[SQUEAKING][RUSTLING][CLICKING]

JASON KU: Welcome, everybody, to our lecture 14 of 6.006. This is our last lecture on graph algorithms, in particular, the last lecture we'll have on weighted shortest paths. But we're going to talk about a slightly different problem today, different than single source shortest paths. We're going to be talking about all-pairs shortest paths.

But first, let's review our single source shortest paths algorithms. We had BFS, DAG relaxation, Dijkstra, which we saw last time, which gets pretty close to linear. V log V plus E is pretty close to linear. It's much closer to linear than the general algorithm we have for solving single source shortest paths, namely, Bellman-Ford, which is a little bit more like quadratic.

This is like the difference between-- for sparse graphs, this is like the difference between sorting, using insertion sort and n squared, and merge sort in N log N, for example. We're going to get actually quite a big bonus for large input sizes by using Dijkstra when we can.

Today, we're going to be focusing on this new problem called all-pairs shortest paths. It's not really complicated. Instead of having a single source, we are essentially wanting to, given an input-- this is our weighted graph, where we've got a graph V, E, and we've got a weight function from the edges to the integers.

This is our general weighted graph. We want our output to be something like, the shortest path distance from u to v for every u and v in our vortex set. That's what we want to return.

Now, there's one caveat here that, if there's a negative weight cycle in our graph-- in other words, if any of these delta u, v's is minus infinity, there's a negative rate cycle in our graph. So unless, I guess-- or abort if G contains a negative weight cycle. So we're not actually going to worry about negative weight cycles in today's class.

If we have a graph, it could have negative weights. These are any integers. It could include negative weight edges. But as long as all of our path distances are bounded from below, none of them are negative infinity, we don't have any negative weight cycles, then I want you to output all of these shortest path distances.

Now, in particular, this output could-- any of these outputs needs to have size theta of V squared. Because for every pair of vertices, I need to return to you a number, or infinity, or minus infinity or something like that. But we are not dealing with a case with minus infinity. The output could have size-- this is a theta here. It does have size V squared.

But in particular, it's at least V squared because I need to give a number for each pair of vertices. And so we couldn't hope for linear time in the size of this graph for this problem, right? Single source shortest paths, for certain versions of the problem, we need to read the graph. And so we need to use linear time.

But in this problem, our output has quadratic size in the number of vertices. So in a sense, we can't do better than this. We can't do better than quadratic. And actually, what's one way we could solve all-pairs shortest paths by using stuff we've already done in this class? That's why I put this slide up here.

Yeah, we could just solve a single source shortest paths algorithm from every vertex in my graph. That seems like a stupid thing to do. It's almost brute force on the vertices. But it's certainly a way we could solve this problem, in polynomial time.

And we could definitely solve it in order V squared E time, using Bellman-Ford. We just take V steps of Bellman-Ford and deal with a graph on any set of vertices. We can do better than this. We can do better than this for graphs that are special in some way. We can do V times V plus E, V times linear.

If our weights are positive and bounded, we can use BFS V times. Or if our graph doesn't have cycles, we could use DAG relaxation V times. Or if our graph had non-negative edge weights, we could get, basically, V squared log V plus V times E. And that's actually not bad.

In sparse graphs, this is what Bellman-Ford would give us. But if we had Dijkstra's, for example, if we had all positive edge weights-- or non-negative, sorry, we could get V squared log V plus V, E time. This is V times Dijkstra.

OK, so how do these running times compare? This is V times Bellman-Ford. This is V times Dijkstra. Let's just get a feel for this separation here. If we had a sparse graph where V is upper-bounded by the number of vertices, this one looks like V squared log V.

This one looks like V cubed. And we need to spend at least V squared time. So actually, this is really close to linear in the size of the graph, just off by a log factor, just like sorting would entail. And this one would have a linear factor. In the sparse graph, this would be a linear factor worse than this, instead of a logarithmic factor-- again, this linear to log separation.

We don't want to have to do this running time if we don't have to. That's the name of the game. And really, all we're going to do in this lecture is try to solve how we can make this running time faster by doing something a little bit more intelligent than running a single source shortest path algorithm from every vertex.

How are we going to do that? Well, we could-- let's see. What are we doing? Right. The idea here, if we had a graph-- should my graph be directed or undirected? I'm not sure. Let's see if we can make a directed graph. OK, so here's a directed graph.

Why do I not care about undirected graphs? Can anyone tell me? Yeah, it's because-- I don't care about undirected graphs because, if I had an undirected graph, I could detect whether I had negative weight cycles in constant time-- I'm sorry, in linear time. I could just check each edge, see if it has negative weight, because a negative weight edge, an undirected edge is a cycle of negative weight.

So I could just-- if it has any negative edge weights, I could return in linear time that it does, and I can abort. Or it has only positive weights, and I can still use Dijkstra. So that's all good. So we're only concerned about needing to run Bellman-Ford on directed graphs that potentially have negative edge weight.

OK, so here's a graph. Let's see. Is this a graph that I want? Sure. Let's say we've got that direction and this direction. Say we have a directed graph like this. And let's say this is s. This is our source. And we have weights being 2-- sorry, weights being 4, 1, 1, 2, 2, 2, 2.

So this is an example of a graph we might want to run all-pairs shortest paths on. Maybe we also have negative weights in this graph. In particular, this has a negative weight cycle. I don't want negative weight cycles, so I'm going to make this 0. So this graph doesn't have negative weight cycles. Great. That's true, great.

All right, so here's an example that we might want to compute shortest paths on. There's no s in all-pairs shortest paths. But I'm going to be talking about a couple of shortest paths from s in my next argument, so I'm just labeling that vertex as s.

OK, the claim-- the approach we're going to do, we're going to try to take a graph that has negative edge weights directed graph. We don't know if it has negative cycles or not yet. But we want to compute all-pairs shortest paths, not in this running time, but in this running time.

How could we do that? Well, maybe it's possible that we can change the weight of every edge so that they're all positive, but shortest paths are preserved. So basically, if a particular path-- like OK, the shortest path from s to t here is 1, 2, 3.

I could change edge weights in this graph. Say, for example, if I changed 1 to 0 here, that would still make this a shortest path. I haven't done-- I've reweighted the graph. Shortest paths have to be the same in this graph.

But now-- sorry. Yeah, this is not a shortest path. OK, I'll make that minus 2, and then these both 2, and I think this 4. Man, I really should have done my example beforehand.

OK, so this still doesn't have negative weight cycles. It has a negative weight edge. But this path is longer than this path. So when this was 1, this had length of 3, which was shorter than this path. That is length 4. OK, cool. So this is the shortest path from s to t.

I could change weights in this graph, for example, changing 2 to 3 and 1 to 0. That changed weights in my graph, but shortest paths remain the same. So maybe there's a way I could reweight my edges so that shortest paths stay the same, shortest paths are preserved. So I'm going to put this back to where it was.

So idea-- make edge weights non-negative while preserving shortest paths. In other words, just reweight the edges here so that shortest paths in G-- this is G-- after we reweight will go to some graph G prime, with the same combinatorial structure just different edge weights. And we want shortest paths-- we want these weights to all be non-negative. And we want shortest paths, if there's a shortest path in G, it continues to be a shortest path in the reweighted graph.

That's the goal for today. If we can do that transformation and this is not negative, then we're done. We're done because we can just run Dijkstra V times on that new graph, get the shortest path distances, construct a shortest path tree from those distances, and then traverse that tree in the original graph, and compute shortest paths along that tree. So that's the claim.

Claim-- we can compute distances in G-- so we're going to restrict G prime to have with non-negative edge weights. If we have such a G prime with non-negative edge weights, we can compute distances in G from distances in G prime in V times V plus E, which is smaller than our Dijkstra running time. This is our Dijkstra running time. And this is smaller than that.

So that would be fine, right? What do we do? We have our new graph G prime. It has all positive edge weights, so we run all-pairs shortest paths in here by just running Dijkstra V times. And then for each vertex s, we have some shortest path tree to all the things that it's connected to.

We can look at that path in G. This is the same common [INAUDIBLE] graph, just with different edge weights. We can look at that same tree. I'm not going to be able to draw the exact same tree here. But what I can do is I can just BFS or DFS along this tree.

And every time I have an edge, because each one of these is the shortest path in this tree, I can just compute every time I traverse an edge what its shortest path distance is in G in linear time for that vertex for that particular s. I do that over all s's, and I get this running time. So that's the goal here.

All right, so we first wanted to find-- we were wanting to make edge weights non-negative while preserving shortest paths. Because if we could do that, we could solve our original all-pairs shortest paths problem. So this is the proof sketch. But how can we do this?

But how? It doesn't even seem like this could be possible, generally. How can I just reweight the edges and maintain that the shortest path trees are the same? That seems hard to do. And in particular, if there's a negative weight cycle in this graph, it's impossible to do.

So claim-- not possible if G contains a negative weight cycle. OK, the exclamation point is just my comment here. So if G contains a negative weight cycle, then in particular, the shortest path distance or a shortest path in G-- if this is G, say we have a negative weight cycle directed cycle C here.

In particular, the shortest path from this vertex on the cycle to this vertex on the cycle, what is its shortest path? It's infinite. A shortest path is infinite around this cycle. You just keep going around the cycle over and over and over again because it has a negative weight. Weight of C is less than 0, strictly. That's what a negative weight cycle is.

OK, so the shortest path-- a shortest path from s to t has infinite length and, in particular, is non-simple. However-- so shortest path from s to t is non-simple. But as we proved in the last lecture, shortest paths in a graph with non-negative weights are what? Are simple, right? Because they're just shortest path trees. So that's a contradiction. So this is not possible.

So given a graph with negative weights but no negative cycle, it's still not clear how we could find such a reweighting of the graph. Can we do this? Well, we're going to exploit a little idea here. How can we transform the weights of a path?

Well, how-- what's a-- a silly idea, I have this silly idea. If I don't want negative edge weights in this graph-- ugh, this is messy in the back. You got edge weights 1, minus 2, 0, 1, 4, 5, and 1. There's only one negative edge weight here.

What if I just added a large number, or in particular, the negative of the smallest edge in my graph to every edge in my graph? Then I'll have a graph with non-negative weights. Fantastic. Why is that not a good idea?

Well, in particular, if I did that to this graph, if I added 2 to every edge, the weight of this path, which was the shortest path, changed from weight 3 to weight 9, because I added 2 for every edge. But this path, which wasn't a shortest path in the original graph-- it had weight 4-- increased only by 2. Now that is a shortest path. Or it's a shorter path than this one, so this one can't be a shortest path.

So that transformation, sure, would make all the weights non-negative, but would not preserve shortest paths. In particular, if I added the same edge weight to every edge, I will bias toward taking paths that have fewer edges, not just smaller weight. So that first idea doesn't work.

Idea-- add large number to each edge. This is bad. Makes weights non-negative, but does not preserve shortest paths. So this is not a good idea-- bad idea. Is there any way you can think of to modify the edge weights in a graph in any way that will preserve shortest paths?

So here is an idea for you, which is kind of this critical step in Johnson's and in a lot of graph transformation algorithms. If I have a vertex, say this middle guy, say v, every path from v goes through an outgoing edge of v. And every path going into v goes through an edge going into v.

I haven't said anything-- I've said very stupid things. But that observation is critical here. If I add a number-- or let me see if I got this right in terms of adding and subtracting. If I add a number to all outgoing edges from a vertex, and I subtract that same number from the weights of all of the incoming edges to that vertex, then every path from v is changed by the same amount, because every path from v goes through one of those outgoing edges.

And any path going into v has also changed by the same amount. In particular, it's changed by a negative, whatever we added to the outgoing edges. So such a transformation, adding a number from all the outgoing edges from a vertex and subtracting that same number from all the incoming edges, preserves shortest paths. That's a claim.

Idea-- this is a better idea. Given vertex v, add-- I'm going to put this on two lines. Add weight h to all outgoing edges, and subtract weight to all incoming edges. So that's the idea. And the claim is, this transformation, shortest paths are preserved under this transformation.

And why is that? It's kind of the exact same argument that I had over there. Proof-- consider any path in my graph, either if the path-- path could go through v many times, or it could go through not at all. If my path, if I have a path, in my original graph G, then with path weight w of pi-- this is my path-- it goes through v some number of times.

So I'm going to say this is going from s to t. If it crosses v-- if it never crosses v, if it never touches v, the vertex that I transformed, then I argue that the path weight is the same because I didn't do anything to edges that are in this path. Alternatively, this thing goes through v sometimes. If it goes through v in the middle, how is the weight of my path changed?

Well, it hasn't, because I added a number to all outgoing edges, so there's an outgoing edge here with weight I've changed by weight h. And there's an incoming edge here that I've changed by weight negative h. So these cancel out and you've got 0. So passing through a vertex doesn't change the weight of my path.

The only way I could change the weight of my path is if v is the start vertex or the end vertex. So it's possible that s is my vertex or t is my vertex. Well, for any path leaving v, I will have increased the weight of that path by h, because I added a weight h to all outgoing edges.

So again, while the path weight has changed, since all of the paths leaving v have changed by the same amount, a shortest path will still be a shortest path. And same goes for t. If t, the end vertex is v, I'll have subtracted h from all of my incoming edges, which means that any path ending at t, any directed path ending at t, also changes by the same value. And so shortest paths must be preserved. So shortest paths preserved.

So that's pretty cool transformation. I can assign for any vertex such a transformation which affects all of the edges surrounding it by this h additive factor, either added or subtracted. So maybe-- and I can do this independently for every vertex. The shortest paths were preserved by me doing this to one vertex. Then if I do it to another vertex, then shortest paths are still preserved. And let's prove that real quick.

What I'm going to do is I'm going to want to do this to give me flexibility for changing all the edge weights in my graph to have this property. I'm going to set-- or define a potential function h that maps vertices to integers. So this is the potential h of v. And then we're going to make a graph, G prime based on above transformation for each vertex in v.

So I'm going to set a number, an h for each vertex. These are independent now. And I'm going to add that potential to all outgoing edges. And I'm going to subtract that potential from all incoming edges. This transformation is going to preserve shortest paths. Let's actually be a little bit more rigorous that that's the case when we do this multiple times.

So claim-- shortest paths are still preserved. All right, well, that's, again, not so difficult to see. Let's consider a path from s to t. It passes through a bunch of vertices. I'm going to label these as v0 to vk so that I can kind of number them.

All right, this is v1 here. This is a directed path, v1, 2, 3, 4, all the way up to k. There are k edges in this graph. I claim to you that any path from v0 to vk, any shortest path from v0 to vk remains a shortest path after I reweight everything in this way.

So let's say this is path pi, and so it has weight w pi, which is really just the sum of all of the edge weights from vi minus 1 to vi, for i equals 1 to k. This is poor notation. This is the weight of the edge from the vi minus 1 to i. And we've got-- it indexes from 1-- that's the first edge-- to k, which is the last edge. So that's the weight of my path.

The weight of my transformed path-- I'm going to do it down here. It's a little iffy. The weight of my transformed path I'm going to say is the weight in this new weighted graph G prime. This weight of that same path-- it's the same path-- is just going to be the sum of all of the reweighted edges. So i equals 1 to k of my original weight of my edge, so from 0, i minus 1 to vi.

But what did I do? This edge is outgoing from vi minus 1. So it's outgoing, so I add that weight-- that potential, sorry. But that edge is also incoming into vi. So when I reweighted the thing, I got a subtraction of h, vi.

Now, what happens here in the sum, this term, if I just took the sum over this term, that's exactly my original pathway. So that's good. But you'll notice that this sum has k terms, and this sum has the subtraction of k other terms. But most of these terms are equal.

Along the path, all the incoming and outgoing edges cancel out. So we're left with only adding the potential at the starting vertex and subtracting the potential at the final vertex. So we've got, add h, v0 minus h, vk.

And why is that good? Well, that's good because every path from v0 to vk starts at v0 and ends at vk. That's just-- that's how it is. That's how we've defined paths going from v0 to vk. But every such path, we transform the weight of that path by adding a constant associated with the start and adding this value associated with the end.

And so every path going from v0 to vk changes by the same amount. And so if this path pi was shortest, it's still shortest in the reweighted graph because I've just changed all paths between those two vertices by the same amount. This is kind of like a telescoping argument here in that kind of proof.

Right. So we have, the weight changes. It could change, but it changes all paths between these two vertices by the same amount, which means that shortest paths are still shortest. Awesome.

OK, so the name of the game here is now, we have this really flexible tool. We have this tool where we can add or subtract weight from various edges. But we have to do so in a kind of localized, constrained way. We have to do the same thing around each vertex. But it seems like a powerful transformation technique that maybe we can get this thing that we want, which is a G prime, a reweighting of the graph where all the edge weights are positive or non-negative.

So does there exist an h such that the weights are all positive? What does that mean? w prime u, v, the weight in my new graph, in G prime, I want these modified weights, this modified weight of my graph, I want each of these to be non-negative. So does there exist such a thing? Huh.

Well, if I rearrange this equation a little bit, this side, I get something that looks like this. h of v needs to be less than or equal to h of u, plus the weight of some edge from u to v. What does that look like? That looks like almost exactly the definition of the triangle inequality.

Shortest path from some vertex here and its shortest path distance from the same vertex here, this is just a statement of the triangle inequality. So if we can set these h's to be the shortest path distance from some vertex and those shortest path distances are finite, and not minus infinity, then this thing will hold by triangle inequality. And in particular, if we were to reweight the edges based on those values of h, then we get new edge rates that are non-negative.

Awesome. OK. But there might not be any vertex from which we can access, which we can reach all vertices in the graph. In particular, my graph might not even be connected. If I want this property, I need all of these-- I don't gain any information if these things are infinite. It's exhaustively true. Infinity is-- I don't even know how to compare infinity and infinity plus a constant. I don't know.

So I need all of these things to be finite. So how can I make those things finite? So here's the next idea. Add new vertex s with 0-weight edge to every vertex, V in V. We take our original graph. We add a new super node or auxiliary vertex s, with a 0-weight edge to every vertex in my graph.

What does that look like? This is like-- there's my original graph, and now I have this vertex s. But it has directed edges into all of the vertices with 0-weight. That's our picture. And this new thing I'm going to call, maybe, my s graph now.

And the claim is, well now, if I run some shortest path algorithm, single source shortest path algorithm this time, from s to compute the shortest path distance to all of the vertices, the shortest distance to each of the vertices can't be positive, because there's a 0-weight edge. So a minimum weight path is going to be no bigger than 0. If it's finite, then there's a finite length shortest path. If it's minus infinity, then there's a negative rate cycle in my graph and I can stop.

So there are either two situations. If delta s,v equals minus infinity-- so I guess this is, run single source shortest paths from s. And really, because this graph could contain negative edge weights and could contain negative cycles, we can't really do better than running Bellman-Ford here from s to compute these paths. If there exists in this new graph this Gs, if there exists a vertex that has negative infinite weight in the reweighted graph-- sorry, in the original graph G-- G hasn't been reweighted yet. If there's a negative weight distance from s, then there was a negative weight cycle in the original graph.

Why is that? Well, if this was set to minus infinity, then there is some negative weight cycle in the graph. The worry is that that negative weight cycle was added to my graph by adding this vertex s. But what do I know about vertex s? It has no incoming edges. So no negative weight cycle could go through s. So any negative weight cycle was in the original graph, and so I can abort. Abort. Yay.

Otherwise, what do I do? Well, I know the shortest path distances here would satisfy the triangle inequality. So if I reweight with h of v equal to delta s of v, if we set our potentials in our reweighted graph to be the shortest path distance from our super node s, it satisfies the triangle inequality. And because there's no negative cycles, all of these values are finite. And then this reweighting will lead to a graph with strictly-- or not strictly-- strictly no negative weights or non-negative weights. OK, great.

So that's basically it. That's the idea behind Johnson's algorithm. It's really a reduction problem or a reduction algorithm. We reducing from solving kind of signed all-pairs shortest paths, graphs where their weights could be positive or negative, and we're reducing to creating a graph that has the same shortest paths properties, but only has non-negative edge weights. So we're reducing from a signed context to a non-negative weight context.

So Johnson's algorithm, what are the steps? Construct Gs from G, just as up here. I make a new vertex s. I put a 0 weight directed edge from s to every vertex. So that's the first step.

Second step-- compute E, s,v for all V in V, i.e-- or e.g-- I guess really it should be i.e. because I don't really have another option here-- but by Bellman-Ford. This is just a single run of Bellman-Ford here. Compute. And then there are two possibilities.

If there exists a delta s, v that's minus infinity, then abort. Else, make-- or reweight the graph according to this reweighting scheme, by reweighting each edge in my original graph to have weight-- our new weight, which is our old weight, plus our transformation. Now, our transformation is now going to set h, v to this delta s, v. So I'm going to add delta s, u, and subtract delta s, v. That's our reweighting scheme. I'm just identifying h, v with this shortest path distance here.

And after I reweighted that, I can just solve all-pairs shortest paths on G prime with Dijkstra. And then compute G shortest path distances from G prime shortest path distances. Compute these distances from the other using this algorithm up here-- can compute distances in G from distances in G prime in linear time-- or sorry, v times linear time, linear time for each s-- for each vertex in my graph.

OK, so that's the algorithm. It's basically, correctness is trivial. We already proved-- the whole part of this lecture, the interesting part of this lecture was proving that, if we had a transformation based on a potential function that changed outgoing edges in a symmetrically opposite way as incoming edges, then that preserves shortest paths. And then realizing that the triangle inequality enforces this condition that edge weights will be non-negative under this reweighting, so we find shortest path distances from some other arbitrary vertex, and set our potential functions to be those shortest path distance weights.

We do the reweighting, because that reweighting preserves shortest paths, which we already argued. Then we can do-- then this has positive edge weights, so Dijkstra applies. And then computing this takes a small amount of time.

OK, what is the running time of this algorithm? So this part, reconstructing this thing, this takes linear time. I'm just adding. I'm just making a new graph of the same size, except I added v edges and one vertex. Computing-- doing Bellman-Ford on this new modified graph, that's just-- I'm doing that once. That takes V times E time.

Doing this check, that just takes-- I'm looping over my vertices. That just takes V time. Otherwise, doing this reweighting, I change the weight of every edge. That takes order E time. And then solving G prime-- solving all-pairs shortest paths on the modified edge weight graph with Dijkstra takes V times Dijkstra.

That's-- I could use a little bit more board space here. That's V times V log, V plus E time, which is actually the running time that we're looking for. I wanted to reduce to not using more than this time. We used this amount of time. Let's make sure we still didn't use even more.

After that, we compute these paths, as proofed before, in V times V plus E, which is smaller than that. And so summing up all of these running times, this one dominates. And so Johnson's can solve signed weighted all-pairs shortest paths, signed all-pairs shortest paths, not in V times Bellman-Ford, like we had before up here, but faster, in nearly linear for sparse graphs, just without this log factor. So we got quite a big improvement.

So that's the nice thing about all-pairs shortest paths is that, really, we don't have to incur this big cost in the context of negative weights. Essentially, we just run Bellman-Ford once to see if there is a negative weight cycle in my graph. If it is, I save a lot of work by stopping early.

So that's Johnson's. That's the end of our graphs lectures. We'll be having a review and problem session about how to solve problems, graph problems using this material. But we've talked about a lot of different things so far. We've talked about graph reachability, connected components, detecting cycles, detecting topological sort orders of a DAG.

We've talked about finding negative weight cycles, single source shortest path algorithms, and now finally, today, all-pairs shortest path algorithms, with a new algorithm that's really not an entirely new algorithm. We didn't have to do any proof by induction here. Really, the heavy work that's happening is we're reducing to using either Dijkstra or Bellman-Ford to do the heavy lifting of finding single source shortest paths efficiently. So Johnson's is really just glue to transform a graph in a clever way, and then reducing to using some of the shortest paths algorithms faster.

So that's our unit on graphs. Our next lecture, we'll start talking about a general form of, not presenting you with an algorithm, but how to design your own algorithm in the context of dynamic programming. So see you next lecture.