**PROFESSOR:** Good morning, everyone. So lecture three of four in the shortest path module and today we'll finally confront our nemesis, which are negative cycles and negative edges. And we will describe an algorithm that is due to two different people. They didn't collaborate to produce this algorithm.

Bellman and Ford. This computes shortest paths in a graph with negative edges. And not only that, even in the graph has negative cycles in it, the algorithm will be correct in the sense that it will report the existence of a negative cycle and, essentially, abort the computation of shortest paths that are undefined.

And for the few vertices that do not have negative cycles in between them and the source, the algorithm will report correct shortest paths. So it is a polynomial time algorithm. It's fairly easy to describe. And what we'll do is describe it, analyze its complexity and, for once, we'll do a formal proof of its correctness to show that it reports the existence of negative cycles if they do exist. And if they don't exist, it correctly computes shortest path weights.

So recall that when we look at the general case of the shortest path problem. We're going to have, let's say, a vertex u that, in this case, happens to be our source. And let's say for argument's sake that we have a negative weight cycle like so. So let me to draw this in bold. And this happens to be a negative rate cycle.

Let's assume that all of these edges have positive weights. Then, if you have an algorithm that needs to work on this type of graph, what you want to be able to do is to detect that this negative cycle exists. And you're going to, essentially, say if this vertex is v1, for example, you want to be able to say delta u v1 is undefined and similarly for v2, v3, et cetera.

For all of these things, the shortest path lengths are undefined because you can essentially run through this negative cycle any number of times and get whatever shortest path weight you want. For this node, let's call that v0, we have delta u v0 equals 2. And there's a simple path of length 1 in this case that gets you from u to v0.

You don't encounter a cycle or negative cycle in between. So that's cool. All right? And of course, if you have a vertex over here, z, that can't be reached from u then we're going to have delta uz being infinity.

And you can assume at the beginning of these algorithms that the source-- in this case, I call the source u-- but the shortest path to u would be 0. And all of the other ones are infinity. And some of them may stay infinity. Some of them may obtain finite shortest path weights. And some of them will be undefined if you have a graph with negative cycles in it.

So that's sort of the specification, if you will, of the requirements on the Bellman-Ford algorithm. We want it to be able to do all of the things I just described. OK? So let's take a second look at our generic shortest path algorithm that I put up, I think, about a week ago. And this is a good review of our notation.

But there are a couple more things I want to say about this algorithm that I didn't get to last time. So you're given a graph and you set all of the vertices in the graph to have infinite shortest path weights, initially. Set the predecessors to be null. And then we'll set d of s to be 0. That's your source. And the main loop would be something like repeat, select, and edge.

And we have a particular way of selecting this edge. And we have positive edge weights that corresponds to the minimum priority. And we talked about Dijkstra but we have, maybe, different ways of doing that. We have to select an edge somehow. And then, we relaxed that edge. u, v, w.

And you know about the relaxation step. I won't bother writing it out right now. But it's basically something where you look at the value of d v. And if d v is greater than

d u plus the weight, you relax the edge. And you keep doing this. The other thing that you do in the relaxation is to set the predecessor pointers to be correct. And that's part of the relax routine.

And you keep doing this until you can't relax anymore. All right? So that's our generic shortest path algorithm. There are two problems with this algorithm. The first, which we talked about and both of these have to do with the complexity but the first one is that the complexity could be exponential time, even for positive edge weights.

And the particular example we talked about was something where you had an exponential number of paths. And if you had a graph that looks like this, then it's possible that a pathological selection of edges is going to make you relax edges an exponential number of times. And in particular, if you have n nodes in this graph, it's plausible that you'd end up getting the complexity of order 2 raised to n over 2. OK? So that's one problem.

The second problem, which is actually a more obvious problem, is that this algorithm might not even terminate if this-- actually will not terminate the way it's written if there's a negative weight cycle reachable from the source.

All right, so there's two problems. We fixed the first one. In the case of positive edges are non-negative edges. We have a neat algorithm that is an efficient algorithm called Dijkstra that we talked about last time that fixed the first part. But we don't know yet how we're going to handle negative cycles in the general case. We know how to handle negative edges in the case of a DAG-- a directed acyclic graph-- but not in the general case. OK?

So there's this great little skit from Saturday Night Live from the 1980s-- so way before your time-- called The Five Minute University. Anybody seen this? All right. Look it up on YouTube. Don't look it up during lecture but afterwards.

So the character here is a person by the name of-- I forget his real name but his fake name is Father Guido Sarducci. All right? So what's this Five Minute University

about? Five Minute University, he's selling this notion and he says, look, five years after you graduate you, essentially, are going to remember nothing. OK?

I mean, you're not going to remember anything about all the courses you took, et cetera. So why waste your time on a college education or waste money-- $100,000-- on a college education? You know, for $20 I'll teach you in five minutes what you're going to remember five years after you graduate. All right?

So let's take it to an extreme. Here's a 30 second version up 6006. And this is what I want you to remember five years or 10 years or whatever after you graduate. All right? And maybe the 10 second version as polynomial time is great. OK? Exponential time is bad. And infinite time gets you fired. OK?

So that's all you need to remember. No, that's all you need to remember for the final. This happens, you know, five years after you graduate. So you need to remember a lot more if you want to take your quiz next week and the final exam.

But I think that summarized over here. You have a generic shortest path algorithm. And you realize that if you do this wrong you could very easily get into a situation where a polynomial time algorithm, and we know one for Dijkstra, turns into exponential time in the worst case, you know, for a graph like that because you're selecting edges wrongly.

And in particular, that's problem number one. And problem number two is if you have a graph that isn't what you expect. In this case, let's say you expected that a graph with no negative cycles or maybe not even negative edges in it. You could easily get into a situation where your termination condition is such that your algorithm never completes.

So we need to fix problem number two today using this algorithm called Bellman-Ford. And as it turns out, this algorithm is incredibly straightforward. I mean, its complexity we'll have to look at. But from a description standpoint, it's four lines of code.

And let me put that up. So Bellman-Ford takes a graph, weights, and a source s.

And you can assume an adjacency list specification of the graph or the representation of the graph.

And we do some initialization. It's exactly the same as in the generic case except the d values will still be looking at the d values and talking about the relaxation operation. So we do an initialization. And then, this algorithm has multiple passes because for I equals 1 to v minus 1. So it does v minus 1 passes roughly order v passes where v is the number of vertices.

And in each of these passes for each edge u v belonging to e relaxes every edge. And just so everyone remembers, relax u, v, w is if d of v is greater than d of u plus w u v then we'll set d v to be-- and we also set pi v to be u. OK.

That's relax operation over here. So that's the algorithm. And if you know magically that they're no negative cycles in the graph. So if they're no negative cycles in the graph, then after these-- we'll have to prove this. But after these v minus 1 passes you're going to get the correct shortest pathways. OK?

You want to do a little bit more, right? I motivated what we want Bellman-Ford to do earlier in the lecture. So you can also do a check. So you may not know if they're negative weight cycles or not. But at this point, you can say I'm going to do one more pass so the v path-- the v is the number of vertices-- over the graph.

So for each edge in the graph, if you do one more relaxation and you see that d v is greater than d u plus w u v. So you're not doing a relaxation. You're doing a check to see if you can relax the edge. Then report minus v negative cycle exists.

So this is the check. And the first part is the computation. So that's kind of neat. I mean, it fit's on a board. We talk about the correctness. The functionality, I hope everyone got. Do people understand what's happening here with respect to functionality? Any questions? Not about correctness but functionality? Yeah?

**AUDIENCE:** Where does the [INAUDIBLE] get used in the formula?

**PROFESSOR:** Oh, it doesn't. It's just a counter that makes sure that you do v minus 1 passes. So

what's that complexity of this algorithm using the best data structure that we can think of? Anyone? Yeah, go ahead.

**AUDIENCE:** [INAUDIBLE] v plus e if you're using a [INAUDIBLE] to access [INAUDIBLE]?

**PROFESSOR:** v plus e?

**AUDIENCE:** Or v e plus e.

**PROFESSOR:** So that would be?

**AUDIENCE:** That's using a dictionary?

**PROFESSOR:** Yeah, I know. v e plus e would be? That's correct but.

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** Right. But I mean when do v e plus e you can ignore the e. So say you have just v times e. All right. Good. Here you go.

So this part here is v times e. And it doesn't really matter. I mean, you can use an array structure adjacency list. It's not like Dijkstra where we have this neat requirement for a priority queue and there's different ways of implementing the priority queue. This part would be order of v e. And that gives you the overall complexity.

This part here is only one pass through the edges. So that's order e, like you said. So the complexities order v e. And this could be large, as I said before in, I think, the first lecture. e is order of v square in a simple graph. So you might end up with a v cubed complexity if you run Bellman-Ford.

So there's no question that Bellman-Ford is, from a practical standpoint, substantially slower than Dijkstra. You can get Dijkstra down to linear complexity. But this would potentially, at least in terms of vertices, be cubic complexity.

So when you have a chance, you want to use Dijkstra. And you're forced to use Bellman-Ford because you could potentially have negative weight cycles while

you're stuck with that. All right? OK, so why does this work? This looks a bit like magic. It turns out we can actually do a fairly straightforward proof of correctness of Bellman-Ford.

And we're going to do two things. We're going to not only show that if negative weight cycles don't exist that this will correctly compute shorter stats. But we also have to show that it will detect negative weight cycles if they in fact exist. So there's two parts to this. And let's start.

So what we have here for this algorithm is that it can guarantee in a graph g equals v E. If it contains no negative weight cycles then after Bellman-Ford finishes execution, d v equals delta s v for all v belonging to v. All right?

And then there's that. That's the theorem you want to prove. And the second piece of it is corollary that we want to prove. And that has to do with the check. And this says if a value of d of v fails to converge after v minus 1 passes there exists a negative weight cycle reachable from s.

So those are the two things that we need to show. I'll probably take a few minutes to do each of these. That theorem is a little more involved. So one of the first things that we have to do in order to prove this theorem is to think about exactly what the shortest path corresponds to in a generic sense.

So when we have source vertex s and you have a particular vertex v then there's the picture that we need to keep in mind as we try and prove this theorem. So you have v0, v1, v2, et cetera all the way to vk.

This is my vertex v. This is s. So s equals v0. V equals vk. All right? So I'm going to have a path p. That is v0, v1, all the way to vk. OK?

How big is k in the worst case? How big is k? Anybody? How big is k? It's up on the black board.

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** v minus 1, right? Why? What would happen if k is larger than v minus 1? I'd have a

cycle. I'd be visiting a vertex more than once. And it wouldn't be a simple path. Right?

So k is less than or equal to v minus 1 else I'd have a cycle. OK? I wouldn't have a simple path. And we're looking for the shortest, simple paths because if you ever get to the point where-- why are we looking for shortest, simple paths?

Well, in this case, we're looking for shortest, simple paths. And if there's a negative cycle, we're in trouble because the shortest path is not necessarily the simple path because you could go around the cycle a bunch of times. I'll get back to that.

But in the case where we're trying to prove the theorem, we know that no negative cycles exist. We can assume that no negative cycles exist for the case of the theorem. And we want to show that Bellman-Ford correctly computes each of the shortest path weights. And in that case, there's no negative weight cycles.

We're guaranteed that k is less than or equal to v minus 1. All right? Everybody buy that? Good. All right. So that's the picture I want you keep in mind. Let's dive in and prove this theorem. And we prove it using induction.

So let v be any vertex. And let's say that we're looking at a path. v0, v1, v2, to vk. And like I said, from v0 equals s to vk equals v. And in particular, I'm not going to say that this path p is a shortest path with the minimum number of edges.

So there may be many shortest paths. And I'm going to pick the one that has the minimum number of edges. If there's a unique shortest path, then that's a given. But it may be that I have a path with four edges that has the same weight as another path with three edges. I'm going to pick the one that has three edges. OK?

So it may not be unique with respect that they're not necessarily unique shortest paths. But I can certainly pick one. And no negative weight cycles implies that p is simple. And that implies that k is less than or equal to v minus 1, which is what I just argued.

Now keep in mind that picture over there to the right. And basically, the argument is

going to go as follows. Remember that I'm going to be relaxing every edge in each pass of the algorithm. OK? There's no choices here. I'm going be relaxing every edge in each pass of the algorithm.

And essentially, the proof goes as follows. I'm going to be moving closer and closer to vk and constructing this shortest path at every pass. So at some point in the first pass, I'm going to relax this edge v0, v1. OK? And at that point, thanks to the optimal substructure property, given that this is the shortest path, this has to be a shortest path, as well.

Any subset of the shortest path has to be a shortest path. I'm going to relax this edge and I'm going to get the value of delta from s to v1. And it's going to be this relaxation that's going to get me that value.

And after the first pass, I'm going to be able to get to v1. After the second pass, I can get to v2. And after k passes, I'm going to be able to get to vk. So I'm just growing this frontier one node every pass. And that's your induction. And you can write that out. And I'll write it out here. But that's basically it.

So after one pass through all of the edges e, we have d of v1 to be delta s v1. And the reason for this is because we'll relax. We're guaranteed to relax all the edges. And we'll relax the edge v0, v1 during this pass.

And we can't find a shorter path than this path because, otherwise we'd violate the optimum substructure property. And that means that it's a contradiction that we selected a shortest path in the first place. So can argue that we have delta s v1 after the first pass.

And this goes on. I'm going to write out this proof because I think it's important for you guys to see the full proof. But you can probably guess the rest at this point. After one pass, that's what you get. After two passes through e we have d v2 equals delta s v2 because in the second pass we're going to relax edge v1, v2.

So it' a different edge that needs to be relaxed. But that's cool because I'm relaxing all the edges. And I'm going to be able to grow my frontier. I'm going to be able to

compute delta s v2 and the end of my second pass and so on and so forth.

So after k passes, we have d vk equals delta s vk. And if I run through v minus 1 passes, which is what I do in the algorithm, all reachable vertices have delta values. All right? That's basically it. Any questions?

It's actually a simpler proof than the Dijkstra proof, which I just sketched last time. I'll just give you some intuition of the Dijkstra proof. It's probably a little too painful to do in a lecture. But this one is, as you can see, nice and clean and fits on two boards, which is kind of an important criterion here. So good.

All right, so that takes care of the theorem. Hopefully you're all on board with the theorem. And one thing that we haven't done is talk about the check. So the argument with respect to the corollary bootstraps this particular argument for the theorem. But this requires the insight that if after v minus 1 passes, if you can find an edge that can be relaxed, well what does that mean?

So at this point, let's say that I've done my v minus 1 passes and we find an edge that can be relaxed. Well, this means that the current shortest path from s to some vertex that is obviously reachable v is not simple once I've relaxed this edge because I have a repeated vertex.

So that means it's not simple to have a repeated vertex that's the same as I found a cycle. And it's a negative weight cycle because I was able to relax the edge and reduce the weight after I added a vertex that cost a cycle. All right? So this cycle has to be negative weight. Found a cycle that is negative weight.

All right. That's pretty much it. So it's, I guess, a painful algorithm from a standpoint of it's not particularly smart. It's just relaxing all of the edges a certain fixed number of times.

And it just works out because you will find these cycles. And if you keep going, it's like this termination condition. What is neat is that I don't have the generic shortest path algorithm up there anymore. But in effect, what you're saying is after a certain

number of passes, if you haven't finished, you can quit because you have found a negative cycle.

So it's very similar to the generic shortest path algorithm. You're not really selecting the edges. You're selecting all of them, in this case. And you're running through a bunch of different passes. All right?

So that's it with respect to Bellman-Ford. I want to do a couple of special cases and revisit the directed acyclic graph. But stop me here if you have any questions about Bellman-Ford. You first and then back there. Yeah?

**AUDIENCE:** Maybe I'm just confused about the definition of a cycle. But if you had, like, a tree, which had a negative weight edge, wouldn't it produce the same situation where you relaxed that edge.

**PROFESSOR:** But you would have relaxed that edge previously.

**AUDIENCE:** But it wouldn't be a cycle, right?

**PROFESSOR:** Yeah, it wouldn't be a cycle. So let's look at that. That's a fine question.

**AUDIENCE:** Doesn't that make assumptions about this structure?

**PROFESSOR:** Well if you had a tree-- I mean, a tree is a really simple case. But if you had something like this and if you did have a minus 1 edge here, right-- we'll do a more complicated example. But let's say you had something like this. 2 3 minus 1.

And what will happen is if this happens to be your s vertex and in the first step you relax all the edges. And this one would get two. And then, depending on the order in which you relaxed, it's quite possible that if you relax this edge first-- let's say in the first pass the ordering of the relaxation is 1, 2, and 3.

So the edges are ordered in a certain way each time, and you're going to be relaxing the edges in exactly the same order each time. All right? It doesn't matter. The beauty of Bellman-Ford is that-- let's say you relax this edge. Initially, this is at infinity. So this is at 0. This is at infinity. This is at infinity. This is at infinity.

If you relax this edge, nothing happens. All right? Then you relax, let's say, this edge because that's number two. This gets set to two. You relax this edge because that's 3. And this is infinity so nothing happens.

Of course, this is already at two so nothing would happen. So the end of the first pass, what you have is this is 0. That's 2. This is still infinity. That's still infinity. OK? That's going to stay infinity because you can't reach it from s. So we can, sort of, ignore that.

And then, of the second pass, what you have is you start with this edge again because that's the ordering. And this 2 minus 1 would give this a 1. And then you relax this edge or try to relax this edge. Nothing happens. Try to relax this edge. Nothing happens.

And at this point, you have one more pass to go because you got 4 vertices. And in that past, nothing changes again. So that's what you end up with. You end up with 2 for this and 1 for that. OK? That makes sense?

So the important thing to understand is that you are actually relaxing all of the edges in every pass. And there's a slightly more complicated example than this that is in the notes. And you can take a look at that offline. There's another question in the back. Did you have a question? Someone raised their hand. Yeah?

**AUDIENCE:** Yes, I'm just curious-- is there a unknown better algorithm that can do the same thing?

**PROFESSOR:** No, there's no known better algorithm for solving the general case like this. There are a couple of algorithms that assume weights within a certain range. And then there complexities include both v and e, as well as w where w is the dynamic range of the weights.

And depending on what w is, you could argue that they have better complexity. But they're kind of incomparable in the sense that they have this extra parameter, which is the dynamic range of the w. OK? Now there's lots of special cases, like I said, and

well take a look at the DAG special case in a second where you could imagine doing better but not for the case where you have an arbitrary graph that could have negative cycles in it because it's got negative rate edges. Yeah?

**AUDIENCE:** In the corollary, does that assume you have a connected graph because, you know, you could have a negative weight edge in a separate part of the graph, which isn't reachable from this.

**PROFESSOR:** Yeah. So you're going to start when you have an undefined weight. Remember your initialization condition. What is affected by s? Initialize is affected by s. The rest of it isn't affected by s because you're just relaxing the edges.

Initialize is affected by s because d of s starts out being 0, like I put over here, and the rest of them are infinity. So there is an effect of the choice of the starting vertex. And the rest of it follows that you will get an undefined value, or you will find that negative cycle exists based on whether you can reach it from s or not.

So if you happen to have s over here, and it's just the one node, and then it has no edges going out of it, this algorithm would just be trivial. But it wouldn't detect any negative cycles that aren't reachable from s. That make sense?

**AUDIENCE:** Yeah.

**PROFESSOR:** So there is this-- it's kind of hidden over there. So I'm glad you asked that question. But initialize is setting things up. And that is something that affects the rest of the algorithm because d of s is 0 and the rest of them are set to infinity. All right?

So if there are no other questions, I'll move on to the case of the DAG and talk a little bit about shortest paths versus longest paths. And this is somewhat of a preview of a lecture that Eric is going to give a month from now on complexity and the difference between polynomial time and exponential time, though I'm not going to go into much depth here.

But there's some interesting relationships between the shortest path problem and the longest path problem that I'd like to get to. But any other questions on this? OK,

so let me ask a question.

Suppose I wanted to find longest paths in a graph and let's say that this graph had all positive edge weights. OK. What if I negated all of the edge weights and ran a Bellman-Ford? Would I find the longest path in the graph? Do people understand the question? I don't need this.

So maybe we can talk about what a longest path means first. So if this was s and this v1, v2, v3, fairly straightforward, you know how to compute shortest paths now. These are all positive. Even easier.

The longest path to v3 is of length. Six because I go here, go there, and go there, right? So that's my longest path. OK? And the shortest path to v3 is of length 4.

So shortest path, longest paths, have these nice duality. What if I said, well, you know, I can solve the longest path problem, as well, given all of what I've learned about shortest paths simply by negating each of these edges and running Bellman-Ford. What would happen? Yeah?

**AUDIENCE:** [INAUDIBLE] shortest path branch [INAUDIBLE] values, and if you switched to absolute value, it will give you the longest path.

**PROFESSOR:** So you think it works?

**AUDIENCE:** Yeah. It will also check the cycles. So the negative cycles will be the longest path cycles that [INAUDIBLE].

**PROFESSOR:** But I think that's the key question. What will Bellman-Ford do when it is run on this? What would it return?

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** No, what will Bellman-Ford return? I'm asking. Someone else? What will Bellman-Ford return if I ran this? Undefined. Right? Undefined because you got this negative weight cycle here.

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** Sorry? Oh! Let's put another one in there. Oops, sorry. Now I see. You're right. You're right. I'm wrong. And why did you say undefined?

**AUDIENCE:** I was wrong.

**PROFESSOR:** OK, good. I got company. Thank you. Thank you. Good. Let's take it all over again. All over again. All right. All right, start over. s v1 v2 v3. Yeah, that is a cycle. All right, good. Cycle.

So when you actually negate each of these edges, you end up with a negative weight cycle. So it's plausible that you could have a graph like this one where this strategy won't work because what would happen is Bellman-Ford would come back with, essentially, an abort that says I can't compute shortest paths because they're undefined. All right?

Now it turns out it's actually more subtle than that. What we're trying to do in Bellman-Ford is, in the case where negative weight cycles don't exist, we report on the shortest simple path. That's the whole notion of the proof.

We say that the path has a certain length, which is, at most, v minus 1 and so on and so forth. We get the shortest simple path. But if you actually have a problem where you say-- let me start over again.

Let's say I want to find the shortest simple path for a different graph and it happens to have a negative weight cycle in it. So I have something like this. 2 3 minus 6, 3 over here, 3 over here, and so on. Maybe 2 here. And I want to find the shortest simple path that reaches v from s. OK?

What is the shortest simple path that reaches v from s? It's this path that goes horizontally, which has a weight 3 plus 2, 5, 5 plus 3, 8, 8 plus 3, 11, 11 plus 2, 13. All right? So the shortest simple path is 13. Will Bellman-Ford give you any information about this path?

**AUDIENCE:** [INAUDIBLE].

**PROFESSOR:** No because in [INAUDIBLE]. After it does its v minus 1 passes, v is reachable from s. But you potentially go through a negative weight cycle before you reach v. OK? So it turns out that if you have a graph with negative weight cycles, finding the shortest simple path is an NP-hard problem. It's a really hard problem. That's what NP means.

No, it means something else that Eric will explain to you in a month or so. But it means that we don't know any algorithm that is better than exponential time to solve this problem. OK? So amazingly, all you've done is taken the shortest path problem and changed it ever so slightly.

You said I want to look for the shortest simple path in the general case where I could, potentially, have negative weight cycles in my graph. And when you do that, all bets are off. You're not in the polynomial time complexity domain anymore. At least, not that we know of. And the best that you can do is an exponential time algorithm to find shorter simple paths.

And this problem, as it turns out, is equivalent to the longest path problem in the sense that they're both NP-hard. If you can solve one, you could solve the other. So to summarize, what happens here simply is that in the case of Bellman-Ford running on the original shortest path problem, you're allowed to abort when you detect the fact that there's a negative cycle.

So given that you're allowed to abort when there's a negative cycle, you have a polynomial time solution using Bellman-Ford that is not necessarily going to give you shortest path weights but will in the case of no negative cycles. All right? But if you ask for more-- a little bit more-- you said, you know, it'd be great if you could somehow process these negative cycles and tell me that if I had a simple path and I don't go through cycles what would the shortest weight be, it becomes a much more difficult problem.

It goes from order of ve complexity to exponential time complexity to the best of our knowledge. So that's what I'd like to leave you with. That there's much more to

algorithms than just the ones that we're looking at. And we get a little bit of a preview of this-- so the difference between polynomial time an exponential time-- later on in the term.