The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high-quality educational resources for free. To make a donation, or to view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR: All right, welcome back. Today, we start a series of lectures on approximation algorithms. So something rather different, although we're still going to be doing reductions. They're just going to be stronger sense of reductions. So let me start with reminding you-- I assume you've seen approximation algorithms at some level before, but I want to define a few things which we'll need for the lower bound side.

So pretty much throughout, we've been-- in the context of NP, we've been thinking about decision problems, because that's where NP make sense. Now, we need to change the setup to where the output is not just yes or no, but it's some kind of solution with a cost. So in general, the goal in an optimization problem is going to be to go from some instance to a solution with min or max cost. So there's minimization problems, maximization problems.

And so you are given-- in order to specify this, you're given a set of possible instances. That's your usual notion of input. And then for each instance, we are given a set of solutions. Usually, these are called feasible solutions or valid solutions. I'll just call them solutions. Those are the allowable outputs to that problem.

And then for each of those solutions, we're going to define some cost. I will assume that they are non-negative, so I don't have to worry about signs. And then you're also given what's usually called the objective, which is either min or max. OK, so you're going to be given an item from this set. You want to produce an item from this set that minimizes or maximizes this cost. That's not going to be possible, because all these problems are going to be NP complete. But the point of approximation is to relax getting the best solution, and aim to get an approximately best solution, which we will define in a moment.

But let me start with some useful notation, which is opt of x, x is an instance, this is going to be two things. No definition should be unambiguous. So it's going to be the cost of a min or a max solution. Cost solution. And it's also going to be such a solution itself, sometimes. Cool.

So this is sort of the usual setup. Now let's make it a little bit more interesting by defining an NP optimization problem. This is going to be the analog of NP for optimization. So one thing is that all solutions have polynomial length, polynomial in the length of the input, the instance. Another thing is that instances and solutions can be recognized.

So there's a polynomial time algorithm that tells you yes, that's a solution, yes, that's an instance, or no. And let's see, the cost function should be polynomially-computable. And that's it. OK, so just making these actually reasonable from an algorithmic standpoint. And so now, you know, thinking like NP, the solution is going to be something like the certificate that you had a yes answer. Here, it's going to be a certificate that the cost is a particular thing.

So given an empty optimization problem, you can define a decision problem, or decision version of NPO. So NPO is going to be the class of all such problems, NP optimization problems. Just like NP was all the problems solvable in nondeterministic polynomial time. These things I claim, if we convert to the decision version, which is for a min problem, the question is is opt of x, the cost, less than or equal to some query value. And for a max version, it is is opt of x greater than or equal to the query value.

This thing is an NP. So let's see why. If you have these properties, then the resulting decision problem will be an NP because if I give you a solution, and solutions have polynomial length, you can compute its cost, verify that it was a valid solution, and then you know that opt-- opt is some min, so it's going to be less than or equal to any particular solution.

So if I give you a solution, then I verify that opt of x is less than or equal to q in

polynomial time. For max, it's the reverse. And for example, if you wanted to check-- if you wanted to know whether the min was greater than or equal to something, well that's a co NP problem. So that's-- these are the correct decision versions of those NP optimization problems in the sense that you get a problem in NP.

So this is a strict generalization, in some sense, of NP problems, or specialization I suppose. This is a particular-- well, we have optimization versions which still give us NP problems. We can still talk about NP completeness of these problems. But we're interested-- we're setting all this up so we can actually talk about costs of not just the best solution, but costs of suboptimal solutions. So let's get to approximation.

So we're going to call an algorithm, ALG, a c approximation. I write c, but it's not necessarily a constant. It could be a function of N. If-- in the min case, what we want is the cost of the algorithm applied to an input, x, divided by the cost of the optimal of x. Here, for whatever reason, I write cost of opt instead of just opt. That should be less than or equal to c. We look at the ratio, we want that to be bounded by some value.

In the max case, there are two definitions that are standard in the literature that correspond to different ways of looking at the same thing. One would be cost of OPT divided by cost of ALG. That should be, at most, c. This would correspond to c being greater than or equal to 1. And the alternative is to flip the inequality instead of the ratio.

First, let's think about min. With a minimization problem, whatever the algorithm produces will be greater than or equal to OPT. So this is a ratio that will give something greater than or equal to 1. And so usually we think about two approximation, 1.5 approximation, 100 approximation, whatever, just saying the algorithm is within a factor of 100 of the optimal.

For maximization, the thing that the algorithm produces will be less than or equal to OPT. And so if you want something that is greater than 1, you have to flip the ratio. In this situation, c would be less than or equal to 1. So some people call-- if you have an algorithm that produces a solution that is at least 1/2 times optimal, you

might call it a two approximation or you might call it a 1/2 approximation, depending on the paper. I think this is, by now, more common, but it really depends on the era and the person and so on. So good to be aware of both. It's measuring the same thing, of course, just c is either bigger than 1 or less than 1. Yeah?

AUDIENCE: Is that over all instances?

PROFESSOR: Oh yeah, this should be for all x. For all valid instances x. Good. So usually when we say a c approximation algorithm, usually we're interested in polynomial time c approximation algorithm. Almost all the time that is the case. I will probably forget to say polynomial time, but I always mean it unless I say otherwise, which will probably be never. But there are a few papers that look at exponential time approximation algorithms that have better exponent than the exact counterparts, and so on.

So this is interesting because while your-- the straight-up decision problem, which is basically deciding OPT exactly, that might be NP complete. The approximation version, like finding a two approximate solution, might be polynomial. And that would be observed by finding a polynomial to approximation algorithm. Sometimes you can do even better than con-- well.

So here, we've been thinking about constant factors for c. Sometimes you can do even better than any constant factor, or you can achieve any constant factor would be another way of saying it. This is called a polynomial time approximation scheme, usually PTAS. You can think of this as a 1 plus epsilon approximation for any epsilon. But the general-- or you can think of it as an algorithm that, given epsilon, produces a 1 plus epsilon approximation algorithm. Or you can think of it as an algorithm with an additional input, not just the instance, but also a value epsilon. It's a rational epsilon greater than zero. And then the-- let's say produces a solution that's a 1 plus epsilon approximation.

So this would be a sort of ideal situation. You get to specify what the error bound is, and the algorithm will find a suitable solution. And the polynomial time part is that this algorithm must be polynomial time for any fixed epsilon, which means that the dependence on epsilon could be horrible. You could have an algorithm, if N is your

input size, that runs in something like N to the 2 to the 2 to the 1 over epsilon or whatever. That's polynomial time for any fixed value of epsilon. It's rather large for any reasonable value of epsilon, like a half, but anyway. Or even 1 is still getting up there.

But that's considered a P test. Now there are stronger notions of P tests that prevent this kind of thing. We will get to that when we get to fixed parameter tractability. But for now, this would be considered the gold standard. This is the best you could hope for at this level of detail. And for-- let me give you some more classes.

So we defined NPO. There's the class PTAS. Again, reuse of term, but this is all problems-- NPO problems with a PTAS algorithm. And more generally, if we have some class of functions, then I'm going to write FAPX to be all the NPO problems with poly-time f approximation algorithms. I'll write f of N to be clear what we're depending on for some little f in the class big F.

So for example, f of N could be 3. That would be a constant factor approximation. In that case, we just call it APX. APX is what you might otherwise call order one APX, things that you can approximate in some constant factor, any constant factor. Another class that's commonly studied is log APX. This is log N approximable some constant factor. And there are more. There's poly APX, where you just want N to some constant, usually less than one but not always. But I think this will be enough for us.

And so we're interested in distinguishing these classes. We're going to do that by defining reductions, using reductions, defining hardness, and then getting the hardest problems in certains of these classes to show that's the best kind of approximability you can get. Today, we'll be thinking in particular about the boundary between PTAS and APX. So PTAS, we can get 1 plus epsilon for any epsilon. APX, you can get a constant factor, but there's some limit to how small that constant factor could be, at least in the hardest case. APX includes PTAS, but there's problems in APX minus PTAS. Those have a limit how far down you can get.

OK. We have PTAS is a subset of APX. And furthermore, if P is not equal NP, it's a strict subset. And let's say log APX or whatever. You pick your favorite approximation factor, and there are problems where you can achieve that and nothing better. It's actually an easy exercise to come up with such a thing. This is if P does not equal NP.

So you can take, I don't know, Hamiltonian cycle or something, or pick your favorite NP hard problem. And if the answer is yes to that problem, then you construct a solution with some cost. And if the answer is no, it's way, way smaller. And so any approximation within your desired factor will be hard to find.

Let me get a little bit more specific, and let's talk about some real problems. So in the world of graph algorithm approximability, these are the typical kinds of approximation factors you see. Of course, it depends exactly what subdomain. This is not a complete list, but it starts to give you a flavor. And today, we'll be thinking mostly at the top level. But let me define some of these problems for you.

A lot of them we have seen, or a few of them we have seen, such as Steiner tree. We talked about rectilinear Steiner tree. That was a Euclidean problem. You were given points in the plane, and you wanted to connect them up by the shortest connected network. In a graph-- so that problem has a PTAS rectilinear Steiner tree, because it's a Euclidean problem. A lot of Euclidean problems have PTAS's. And I'm denoting PTAS here by 1 plus epsilon.

In general, epsilon here means for all epsilon greater than zero. Steiner tree in a graph is I give you a graph and you have a bunch of special vertices, k special vertices. You want to find a connected subgraph of that graph that hits all of the special vertices. That problem has a constant factor approximation, and there's no PTAS for a general graph.

Steiner forest is almost the same. Instead of giving vertices that all have to connect to each other, you say which pairs of vertices need to connect to each other. This guy and this guy, and this guy and this guy. You can still-- this is a generalization. In the case where you want to connect them in a [? Kleek ?] pattern, that's Steiner

tree. But Steiner forest has the same approximability up to constant factors.

Traveling salesman in a graph, you've probably seen a 1.5 approximation to that. Definitely two approximation is really easy, like MST. And that's-- for in a graph, that's the best you can do in 2D. Or in a planar graph or in an H-minor free graph, there's a PTAS. I think H-minor free, traveling salesman problem weighted was solved just a couple years ago. But that has a PTAS.

Let's see. Usually most people like to think about minimization problems, but there are maximization problems, too. Let's see, what else haven't I defined. Did we talk about set cover? You have-- I think we did briefly. You have sets and you have elements. You want to choose the fewest sets that hits all elements. Each set contains some of the elements. You can think of it as a bipartite graph, sets on one side, elements on the other. You want to choose the fewest vertices on the left that hit all the vertices on the right.

Dominating set is the non-bipartite version of that problem. You're just given a graph. And if you choose a vertex, it covers that vertex and its neighboring vertices. And your goal is to cover all vertices using the smallest dominating set, by choosing the fewest vertices. So these problems turn out to be equivalent from an approximability standpoint. And in a stronger sense, they're both log N approximable, and that's the best you can do. This is assuming p does not equal NP.

Some of these results assume slightly stronger things than p versus NP, but we won't worry about that too much. Let's see. Another fun problem-- I'm just highlighting the ones you should know about. Chromatic number, we've seen what we thought about three coloring. But the chromatic number problem is to minimize the number of colors, k, such that your graph is k-colorable.

And that's really hard to approximate. The best approximation algorithm, I think, is N divided by log N, or something roughly N. That's not so good. And there's a lower bound that you can't do better than N to the 1 minus epsilon for all epsilon. So you really-- that's pretty tight. Not completely tight, but pretty close. And let's see.

Other good problems. So maybe out here, another really hard problem to approximate-- these are problems you should try to avoid if you're trying to solve things approximately. Or if you're driving hardness, you'd want to use these if you can, if your problem looks like this. Independence set, just find me a set of vertices that induce no edges. So no edges connecting them. That's really hard.

If you complement the graph everywhere there's an edge deleted and everywhere there wasn't an edge added, that's the same problem as Kleek. So these are the same also from an approximation standpoint. OK. This is annoying because [? Kleeks ?] are useful in practice, but. There's another problem called [? densest ?] subgraph, which is approximate [? Kleeks. ?] That's actually easier to approximate. Yeah.

AUDIENCE: What's the hardness of something for the chromatic number lower bound?

PROFESSOR: I think that's like P does not equal ZPP, or NP does not equal ZPP, I think. Yeah. For that-- that's strong lower bounds. There are weaker lower bounds assuming P does not equal NP. OK. One other good one, fun one, to think about is set cover. You want to choose the fewest sets to hit all elements. Maximum coverage, you don't have to hit all the elements but you want to hit as many elements as possible using k sets.

So in the decision problem, these are the same in some sense. Here, I give you k sets. I want to know can I hit all the elements. For example, here I'm giving k sets and I want to know can I hit at least J of the elements. So this you could think is a generalization from a decision standpoint, but it's actually easier to approximate because the objective is different.

Here we have to hit every element, no questions asked. Here, it's OK if we miss some of the elements because we only need a constant factor approximation. And we can get a constant factor here, whereas the best we can do is log N here. Cool. But unique coverage, this is a result of ours. If you're trying to maximize coverage but if you double cover an element, you no longer get points for it. That requires log

N approximation. So I think I will leave it at that.

Directed graphs are much harder, although it's not known exactly how hard. Here's an example of a big gap in what we between log squared and N to the epsilon. Something we will get to in a later class is called label cover. I won't define the problem here, because it takes a while to define. But there, there's very strong lower bounds on-- in approximability. It's similar to N to the 1 minus epsilon, but it's not. It's a little smaller.

So you see 2 to the log N would be N. That would be great. And so you'd really like 2 to the log N to the 1 minus epsilon. But what we have here is something a little smaller than log N, log N to the 1 minus epsilon power. And then you exponentiate that. So this is somewhat smaller than what we'd actually like, which is something like N to the 1 minus-- or N to some epsilon, I guess, would be ideal. This is smaller than any N to the epsilon.

The best upper bounds, it depends on the problem. With N to some constant, like for label cover, it's N to the 1/3. For directed Steiner forest, which is you have pairs of vertices you want to connect, but with a directed path. So similar to a problem we've seen. Best approximation known for that is N to the 4/5 plus epsilon. And this is just a couple years ago.

So that seems very hard to do better than any constant. Probably there's an N to the epsilon lower bound, but this is the best we know so far, a bit smaller. Cool. Any questions about that table? We will probably come back to it a few times, but in more depth.

So how do you prove the lower bound side? We're going to use reductions. Now this field is a little bit messier in terms of reductions. For NP completeness, NP hardness, we just worried about [? carp ?] style, one call reductions. That's all we had to think about. In this universe, there are 1, 2, 3, 4, 5, 6, 7, 8, 9-- at least nine definitions of reduction.

You don't need to know them all. I will define four of them, I think. They're all very

similar, and they all lead to slightly different notions of hardness. So you have to be a little careful when you say oh, this problem is something hard. But I'll give you some definitions that are pretty easy to work with, I think.

In general, this family is called approximation-preserving reductions. So let's say we want to go from some problem A to some problem B. So you're given an instance of A, let's call it x, and you want to produce an instance of B. Let's call it x prime. And we're going to do that by a function f, so x prime is f of x. So far, just like NP reductions.

But now-- usually what we said is that the answer to x equals the answer to x prime. So you could say OPT of x equals the OPT of x prime. But that's not going to be strong enough. What we want is that, if we can find a solution to this B problem-- let's call it y-prime to the problem x-prime. So the x-prime is an instance of B. You can find some solution-- any solution y-prime-- to x-prime, then we want to be able to convert it back by a function, g, into a solution to A, which we'll call y.

And so it's g-- you might think it's g of y-prime, but we'll make it g of x comma y prime. So it can also depend on what you started with. So this is the general flavor.

A reduction consists of two steps. First, like before, we convert instances of A to instances of B. Now in addition, we want to be able to recover solutions of B into similarly good solutions to A. And there's many ways to define similarly good. Let me give you one.

So if we're just interested in PTAS's, then-- like you want to know does my problem have a PTAS or I want to prove impossibility of PTAS's, then I think there's one clear notion of reduction, which is obviously enough called PTAS reduction. So what we want to say is-- I hope you've seen calculus at some point. In calculus, there's this notion of epsilon delta proofs, or definitions. So I'm going to say-- you don't have to think of it this way, but I find it useful to think of it this way.

Let me get to what the statement is. If y-prime is a 1 plus delta approximation to B, then y is a 1 plus epsilon approximation to A. So ultimately we're interested in

PTAS's, and we want to say that if you have a PTAS for B, then you get a PTAS for A.

It used to be, when we did a reduction from A to B, that showed that if B had a polynomial time algorithm, then so did A, because you just plug in these chains together. You convert A into the B instance, you run your poly algorithm and get a solution, and convert it back. Well, with NP, we didn't even have to convert. The answer was the same. Now we could do that.

We want to do the same thing with PTAS's. So if we have a PTAS for B, we can convert A into B, run the PTAS, get a solution, and come back and get-- what we want is for it to be a 1 plus epsilon approximation to A. So that's the goal, 1 plus epsilon approximation. And what this is saying is there's some function delta of epsilon-- for any epsilon, there's some value delta where you can give that value to the PTAS for B, and it will give you what you want for A.

Because PTAS is supposed to run for any fixed constant-- we called it epsilon, but it could also be delta. And so you plug in this to the PTAS. You will get a 1 plus delta approximation to your B problem. And then what we want is that this conversion-- so y here is g of xy-prime. So we want g-- this is a constraint on g. We want g to have the property that if y-prime is that good, then y will still be as good as we need.

So maybe you just-- you want to get a 1.1 approximation. You want to get within 10% of the right answer. Maybe you have to call B with a much smaller thing. Maybe delta is 0.01. You need to be within 1% of the right answer. But there's-- for a PTAS reduction, you want there to be some approximability you asked for so that you get what you actually want. This is sort of the obvious-- this the natural thing if you want to preserve PTAS-ness in this direction, from B to A.

One slight note is that we also allow f and g to depend on epsilon. So this reduction-- it would be hard to get this property for all epsilon with the same instance conversion. So now we allow the conversion of the instances to depend on epsilon, so you can do fun things that way for the purposes of PTAS reductions. But the key thing we get is that if B is a PTAS, then-- or has a PTAS-- then A as a PTAS, so it's

in the complexity class PTAS.

Of course, what we care about is the contrapositive of that. So if A is not in PTAS, then B is not in PTAS. So we can use this-- we reduced from our hard problem that we know is not in PTAS, assuming P does not equal NP, and we get that our new problem B is not in PTAS. That's the point.

This is also true for something like APX. So if you just want constant factor approximation, this will convert one constant factor into another constant factor. So yeah, if you have-- I mean, it's maybe stronger than what you need, but will give you constant factor approximations, as well.

So here's some easy more definitions. Oh, OK, so a few more fun facts. If this statement also holds true for epsilon equals 0 when you plug in epsilon equals zero, delta equals zero, then you're just getting the regular notion of reduction from NP land. This is saying the OPTs are equal. So usually your solution also works in that setting. That's usually the easy thing to do.

So this is a strictly stronger notion of NP reduction if you allow the 0,0 case. Also, reductions change. So if A reduces to B and B reduces to C, then A reduces to C. All the usual things you'd expect.

Some special cases of interest are AP reduction. This is when the delta function is linear in epsilon. So this is nice because it tells you, for example, if B is in-- has some order f approximation, then A does, as well. It just changes this constant factor. So this is useful if you care about log N approximation. If you use this definition, things could change, because you're allowed to change what the factor is out here in crazy ways.

So over here, [? before-- ?] so [? it's ?] just a linear blow up in the approximation factor, then we only lose constant factors in approximability. And one more is strict reduction. This is when there's no blowup. If you have a c approximation here, you get a c approximation there. That's nice when you can get it. We're not going to aim for this in particular, but there are a bunch of reductions we'll talk about today that

are strict. So it's nice to just have a word to say oh, this is even strict. You don't even need any [? blowup ?] like this stuff. So that was strict AP and PTAS reductions.

There's one more, but maybe first let me define hardness. So today, we will focus on APX hardness. And these are supposed to be the hardest problems in APX. And so, I mean, because we're trying to distinguish between PTAS-able problems and just constant factor approximable problems, we are interested in PTAS reductions.

So it's going to turn out that APX hard-- well I guess we sort of already know. This implies you're not in PTAS if P equals NP-- P does not equal NP-- because of this strict containment, P does not equal NP, then PTAS is different from APX. And so if you show your hardest problem in APX, using a reduction that preserves PTAS-ability, then you know-- I mean, the idea is that-- all right.

When we're doing these reductions, we know that if B had a PTAS then A had a PTAS. And what we're saying is you can re-- your APX hard means you can reduce from any problem in APX to your problem. So if your problem had a PTAS, that would mean all problems had PTAS's. All APX problems had PTAS's. But we know that's not the case. Therefore, your problem does not have a PTAS. So it's just like MP completeness, just with different letters. Cool. And different reductions.

These reductions are all a little bit awkward to work with directly for-- or these definitions are a little awkward to check directly, although ultimately it's the same thing. In practice, people seem to use a different notion of reduction, for the most part, called L reductions. That's stronger than AP reduction, not as strong as strict. I guess it's not directly related either way to strict, but, it's OK.

So while I define this in terms of PTAS reduction, because that's what you need to get this result, most people think about L reduction, which is going to imply-- it's going to be stronger than the other two notions of reduction, AP and PTAS. So what's the definition? We want to satisfy two properties. The first property is a kind of blowup, going left to right.

Funnily enough, the other reductions do not have this property, because they don't quite need it in the way that they're going. But we want that the transform problem, its optimal solution is not much bigger. It's bigger by only a constant factor. This is, like, less than or equal to alpha times the optimal-- the original optimal solution given instance, so A. Second property-- and we'll see why we need this in a moment-- is if you look at the absolute error instead of the ratio between the computed solution y-- and y was the thing that we got in the lower left corner, we produced the solution y-- we look at the cost that we get out of-- y is the solution. So we look at the cost of that solution, we compare it to the optimal solution to our original instance-- because it is a solution to that instance. That should not be much bigger than the error on the right side. Again, absolute error.

So we want this to be big O of cost in B's problem of y-prime minus OPT of B of x prime. So these are both in B land. Again, we look at the optimal solution to the produced instance versus some solution that we were given. We didn't produce that solution, so this has to hold no matter what solution y-prime you're given to the instance x-prime, when you convert it using g to a solution to x in problem A, you want that the gap-- absolute gap-- is not stretched by more than a constant factor.

Again, this is at most some constant times [? that ?] [? thing ?]. OK? Now it should not be obvious that this implies this, but it does, with delta equal to, I think-- get some color. Let's say the constant in here is alpha, the constant here is beta. And this should be alpha times beta times epsilon. That is delta of epsilon.

That's the claim. Let's prove the claim, ideally without cheating. So I claim that if I have such an L reduction, that this holds where-- when delta is that linear function of epsilon. So I want to conclude that y is a pretty good approximation. So to do that, I will look at the cost of y divided by OPT. OPT of X would be the original input.

So we want to show that this is, at most, 1 plus epsilon. Now what do we know? I should write A, I guess, because this is all about problem A. What do we know? We know this property-- I wrote absolute value here, so this works for maximization and minimization. You can even convert minimization to maximization problems using

this definition. We will do that at some point.

OK. So let's just try to plug this into here. So I think I need to split into cases, and I'll just worry about the minimization case. So let's say that cost of A is bigger than OPT. So then cost of A is going to be OPT plus this thing, at most. So this is going to be at most OPT of x plus beta-- I'm going to not use the big O notation, and just write the beta thing. Beta times-- I guess there's no absolute value needed, because this is a minimization problem. Let's say-- so this will be cos sub B of y-prime minus OPT of B of x-prime. So that's what I get from expanding out the numerator, and the denominator is the same. Clear? Just rearranging terms here, essentially.

OK. Now these two terms cancel, so we get 1 plus that. That's good, because I wanted that to be epsilon. Let's see. What else do we know. It's 1 plus this thing. Let me scroll down.

OK, we have one other property. We've got to use it. The other property is that OPT sub B is related to OPT sub A. Now we have OPT sub A here, but it's in the denominator, which flips the relation. So we can write that relation up there as OPT sub A of x is omega of OPT sub B of x-prime. That's the same as statement one.

And because the denominator is omega, that means the whole thing is big L. So this is going to be, at most, one plus-- so now we have this thing. So the numerator is the same. We had before beta times cost sub B of y-prime minus OPT sub B of x-prime. So that's unchanged.

But now, instead of dividing by OPT sub A, we're going to divide by OPT sub B of x-prime. And we lost a constant factor, that constant factor translated to B alpha in the numerator because we inverted the equation. We divided by the constant factor there. So what-- well now this cancels with that.

So this is going to be 1 plus alpha beta times cost sub B y-prime over OPT sub B of x-prime minus 1. Still looks kind of weird. But what we-- there's one more thing we didn't use, which is we assumed the-- we were given some solution y-prime that

was a 1 plus delta of epsilon approximation, where delta is this thing that we defined there. It's obviously set to cancel out everything that happened here.

So this thing should be, at most, 1 plus delta, because this is exactly the approximation ratio for y-prime versus the optimal solution to x-prime. And 1 plus delta is 1 plus alpha beta epsilon. So this 1 cancels with that 1, these alpha betas-- whoops.

AUDIENCE: I think it's backwards.

PROFESSOR: Which is backwards? The definition of delta? OK. Whoops. Let's try epsilon over alpha beta. That makes sense, because delta should be smaller than epsilon probably, and alpha and beta are probably bigger than 1.

So now we get epsilon over alpha beta here. And then that alpha beta cancels with that alpha beta, and we are left with 1 plus epsilon. OK. So this is why it's not obvious, but it's just plugging in all the things and it works. And the funny thing is, somehow L reductions, though they-- a little bit less natural, I feel like. I mean, if you're thinking about constant factor approximation, why should you care about absolute error?

The short answer that I've seen written in various papers is because it's easier to think that way for a lot of the typical reductions that you do. So let us do some of those reductions. And we'll get some intuition.

L reductions also-- they do work in the zero case. If you have an optimal solution to x-prime, you will get an optimal solution to x. So they are also NP reductions. Again, that's a generalization of-- or strengthening of the type of reduction we saw in all previous lectures.

All right. So I want to simultaneously tell you a bunch of problems that are APX complete so that you know things to reduce from, but also I'll show you examples of such things. So some of these I will omit the proofs, because they're messy and it's just useful to know that they're there so you can reduce from them. Others I-- most of them, I will cover the proofs.

All right. And we're going to return to an old issue from the first SAT lecture. And I'm going to introduce some new notation. So a lot of the starting points here are just max SAT. Or I should-- let's say a max CNF SAT. You're given a CNF formula. It's picked out a bunch of clauses. You want to maximize the number of clauses that you satisfy with some variable assignment. So that's going to be APX complete. There are constant factor approximations if the clauses are constant size, I should say. So like max 3SAT has a constant factor approximation, I forget what the current best is. And no better. There's no PTAS for a max 3SAT.

This is a stronger form of max 3SAT, which we have not seen before, though we've hinted around it. The E means every clause has exactly three distinct literals. This is an issue that we stumbled into. Oh, do we allow clauses to have only two literals? I said no, in the original definition. But I did allow repeated literals, which is the same thing. So when I say three set, I mean you can repeat literals, or you can have fewer than three literals.

When I say E three set, then you're not allowed to do that. Then you may remember we talked about 3SAT five. That meant every clause has only three literals, and every variable appears, at most, five times. E5 means exactly five times. Each variable appears exactly five times. Do you have a question?

AUDIENCE: [INAUDIBLE] gives you a condition on the number of clauses.

PROFESSOR: Oh, you're right. That gives you a linear relation between the number of clauses and the number of variables. You can work it out. This is hard. I will not prove it. It's not hard, but it's just a little bit messy. Sorry. It's not difficult.

I will prove a slightly different result, which this one is based on, which is a little bit more familiar but also introduces something new we hadn't seen before. This is the regular 3SAT. Each clause has, at most, three literals. And every variable appears, at most, three times.

This is something I didn't realize was hard. It's not written in most places. But it is here, in the world of approximability. This is the reduction. It's kind of funny.

So suppose you have a variable, x, which appears a million times. You're going to make a cycle, so to speak. It's a formula, so it's not really a cycle, but of size 1 million. And you're going to write down this 2SAT constraint-- not xi or xi plus 1 for all i. And do it around in the cycle, so not x6 or x1-- or x a million or x1. That's, of course, equivalent to saying if xi is set to true, then xi plus 1 must also be set to true. So if any of these are set to true, they all are. So the only satisfying assignments here are everybody true, everybody not true.

So if you're just worried about NP reductions, this is a reduction from 3SAT to 3SAT-3. Because every variable now will appear in exactly three-- exactly three, in fact-- so constraints. This one, this one, and whatever it originally-- whatever you want to plug it into. So each of these was an occurrence of that variable. Maybe use the positive form, maybe use the negative form, but each variable appears three times in this first picture.

Now that's a great reduction for 3SAT-3. Notice I didn't write max, because this reduction will not work as an L reduction, say. You cannot prove that max 3SAT is hard using this reduction, because you could, for example, just violate these two constraints and make all of these guys true and all of these guys false. And if this is size a million, that means you're saying-- I mean, you're setting the variable half true, half false-- or you could use any ratio you want-- at a very small cost.

You're only paying a penalty of two constraints violated, and yet you're able to satisfy 1/2 of the clauses using xi equal to true, and some other set of clauses using xi set to false. And there's no way to bound how many clauses you can sort of misrepresent. And the objective in max 3SAT is to maximize the number of clauses satisfied. So if you allow this kind of thing where you can flip in crazy ways and change a huge number of clauses one way or the other in this sort of false way, how would-- if you're trying to actually construct a solution to max 3SAT from a solution to this 3SAT-3 instance, you wouldn't know which way to set the variable x to.

If you set it to true, there's a whole bunch of clauses that wanted it to be false. If you

set it to false, a whole bunch you wanted to be true. Yeah.

AUDIENCE: If we're just talking about NP hardness here, can't you make those edges very heavy by repeating the clause many times?

PROFESSOR: If we're just worried about NP hardness, this is fine. Because then all of these have to be true.

AUDIENCE: Uh, I mean for the max decision problem.

PROFESSOR: Well then the decision problem is, can you satisfy all of them. That is a special case of the max--

AUDIENCE: Satisfy k of--

PROFESSOR: Yeah, but if you set k equal to N, that's a special case of the general form.

AUDIENCE: Oh, OK.

PROFESSOR: So--

AUDIENCE: I see what you're saying.

PROFESSOR: Yeah. If you didn't-- if you wanted to make k and N different, you could just add a bunch of things that are unsatisfiable. Lots of ways to change how many there are. OK.

Nonetheless, we can prove max 3SAT-3 is hard with an L reduction from 3SAT. So I'm not going to prove that 3SAT is APX hard. We might do that in a future lecture, but just take that as given. What I want to show you is how to convert max 3SAT into max 3SAT-3. Not like this, but using a different trick.

So reducing from max 3SAT. We're given a formula, we're given some variables. Let's say-- let's look at variable x. And let's say it appears k times. What we're going to do, just like before, we're going to make k new variables. We're going to make k variables x1 through xk that replace x, and we're going to use those instead of x.

And I want to force all the values of x to be the same, but I want to force it even when you're allowed to cheat and set some of the things incorrectly. And we're going to do this using a powerful tool, which is good to know about, called expander graphs. So there's two things to tell you. One is, what is an expander graph, and the other thing is, once I have an expander graph, what do I do. Let me start with the latter, because it's a little simpler.

I guess a lot of you have heard about expander graphs. And the short answer is they're complicated and confusing, but really cool and powerful. We're not going to try to prove that expanders exist here. I'm just going to tell you they exist, and that's actually pretty simple what they-- what properties they have.

So what I want to do, whenever I have an edge in this graph-- this is a graph whose vertices are the x1 through xk-- I'm going to convert an edge into a constraint, which is effectively xi equals xj, which in reality is not xi or xj, and not xj or xi. So I really probably shouldn't think of it as xi equals xj. That's what I want to do. I want to force lots of things to be equal.

But really, we have to, in the end, think about it in terms of constraints, because some of them might be violated. So what is an expander graph? There are different types. But what we will use is two properties. So this is for k nodes. We have bounded degree, and we have, for every cut AB-- and I think we've talked about cuts in this context, in the context of max cut. So the idea is that A and B are disjoint, and their union is the set of all vertices.

So A is 1 side of the cut, B is the other side of the cut. We want the number of cross edges, the number of edges between A and B, is big. It's at least the min of A and B.

Some intuition. Imagine a Kleek. A [? Kleek ?] is not bounded degree, but it has this property. If you look at any cut, there's a lot of edges between them. It's actually more like the product. But in particular, it's at least the min.

So you can think of the expander as a sparse [? Kleek. ?] It's [? Kleek-y ?] enough,

in this sense, which we'll see why this is the property we want. But it has bounded degree, therefore linear number of edges. So it's very sparse. In the construction we are applying, which is due to [INAUDIBLE] Philips and [? Sarnak, ?] the degree is 14. But it doesn't matter. Constant. It's actually 14 regular, so that's nice.

So we take this graph, which was known to be out there-- basically random graphs [INAUDIBLE] property, but we won't worry about how to construct it too much, although you do have to. But there's tons of papers on how to construct them. And then for every edge, we convert it into these two constraints. So let's prove that this is an L reduction.

Claim L reduction. So maybe-- I don't have anything to point at, because I can't point at an expander graph. But let me draw the graph so I can point at something. Let's draw the [? Kleek, ?] just so it's a little easier to think about. This is an expander, [? k4 ?]. And let's say, in your solution, some of these end up getting assigned true value-- let's represent that by red-- some of them not. Maybe three of them are true, one of them is false or whatever.

In general, I'm going to choose-- so I mean, we're given some solution, right? This is the solution y-prime to the constructed instance x-prime. X-prime has this expander. So we look at a solution y-prime to x-prime, we have no control over what it is. If you look at a variable, some fraction of them are set to true, some of them are set to false. We're going to choose the majority. So here it's majority red, so we're going to change this guy to be red.

Now does that hurt us? Well, we can think of there as being a cut of the red nodes versus the not red nodes, the black nodes. And I claim the number of edges here is big. It's at least the minimum of A and B. Now what we were doing is taking the minimum of A and B and recoloring that side to be the other side. When we do that, we have these constraints, which are supposed to be equality constraints.

These things are supposed to be equal, but they weren't before. Before I recolored this, these were-- at least one constraint here was violated, because this one of them was red, one of them was black. When I fill this in, I improve my solution,

because-- by at least the size of the cut. Each of these guys now becomes satisfied. It wasn't before. So I improve my solution by this much.

I also worsen my solution, potentially, because that node appears in one clause. And so it gets worse by 1. So suppose there are B nodes on the smaller side, and we're recoloring B nodes. So we got an improvement by P, and also we worsened things by up to P. Because these P guys appear in P different clauses. Each one potentially we mess up is no longer satisfied.

But for every one that we mess up-- so these guys up here in some actual clause-- each one we mess up, we also make at least one thing happy. Because we fixed the cut. So if we have any solution, we can convert it into one where our variables are all true or all false, and not lose anything. Therefore, there exists an optimal solution.

There exists an optimal solution. And you do this variable by variable, where variables are all true or all false. Now, we do change the value of OPT here. Because we added a ton of constraints, and we just said well the-- in the optimal solution, or an optimal solution-- in fact, all of these constraints will be satisfied. Which means OPT has increased. So the OPT for x-prime, this thing, is going to be larger than the OPT for x.

OPT of x-prime is going to equal OPT of x plus order the total number of occurrences of all variables, which is at most three times the number of clauses. So we want to know, does it satisfy this definition of L reduction. We need to know that the OPT does not explode by more than a constant factor. And yet, we added this big term.

But the good news is, in 3SAT, you can always satisfy a constant fraction of the clauses. So OPT is always at least-- I think I have written here, like, half of them. I think you just randomly assign the variables, and some constant fraction will be satisfied in expectation. So there's definitely a solution where OPT of-- so OPT of x is definitely at least some constant times the number of clauses. And so this adding some constant times the number of clauses doesn't change the overall cos by more

than a constant factor. So property one holds.

Property two holds, in fact, with beta equal 1. You're not making your solution in-- you see we have this additive thing, because we add these gadgets and stuff. Multiplicatively, it's confusing, and that's what we were worrying about here. But additively, it's very clean. We always add the exact same amount to your solution.

So if you have a solution y-prime and you convert it back to a solution y, the gap-- the additive gap between the cost of y versus OPT of x will be equal to the additive gap between the cost of y-prime versus OPT of x-prime. Question.

AUDIENCE: So here, how many times are you using each variable?

PROFESSOR: We are using each variable 29 times. We're using it-- why 29? Because-- right. We have degree 14, but then for every edge we actually have two constraints, the implication [? in ?] both ways. So that's 28. Plus the vari-- each of those nodes actually appears in one actual clause. OK.

So this proves that max 3SAT-29 is hard. Yep, good question. Why did I claim max 3SAT-3? Because we can use this reduction now. So there is another reason I showed you this. I haven't seen this explicitly said in the literature, but all the pieces are out there.

So once you show max 3SAT some constant is hard, then you can do an L reduction from that problem to max 3SAT-3, just like this. So first we do the expander. And then, still these nodes have too high a degree. They're degree 29. Now we're going to expand those into little cycles of constraints.

Now this is actually OK when the cycle has constant length, because maybe-- suppose some of these are set to true, some of them are set to false. Then just set them all to false. Don't even take majority. Set them all to false. How much does that hurt you?

Well, you know that each of these variables appeared in a constant number of clauses. So it only hurt you by a constant amount. Every time you flip a variable

from true to false, you only lose an additive constant in your solution. So when we're converting from a solution y-prime, which does weird things on a cycle, potentially, when we convert it to y and set them all to false, we-- I mean we know-- so there's two cases. One is, all of these constraints are satisfied.

Then, you should choose exactly what's written, and then they will all be true or all be false. But if at least one of them is violated, you can charge to that violation and set them all to false. And when you do that, I don't know how many-- you know, at most 1,000 violations happen, some constant. Like three times 20-- whatever. Some constant.

And then you know that you can charge that cost to the violation that you were given in y-prime. And so you can get that L reduction property, too, and say oh good, the additive gap in my produced solution where I just set all those to false is at most 1,000 times the original additive gap. And so that's an L reduction from max 3SAT constant to max 3SAT-3. Questions? Yeah.

AUDIENCE: So how do you know there's a constant number of violations?

PROFESSOR: Because now we were given an instance of 3SAT constant, meaning each variable appears in a constant number of clauses. So we were given a situation-- right, sorry. Also, when we do this, each-- we set it up so each of these variables appears in one original clause. Yeah And we know the total size of the cycle is constant. So each of these, every time we turn one of these to false, we lose one point. And there's only a constant number of these, so it's actually-- the constant is only 27. Or 29, sorry. 29.

It's a little weird to get used to L reductions. I'm still getting used to them. But as you can see, it's pretty powerful. You can do a lot, and it's just OK. You definitely have to be careful. In general, you want things to have bounded degree. That makes things really helpful. That's why I'm telling you about these two problems.

What next. Let's just continue-- let me at least mention, in case I run out of time, max not all equal 3SAT, also hard, positive 1 in 3SAT, even one in E3SAT. Also,

APX hard. APX complete. So those are good friends from 3SAT land to carry over here. But let's prove some-- and I will eventually prove those.

Let's prove some other fun problems. Next one is independent set. Now I have to be a little bit careful here. Independent set is really really, really hard. But an interesting special case is bounded degree independent set. So that's what I'll talk about next.

I think I'm going to prove max degree for independent set, although it's known that max degree 3 is hard also. So constant degree independent set is actually APX complete. So there is a constant factor approximation, which is take any maximal independent set-- just keep adding vertices until you can't anymore. That will be within a factor of this of optimal, you can show. That's pretty clear.

So that puts it in APX. And furthermore, we claim that's [INAUDIBLE]. So there's a constant factor, and there's also a constant factor in approximability. Some constant you cannot go below. And the proof is this, although it's a little bit-- so it's a reduction from, let's say, the one we just did, max 3SAT-3.

And it's going to be a strict reduction. We're not going to lose anything if I did things right. Now in fact-- so I drew here six recurrences of xi, but there's really only three of them. But the idea is complete bipartite graph between the positive instances of xi and the negative instances. And then these are going to be plugged in directly to the clause gadgets. These are the same variable.

And so the idea is each of these was plugged into only one clause. So I need to make copies. I do this complete bipartite graph between them. And now we're trying to do max independent set, which means whatever solution we find, it will be an independent set. That's cool. There's no slack in independence here. It's just about how many we choose. Question?

AUDIENCE: What is an independent set?

PROFESSOR: Independence set is you want to choose a set of variables that have no edges between them. Sorry, instead of vertices, they have no edges between them. So

that means that if I choose any one of the blue nodes here, I can't choose any of the red nodes, and vice versa.

So I may not be able to choose all of the blue or all of the red, but that's how it goes. Now if we look at a clause, there's-- in this case, we have to actually handle the case of clauses of size 2 and clauses of size 3. If you have a clause of size 3, you'd just build a triangle on them. And the idea is only one of those can be chosen.

But it could be zero get chosen, because you might be screwed. Maybe you chose a red xi, and then you won't be able to choose this blue xi. Maybe you chose a red xj. Maybe you chose a blue xk. In that case, you won't be able to choose any of these vertices. But if at least one of these [INAUDIBLE] is true, if you chose-- if you're either choosing the blue xi's or the blue xj's or the red xk's, then in fact you get one point for each clause.

And in general, the number of points you get, the number of things you'll be able to put into your independent set, is exactly the number of clauses you'll be able to satisfy. And just by looking at whether any of the blue xi's are true, then you set xi to true, looking at whether any of the red xi's are set-- are chosen in the independent set, then you set xi to false. That will recover the assignment, and it'll have exactly the same cost as the independent set size.

Number of clauses you satisfy will be exactly the number of independent set size. And similarly, for a clause of size 2. So that's cool. Independent set is really easy to reduce from max 3SAT-3. In fact, it would work for max 3SAT constant. But if you do a max at 3SAT-3, there's only three of these guys. Then I think the biggest degree you get is 4. This guy maybe is attached to two things, and then also to two things over here. Great.

Next problem is vertex cover. So this is a funny one. So let's do a constant degree vertex cover. In general, there's a two approximation for vertex cover. So we don't need the constant degree to be an APX, but. This is also APX complete. And it's kind of identical to the independent set in a funny way, which is for any graph, if you look at a vertex cover, its complement is an independent set.

If you look at any independent set, its complement is a vertex cover. Sorry, any maximal independent set, its complement is a vertex cover. So they're kind of duel in that, if you look at the size of a vertex cover plus size of a maximal independent set, it will always equal the number of vertices. So maximizing this is the same as minimizing this. But approximating this is not necessarily the same as approximating this.

One's a maximization problem, one's a minimization problem. But it's still an L reduction for bounded degree graphs, because if you have degree at most delta, there's always an independent set size of at least N over delta. And there's always a vertex cover of size N. So they're within constant factors of each other.

In fact, these are both always theta the number of vertices. OK? So the reduction is you give me an instance of independent set, I give you that exact same instance to vertex cover. And then-- so f is trivial. G takes the complement. Whatever you had in the vertex cover, you don't put it in the independent set, and vice versa. And then you just have to check that this is an L reduction.

So the first thing is the OPTs are within a constant factor of each other. That's true, because they're both within a constant factor of the number of vertices. And then you prove that the additive gap is fixed. And it's the same thing if you decrement this accidentally, then you increment this accidentally. They're one for one.

So this is kind of cool. It feels a little scary, but we can convert a maximization problem into a minimization problem with L reductions. This would be very hard to even think about in the other reduction types, which is one of the reasons L reductions are so successful, I think. That was vertex cover. We can do one more.

OK, really easy. Dominating set. Remember, dominating set-- with vertex cover, when you put a vertex in your cover, you cover all the edges incident to it, and you want to cover all the edges. Dominating set, when you put a vertex in, you cover all the neighboring vertices. You want to cover all vertices. So I'm going to reduce from vertex cover.

If you have an edge, what you do is convert it into a path of length 2 plus that edge. So then you know that, if this is in the dominating set, you can just move it over to [INAUDIBLE] or w. It will cover all the things it could cover before, and maybe even more. So then in the optimal solution over here, you'd never need to choose one of these vertices, which means we can assume that on the original vertices, which means you are just solving vertex cover, because covering that is the same as covering that edge.

Good. I think that's good for now. We'll do a bunch more reductions next time.