PROFESSOR: OK, so up until now, we've been talking about the shortest path in graphs. And we've been talking about these game problems where you came up with a way to represent the state of a game as noting graphs. And then we drew edges between those vertices that represented states, to represent moves that you can do in a game.

It turns out you can solve a lot of problems this way. And you can do it without thinking of a graph at all. So we usually think of a graph, build the graph, then run dynamic programming. You can do better without building the graph.

And the good part about that is that your code is smaller, and it runs faster. The bad part about that is it's harder to understand. The dynamic programming code looks like black magic.

And, when we're going to give you dynamic programming problems on the final, you can't say, oh, we're going to build something and run [? dystring ?]. And get partial credit. You actually have to understand the magic, and write the formulas, and hopefully get something that works.

So, we're going to spend all the time that we have left in the semester, I think, building up your intuition for how to build dynamic programming. So, from lecture, dynamic programming is just a catchy name for for optimization problems. So, problems where you're trying to compute the minimum or the maximum of something.

So let's start with a problem. And then, see how all the concepts in lecture relate to that. So this problem is called Crazy Eights.

And we start out with a deck of cards, randomly shuffled. And then some of the cards are pulled out. Did everyone see playing cards? Poker cards? Does everyone know what they are or do I have to define them?

Does anyone need me to define poker cards? Let's put it that way. OK. Cool. So, suppose we have a bunch of these.

AUDIENCE: I don't know what Crazy Eights is.

PROFESSOR: We'll get to that.

AUDIENCE: OK. It's an 8.

AUDIENCE: I know what an 8 is, but I don't know how the game works.

PROFESSOR: OK. We'll get to it. So, let's see. 4 of spades. OK, so this is a bunch of cards that were pulled out of the deck. And we want the longest crazy subsequence. And the crazy subsequence is a subsequence where two cards are like each other.

The way we define like is that either they have the same number. Or they have the same suit. Or one of them is an 8. So an 8 is like anything else. OK. So 4 and 8. Are they like each other?

AUDIENCE: Yes.

PROFESSOR: 8 and 5. Are they like each other?

AUDIENCE: Yep.

PROFESSOR: OK. this 5, 5 of hearts and 9 of diamonds. Are they like each other? No. 5 of hearts and 7 of hearts. Are they like each other?

AUDIENCE: Yes.

PROFESSOR: 5 of spades and 7 of hearts, like each other? No. OK. So we have some cards that are like each other. We want the longest possible subsequence.

AUDIENCE: What about 5 and 5? Those are like each other.

PROFESSOR: They're like each other, yep. They are more cards that are like each other than the ones I drew here. For example, this guy is like everything else. Oh, we already drew that one. OK.

So how do we model this problem using graphs? So, stuff that we knew before the last lecture.

AUDIENCE: Those could all be nodes.

PROFESSOR: OK. All the cards are nodes. That's good. And, I want the longest path between what and what?

AUDIENCE: One node to another?

PROFESSOR: OK.

AUDIENCE: The longest path in the graph.

PROFESSOR: Yeah. The longest path in the graph. One trick to reduce it to a known problem is to add the fake source. So this is a fake source. And it's going to connect to everything. And this way, I want the longest path, starting from the source.

AUDIENCE: And the other source?

PROFESSOR: Nope. The longest path, starting from the source, ending anywhere in the graph.

AUDIENCE: Such that it doesn't go touch another card again?

PROFESSOR: Yeah. So, if I want a longer subsequence, that's a good question. How would the edges look?

AUDIENCE: Directed.

PROFESSOR: Yep. Directed. And which direction?

AUDIENCE: Arbitrary [INAUDIBLE].

PROFESSOR: So, if I choose this one and this one, can I go back afterwards?

AUDIENCE: You just can't go back to the 4.

PROFESSOR: So, in the longest common subsequence, all the cards have to be in increasing order. So I can say I'm going to choose this one, this one, and this one. And they have to match in this order. So I can only go forward.

So I can't reorder the cards. My answer has to be, say, 4 of diamonds, 8 of diamonds. And then, what else would match that? 9 of diamonds.

AUDIENCE: You can put the 8 anywhere, though, right?

PROFESSOR: No. I can't move them. So the cards have to be in the initial order. I don't have to choose all the cards, but the cards that I choose have to respect the initial ordering.

AUDIENCE: So everything points right.

PROFESSOR: Everything points right. Forward. Yep. And, I need one more edge from here to here.

AUDIENCE: Oh, so you're saying in the game you get an initial order. And you can't reorder it.

PROFESSOR: Yeah. But otherwise I can, if I have all the cards, I'm just going to order them.

AUDIENCE: Well, that's the point right? I was picturing more like you get a set of cards and you try to figure out--

PROFESSOR: No. So, that's a different game. That might be harder to solve, so let's stick with this. Yes?

AUDIENCE: Why is 8 in there, with 7?

PROFESSOR: 8 is similar to anything. Just for the heck of it. These are the rules. OK, so we have a graph and we want to compute the longest path.

We do not have an algorithm to compute the longest path. We only have algorithms to compute shortest paths. So how do I deal with that? We had that in problem set

six.

AUDIENCE: Make all weights negative because there's no cycle [INAUDIBLE].

PROFESSOR: OK. We know it's negative. And, I guess these ones don't matter. But all these are going to be, instead of being 1, they're all going to be -1. Very good.

And there are no cycles in this graph, so we know that the answer will be well defined. OK. Everyone with me so far? Happy nods? Yes. So, what algorithm do I know that solves this problem?

AUDIENCE: Bellman-Ford.

PROFESSOR: Bellman-Ford. Good. What is the running time of Bellman-Ford?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Cool. Suppose I have N cards. How many vertices do I have?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Plus the source. It's ordering. So you're right. How many edges do I have? Worst case.

AUDIENCE: N squared.

PROFESSOR: Yep.

AUDIENCE: [INAUDIBLE] you mean, like, the directed [INAUDIBLE]?

PROFESSOR: No, I mean Bellman-Ford. Then, we're going to go to that algorithm and get the better running time.

AUDIENCE: Cause isn't Bellman-Ford e times e?

PROFESSOR: Oh. What?

AUDIENCE: Yeah.

PROFESSOR: Yeah, that's what I-- Did someone say, v plus e? Or did I write v plus e? Yep. You did. Thank you. So it's v times e. So the total running time is.

aUDIENCE: N squared.

aUDIENCE: N cubed.

AUDIENCE: Or, n cubed. Yeah, it's multiplying [INAUDIBLE], too.

PROFESSOR: You guys are mean today. OK, so n cubed applying Bellman-Ford. There is a better way of solving this problem, right? The Directed Acyclic Graph Bellman-Ford. So let's look at the Directed Acyclic Graph, not the one to be generated by this because that might be a bit messy. And let's try to compute shortest path. s, a, b, c.

OK, so let's see how we'd compute the shortest path in this graph. It's Acyclic, right? All the the edges are pointing downwards. So let's try to compute them directly.

I'm not going to write pseudocode first. I'm going to write the formulas, and that will get the intuition for it. Then, maybe right psuedocode. So what is the distance from the source to itself?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Good. The first ones are easy, so I'm going to do them myself. sa is 1. sb is 2. sc is 3. And, of course, you guys get to do the hard ones. So the distance from s to d is what? How would I compute it?

AUDIENCE: There's two paths to get to [INAUDIBLE].

PROFESSOR: OK. So, I would want the shortest path from those, right? OK. So it's 4. And the formula for it is the minimum of two paths, right?

AUDIENCE: [INAUDIBLE] S to b, and then b to b.

PROFESSOR: So, one path is--

AUDIENCE: [INAUDIBLE] and s [? and b ?].

AUDIENCE: [INAUDIBLE].

PROFESSOR: Wait. sa. So, first there's a path from s to a. And the edge a to d. And then, there's a path from s to b. And the 8 from b to d, right?

And I already know the values for sa and sb, so this is well defined. We can compute it right away. No recursion. No metrics. I mean, there's recursion, but there's no infinite recursion. Does this make sense?

So, in order to get to d, there are two edges pointing into d. One of them coming from a. One of them coming from b. So I can either get to a and take this edge. Or get to b and take this edge. This is what the formulas are saying. How about se?s What's the formula?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. What are they?

AUDIENCE: s3.

AUDIENCE: Yeah, sa, sb, sc.

AUDIENCE: [INAUDIBLE]. Aed's.

PROFESSOR: And the number?

AUDIENCE: [INAUDIBLE] 3? I think. e to e.

PROFESSOR: OK. f. Someone that hasn't spoken today. So that I can see that everyone gets it. Please.

AUDIENCE: dsb--

PROFESSOR: OK.

AUDIENCE: --plus wef.

PROFESSOR: OK.

AUDIENCE: And dsc plus wcf.

PROFESSOR: Awesome. And that is--

PROFESSOR: 4.

PROFESSOR: I'll take your word for it. OK, so now the last one. Distance from std.

AUDIENCE: 5. Wait. No.

PROFESSOR: So, let's write a formula.

AUDIENCE: Oh, g, not a.

AUDIENCE: That's 5.

AUDIENCE: 5.

PROFESSOR: OK, so let's write a formula to make sure that you guys are computing it in the fastest possible way for me. So--

sd plus sd to g.

PROFESSOR: OK. Does anyone else? Is this the only path?

AUDIENCE: No. There's more.

PROFESSOR: OK.

AUDIENCE: Plus e as to f.

PROFESSOR: Plus e as f.

AUDIENCE: And n.

AUDIENCE: c to f. I mean, c to g. Or, No. no. s to c.

PROFESSOR: OK. See, there's a trick. OK. Cool. And then, what are the weights?

AUDIENCE: w e to g.

PROFESSOR: e to g. Very good.

AUDIENCE: fg.

PROFESSOR: fg.

AUDIENCE: cg.

PROFESSOR: cg.

AUDIENCE: Why are you adding in c?

AUDIENCE: Because there's the extra c.

AUDIENCE: Oh, I didn't see that.

PROFESSOR: So, the point of this guy's, is that if you do bfs, you'll get tricked. Because bfs would put g on the same level as these. And might compute the value for g before it has the values for these. OK, so what order do I need to compute these numbers in, for this to work?

AUDIENCE: That way you computed them?

PROFESSOR: So, that is a, that is a, something of the graph. Yeah.

AUDIENCE: Oh, topological.

PROFESSOR: Yeah. Stole your answer. Topological sort. So, any of the topological sorts works. The one we used this time is sabcdefg. Why am I using a topological sort?

AUDIENCE: Because there's dependency.

PROFESSOR: Yep. So, this depends on this and this, right? As c depends on sa, as b as c. So basically, every edge here indicates a dependency. In order to compute the shortest distance from the source to here, I need to know the shortest distance from the

source to these two. And then I can look at the edges.

So, the nice thing about a topological sort is, after I write the vertices this way, all the edges point forward. Right? s to a. s to b. s to c. a to d. a to e. b to d. b to e. b to f. I can keep going. But the point is, there's no such thing as a backward edge.

So if I compute the numbers in this order, when I get to se, I know that I computed the distance from s to abcd. And if I have any edge coming into e, I know that I've already computed the shortest distance to the node that it is coming from.

yes? Did I lose you guys? Was this too--

AUDIENCE: So basically, like, at a, you relaxed all the edges going out of it.

PROFESSOR: You can look at it that way. But what we're doing here is I'm looking at a node and I'm relaxing all the edges coming into a node.

AUDIENCE: OK.

PROFESSOR: So this matches this order.

AUDIENCE: OK.

PROFESSOR: What you said doesn't match this order. But it's exactly the same thing.

AUDIENCE: Oh.

PROFESSOR: It'll give you the same result.

AUDIENCE: You'd get the first value at g when you reach c. But, like, end up with the same answer.

PROFESSOR: Yep.

AUDIENCE: We're still going backwards then, instead of--

PROFESSOR: As long as you're processing the nodes in the topological sort order, all the

algorithms will work because you're just computing these terms in a different order. But as long as the dependencies are satisfied, you're still going to get the right thing. OK, so what's the running time of this? How many people know the running time without having to write pseudocode for this? I know the answer beforehand, so I cheated, obviously. OK.

AUDIENCE: Is it a? [INAUDIBLE] edges.

PROFESSOR: Almost. Very close.

[INTERPOSING VOICES]

PROFESSOR: So, it's the topological. So, running time plus the running time for evaluating this. The running time for evaluating this is v plus e because you have every h shows up exactly once in here. So you have e terms. And you have v vertices, even if you don't have any edge, you have to initialize the verdicts. So that's why it's v plus e.

AUDIENCE: It's also topological sort.

PROFESSOR: So, this is the order in which we process this. So everything is v plus e. If we use this algorithm to solve this problem, what will the running time be?

AUDIENCE: n.

AUDIENCE: n squared.

AUDIENCE: It's an no.

PROFESSOR: So, it's v prime plus u prime, which is n squared plus [INAUDIBLE] squared, which is n squared. So, by observing that this graph is acyclic, we have a better running time than Bellman-Ford. Even though we used exactly the same intuition that we used up until now.

Model the problem as a graph. Figure out what the edges are. Run a shortest path algorithm. We have a better shortest path algorithm, which works for Directed Acyclic Graphs, so we get a better running time. OK, questions for what we did so

far?

AUDIENCE: It's [INAUDIBLE] on this graph here, then you would do the same thing. Take each node and then relax the incoming edges.

PROFESSOR: Yeah. So what is the topological sort of this?

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's exactly the cards in the order that they're on the board, right? So, in dynamic programming, the topological sort order is obvious. So the hard part is representing the state and figuring out what the dependencies are. So what the edges are. And after that, the topological sort usually comes, it's fairly straightforward.

OK, so what we can do now, we have a few directions in which we can go. We can write the pseudocode for this. I mean, it's basically, it's just, we're going to write abstract things instead of this. So we're going to write one piece of pseudocode that evaluates these. So it's just generalizing this thing.

Something else we can do is we can look at how would these things get computed using memoization? We can look at how we would compute shortest path in graphs with cycles. So this assumes we have a DAG. What if we have cycles? How do we deal with them? Or we can do another DP problem, and see how we'd model that.

So, votes? What do people want to see? I think we might have time for two things, out of these four. So, everyone votes for one. And we'll start with that. Sorry?

AUDIENCE: DP problem.

PROFESSOR: OK, so one vote for DP problem.

AUDIENCE: Yeah.

PROFESSOR: OK.

AUDIENCE: Cyclic.

AUDIENCE: Wait, you got two or one votes?

PROFESSOR: One vote.

AUDIENCE: OK. I'll DP.

AUDIENCE: I'll stick with [INAUDIBLE].

AUDIENCE: DP.

PROFESSOR: I think we're done here. All right. So, new DP problem. So, suppose you have something like Manhattan's map, which is basically a lattice. Fancy math term for a grid. And suppose it we can only go forward and down.

So, all the streets are one way because they liked how people drive in San Francisco, and decided they're going to do the same craziness. So, we want to go from s to t. And there are different cost on all these edges.

This is an n by m matrix. And I want to get from s to t in the shortest possible way. So, let's model the problem. And then let's do recursion formulas and make it look like a DP instead of like a graph problem because-- it might be too easy of a graph problem in hindsight. OK, so what's the graph problem?

AUDIENCE: It's not equivalent.

AUDIENCE: Topological sort. And then bfs.

PROFESSOR: OK. Topological sort. And then, and then that. Well, let's write this as, let's say that each node is so let's say that the nodes have numbers, right? So this is 1,1 and this is 5,4. And I want to write this using math, so 2,1. 3,1. 4,1. 5,1. And then, 1,2. 1,3. 1,4.

OK, so this is the graph. The shortest path from here to here is obvious. By the way, this is a real problem for people who don't see the connection between dynamic programming and graphs. I This problem has tripped up people on exams before. So it's not a toy problem. OK.

AUDIENCE: So, on an examine, would you have the option of just using a straight up graph algorithm--

PROFESSOR: Well, we're going to ask you to solve this using DP. So let's try to solve this using dynamic programming by writing recursion formulas. So, what's the shortest distance to 1,1.

AUDIENCE: 0.

PROFESSOR: OK, then. If I have a general distance, if I have some random node here, dij, how do I compute this?

AUDIENCE: It's a minimum distance between the distance of law plus weights.

AUDIENCE: Or the distance of j. i minus one. Yeah.

PROFESSOR: So, i minus 1 j.

AUDIENCE: Yeah. We get it. Plus the weight of the, yeah.

AUDIENCE: Going from one to the other.

PROFESSOR: OK, so the weight from i minus 1 j to ij. OK. And?

AUDIENCE: j minus 1i. For i, j minus 1.

PROFESSOR: ij minus 1.

AUDIENCE: Plus weight of that ij. ij minus j1 to ij. ij [INAUDIBLE]

PROFESSOR: OK. So these are the recursions. Now, how would I write the full set of code for this? So what's a valid topological sort for these guys?

AUDIENCE: 1,1. 2,1. 3,1.

AUDIENCE: Dials.

AUDIENCE: 1,2.

AUDIENCE: The right dials.

PROFESSOR: That's going to be hard to code. That's going to be easy to code. So I'm going to take your answer.

AUDIENCE: OK.

AUDIENCE: What?

PROFESSOR: I'm not going to take your answer because your answer is correct, but it's hard to code.

AUDIENCE: But isn't that the same thing [INAUDIBLE]?

PROFESSOR: She says, go like this.

AUDIENCE: Oh. Oh.

PROFESSOR: So, here's how I'm going to code them. 4i in 1,2n. 4j in 1 to m. So, I guess, first off, if i is 1 and j is 1, then d of ij is 0, right? This is the base case. Otherwise, d of ij equals big bad formula that we have up there. OK. Do we need anything else?

AUDIENCE: DP.

PROFESSOR: This is DP. We're done.

AUDIENCE: Is it?

PROFESSOR: Almost work. Yeah, this is--

AUDIENCE: Over [INAUDIBLE]--

AUDIENCE: I thought this was just programming.

AUDIENCE: --it's a dictionary, though.

PROFESSOR: Yeah. It's programming. So this is the DP solution to the program because, instead of building the graph, you're writing the recursion. And you're writing using this implicit representation of the graph.

AUDIENCE: Oh.

PROFESSOR: This is it. This is DP. Most people are really afraid of it. This is the hardest thing in the course. If you get the connection between graphs and this, and if you know how to model graphs, that it. You're one month in the term, you're done with six double six. You already know everything to ace the exam. OK.

AUDIENCE: Aren't graphs just programming, as well?

PROFESSOR: Ah. But there, you're building a graph structure. Here, we don't need to build that structure. Because we see the connection directly. Like, this code is much smaller, right? It's much easier to look at. I mean, sorry. It's faster to read. But it looks like black magic if you don't see the underlying graph. Yes? Did you have a question?

AUDIENCE: Yeah, so. What exactly is dynamic programming? Unless they give at least one or two examples of, like, using something that you had calculated already, or--

PROFESSOR: Yeah.

AUDIENCE: --Fibinocci. So.

PROFESSOR: Yeah. So--

AUDIENCE: Ah, so now you just have to have a dictionary to store the minimum cost.

AUDIENCE: That's what d is, though. The d of ij.

AUDIENCE: Wait.

PROFESSOR: So. So I like your question. And I'm going to address all the other ones first. And then I'm going to spend about five minutes addressing your question. What is dynamic programming, right? What is the point of this? Like, what is dynamic programming?

We're going to come back to this. So any questions about this?

AUDIENCE: Wait. As you're going through--

AUDIENCE: Is there a dictionary?

PROFESSOR: Sure. This is an array. Or dictionary. So, say this is ij. If it's an array, or if it's a dictionary, it would be d of the tuple ij. So, I can write this in Python, right? This is almost Python. What am I missing?

AUDIENCE: [INAUDIBLE], or it fits inside.

PROFESSOR: Yes. So, I have some boundary conditions, right? Because this guy would depend on this guy, which is inside. And left to depend on this guy. Which doesn't exist.

AUDIENCE: Oh. I wasn't-- OK. Sure.

PROFESSOR: So, we need a few more ifs here, for boundary conditions.

AUDIENCE: I mean, in theory, though, you could just run through the new dfs and create a topological of the source. Right? And just run through that.

PROFESSOR: Yeah. But that's so much code to write. Look at this. This is five lines.

AUDIENCE: Well, yeah. For this particular problem, it's five lines.

PROFESSOR: Well, for dynamic programming, the solutions are 5 to 10 lines in general. And the only hard thing in dynamic programming is figuring out what is the state going to be. So, after you get used to them, after you solve 10 or 20, when people come out of programming contests, and someone says, my solution's dynamic programming.

Really? Wait, you can solve it that way. And he says, yeah. This is the state. And then everything else is obvious. Like, it's pretty easy to figure out everything else. The hard part is the state.

So, it's enough to say my solution is dynamic programming. This is my state. You guys are probably going to have to say a bit more than that on the exam. But this is the hard part.

AUDIENCE: DP.

PROFESSOR: No, you can't just say DP. You'll definitely need at least a state. OK. So.

AUDIENCE: You're also missing a state for deciding stuff in the dictionary, though, right?

PROFESSOR: Before deciding if my key is already in the dictionary? Well, so, aside from the boundary conditions here, if I compute this here, it depends on theses two, right? Are they going to be in the dictionary?

AUDIENCE: Not yet.

PROFESSOR: Why? If I'm running this way, so if I'm computing all my values in this order. So the first line one. Then line two. Then line three. They're already going to be in the dictionary. So this is because I'm doing topological sort. You don't need memoization if you're not using the topological sort of the graph. You only need memoization if you don't.

AUDIENCE: So, why are we doing boundaries commissions if--

PROFESSOR: Because--

AUDIENCE: [INAUDIBLE].

PROFESSOR: So, I'm doing a boundary condition because if I paste this in if I paste this thing in, then when I am here, this is going to refer to this guy. Which is fine. But it's also going to refer to 2,0. Which doesn't exist.

So the code might be a bit obfuscated by the boundary conditions. But it's 10 lines of code. It's pretty nice and straightforward. OK, now what is dynamic programming? I like that question.

PROFESSOR: So, a key property, I don't think it was mentioned in lecture. Guys, correct me if I'm wrong. It's called optimal substructure. Does that ring a bell? Probably going to hear about it next time, so. Optimal substructure.

So, the point of optimal substructure is, suppose I have a shortest path from s to g.

Right? so, suppose I have shortest path from s to g. And that path is called p. Now, suppose this path goes through d. So, path from s to g is actually s making a path through d. And then there's another path going from d to g.

This path over here, p1. p1 is guaranteed to be the shortest path, or a shortest path, from s to d. So this is the big solution. It's optimal because we say it's the solution to the problem. This is a part of the solution. This part of the solution is optimal for this part of the problem.

So the part of the problem is getting from s to d. The part of the big optimal solution is optimal-- so the small part of the big solution is optimal with respect to the small problem. So p1 has to be a shortest path from s to d. Do you guys want to see a proof by contradiction? Or do take my word for it? Does anyone want to?

So, intuitively, the idea is that if you had the better path here, say that path would be s3, then I could replace this, sorry. p3. I could replace this with p3. And I would have a better path overall. And that would contradict the fact that this is the best path.

So this part of the path has to be a shortest path to get from s to d. So I've broken up my problem to get from s to g, into saying, I want to get from s to d, from s to e. Or, from s to f and then cross one edge. And then the ways I get from s to d from s to e, or from s to f, have to be optimal.

I've already encoded that here. And nobody asked me, yo, is this true? Can I take a longer path here? And have a better solution? The answer is no. In some problems, the answer is yes. Those are not problems that you can solve with dynamic programming. if, in your problems, that's the case, you probably forgot to account for some part of the state.

AUDIENCE: What kind of problems would it be where that wouldn't be true?

PROFESSOR: Well, remember the quiz problem with the gas stations? If you don't account for the gas, if you do Dijkstra, then, well. Guess what? Shortest path in the graph, if that doesn't account for gas stops, if you start accounting for the cost of refilling, this path might be longer than a path that goes like this.

So, it's longer in terms of road stalls. But has fewer, or has cheaper, gas stations on the way. So then, there's no optimal substructure. And that's because you didn't account for the fuel states.

OK. Probably not the best example. Sorry for bringing up painful memories. But the point is, usually when you have this with our problems, you didn't account for the state. All the problems that are solved with dynamic programming have this thing called optimal substructure. And this is sort of how it works.

OK. I have no idea how much time I have because my phone crashed. So can anyone help me? Five minutes. 10 minutes. Sorry.

AUDIENCE: Seven minutes.

PROFESSOR: OK. What else do you guys want to see?

AUDIENCE: Cycles. [INAUDIBLE].

PROFESSOR: OK. Is everyone happy with cycles? OK. Almost everyone, so that's good enough. OK. Let's do cycles. We have seven minutes.

So, suppose I have this graph. Source going a going to be going to c. And then the costs are 1 minus 1. 1. 1. Can I solve it using that method? Probably not. Let's try to write the recursions to see what we get for the formulas.

So, dsa is--

AUDIENCE: 1.

PROFESSOR: --minimum. Yeah, it's 1. But it's the minimum of dss plus weight sa. ds-- well, almost. Actually, it's not 1. Likely. You're confusing me again. There's one more edges that they have to account for.

AUDIENCE: Oh.

PROFESSOR: dsc plus weight. I'm going to fail today. I'm tired.

AUDIENCE: ca.

PROFESSOR: OK. So we accounted for both edges coming in now. dsb is minimum of dsa plus weight ab. dsc is minimum of dsb plus weight bc. Right? And dss is 0 because we promised that's how we start.

OK. Now what if I try evaluate these? Is there a sane order in which I can evaluate them? Nope. Let's see why. If I try to evaluate sa, this depends on sc.

AUDIENCE: Which we don't have yet.

PROFESSOR: sc is here. sc depends on sb. sb is here. And it depends on sa. Which was. So we have this infinite recursion, right? They all depend on each other. There's a loop here. There's a negative cycle. Can't use this algorithm. That's a shame. What can we do instead?

PROFESSOR: Show a negative node. A negative weight. half.

PROFESSOR: Sorry?

AUDIENCE: Can we just get rid of the negative weight half?

PROFESSOR: No. That's the best edge. That's probably going to be part of the solution, right?

AUDIENCE: Can we add 1 to all the edges?

PROFESSOR: That's still going to have a cycle. I still won't be able to run this.

AUDIENCE: Oh, yeah.

AUDIENCE: Bellman-Ford.

PROFESSOR: Bellman-ford.

AUDIENCE: That's not dynamic.

PROFESSOR: OK. Well, one way to do it is Bellman-Ford, right? Another way, which we went

through last time, is to break the cycle. And the way we break the cycle is we add the path length into the equation.

So, I'm going to look at the distance from a source to some node, so the distance from a source to some node, as the distance from the source to some other node plus the edge weight. This is what I had before, right? Nothing new here. So dsv is the minimum over all the edges of dsu plus weight uv, Right? Minimum over all uv. That's our edges. Right? This is the old stuff.

Now, we're going to say this instead. The distance from s to v, using a path of length k, is the minimum over all the edges of the distance from s to u, using a path of what length? If this path is length k. So k edges. How many edges do I have here?

AUDIENCE: k equals 1.

PROFESSOR: So this is distance su using k minus 1 plus the weight of uv.

AUDIENCE: So what's the difference between the two?

PROFESSOR: So, this will always decrease. So I guarantee that they will not have an infinite recursion. This is the magic that makes it work. Now, an equivalent way of looking at this, is building a graph. That's what we've been doing so far. So let's build an equivalent graph to this.

AUDIENCE: [INAUDIBLE]

PROFESSOR: It's going to get to 0 eventually, right? So when k is 0, the distance from the source to itself is 0. But the distance from the source to any other nodes is infinity. Because from the source, we can't get to anywhere else in 0 edges. We can't teleport.

So let's build a graph to get the intuition for this. This looks mathy. This looks too mathy. So, at the first layer, you only have the source. You can only get from the source to itself by not crossing any edges.

At level one, you potentially have all the nodes. If the source is connected to

everything, you might have all the nodes. So, s1, a1, b1, c1. Where can you get from the source?

AUDIENCE: To a.

PROFESSOR: To a. What's the cost of the edge?

AUDIENCE: 1.

PROFESSOR: OK. Now, let's build a second layer. s2, a2, b2, c2. So someone tell me the edges. So, assuming I can get to this node using one edge, how can I get to this other node using two edges?

AUDIENCE: A to b.

PROFESSOR: OK. a to b. a1 to b2.

AUDIENCE: 2.

PROFESSOR: OK. Cost?

AUDIENCE: 1 [INAUDIBLE]. A1 to c--

AUDIENCE: That's negative 1.

PROFESSOR: a1 to b2. Thank you. OK.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Sorry?

AUDIENCE: Are we doing anything else?

PROFESSOR: I think so. So, if I'm at b using one edge, I can get to c using two edges. Cost?

AUDIENCE: 1.

PROFESSOR: OK. If I'm at c using one edge--

AUDIENCE: a.

PROFESSOR: --using two edges. Cost?

AUDIENCE: 1.

PROFESSOR: OK. And if I'm at s using one edge?

AUDIENCE: a.

AUDIENCE: [INAUDIBLE]

PROFESSOR: Can you?

AUDIENCE: There's now edge.

PROFESSOR: There's no edge.

AUDIENCE: I thought it was just like an assumed edge.

PROFESSOR: Nope. Nope. Not an edge. As to a, cost?

AUDIENCE: 1.

PROFESSOR: 1. OK. Let's build a third level. s3, a3, b3, c3. Someone dictate the edges please.

AUDIENCE: S2 to a3. With 1. I mean, it's going to be the exact same.

PROFESSOR: Yep. a2 to b3 minus 1. b2 to c3 1. And c2 to a3 1. So these are exactly the same edges, right? Because they're the original edges in the graph. Every edge in the graph can connect to levels here.

So, all edges. All edges. All edges from s. How many layers do I need?

AUDIENCE: e.

PROFESSOR: OK. Pretty close. Let's try something smaller.

AUDIENCE: a.

AUDIENCE: Oh. That makes no sense.

PROFESSOR: So, what's the longest path in a graph? Graph of v vertices. What is the longest path?

AUDIENCE: The number [INAUDIBLE].

PROFESSOR: The longest shortest path.

AUDIENCE: v minus 1.

PROFESSOR: v minus 1. That's how Bellman-Ford has v minus 1 runs, right?

AUDIENCE: Oh.

PROFESSOR: So, a shortest path can't have a cycle. If it has a cycle, then it means it's an infinite cycle. So there's no solution. Shortest paths have no cycles therefore, even if they go through the entire graph, they're going to have v minus 1 edges.

So I'm going to have v minus 1 layers. Here, I drew three layers, so I'm done. That's why I stopped at three. So let's see how many nodes and how many edges we're going to have if we do this transformation. v prime is-- so, how many times am I going to copy the graph?

AUDIENCE: Two or thee times.

AUDIENCE: Three times.

PROFESSOR: OK. And in general terms?

AUDIENCE: v minus 1.

AUDIENCE: v minus 1 times.

PROFESSOR: OK. So I'm going to copy the graph v minus 1 times. And then I'm going to add that one source, right? Doesn't really matter because it's order of b squared. How many edges?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Order of v times e. We can do the math that's whatever. Let's say it's something like this. So the running time-- this graph is acyclic, right? All the nodes are going forward. The new graph that I have here.

AUDIENCE: Yeah.

PROFESSOR: So, I can use the DAG algorithm. So the running time, if I use the DAG algorithm, is v prime plus e prime. Which is?

AUDIENCE: v.

PROFESSOR: Thank you.

AUDIENCE: Or is it ve?

PROFESSOR: So, it's v squared plus ve, which is ve, for most purposes. And this is?

AUDIENCE: Bellman-Ford.

PROFESSOR: Bellman-Ford. So this is Bellman-Ford. This is what Bellman-Ford does. Except when you're coding it up, it relaxes the edges in a different way. But this is the intuition behind Bellman-Ford. And this is an easy way to see why Bellman-Ford works.

AUDIENCE: So, practically, we really wouldn't want to do dynamic programming. We just want to run Bellman-Ford because that's less code, right?

PROFESSOR: So, this is the dynamic programming view of Bellman-Ford. Write Bellman-Ford. There's a reason why we taught you to write it that way. It's going to be shorter.

This just gives you more intuition. And it shows you how the DAG algorithm relates to Bellman-Ford. And this is how we handle cycles, which are removed. So, that means I have fulfilled my promise of covering two of the issues that I had on the board.

Yes. So any questions about this? So, we didn't do the pseudocode for the shortest path using DAGs. The code that we gave you in the code handout matches the pseudocode that you'd write. Yes?

AUDIENCE: So, if all edges were negative 1 here, except for the top edge, looking at this graph over here, how would the search go through, such that it would find, lik,e negative two weight half?

PROFESSOR: So these edges are minus 1?

AUDIENCE: Yeah.

PROFESSOR: Well, do you have a solution in this case?

AUDIENCE: It's no.

PROFESSOR: No. But you could have this, right? And expect the whole thing to work.

AUDIENCE: No, that's still not, that's not making sense.

PROFESSOR: Is it?

AUDIENCE: It's negative 1 cycle.

PROFESSOR: Oh, yeah. That's unfortunate. OK. Never mind. Do like this? OK.

AUDIENCE: So, also, looking at the graph, there's only one natural path that you can take.

PROFESSOR: Yep. So, if I go from s to c, like this, sabc, this is going to be s0, a1, b2, c3. So, all the paths go ahead.

AUDIENCE: So, what if I wanted to find the shortest path to s to b? Like, in terms of actually writing an algorithm, would it be s0, a1, b2, or s1? a2, b3.

PROFESSOR: OK. So, if you actually want to read the shortest path, then the shortest path could have length 1, length 2, or length 3, right? I don't know. So I would have to look at all these.

AUDIENCE: Oh. OK. So you just run it from s0 to any b, basically.

PROFESSOR: So, the algorithm that we have there computes the path from one source to everything else. So I run it. It runs. Computes all the shortest paths. And then I have to read these ones. And get the smallest one.

That's a question. Thank you. Yeah, that is a detail that I left out. Thank you. So, no more cycles. OK. Any other questions? Yes?

AUDIENCE: I'm still, like, on the initial problem and stuff. A bit of a disconnect. When you were underlining stuff, like the sa, it seemed to me that, like, in that case, when you're going forward, you're never going to stop because you're doing recursion. So you never have a beginning point, almost. Effectively, if you, like--

PROFESSOR: So, do you mean here?

AUDIENCE: Well, yeah.

PROFESSOR: So, in that case, this is the beginning point.

AUDIENCE: Right.

PROFESSOR: If I go through the nodes in the topological sort order, then all I need is one beginning point. Because everything else will refer back to that. There has to be a topological sort order. And the first node in that order is my source.

And, if we have cycles, then the beginning conditions are here. So that's why I'm only drawing this vertex. Because these other vertices wouldn't be useful. OK. So then, that being said, don't forget your quizzes and happy Thanksgiving.