[SQUEAKING]

[RUSTLING]

[CLICKING]

JASON KU: Good morning, everyone. Welcome to the 13th lecture of 6.006. Just to recap from last time, we've been talking about shortest-- single source shortest paths on weighted graphs for the past two lectures. Previously we were only talking about unweighted graphs. And so far, up until today, we've talked about three ways to solve single source shortest paths on weighted graphs.

Namely the first one used BFS. If you can kind of transform your graph into a linear-sized graph that's unweighted that corresponds to your weighted problem, essentially replacing each weighted edge with of weight w with w single edges. Now that's only good for positive weight things and if the sum of your weights are small.

But if the sum of your weights is linear in the combinatorial size of your graph, V plus E, then we can get a linear time algorithm to solve weighted shortest paths using breadth-first search. Then we talked about how we could-- if we-- the problem with weighted shortest paths is if our weights were negative and there could exist cycles, then we could have negative weight cycles and that would be more difficult to handle, because then you have vertices where you have an unbounded number of edges you might have to go through for a shortest path. There might not be a finite length shortest path.

But in the condition where we didn't have cycles in the graph-- of course, we couldn't have negative weight ones, so we were also able to do that in linear time by exploiting the fact that our vertices could be ordered in a topological order, and then we could kind of push shortest path information from the furthest one back to the ones forward. By relaxing edges forward. By maintaining this invariant that we had shortest paths as we were processing these things in topological order.

Then last time, we were talking about general graphs, graphs that could contain cycles, and this is our most general algorithm, because if there are negative weight cycles, Bellman-Ford, which we talked about last time, can detect them. And in particular, for any vertex that had a finite weight shortest paths-- path, we could compute that shortest path for it, compute its distance.

And for any one that is reachable from a negative weight cycle, not only could we mark it as minus infinity distance, but we could also find a negative weight cycle essentially by duplicating our graph to make it a DAG and being able to follow pointers back in this expanded DAG that had multiple layers.

So that's what we've done up until now. We've gotten linear for some types of graphs. And we've gotten kind of quadratic V times E for general graphs, ones that could contain negative cycles. Now how bad is this? Well, if the graph is sparse, if the number of edges in our graph is on the order of V, then this is quadratic time and V, V squared.

But if the graph is dense where we have quadratic-- like the complete graph where every edge is present, then we have quadratically many edges in our graph in V. And so this running time is V cubed. V cube's not great in terms of its running time. We would like something closer to linear.

And so that's what we're going to do today. If we have this restriction where we have non-negative weights, we can have negative weight cycles. And this is a restriction that comes up a lot for many graphs you might encounter. A lot of times you don't have both positive and negative weight. I don't have a negative distance to my house. In any metric we have non-negative weights.

So these things come up a lot, and we can actually do quite a bit better, since there are no negative weight cycles, we can get almost linear. It's not going to be quite V plus E as you see up here on the slide. We're going tc get something very close. It's V plus the E, but on the V term, we have this logarithmic factor in V.

Which remember for all intents and purposes, this log of that thing in real life is not going to be bigger than like a factor of 30 or something like that. Maybe 60. But it's a small number. And so this is actually pretty good performance. It's almost linear-- that's what I'm saying almost linear here, and that's what we're going to try to do today.

So, how do we do this? Well, I'm going to make two observations here, first off. Our idea is going to be to generalize the notion of BFS. When we had BFS, we split up our graph-- to solve unweighted-- solve weighted shortest paths in BFS, we could take our positive edge weights, break them up into individual edges. But if the total weight of our edges was large, then we'd have a problem, because now we've expanded the size of our graph.

This is the same issue that we had with something like radix sort where we don't want our algorithm to run in the size of the numbers in our input, we want our algorithm to run in the number of numbers in our input. This is the difference between N and U back when we were talking about data structures.

Here, if the size of our weights are large compared to V and E, then doing this expansion is going to be difficult. But if we had, say, some graph-- this is my graph G, and we had a source vertex s, the idea here is going to still be to try to grow a frontier of increasing distance from my source and try to maintain all of the things within a certain distance from my source.

So that's the idea, grow a sphere centered at my source, repeatedly explore closer vertices before I get to further ones. But how can I explore closer vertices if I don't know the distances beforehand? This is kind of-- seems like a circular logic. I'm going to use the distance to my things to compute the distances to my things. That doesn't work so well.

So how do we do this? Well, the idea here is to gradually compute the distances-- compute the distances as we gc so that we maintain this property. Now this property, this idea wouldn't work necessarily in the context of negative edge weights. Here, we have this growing frontier, this ball around my source. And as I grow my thing, these things are at further and further distance, because any edge from something back here as I'm growing my ball a certain distance, these things are outside that distance.

We're kind of using a key observation here. Here's my observation 1. If weights greater than or equal to 0, then distances increase along shortest paths. Maybe weakly monotonically increase if there are zero-weight edges.

But in general, if I had a path going from s to some v, and it's going through some vertex u, I have some shortest path. This is the shortest path from s to v, and it goes through some point u, some vertex u. Then this monotonicity more specifically means that the shortest path from s to u and the shortest path from s to v, which is this whole thing, how do these relate to each other?

If this is along that path, then this has to be at least as large as the subpath. Because all of these-- the weight of this path cannot be negative. So that's the thing that Dijkstra's going to exploit. It essentially means that when I'm expanding this frontier of distance away from x, it's possible if I had negative weight, that this line-- if I had some very negative weight going from a vertex here to a vertex here, this vertex could be within this boundary. Maybe if this distance is x, this guy could be within x.

The things that are within distance x of s might not be all contained. There could be a path from here to this other vertex width distance x. It doesn't have this property because I could decrease in distance along the path. So that's the first observation. Second observation, well, let's see if we can piggyback on DAG relaxation.

I claim to you that we can solve single source shortest paths faster if we're given an order of vertices in increasing distance beforehand. Distance from s. So here's the idea. I'm not going to give you the distances to all these vertices. Instead I'm going to give you the order of the vertices in some increasing distance from s.

So basically I'm saying, if I had some, I don't know, here's a graph. Let's see if I can remember. OK, and I'm going to put some edges on here. OK. And I'm going to call these vertices 0, 1, 2, 3, and 4. OK. So here's a graph. Maybe I put some edge weights on here. I'm going to say this one is 3, this one is 2, this one is 3, this is 1, this is 1, this is 0, and this is 0. So from vertex 1 to 2, that was the 2 for the labeling of that vertex. That edge is zero-weight.

OK. So here's a weighted graph And I don't necessarily know-- I could use Bellman-Ford to find shortest paths from this vertex 0, but the idea here is I'm not going to give you shortest paths, I'm going to try to compute shortest paths, but I'm going to give you some additional information. I'm going to give you the order of their shortest path distance from the source.

And I can just-- I'm going to eyeball this and say-- I'm going to change this slightly to make it a little bit more interesting. I'm going to say this is distance 4. OK. All right, so now what we have is the shortest path distance-- I'm just eyeballing this. The shortest distance to-- bad example. All right.

So, these are the weights. Shortest-path distance to 3 is going to be 2, I'm going to say, through there. Shortest-path distance here is 2 also. Shortest-path distance here is also 2 because I can go through both of these 0's and it's not a problem. And then the shortest-path distance here is 2 to here and a 1/3 to there.

So these are listed in increasing distance from my source. I had to compute those deltas to convince you that this was the right ordering, but this is a right ordering of these things. Now it's not the only right ordering, but it is a right ordering. OK, so I'm told-- I'm arguing to you that I could solve a single source shortest paths in linear time if I were to give you the vertices in increasing distance?

How could I do that? Well, because of this first observation, I know that if these are increasing in distance, any edge going backwards with respect to this ordering can't participate in shortest paths with one exception. Anyone know what that exception is? No edge can go backwards in this ordering based on this observation except under what condition? Yeah?

AUDIENCE: If the weight is 0?

JASON KU: If the weight to 0, yeah. So if the weight to 0, just like this situation here, then I could go backwards in the ordering. See, it's problematic. The idea is I'm going to want to construct a DAG so that I can run DAG relaxation. Well, if I have a component here that has 0 weights, I can coalesce this thing down-- I can deal with this component separately. Let's worry about that separately.

If we do, we can collapse this edge down into a single vertex and transform this graph so it does respect the ordering. So I'm going to transform this graph into a new graph. This is a graph-- contains vertex 2 and vertex 0, vertex 1 and 3 here, and vertex 4. OK, now we have-- and I'm only going to keep edges going forward in the-- I'm going to need to collapse this entire section down into one vertex. This doesn't quite work. OK.

Let's ignore zero-weight edges for now. Let's assume these are-- all right, there's something broken here. If I have a cycle here-- right now I don't have a cycle of zero-weight. So what I could do is I could take this vertex and put it after both of these vertices. And now I would-- or I could rearrange the order of these three vertices where there's a path of length 0 and get a new ordering that still satisfies the property.

And that's always the case because paths can't increase-- paths can't decrease in weight. I can rearrange the ordering of these things so that 3 comes first, 1 comes second, and 2 comes third of those three vertices. Yeah. So for every set of 0 edges, I can just flip the relationship if they have the same distance.

In my input, I'm given vertices that have the same distance from the source. And so if those are the same distance from the source and they're connected by a zero-weight edge, it doesn't hurt me to flip their ordering. So I'm going to do that. So let's convert that into a graph with a different ordering. 0 3 now, 1 2. OK and I have this distance, this edge, this edge, this edge, this edge. This edge. What am I missing? 2 to 3. And here. I think I have all of those edges. Yeah? OK.

Now I have the property that every edge that could participate in the shortest path are going forward in the ordering, because all of these are zero-weight. So we flip those around so they're going correct with respect to the ordering. And any edge going backwards that is positive weight certainly can't be used in any shortest path. So I'm just going to get rid of them. Yeah?

What do I do if there's a zero-weight cycle?

JASON KU: If there's a zero-weight cycle, I can just coalesce them all together down to a single vertex, because if I reach one of them, I can reach all of them.

AUDIENCE: You're getting a topological ordering of--

JASON KU: Exactly. I'm computing-- so the idea here is we're trying to construct a DAG. I can construct this DAG in linear time. And then I can run DAG relaxation on this graph in linear time to get shortest paths. So that's an approach. If I knew the ordering of the vertices in increasing distance, then I could use DAG relaxation.

So we're going to use both of these observations. That's how we're going to solve this single source shortage problem with non-negative weights using Dijkstra. So that's finally now where we're coming to. Sorry, I missed a case here when I was writing up my notes, and I tried to fix it live and hopefully you guys followed me. OK.

Dijkstra's algorithm. Did I spell that right? Kind of. OK. What? Dijkstra. OK. Now Dijkstra was this Dutch computer scientist. This is him. Pretty famous, he wrote a monograph on why programming languages should start with 0 indexing as opposed to 1 indexing, so I like him. But in particular, he designed this very nice generalization of BFS for weighted graphs.

But maybe I didn't spell this right because when he writes his name, he writes it with a Y with a dash over it. So in reality on a Dutch typewriter, you might have a character that looks like this, Y with a umlaut on top of it. But on modern-- on an English keyboard, this looks pretty similar to an IJ. So in a lot of manuscripts, we write it as D-I-- there's no J sound in Dijkstra. It's coming from this is Y here. That's an interesting way to remember how to spell Dijkstra.

But the basic idea behind Dijkstra is the following idea. Relaxed edges from vertices in increasing distance from source. OK. This is the same kind of difficulty we had before when we were trying to generalize BFS. So how do we know what the next vertex is with increasing distance to s?

Well, the second idea is find the next vertex efficiently using a data structure. And the data structure we're going to use is something I like to call a changeable priority queue. So this is a little different than a normal priority queue that we had at the end of our data structures unit. This changeable priority queue has three operations. We're going to say it's a queue. We can build it on an iterable set of items. Just stick x-- like n items in there.

We can delete min from the queue. OK, this is the same now as the priority queue. It's this third operation that's going to be different. Decrease the key of an item that has id, id. OK, so this is a little strange. What the heck is this id? All right, with a change of priority queue, each of our items has two values instead of one value. It has a key, but it also-- on which the priority queue is leading the min item with the minimum key. But also, each item has an ID associated with it, a unique integer.

So that when we perform this operation, decrease_key, it can find some item in our data structure with the given ID. And if it's contained there, it's going to change its key to some smaller value k. And don't worry about the edge cases here. We're always going to make sure this k is going to be smaller then whatever that key was to begin with.

So this is really a kind of a funky operation. If I had a priority queue, not a changeable priority queue, but I had a priority queue and I wanted to implement a change of priority queue, how could I do it? Well, a regular priority queue is already going to get me these two operations. It's just this one. I essentially need to find something by an ID and then update its key.

So the idea how to implement this is going to be to use a regular priority queue. I'm going to call it Q prime. And I'm going to cross-link it with a dictionary D. So these are just regular priority queue on my items that has the key as defined above. But I'm going to cross-link it with a dictionary, a dictionary that maps IDs to their location in the priority queue. We've done this many times in the data structures section. We're trying to cross link to data structures to make a query on a different type of key to find its place in another data structure.

So, if we had a priority a dictionary, we could do this stuff pretty fast. In particular, I'm going to assume that our IDs of our vertices are the integers between 0 and v minus 1. And so for my dictionary, I could get constant time looking up of that ID by using what data structure?

AUDIENCE: Hash table.

JASON KU: We could get-- OK, so we could get expected constant time if we used a hash table. But if we knew that our verte IDs were just the numbers from 0 to v minus 1, we could get rid of that expected time by using a direct access array. Great. OK, so that's the assumption.

And so really, the name of the game here is to choose a priority queue here that's going to make these things fast when we start to look at Dijkstra. OK, so we're going to use this data structure to keep track of our distance estimates to all of the vertices away from s. OK, so this is Dijkstra's algorithm. OK.

Set-- so same initialization step. We're going to set-- this is a distance estimate d, not delta. We're going to want the d's be our delta is at the end of the algorithm. That's what we're going to have to prove. So we first set all of them to infinity, and then set d of s, s equal to 0.

And here, we're never going to update it again, because our shortest distance is in a graph with non-negative edge weights certainly can't go below 0. All right. Now we build our-- build our changeable priority queue-- queue-- with an item-- I'm going to say an item is-- x is represented by a tuple of its ID, and then its key just for brevity here. With an item v, d of s, v.

So I'm going to be storing in my changeable priority queue the vertex label and its shortest-path distance estimate d. And that's going to be the key, the minimum that I'm trying going to be querying on for each the v and V. So I'm going to build that thing. It's going to then have all of my vertices in my graph.

Then while my changeable priority queue still has items, not empty, I'm going to delete some u, d s, u. So some item such that its distance is minimized from Q that has minimum distance. OK. So I'm going to I'm going to look at all the things in my priority queue. At the start it's just going to be s, because everything as shortest-path distance estimate infinite except for s. And so that's clearly the smallest.

OK, so I'm going to remove that from my queue, and then I'm going to process it. How am I going to process it? It's the exact same kind of thing as DAG relaxation. I'm going to relax all its outgoing edges. So just for completeness for v in the outgoing adjacencies of u, I'm going to relax-- sorry. We have to check whether we can relax it.

Basically if the shortest-path distance estimate to v is greater than going to u first and then crossing that edge, if going through that is better, this is violating our triangle inequality. And so we relax edge u, v, and by that we mean set this thing to be equal to that thing. That's what we meant by relax.

And then we have one other thing to do. We have changed these distance estimates but our Q doesn't know that we change these things. We added these items in here. But it doesn't know that my distances have changed. So we to tell the Q to remember to change its key value associated with the item v.

So decrease-- what is it? Decrease key vertex v in Q to the new d s, v, the one that I just decreased here. And I know that I decreased it because I said it to a smaller value. That makes sense. All right, so that's Dijkstra. Let's run it on an example.

So here's an example. I have a directed graph. It does contain cycles. In particular, here are some cycles. I think those are the main ones. There are definitely cycles in this graph. But as you see, all of the weights are non-negative, in particular-- they're positive, actually. It's going to be just helpful in writing out this example.

So let's run Dijkstra on this graph. First we initialize and we set the shortest-path distance. I'm going to label it in white here to all of the things. Then I'm going to, as I update it, I'm just going to cross them out and write a new number. So that's what it is at the start. That's initialization, that's after step 1.

And then I stick things into my Q. What's in my Q? Here's my Q. It's everything. It's vertices s, a, b, c, d. I got five items in my Q. Really, it's the item pair with its shortest distance estimate, I'm just not going to rewrite that here. So the idea here is-- the while loop, OK. Q is not empty, great. We're going to delete the one with the smallest distance estimate, which is s, right, yeah.

So I remove that, and then I relax edges out of s. So I relax edge here to a. That's better than the distance estimate-- 10 is better than the distance estimate infinite, so I'm going to change this to 10. And then here's another outgoing edge. 3 is better than infinite, so I'm going to change its delta to 3. OK. So now I go back in here and I change the distance estimates associated with my Q.

Now, next step of the algorithm, s is done. I've processed everything distance 0 away. But I'm now going to use my priority queue to say which of my vertices has the shortest distance estimate now. So which one is it? a, b, or c, or d? Yeah, it's 3 and c. 3 is smaller than 10.

So Q is going to magically delete c for me, tell me what that is, and now I'm going to process that. Now I've changed my boundary to this. And now I relax edges out of c. So here's an edge at a c, that's a 4. A 4 plus the 3 is smaller than 10, so I update it. 3 plus 8 is 11, that's smaller than infinite, so I update it, I relax. 3 plus 2 is smaller than infinite, so I relax that as well.

Now of the things still left in my Q, I'm actually going to remove it from my Q instead of crossing it out, maybe that's better. Of the vertices still left in my Q, which has smallest distance? Yeah. d. d has 5, 7, or 11. 5 is the smallest. So I remove d from my cue and I relax edges from it. And now my boundary looks something like this.

I relax edges out of it. 5 plus 5, that's 10. 10 is smaller than 11, so that's a 10. And that's the only outgoing edge from d. so I'm done. And then the last, 7 is smaller than 10, I relax edges out of a. a to b, 7 plus 2 is smaller than 10. And now I'm done. So what I did every time I removed s-- or I removed a vertex, I said its shortest-path distance to the small-- the last value I assigned to it. So this was then 3, and then a was 7, b was 9, and then d was 5.

So that's Dijkstra in action. It seems like these are the shortest-path distances, but how do we prove that? Did it do the right thing? Well, let's find out. So that's what we're going to spend some time on right now, just talking about the correctness of Dijkstra's algorithm.

OK. Correctness follows from two main observations. So the claim here that we're trying to prove is that d of s equals the delta s-- so the estimates equal the shortest-path distance is at the end of Dijkstra for all v and V at end. And this is going to follow from two observations.

So the proof here, first, if ever relaxation sets d of s of v-- it sets the estimate equal to the shortest-path distance, if it ever does that, I argue to you that still true at end. OK, that's not a very strong statement. This is saying if I ever set the distance estimate to the true distance, I'm never going to set it to a different value later on. And why is that?

Well, relaxation only ever decreases the distance. Relaxation only decreases d s, v. But we proved in lecture 11-- so two lectures ago that relaxation is safe. And what does safe mean? Safe means that relaxation-- that relaxation will only ever change these distant estimates to be either infinite-- it was never-- there was never a path to my vertex. Or it was the length of some path to v. Length of some path.

OK. So what does that mean? It only decreases, but it's always the length of some path to v. So if this is the length of the shortest path to v, I could never set it to a smaller length, because there are no paths with shorter distance. That's the whole point. OK. So with this observation, I'm going to argue this final claim. It suffices to show that my estimate equals the shortest distance when v is removed from the Q.

And since I removed every vertex from the Q in this while loop, I will eventually said to all of the distance estimates to the real distance and we'll be golden. Happy days. All right. So we'll be done if we can prove that statement. All right. So we're going to prove this by induction obviously. Induction on first k vertices removed from the Q.

So the Q, we're popping vertices from this Q in some order. So I'm going to just argue that this claim is true for the first k. Clearly that's true for k equals 1. Base case, k equals 1. What is k equals 1? That means the first word vertex that I pop has this property, which is definitely true, because we set the shortest distance to s to be 0. That's all good.

Now we have our inductive step. Assume it's true for k prime-- sorry, k less than k prime. And let's let v prime be k prime vertex popped. v prime. OK. And now let's look at some shortest path from s to v prime.

So we got the shortest path from s to v prime. It exists. v prime is accessible. Let's say we pruned our graph to be only the things accessible from s so that, yeah, there exists the shortest path to v prime. And now let's think about these vertices. Some of them were removed from the Q and some of them were not. s was definitely removed from the Q. But some of these other vertices might not be.

I want to be able to induct on this path, in particular, the vertex before me so that I can say that when I removed it and I relax the edge to v prime, then we're all golden. But that might not be the case. There could be a vertex, the vertex preceding me in the graph in this shortest path that was not popped from Q. I need to argue that it was or some other thing.

So let's consider the first vertex in this path from s to v. I'm going to call it y, I think. Yeah. A vertex y that is not in Q. After I pop v prime, this is the first-- or before I pop v prime, y is not in the Q. Now these might be the same vertex if all of the preceding ones on this path were in the Q. But in particular, we're going to look at this guy. And say its predecessor's x in the path.

Well what do I know? I know that x is in the queue. Everything here was popped from the Q-- not in. Which means that by induction, the shortest-path distance was set here correctly. So that the distance estimate at y can't be bigger than the shortest path to x plus w x, y.

But this is on the shortest path to y, because the subpaths of shortest paths or shortest paths. So this has to equal d s, y, the distance to y. So actually, y is all good here. And so if v prime were y, we'd be done. That's the same argument is DAG relaxation. But we need to prove something about v prime.

Well, because we have non-negative weights, the distance to v prime has to be at least as big as this distance, because it's a subpath. So this has to be less than or equal to the true distance to v prime. Because of negative-- non-negative weights, because the weights are non-negative.

But because relaxation is safe, we know that our distance estimate for v prime has to be at least the shortest-path distance. This is because it's safe. This is-- weights are greater than or equal to 0. The last step here is that because we're popping the minimum from our priority queue, the thing with the smallest shortest-path distance, this has to be less than or equal to the shortest-path distance estimate to y. Because this is the smallest among all such vertices in my Q.

But these are the same value. So everything between here is the same value. In particular, the estimate here is equal to my true shortest-path distance, which is exactly what we're trying to prove. OK, so that's why Dijkstra's correct. I'm going to spend the last five minutes on the running time of Dijkstra.

We set this up so that we did everything in terms of these Q operations. Right so we have these Q operations, we have three of them. I'm going to say if I have a build operation, let's say it takes B time; to lead min, I'm going to say it takes M time; and this decreased key, I'm going to say it takes D time.

So what is the running time of Dijkstra? If I take a look at that algorithm over there-- well I guess let's switch these back up again. OK, so what does this do? We build once. Then we delete the minimum from the Q how many times? v times. We remove every vertex from our Q.

Then for every possible edge, we may need to relax and decrease the key in our queue once for every outgoing edge. So the running time is B plus V times M plus E times D. OK. So how could we implement this priority queue? Well, if we use the stupidest priority queue in the world, here's a list of different implementations we could have for our priority queues. And when I say priority queue, I mean this priority queue. We're already implementing the changeable priority queue by linking it with a dictionary that's efficient

If I just use an array, I can find the min in linear time, sure. And I don't have to update that array in any way. I mean, I can just keep the distances in my direct access array. I don't have to store a separate data structure. I just store the distances in my direct access array D, and so I can find it in constant time and I can update the values stored there. And then whenever I want the minimum, I can just loop through the whole thing.

So that gives me a really fast decrease key, but slow delete min. But if we take a look at the running time bound here, we get something, if we replace n with v, we get a quadratic time algorithm in the number of vertices, whicl for a dense graph, this is in linear time. That's actually pretty good. Dense meaning that I have at least a quadratic number of vertices. So that's actually really good, and it's the stupidest possible data structure we could use for this priority queue.

Now we can do a little better, actually, for not dense-- I mean, for sparse graphs where the number of edges is at most v, then this is pretty bad, it's quadratic. We want to do something a little better. Now if we're sparse, a binary heap can delete min in logarithmic time, but it can actually, if I know where I am in the heap and I decrease the key and I'm in a min heap, I can just swap with my parent upwards in the tree in log n time and rebalance the-- refix the binary heap property. And so I can do that in logarithmic time.

And if I do that and I put it into this formula, I actually get n-- or V plus V times log V plus E times log V. And so that's going to give me E log V if I'm assuming that I'm first pruning out all of the things not connected to me, then E asymptotically upper bounds V, and I get this E log V running time, which is pretty good. That's just an extra log factor on linear.

Now there's an even better-- well, better is hard to say. Really, there's a different data structure that achieves both bounds for sparse and dense graphs and everything in between. It gives us an E plus V log V running time bound. This data structure is called the Fibonacci heap. We're not going to talk about it in 6.006. They talk about it-- and you can look at chapter 19 in CLRS or you can look at-- I think they talk about it in 6.854 if you're interested in learning about Fibonacci heaps.

But these are almost never-- I mean, they get good theoretical bounds. So what you want to say is, whenever we give you a theory problem where you might want to use Dijkstra, you want to use this theoretical running time bound for your problem E plus V log V. But if you happen to know that your graph is sparse or dense, just using an array or a heap is going to get you just as good of a running time. Very close to linear.

And so in practice, most people, when they are implementing a graph search algorithm, they know if their graph is sparse or dense, and so they never bother implementing a Fibonacci heap, which is a little complicated. So they're usually either in one of these first two cases where V squared is linear when your graph is dense, or we're very close to linear, E times log V, which is V log V if your graph is sparse. So that's the running time of Dijkstra.

So so far, we've gotten all of these nice bounds. Some special cases where we're-- I mean, special cases where we're linear. Dijkstra where we're close to linear. And Bellman-Ford, if we throw our hands up in the air, there might be negative cycles in our graph, we gotta spend that quadratic running time bound. Now there are faster algorithms, but this is the fastest we're going to teach you in this class.

Now and in the next lecture we're going to be talking about all pair shortest paths, and we'll pick it up next time.