-- shortest paths. This is the finale.

Hopefully it was worth waiting for.

Remind you there's a quiz coming up soon, you should be studying for it. There's no problem set due at the same time as the quiz because you should be studying now. It's a take-home exam.

It's required that you come to class on Monday.

Of course, you'll all come, but everyone watching at home should also come next Monday to get the quiz.

It's the required lecture. So, we need a bit of a recap in the trilogy so far. So, the last two lectures, the last two episodes, or about single source shortest paths. So, we wanted to find the shortest path from a source vertex to every other vertex.

And, we saw a few algorithms for this.

Here's some recap. We saw in the unweighted case, that was sort of the easiest where all the edge weights were one. Then we could use breadth first search. And this costs what we call linear time in the graph world, the number of vertices plus the number of edges. The next simplest case, perhaps, is nonnegative edge weights.

And in that case, what algorithm do we use?

Dijkstra, all right, everyone's awake.

Several answers at once, great.

So this takes almost linear time if you use a good heap structure, so, V log V plus E.

And, in the general case, general weights, we would use Bellman-Ford which you saw.

And that costs VE, good, OK, which is quite a bit worse. This is ignoring log factors.

Dijkstra is basically linear time, Bellman-Ford you're quadratic if you have a connected graph.

So, in the sparse case, when E is order V, this is about linear. This is about quadratic.

In the dense case, when E is about V^2, this is quadratic, and this is cubic.

So, Dijkstra and Bellman-Ford are separated by about an order of V factor, which is pretty bad.

OK, but that's the best we know how to do for single source shortest paths, negative edge weights, Bellman-Ford is the best. We also saw in recitation the case of a DAG. And there, what do you do?

Topological sort, yeah.

So, you can do a topological sort to get an ordering on the vertices. That you run Bellman-Ford, one round. This is one way to think of what's going on. You run Bellman-Ford in the order given by the topological sort, which is once, and you get a linear time algorithm.

So, DAG is another case where we know how to do well even with weights. Unweighted, we can also do linear time. But most of the time, though, will be, so you should keep these in mind in the quiz. When you get a shortest path problem, or what you end up determining is the shortest path problem, think about what's the best algorithm you can use in that case? OK, so that's single source shortest paths. And so, in our evolution of the Death Star, initially it was just nonnegative edge weights.

Then we got negative edge weights.

Today, the Death Star challenges us with all pair shortest paths, where we want to know the shortest path weight between every pair of vertices.

OK, so let's get some quick results.

What could we do with this case?

So, for example, suppose I have an unweighted graph. Any suggestions of how I should compute all pair shortest paths? Between every pair of vertices, I want to know the shortest path weight.

BFS, a couple more words? Yeah?

Right, BFS V times. OK, I'll say V times BFS, OK? So, the running time would be V^2 plus V times E, yeah, which is assuming your graph is connected, V times E.

OK, good. That's probably about the best algorithm we know for unweighted graphs.

So, a lot of these are going to sort of be the obvious answer.

You take your single source algorithm, you run it V times.

That's the best you can do, OK, or the best we know how to do. This is not so bad.

This is like one iteration of Bellman-Ford, for comparison. We definitely need at least, like, V^2 time, because the

size of the output is V^2, shortest path weight we have to compute.

So, this is not perfect, but pretty good.

And we are not going to improve on that.

So, nonnegative edge weights: the natural thing to do is to run Dijkstra V times, OK, no big surprise.

And the running time of that is, well, V times E again, plus V^2, log V, which is also not too bad.

I mean, it's basically the same as running BFS.

And then, there's the log factor.

If you ignore the log factor, this is the dominant term.

And, I mean, this had an [added?] V^2 as well. So, these are both pretty good.

I mean, this is kind of neat. Essentially, the time it takes to run one Bellman-Ford plus a log factor, you can compute all pair shortest paths if you have nonnegative edge weights. So, I mean, comparing all pairs to signal source, this seems a lot better, except we can only handle nonnegative edge weights.

OK, so now let's think about the general case.

Well, this is the focus of today, and here's where we can actually make an improvement. So the obvious thing is V times Bellman-Ford, which would cost V^2 times E.

And that's pretty pitiful, and we're going to try to improve that to something closer to that nonnegative edge weight bound. So it turns out, here, we can actually make an improvement whereas in these special cases, we really can't do much better.

OK, I don't have a good intuition why, but it's the case. So, we'll cover something like three algorithms today for this problem.

The last one will be the best, but along the way we'll see some nice connections between shortest paths and dynamic programming, which we haven't really seen yet.

We've seen shortest path, and applying greedy algorithms to it, but today will actually do dynamic programming.

The intuition is that with all pair shortest paths, there's more potential subproblem reuse.

We've got to compute the shortest path from x to y for all x and y. Maybe we can reuse those shortest paths in computing other shortest paths.

OK, there's a bit more reusability, let's say.

OK, let me quickly define all pair shortest paths formally, because we're going to change our notation slightly.

It's because we care about all pairs.

So, as usual, the input is directed graph, so, vertices and edges. We're going to say that the vertices are labeled one to n for convenience because with all pairs, we're going to think of things more as an n by n matrix instead of edges in some sense because it doesn't help to think any more in terms of adjacency lists.

And, you have edge weights as usual.

This is what makes it interesting.

Some of them are going to be negative.

So, w maps to every real number, and the target output is a shortest path matrix. So, this is now an n by n matrix. So, n is just the number of vertices of shortest path weights.

So, delta of i, j is the shortest path weight from i to j for all pairs of vertices.

So this, you could represent as an n by n matrix in particular.

OK, so now let's start doing algorithms.

So, we have this very simple algorithm, V times Bellman-Ford, V^2 times E, and just for comparison's sake, I'm going to say, let me rewrite that, V times Bellman-Ford gives us this running time of V^2 E, and I'm going to think about the case where, let's just say the graph is dense, meeting that the number of edges is quadratic, and the number of vertices.

So in that case, this will take V^4 time, which is pretty slow. We'd like to do better.

So, first goal would just be to beat V^4, V hypercubed, I guess. OK, and we are going to use dynamic programming to do that. Or at least that's what the motivation will come from. It will take us a while before we can even beat V^4, which is maybe a bit pathetic, but it takes some clever insights, let's say.

OK, so I'm going to introduce a bit more notation for this graph. So, I'm going to think about the weighted adjacency matrix. So, I don't think we've really seen this in lecture before, although I think it's in the appendix.

What that means, so normally adjacency matrix is like one if there's an edge, and zero if there isn't. And this is in a digraph, so you have to be a little bit careful.

Here, these values, the entries in the matrix, are going to be the weights of the edges.

OK, this is this if ij is an edge.

So, if ij is an edge in the graph, and it's going to be infinity if there is no edge. OK, in terms of shortest paths, this is a more useful way to represent the graph.

All right, and so this includes everything that we need from here. And now we just have to think about it as a matrix. Matrices will be a useful tool in a little while. OK, so now I'm going to define some sub problems. And, there's different ways that you could define what's going on in the shortest paths problem. OK, the natural thing is I want to go from vertex i to vertex j. What's the shortest path?

OK, we need to refine the sub problems a little but more than that. Not surprising.

And if you think about my analogy to Bellman-Ford, what Bellman-Ford does is it tries to build longer and longer shortest paths. But here, length is in terms of the number of edges. So, first, it builds shortest paths of length one. We've proven the first round it does that. The second round, it provides all shortest paths of length two, of count two, and so on.

We'd like to do that sort of analogously, and try to reuse things a little bit more. So, I'm going to say d_ij^(m) is the weight of the shortest path from i to j with some restriction involving m. So: shortest path from i to j using at most m edges. OK, for example, if m is zero, then we don't have to really think very hard to find all shortest paths of length zero.

OK, they use zero edges, I should say.

So, Bellman-Ford sort of tells us how to go from m to m plus one. So, let's just figure that out.

So one thing we know from the Bellman-Ford analysis is if we look at d_ij^(m-1), we know that in some sense the longest shortest path of relevance, unless you have negative weight cycle, the longest shortest path of relevance is when m equals n minus one because that's the longest simple path you can have.

So, this should be a shortest path weight from i to j, and it would be no matter what larger value you put in the superscript. This should be delta of i comma j if there's no negative weight cycles.

OK, so this feels good for dynamic programming.

This will give us the answer if we can compute this for all m.

Then we'll have the shortest path weights in particular.

We need a way to detect negative weight cycles, but let's not worry about that too much for now.

There are negative weights, but let's just assume for now there's no negative weight cycles.

OK, and we get a recursion recurrence.

And the base case is when m equals zero.

This is pretty easy. They have the same vertices, the weight of zero, and otherwise it's infinity.

OK, and then the actual recursion is for m.

OK, if I got this right, this is a pretty easy, intuitive recursion for d_ij^(m) is a min of smaller things in terms of n minus one. I'll just show the picture, and then the proof of that claim should be obvious.

So, this is proof by picture. So, we have on the one hand, I over here, and j over here.

We want to know the shortest path from i to j.

And, we want to use, at most, m edges.

So, the idea is, well, you could use m minus one edges to get somewhere. So this is, at most, m minus one edges, some other place, and we'll call it k. So this is a candidate for k.

And then you could take the edge directly from k to j.

So, this costs A_k^j, and this costs DIK m minus one.

OK, and that's a candidate path of length that uses, at most, m edges from I to j. And this is essentially just considering all of them. OK, so there's sort of many paths we are considering. All of these are candidate values of k. We are taking them in over all k as intermediate nodes, whatever.

So there they are. We take the best such path.

That should encompass all shortest paths.

And this is essentially sort of what Bellman-Ford is doing, although not exactly. We also sort of want to think about,

well, what if I just go directly with, say, m minus one edges? What if there is no edge here that I want to use, in some sense?

Well, we always think about there being, and the way the A's are defined, there's always this zero weight edge to yourself.

So, you could just take a path that's shorter, go from d i to j, and j is a particular value of k that we might consider, and then take a zero weight edge at the end from A and jj. OK, so this really encompasses everything. So that's a pretty trivial claim. OK, now once we have such a recursion, we get a dynamic program.

I mean, there, this is it in some sense.

It's written recursively. You can write a bottom up.

And I would like to write it bottom up it little bit because while it doesn't look like it, this is a relaxation.

This is yet another relaxation algorithm.

So, I'll give you, so, this is sort of the algorithm. This is not a very interesting algorithm. So, you don't have to write it all down if you don't feel like it.

It's probably not even in the book.

This is just an intermediate step.

So, we loop over all m. That's sort of the outermost thing to do. I want to build longer and longer paths, and this vaguely corresponds to Bellman-Ford, although it's actually worse than Bellman-Ford. But hey, what the heck?

It's a stepping stone. OK, then for all i and j, and then we want to compute this min.

So, we'll just loop over all k, and relax.

And, here's where we're actually computing the min.

And, it's a relaxation, is the point.

This is our good friend, the relaxation step, relaxing edge. Well, it's not, yeah. I guess we're relaxing edge kj, or something, except we don't have the same clear notion. I mean, it's a particular thing that we're relaxing. It's not just a single edge because we don't have a single source anymore.

It's now relative to source I, we are relaxing the edge kj, something like that. But this is clearly a relaxation. We are

just making the triangle inequality true if it wasn't before.

The tribal inequality has got to hold between all pairs.

And that's just implementing this min, right?

You're taking d ij. You take the min of what it was before in some sense. That was one of the possibilities we considered when we looked at the zero weight edge. We say, well, or you could go from i to some k in some way that we knew how to before, and then add on the edge, and check whether that's better if it's better, set our current estimate to that. And, you do this for all k.

In particular, you might actually compute something smaller than this min because I didn't put superscripts up here. But that's just making paths even better. OK, so you have to argue that relaxation is always a good thing to do.

So, by not putting superscripts, maybe I do some more relaxation, but more relaxation never hurts us. You can still argue correctness using this claim. So, it's not quite the direct implementation, but there you go, dynamic programming algorithm. The main reason I'll write it down: so you see that it's a relaxation, and you see the running time is n^4, OK, which is certainly no better than Bellman-Ford. Bellman-Ford was n^4 even in the dense case, and it's a little better in the sparse case. So: not doing so great.

But it's a start. OK, it gets our dynamic programming minds thinking. And, we'll get a better dynamic program in a moment. But first, there's actually something useful we can do with this formulation, and I guess I'll ask, but I'll be really impressed if anyone can see. Does this formula look like anything else that you've seen in any context, mathematical or algorithmic? Have you seen that recurrence anywhere else? OK, not exactly as stated, but similar. I'm sure if you thought about it for awhile, you could come up with it.

Any answers? I didn't think you would be very intuitive, but the answer is matrix multiplication. And it may now be obvious to you, or it may not. You have to think with the right quirky mind. Then it's obvious that it's matrix multiplication. Remember, matrix multiplication, we have A, B, and C. They're all n by n matrices.

And, we want to compute C equals A times B.

And what that meant was, well, c_ij was a sum over all k of a_ik times b_kj. All right, that was our definition of matrix multiplication.

And that formula looks kind of like this one.

I mean, notice the subscripts: ik and kj.

Now, the operators are a little different.

Here, we're multiplying the inside things and adding them all together. There, we're adding the inside things and taking them in. But other than that, it's the same. OK, weird, but here we go.

So, the connection to shortest paths is you replace these operators. So, let's take matrix multiplication and replace, what should I do first, plus this thing with min. So, why not just change the operators, replace dot with plus?

This is just a different algebra to work in, where plus actually means min, and dot actually means plus.

So, you have to check that things sort of work out in that context, but if we do that, then we get that c_ij is the min overall k of a_ik plus, a bit messy here, b_kj. And that looks like what we actually want to compute, here, for one value of m, you have to sort of do this m times.

But this conceptually is d_ij^(m), and this is d_ik^(m-1). So, this is looking like a matrix product, which is kind of cool.

So, if we sort of plug in this claim, then, and think about things as matrices, the recurrence gives us, and I'll just write this now at matrix form, that d^(m) is d^(m) minus one, funny product, A.

All right, so these are the weights.

These were the weighted adjacency matrix.

This was the previous d value. This is the new d value.

So, I'll just rewrite that in matrix form with capital letters. OK, I have the circle up things that are using this funny algebra, so, in particular, circled product. OK, so that's kind of nifty.

We know something about computing matrix multiplications. We can do it in n^3 time.

If we were a bit fancier, maybe we could do it in sub-cubic time. So, we could try to sort of use this connection. And, well, think about what we are computing here. We are saying, well, d to the m is the previous one times A.

So, what is d^(m)? Is that some other algebraic notion that we know? Yeah, it's the exponent.

We're taking A, and we want to raise it to the power, m, with this funny notion of product.

So, in other words, d to the m is really just A to the m in a funny way. So, I'll circle it, OK? So, that sounds good.

We also know how to compute powers of things relatively quickly, if you remember how. OK, for this notion, this power notion, to make sense, I should say what A to the zero means.

And so, I need some kind of identity matrix.

And for here, the identity matrix is this one, if I get it right. So, it has zeros along the diagonal, and infinities everywhere else.

OK, that sort of just to match this definition.

d_ij zero should be zeros on the diagonals and infinity everywhere else. But you can check this is actually an identity. If you multiply it with this funny multiplication against any other matrix, you get the matrix back. Nothing changes.

This really is a valid identity matrix.

And, I should mention that for A to the m to make sense, you really knew that your product operation is associative. So, actually A to the m circled makes sense because circled multiplication is associative, and you can check that; not hard because, I mean, min is associative, and addition is associative, and all sorts of good stuff. And, you have some kind of distributivity property. And, this is, in turn, because the real numbers with, and get the right order here, with min as your addition operation, and plus as your multiplication operation is a closed semi-ring. So, if ever you want to know when powers make sense, this is a good rule.

If you have a closed semi-ring, then matrix products on that semi-ring will give you an associative operator, and then, good, you can take products.

OK, that's just some formalism. So now, we have some intuition.

The question is, what's the right.

Algorithm? There are many possible answers, some of which are right, some of which are not.

So, we have this connection to matrix products, and we have a connection to matrix powers.

And, we have algorithms for both.

The question is, what should we do?

So, all we need to do now is to compute A to the funny power, n minus one. n minus one is when we get shortest paths, assuming we have no negative weight cycles. In fact, we could compute a larger power than n minus one.

Once you get beyond n minus one, multipling by A doesn't change you anymore.

So, how should we do it? OK, you're not giving any smart answers. I'll give the stupid answer.

You could say, well, I take A.

I multiply it by A. Then I multiply it by A, and I multiply it by A, and I use normal, boring matrix to multiplication.

So, I do, like, n minus two, standard matrix multiplies. So, standard multiply costs, like, n^3. And I'm doing n of them.

So, this gives me an n^4 algorithm, and compute all the shortest pathways in n^4. Woohoo!

OK, no improvement. So, how can I do better?

Right, natural thing to try which sadly does not work, is to use the sub cubic matrix multiply algorithm.

We will, in some sense, get there in a moment with a somewhat simpler problem. But, it's actually not known how to compute shortest paths using fast matrix multiplication like Strassen's system algorithm.

But, good suggestion. OK, you have to think about why it doesn't work, and I'll tell you.

It's not obvious, so it's a perfectly reasonable suggestion. But in this context it doesn't quite work. It will come up in a few moments. The problem is, Strassen requires the notion of subtraction.

And here, addition is min. And, there's no inverse to min.

Once you take the arguments, you can't sort of undo a min.

OK, so there's no notion of subtraction, so it's not known how to pull that off, sadly.

So, what other tricks do we have up our sleeve?

Yeah? Divide and conquer, log n powering, yeah, repeated squaring.

That works. Good, we had a fancy way.

If you had a number n, you sort of looked at the binary number representation of n, and you either squared the number or squared it and added another factor of A.

Here, we don't even have to be smart about it.

OK, we can just compute, we really only have to think about powers of two. What we want to know, and I'm going to need a bigger font here because there's multiple levels of subscripts, A to the circled power, two to the ceiling of log n. Actually, n minus one would be enough. But there you go.

You can write n if you didn't leave yourself enough space like me, n the ceiling, n the circle.

This just means the next power of two after n minus one, two to the ceiling log. So, we don't have to go directly to n minus one. We can go further because anything farther than n minus one is still just the shortest pathways. If you look at the definition, and you know that your paths are simple, which is true if you have no negative weight cycles, then fine, just go farther.

Why not? And so, to compute this, we just do ceiling of log n minus one products, just take A squared, and then take the result and square it; take the result and square it.

So, this is order log n squares.

And, we don't know how to use Strassen, but we can use the boring, standard multiply of n^3, and that gives us n^3 log n running time, OK, which finally is something that beats Bellman-Ford in the dense case.

OK, in the dense case, Bellman-Ford was n^4.

Here we get n^3 log n, finally something better.

In the sparse case, it's about the same, maybe a little worse. E is order V.

Then we're going to get, like, V3 for Bellman-Ford.

Here, we get n^3 log n. OK, after log factors, this is an improvement some of the time.

OK, it's about the same the other times.

Another nifty thing that you get for free out of this, is you can detect negative weight cycles.

So, here's a bit of a puzzle. How would I detect, after I compute this product, A to the power to ceiling log n minus one, how would I know if I found a negative weight cycle?

What would that mean it this matrix of all their shortest paths of, at most, a certain length?

If I found a cycle, what would have to be in that matrix? Yeah?

Right, so I could, for example, take this thing, multiply it by A, see if the matrix changed at all.

Right, that works fine. That's what we do in Bellman-Ford. It's an even simpler thing.

It's already there. You don't have to multiply.

But that's the same running time.

That's a good answer. The diagonal would have a negative value, yeah.

So, this is just a cute thing. Both approaches would work, can detect a negative weight cycle just by looking at the diagonal of the matrix. You just look for a negative value in the diagonal. OK.

So, that's algorithm one, let's say.

I mean, we've seen several that are all bad, but I'll call this number one. OK, we'll see two more.

This is the only one that will, well, I shouldn't say that.

Fine, there we go. So, this is one dynamic program that wasn't so helpful, except it showed us a connection to matrix multiplication, which is interesting. We'll see why it's useful a little bit more. But, it bled to this nasty four nested loops. And, using this trick, we got down to n^3 log n. Let's try, just for n^3.

OK, just get rid of that log. It's annoying.

It makes you a little bit worse than Bellman-Ford, and the sparse case. So, let's just erase one of these nested loops. OK, I want to do that.

OK, obviously that algorithm doesn't work because it's for first decay, and it's not defined, but, you know, I've got enough variables.

Why don't I just define k to the m?

OK, it turns out that works. I'll do it from scratch, but why not? I don't know if that's how Floyd and Warshall came up with their algorithm, but here you go. Here's Floyd-Warshall.

The idea is to define the subproblems a little bit more cleverly so that to compute one of these values, you don't have to take the min of n things.

I just want to take the min of two things.

If I could do that, and I still only have n^3 subproblems, then I would have n^3 time.

So, all right, the running time of dynamic program is number of subproblems times the time to compute the recurrence for one subproblem. So, here's linear times n^3, and we want n^3 times constant. That would be good.

So that's Floyd-Warshall. So, here's the way we're going to redefine c_ij. Or I guess, there it was called d_ij. Good, so we're going to define something new. So, c_ij superscript k is now going to be the weight of the shortest path from I to j as before. Notice I used the superscript k instead of m because I want k and m to be the same thing.

Deep. OK, now, here's the new constraint. I want all intermediate vertices along the path, meeting all vertices except for I and j at the beginning and the end to have a small label.

So, they should be in the set from one up to k.

And this is where we are really using that our vertices are labeled one up to m. So, I'm going to say, well, first think about the shortest paths that don't use any other vertices. That's when k is zero.

Then think about all the shortest paths that maybe they use vertex one. And then think about the shortest paths that maybe use vertex one or vertex two.

Why not? You could define it in this way. It turns out, then when you increase k, you only have to think about one new vertex. Here, we had to take min over all k. Now we know which k to look at.

OK, maybe that made sense. Maybe it's not quite obvious yet. But I'm going to redo this claim, redo a recurrence. So, maybe first I should say some obvious things. So, if I want delta of ij of the shortest pathway, well, just take all the vertices. So, take c_ij superscript n.

That's everything. And this even works, this is true even if you have a negative weight cycle.

Although, again, we're going to sort of ignore negative weight cycles as long as we can detect them.

And, another simple case is if you have, well, c_ij to zero. Let me put that in the claim to be a little bit more consistent here.

So, here's the new claim. If we want to compute c_ij superscript zero, what is it?

Superscript zero means I really shouldn't use any intermediate vertices. So, this has a very simple answer, a three letter answer. So, it's not zero.

It's four letters. What's that?

Nil. No, not working yet.

It has some subscripts, too.

So, the definition would be, what's the shortest path weight from I to j when you're not allowed to use any intermediate vertices? Sorry?

So, yeah, it has a very simple name.

That's the tricky part. All right, so if i equals j, [LAUGHTER] you're clever, right, open bracket i equals j means one, well, OK.

It sort of works, but it's not quite right.

In fact, I want infinity if i does not equal j.

And I want to zero if i equals j, a_ij, good.

I think it's a_ij. It should be, right? Maybe I'm wrong.

Right, a_ij. So it's essentially not what I said. That's the point.

If i does not equal j, you still have to think about a single edge connecting i to j, right?

OK, so that's a bit of a subtlety.

This is only intermediate vertices, so you could still go from i to j via a single edge. That will cost a_ij.

If there is an edge: infinity.

If there isn't one: that is a_ij.

So, OK, that gets us started. And then, we want a recurrence.

And, the recurrence is, well, maybe you get away with all the vertices that you had before.

So, if you want to know paths that you had before, so if you want to know paths that use one up to k, maybe I just use one up to k minus one.

You could try that. Or, you could try using k.

So, either you use k or you don't.

If you don't, it's got to be this.

If you do, then you've got to go to k.

So why not go to k at the end? So, you go from I to k using the previous vertices. Obviously, you don't want to repeat k in there. And then, you go from k to j somehow using vertices that are not k.

This should be pretty intuitive.

Again, I can draw a picture. So, either you never go to k, and that's this wiggly line. You go from i to j using things only one up to k minus one. In other words, here we have to use one up to k.

So, this just means don't use k.

So, that's this thing. Or, you use k somewhere in the middle there. OK, it's got to be one of the two. And in this case, you go from i to k using only smaller vertices, because you don't want to repeat k.

And here, you go from k to j using only smaller labeled vertices. So, every path is one of the two. So, we take the shortest of these two subproblems. That's the answer.

So, now we have a min of two things.

It takes constant time to compute.

So, we get a cubic algorithm. So, let me write it down.

So, this is the Floyd-Warshall algorithm.

I'll write the name again. You give it a matrix A.

That's all it really needs to know.

It codes everything. You copy C to A.

That's the warm up. Right at time zero, C equals A. And then you just have these three loops for every value of k, for every value of i, and for every value of j. You compute that min.

And if you think about it a little bit, that min is a relaxation. Surprise, surprise.

So, that is the Floyd-Warshall algorithm.

And, the running time is clearly n^3, three nested loops, constant time inside. So, we're finally getting something that is never worse than Bellman-Ford.

In the sparse case, it's the same.

And anything denser, the number of edges is super linear. This is strictly better than Bellman-Ford. And, it's better than everything we've seen so far for all pair, shortest paths.

And, this handles negative weights; very simple algorithm, even simpler than the one before.

It's just relaxation within three loops.

What more could you ask for? And we need to check that this is indeed what min we're computing here, except that the superscripts are omitted.

That's, again, a bit of hand waving a bit.

It's OK to omit subscripts because that can only mean that you're doing more relaxation techniques should be.

Doing more relaxations can never hurt you.

In particular, we do all the ones that we have to. Therefore, we find the shortest path weights. And, again, here, we're assuming that there is no negative weight cycles.

It shouldn't be hard to find them, but you have to think about that a little bit. OK, you could run another round of Bellman-Ford, see if it relaxes in a new edges again. For example, I think there's no nifty trick for that version.

And, we're going to cover, that's our second algorithm for all pairs shortest paths. Before we go up to the third algorithm, which is going to be the cleverest of them all, the one Ring to rule them all, to switch trilogies, we're going to take a little bit of a diversion, side story, whatever, and talk about transitive closure briefly. This is just a good thing to know about. And, it relates to the algorithms we've seen so far. So, here's a transitive closure problem. I give you a directed graph, and for all pair vertices, i and j, I want to compute this number. It's one if there's a path from i to j. From i to j, OK, and then zero otherwise. OK, this is sort of like a boring adjacency matrix with no weights, except it's about paths instead of being about edges. OK, so how can I compute this?

That's very simple. How should I compute this?

This gives me a graph in some sense.

This is adjacency matrix of a new graph called the transitive closure of my input graph. So, breadth first search, yeah, good. So, all I need to do is find shortest paths, and if the weights come out infinity, then there's no path. If it's less than infinity, that there's a path. And so here, so you are saying maybe I don't care about the weights, so I

can run breadth first search n times, and that will work indeed. So, if we do B times B of S, so it's maybe weird that I'm covering here in the middle, but it's just an interlude. So, we have, then, something like V times E. OK, you can run any of these algorithms. You could take Floyd-Warshall for example. Why not?

OK, then it would just be V^ I mean, you could run in any of these algorithms with weights of one or zero, and just check whether the values are infinity or not.

So, I mean, t_ij equals zero, if and only if the shortest path weight from i to j is infinity.

So, just solve this. This is an easier problem than shortest paths. It is, in fact, strictly easier in a certain sense, because what's going on with transitive closure, and I just want to mention this out of interest because transitive closure is a useful thing to know about. Essentially, what we are doing, let me get this right, is using a different set of operators.

We're using or and and, a logical or and and instead of min and plus, OK, because we want to know, if you think about a relaxation, in some sense, maybe I should think about it in terms of this min.

So, if I want to know, is there a pathway from I to j that uses vertices labeled one through k in the middle?

Well, either there is a path that doesn't use the vertex k, or there is a path that uses k, and then it would have to look like that. OK, so there would have to be a path here, and there would have to be a path there.

So, the min and plus get replaced with or and and.

And if you remember, this used to be plus, and this used to be product in the matrix world.

So, plus is now like or. And, multiply is now like and, which sounds very good, right?

Plus does feel like or, and multiply does feel like and if you live in a zero-one world. So, in fact, this is not quite the field Z mod two, but this is a good, nice, field to work in. This is the Boolean world.

So, I'll just write Boole. Good old Boole knows all about this. It's like his master's thesis, I think, talking about Boolean algebra.

And, this actually means that you can use fast matrix multiply. You can use Strassen's algorithm, and the fancier algorithms, and you can compute the transitive closure in subcubic time.

So, this is sub cubic if the edges are sparse.

But, it's cubic in the worst case if there are lots of edges.

This is cubic. You can actually do better using Strassen. So, I'll just say you can do it. No details here.

I think it should be, so in fact, there is a theorem.

This is probably not in the textbook, but there's a theorem that says transitive closure is just as hard as matrix multiply.

OK, they are equivalent. Their running times are the same. We don't know how long it takes to do a matrix multiply over a field.

It's somewhere between n^2 and n^2.3.

But, whatever the answer is: same for transitive closure.

OK, there's the interlude. And that's where we actually get to use Strassen and friends. Remember, Strassen was n to the log base two of seven algorithm. Remember that, especially on the final. Those are things you should have at the tip of your tongue. OK, the last algorithm we're going to cover is really going to build on what we saw last time: Johnson's algorithm. And, I've lost some of the running times here. But, when we had unweighted graphs, we could do all pairs really fast, just as fast as a single source Bellman-Ford. That's kind of nifty.

We don't know how to improve Bellman-Ford in the single source case. So, we can't really help to get anything better than V times E. And, if you remember running V times Dijkstra, V times Dijkstra was about the same. So, just put this in the recall bubble here: V times Dijkstra would give us V times E plus V^2 log V. And, if you ignore that log factor, this is just VE. OK, so this was really good.

Dijkstra was great. And this was for nonnegative edge weights. So, with negative edge weights, somehow we'd like to get the same running time.

Now, how might I get the same running time?

Well, it would be really nice if I could use Dijkstra.

Of course, Dijkstra doesn't work with negative weights.

So what could I do? What would I hope to do?

What could I hope to? Suppose I want, in the middle of the algorithm, it says run Dijkstra n times.

Then, what should I do to prepare for that?

Make all the weights positive, or nonnegative.

Why not, right? We're being wishful thinking.

That's what we'll do. So, this is called graph re-weighting. And, what's cool is we actually already know how to do it. We just don't know that we know how to do it. But I know that we know that we know how to do it. You don't yet know that we know that I know that we know how to do it.

So, it turns out you can re-weight the vertices.

So, at the end of the last class someone asked me, can you just, like, add the same weight to all the edges? That doesn't work.

Not quite, because different paths have different numbers of edges. What we are going to do is add a particular weight to each vertex.

What does that mean? Well, because we really only have weights on the edges, here's what well do.

We'll re-weight each edge, so, (u,v), let's say, going to go back into graph speak instead of matrix speak, (u,v) instead of I and j, and we'll call this modified weight w_h. h is our function.

It gives us a number for every vertex.

And, it's just going to be the old weight of that edge plus the weight of the start vertex minus the weight of the terminating vertex. I'm sure these have good names.

One of these is the head, and the other is the tail, but I can never remember which. OK, so we've directed edge (u,v). Just add one of them; subtract the other. And, it's a directed edge, so that's a consistent definition.

OK, so that's called re-weighting.

Now, this is actually a theorem.

If you do this, then, let's say, for any vertices, u and v in the graph, for any two vertices, all paths from u to v have the same weight as they did before, well, not quite.

They have the same re-weighting.

So, if you look at all the different paths and you say, well, what's the difference between vh, well, sorry, let's say delta, which is the old shortest paths, and deltas of h, which is the shortest path weights according to this new weight function, then that difference is the same.

So, we'll say that all these paths are re-weighted by the same amounts. OK, this is actually a statement about all paths, not just shortest paths.

There we go. OK, to how many people is this obvious already? A few, yeah, it is. And what's the one word?

OK, it's maybe not that obvious.

All right, shout out the word when you figure it out.

Meanwhile, I'll write out this rather verbose proof.

There's a one word proof, still waiting.

So, let's just take one of these paths that starts at u and ends at v. Take any path.

We're just going to see what its new weight is relative to its old weight. And so, let's just write out w_h of the path, which we define in the usual way as the sum over all edges of the new weight of the edge from v_i to v_i plus one. Do you have the word?

No? Tough puzzle then, OK. So that's the definition of the weight of a path. And, then we know this thing is just w of v_i, v_i plus one.

I'll get it right, plus the weight of the first vertex, plus, sorry, the re-weighting of v_i minus the re-weighting of v_i plus one.

This is all in parentheses that's summed over I.

Now I need the magic word. Telescopes, good.

Now this is obvious: each of these telescopes with an extra previous, except the very beginning and the very end. So, this is the sum of these weights of edges, but then outside the sum, we have plus h of v_1, and minus h of v_k.

OK, those guys don't quite cancel.

We're not looking at a cycle, just a path.

And, this thing is just w of the path, as this is the normal weight of the path. And so the change, the difference between w_h of P and w of P is this thing, which is just h of u minus h of v.

And, the point is that's the same as long as you fix the endpoints, u and v, of the shortest path, you're changing this path weight by the same thing for all paths. This is for any path from u to v, and that proves the theorem. So, the one word here was telescopes. These change in weights telescope over any path. Therefore, if we want to find shortest paths, you just find the shortest paths in this re-weighted version, and then you just change it by this one amount. You subtract off this amount instead of adding it. That will give you the shortest path weight in the original weights.

OK, so this is a tool. We now know how to change weights in the graph. But what we really want is to change weights in the graph so that the weights all come out nonnegative. OK, how do we do that?

Why in the world would there be a function, h, that makes all the edge weights nonnegative?

It doesn't make sense. It turns out we already know.

So, I should write down this consequence.

Let me get this in the right order.

So in particular, the shortest path changes by this amount. And if you want to know this value, you just move the stuff to the other side.

So, we compute deltas of h, then we can compute delta.

That's the consequence here. How many people here pronounce this word corollary? OK, and how many people pronounce it corollary? Yeah, we are alone.

Usually get at least one other student, and they're usually Canadian or British or something.

I think that the accent. So, I always avoid pronouncing his word unless I really think, it's corollary, and get it right. I at least say Z not Zed.

OK, here we go. So, what we want to do is find one of these functions. I mean, let's just write down what we could hope to have. We want to find a re-weighted function, h, the signs of weight to each vertex such that w_h of (u,v) is nonnegative. That would be great for all edges, all (u,v) in E. OK, then we could run Dijkstra.

We could run Dijkstra, get the delta h's, and then just undo the re-weighting, and get what we want. And, that is Johnson's algorithm. The claim is that this is always possible. OK, why should it always be possible? Well, let's look at this constraint. w_h of (u,v) is that.

So, it's w of (u,v) plus h of u minus h of V should be nonnegative. Let me rewrite this a little bit. I'm going to put

these guys over here. That would be the right thing, h of v minus h of u is less than or equal to w of (u,v).

Does that look familiar? Did I get it right?

It should be right. Anyone seen that inequality before? Yeah, yes, correct answer.

OK, where? In a previous lecture?

In the previous lecture. What is this called if I replace h with x? Charles knows.

Good, anyone else remember all the way back to episode two?

I know there was a weekend. What's this operator called?

Not subtraction but, I think I heard it, oh man. All right, I'll tell you.

It's a difference constraint, all right?

This is the difference operator.

OK, it's our good friend difference constraints.

So, this is what we want to satisfy.

We have a system of difference constraints.

h of V minus h of u should be, we want to find these.

These are our unknowns. Subject to these constraints, we are given the w's. Now, we know in these difference constraints are satisfiable.

Can someone tell me when these constraints are satisfiable?

We know exactly when for any set of difference constraints.

You've got to remember the math.

Terminology, I can understand.

It's hard to remember words unless you're a linguist, perhaps. So, when is the system of different constraints satisfiable?

All right, you should definitely, very good.

[LAUGHTER] Yes, very good.

Someone brought their lecture notes: when the constraint graph has no negative weight cycles. Good, thank you.

Now, what is the constraint graph?

OK, this has a one letter answer more or less.

I'll accept the one letter answer.

What? A?

A: close. G.

Yeah, I mean, same thing.

Yeah, so the constraint graph is essentially G.

Actually, it is G. The constraint graph is G, good. And, we prove this by adding a new source for text, and connecting that to everyone. But that's sort of beside the point. That was in order to actually satisfy them. But this is our characterization. So, if we assume that there are no negative weight cycles in our graph, which we've been doing all the time, then we know that this thing is satisfiable. Therefore, there is an assignment of this h's. There is a re-weighting that makes all the weights nonnegative.

Then we can run Dijkstra. OK, we're done.

Isn't that cool? And how do we satisfy these constraints? We know how to do that with one run of Bellman-Ford, which costs order VE, which is less than V times Dijkstra.

So, that's it, write down the details somewhere.

So, this is Johnson's algorithm.

This is the fanciest of them all.

It will be our fastest, all pairs shortest path algorithm. So, the claim is, we can find a function, h, from V to R such that the modified weight of every edge is nonnegative for every edge, (u,v), in our graph. And, we do that using Bellman-Ford to solve the difference constraints.

These are exactly the different constraints that we were born to solve that we learned to solve last time.

The graphs here are corresponding exactly if you look back at the definition. Or, Bellman-Ford will tell us that there is a negative weight cycle.

OK, great, so it's not that we really have to assume that there is no negative weight cycle. We'll get to know.

And if your fancy, you can actually figure out the minus infinities from this. But, at this point, I just want to think about the case where there is no negative weight cycle. But if there is, we can find out that it exists, and that just tell the user.

OK, then we'd stop. Otherwise, there is no negative weight cycle. Therefore, there is an assignment that gives is nonnegative edge weights.

So, we just use it. We use it to run Dijkstra.

So, step two is, oh, I should say the running time of all this is V times E. So, we're just running Bellman-Ford on exactly the input graph.

Plus, we add a source, if you recall, to solve a set of difference constraints.

You add a source vertex, S, connected to everyone at weight zero, run Bellman-Ford from there because we don't have a source here. We just have a graph.

We want to know all pairs. So, this, you can use to find whether there is a negative weight cycle anywhere.

Or, we get this magic assignment.

So now, w_h is nonnegative, so we can run Dijkstra on w_h.

We'll say, using w_h, so you compute w_h.

That takes linear time. And, we run Dijkstra for each possible source. I'll write this out explicitly.

We've had this in our minds several times.

But, when we said n times Dijkstra over n times BFS, here it is. We want to compute delta sub h now, of (u,v) for all V, and we do this separately for all u. And so, the running time here is VE plus V^2 log V. This is just V times the running time of Dijkstra, which is E plus V log V.

OK, it happens that this term is the same as this one, which is nice, because that means step one costs us nothing asymptotically. OK, and then, last step is, well, now we know delta h.

We just need to compute delta. So, for each pair of vertices, we'll call it (u,v), we just compute what the original weights would be, so what delta (u,v) is.

And we can do that using this corollary.

It's just delta sub h of (u,v) minus h of u plus h of v.

I got the signs right. Yeah, so this takes V^2 time, also dwarfed by the running time of Dijkstra.

So, the overall running time of Johnson's algorithm is just the running time of step two, running Dijkstra n times -- -- which is pretty cool. When it comes to single source shortest paths, Bellman-Ford is the best thing for general weights. Dijkstra is the best thing for nonnegative weights. But for all pair shortest paths, we can skirt the whole negative weight issue by using this magic we saw from Bellman-Ford.

But now, running Dijkstra n times, which is still the best thing we know how to do, pretty much, for the all pairs nonnegative weights, now we can do it for general weights too, which is a pretty nice combination of all the techniques we've seen.

In the trilogy, and along the way, we saw lots of dynamic programming, which is always good practice. Any questions?

This is the last new content lecture before the quiz.

On Wednesday it will be quiz review, if I recall correctly.

And then it's Thanksgiving, so there's no recitation.

And then the quiz starts on Monday.

So, study up. See you then.