PROFESSOR: Let's get started. Welcome back to 6046. Today, we start an exciting series of algorithms for graphs. We've done a lot of data structures. We're starting to get back into algorithms with dynamic programming last week. And today and the next few lectures, we're going to see lots of cool algorithms about graphs.

First a bit of recall-- we're starting with shortest paths, which you've seen in 6006 in the context of single-source shortest paths. So typically, like you do a Google Maps query, you think of, I want to go from A to B.

But what you solved in 6006 was a harder problem, which is I give you a point A-- here it's called S for the source. I give you a source vertex. And capital V is a set of all vertices, capital E is a set of all edges, remember graph notation. Let's say it's a directed graph. You've got edge weights, like the time it takes to traverse each road. And you want to know how long's it take me to get from S to V for all V. So this is from one given point to everywhere.

Today, we're going to solve a harder problem, which is all-pairs. I want to go from all A to all B. But what you saw in 6006 was single-source, where I just give you one of the vertices, and I want to know how to get to everywhere. The reason you saw this version and not the A to B version is because the best way we know to solve A to B is to solve this problem. So at least from a theory standpoint, we don't know how to beat Dijkstra's algorithm and Bellman-Ford's algorithm for the A to B problem. So you get a little bit more than what you asked for sort of for the same price.

So let me remind you in a few different scenarios what algorithms we have, and how long they take. So the scenarios of interest are the unweighted case, a non-negative weighted case, the general case, arbitrary weights, positive and negative, and DAGs acyclic graphs. These are some interesting special cases. And you should have seen in 006 algorithms for each of them. Let's see if you remember.

So what's a good algorithm for single-source shortest paths in an unweighted graph? BFS, good, Breadth-first Search, that takes how long? V plus E, good. That's-- for graphs, V plus E is considered linear time. That's how long it takes to represent the input. So you got to look at the input, most algorithms, and the BFS is optimal against that. But we're going to start getting worse as we-- well, for these two situations.

So for non-negative edge weights, what do you use? Dijkstra. Ah, everyone's awake this morning. That's impressive. And that takes how long? This is a tricky question. V log V plus E, wow, nice.

So this answer kind of depends on which heap structure you use, but this is the best we know. If you use a Fibonacci heap, which we don't actually cover but it's in the textbook, you achieve log V for extract key and constant amortized for each decreased key operation. So sorry, this is for extracted min, and this is for decrease key. And so this is the best we know how to do with a Dijkstra-type approach.

If you use other heaps, you get slightly worse, maybe you get a log factor here. But this is good. This is almost as good as V plus E. For moderately dense graphs, if E is bigger than V log V, then these are the same. But if your graph is sparse, like E is order V, then you lose a log factor. But hey, it's just a log factor, not too bad. We're going to get worse.

So for general weights, what do you use? Bellman-ford. OK. Which takes how long? VE, that's the usual statement. Technically, you should assume VE is at least V for this bound to hold. But that's the way to think of it. So this is not nearly as good. This is a lot slower.

If you think of-- we can think of two situations. One is when E is theta V, so a very sparse graph like a tree or planar graph or something. And we could think of when E is quadratic. That's the dense case.

So here we get, whatever, V and V squared for BFS. For non-negative edge weights we get V log V in the sparse case. And we get V squared in the dense case. And for Bellman-Ford, we get V squared in the sparse case, and V cubed in the dense case. So this is like a V factor, a linear factor larger than non-negative edge weights-- makes a huge difference.

And finally for acyclic graphs, what do you do?

STUDENT: Dynamic programming.

PROFESSOR: Dynamic programming is one answer, yeah. That works. In some sense all of these algorithms are-- especially Bellman-Ford is a dynamic program. We'll see that little bit. Another interpretation? Topological sort, and then Bellman-Ford, yeah-- say, one round of Bellman-Ford.

So Bellman-Ford actually works really well if you know the order you should relax the edges. And if in an acyclic graph, you can do a topological sort, meaning you visit all the vertices, so that whenever you visit the right endpoint of an edge, you've already visited the left endpoint. If you do Bellman-Ford in that order, then you only have to do one pass and you're done. Whereas normally, here, you had to do it V times.

So the total cost of this is just linear. Good thing to remember, especially on quizzes and so on. If your graph is acyclic, you can achieve linear time. But in the general case, Bellman-Ford is your answer for single source.

Now, these are the best algorithms we know for each of these cases. So I'm not going to improve them today. You saw the state of the art 006. But for all-pair shortest paths, we can in some sense do better, sort of. So let me just quickly define the problem, and then tell you all of the results we know. And also, the results we're going to cover today.

I didn't remind you of the delta definition. I want to go over this briefly. So delta of s comma v is the weight of the shortest path from S to V. The weight is well-defined. Even though there may be many shortest paths, there's one best weight.

But there's some special cases. It could be infinity, if there's no path. That's sort of by definition. Say, well, it's infinite costs to get-- if there's no path, then we said there's infinite weight one.

And it could be minus infinity in the presence of negative weight cycles. So let's say, if there's a negative weight cycle on the way, if you could reach a negative weight cycle from s, and then still get to V from there, then the best way to get there is to go to that cycle loop around infinitely many times, and then go to V.

OK, so the algorithms you saw probably didn't actually compute correctly in this case. They just said, negative weight cycle-- I don't know what to do. But it's actually not that hard. With a little bit more effort, you can figure out where the negative infinities are. We're not going to rely on that, but I'm just throwing it out there to make this a well-defined definition.

Once you have the shortest path weights, you can also store parent pointers, get the shortest path tree, then you can actually find shortest paths. But again, we're not going to talk about here. We'll focus on computing delta, but with the usual techniques you saw in 006, you could also reconstruct paths.

So for all-pairs shortest paths, we have a similar set-up. We have a directed graph, V,E. And we have an edge weight function w-- in general, could have negative weights. And our goal is to find delta of u comma v for all u and v. OK.

Single-source shortest paths is the sort of thing that you might want to do a few-- just given a graph, and you want to find a shortest path from A to B. I said, this is the best way we know how to do A to B, essentially. But all-pairs shortest paths is what you might want to do if you're pre-processing.

If you're Google Maps, and you want to be able to very quickly support shortest path queries between major cities, then you may want to first compute all-pair shortest paths for all major cities, because road networks don't change very much, the large scale. This is ignoring traffic and so on. Pre-compute this, and then given a query of two vertices, come in-- probably get a million queries a second-- you could very quickly know what the answer is.

And this is the basis for real world shortest paths. Typically, you don't compute shortest paths from A to B every single time. You use waypoints along the way. And you have pre-computed all-pair shortest paths between waypoints. So that's the motivation.

Yeah, I guess in some sense, internet routing is another situation where at any moment you may need to know the shortest path to get to-- the fewest hop path say to get to an internet site. You know the IP address. You need to know where to go. You don't need to know the whole path. You need to know the next step. But in some sense, you're computing all-pair shortest paths. That's a more dynamic situation.

OK. So here are the results we know for all-pair shortest paths. I think I'm going to cheat, and reuse this board. So same situations, except I won't think about acyclic graphs here. They're a little less interesting. Actually now, I'm curious, but I didn't intend to talk about acyclic.

And so the obvious thing to do to solve all-pairs shortest paths is just run the single source algorithm V times, once from each source. So I could do V times Breadth-first Search, V times Dijkstra, V times Bellman-Ford. And now, I just need to update my bounds.

OK so VE becomes V squared plus VE. If you're a little bit clever, or you assume E is at least V, that becomes VE. If I run Dijkstra V times, I'm going to get V squared log V plus V times E. And if I run Bellman-Ford V times, I get V squared E.

OK. And over here, everything's just going to increase by a V factor. So a little more intuitive is to think about the sparse case. I get V squared, V squared log V, and V cubed. Check that those match over there-- 1 equals V. And over here, I get V cubed, V cubed, and V to the fourth. OK. So pretty boring so far.

The interesting thing here is that we can beat the last result. The last result, which is the slowest one, could take as long as V to the fourth time. We can shave off a whole V factor.

So a better general case algorithm is called Johnson's algorithm. That will be the last algorithm we cover today. And it achieves this bound, which is the same as running Dijkstra V times. So it's between V squared log V and V cubed.

And that's cool, because this is the best algorithm we know for all-pairs, non-negative edge weight, shortest paths, just running Dijkstra V times. Not very smart-- but it's the best thing we know how to do. And what this says is, even when we have negative edge weights, actually we can achieve the same bound as running Dijkstra.

This is a bit counter intuitive because in 006 you're always told, if you have negative edge weights, can't use Dijkstra. Turns out, in the all-pairs shortest paths case, you kind of can. How can that be? Because this is a harder problem. If you could solve all-pairs shortest paths, of course you could solve single-source.

And that's actually the luxury. Because it's a harder problem, we have this VE term in the running time, which lets us do things like run Bellman-Ford once. And running Bellman-Ford once will let us run Dijkstra V times. That's the reason we can achieve this bound. But we won't be seeing that for a while.

For starters, I want to show you some connections between all-pairs shortest paths, and dynamic programming, and matrix multiplication, which turn out to give-- for dense graphs, we're just achieving V cubed in all situations. So our first goal is going to be to achieve V cubed time for general edge weights. So we're going to first achieve this bound. That will be a lot easier. And then eventually, we will achieve this bound. So the Floyd Warshall algorithm and some of these will get very close to V cubed.

All right. So we're going to start with our first approach to solving all-pairs shortest paths-- that is not using an existing single source algorithm-- is dynamic programming. Someone mentioned that already. It's a natural approach. Shortest paths is kind of dynamic programming. In fact, most dynamic programs, you can convert to single-source shortest paths, typically in a DAG-- not all, but a lot of them. So we could try dynamic programming.

Now, I'm going to preach to you a little bit about my way of thinking about dynamic programs. If you watched the 006 OCW version, you've seen the five easy steps to dynamic programming. And if you haven't, this will be new, otherwise, a reminder.

First thing I like to think about in a dynamic program is, what are the subproblems? The second thing I like to think about is, what am I guessing? I'm going to guess some feature of the solution. Third thing is, I want to write a recurrence relating subproblems solutions. Then, I'm basically done, but there's a couple wrap up things, which I'm going to have to use another board for.

So number four is, I need to check that I can actually resolve these subproblems in some order that's valid. Basically, this is saying that the constraint graph on some problems should be acyclic. Because if there's a cycle in the constraint graph, you take infinite time. Even if you memoize, if you do infinite recursion-- bad news. You'll never actually finish anything, so you never actually write anything in the memo table. So I want to make sure that it's acyclic.

I mean, this is really the same thing we're talking about in this erased row, which is topological ordering. Personally, I like to-- you could argue that it's acyclic. I like to just write down, here's a topological order. That's a nice proof that it's acyclic. If you write that down as for loops, then you actually have a bottom up dp. If you just take the recurrence, stick it inside the for loop, you're done, which we'll do in a moment. I guess I need a row for that.

And finally, you need to solve the original problem. And then, there's analysis and so on. But for specifying the algorithm, these are the key things you need to know.

The hard part is figuring out what the subproblems should be, so that your dp becomes fast. Running time is going to be number of subproblems times time per subproblem. For each subproblem, usually we're going to want to guess some feature of the solution to that problem. Once we do that, the recurrence becomes pretty trivial. Just for each guess, you say what it should be. So these are the really hard two steps.

And then, OK, we checked that it's acyclic. And we make sure that we can actually solve our original problem using one of the subproblems. Sometimes, our original problems are some of the subproblems. I think that will happen here. But sometimes you need to do a little bit of post-computation to get your answer. All right.

So what am I going to do for subproblems? Well obviously, I have a bunch of different problems involving pairs of vertices. I want to find delta of u,v for all u and v. That, I erased. But that's the problem.

So I want to know what is the weight of the shortest path from u to v? If I stop there and said that was my subproblems, bad things are going to happen, because I will end up-- there's no natural way to make this thing acyclic. If I want to solve u to v using, I don't know, u to x, and then x to v, something like that-- there's no way to get out of the infinite recursion loop. OK?

So I need to add more subproblems to add more features to my solution, something that makes it so that when I try to solve my subproblem, I reduce it to other subproblems. Things get smaller, and so I could actually make progress.

So there are actually two natural ways to do this. I'll call them method one a method two. Method two is actually Floyd Warshall. But any suggestions on how we might do this? This is a harder problem. This is in some sense, a kind of guessing, but it's like I'm going to guess ahead of time that somehow there's an important feature of the shortest path. I'm going to parameterize by that, and somehow it's going to get smaller.

Yeah?

STUDENT: [INAUDIBLE]

PROFESSOR: Right, shortest path-- it uses at most a given number of edges. Let's parameterize by how many edges. I think I'll use m. So using at most m edges.

Very good. So good, I think it deserves a purple Frisbee. All right, I'm getting better, slowly. By the end the semester, I'll be pro at frisbee. I should enter a competition.

So this is, of course, an additional restriction. But at the end of the day, the problem I want to solve is going to be essentially duv, let's say, n minus 1. If I want a shortest path that does not repeat any vertices, then certainly it has at most n minus 1 edges. So in fact, the claim would be that duv equals duv n. I mean, and so on. If you go larger than n minus 1, it shouldn't help you. If you know that your shortest paths are simple-- if you know that shortest paths don't repeat vertices.

So this would be if there are no negative weight cycles. If there are no negative weight cycles, then we know it never helps to repeat vertices. So in that situation, we would be done, if we could solve this for all m.

Now, slight catch-- well, how do we know there's no negative weight cycles? You know, we could run Bellman-Ford, I guess. That's a little tricky, because that only finds reachable negative weight cycles.

In fact from this picture, we will end up knowing whether there are negative weight cycles. So there will be no negative weight cycles, if and only if there's no negative diagonal entry, say dvv n minus 1. So it turns out, this algorithm will detect there's a negative weight cycle by finding that the distance from v to v is negative. Initially, it's going to be 0. If it turns out to be negative, we know there's negative weight cycle.

With more work, you could actually find all the reachable pairs, and so on. But I'm not going to worry. I'm just going to say, hey, a negative weight cycle. I'm going to throw my hands up in the air and give up for today. OK. Cool.

So I can solve my original problem, if I can solve these subproblems. And now, things are easier, because we can essentially assume in solving this problem that we've solved smaller subproblems for however we define smaller. That's what's given by this topological order. Obvious notion of smaller is smaller m. Presumably, we want to write this with an m in terms of this with an m minus 1 or smaller.

So this gets to our guessing part. What feature of a shortest path could we guess that would make it one edge shorter? There's probably two good answers. Yeah? The next edge, which I guess you mean the first edge? Sure. Could guess the first edge, or you could guess the second edge. Or no, that would be harder. Or I mean, the last edge, that would also work.

Okay, this is a harder one. Uh, nope. Good thing there's students everywhere to catch it. Cool.

So I'm going to guess the last edge. That's just how I've written the notes. But first edge would also work fine. So I'll call the last edge x comma v. We know we end by going into v. So let's guess the vertex previous to it in the shortest path. Now of course, we don't know what that edge is. The guessing just means try them all as you saw last time.

So now, it's really easy to write the recurrence-- see if I can do it without looking at my notes. So we've got duv of m. We want to write-- we want to find the shortest path, so it's probably going to be a min on the outside. And we're going to consider the paths of the form-- d go from u to x using fewer edges. Right, if this is the last edge, then we use m minus 1 edges to get to x. And then, we follow the edge x comma v.

So I'll just add on the weight of the edge, x comma v. If x was the right answer, this would be the cost to get from u to v via x, where xv is a single edge at the very end. We don't know what x should be, so we're just going to do for loop over x for x in v. So this is using Python notation. And that will find the best answer.

Done, easy. Once you know what the subproblems are, once you know what the guessing is, basically, I'm just adding in a min and a for loop to do the guessing.

So that's my recurrence, except I should also have a base case. Here it's especially important. So base case is going to be when m is the smallest. So that, let's say, is 0. What is the weight of getting somewhere using 0 edges? Well, typically it's going to be infinity.

But there is an interesting situation, where at 0 namely when u equals v. Hey, there is a way to get from u to itself with 0 edges. And that costs 0. But anywhere else is going to cost infinity. There's no path. So the sort of a definition, I should say. If it exists, otherwise it's infinity. So then I get those infinities.

But this is kind of important, because maybe I actually use fewer than m edges. I wrote less than equal to m here. This is also less than equal to m minus 1 edges here. But in some sense, I'm including the case here, where x equals v. So I just stay at v at the very end. So it's because I have a 0 here, I'm implicitly including a situation where actually, I just use duv m minus 1. That's in that min here. So it's important to get the base case right. Cool. Almost done.

I need an acyclic ordering. So as I said, things get smaller when m is smaller. So all that means is do the for loop for m on the outside. Then do the for loop for u in v. For those, it doesn't matter what order you do it, as long as you've done all of m equals 0, before you do all of m equals 1, before you do all of m equals 2. So that's the nested for loops that gives you the right order.

And so, I guess I take this line and put it on the top. And I take this line, the induction of the currents, put it inside the for loops, that is my bottom-up dp. OK, I'm actually going to write it down explicitly here for kicks. Should I bother? Uh, what the hell.

So first we do a for loop on m. Then, we're going to do the for loops on u in V. Now, inside the for loop, I want to compute this min. I could use this single line, but I'm going to rewrite it slightly to connect this back to shortest paths. Because this type of statement should look familiar from Dijkstra and Bellman-Ford.

This is called the relaxation step. OK. It probably would look more familiar if I wrote here w of x v. You could also write wxv. That's an alternative for both of these-- probably should write the same way.

But in either case, we call this a relaxation step, because-- it's kind of a technical reason but-- what we'd like to satisfy-- we know that shortest paths should satisfy the triangle inequality. If you look, there's three vertices involved here, u, v, and x. We're looking at the shortest path from u to v compared to the shortest path for u to x, and the shortest path from x to v. Certainly, the shortest way to get from u to v should be less than or equal to the shortest path of u to x plus x to v, because one way to get from u to v is to go via x.

So this if condition would be a violation of the triangle inequality. It means we definitely do not have the right distance estimates if duv is greater than ux plus xv. OK. So if it's greater, we're going to set it equal. Because here, we know a way to get from u to v via x. We know that it's possible to do this, assuming our d values are always upper bounds on reality. Then, this will be an upper bound on the best way to get from u to v.

So this is clearly a valid thing to do. Relaxations are never bad. If you start high, these will always improve your shortest paths, so you get better estimates. And that's exactly what Dijkstra and Bellman-Ford did, maybe with w instead of d, but they're all about fixing the triangle inequality.

And in general and optimization, there's this notion of, if you have a constraint, an inequality constraint like the triangle inequality, and it's violated, then you try to fix it by the successive relaxations. So that's where the term comes from-- doesn't really matter here, but all of our shortest path algorithms are going to do relaxations. All the shortest path algorithms I know do relaxations.

So this is familiar, but it's also doing the same thing here. I just expanded out the min as a for loop over x, each time checking whether each successive entry is better than what I had already. And if it is, I update. So in the end, this will compute a min, more or less.

I cheated also because I omitted the superscripts here. If I put m here, and m minus 1, here and 1 here, it would be exactly the same algorithm. I'm omitting all the superscripts, because it can only help me. Relaxing more can only get better. And if I was guaranteed correct over there, I'll still be guaranteed correct over here.

You have to improve the invariant, but you never-- relaxation is always safe. If you start with upper bounds, you always remain upper bounds. You're doing at least the relaxations over there. And so you will in the end compute the correct shortest path weights. The advantage of that, mainly, is I save space. And also, it's simpler.

So now I only need quadratic space. If I had a superscript, I'd need cubic space. Right? So I did a little simplification going from the five step process to here-- both of the polynomial time and space, but this is a little bit, a little bit better.

But how slow is this algorithm? How long does it take? Yeah? V cubed, that would be great. V to the fourth, yeah, good. Sadly, we're not doing so great yet-- still V to the fourth. V to the fourth, I guess I already knew how to do. That was if I just run Bellman-Ford V times, I already knew how to do V to the fourth.

So I haven't actually improved anything. But at least you see, it's all dynamic programming in there. So n here is the size of V. That's probably the first time. Cool. I omitted 0, because there was the base case. That's done separately in the line that I didn't write. So that was dp one.

Time for dp two, unless there are questions? Everyone clear so far? Yeah?

STUDENT: When you iterate over x, why do you do you every [INAUDIBLE]

PROFESSOR: As opposed to--?

STUDENT: Like, just adjacent vertices [INAUDIBLE]

PROFESSOR: Oh, yeah. OK. Good, fair question-- why do I iterate over all vertices, not just the incoming? If I'm writing w of xv, I could afford to just say, just consider the incoming vertices. And that would let me improve probably from V to the fourth to V cubed times-- V squared times E, I think, if you do the arithmetic right. You could do that. It would be better.

For dense graphs, it's not going to matter. For sparse graphs it will improve V squared to E, basically. But we're going to do even better, so I'm not going to try to optimize now. But, good question.

When I say this at the moment, if there is no edge from x to v, I'm imagining that w of xv equals infinity. So that will never be the minimum choice to use a non-edge. I should say that. If there's no edge here, I define the weight to be infinity. That will just make algorithms cleaner to write. But you could optimize it the way you said.

So, where were you? Yeah. Ah, perfect. OK. Other questions? More Frisbee practice? No? OK.

So that was dp one. Let me do dp two. Not yet, sorry-- diversion. Diversion is matrix multiplication. Before I get to dp two, I want to talk about matrix multiplication. This is a cool connection. It won't help us directly for shortest paths, but still pretty good. And it will help-- it will solve another problem especially fast.

So shortest paths is also closely linked to matrix multiplication, a problem we've seen a couple of times, first in the FFT lecture, and then in the randomization lecture for checking matrix multiplies. So you remember, you're given two matrices, A and B. And you want to compute C equals A times B. You've seen Strassen's algorithm to do this.

There's also these-- here A and B are squared. OK. So the n by n, product will be n by n. So standard approach for this is n cubed. With Strassen, if I can remember, you can get n to the 2.807. And if you use CopperSmith Winograd, you get 2.376. And then, if you use the new Vassilevska-William's algorithm, you get n to the 2.3728 and so on.

And that's the best algorithm we know now. There's some evidence maybe you can get 2 plus epsilon for any epsilon. Turns out, those are going to help us too much here.

But what I want to show is that matrix multiplication is essentially doing this, if you redefine what plus and dot mean. We redefine addition and multiplication-- talk about whether that's valid in the moment-- so remember what is matrix multiplication? Cij is a dot product of a row and a column. So that's aik with bklj. K equals 1 to n. OK. Now that sum looks a lot like that min. Actually, more like the way I wrote it over here, with the d's instead of the w's. This-- right-

- x is the thing that's varying here. So this is like, aik plus bkj, except, I have plus here, whereas I have times over here. And I have a min out here, but I have a sum over here.

So it sounds crazy, but let's define-- be very confusing if I said define dot equals plus, so I'm going to define a new world called circle world. So if I put a circle around a dot, what I mean is plus. And if I put a circle around a plus, what I mean is min. OK? So now, if I put a circle around this dot, I mean circle everything. So I've got to circle the summation, circle this thing. So then I get shortest paths. Crazy. So all right, I'm going to define d to the m-th power.

I should probably circle-- whatever. Slightly different. OK, I want to define, like, three things at once. So let me write them down, and then talk about them. So many infinities. All right.

OK. I guess, I should write this too.

OK. If I define the vert-- suppose I number the vertices 1 through n, OK? I just assume all vertices are an integer between 1 and n. So then, I can actually express things in a matrix, namely the weight matrix. This kind of defines the graph, especially if I say wij is infinity if there is no edge. Then, this is the matrix of all pairwise edge weights. For every i and j, I have a weight of ij. That gives me a matrix, once I set V to be 1 through n.

Now, I'm also defining this distance estimate matrix. So remember, we defined duvm-- I'm going to now call it dijm, because the vertices are integers. That is, the weight of the shortest path using, at most, m edges. If I define it that way, then I can put it into a matrix, which is for all pairs of vertices ij. What is the distance, shortest pathways that uses it at most m edges? That gives me a matrix, d parenthesis m.

Then, I claim that if I take circle product between d of m minus 1 and w, that is exactly what's happening here, if you stare at it long enough. This is the inner product between row u of d to the m minus 1, and column v of w. And that's exactly what this circle product will compute. So this is dp.

But when you look at that statement, that's saying that d to the parentheses m is really w to the circle power m, right? This is a definition in some sense of power, of exponentiation, using circle product. So when I circle the exponent, that means I'm doing circle exponentiation in circle land, OK? OK so far?

So this is circle land. So you might say, well, then I should compute these products using matrix multiplication. Now, just to see how good we're doing, if I execute this operation n times, because I have to get to d to the n minus 1-- so it's basically d to the n. If I do this product n times, and for each one of I spend n cubed time, then I get an n to the four algorithm. Same algorithm in fact, exactly the same algorithm-- I've just expressed it in this new language.

OK, there are two ideas on the table though. One is, maybe I could use a better matrix multiplication algorithm. Let's shelve that for a moment. The other possibility is, well, maybe I can exponentiate faster than multiplying by myself n times, or multiplying by w n times.

How should I do it? Repeated squaring, good. You've seen that probably in 006. Repeated squaring idea is, we take-- to compute w-- well, I take w to the 0. I multiply it by w to the 0. Sorry, circle 0-- that's this thing. Oh, that seems weird. Let's start with 1. 1 seems better.

I'm not going to get much if I multiply that by itself. I should get exactly the same matrix. So I take the circle product between w to the 1, w to the 1. That gives me w to the 2. And then, I take w to the 2 times w to the 2. Everything's circled. I get w to the 4. Oh cool, I doubled my exponent with one multiplication. If I take w to the 4 by w to the 4, I get w to the 8, and so on.

My goal is to get to n, so I have to do this log n times. Log n squaring operations, each squaring operation is an n cubed thing. So this is repeated squaring.

And I get V cubed log V-- finally, an improvement. So we went from V to the 4, which was in the dense case the same performance as Bellman-Ford, running it V times. But now in the dense case, I'm getting V cubed log V, which is actually pretty good. It's not quite V cubed, but close. All right.

I'm pointing at V cubed. This is actually the one result that is not optimal. This is the one we want to improve. But we're kind of-- we're in this space right now. We're getting close to as good is this algorithm, the Johnson's algorithm. But we still a log V Factor.

So this is great, just by translating into matrix multiplication. Now technically, you have to check that repeated squaring actually gives you the same result. Basically, this works because products are associative. Circle products of matrices are associative, which works because circle land is a semi-ring.

If you want the abstract algebra, a ring is something that you wear on your a finger. No. A ring is an algebra where you define plus and times, and you have distributivity. Semi-ring, there's no minus, because min has no inverse. There's no way from the min to re-compute the arguments, right? No matter what you apply to it, you can't-- you've lost information. So that's the semi-ring. Normally, you have a minus. But semi-ring is enough for the repeated squaring to give you the right answer.

However, semi-ring is not enough for all these fancy algorithms. So if you look at Strassen's algorithm, the one you've seen, it uses minus. There's no way to get around that, as far as we know. So if you have no minus, n cubed is the best we know how to do. So sadly, we cannot improve beyond this with this technique. It sucks, but that's life.

However, we can do something. If we just change the problem, there is another problem which this is the best way to do. So let me briefly tell you about that problem. It's called transitive closure.

Transitive closure is, I just want to know is there a path from i to j. So it's going to be 1, if there exists a path from i to j. And it's going to be 0 otherwise. OK. I guess it's kind of like if you set all the weights to 0 or infinity. Then, either there's going to be as 0 way path, or there's no path, meaning there's an infinite way path.

So it's not quite the same. Here, I want 1 and 0. I flipped. It used to be, this was infinity, and this was 0. This is one saying there is a path from i and j, 0 otherwise. If I write it this way, and then I think about what I need to do here, it is still in some sense plus and min, but not really. Because I just want to know, is there a path? So if I have a way to get there and a way to get there, instead of adding up those values, really I'm taking some other operator. So I want to know OR.

Yeah, exactly-- who said OR? Yeah, all right, tough one. Close, close, close.

So here, we have basically a circle product is OR, and circle sum is AND. OK? I mean plus and min would work, but it's a little bit nicer over here. Sorry, it's the other way around, I think. It's definitely Booleans. We want to know there is a way to get to x, and then from x to where we're going. That's an AND. And then, to get a path in general, it has to work for some x. So that's the OR.

And this is a ring. And once you're ring, you have negation. You can apply Vassilevska-Williams. And you solve this problem in n to the 2.3728. And if I just make a little change in the dot dot dot, I can absorb the log. So you could put a log n here. And it's log n if you get the exponent exactly right. But if you just tweak the exponent by 0.00000001, that's bigger than log n. So we usually omit the log there. Cool.

Transitive closure-- so it's a problem you didn't know you want to solve, but it is actually a common problem. And this is the best way we know how to solve it for dense graphs. OK, it beats, you know, V cubed. This is the algorithm we're aiming for for dense graphs. For sparse graphs, we can do better. But for dense graphs, this is better.

Finally, we get to go to dynamic programming number two, also known as the Floyd-Warshall algorithm. So we had this dp in V the fourth. If we forget about transitive closure, we've now are down to V cubed log V. Our next goal is to achieve V cubed, no log V. Let's do that.

So again, I'm going to express it in my five steps. First step is, what are subproblems? And this is the key difference, and the key insight for Floyd-Warshall is to redefine the dij problems. To avoid conflict, I'm going to call them cij, or in this case, cuv, because here, the matrix product view will not work, I think. Yeah, it won't work. So it's a totally different universe.

I'm still going to assume that my vertices are numbered 1 through n. And now, the idea is, first I'm going to think about the graph formed by the vertices 1 though k, roughly. And I want to know for every vertex u and every vertex v, what is the shortest path from u to v, or the weight of the shortest path from u to v that only uses intermediate vertices from 1 through k. So actually, u and v might not be-- they might be larger than k. But I want all the vertices in the path to be 1 through k.

This is a different way to slice up my space, and it's the right way. This is going to do a factor of n better. It turns out, and that's just an insight you get from trying all the dp's you could think of. And eventually, Floyd and Warshall found this one, I think in the '70s. So it was easier back then to get a new result. But I mean, this is very clever-- so very cool idea.

So now the question is, what should I guess? Before I guessed what the last edge was. That's not going to be so useful here. Can anyone think of a different thing to guess? We're trying to solve this problem where I get to use vertices 1 through k, and presumably I want to use subproblems that involve smaller k, that say involve vertices 1 through k minus 1. So vertex k is relevant. What should I guess about vertex k?

Yeah?

STUDENT: Guess that vertex k is the [INAUDIBLE]

PROFESSOR: You want to guess vertex k is the i-th intermediate vertex. That would work, but I would need to parameterize by i here, and I lose another factor of n if I do that. So I'd like to avoid that. That is a good idea. Yeah?

STUDENT: [INAUDIBLE] visit k, before you visit v.

PROFESSOR: You're going to guess that I visit k, and then I go to where I'm trying to go. OK. That's not a-- OK. That's a statement. But to guess, I should have multiple choices. What's my other choice?

STUDENT: [INAUDIBLE]

PROFESSOR: Yes. So either I use vertex k, or I don't. That's the guess-- is k in the path at all from u to v? So that's a weaker thing than saying, k is at position i in the path. Here I'm just saying, is k in the path at all?

And that's nice, because as you say, I already know how to get there without using k. Because that's cuvk minus 1. And then, you just also have to consider the situation where I go to k, and then I leave. So the recurrence is going to be cuvk is the min of two things. One is when k is not in the path, that's cuvk minus 1.

And the other option is that I go to x first-- or sorry, I go to k first. It used to be x. Now, I've renamed it k. I don't know why. k minus 1-- and then I go from k to the v-- k minus 1. That's it.

Min of two things-- before, I was taking the min of n things. Before, there were n choices for my guess. Now, there are two choices for my guess. Number of subproblems is the same, still V cubed. But the guessing part and the recurrence part is now constant time instead of linear time. So I'm now V cubed time-- progress.

OK? This is pretty cool. The old dp led us to this world of matrix multiplication. That's why I covered it. This new dp is just a different way of thinking about it-- turns out to be faster, just by log factor, but a little bit faster.

I need some base cases-- cuv of 0 is going to be-- now it's the weight of the edge uv. It's a different base case. Before, I was using 0 edges. Now, it's not using any intermediate vertices. So that's how the weights come into the picture, because actually there are no weights of edges up here. So that's a little weird. The only place the weights come in is when k equals 0.

This is in some sense still relaxation, but it's a little bit weirder, little bit different order. I mean the key thing here is, because the way we set up these subproblems with the intermediate vertices, we know k is the only vertex in question. Before it's like, well, I don't know where you go at the end. But now we know that either k is in there, or it's not. And in each case, we can compute it using smaller subproblems and so we save that linear factor.

STUDENT: Is this only for [INAUDIBLE] graphs, or is does it also [INAUDIBLE]

PROFESSOR: This is for directed graphs. u and v are ordered here. And this is the weight from u to v-- will work just as well. It's probably a little bit instructive to write this down as nested for loops again. Why not? Because then, you'll see it's just relaxation again.

So I'll even write the base case here, because it's very simple. We're doing k in order, let's say. These are really the same kinds of for loops. But I'll write them slightly differently, because here we care about the order slightly. Here, we do care about the order. Here, we don't care about the order. Vertices, and all we're saying is-- almost exactly the same code as before.

This is, again, just a relaxation step. We're just relaxing different edges in different orders, basically, because k is evolved in here. We do that for k equals 1. Then for k equals 2, and so on. But in the end, it's just relaxations, so you can use that to prove that this actually computes the right shortest paths. I won't do that here. But clearly, cubic time instead of quartic. Pretty cool.

That's Floyd-Warshall. It's very simple. And so a lot of people-- if you need to solve all-pairs shortest paths in dense graphs, this is the best we know how to do. So this is what you should implement. It's like, five lines of code. And you achieve this bound.

But for our sparse graphs, we can do better. And the rest of lecture is going to be about Johnson's algorithm, where for sparse graphs we're going to get closer to quadratic time. We're going to match running Dijkstra, and it's V squared log V plus E times V, sorry. So when E is small, that's going to be close to quadratic. When E is big, it's going to be cubic, again. So we'll never be worse than this Floyd-Warshall algorithm. But for sparse graphs it's better.

OK. Johnson's algorithm. I was going to make some joke about Johnson and Johnson, but I will pass. So Johnson's algorithm-- I mean, dp is five steps, but Johnson's algorithm's only three steps-- clearly simpler. It's actually much more complicated, but it's all about what the steps are.

So here's the crazy idea in Johnson's algorithm. We're going to change the weights on the edges. And to do that, we're going to assign weights to vertices.

We're going to choose a function h. Think of it as a height function, I guess, that maps vertices to real numbers. And then, we're going to define w sub h of u,v. This is a new way to think about edge weights that depends on h that's defined in a simple way. It's the old edge weight plus h of u minus h of v. You could define it the other way, but it's better to be consistent here.

So this is a way to tweak edge weights. For every edge-- this is for directed graphs clearly. For u, u is the beginning, the head of the-- I don't know if it's the head or the tail-- the beginning of the edge. v is the end of the edge. I'm going to add on the height of h, and subtract out the height of v. OK?

Why? Because that's the definition. I want this to be greater than or equal to 0. That's the such that. I want to assign a function h, so that these new weights are all greater or equal to 0. This is for all u and v. Why would I do that?

STUDENT: To use Dijkstra instead of--

PROFESSOR: To use Dijkstra instead of Bellman-Ford, exactly. So that's step 2. Run Dijkstra on, I guess, the usual graph. But now, this new weight function, w sub h, if all the weights are non-negative, I can run Dijkstra. So this will give me what I call the shortest path sub h of u comma v for all u and v.

It doesn't give me the actual shortest path weights I want. It gives me the shortest path weights using this wh. But I claim that's almost the same. I claim that this re-weighting preserves which paths are shortest. Because-- so in particular, I claim that delta of u,v is delta sub h of u,v-- should be the other way-- minus h of u plus h of v. OK.

If this was a single edge, you can see I'm just cancelling off these terms. But in fact, I claim for a whole path, every path from u to v gets changed by exactly the same amount. So this is a claim about the shortest path-- in effect, a claim for every path from u to v, shortest or not. If I measure it in regular weights w, versus weights w sub h, the only difference is this fixed amount, which depends only on u and v-- does not depend on the path. And therefore, which paths are shortest are preserved.

And so when we compute these shortest path weights, we can translate them back to what they should be in the original weighting function. And furthermore, if you have parent pointers, and you actually find the paths, the paths will be the same. Shortest paths will be the same. OK. So let's prove that claim. It's actually really simple.

Let's look at a path from u to v. So I'm going to label the vertices along the path. V0 is going to be u. That's the first one, then V1, then V2, and so on. Let's say path has length k. And Vk is v. OK, that's just a generic path from u to v. And now, I want to compute the w sub h of that path. Excuse me.

So the weight of a path is just the sum of the weights the edges. So I could write this as a sum from i equals 1 to k of w sub h of Vi minus 1 comma Vi. I think that works, got to be careful not to get the indices wrong. OK, now, w sub h is defined to be this thing-- w plus h of u minus h of v. So this is the sum i equals 1 to k of w Vi minus 1 Vi plus h of Vi minus 1 minus h of Vi. What does the sum do? Telescope.

So success-- this Vi is going to-- this negative h of Vi is going to cancel with the plus h of Vi minus 1 in the next term, except for the very first one and the very last one. So this is going to be this sum, which is just the weight of the path according to regular weight function, plus h of V0 minus h of Vk. And that is just the weight to the path plus h of u minus h of v. Did I get it right? Nope. Yes?

STUDENT: [INAUDIBLE] subtract [INAUDIBLE]

PROFESSOR: But it's not-- it's opposite of what I claimed. So right, because it's the other side, good. This has h on the right hand side. This has not h on the left hand side. But here, I have h on the left hand side, and not h on the right hand side. So if I flip it around, if I take these two terms, put them on the left hand side, then I get this with the right sign. Cool, whew. Self-consistent. OK.

So this was talking about are arbitrary path. And so this is proving the stronger thing I said, that every path gets lengthened by this function, which is purely a function of the endpoints. So in particular, that means the shortest path in w land will still be the shortest path in w sub h land-- slightly less cool name than circle land, but oh well.

All right, so this means shortest paths are preserved. Shortest paths are still shortest. And therefore, if I look at the delta function, which is about the shortest path weights, this claim holds. So that's the proof of the claim. Cool.

There's one gaping problem with this algorithm, which is how in the world do you find this h? If we could find h, then we know we could run Dijkstra, and we can do this thing. And Dijkstra is going to cost the VE plus V squared log V. I didn't say it, but we run V times Dijkstra. All right, we run it V times. That's going to take V squared log V plus VE to do. This is just going to take quadratic time, V squared to update all the weights, update all the delta functions.

The missing step is how do we find this weight function? I claim this problem of finding h that has this property, is very closely related to shortest paths. It's weird, but we're going to use shortest paths to solve shortest paths. So let's do it.

Step 1, finding h. What I want to do, so I want to have w of u,v-- let me just copy that down-- plus h of u plus h of v to be greater than or equal to 0. Whoops, minus. I'm going to put the h's on to the right hand side, and then flip it all around. So this is like saying h of v minus h of u is less than or equal to w of u,v for all u and v.

This is a problem we want to solve, right? w's are given. h's are unknowns. This is called a system of difference constraints.

If you've heard about linear programming, for example, this is a special case of linear programming. Don't worry if you haven't heard, because this is an easy special case. We're going to solve it much faster than we know how to solve lineal programs. It's a particular kind of thing. This is actually useful problem.

You could think of, these are maybe times that various events happen. And these are constraints about pairs of them. Says, well, the start time of this event minus the end time of that event should be less than or equal to 1 second. You can use this to do temporal programming, if you could solve these systems.

We're going to solve these systems, when they have a solution. They don't always have a solution, which is a bit weird, because we're relying on them always having a solution. How can that be? Negative weight cycles. This is all going to work, when we don't have negative weight cycles.

And that's exactly going to be the case when this system of difference constraints has no solution. So let me show you that in a couple of steps. First theorem is that if the graph V,E,w has a negative weight cycle, then that system has no solution-- no solution to the difference constraints. This is going to be, again, an easy proof, kind of similar to last one actually.

So consider a negative weight cycle. Let's call it V0, to V1, to V2, to Vk, back to V0. So the claim is the sum of these weights is negative. And now, I'm just going to write down these constraints, which are supposed to have a solution, or maybe they won't.

So if it has a solution, then this must be true, where u and v are plugged into be Vi and Vi minus 1, because those are all edges. So I'm going to write h of V1 minus h of V0 is less than or equal to w of V0 wV1. And then h of V2 minus h of V1 less than or equal to w of V1 V2. Repeat that k times, I'm going to get h of Vk minus h of Vk minus 1. And then, the last one, the wrap around h of V0 minus h of Vk w of V-- did I get it right-- Vk V0.

What do I do with these inequalities? Sum them. Time for a *Good Will Hunting* moment-- do you remember? I hope we've all seen *Good Will Hunting.* I don't have a janitor here, so I have to do all cancels by hand-- and this.

So I end up with 0 at the bottom. Everything cancels, and then, over here I have less than or equal to the weight of the whole cycle. I'm just adding up the weight of the cycle. I didn't give the cycle a name-- call it C.

Now, the cycle has negative weight. So this is less than zero, strictly less than zero. So we're saying that 0 is strictly less than 0. That's not true. So that means there's no way to get all of these constraints simultaneously true-- proof by a contradiction.

So that establishes a connection in the direction we don't want it. What we want is they're converse, which is if there's no negative weight cycle, then there is a solution. Luckily, that is also true. But this is a little easier to see. So now, we do the other half.

OK. And this will-- I mean, it's going to be constructive proof. So we're going to actually know how to solve this problem with an algorithm. So it's going to be-- there is a negative weight cycle if and only if there's no solution. So in particular, the case we care about is if there's no negative weight cycle, then there is a solution. We kind of care about both, but this is the more practical direction. So let's prove it.

You can already see-- you've seen that there's a connection in negative weight cycles. Now, I'm going to show there's a real connection to shortest paths. Negative weight cycles are just kind of a symptom of the shortest paths being involved. So now, we're going to use shortest paths.

Suppose we have some graph. I'm going to draw a simple little graph with weights. What I'd like to do is compute shortest path from a single source in this graph. The question is, which source? Because none of the vertices-- I guess in this case, this would be a pretty good source, because it can reach. From here, I can get to every node.

But in general-- maybe there's another vertex here-- draw a more complicated picture. It could be, there's no one vertex that can reach all the others. For example, it may be the graph is disconnected. That's a good example. So there's no single source that can reach everywhere.

I really want to reach everywhere. So what am I going to do? Add a new source. Call it s. I'm going to add an edge to every other vertex. Now, I can get everywhere from s. OK? What are the weights? 0. 0 sounds good. I don't want to change the weights, in some sense. So I put 0, and add 0 to everything. That's not going to change much.

Now, notice I add no cycles to the graph. So if there were no negative weight cycles before, still no negative weight cycles, because the cycles are the same as they were before. But now, from s I can reach everywhere. If there's no negative weight cycles, that means there's a well-defined, finite value for delta of s comma v for all V. And that is h.

What? It's crazy man. All right, so we add s to V. We're going to add s comma v to e for all V. That's the old V. And I'm going to set the weight of s comma v to be 0 for all V. OK, that's what i just did. And so now, delta of s comma v is finite for all V. It's not plus infinity, because I know there is-- it's got to be less than 0, right? I can get from s to everywhere. So it's less than positive infinity. It's also not negative infinity, because I've assumed there's no negative weight cycles anywhere.

So I'm going to let h of v be delta of s,v. I claim that just works, magically. That's insane. Every time I see it, it's like, got to be crazy man-- crazy but correct. That's Johnson. It's like you just pray that this happens, and it works. Why would it happen? Why would it be that-- what do we want to say-- w of u,v plus h of u minus h of v-- we want this to be greater than or equal to 0.

I guess I had already rewritten this way. Neither way is the right way, so it doesn't matter. So let's see. We have a weight of u,v. We have the shortest path from s to u. And we have this the shortest pathway from s to v. We want that to be greater than or equal to 0. Why? Put this over there, and I get delta s,v is less than or equal to delta of s,u plus w of u,v, which is? Triangle inequality, which is true.

It turns out, this thing we've been staring at for so long is actually just triangle inequality. So of course we want to compute shortest paths, because shortest paths satisfy triangle inequality. The whole name of the game in shortest paths is to find a place where you don't satisfy triangle inequality and fix it.

So if it makes sense, if that's possible to do, Bellman-Ford will do it. So how we're going to do step 1? We're going to run Bellman-Ford once. We're going to add this source vertex, so that there is a clear source to run Bellman-Ford from, and then, run Bellman-Ford from there only. That will give us a weight function for the vertices, namely how long does it take to get from s to those vertices. Those weights will actually all be negative.

But then, we're going to modify all the edge weights according to this formula, which negates some of them. So some of them are going to go up some, some of them are going to go down. It's kind of weird. But when we're done, all of the weights will be non-negative because we had triangle inequality. And now, we can run Dijkstra from every vertex.

So it's like we bootstrap a little bit. We run Bellman-Ford once, because we know it handles negative weights. It will also tell us if there are any negative weight cycles. That's why we want this theorem. Maybe Bellman-Ford says, I can't satisfy triangle inequality, because there's a negative weight cycle. I don't know what to do. Then, we know, well actually, then there was no solution.

OK, that's kind of interesting. But then, we'll have to deal with the shortest paths-- sorry-- deal with those negative weight cycles. I won't cover how to do that here. But you can. And otherwise, there's no negative weight cycles, then Bellman-Ford finds valid h. Then, we plug that h into here. Then, we have non-negative weights.

So in VE time, we've reduced to the non-negative all-pair shortest paths. And then, we run Dijkstra V times. Then, we get almost our answers, but we have to modify them to get back the correct weights on our shortest paths.

And so we computed shortest paths in V squared log V plus VE, because this is how much Dijkstra costs, and because Bellman-Ford takes less time. We're good. That's the magic. And that's all-pairs shortest paths.