MIT OpenCourseWare
http://ocw.mit.edu

6.046J Introduction to Algorithms, Fall 2005

Please use the following citation format:

Erik Demaine and Charles Leiserson, *6.046J Introduction to Algorithms, Fall 2005*. (Massachusetts Institute of Technology: MIT OpenCourseWare). http://ocw.mit.edu (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms

Good morning, everyone. Glad you are all here bright and early. I'm counting the days till the TA's outnumber the students. They'll show up. We return to a familiar story. This is part two, the Empire Strikes Back. So last time, our adversary, the graph, came to us with a problem. We have a source, and we had a directed graph, and we had weights on the edges, and they were all nonnegative. And there was happiness. And we triumphed over the Empire by designing Dijkstra's algorithm, and very efficiently finding single source shortest paths, shortest path weight from s to every other vertex.

Today, however, the Death Star has a new trick up its sleeve, and we have negative weights, potentially. And we're going to have to somehow deal with, in particular, negative weight cycles. And we saw that when we have a negative weight cycle, we can just keep going around, and around, and around, and go back in time farther, and farther, and farther. And we can get to be arbitrarily far back in the past. And so there's no shortest path, because whatever path you take you can get a shorter one. So we want to address that issue today, and we're going to come up with a new algorithm actually simpler than Dijkstra, but not as fast, called the Bellman-Ford algorithm.

And, it's going to allow negative weights, and in some sense allow negative weight cycles, although maybe not as much as you might hope. We have to leave room for a sequel, of course. OK, so the Bellman-Ford algorithm, invented by two guys, as you might expect, it computes the shortest path weights. So, it makes no assumption about the weights. Weights are arbitrary, and it's going to compute the shortest path weights. So, remember this notation: delta of s, v is the weight of the shortest path from s to v. s was called a source vertex.

And, we want to compute these weights for all vertices, little v. The claim is that computing from s to everywhere is no harder than computing s to a particular location. So, we're going to do for all them. It's still going to be the case here. And, it allows negative weights. And this is the good case, but there's an alternative, which is that Bellman-Ford may just say, oops, there's a negative weight cycle.

And in that case it will just say so. So, they say a negative weight cycle exists. Therefore, some of these deltas are minus infinity. And that seems weird. So, Bellman-Ford as we'll present it today is intended for the case, but there are no negative weights cycles, which is more intuitive. It sort of allows them, but it will just report them. In that case, it will not give you delta values. You can change the algorithm to give you delta values in that case, but we are not going to see it here. So, in exercise, after you see the algorithm, exercise is: compute these deltas in all cases.

So, it's not hard to do. But we don't have time for it here. So, here's the algorithm. It's pretty straightforward. As I said, it's easier than Dijkstra. It's a relaxation algorithm. So the main thing that it does is relax edges just like Dijkstra. So, we'll be

able to use a lot of dilemmas from Dijkstra. And proof of correctness will be three times shorter because the first two thirds we already have from Dijkstra. But I'm jumping ahead a bit. So, the first part is initialization. Again, d of v will represent the estimated distance from s to v.

And we're going to be updating those estimates as the algorithm goes along. And initially, d of s is zero, which now may not be the right answer conceivably. Everyone else is infinity, which is certainly an upper bound. OK, these are both upper bounds on the true distance. So that's fine. That's initialization just like before. And now we have a main loop which happens v minus one times. We're not actually going to use the index i. It's just a counter.

And we're just going to look at every edge and relax it. It's a very simple idea. If you learn about relaxation, this is the first thing you might try. The question is when do you stop. It's sort of like I have this friend to what he was like six years old he would claim, oh, I know how to spell banana. I just don't know when to stop. OK, same thing with relaxation. This is our relaxation step just as before. We look at the edge; we see whether it violates the triangle inequality according to our current estimates we know the distance from s to v should be at most distance from s to plus the weight of that edge from u to v. If it isn't, we set it equal. We've proved that this is always an OK thing to do.

We never violate, I mean, these d of v's never get too small if we do a bunch of relaxations. So, the idea is you take every edge. You relax it. I don't care which order. Just relax every edge, one each. And that do that V minus one times. The claim is that that should be enough if you have no negative weights cycles. So, if there's a negative weight cycle, we need to figure it out. And, we'll do that in a fairly straightforward way, which is we're going to do exactly the same thing. So this is outside before loop here. We'll have the same four loops for each edge in our graph. We'll try to relax it. And if you can relax it, the claim is that there has to be a negative weight cycle.

So this is the main thing that needs proof. OK, and that's the algorithm. So the claim is that at the ends we should have d of v, let's see, L's so to speak. d of v equals delta of s comma v for every vertex, v. If we don't find a negative weight cycle according to this rule, that we should have all the shortest path weights. That's the claim. Now, the first question is, in here, the running time is very easy to analyze. So let's start with the running time. We can compare it to Dijkstra, which is over here.

What is the running time of this algorithm? V times E, exactly. OK, I'm going to assume, because it's pretty reasonable, that V and E are both positive. Then it's V times E. So, this is a little bit slower, or a fair amount slower, than Dijkstra's algorithm. There it is: E plus V log V is essentially, ignoring the logs is pretty much linear time. Here we have something that's at least quadratic in V, assuming your graph is connected. So, it's slower, but it's going to handle these negative weights. Dijkstra can't handle negative weights at all. So, let's do an example, make it clear why you might hope this algorithm works.

And then we'll prove that it works, of course. But the proof will be pretty easy. So, I'm going to draw a graph that has negative weights, but no negative weight cycles so that I get an interesting answer. Good. The other thing I need in order to make the output of this algorithm well defined, it depends in which order you visit the

edges. So I'm going to assign an arbitrary order to these edges. I could just ask you for an order, but to be consistent with the notes, I'll put an ordering on it. Let's say I put number four, say that's the fourth edge I'll visit. It doesn't matter. But it will affect what happens during the algorithm for a particular graph.

Do they get them all? One, two, three, four, five, six, seven, eight, OK. And my source is going to be A. And, that's it. So, I want to run this algorithm. I'm just going to initialize everything. So, I set the estimates for s to be zero, and everyone else to be infinity. And to give me some notion of time, over here I'm going to draw or write down what all of these d values are as the algorithm proceeds because I'm going to start crossing them out and rewriting them that the figure will get a little bit messier. But we can keep track of it over here. It's initially zero and infinities. Yeah?

It doesn't matter. So, for the algorithm you can go to the edges in a different order every time if you want. We'll prove that, but here I'm going to go through the same order every time. Good question. It turns out it doesn't matter here. OK, so here's the starting point. Now I'm going to relax every edge. So, there's going to be a lot of edges here that don't do anything. I try to relax n minus one. I'd say, well, I know how to get from s to B with weight infinity.

Infinity plus two I can get to from s to E. Well, infinity plus two is not much better than infinity. OK, so I don't do anything, don't update this to infinity. I mean, infinity plus two sounds even worse. But infinity plus two is infinity. OK, that's the edge number one. So, no relaxation edge number two, same deal as number three, same deal, edge number four we start to get something interesting because I have a finite value here that says I can get from A to B using a total weight of minus one.

So that seems good. I'll write down minus one here, and update B to minus one. The rest stay the same. So, I'm just going to keep doing this over and over. That was edge number four. Number five, we also get a relaxation. Four is better than infinity. So, c gets a number of four. Then we get to edge number six. That's infinity plus five is worse than four. OK, so no relaxation there. Edge number seven is interesting because I have a finite value here minus one plus the weight of this edge, which is three. That's a total of two, which is actually better than four. So, this route, A, B, c is actually better than the route I just found a second ago. So, this is now a two. This is all happening in one iteration of the main loop.

We actually found two good paths to c. We found one better than the other. OK, and that was edge number seven, and edge number eight is over here. It doesn't matter. OK, so that was round one of this outer loop, so, the first value of i. i equals one. OK, now we continue. Just keep going. So, we start with edge number one. Now, minus one plus two is one. That's better than infinity. It'll start speeding up. It's repetitive. It's actually not too much longer until we're done. Number two, this is an infinity so we don't do anything. Number three: minus one plus two is one; better than infinity. This is vertex d, and it's number three. Number four we've already done.

Nothing changed. Number five: this is where we see the path four again, but that's worse than two. So, we don't update anything. Number six: one plus five is six, which is bigger than two, so no good. Go around this way. Number seven: same deal. Number eight is interesting. So, we have a weight of one here, a weight of minus three here. So, the total is minus two, which is better than one. So, that was d. And, I believe that's it. So that was definitely the end of that round. So, it's I plus

two because we just looked at the eighth edge. And, I'll cheat and check. Indeed, that is the last thing that happens.

We can check the couple of outgoing edges from d because that's the only one whose value just changed. And, there are no more relaxations possible. So, that was in two rounds. The claim is we got all the shortest path weights. The algorithm would actually loop four times to guarantee correctness because we have five vertices here and one less than that. So, in fact, in the execution here there are two more blank rounds at the bottom. Nothing happens. But, what the hell?

OK, so that is Bellman-Ford. I mean, it's certainly not doing anything wrong. The question is, why is it guaranteed to converge in V minus one steps unless there is a negative weight cycle? Question? Right, so that's an optimization. If you discover a whole round, and nothing happens, so you can keep track of that in the algorithm thing, you can stop. In the worst case, it won't make a difference. But in practice, you probably want to do that. Yeah? Good question. All right, so some simple observations, I mean, we're only doing relaxation. So, we can use a lot of our analysis from before. In particular, the d values are only decreasing monotonically. As we cross out values here, we are always making it smaller, which is good. Another nifty thing about this algorithm is that you can run it even in a distributed system.

If this is some actual network, some computer network, and these are machines, and they're communicating by these links, I mean, it's a purely local thing. Relaxation is a local thing. You don't need any global strategy, and you're asking about, can we do a different order in each step? Well, yeah, you could just keep relaxing edges, and keep relaxing edges, and just keep going for the entire lifetime of the network. And eventually, you will find shortest paths. So, this algorithm is guaranteed to finish in V rounds in a distributed system. It might be more asynchronous. And, it's a little harder to analyze.

But it will still work eventually. It's guaranteed to converge. And so, Bellman-Ford is used in the Internet for finding shortest paths. OK, so let's finally prove that it works. This should only take a couple of boards. So let's suppose we have a graph and some edge weights that have no negative weight cycles. Then the claim is that we terminate with the correct answer. So, Bellman-Ford terminates with all of these d of v values set to the delta values for every vertex.

OK, the proof is going to be pretty immediate using the lemmas that we had from before if you remember them. So, we're just going to look at every vertex separately. So, I'll call the vertex v. The claim is that this holds by the end of the algorithm. So, remember what we need to prove is that at some point, d of v equals delta of s comma v because we know it decreases monotonically, and we know that it never gets any smaller than the correct value because relaxations are always safe.

So, we just need to show at some point this holds, and that it will hold at the end. So, by monotonicity of the d values, and by correctness part one, which was that the d of v's are always greater than or equal to the deltas, we only need to show that at some point we have equality. So that's our goal. So what we're going to do is just look at v, and the shortest path to v, and see what happens to the algorithm relative to that path.

So, I'm going to name the path. Let's call it p. It starts at vertex v_0 and goes to v_1, v_2, whatever, and ends at v_k. And, this is not just any shortest path, but it's one that starts at s. So, v_0's s, and it ends at v. So, I'm going to give a couple of names to s and v so I can talk about the path more uniformly. So, this is a shortest path from s to v. Now, I also want it to be not just any shortest path from s to v, but among all shortest paths from s to v I want it to be one with the fewest possible edges.

OK, so shortest here means in terms of the total weight of the path. Subject to being shortest in weight, I wanted to also be shortest in the number of edges. And, the reason I want that is to be able to conclude that p is a simple path, meaning that it doesn't repeat any vertices. Now, can anyone tell me why I need to assume that the number of edges is the smallest possible in order to guarantee that p is simple?

The claim is that not all shortest paths are necessarily simple. Yeah? Right, I can have a zero weight cycle, exactly. So, we are hoping, I mean, in fact in the theorem here, we're assuming that there are no negative weight cycles. But there might be zero weight cycles still. As a zero weight cycle, you can put that in the middle of any shortest path to make it arbitrarily long, repeat vertices over and over. That's going to be annoying. What I want is that p is simple.

And, I can guarantee that essentially by shortcutting. If ever I take a zero weight cycle, I throw it away. And this is one mathematical way of doing that. OK, now what else do we know about this shortest path? Well, we know that subpaths are shortest paths are shortest paths. That's optimal substructure. So, we know what the shortest path from s to v_i is sort of inductively. It's the shortest path, I mean, it's the weight of that path, which is, in particular, the shortest path from s to v minus one plus the weight of the last edge, v minus one to v_i.

So, this is by optimal substructure as we proved last time. OK, and I think that's pretty much the warm-up. So, I want to sort of do this inductively in I, start out with v zero, and go up to v_k. So, the first question is, what is d of v_0, which is s? What is d of the source? Well, certainly at the beginning of the algorithm, it's zero. So, let's say equals zero initially because that's what we set it to. And it only goes down from there. So, it certainly, at most, zero.

The real question is, what is delta of s comma v_0. What is the shortest path weight from s to s? It has to be zero, otherwise you have a negative weight cycle, exactly. My favorite answer, zero. So, if we had another path from s to s, I mean, that is a cycle. So, it's got to be zero. So, these are actually equal at the beginning of the algorithm, which is great. That means they will be for all time because we just argued up here, only goes down, never can get too small. So, we have d of v_0 set to the right thing. Great: good for the base case of the induction. Of course, what we really care about is v_k, which is v. So, let's talk about the v_i inductively, and then we will get v_k as a result.

So, yeah, let's do it by induction. That's more fun. Let's say that d of v_i is equal to delta of s v_i after I rounds of the algorithm. So, this is actually referring to the I that is in the algorithm here. These are rounds. So, one round is an entire execution of all the edges, relaxation of all the edges. So, this is certainly true for I equals zero. We just proved that. After zero rounds, at the beginning of the algorithm, d of v_0 equals delta of s, v_0.

OK, so now, that's not really what I wanted, but OK, fine. Now we'll prove it for d of v_i plus one. Generally, I recommend you assume something. In fact, why don't I follow my own advice and change it? It's usually nicer to think of induction as recursion. So, you assume that this is true, let's say, for j less than the i that you care about, and then you prove it for d of v_i. It's usually a lot easier to think about it that way. In particular, you can use strong induction for all less than i. Here, we're only going to need it for one less. We have some relation between I and I minus one here in terms of the deltas. And so, we want to argue something about the d values.

OK, well, let's think about what's going on here. We know that, let's say, after I minus one rounds, we have this inductive hypothesis, d of v_i minus one equals delta of s v_i minus one. And, we want to conclude that after i rounds, so we have one more round to do this. We want to conclude that d of v_i has the right answer, delta of s comma v_i. Does that look familiar at all? So we want to relax every edge in this round. In particular, at some point, we have to relax the edge from v_i minus one to v_i. We know that this path consists of edges. That's the definition of a path.

So, during the i'th round, we relax every edge. So, we better relax v_i minus one v_i. And, what happens then? It's a test of memory. Quick, the Death Star is approaching. So, if we have the correct value for v_i minus one, that we relax an outgoing edge from there, and that edge is an edge of the shortest path from s to v_i. What do we know? d of v_i becomes the correct value, delta of s comma v_i. This was called correctness lemma last time. One of the things we proved about Dijkstra's algorithm, but it was really just a fact about relaxation. And it was a pretty simple proof.

And it comes from this fact. We know the shortest path weight is this. So, certainly d of v_i was at least this big, and let's suppose it's greater, or otherwise we were done. We know d of v_i minus one is set to this. And so, this is exactly the condition that's being checked in the relaxation step. And, the d of v_i value will be greater than this, let's suppose. And then, we'll set it equal to this. And that's exactly d of s v_i.

So, when we relax that edge, we've got to set it to the right value. So, this is the end of the proof, right? It's very simple. The point is, you look at your shortest path. Here it is. And if we assume there's no negative weight cycles, this has the correct value initially. d of s is going to be zero. After the first round, you've got to relax this edge. And then you get the right value for that vertex. After the second round, you've got to relax this edge, which gets you the right d value for this vertex and so on.

And so, no matter which shortest path you take, you can apply this analysis. And you know that by, if the length of this path, here we assumed it was k edges, then after k rounds you've got to be done. OK, so this was not actually the end of the proof. Sorry. So this means after k rounds, we have the right answer for v_k, which is v. So, the only question is how big could k be? And, it better be the right answer, at most, v minus one is the claim by the algorithm that you only need to do v minus one steps. And indeed, the number of edges in a simple path in a graph is, at most, the number of vertices minus one.

k is, at most, v minus one because p is simple. So, that's why we had to assume that it wasn't just any shortest path. It had to be a simple one so it didn't repeat any vertices. So there are, at most, V vertices in the path, so at most, V minus one

edges in the path. OK, and that's all there is to Bellman-Ford. So: pretty simple in correctness. Of course, we're using a lot of the lemmas that we proved last time, which makes it easier. OK, a consequence of this theorem, or of this proof is that if Bellman-Ford fails to converge, and that's what the algorithm is checking is whether this relaxation still requires work after these d minus one steps.

Right, the end of this algorithm is run another round, a V'th round, see whether anything changes. So, we'll say that the algorithm fails to converge after V minus one steps or rounds. Then, there has to be a negative weight cycle. OK, this is just a contrapositive of what we proved. We proved that if you assume there's no negative weight cycle, then we know that d of s is zero, and then all this argument says is you've got to converge after v minus one rounds. There can't be anything left to do once you've reached the shortest path weights because you're going monotonically; you can never hit the bottom.

You can never go to the floor. So, if you fail to converge somehow after V minus one rounds, you've got to have violated the assumption. The only assumption we made was there's no negative weight cycle. So, this tells us that Bellman-Ford is actually correct. When it says that there is a negative weight cycle, it indeed means it. It's true. OK, and you can modify Bellman-Ford in that case to sort of run a little longer, and find where all the minus infinities are.

And that is, in some sense, one of the things you have to do in your problem set, I believe. So, I won't cover it here. But, it's a good exercise in any case to figure out how you would find where the minus infinities are. What are all the vertices reachable from negative weight cycle? Those are the ones that have minus infinities. OK, so you might say, well, that was awfully fast. Actually, it's not over yet. The episode is not yet ended. We're going to use Bellman-Ford to solve the even bigger and greater shortest path problems.

And in the remainder of today's lecture, we will see it applied to a more general problem, in some sense, called linear programming. And the next lecture, we'll really use it to do some amazing stuff with all pairs shortest paths. Let's go over here. So, our goal, although it won't be obvious today, is to be able to compute the shortest paths between every pair of vertices, which we could certainly do at this point just by running Bellman-Ford v times.

OK, but we want to do better than that, of course. And, that will be the climax of the trilogy. OK, today we just discovered who Luke's father is. So, it turns out the father of shortest paths is linear programming. Actually, simultaneously the father and the mother because programs do not have gender. OK, my father likes to say, we both took improv comedy lessons so we have degrees in improvisation.

And he said, you know, we went to improv classes in order to learn how to make our humor better. And, the problem is, it didn't actually make our humor better. It just made us less afraid to use it. [LAUGHTER] So, you are subjected to all this improv humor. I didn't see the connection of Luke's father, but there you go. OK, so, linear programming is a very general problem, a very big tool. Has anyone seen linear programming before? OK, one person. And, I'm sure you will, at some time in your life, do anything vaguely computing optimization related, linear programming comes up at some point. It's a very useful tool. You're given a matrix and two vectors: not too exciting yet. What you want to do is find a vector. This is a very dry description.

We'll see what makes it so interesting in a moment. So, you want to maximize some objective, and you have some constraints. And they're all linear. So, the objective is a linear function in the variables x, and your constraints are a bunch of linear constraints, inequality constraints, that's one makes an interesting. It's not just solving a linear system as you've seen in linear algebra, or whatever. Or, of course, it could be that there is no such x. OK: vaguely familiar you might think to the theorem about Bellman-Ford. And, we'll show that there's some kind of connection here that either you want to find something, or show that it doesn't exist.

Well, that's still a pretty vague connection, but I also want to maximize something, or are sort of minimize the shortest paths, OK, somewhat similar. We have these constraints. So, yeah. This may be intuitive to you, I don't know. I prefer a more geometric picture, and I will try to draw such a geometric picture, and I've never tried to do this on a blackboard, so it should be interesting. I think I'm going to fail miserably. It sort of looks like a dodecahedron, right? Sort of, kind of, not really. A bit rough on the bottom, OK. So, if you have a bunch of linear constraints, this is supposed to be in 3-D. Now I labeled it. It's now in 3-D. Good.

So, you have these linear constraints. That turns out to define hyperplanes in n dimensions. OK, so you have this base here that's three-dimensional space. So, n equals three. And, these hyperplanes, if you're looking at one side of the hyperplane, that's the less than or equal to, if you take the intersection, you get some convex polytope or polyhedron. In 3-D, you might get a dodecahedron or whatever. And, your goal, you have some objective vector c, let's say, up. Suppose that's the c vector.

Your goal is to find the highest point in this polytope. So here, it's maybe this one. OK, this is the target. This is the optimal, x. That is the geometric view. If you prefer the algebraic view, you want to maximize the c transpose times x. So, this is m. This is n. Check out the dimensions work out. So that's saying you want to maximize the dot product. You want to maximize the extent to which x is in the direction c. And, you want to maximize that subject to some constraints, which looks something like this, maybe. So, this is A, and it's m by n. You want to multiply it by, it should be something of height n.

That's x. Let me put x down here, n by one. And, it should be less than or equal to something of this height, which is B, the right hand side. OK, that's the algebraic view, which is to check out all the dimensions are working out. But, you can read these off in each row here, when multiplied by this column, gives you one value here. And as just a linear constraints on all the x sides. So, you want to maximize this linear function of x_1 up to x_n subject to these constraints, OK? Pretty simple, but pretty powerful in general. So, it turns out that with, you can formulate a huge number of problems such as shortest paths as a linear program. So, it's a general tool.

And in this class, we will not cover any algorithms for solving linear programming. It's a bit tricky. I'll just mention that they are out there. So, there's many efficient algorithms, and lots of code that does this. It's a very practical setup. So, lots of algorithms to solve LP's, linear programs. Linear programming is usually called LP. And, I'll mention a few of them. There's the simplex algorithm. This is one of the first. I think it is the first, the ellipsoid algorithm. There's interior point methods, and there's random sampling. I'll just say a little bit about each of these because we're not going to talk about any of them in depth.

The simplex algorithm, this is, I mean, one of the first algorithms in the world in some sense, certainly one of the most popular. It's still used today. Almost all linear programming code uses the simplex algorithm. It happens to run an exponential time in the worst-case, so it's actually pretty bad theoretically. But in practice, it works really well. And there is some recent work that tries to understand this. It's still exponential in the worst case.

But, it's practical. There's actually an open problem whether there exists a variation of simplex that runs in polynomial time. But, I won't go into that. That's a major open problem in this area of linear programming. The ellipsoid algorithm was the first algorithm to solve linear programming in polynomial time. So, for a long time, people didn't know. Around this time, people started realizing polynomial time is a good thing. That happened around the late 60s. Polynomial time is good. And, the ellipsoid algorithm is the first one to do it. It's a very general algorithm, and very powerful, theoretically: completely impractical. But, it's cool. It lets you do things like you can solve a linear program that has exponentially many constraints in polynomial time. You've got all sorts of crazy things. So, I'll just say it's polynomial time.

I can't say something nice about it; don't say it at all. It's impractical. Interior point methods are sort of the mixture. They run in polynomial time. You can guarantee that. And, they are also pretty practical, and there's sort of this competition these days about whether simplex or interior point is better. And, I don't know what it is today but a few years ago they were neck and neck. And, random sampling is a brand new approach. This is just from a couple years ago by two MIT professors, Dimitris Bertsimas and Santosh Vempala, I guess the other is in applied math.

So, just to show you, there's active work in this area. People are still finding new ways to solve linear programs. This is completely randomized, and very simple, and very general. It hasn't been implemented, so we don't know how practical it is yet. But, it has potential. OK: pretty neat. OK, we're going to look at a somewhat simpler version of linear programming. The first restriction we are going to make is actually not much of a restriction. But, nonetheless we will consider it, it's a little bit easier to think about. So here, we had some polytope we wanted to maximize some objective. In a feasibility problem, I just want to know, is the polytope empty? Can you find any point in that polytope? Can you find any set of values, x, that satisfy these constraints?

OK, so there's no objective. c, just find x such that AX is less than or equal to B. OK, it turns out you can prove a very general theorem that if you can solve linear feasibility, you can also solve linear programming. We won't prove that here, but this is actually no easier than the original problem even though it feels easier, and it's easier to think about. I was just saying actually no easier than LP. OK, the next restriction we're going to make is a real restriction. And it simplifies the problem quite a bit.

And that's to look at different constraints. And, if all this seemed a bit abstract so far, we will now ground ourselves little bit. A system of different constraints is a linear feasibility problem. So, it's an LP where there's no objective. And, it's with a restriction, so, where each row of the matrix, so, the matrix, A, has one one, and it has one minus one, and everything else in the row is zero.

OK, in other words, each constraint has its very simple form. It involves two variables and some number. So, we have something like x_j minus x_i is less than or equal to w_{ij}. So, this is just a number. These are two variables. There's a minus sign, no values up here, no coefficients, no other of the X_k's appear, just two of them. And, you have a bunch of constraints of this form, one per row of the matrix. Geometrically, I haven't thought about what this means. I think it means the hyperplanes are pretty simple. Sorry I can't do better than that. It's a little hard to see this in high dimensions.

But, it will start to correspond to something we've seen, namely the board that its next to, very shortly. OK, so let's do a very quick example mainly to have something to point at. Here's a very simple system of difference constraints -- -- OK, and a solution. Why not? It's not totally trivial to solve this, but here's a solution. And the only thing to check is that each of these constraints is satisfied. x_1 minus x_2 is three, which is less than or equal to three, and so on.

There could be negative values. There could be positive values. It doesn't matter. I'd like to transform this system of difference constraints into a graph because we know a lot about graphs. So, we're going to call this the constraint graph. And, it's going to represent these constraints. How'd I do it? Well, I take every constraint, which in general looks like this, and I convert it into an edge. OK, so if I write it as x_j minus x_i is less than or equal to some w_{ij}, w seems suggestive of weights.

That's exactly why I called it w. I'm going to make that an edge from v_i to v_j. So, the order flips a little bit. And, the weight of that edge is w_{ij}. So, just do that. Make n vertices. So, you have the number of vertices equals n. The number of edges equals the number of constraints, which is m, the height of the matrix, and just transform. So, for example, here we have three variables. So, we have three vertices, v_1, v_2, v_3. We have x_1 minus x_2. So, we have an edge from v_2 to v_1 of weight three. We have x_2 minus x_3. So, we have an edge from v_3 to v_2 of weight minus two. And, we have x_1 minus x_3. So, we have an edge from v_3 to v_1 of weight two. I hope I got the directions right.

Yep. So, there it is, a graph: currently no obvious connection to shortest paths, right? But in fact, this constraint is closely related to shortest paths. So let me just rewrite it. You could say, well, an x_j is less than or equal to x_i plus w_{ij}. Or, you could think of it as $d[j]$ less than or equal to $d[i]$ plus w_{ij}. This is a conceptual balloon. Look awfully familiar? A lot like the triangle inequality, a lot like relaxation. So, there's a very close connection between these two problems as we will now prove.

So, we're going to have two theorems. And, they're going to look similar to the correctness of Bellman-Ford in that they talk about negative weight cycles. Here we go. It turns out, I mean, we have this constraint graph. It can have negative weights. It can have positive weights. It turns out what matters is if you have a negative weight cycle. So, the first thing to prove is that if you have a negative weight cycle that something bad happens. OK, what could happen bad? Well, we're just trying to satisfy this system of constraints. So, the bad thing is that there might not be any solution. These constraints may be infeasible. And that's the claim. The claim is that this is actually an if and only if.

But first we'll proved the if. If you have a negative weight cycle, you're doomed. The difference constraints are unsatisfiable. That's a more intuitive way to say it. In the

LP world, they call it infeasible. But unsatisfiable makes a lot more sense. There's no way to assign the x_i's in order to satisfy all the constraints simultaneously. So, let's just take a look. Consider a negative weight cycle. It starts at some vertex, goes through some vertices, and at some point comes back. I don't care whether it repeats vertices, just as long as this cycle, from v_1 to v_1 is a negative weight cycle strictly negative weight.

OK, and what I'm going to do is just write down all the constraints. Each of these edges corresponds to a constraint, which must be in the set of constraints because we had that graph. So, these are all edges. Let's look at what they give us. So, we have an edge from v_1 to v_2. That corresponds to x_2 minus x_1 is, at most, something, w_{12}. Then we have x_3 minus x_2. That's the weight w_{23}, and so on.

And eventually we get up to something like x_k minus $x_{(k-1)}$. That's this edge: $w_{(k-1),k}$, and lastly we have this edge, which wraps around. So, it's x_1 minus x_k, w_{k1} if I've got the signs right. Good, so here's a bunch of constraints. What do you suggest I do with them? Anything interesting about these constraints, say, the left hand sides? Sorry? It sounded like the right word. What was it?

Telescopes, yes, good. Everything cancels. If I added these up, there's an x_2 and a minus x_2. There's a minus x_1 and an x_1. There's a minus XK and an XK. Everything here cancels if I add up the left hand sides. So, what happens if I add up the right hand sides? Over here I get zero, my favorite answer. And over here, we get all the weights of all the edges in the negative weight cycle, which is the weight of the cycle, which is negative.

So, zero is strictly less than zero: contradiction. Contradiction: wait a minute, we didn't assume anything that was false. So, it's not really a contradiction in the mathematical sense. We didn't contradict the world. We just said that these constraints are contradictory. In other words, if you pick any values of the x_i's, there is no way that these can all be true because that you would get a contradiction. So, it's impossible for these things to be satisfied by some real x_i's. So, these must be unsatisfiable.

Let's say there's no satisfying assignment, a little more precise, x_1 up to x_m, no weights. Can we satisfy those constraints? Because they add up to zero on the left-hand side, and negative on the right-hand side. OK, so that's an easy proof. The reverse direction will be only slightly harder. OK, so, cool. We have this connection. So motivation is, suppose you'd want to solve these difference constraints. And we'll see one such application. I Googled around for difference constraints. There is a fair number of papers that care about difference constraints. And, they all use shortest paths to solve them. So, if we can prove a connection between shortest paths, which we know how to compute, and difference constraints, then we'll have something cool. And, next class will see even more applications of difference constraints.

It turns out they're really useful for all pairs shortest paths. OK, but for now let's just prove this equivalence and finish it off. So, the reverse direction is if there's no negative weight cycle in this constraint graph, then the system better be satisfiable. The claim is that these negative weight cycles are the only barriers for finding a solution to these difference constraints. I have this feeling somewhere here. I had to talk about the constraint graph.

Good. Satisfied, good. So, here we're going to see a technique that is very useful when thinking about shortest paths. And, it's a bit hard to guess, especially if you haven't seen it before. This is useful in problem sets, and in quizzes, and finals, and everything. So, keep this in mind. I mean, I'm using it to prove this rather simple theorem, but the idea of changing the graph, so I'm going to call this constraint graph G. Changing the graph is a very powerful idea. So, we're going to add a new vertex, s, or source, use the source, Luke, and we're going to add a bunch of edges from s because being a source, it better be connected to some things.

So, we are going to add a zero weight edge, or weight zero edge from s to everywhere, so, to every other vertex in the constraint graph. Those vertices are called v_i, v_1 up to v_n. So, I have my constraint graph. But I'll copy this one so I can change it. It's always good to backup your work before you make changes, right? So now, I want to add a new vertex, s, over here, my new source. I just take my constraint graph, whatever it looks like, add in weight zero edges to all the other vertices.

Simple enough. Now, what did I do? What did you do? Well, I have a candidate source now which can reach all the vertices. So, shortest path from s, hopefully, well, paths from s exist. I can get from s to everywhere in weight at most zero. OK, maybe less. Could it be less? Well, you know, like v_2, I can get to it by zero minus two. So, that's less than zero. So I've got to be a little careful. What if there's a negative weight cycle? Oh no? Then there wouldn't be any shortest paths. Fortunately, we assume that there's no negative weight cycle in the original graph. And if you think about it, if there's no negative weight cycle in the original graph, we add an edge from s to everywhere else. We're not making any new negative weight cycles because you can start at s and go somewhere at a cost of zero, which doesn't affect any weights.

And then, you are forced to stay in the old graph. So, there can't be any new negative weight cycles. So, the modified graph has no negative weight cycles. That's good because it also has paths from s, and therefore it also has shortest paths from s. The modified graph has no negative weight because it didn't before. And, it has paths from s. There's a path from s to every vertex. There may not have been before. Before, I couldn't get from v_2 to v_3, for example. Well, that's still true. But from s I can get to everywhere. So, that means that this graph, this modified graph, has shortest paths.

Shortest paths exist from s. In other words, if I took all the shortest path weights, like I ran Bellman-Ford from s, then, I would get a bunch of finite numbers, d of v, for every value, for every vertex. That seems like a good idea. Let's do it. So, shortest paths exist. Let's just assign x_i to be the shortest path weight from s to v_i. Why not? That's a good choice for a number, the shortest path weight from s to v_i. This is finite because it's less than infinity, and it's greater than minus infinity, so, some finite number. That's what we need to do in order to satisfy these constraints.

The claim is that this is a satisfying assignment. Why? Triangle inequality. Somewhere here we wrote triangle inequality. This looks a lot like the triangle inequality. In fact, I think that's the end of the proof. Let's see here. What we want to be true with this assignment is that x_j minus x_i is less than or equal to w_{ij} whenever ij is an edge. Or, let's say v_i, v_j, for every such constraint, so, for v_i, v_j in the edge set.

OK, so what is this true? Well, let's just expand it out. So, x_i is this delta, and x_j is some other delta. So, we have delta of s, vj minus delta of s_vi. And, on the right-hand side, well, w_{ij}, that was the weight of the edge from I to J. So, this is the weight of v_i to v_j. OK, I will rewrite this slightly. Delta s, vj is less than or equal to delta s, vi plus w of v_i, v_j. And that's the triangle inequality more or less. The shortest path from s to v_j is, at most, shortest path from s to v_i plus a particular path from v_i to v_j, namely the single edge v_i to v_j. This could only be longer than the shortest path.

And so, that makes the right-hand side bigger, which makes this inequality more true, meaning it was true before. And now it's still true. And, that proves it. This is true. And, these were all equivalent statements. This we know to be true by triangle inequality. Therefore, these constraints are all satisfied. Magic. I'm so excited here. So, we've proved that having a negative weight cycle is exactly when these system of difference constraints are unsatisfiable.

So, if we want to satisfy them, if we want to find the right answer to x, we run Bellman-Ford. Either it says, oh, no negative weight cycle. Then you are hosed. Then, there is no solution. But that's the best you could hope to know. Otherwise, it says, oh, there was no negative weight cycle, and here are your shortest path weights. You just plug them in, and bam, you have your x_i's that satisfy the constraints. Awesome. Now, it wasn't just any graph.

I mean, we started with constraints, algebra, we converted it into a graph by this transform. Then we added a source vertex, s. So, I mean, we had to build a graph to solve our problem, very powerful idea. Cool. This is the idea of reduction. You can reduce the problem you want to solve into some problem you know how to solve. You know how to solve shortest paths when there are no negative weight cycles, or find out that there is a negative weight cycle by Bellman-Ford.

So, now we know how to solve difference constraints. It turns out you can do even more. Bellman-Ford does a little bit more than just solve these constraints. But first let me write down what I've been jumping up and down about. The corollary is you can use Bellman-Ford. I mean, you make this graph. Then you apply Bellman-Ford, and it will solve your system of difference constraints. So, let me put in some numbers here. You have m difference constraints. And, you have n variables. And, it will solve them in order m times n time.

Actually, these numbers go up slightly because we are adding n edges, and we're adding one vertex, but assuming all of these numbers are nontrivial, m is at least n. It's order MN time. OK, trying to avoid cases where some of them are close to zero. Good. So, some other facts, that's what I just said. And we'll leave these as exercises because they're not too essential. The main thing we need is this. But, some other cool facts is that Bellman-Ford actually optimizes some objective functions. So, we are saying it's just a feasibility problem. We just want to know whether these constraints are satisfiable. In fact, you can add a particular objective function. So, you can't give it an arbitrary objective function, but here's one of interest.

x_1 plus x_2 plus x_n, OK, but not just that. We have some constraints. OK, this is a linear program. I want to maximize the sum of the x_i's subject to all the x_i's being nonpositive and the difference constraints. So, this we had before. This is fine. We

noticed at some point you could get from s to everywhere with cost, at most, zero. So, we know that in this assignment all of the x_i's are negative. That's not necessary, but it's true when you run Bellman-Ford. So if you solve your system using Bellman-Ford, which is no less general than anything else, you happen to get nonpositive x_i's. And so, subject to that constraint, it actually makes them is close to zero as possible in the L1 norm.

In the sum of these values, it tries to make the sum as close to zero, it tries to make the values as small as possible in absolute value in this sense. OK, it does more than that. It cooks, it cleans, it finds shortest paths. It also minimizes the spread, the maximum over all i of x_i minus the minimum over all i of x_i. So, I mean, if you have your real line, and here are the x_i's wherever they are. It minimizes this distance. And zero is somewhere over here. So, it tries to make the x_i's as compact as possible. This is actually the L infinity norm, if you know stuff about norms from your linear algebra class. OK, this is the L1 norm. I think it minimizes every LP norm.

Good, so let's use this for something. Yeah, let's solve a real problem, and then we'll be done for today. Next class we'll see the really cool stuff, the really cool application of all of this. For now, and we'll see a cool but relatively simple application, which is VLSI layout. We talked a little bit about VLSI way back and divide and conquer. You have a bunch of chips, or you want to arrange them, and minimize some objectives. So, here's a particular, tons of problems that come out of VLSI layout. Here's one of them. You have a bunch of features of an integrated circuit.

You want to somehow arrange them on your circuit without putting any two of them too close to each other. You have some minimum separation like at least they should not get top of each other. Probably, you also need some separation to put wires in between, and so on, so, without putting any two features too close together. OK, so just to give you an idea, so I have some objects and I'm going to be a little bit vague about how this works. You have some features. This is stuff, some chips, whatever. We don't really care what their shapes look like. I just want to be able to move them around so that the gap at any point, so let me just think about this gap. This gap should be at least some delta. Or, I don't want to use delta. Let's say epsilon, good, small number.

So, I just need some separation between all of my parts. And for this problem, I'm going to be pretty simple, just say that the parts are only allowed to slide horizontally. So, it's a one-dimensional problem. These objects are in 2-d, or whatever, but I can only slide them an x coordinate. So, to model that, I'm going to look at the left edge of every part and say, well, these two left edges should be at least some separation.

So, I think of it as whatever the distance is plus some epsilon. But, you know, if you have some funky 2-d shapes you have to compute, well, this is a little bit too close because these come into alignment. But, there's some constraint, well, for any two pieces, I could figure out how close they can get. They should get no closer. So, I'm going to call this x_1. I'll call this x_2. So, we have some constraint like x_2 minus x_1 is at least d plus epsilon, or whatever you compute that weight to be.

OK, so for every pair of pieces, I can do this, compute some constraint on how far apart they have to be. And, now I'd like to assign these x coordinates. Right now, I'm assuming they're just variables. I want to slide these pieces around horizontally in order to compactify them as much as possible so they fit in the smallest chip that I

can make because it costs money, and time, and everything, and power, everything. You always want your chip small.

So, Bellman-Ford does that. All right, so Bellman-Ford solves these constraints because it's just a bunch of difference constraints. And we know that they are solvable because you could spread all the pieces out arbitrarily far. And, it minimizes the spread, minimizes the size of the chip I need, a max of x_i minus the min of x_i. So, this is it maximizes compactness, or minimizes size of the chip. OK, this is a one-dimensional problem, so it may seem a little artificial, but the two dimensional problem is really hard to solve. And this is, in fact, the best you can do with a nice polynomial time algorithm.

There are other applications if you're scheduling events in, like, a multimedia environment, and you want to guarantee that this audio plays at least two seconds after this video, but then there are things that are playing at the same time, and they have to be within some gap of each other, so, lots of papers about using Bellman-Ford, solve difference constraints to enable multimedia environments. OK, so there you go. And next class we'll see more applications of Bellman-Ford to all pairs shortest paths. Questions? Great.