v, if the shortest path distance from s to u-- let's draw a picture here. We've got s, we've got some path here to u, and we know we've got an edge from u to v. If this shortest path distance plus this edge weight is equal to the shortest path distance from s to v, then it better-- I mean, there may be more than one shortest path, but this is certainly a shortest path, so we can assign a parent pointer back to u.

So let's write that condition down. If the shortest path distance from s to v equals the shortest path distance from s to u, and then traversing the edge from u to v, then exists shortest path that uses edge u, v, in particular this one. So set the parent of u-- of v to u.

OK, so this is the algorithm. I'm not going to prove to you that this is correct. But it kind of intuitively makes sense, right? If I have these shortest path distances, you can prove by induction that not only does this parent pointer point to the right place along some shortest path here, but it also does so in linear time, because I'm looping over all the vertices and looping over its outgoing adjacencies once. Same analysis as we had for both BFS and DFS, essentially.

And then, since we can do this, since we can compute parent pointers from these distances, we're going to ignore computing these parent pointers from now on. We're just going to concentrate on computing the distances, because we're going to have to take linear time anyway at least. And all these other things take more time. So we can compute the distances in more time, and then compute the parents after. OK, so that's what we're going to do.

So now, with all that buildup, let's show an algorithm. [CHUCKLES] How do we compute single-source shortest paths in a DAG in linear time? Well, a DAG-- I mean, this is actually a super useful, convenient thing in algorithms in general. DAGs are just nice things. They're kind of ordered in a way. There's this topological sort order that we were talking about before. This is going to play a key role.

There's a really nice structure to DAGs not having cycles, not having to deal with this negative weight cycle problem. You can only go in one direction along this graph. It's a very nice structure to exploit. And so we're going to exploit it.

And here's the idea. DAG relaxation, what it's going to do is it's going to start out with some estimates of what these distances should be. So maintain distance estimates. And now I'm going to try to be careful here about how I draw my Ds. This is a d, this is a delta. This is shortest paths. This is a distance estimate. So that's what I'm going to be using for the rest of this time.

So we're going to maintain these estimates of distance d, which are going to start at initially infinite. I don't know what they are. I don't know what the shortest paths are, but they better be less than infinite or else I don't care. So that's the worst case scenario. It can't be worse than this-- for every vertex. And we're going to maintain the property that estimates upper bound-- that should probably be two words-- upper bound delta s, v-- we're going to maintain that they upper-bound this thing and gradually lower until they're equal.

So this is the idea. We start from an over-estimate, an upper bound on the distance estimate. And then we're repeatedly going to lower that value as we gain more information about the graph, maintaining that we're always upper-bounding the distance. And we're going to keep doing it, keep doing it, keep doing it, until, as we will try to prove to you, these estimates reach, actually reach down, all the way to our shortest path distances.

So when do we lower these things? When do we lower these things? We are going to lower these distance estimates whenever the distance estimates violate what we're going to call the triangle inequality. OK, what is the triangle inequality?

Triangle inequality is actually a pretty intuitive notion. It's basically saying, if I have three points-- thus, triangle-- maybe bigger so I can write a letter in them. It's basically saying that if I have a vertex u, a vertex v, vertex x, for example, the shortest path distance-- the shortest path distance delta of u, v-- that's the shortest distance from u to v-- it can't be bigger then a shortest path from u to v that also goes through x.

Of my paths, I'm now restricting the paths I have to the ones that go through x. The shortest path distance from u to v can't be bigger than restricting paths that go through x and taking that shortest distance, getting the shortest path distance from here and adding it to the shortest path distance here-- delta u, x, delta x, v. That's just a statement of, I'm restricting to a subset of the paths. I can't decrease my minimum distance.

So this is the statement of the triangle inequality, that the shortest path distance from u to v can't be bigger than the shortest path distance from u to x plus the shortest path distance from x to v for any x in my graph that's not u and v. So that's the triangle inequality. Pretty intuitive notion, right?

Why is this useful? OK, well, if I find-- if I find an edge in my graph, if there's an edge u, v, in my graph such that this condition is violated for the estimates that I have-- it obviously can't be violated on my shortest path distances, but if it violates it on the estimates-- u, v, is bigger than u, x-- sorry, u-- how am I going to do this?

I want this to be s. I'm calculating shortest path distances from s and shortest path distances from s to some incoming vertex u plus the edge weight from u to v. All right, so what is this doing?

I have some edge u, v in my graph. Basically, what I've said is that I have some distance estimate to u, but going through-- making a path to v by going through u, and then the edge from u to v is better than my current estimate, my shortest path estimate to v. That's bad, right? That's violating the triangle inequality. These cannot be the right weights. These cannot be the right distances.

So how we're going to do that is lower-- this is what we said, repeatedly lower these distance estimates. I'm going to lower this guy down to equal this thing. In a sense, this constraint was violated. But now we're relaxing that constraint so that this is no longer violated.

So relax is a little weird word here. We're using it for historical reasons. But that's what we mean by when we say relax. This thing is a violated constraint. It's got some pressure to be resolved. And so what we're doing is, to resolve it, we're just setting this guy equal to this, so it at least resolves, locally, that constraint. Now, it may violate the triangle inequality other places now that we've done this change. But at least this constraint is now relaxed and satisfied.

OK, so relax edge by lowering d of s, v, to this thing. That's what we're going to mean by relaxing an edge. And relaxing an edge is what I'll call safe. It's safe to do. What do I mean by relaxation is safe? It means that as I am computing these shortest path distances, I'm going to maintain this property that each one of these estimates-- sorry, these estimates here-- has the property that it's either infinite or it is the weight of some path to v. So that's the thing the-- relaxation is safe.

OK, so each distance estimate, s, v, is weight of some path from s to v or infinite. And this is a pretty easy thing to prove. If I had the invariant that these were all weights of shortest paths, let's try to relax an edge. And we need to show that this property is maintained. Relax edge u, v, OK?

Now, if I relax edge u, v, what do I do? I set this thing-- or, sorry, I set this thing, my shortest path distance to v, to be this thing plus this thing. There's a weight of an edge from u to v.

Now, by my assumption that we're maintaining that this is the weight of some path in my graph, if this thing is bigger, I'm setting it to the weight of some path on my graph to u, plus an edge from u to v, and so this checks out. So assign d of s, v, to weight of some path.

I'm not going to write down all the argument that I just had here. But basically, since this distance estimate was by supposition before the weight of some path to v-- to u, then this is, again, the weight of some path to v. OK, great. So now we're ready to actually go through this algorithm.

So DAG relaxation, from over there, initializes all of our distance estimates to equal infinity, just like we did in BFS. Then, set my distance estimate to myself to be 0. Now, it's possible that this might be minus infinity or negative at some point. But right now, I'm just setting it to 0. And either way, 0 is going to upper-bound the distance to s.

So in particular, at initialization, anything not reachable from s is set correctly. And s itself is set as an upper bound to the shortest path distance. Now we're going to process each vertex u in a topological sort order. So remember, our input to DAG relaxation is a DAG. So this thing has a topological sort order. We're going to process these vertices in that order.

You can imagine we're starting at the top, and all my vertices are-- all my edges are pointed away from me. And I'm just processing each vertex down this topological sort line. And then for each of these vertices, what am I going to do? I'm going to look at all the outgoing neighbors. And if the triangle inequality is violated, I'm going to relax that edge. The algorithm is as simple as that.

For each outgoing neighbor of v-- sorry, of u-- I always get u and v mixed up here. If my shortest path estimate to v violates the triangle inequality for an edge, for an incoming edge, then I'm going to set-- relax u, v, i.e. set d, s,v, equal to d, s,u, plus w, u,v. So that's the algorithm.

So if I were to take a look at this example graph over here, maybe a is my start vertex. I initialize it to--

AUDIENCE: DAG.

JASON KU: This is not a DAG. Thank you. Let's make it a DAG. I claim to you now this is a DAG. In particular, a topological sort order is when there's a path through all the vertices, then there's a unique topological sort order-- a, b, e, f, g, h, d, c. This is a topological order. You can check all the edges.

So I'm going to start by setting the-- actually, let's use e. Let's use shortest paths from e. Why not? Shortest paths from e.

Vertex a actually comes before e in the topological order. So it has no-- I mean, its shortest path distance, when I initialize, I'm going to initialize this to 0. I'm going to initialize this to infinite, infinite, infinite, infinite, all these things to infinite. These are my estimates. These are not quite the shortest paths yet-- distances.

But when I get here, clearly I can't be-- distance to me being infinite can never violate the triangle inequality with something infinite or finite. It doesn't matter, right? So I don't do anything to a. Anything before my source vertex in the topological order can't be visited, because it's before in the topological order. That's kind of the point. There's no path from my source vertex to anything before it in the topological order.

So same with b. b is before a in the topological order. Now, I'm at e, and it's possible we are violating triangle inequality, in particular here. I think the shortest path distance to f is infinite. But actually, if I go to e through this edge with a weight 3, I know that this is violating triangle inequality. So actually, this thing is wrong, and I can set it equal to 3.

Now, that might not be the shortest path distance. But right now it's a better estimate, so we've set it.

Now, I'm moving on. I'm done with this guy. I move to the next vertex in my topological order. And again, I relax edges out of f. OK, so here, looking at 8, 3 plus 8 is better than infinite, so we'll say that that's 11. And 3 plus 2 is better than infinite, so that's 5. Now, I keep going in the topological order. g is the next. 5 plus 1 is 6. OK, so we found a better estimate here. So this 11 is not as good. 6 is better, so we replace it.

Here, we haven't visited before. It's still infinite. So 5 plus minus 2 is 3. This is the next in the topological order. 3 plus 9 is bigger than 6, so that's not a shorter path. 3 plus 4 is certainly smaller than infinite, so set that equal to 7. Then, 7 plus 5 is also bigger than 6. And actually, you can confirm that these are all the shortest path distances from e.

So this algorithm seems to work. Does it actually work? Let's take a look. The claim to you is that at the end of relaxation, this algorithm, we've set-- claim at end, all the estimates equal shortest path distances. The basic idea here is that if I take the k-th vertex in the topological order, assuming that these distances are all equal for the ones before me in the topological order, I can prove by induction.

We can consider a shortest path from s to v, the k-th vertex, and look at the vertex preceding me along the shortest path. That vertex better be before me in the topological order or we're not a DAG. And we've already set its shortest path distance to be equal to the correct thing by induction.

So then when we processed u-- s to u to v-- when we processed u in DAG relaxation here, processed the vertex and looked at all its outgoing adjacencies, we would have relaxed this edge to be no greater than that shortest path distance. So this is correct.

You can also think of it as the DAG relaxation algorithm for each vertex looks all at its incoming neighbors, assuming that their shortest path distances are computed correctly already. Any shortest path distance to me needs to be composed of a shortest path distance to one of my incoming neighbors through an edge to me, so I can just check all of them. That's what DAG relaxation does.

And again, we're looping over every vertex and looking at its adjacencies doing constant work. This, again, takes linear time. OK, so that's shortest paths and a DAG. Next time, we'll look at, for general graphs, how we can use kind of the same technique in an algorithm we call Bellman-Ford. OK, that's it for today.