We're going to talk about shortest paths, and we're going to talk about shortest paths for three lectures. So, this is a trilogy.

Today will be Shortest Paths One.

I've been watching far too many versions of Star Wars this weekend. I saw the musical yesterday, matinee. That was an MIT musical.

That was fun, of all three movies in about four hours. That was a bit long and then I saw the one-man show on Friday. One-man Star Wars: the original three movies in one hour.

That was the opposite of too long.

Both were fun. So I get my trilogy fix.

All episodes, first we're going to start with The New Hope, and we're going to talk about the shortest paths problem and solve one particular problem of it, a very interesting version. And then we're going to look at increasingly more general versions as we go on.

Shortest paths are sort of an application of dynamic programming, which we saw last week, and greedy algorithms, which we also saw last week. So, were going to build that and get some pretty interesting algorithms for an important problem, which is how to get from Alderon to, I don't know, Cambridge as quickly as possible, OK, when you live in a graph.

So, there's geometric shortest paths which is a little bit harder. Here, we're just going to look at shortest paths in graphs. Now, hopefully you all know what a path in a graph is. But, so, very quick review in particular because we're going to be looking at weighted graphs. So, the usual setup: suppose we have directed graph, G, have some vertices, some edges. We have edge weights, make it a little more interesting.

So, this is just a real number on each edge.

So, edge weights are usually given by function, w. For every edge, you get a real number.

And then, if we look at the paths in the graph, so we're going to use some simple notation for paths called a path, p, starts at some vertex, and it goes to some other vertex, and so on.

Say the last vertex is v_k, and each of these should be a directed edge in the digraph. So, this is a directed path.

It has to respect edges in here.

And, we'll say that the weight of such a path is just the sum of the weights of the edges along the path.

And, we'll call that w(p). This is sum, i equals one to k minus one of w(v_i, v_(i+1)) plus one.

OK, so just to rub it in, and in particular, how general this can be, we have some path, it starts at some vertex, there's some edge weights along the way. This is some arbitrary path in the graph, in some hypothetical graph.

OK, this is mainly to point out that some of the edge weights could be negative. Some of them could be zero.

This sum here is minus two. So, the weight of this path is minus two. And, presumably, the graph is much bigger than this.

This is just one path in the graph.

We're usually thinking about simple paths that can't repeat a vertex. But, sometimes we allow that.

And then, what we care about is the shortest path, or a shortest path. Again, this may not be unique, but we'll still usually call it the shortest path.

So, we want the shortest path from some A to some B.

Or, we'll call the vertices u and v.

And we want this to be some path of minimum possible weight, subject to starting at u, and going to v.

OK, so that's what we're looking for.

In general, give you a vertex, u, give you a vertex, v, find a shortest path as quickly as possible.

What's a good algorithm for that?

That's the topic for the next three lectures.

We'll usually think about a slightly simpler problem, which is just computing the weight of that path, which is essentially computing the distance from A to B.

So, we'll call this the shortest path weight from u to v. And, we'll denote it by delta of (u,v), delta . So, I mean, it's the weight of the shortest path, or a weight of every shortest path. Or, in other words, it's the Min over the weight of each path from u to v.

So, p here is a path. OK, so you just consider, there could be a lot of different paths.

There could, in principle, be infinitely many, if you're allowed to repeat vertices. You look at all those paths hypothetically. You take the minimum weight.

Question? Good.

My next question was going to be, when do shortest paths not exist? And you've hit upon one version, which is when you have negative edge weights.

So, in principle, when you have negative edge weights, some shortest paths may not exist in the sense that there is no shortest paths. There are no shortest paths.

There is no shortest path from u to v.

OK, in particular, if I have two vertices, u and v, and I want the shortest path between them, and I have negative edge weights, well, this is fine. I mean, I can still compute the weight of a path that has negative weights.

But when specifically won't I have a single shortest path from u to v? So, go ahead.

Good. So, if I can find the cycle somewhere along here whose total weight, say, the sum of all the weights of these images is negative, then I get there, I go around as many times as I want.

I keep decreasing the weight because the weight is negative.

I decrease it by some fixed amount, and then I can go to v.

So, as long as there is a negative weights cycle reachable from u that can also reach v, then there's no shortest path because if I take any particular path, I can make it shorter by going around a couple more times.

So, in some sense, this is not really a minimum.

It's more like an infimum for those who like to get fancy about such things. But we'll just say that delta of (u,v) is minus infinity in this case.

There's a negative weights cycle from u to v.

So, that's one case we have to worry about in some sense.

But, as long as there are no negative weight cycles, delta of (u,v) will be something bigger than minus infinity, bounded below by some finite value even if you could have negative weights, but still no negative weights cycle for example, there might not be any cycles in your graph. So that's still interesting.

And, I guess it's useful to note that you can get from A to B in negative infinite time. It's time travel, if the weights happen that correspond to time.

But when else might shortest paths not exist?

So, this is one case, but there's another, simpler case. It's not connected.

There might not be any path from u to v.

This path might be empty. There may be no path from u to v. Here we have to define what happens, and here, we'll say it's infinity if there's no path from u to v. So, there are these exceptional cases plus infinity and minus infinity, which are pretty intuitive because it takes a really long time to get from u to v if there's no path there. You can't get there from here.

OK, but that's the definition. Most of the time, this is the case we care about, of course.

Usually this is a finite set. OK, good, so that's the definition. We're going to get a few basic structural properties about shortest paths that will allow us to obtain good algorithms finding these paths when they exist. And, in particular, we want to use ideas from dynamic programming.

So, if I want to use dynamic programming to solve shortest paths, what do I need to establish?

What's the first thing I should check?

You've all implemented dynamic programming by now, so should make complete sense hopefully, at least more sense than it did a couple of weeks ago, last week, when we learned it. Dynamic programming is something that grows on you. Every year I think I understand it better than the previous year.

But, in particular, when you learned dynamic programming in this class, there is this nice key property that you should check. Yeah?

Optimal substructure: good.

This is the phrase you should keep in mind.

It's not really enough for dynamic programming to be useful in an efficient way, but it at least tells you that you should be able to try to apply it.

That's a pretty weak statement, but it's something that you should check. It's definitely pretty much a necessary condition for dynamic programming to make sense.

And so, optimal some structure here means that if I take some shortest path, and I look at a subpath of that shortest path, I claimed that it too is a shortest path, OK, with its respective endpoints; obviously not between the same endpoints.

But if I have some shortest path between two endpoints, I take any subpath and that's also the shortest path.

This is one version of optimal substructure.

This one turns out to be true for this setup.

And, how should I prove an optimal substructure property?

Cut and paste. Yep, that works here too.

I mean, this isn't always true. But it's a good technique here.

So, we're going to think about, and I'll do essentially a proof by picture here. So, suppose you have some subpath of some shortest path. So, let's say the subpath is x to y. And, the path goes from u to v.

So, we assume that (u,v) is a shortest path.

We want to prove that (x,y) is a shortest path.

Well, suppose (x,y) isn't a shortest path.

Then there is some shorter path that goes from x to y.

But, if you have some shorter path from x to y than this one.

Then I should just erase this part of the shortest path from u to v, and replace it with this shorter one.

So, this is some hypothetical shorter path.

So, suppose this existed. If that existed, then I should just cut the old path from x to y, and paste in this new one from x to y.

It's strictly shorter. Therefore, I get a strictly shorter path from u to v. But I assumed u to v was a shortest path: contradiction.

OK, so there is no shorter path.

And that proves the lemma that we have this: subpaths of shortest paths are shortest paths.

OK, this should now be a pretty familiar proof technique.

But, there is yet another instance of cut and paste.

OK, so that's a good sign for computing shortest paths.

I mean, in terms of dynamic programming, we won't look directly at dynamic programming here because we are going to aim for greedy, which is even stronger.

But, next Monday we'll see some dynamic programming approaches.

Intuitively, there are some pretty natural sub-problems here. I mean, going from u to v, if I want to find what is the shortest path from u to v, well, that's a particular problem.

Maybe it involves computing shortest paths from u to some intermediate point, x, and then from x to u, something like that. That feels good.

That's like, quadratically, many subproblems. And, V^2 subproblems, it sounds like that would lead to a dynamic program.

You can make it work out; it's just a little bit trickier than that. We'll see that next Monday.

But thinking about this intermediate point we get something called the triangle inequality.

So, you've probably heard some form of the triangle inequality before. It holds in all sorts of geometric spaces, but it also holds for shortest paths, which is slightly less obvious, or more obvious, I guess, depending on your inclination.

So, if you have any triple of vertices, the shortest path from u to v is, at most, the shortest path from u to x plus the shortest path from x to v.

Of course, here I need a shortest path weight from u to x, and shortest path weight from x to v.

So, this should be pretty natural just from the statement, even more natural if you draw the picture.

So, we have some vertex, u.

I'm using wiggly lines to denote potentially long paths as opposed to edges. We have some intermediate point, x, and we have some target, v, and we are considering these three shortest paths.

This is the shortest path from u to v, or this is its weights.

This is the shortest path from u to x.

And here's its weight, and the shortest path from x to v. And here's its weight.

And, the point is, this should be the shortest path or a shortest path from u to v.

And, in particular, one such path is you go from u to x, and then you go from x to v.

So, I mean, this sum is just measuring the length of this particular path. Take the shortest path here; take the shortest path here. And, this is supposed to be the Min over all paths. So, certainly this is, at most, this particular path, the sum of these two values, OK, another proof by picture. Clear?

OK, this stuff is easy. I assume we'll get into some more set exciting algorithms in particular, which is always more exciting. Today, we're going to look at a particular version of shortest paths called, or the shortest paths problem called the single source shortest path problem. OK, it's a little bit more general than go from A to B. The problem is, you're given a source vertex, and you want to know how to get from that source vertex to everywhere else.

So, we'll call this source vertex s.

And from that source, we want to find, let's say, the shortest path weights from s to everyone.

In particular, we'd also like to know the shortest paths, but that isn't too much harder.

So, that's delta of s, v for all vertices, v. OK, so this is actually a little bit harder than the problem we started with a getting from Alderon to Cambridge.

Now, we want to get from Alderon to the entire universe.

OK, it turns out, this is one of the weird things about shortest paths, according to the state-of-the-art we know today, it seems like the following statement will remain true for all time.

But we don't know. The best algorithms for solving the A to B problem, given s, given t, go from s to t, is no easier than this problem.

It's the best ways we know how to solve going from A to B is to solve how to go from A to everywhere else.

So, we sort of can't help ourselves, but to solve this problem it turns out. Today, we're going to look at a further restriction on this problem because this is a bit tricky. Will solve it next class.

But, today we're going to get rid of the negative weight cycle issue by forbidding negative weights.

So, we're going to assume that all of the edge weights are nonnegative, so, for all vertices, u and v. So, in particular, shortest paths exist, provided paths exist.

And, we don't have to worry about these minus infinities.

Delta of (u,v) is always bigger than minus infinity.

It still might be plus infinity if there is no path, but this will make life a lot easier.

And the algorithm we'll cover today really requires this property. You can't get away without it.

Next class, we'll get away without it with a fancier and slower algorithm. So, as I hinted at, the main idea we're going to use for the algorithm today is greedy, which should be faster than dynamic programming generally. And, the tricky part will be proving that the greedy algorithm actually works.

So, I think there's pretty much only one natural way to go about, well, there's one way that works to go about greedy, let's say. This may be not the obvious one. So, let me give you a little bit of setup. The invariant we are going to maintain is that at all times, we have estimates on the distances from the source to every vertex.

When I say distance, I mean shortest path weight.

I'm going to use weight and distance interchangeably here for more intuition. And, in particular, I want to maintain the set of vertices where those estimates are actually the right answer.

OK, this is little s. This is big S.

So, the big S will be the set of all vertices where I know the answer. What is the shortest path distance from little S to that vertex in big S?

So, for starters, which distance do I know?

Sorry? s.

I know the shortest path distance from s to s because if I assume that all of my weights are nonnegative, I really can't get from s to s any faster than not doing anything. OK, if I had a negative weight cycle, maybe the distance from s to s is minus infinity.

OK, but I can't have negative weights so there's no way I can get from s to s any faster than zero time.

There might be a longer path that still has zero cost, but it can't be any better than zero.

So, in particular, I know that.

So, initially, S is certainly an s.

OK, and the idea is we're going to accumulate more and more vertices that we know. So, at some point we know the distances from some of the vertices.

So, we have some cloud here. This is S, and this is everything else. This is the graph, G. This is the subset of the vertices. And, there's some edges that go out from there. And, so we have estimates on how to get to these vertices. Some of them, we may not have even seen yet. They may not be connected to this portion of S. I mean: not directly.

They might be connected by some longer path.

They might be in a completely different connected component.

We don't know yet. Some of them, we have estimates for because we've sort of seen how to get there from S. And the idea is, among all of these nodes where we have estimates, and on to get from little S, which is some vertex in here, to these vertices, we're going to take the one for which the estimate is smallest. That's the greedy choice.

And, we're just going to add that vertex to S.

So, S grows one vertex per step.

Each step, we're going to add to S, the vertex.

Of course, again, this is not a unique, it's a vertex, v, in V minus S.

So, it's something we haven't yet computed yet whose estimated distance from S is minimum. So, we look at all the vertices we haven't yet added to S. Just take the one where we have the estimated smallest distance. The intuition is that that should be a good choice. So, if I pick the one that's closest to little s among all the ones that I've seen, among all the paths that I've seen, I sort of have to buy into that those are good paths. But, I mean, maybe there's some path I didn't see.

Maybe you go out to here and then you take some other path to some vertex, which we've already seen.

OK, the worry is, well, I'd better not say that that's the shortest path because there may have been some other

way to get there. Right, as soon as I add something to S, I declare I've solved the problem for that vertex. I can't change my answer later.

OK, the estimates can change until they get added to S.

So, I don't want to add this vertex to S because I haven't considered this path. Well, if all my weights are nonnegative, and I take the vertex here that has the shortest estimate from S, so let's suppose this one is the shortest one, then this can't be a shorter path because the distance estimate, at least, from S to that vertex is larger from S to that vertex.

So, no way can I make the path longer and decrease the distance. That's the intuition.

OK, it's a little bit fuzzy here because I don't have any induction hypotheses set up, and it's going to be a lot more work to prove that. But that's the intuition why this is the right thing to do. OK, you have to prove something about the distance estimates for that to be a proof.

But, intuitively, it feels good.

It was a good starting point. OK, and then presumably we have to maintain these distance estimates.

So, the heart of the algorithm is updating distance estimates, I mean, choosing the best vertex to add to S, that's one step. Then, updating the distance estimates is sort of where the work is.

And, it turns out we'll only need to update distance estimates of some of the vertices, the ones that are adjacent to v. v was the vertex we just added to S. So, once we add somebody to S, so we grow S by a little bit, then we look at all the new edges that go out of S from that vertex.

We update something. That's the idea.

So, that's the idea for how we're going to use greedy.

Now I'll give you the algorithm.

So, this is called Dijkstra's algorithm.

Dijkstra is a famous, recently late, if that makes sense, computer scientist from the Netherlands. And, this is probably the algorithm he is most famous for. So, the beginning of the algorithm is just some initialization, not too exciting. OK, but let me tell you what some of the variables mean. OK, so d is some array indexed by vertices, and the idea is that d of x is the distance estimate for x, so, from S to x.

so in particular, it's going to equal the real shortest path weight from S to x when we've added x to our set capital, S. OK, so this is, in particular, going to be the output to the algorithm. Did you have a question?

Or were you just stretching? Good.

So, in d of x, when we are done, d of x is the output. For every vertex, it's going to give us the shortest path weight from S to that vertex. Along the way, it's going to be some estimated distance from S to that vertex.

And, we're going to improve it over time.

This is an infinity. So initially, we know that the distance, we know the distance from S to S is zero. So, we're going to set that to be our estimate. It's going to be accurate.

Everything else we're going to just set to infinity because we may not be connected. From the beginning, we don't know much. S, initially, is going to be infinity. Immediately, we're going to add little s to big S.

And then, the interesting part here is Q, which is going to consist of, initially all the vertices in the graph.

And, it's going to not just be a queue as the letter suggests.

It's going to be a priority queue.

So, it's going to maintain, in particular, the vertex that has the smallest distance estimate.

So, this is a priority queue. This is really an abuse of notation for a data structure. OK, so this could be a heap or whatever. The vertices are keyed on d, our distance estimate. So, in particular, S will have the, this is going to be a Min heap.

S will be the guy who has the minimum.

Everyone else has the same key initially.

And, we're going to repeatedly extract the minimum element from this queue and do other things. OK, so this is initialization.

OK, I'm going to call that initialization.

It's a pretty simple thing. It just takes linear time, nothing fancy going on. The heart of the algorithm is all in six lines. And, so this is not really a step. The first step here that we need to do is we take the vertex whose distance estimate is minimum. So that, among all the vertices, not yet, and that's currently S is empty. Q has everyone.

In general, Q will have everyone except S.

So, we'll take the vertex, u, that has the minimum key in that priority queue. So, extract the Min from Q.

OK. We're going to add a little u to S, claim that that is now, I mean, that's exactly what we're saying here. We add to S that vertex that has minimum distance estimate. And now, we need to update the distances. So, we're going to look at each adjacent vertex for each v in the adjacency list for u.

We look at a few distances.

So that's the algorithm or more or less.

This is the key. I should define it a little bit what's going on here. We talked mainly about undirected graph last time. Here, we're thinking about undirected graphs. And, the adjacency list for u here is just going to mean, give me all the vertices for which there is an edge from u to v.

So, this is the outgoing adjacency list, not the incoming adjacency list.

Undirected graphs: you list everything.

Directed graphs: here, we're only going to care about those ones. So, for every edge, (u,v), is what this is saying, we are going to compare the current estimate for v, and this candidate estimate, which intuitively means you go from s to u.

That's d of u because we now know that that's the right answer. This, in fact, equals, we hope, assuming the algorithm is correct, this should be the shortest path weight from s to u because we just added u to S. And whenever we add something to S, it should have the right value.

So, we could say, well, you take the shortest path from S to u, and then you follow this edge from u to v. That has weight, w, of (u,v). That's one possible path from S to v. And, if that's a shorter path than the one we currently have in our estimate, if this is smaller than that, then we should update the estimate to be that sum because that's a better path, so, add it to our database of paths, so to speak: OK, very intuitive operation; clearly should not do anything bad. I mean, these should be paths that makes sense. We'll prove that in a moment.

That's the first part of correctness, that this never screws up. And then, the tricky part is to show that it finds all the paths that we care about.

This step is called a relaxation step.

Relaxation is always a difficult technique to teach to MIT students. It doesn't come very naturally.

But it's very simple operation. It comes from optimization terminology, programming terminology, so to speak.

And, does this inequality look familiar at all especially when you start writing it this way? You say, the shortest path from S to v and the shortest path from S to u in some edge from u to v, does that look like anything we've seen?

In fact, it was on this board but I just erased it.

Triangle inequality, yeah.

So, this is trying to make the triangle inequality true.

Certainly, the shortest path from S to v should be less than or equal to, not greater than. The shortest path from S to u, plus whatever path from u to v, the shortest path should be, at most, that. So, this is sort of a somewhat more general triangle inequality.

And, we want to, certainly it should be true.

So, if it's not true, we fix it.

If it's greater than, we make it equal.

But we don't want to make it less than because that's not always true. OK, but certainly, it should be less than or equal to.

So, this is fixing the triangle inequality.

It's trying to make that constraint more true.

In optimization, that's called relaxing the constraint. OK, so we're sort of relaxing the triangle inequality here. In the end, we should have all the shortest paths. That's a claim.

So: a very simple algorithm. Let's try it out on a graph, and that should make it more intuitive why it's working, and that the rest of the lecture will be proving that it works. Yeah, this is enough room.

So, oh, I should mention one other thing here.

Sorry. Whenever we change d of v, this is changing the key of v in the priority queue.

So, implicitly what's happening here in this assignment, this is getting a bit messy, is a decreased key operation, OK, which we talked briefly about last class in the context of minimum spanning trees where we were also

decreasing the key.

The point is we were changing the key of one element industry like station step in the priority queue so that if it now becomes the minimum, we should extract here.

And, we are only ever decreasing keys because we are always replacing larger values with smaller values.

So, we'll come back to that later when we analyze the running time. But, there is some data structure work going on here. Again, we are abusing notation a bit. OK, so here is a graph with edge weights.

OK, and I want my priority queue over here.

And, I'm also going to draw my estimates.

OK, now I don't want to cheat. So, we're going to run the algorithm on this graph. s will be A, and I want to know the shortest path from A to everyone else.

So, you can check, OK, paths exist.

So, hopefully everything should end up a finite value by the end. All the weights are nonnegative, so this algorithm should work.

The algorithm doesn't even need connectivity, but it does mean that all the weights are nonnegative.

So, we run the algorithm. For the initialization, we set the distance estimate for our source to be zero because, in fact, there's only one path from A to A, and that to do nothing, the empty path.

So, I'm going to put the key of zero over here.

And, for everyone else, we're just going to put infinity because we don't know any better at this point.

So, I'll put keys of infinity for everyone else.

OK, so now you can see what the algorithm does is extract the minimum from the queue. And, given our setup, we'll definitely choose s, or in this case, A. So, it has a weight of zero.

Everyone else has quite a bit larger weight.

OK, so we look at s, or I'll use A here.

So, we look at A. We add A to our set, S. So, it's now removed from the queue. It will never go back in because we

never add anything to the queue, start with all the vertices, and extract, and decrease keys.

But we never insert. So, A is gone.

OK, and now I want to update the keys of all of the other vertices. And the claim is I only need to look at the vertices that have edges from A.

So, there's an edge from A to B, and that has weight ten.

And so, I compare: well, is it a good idea to go from A to A, which costs nothing, and then to go along this edge, AB, which costs ten?

Well, it seems like a pretty good idea because that has a total weight of zero plus ten, which is ten, which is much smaller than infinity.

So, I'm going to erase this infinity; write ten, and over in the queue as well. That's the decreased key operation. So now, I know a path from A to

B. Good. A to C is the only other edge. Zero plus three is less than infinity, so, cool.

I'll put three here for C, and C is there.

OK, the other vertices I don't touch.

I'm going to rewrite them here, but the algorithm doesn't have to copy them. Those keys were already there.

It's just touching these two. OK, that was pretty boring.

Now we look at our queue, and we extract the minimum element. So, A is no longer in there, so the minimum key here is three.

So, the claim is that this is a shortest path; from A to C, here is the shortest path from A to C. There's no other shorter way.

You could check that, and we'll prove it in a moment.

Cool, so we'll remove C from the list.

It's gone. Then we look at all of the outgoing edges from C. So, there's one that goes up to B, which has weight four, four plus three, which is the shortest path weight from A to C.

So, going from A to C, and C to B should cost three plus four, which is seven, which is less than ten.

So, we found an even better path to get to B.

It's better to go like this than it is to go like that.

So, we write seven for B, and there's an outgoing edge from C to d which costs eight. Three plus eight is 11.

11 is less than infinity last time I checked.

So, we write 11 for d. Then we look at E.

We have three plus two is five, which is less than infinity.

So, we write five for the new key for E.

At this point, we have finite shortest paths to everywhere, but they may not be the best ones. So, we have to keep looking.

OK, next round of the algorithm, we extract the minimum key among all these. OK, it's not B, which we've seen though probably know the answer to.

But it's E. E has the smallest key.

So, we now declare this to be a shortest path.

The way we got to E was along this path: A to C, C to E, declare that to be shortest.

We claim we're done with E. But we still have to update.

What about all the outgoing edges from E?

There's only one here. It costs five plus nine, which is 14, which is bigger than 11.

So, no go. That's not an interesting path.

Our previous path, which went like this at a cost of the 11, is better than the one we are considering now.

I'm drawing the whole path, but the algorithm is only adding these two numbers. OK, good.

So, I don't change anything. Seven, 11, and five is removed, or E is removed. Our new keys are seven and 11.

So, we take the key, seven, here, which is for element B, vertex B.

We declare the path we currently have in our hands from A to B, which happens to be this one.

Algorithm can't actually tell this, by the way, but we're drawing it anyway. This path, A, C, B, is the candidate shortest path.

The claim is it is indeed shortest.

Now, we look at all the outgoing edges.

There's one that goes back to C at a cost of seven plus one, which is eight, which is bigger than three, which is good. We already declared C to be done. But the algorithm checks this path and says, oh, that's no better.

And then we look at this other edge from B to d.

That costs seven plus two, which is nine, which is better than 11. So, we, in fact, found an even shorter path. So, the shortest path weight, now, for d, is nine because there is this path that goes A, C, B, d for a total cost of three plus four plus two is nine. Cool, now there's only one element in the queue. We remove it.

d: we look at the outgoing edges.

There's one going here which costs nine plus seven, which is 16, which is way bigger than five.

So, we're done. Don't do anything.

At this point, the queue is empty.

And the claim is that all these numbers that are written here, the final values are the shortest path weights.

This looks an awful lot like a five, but it's an s.

It has a weight of zero. I've also drawn in here all the shortest paths. And, this is not hard to do.

We're not going to talk about it too much in this class, but it's mentioned in a little bit more detail at the end of the textbook. And it's something called the shortest path tree. It's just something good to know about if you actually want to compute shortest paths.

In this class, we mainly worry about the weights because it's pretty much the same problem.

The shortest path tree is the union of all shortest paths.

And in particular, if you look at each vertex in your graph, if you consider the last edge into that vertex that was

relaxed among all vertices, u, you look at the edges, (u,v), say, was that last one to relax?

So, just look at the last edges we relaxed here.

You put them all together: that's called a shortest path tree. And, it has the property that from S to everywhere else, there is a unique path down the tree. And it's the shortest path.

It's the shortest path that we found.

OK, so you actually get shortest paths out of this algorithm even though it's not explicitly described.

All we are mainly talking about are the shortest path weights.

Algorithm clear at this point? Feels like it's doing the right thing? You can check all those numbers are the best paths. And now we're going to prove that.

So: correctness.

So the first thing I want to prove is that relaxation never makes a mistake. If it ever sets d of v to be something, I want to prove that d of v is always an upper bound on delta. So, we have this variant.

It's greater than or equal to delta of s, v for all v.

And, this invariant holds at all times.

So, after initialization, it doesn't hold before initialization because d isn't defined then.

But if you do this initialization where you set S to zero, and everyone else to infinity, and you take any sequence of relaxation steps, then this variant will hold after each relaxation step you apply.

This is actually a very general lemma.

It's also pretty easy to prove. It holds not only for Dijkstra's algorithm, but for a lot of other algorithms we'll see. Pretty much every algorithm we see will involve relaxation. And, this is saying no matter what relaxations you do, you always have a reasonable estimate in the sense that it's greater than or equal to the true shortest path weight. So, it should be converging from above. So, that's the lemma.

Let's prove it. Any suggestions on how we should prove this lemma? What technique might we use?

What's that? Cut and paste?

It would be good for optimal substructure.

Cut and paste: maybe sort of what's going on here but not exactly. Something a little more general. It's just intuition here; it doesn't have to be the right answer.

In fact, many answers are correct, have plausible proofs.

Induction, yeah. So, I'm not going to write induction here, but effectively we are using induction. That's the answer I was expecting. So, there is sort of an induction already in time going on here.

We say after initialization it should be true.

That's our base case. And then, every relaxation we do, it should still be true. So, we're going to assume by induction that all the previous relaxations worked, and then we're going to prove that the last relaxation, whatever it is, works.

So, first let's do the base case.

So, this is after an initialization, let's say, initially. So, initially we have d of s equal to zero. And we have d of v equal to infinity for all other vertices, for all vertices, v, not equal to little s. OK, now we have to check that this inequality holds. Well, we have delta of s, s. We've already argued that that's zero. You can't get negative when there are only nonnegative edge weights.

So, that's the best. So, certainly zero is greater than or equal to zero. And, we have everything else, well, I mean, delta of S, v is certainly less than or equal to infinity. So this holds.

Everything is less than or equal to infinity.

So: base case is done. So, now we do an induction.

And, I'm going to write it as a proof by contradiction.

So, let's say, suppose that this fails to hold at some point. So, suppose for contradiction that the invariant is violated. So, we'd like to sue the violator and find a contradiction.

So, it's going to be violated. So, let's look at the first violation, the first time it's violated.

So, this is, essentially, again, a proof by induction. So, let's say we have some violation, d of v is less than delta of s, v.

That would be bad if we somehow got an estimate smaller than the shortest path. Well, then I think about looking

at the first violation is we know sort of by induction that all other values are correct.

OK, d of v is the first one where we've screwed up.

So, the invariant holds everywhere else.

Well, what caused this to fail, this invariant to be violated, is some relaxation, OK, on d of v.

So, we had some d of v, and we replaced it with some other d of u plus the weight of the edge from u to v.

And somehow, this made it invalid.

So, d of v is somehow less than that.

We just set d of v to this. So, this must be less than delta of s, v. The claim is that that's not possible because, let me rewrite a little bit.

We have d of u plus w of (u,v). And, we have our induction hypothesis, which holds on u, u of some other vertex.

We know that d of u is at least delta of s, u.

So, this has to be at least delta of s, u plus w of u, v. Now, what about this w of u, v? Well, that's some path from u to v. So, it's got to be bigger than the shortest path or equal. So certainly, this is greater than or equal to delta of u, v. OK, it could be larger if there's some multi-edged path that has a smaller total weight, but it's certainly no smaller than delta of u, v. And, this looks like a good summation, delta of S to u, and u to v is a triangle inequality, yeah. So, that is, it's upside down here. But, the triangle S, u, u to v, so this is only longer than S to v.

OK, so we have this thing, which is simultaneously greater than or equal to the shortest path weight from S to v, and also strictly less than the shortest path weight from S to v. So, that's a contradiction.

Maybe contradiction is the most intuitive way isn't the most intuitive way to proceed. The intuition, here, is whatever you assign d of v, you have a path in mind.

You inductively had a path from s to u.

Then you added this edge. So, that was a real path.

We always know that every path has weight greater than or equal to the shortest path. So, it should be true, and here's the inductive proof. All right, moving right along, so this was an easy warm-up. We have greater than or equal to. Now we have to prove less than or equal to at the end of the algorithm.

This is true all the time; less than or equal to will only be true at the end. So, we are not going to prove less than or equal to quite yet. We're going to prove another lemma, which again, so both of these lemmas are useful for other algorithms, too.

So, we're sort of building some shortest path theory that we can apply later. This one will give you some intuition about why relaxation, not only is it not bad, it's actually good. Not only does it not screw up anything, but it also makes progress in the following sense.

So, suppose you knew the shortest path from s to some vertex. OK, so you go from s to some other vertices. Then you go to u.

Then you go to v. Suppose that is a shortest path from s to v. OK, and also suppose that we already know in d of u the shortest path weight from s to u. So, suppose we have this equality. We now know that we always have a greater than or equal to. Suppose they are equal for u, OK, the vertex just before v in the shortest path.

OK, and suppose we relax that edge, (u,v), OK, which is exactly this step. This is relaxing the edge, (u,v). But we'll just call it relaxation here. After that relaxation, d of v equals delta of (s,v). So, if we had the correct answer for u, and we relax (u,v), then we get the correct answer for v.

OK, this is good news. It means, if inductively we can somehow get the right answer for u, now we know how to get the right answer for v. In the algorithm, we don't actually know what the vertex just before v in the shortest path is, but in the analysis we can pretty much know that. So, we have to prove this lemma. This is actually even easier than the previous one: don't even need induction because you just work through what's going on in relaxation, and it's true. So, here we go.

So, we're interested in this value, delta of Ss v.

And we know what the shortest path is.

So, the shortest path weight is the weight of this path.

OK, so we can write down some equality here.

Well, I'm going to split out the first part of the path and the last part of the path. So, we have, I'll say, the weight from s, so, this part of the path from s to u, plus the weight of this edge, u, v.

Remember, we could write w of a path, and that was the total weight of all those edges. So, what is this, the weight of this path from S to u?

Or, what property should I use to figure out what that value is? Yeah?

s to v is the shortest path, right?

So, by optimal substructure, from s to u is also a shortest path. So, this is delta of s, u. Cool.

We'll hold on for now. That's all we're going to say.

On the other hand, we know from this lemma that matter what we do, d of v is greater than or equal to delta of s, v.

So, let's write that down. So, there's a few cases, and this will eliminate some of the cases.

By that lemma correctness one, we know that d of v is greater than or equal to delta of s, v.

So, it's either equal or greater than at all times.

So, I'm thinking about the time before we do the relaxation, this (u,v). So, at that point, this is certainly true. So, either they're equal before relaxation or it's greater.

OK, if they are equal before relaxation, we're happy because relaxation only decreases values by correctness one.

It can't get any smaller than this, so after relaxation it will also be equal. OK, so in this case we're done.

So, that's a trivial case. So let's now suppose that d of v is greater than delta of s, v before relaxation.

That's perfectly valid. Hopefully now we fix it.

OK, well the point is, we know this delta s, v. It is this sum.

OK, we also know this. So, delta of s, u we know is d of u. And, we have this w u, v. So, delta of s, v is d of u plus w of (u,v) because we are assuming we have this shortest path structure where you go from s to u, and then you follow the edge, (u,v).

So, we know this. So, we know d of v is greater than d of u plus w of (u,v). By golly, that's this condition in relaxation. So, we're just checking, relaxation actually does something here.

OK, if you had the wrong distance estimate, this if condition is satisfied. Therefore, we do this.

So, in this case, we relax.

So, I'm just relaxing. Then, we set d of v to d of u plus WUV, which is what we want. OK, so we set d of v to d of u plus w of (u,v). And, this equals, as we said here, delta of S, v, which is what we wanted to prove.

Done. OK, I'm getting more and more excited as we get into the meat of this proof.

Any questions so far? Good.

Now comes the hard part. These are both very easy lemmas, right? I'll use these two boards.

We don't need these proofs anymore.

We just need these statements: correctness one, correctness lemma; great names.

So, now finally we get to correctness two.

So, we had one and one and a half.

So, I guess correctness is, itself, a mini-trilogy, the mini-series. OK, so correctness two says when the algorithm is done, we have the right answer.

This is really correctness. But, it's going to build on correctness one and correctness lemma.

So, we want d of v to equal delta of s, v for all vertices, v at the end of the algorithm. That is clearly our goal.

Now, this theorem is assuming that all of the weights are nonnegative, just to repeat. It doesn't assume anything else. So, it's going to get the infinities right. But, if there are minus infinities, all bets are off. OK, even if there's any negative weight edge anywhere, it's not going to do the right thing necessarily. But, assuming all the weights are nonnegative, which is reasonable if they're measuring time. Usually it costs money to travel along edges. They don't pay you to do it.

But who knows? So, I need just to say a few things. One of the things we've mentioned somewhere along the way is when you add a vertex to S, you never change its weight. OK, that actually requires proof. I'm just going to state it here. It's not hard to see.

d of v doesn't change. OK, this is essentially an induction once v is added to S. OK, this will actually followed by something we'll say in a moment.

OK, so all I really care about is when a vertex is added to S, we better have the right estimate because after that, we're not going to change it, let's say.

OK, we could define the algorithm that way.

We are not, but we could. I'll say more about this in a second. So, all we care about is whether d of v equals delta of s, v.

That's what we want to prove. So, it's clearly that.

It should be true at the end. But, it suffices to prove that it holds when v is added to S, to capital S.

OK, this actually implies the first statement.

It has sort of a funny implication.

But, if we can prove this, that d of v equals delta of s, v, when you add to S, we know relaxation only decreases value. So, it can't get any smaller.

It would be from correctness one.

Correctness one says we can't get any smaller than delta.

So, if we get a quality at that point, we'll have a quality from then on. So, that actually implies d of v never changes after that point.

OK, so we're going to prove this.

Good. Well, suppose it isn't true.

So this would be a proof by a contradiction.

Suppose for contradiction that this fails to hold.

And, let's look at the first failure.

Suppose u is the first vertex -- -- that's about to be added to S.

I want to consider the time right before it's added to S, for which we don't have what we want.

These are not equal. d of u does not equal delta of s, u. Well, if they're not equal, we know from correctness one that d of E is strictly greater than delta of s, u, so, d of u.

So, we have d of u is strictly greater than delta of s, u. OK, that's the beginning of the proof, nothing too exciting yet, just some warm-up.

OK, but this, used already correctness one.

I think that's the only time that we use it in this proof.

OK, so I sort of just want to draw picture of what's going on.

But I need a little bit of description.

So, let's look at the shortest path.

Somehow, d of u is greater than the shortest path.

So, consider the shortest path or a shortest path.

Let p be a shortest path, not just any shortest path, but the shortest path from s to u.

OK, so that means that the weight of this path is the shortest path weight. So, we have some equations for what's going on here. So, we care about delta of s, u. Here's a path with that weight.

It's got to be one because shortest paths exist here; slight exceptional cases if it's a plus infinity, but I'm not going to worry about that.

So, let me draw a picture somewhere.

So, we have s. We have u.

Here is the shortest path from s to u.

That's p. No idea what it looks like so far. Now, what we also have is the notion of capital S. So, I'm going to draw capital S. So, this is big S.

We know that little s is in big S.

We know that u is not yet in big S.

So, I haven't screwed up anything yet, right? This path starts in S and leaves it at some point because until we are about to add u to S, so it hasn't happened yet, so u is not in S.

Fine. What I want to do is look at the first place here where the path, p, exits S.

So, there is some vertex here. Let's call it x.

There's some vertex here. We'll call it y.

OK, possibly x equals S. Possibly y equals u.

But it's got to exit somewhere, because it starts inside and ends up outside. And it's a finite path.

OK, so consider the first time it happens; not the second time, the first. OK, so consider the first edge, (x,y), where p exits capital S. The shortest path from s to u exits capital S. It's got to happen somewhere.

Cool, now, what do we know? Little x is in S.

So, it has the right answer because u, we were about to add u to S, and that was the first violation of something in S that has the wrong d of x estimate. So, d of x equals delta of s, x. Because we are looking at the first violation, x is something that got added before. So, by induction on time, or because we had the first violation, d of x equals the shortest path weight from S to x.

So, that's good news. Now we are trying to apply this lemma. It's the only thing left to do.

We haven't used this lemma for anything.

So, we have the setup. If we already know that one of the d values is the right answer, and we relaxed the edge that goes out from it, then we get another right answer. So that's what I want to argue over here. We know that the d of x equals this weight because, again, subpaths of shortest paths are shortest paths. We have optimal substructure, so this is a shortest path, from S to x.

It might not be the only one, but it is one.

So we know that matches. Now, I want to think about relaxing this edge, (x,y).

Well, x is in capital S. And, the algorithm says, whenever you add a vertex, u, to the big set, S, you relax all the edges that go out from there.

OK, so when we added x to S, and we now look far in the future, we're about to add some other vertex.

Right after we added x to S, we relax this edge, (x,y), because we relaxed every edge that goes out from x, OK, whatever they were. Some of them went into S.

Some of them went out. Here's one of them.

So, when we added x to S, we got XS.

When we added x to S, we relaxed the edge, (x,y). OK, so now we're going to use the lemma. So, by the correctness lemma -- What do you get? Well, we add this correct shortest path weight to x now. We relax the edge, (x,y). So, now we should have the correct shortest path weight for y.

d of y equals delta of s, y.

OK, this is sometime in the past.

In particular, now, it should still be true because once you get down to the right answer you never change it. OK, we should be done.

OK, why are we done? Well, what else do we know here? We assumed something for contradiction, so we better contradict that.

We assume somehow, d of u is strictly greater than delta of s, u. So, d of u here is strictly greater than the length of this whole path.

Well, we don't really know whether u equals y.

It could, could not. And, but what do we know about this shortest path from S to y? Well, it could only be shorter than from S to u because it's a subpath.

And it's the shortest path because it's the subpath of the shortest path. The shortest path from S to y has to be less than or equal to the shortest path from S to u.

OK, S to y: less than or equal to s, u, OK, just because the subpath. I'm closer.

I've got delta of s, u now.

Somehow, I want to involve d of u.

So, I want to relate d of y to d of u.

What do I know about d of u? Yeah?

d of u is smaller because we have a Min heap, yeah. We always chose, let's erase, it's way down here.

We chose u. This is the middle of the algorithm. It's the reason I kept this to be the minimum key. This is keyed on d.

So, we know that at this moment, when we're trying to add u to S, right, y is not in S, and u is not in S. They might

actually be the same vertex. But both of these vertices, same or not, are outside S.

We chose u because d of u has the smallest d estimate.

So, d of y has to be greater than or equal to d of u.

It might be equal if they're the same vertex, but it's got to be greater than or equal to.

So, d of y here is greater than or equal to d of u.

So, here we're using the fact that we actually made a greedy choice. It's the one place we're using the greedy choice. Better use it somewhere because you can't just take an arbitrary vertex and declare it to be done. You've got to take the greedy one. OK, now we have d of u is less than or equal to delta of s, u, which contradicts this.

OK, sort of magical that that all just worked out.

But sort of like the previous proofs, you just see what happens and it works. OK, that's the approximation.

The only real idea here is to look at this edge.

In fact, you could look at this edge too.

But let's look at some edge that comes from S and goes out of S, and argue that while x has to be correct, and what we made x correct, y had to be correct, and now, why the hell are we looking at u?

y is the thing you should have looked at.

And, there you get a contradiction because y had the right answer. If u equals y, that's fine, or if u and y were sort of equally good, that's also fine if all these weights were zero. So, the picture might actually look like this. But, in that case, d of u is the correct answer. It was delta SU.

We assumed that it wasn't. That's where we're getting a contradiction. Pretty clear?

Go over this proof. It's a bit complicated, naturally. OK, we have a little bit more to cover, some easier stuff. OK, the first thing is what's the running time of this algorithm?

I'll do this very quick because we're actually seen this many times before last class. There was some initialization.

The initialization, which is no longer here, is linear time. No big deal.

OK, extract Min. Well, that's some data structure. So, we have something like size of V. Every vertex we extract

the Min once, and that's it. So, size of V, extract mins. OK, so that's pretty simple.

OK, then we had this main loop. This is a completely conceptual operation. S is not actually used in the algorithm. It's just for thinking.

OK, so this takes zero time. Got to love it.

OK, and now the heart is here. So, how many times does this loop iterate? That's the degree of u.

So, what is the total number of times that we execute a relaxation step? It doesn't necessarily mean we do this, but we at least execute this body.

Over the whole algorithm, how many times do we do this?

Every vertex, we look at all the outgoing edges from there. So, the total would be?

Number of edges, yeah.

So, this number of edges iterations.

OK, this is essentially the handshaking lemma we saw last time, but for directed graphs. And we are only looking at the outgoing edges. So, it's not a factor of two here because you're only outgoing from one side.

So, we have number of reiterations.

In the worst case, we do a decreased key for everyone. So, at most: E decreased keys. OK, so the time is, well, we have v extract Mins, so the time to do an extract Min, whatever that is. And we have E decreased keys, whatever that is, and this is exactly the running time we had for Prim's algorithm for a minimum spanning tree last time. And, it depends what data structure you use, what running time you get.

So, I'm going to skip the whole table here.

But, if you use an array, the final running time will be V^2 because you have order of v extract Min, and you have constant time decreased key. If you use a binary heap, which we know and love, then we have order log v for each operation. And so, this is V plus E log V.

And, so that's what we know how to do.

And, if you use this fancy data structure called a Fibonacci heap, you get constant time decreased key amortized.

And, you get an E plus v log v worst case bound on the running time. So, this is the best we know how to solve

shortest paths without any extra assumptions, single source shortest paths with non-negative edge weights in general. OK, this is almost as good and this is sometimes better than that.

But these are essentially irrelevant except that you know how to do these. You don't know how to do a Fibonacci heap unless you read that in the chapter of the book.

That's why we mention the top two running times.

OK, I want to talk briefly about a simpler case, which you may have seen before. And so it's sort of fun to connect this up to breadth first search in a graph.

So, I mean that ends Dijkstra, so to speak.

But now I want to think about a special case where the graph is unweighted, meaning w of (u,v) equals one for all vertices, u and v. OK, suppose we had that property. Can we do any better than Dijkstra? Can we do better than this running time? Well, we probably have to look at all the edges and all the vertices.

So, the only thing I'm questioning is this log v.

Can I avoid that? I gave away the answer a little bit. The answer is called breadth first search, or BFS, which you have probably seen before. Next to depth first search, it's one of the standard ways to look at the graph.

But we can say a little bit more than you may have seen before. Breadth for search is actually Dijkstra's algorithm: kind of nifty.

There are two changes. First change is that breadth for search does not use a priority queue.

I'll just tell you what it uses instead.

You can use a queue first in first out honest-to-goodness queue instead of a priority queue.

OK, it turns out that works. Instead of doing extract Min, you just take the first thing off the queue.

Instead of doing decreased key, OK, here's a subtlety.

But, this if statement changes a little bit.

So, here is the relaxation step.

So, in order to relax, you say this much simpler thing. If we haven't visited v yet, then we declare it to have the shortest path weight, say, d of v is d of u plus one, which is the weight of the edge, (u,v). And we add v to the end

of the queue. So, now, we start with the queue empty. Actually, it will just contain the vertex, S, because that's the only thing we know the shortest path for. So, the queue is just for, I know the shortest path of this thing.

Just deal with it when you can't look at all the outgoing edges when you can. So, initially that's just S.

You say, well, for all the outgoing edges, S has zero. All the outgoing edges from there have weight one. The shortest path weight from the source is one. You certainly can't do any better than that if all the weights are one.

OK, so we add all those vertices to the end of the queue. Then, we process things in order, and we just keep incrementing, if their value is d of u, add one to it.

That's d of v. And then we are going to add v to S what we get to it in the queue.

OK, that is breadth for search, very simple.

And, you can look at the text for the algorithm and for an example because I don't have time to cover that.

But the key thing is that the time is faster.

The time is order V plus E because as before, we only look at each edge once we look at all the outgoing edges from all the vertices. As soon as we set d of v to something, it will remain that. We never touch it.

We are going to add it to S. That only happens once.

So, this if statement, and so on, in the in-queuing, is done order E times, or actually E times, exactly. An in-queuing to a queue, and de-queuing from a queue, that's what we use instead of extract Min, take constant time, so the total running time, number of vertices plus the number of edges.

OK, not so obvious that this works, but you can prove that it works using the Dijkstra analysis.

All you have to do is prove that the FIFO priority queue.

Once you know that, by the correctness of Dijkstra you get the correctness of breadth for search.

So, not only is breadth for search finding all the vertices, which is maybe what you normally use it for, but it finds the shortest path weights from S to every other vertex when the weights are all one.

So, there we go: introduction to shortest paths.

Next time we'll deal with negative weights.