PROFESSOR: All right. Today we continue our theme of approximation, lower bounds inapproximability. Quick recap of last time. We talked about lots of different reductions. We, I guess, in particular talked about P-tests, AP and L. And in particular we'll be using L-reductions almost exclusively today, except the occasional strict reduction, which is even stronger, in a sense.

So what's an L-reduction? We're trying to go from one problem A to another problem B. We're given an instance x of A. We convert it via function f to an instance x prime of B. Then we imagine that somehow we obtain a solution. We don't know anything about it. y prime to x prime. That's in B space.

And then, in the reduction, we're supposed to be able to map any such solution y prime to x prime via g into solution y of x in A problem-- so that's given by the function g-- such that two things hold. The first one is that for f, optimal solution of x prime should be at most some constant times the optimal solution to x. So we don't blow up OPTs too much. And secondly the absolute difference between the cost of y versus the optimal solution for x should be within a constant factor of this kind of gap-- additive gap between the cost of y prime versus the optimal solution to x prime, meaning that if we were given a y prime that's very close to optimal for x prime, then the y we produce is very close to optimal for x.

And we want that in an additive sense that will imply that things are good in a multiplicative sense. Last time we proved that for the min case, for minimization problems. If you're curious, I worked out the details for maximization problems. It's a little bit uglier in terms of the arithmetic. But you again get that if you had a constant factor approximation over here, you preserve a constant factor approximation over here, and you only-- you lose a reasonable factor.

We also have that if you can get a PTAS over here, so you can get an arbitrarily good approximation, you also get a PTAS over here. That was the PTAS reduction. And it turns out the constant in the end is roughly epsilon over alpha beta, where alpha was this constant, and beta was this constant. That's what we had before. It's a little bit different. For small epsilon, it's about the same. But for large epsilon, it does make a difference.

And this is why, in case you were confused, an L-reduction does not imply in the maximization case an AP-reduction, because you have this non-linear term. Here, everything was linear in epsilon. With minimization, that's true. The L implies AP. But for maximization it's not quite true. It's close. So there's some funny. What I said didn't quite match this picture. That's an explanation.

And then we did a few reductions. I claimed that Max E3SAT-E5, this was exactly three distinct literals per clause, exactly five occurrences of each variable in five different clauses.

I claimed that was APX-complete. We didn't prove it. What we did prove is that assuming Max 3SAT is APX-complete, we reduce that to Max 3SAT3, which is at most three occurrences, each thing, first by using expander, and then splitting the constant size-- constant occurrence variables-- with the cycle of implications trick.

And then we reduced from that to bounded degree. I think we did like max degree 4. But all of these can be done in max degree 3. Independent set, vertex cover, and dominating set.

Vertex cover we've seen a lot. You want to cover all the edges by choosing vertices. Dominating set, you want to cover all the vertices by choosing vertices. Each vertex covers its neighbor set. And independent set, for general graphs this is super hard. But for bounded degree graphs, there is a constant factor approximation. This was choosing vertices that induced no edges.

So with that in mind, let's do some more APX-reductions, APX-hardness, using L-reductions.

So the next problem we're going to do is Max 2SAT. So because we're in the world of optimization, in some sense the distinction between 2SAT and 3SAT is not so important. It turns out Max 2SAT will be APX-complete just like Max 3SAT was. So when we didn't have Max, of course the complexities were quite different. 3SAT was hard, 2SAT was easy. With maximization, they're going to be equivalent in this perspective.

So I'm going to do an L-reduction from independent set of, let's say a degree 3. So it'll work with any constant degree, but we'll get a different number of occurrences. And the reduction is the following. There are two types of gadgets for every vertex.

So I'm given an independent set instance. For every vertex v, we're going to convert that into a clause-- namely v. I want v to be true, if possible. It's a funny way of thinking when you're maximizing a number of causes, because a lot of the clauses won't be satisfied. But you're going to try to put v in the independent set if you can. That's the meaning of that clause.

Then for every edge-- let's say connecting v to w-- we're going to convert that into a clause which is not v or not w. We don't want them both to be in the independent set. That's the meaning of-- yeah. I'm trying to simulate independent sets. So I don't want these both to be in. This is a 2SAT clause.

So what's the claim here? Suppose you have some assignment to the variable. So there's one variable per vertex over here. The idea is that variable should indicate whether the vertex is in the independent set.

And the claim is that we will never violate an edge constraint, or it's never useful to violate. The claim is that there exists an OPT-- optimal solution-- satisfying all of these edge constraints.

So we're doing Max 2SAT. So we get a point for every one of these things that we satisfy. And so in particular, if you didn't get this point-- not v or not w-- the converse of this is that they are both in. Then the idea is that you instead take one of those vertices out of the independent set, and that will be better for you.

In general, when you put a variable in an independent set, it only helps you for one clause. There's only one occurrence of positive v in all of these things. You might have many edges coming into a vertex, and they all prefer the case that v is false. So things are going to be easier if you set v to false.

So if you discover a clause like this, which is currently false, meaning both v and w are true, you're going to gain a point by setting v to false. You'll also lose a point, but you'll only lose one point. Potentially, you gain many points, but you gain at least one point and lose at most one point by switching from both v and w true into just one of them true.

So you can always convert without losing anything in OPT into a solution that satisfies all edge constraints. And then we know we have an independent set. That's what the edge constraints say. And therefore the remaining problem is to maximize the number vertices that are in the independent set.

So that means if we're given any solution y prime to this Max 2SAT instance, we can convert it back to an independent set. Now it's not quite of the same value. In general, the optimal solution here for the 2SAT instance is going to be the optimal solution for the independent set instance plus the total number of edges, because we're going to satisfy all of these. That's what we just showed.

So this is where we get a kind of additive behavior, like in this L-reduction. The gap is an additive thing. But here it's a nice fixed thing. And so these are pretty much the same. There's just this additive offset.

So that's going to be fine in terms of the second property. The additive difference between one of these solutions and OPT will be exactly the same. The beta here at this constant will be 1. But we do have to worry about the first condition. We need to make sure OPT doesn't blow up too much, because we did make it bigger.

So for that, all we need is this is omega, the number of vertices. And that's because we assumed our graph had bounded degree, and so we can always find an independent set of size something like n over constant.

So because that's already linear, we only added another linear thing. Again, also this is order, number of vertices. So we're not adding too much relative to this, because bounded degree. Cool? So that's Max 2SAT, APX-hardness.

Fun fact which I won't prove. Max E2SAT-E3 is also APX-complete. So here we got some bounded number of occurrences. I guess each variable is going to appear in one plus three, four clauses. You can get that down to three clauses per variable.

OK. Now that we have Max 2SAT, we can do another one, which is Max not all equal 3SAT. So from SAT-land, we have 3SAT, not all equal 3SAT, and 1 and 3SAT. We're going to get all of those. Actually, we can even get 1 and 2SAT. Little bit stronger. But let's do not all equal 3SAT.

So here we are going to do, I believe, a strict reduction from Max 2SAT which we just proved, APX-complete. Yeah.

It's again in APX, because you can, say, take your random assignment, and you'll satisfy some constant fraction of the clauses. And OK. So here's the reduction. Again, very easy. Suppose we're starting from Max 2SAT, so all our clauses look like this. These may be negated or not. And we're going to convert it into not all equal of x, y, and a. a is a new variable, and it appears in every single clause. OK? So this is kind of funny.

So a appears everywhere. And not all equal has this nice symmetry, right? There wasn't really a zero or one. You can think of them as red, as blue. Doesn't matter whether red is true or blue is true.

So in particular, we can use that symmetry to make a consider it as false. So by a possible flipping everything, we can imagine that a equals zero. If not, flip all the bits, and you'll still be not all equal. Or all the things that were not all equal before will still be not all equal. You'll preserve OPT.

Now once you think of a is false, then not all equal is saying that these are not both

0, which is the same thing as saying 2SAT. Duh.

OK. Again, I mean this is saying OPT is preserved. But if you take any solution to this problem, you first possibly flip it so that a is zero, and then convert the xy is just exactly the xy's over here, and you'll preserve the size of the solution. You won't get any scale here, and you also preserved OPT exactly. So it's in particular an L-reduction, but it's even a strict reduction. Didn't lose anything. No additive slop or whatever.

OK. That's nice. Next is usually called Max-Cut. You're given a graph. You want to split it into two parts to maximize the number of edges between the two parts. But this is the same thing as max positive 1 and 2SAT, which is simpler than 1 and 3SAT.

You have, I mean, in a cut, again, you have two sides. Call them true or false, or red and blue, or whatever. You would like to assign exactly one of these to be true. Then that edge will be in the cut. So it's the same problem.

And you can also think of it as max positive XOR-SAT. Maybe actually call it 2XOR-SAT. Same thing. It's just every constraint is of the form this x or this. You want to maximize the number of those constraints. So a lot of these problems have different formulations depending on whether you're thinking about logic, or thinking about a graph problem.

So we're going to get all of these four with one reduction. And it's going to be from probably this one. Yes. The great chain of reductions here.

So we're going to reduce from Max not all equal 3SAT. I should mention, all of the reductions we've been seeing, including this initial batch where we started from 3SAT, converted into 3SAT 3, converted it into an independent set, to vertex cover, to dominating set to Max 2SAT, to Max not equal 3SAT to Max-Cut, are all in this seminal paper by Papadimitriou and Yannakakis, 1991.

This is before APX was really a thing. It had a different name at that point-- Max SMP-- which later is proved to be essentially equal to APX, or the completeness version is the same. You don't need to know about that. It comes from a different world, but all the reductions apply here.

So here is the reduction for a Max-Cut. So again we're trying to simulate Max not all equal 3SAT. Now we actually saw in the planar lecture, planar 3SAT, that you can reduce planar not all equal 3SAT to planar Max-Cut, and that we use that to get a polynomial time algorithm for planar not all equal 3SAT. We're just going to do the reverse.

And if you recall, this was the heart of that reduction. The point is that you can represent a not all equal clause as a cut, as a Max-Cut problem on a triangle. Because in a triangle, either they're all equal, and then there's no cut edges, or they're not all equal, and then there's exactly two cut edges.

So that's for a cause of size 3. We also need to handle the case of a cause of size 2. But that's a two-gon, I guess, instead of a triangle. It works the same way here. You get 1 if they're not all equal, and zero otherwise. This is shown as the zero case.

OK. Now the one thing we need, because not all equal 3SAT here, we need negation. So we're going to build each variable and its negation with this gadget. This is a new gadget, variable gadget. It's just a whole bunch of edges connecting xi and xi bar. And you can make this. You can avoid the multigraph aspect here. But let's not worry about it here.

So in general, if there are k occurrences of this variable, then we're going to have 2k parallel edges, because the cost over here, the potential benefit here is 2.

Again, we want to argue that if we take an optimal solution, we can make it another optimal solution where xi and xi bar are on opposite sides of the cut. And the reason is, if they're both on the same side of the cut, you're not getting this benefit.

If you flip one of the sides, you get this huge benefit, which is 2k. And you say, well, how much do I lose if I flip this from one side of the cut to the other. Well, it appears in at most k different clauses, each of them gives me at most two points. So I'm losing, at most, 2k points by making these opposite. But I gain 2k points. So it never hurts me to do that switch.

So I can assume these two guys are on opposite sides, and therefore I can assume it's sort of validly doing the negation part. And then it just reduces to not all equal 3SAT. There's a difference between this one, where we only get one point, and this one we only get two points.

AUDIENCE: You get two points.

PROFESSOR: You get two points here? Oh yeah. You get two points. That's why we doubled the edge. So that's cool. I think you would be fine. It'd still be an L-reduction even if you have one edge. But this is nicer. And yeah. That's it.

Cool. This is Max-Cut. It will be a bounded degree based on the number of occurrences we got, which was like four. I mean, we can use three, and then we'll multiply. In general you can prove Max-Cut remains APX-complete for degree three graphs. So we're not going to prove it here.

So another kind of reduction trick to reduce degrees, just say degree 3 is possible. It's also Max Cut in degree 3 graphs is APX-complete. So you could call that max positive 1 and 2SAT, hyphen 3. Maybe even E3. All right.

So this gives you a flavor. This is a fun series of reductions, each one building on the previous one. But it gives you kind of starting point. A lot of the problems we're familiar with in NP completeness land, if you just add "Max" in front, they become hard.

I mean I guess Max-Cut always had a Max in front. Max 2SAT for NP completeness, we also had a Max in front. So those are familiar, and they're APX-complete. All of the problems, I've described, at least for bounded degree graphs, have constant factor approximations. So this is the right level. They are APX-complete. And that determines their approximability. Constant factor, no PTAS.

Now it would be nice to know which problems are hard. With NP-completeness, and in the SAT universe, we had Schaefer's dichotomy theorem that said-- let me cheat and look at my notes from, I think, lecture four-- that SAT is polynomial if and only if the clauses that you're allowed to do-- the operations you're allowed to do with variables-- are either have the property that when you set all the variables true, everything's satisfied. Or you set all the variables false, everything satisfied. Or every single clause is a conjunction of Horn causes. Horn clauses were a few variables, and at most one of them is positive. Or all the causes you have are conjunctions of Dual-Horn, which was, in every clause at most one of them is negated, or all of the clauses are conjunctions of 2CNF, only like 2SAT.

Or what I didn't give a name at the time, but is essentially a slight generalization of XOR-SAT. Let me give it a name here. I'm going to call it X(N)OR-SAT. You can also phrase them as linear equations over Z2. So this is zero and one. And it's either X OR, meaning you take the X OR of all the things-- that's like the summation of all things, or it's X(N)OR, meaning when you take that sum, it should equal zero.

And such systems of linear equations can be solved in polynomial time using Gaussian elimination over Z2. And all of the things I just mentioned are all the situations where SAT is polynomial. Every other type of clause, SAT is NP-complete-- or set of classes.

Now why do I mention this? Because there is an analogous theorem for it's not quite SAT, because we need something like this Max. We need to turn it into an optimization problem. SAT is not normally an optimization problem by itself. And characterizing how approximal those problems are.

Now it is a complicated theorem-- so complicated, that I don't want to write it on the board, because there's a lot of cases. But the point is, it's exhaustive. It will tell you if you have anything of the type we had with Schaefer, which was you define a kind of clause function. It's either satisfied or not. It applies to some number of variables. And then, once you've defined that clause type, you can apply it to any combination of variables you want. That family of problems with no other restrictions is what we get.

And I will just tell you what the problems are. There's four of them. This is part of what makes the theorem long, but also extremely powerful. The first dichotomy is max verses min. And then the second dichotomy is they call it CSP for constraint satisfaction problem. So you have a bunch of constraints. You want to satisfy as many as possible. So this would be the number of satisfied constraints is your objective, or your cost function.

Or the other version is what's called the ones problem, or max ones, or min ones. This is the number of true variables. So again, we have a Schaefer-like SAT style of set of clauses. Either we want to maximize the number of satisfied clauses, or we want to minimize the number satisfied clauses, or we want to maximize the number of true variables and satisfy everything. Or we want to minimize the number of true variables and satisfy everything.

OK. Now obviously, if the SAT problem is hard, it's going to be hard to do this. But it's still interesting. You can still think about it. And even when the SAT problem is easy, Max ones can be hard. So I am going to-- I wrote it all down, and then I realized how long it was. And so I will just show you. Imagine I just hand-wrote this.

So this is the easy case. Max CSP. So we want to maximize the number of constraints that we satisfy. And I'm going to characterize when it is polynomial. Now here, PO I haven't defined, but that's the analog of P for optimization problems. So it's the set of all optimization problems that are in P that have a polynomial timed algorithm to solve them exactly. So it turns out in this situation you are either polynomial or APX-complete. So it's only about constant factor verses perfect. There's never a PTAS, unless there's a polynomial time algorithm.

And the cases should look familiar. It's either when you set all the variables true or all the variables false, that satisfies everything. In that case, Max CSP is, of course, easy. You can satisfy everything.

Another case is if you write the clauses in disjunctive normal form-- this is a new type that we hadn't seen before, all your causes are-- when you write them in DNF, they have exactly two terms. So it's the OR of two things that are anded together.

Sorry. There's an "or" in the middle. And you have a bunch of things anded together in each of my hands. And all the ones in here and positive, and all the ones in here are negative. If every clause looks like that, then you can solve this in polynomial time. And in all other cases, this problem is APX-complete. So that's a nice, very clean characterization.

AUDIENCE: Wait. [INAUDIBLE] that we learned about earlier. Is this the [INAUDIBLE]?

PROFESSOR: Yes. This is disjunctive normal form. So it's the or of ands. We usually, we deal with CNF ands of ors. But for this characterization, every clause can be uniquely converted into a DNF, and uniquely converted into CNF. So that's a well-defined thing to say. With Schaefer, we just had to look at the CNF form. But here we get a new set of things.

All right. That was one out of four. Max Min CSP Ones. Next one is Max Ones. This is not the most complicated. But let's go through them. So again, we want to maximize the number of true variables. So of course, if we set all the variables to true, and everything is satisfied, yay, a polynomial, OK? But curiously, if you settle the variables to false, and that satisfies everything, that's going to be here. That's Poly-APX-complete.

Poly-APX-complete, you can translate to something like n to the 1 minus epsilon, approximable, and that's the best you can do. Or there's a lower bound of n to the 1 minus epsilon. Upper bound might be n or something. OK. So because maximizing ones, when setting things all at false, does not necessarily help you.

There are some more positive cases. If you have a Dual-Horn set up. So this is another one of the Schaefer situations. If every clause when you write it in CNF every subclause is Dual-Horn, at most, one negated thing, that is a good situation for maximizing ones, because only one of them has to be negative. But with Horn, for example, you get Poly-APX-complete, because we have an asymmetry here between ones and zeros. Question?

AUDIENCE: In this list, do we just read down it until we hit the thing?

PROFESSOR: Yes. Good question. This is a sequential algorithm for determining what you have. If any of these says, oh, you're in PO, then you should stop reading the rest of the theorem. The way they write the theorem is less is probably clearer.

They write an else if for each one, but I wrote it backwards, so it's hard for me to write else if. Yeah. Occasionally I'll mention that the previous things don't apply. But you should read this sequentially.

OK. So it was Dual-Horn. Another polynomial case is what I call 2-X(N)OR-SAT, where the N is in parentheses. So in other words, you have linear equations. Each equation only has two terms, sort of like 2SAT. And you have equations that say equal zero or equal one on those two terms. That is also polynomially solvable. This is a special case. We didn't need the 2 for Schaefer. Here we need the 2, because if you have X(N)OR-SAT in general.

And when I say this, I mean that all constraints fall into this category. If all constraints are of this form, all clauses are of this form, then you're good. If all clauses are of the form X(N)OR-SAT, but they're not in this class, they're not all of length 2, then the problem becomes APX-complete, by contrast to Schaefer, where, I mean, deciding whether you can satisfy all those things is easy-- maximizing the number of ones when you do it is APX-complete. So that's particularly interesting.

AUDIENCE: Not all equal 3SAT fall in that? Is that?

PROFESSOR: Not all equal 3SAT.

AUDIENCE: Those are X(N)OR clauses, right?

PROFESSOR: No. They should not be X(N)OR clauses, because it's NP-complete. And when you have X(N)OR clauses, it's always polynomial to decide whether you can satisfy everything. So it's in the other case. But good question, because we should be getting APX-completeness. Yeah, but Max not all equal 3SAT is different. Here we're trying to maximize the number of clause that were satisfied.

So if you have not all equal 3SAT, and you want to maximize the number of ones, that means first you have to satisfy not all equal 3SAT, which is hard. So that's going to fall into this.

The bottom one is feasibility. Just finding a feasible solution is NP hard. The X(N)OR-SAT is this thing-- linear equations over Z2. And it could be equal to 0, or equal to 1. This is what you might call an X OR clause, or this is an X OR clause, this is an X(N)OR clause. So if they don't all have size two, then you're APX-complete. But you can find a solution by Schaefer's theorem.

OK. So as I mentioned, Horn clauses and 2AT clauses are actually really hard. They're Poly-APX-complete, n to the 1 minus epsilon. Also these are all situations where you can find feasible solutions easily by Schaefer, like when you can set them all false, and that satisfies everything. It doesn't help you when you're trying to maximize the number of ones. It just gets you to zero. Then you want to do better. And it's really hard to get any better factor.

One more situation. Sorry. There's a slight distinction here. So suppose you have the feature that you can set one variable true, and the rest false. If that satisfies all your constraints, than great, you found the value 1.

And there's a big difference between 0 and 1 when you're looking at relative approximation, because anything divided by 0 is huge. So it's really hard to get a good factor. That's the situation. Distinguishing between 0 and greater than 0, which is an infinite ratio, it could be NP-hard. That's when you, in this situation, we set all the variables false. You get zero. But finding any other solution is going to be NP-hard.

Here, if you can at least get 1, you can get an N approximation, whereas here you can't get an N approximation. Here you can get Poly approximation. And finally, if you have none of this above situations, then testing feasibility is NP-hard by Schaefer's theorem. So it's like Schaefer theorem, but some of the cases split up into parts. Now, that was maximization. Question?

AUDIENCE: So, what's special about 1 here? It seems to me if you replace that 1 by K it should still be in that case.

PROFESSOR: This case.

AUDIENCE: Yeah. If I just replace that one with a fixed K. Like 2.

PROFESSOR: Yes. So that problem will still be-- so if you can set all but K of them true, I think you can also set all but one of them true, and still satisfy. Yeah.

So here's the thing. This is all variables, right? So the idea is you have tons of variables, and let's say two of them are set to true. So if you look at a clause, the clause might just apply to these guys-- all the false guys-- or it might apply to false guys and one of the true guys, or it might apply to false guys and two of the true guys. All of those would have to be satisfied in your hypothetical situation.

If that's true, that implies that all the clauses are satisfied when only one of them is set true, and the rest are false. So your case would fall into this case as well, and you'd get Poly-APX-completeness again.

So it's not totally obvious when these things apply. But this is the complete list of different cases. Any questions?

OK. Two out of four. Next one, this is the longest one, is Min CSP. Now here we don't get as nice a characterization, because there are some open problems left. I haven't checked whether all of these open problems remain open, but as of 2001 they were open, which was a while ago. And we can check whether there's more explicit status. But I have the status as of this paper here.

So Min CSP. This is, you want to minimize the number of constraints that are satisfied, whereas before we looked at maximization. There are only three cases which were something like this. Again, if setting all the variables false or true satisfies all the clauses, this is good, apparently. That's less obvious in this case.

In general, minimization problems behave quite differently from maximization problems in terms of approximability. Maximization is generally easier to approximate, because your solutions tend to be big, and it's easier to approximate big things. Minimization-- small-- is hard.

Also we had the situation from Max CSP, if when you write it in DNF, is exactly two terms for every clause. One of them is all positive variables, and the other is all negative variables. That's also easy.

And here's a new case of APX-completeness. So if the problem you're trying to solve is exactly this problem, they call this, I think, implication hitting set. So you have a clause which lets you say x1 implies x2 for any two variables. And you have some set of clauses like this, where you can say here's five variables. The OR of them is true. No negation here.

So this is called hitting set, meaning I give you a set of vertices and a graph, and I want at least one of them to be hit, to be included, to be true. And we're trying to minimize the number of such things that we satisfy. So this turns out to be hard, but only there's no PTAS, but there's a constant factor approximation.

And then we have these four cases which show that they are equivalent to known studied problems. So there are these special cases. Other than these getting any approximation factor of less than infinity would require you to distinguish between zeros OPT, and OPT is greater than zero, and it's NP-complete, unless you have these.

So there are some special cases like Min Uncut. This is the reverse of Max Cut. You want to minimize the number of uncut edges. So that plus Max Cut should be equal to the number of edges. But the approximability of the two sides is quite different. And here are the best results of our APX-hardness, and log and upper bound for approximation. So that's a little bit harder maybe. It's at least as hard as this.

And that happens when you are in the 2x (N)OR-SAT situation, something we saw from the last slide. So here it reduces to this other problem. Basically the same, but the X(N)ORs don't buy you anything new.

In the case of 2SAT, you get a problem known as Min 2CNF deletion. And it's similar-- APX-hard, and best approximation is log times log log. If in the case where you have X(N)OR-SAT in general, but it's not all of the linear equations have only two terms-- so we have some larger ones-- then it turns out to be equivalent to nearest Codeword.

So it turns out you can write all such equations using either equations of length, by using equations of length 3 always. So this is linear equation. This should equal 1, or this says equals zero. And from that, you can construct all such things.

This is a really hard problem. Poly-APX-hardness is not known. Current lower best lower bound is this 2 to the log to the 1 minus epsilon, which we saw in the table of various inapproximability results last time. So this is a little bit smaller than n to the epsilon, but it's kind of close-ish.

And finally, in the-- I didn't write it. If you're in CNF form, and all of the subclauses are either Horn, or all of the subclauses are Dual-Horn, then you get something called Min Horn Deletion. And this has the same inapproximability. Here it's known.

So up here, the best approximation is n-- nothing, basically. Put them all in. And here there's a slightly better approximation known , I think, n to the 1 minus epsilon, or something. But these are all super hard.

The main point of this is so that you're aware of these problems. If you ever encounter a problem that looks anything like this, or it looks like some kind of CSP problem, you should go to this list and check it out. So don't memorize these, but look at the notes. Definitely memorize these guys. These are good to know. But there's a few obscure problems here.

OK. Last one is minimizing the number of ones. So this is like the hardest of two worlds. Minimization is kind of harder. And here you have to satisfy everything, but minimize the number of true variables.

So this is easy if you can set them all false. And then you win. This is easy in the Horn case. The Horn case is when at most one is positive, so most of them can be set to zero. This is easy in the 2X(N)OR case. So if you have linear equations, two terms each, equal to 0 or equals 1, that's also. And you want to minimize the number of true variables. That's good.

If you're in 2CNF form, there's a constant factor approximation. That's the best you can do. APX-complete.

This is a case from the last slide. If you have the hitting set constraints on constant number of constant size vertex sets, and you have implication constraints, then your problem is APX-complete again.

And then we have these guys appearing, again nearest Codeword. N Min Horn deletion. This one we get in the Dual-Horn case. The Horn case is good. Dual-Horn, we get this thing, which was like log N approximal. Or no. This was the 2 to the log N to the 1 minus epsilon. And this is X(N)OR-SAT when they're not all binary. Then we get nearest Codeword-complete.

And finally, oh, two more. The dual to this, if all the variables being set true satisfies your constraint, that gives you a solution, but it's like the worst solution possible, because you get N. And so in that case, you can get probably a poly approximation. Not very impressive. And that's actually the best you can do, at some N to the 1 minus epsilon. And in all other cases, by Schaefer's theorem, deciding whether even finding a feasible solution is NP-hard. So, good luck approximating.

Cool? This is the Khanna, Sudan, Trevisan, Williamson multichotomy theorem.

All right. So let's do some more reductions. My goal on this page is to get to our good friend from one of the first lectures, edge-matching-puzzles.

You have little square tiles, colors on the edges. Normally we want to satisfy all of the edge constraints. Only equal colors match, are adjacent to each other. Now the problem is going to be maximize the number of satisfied edge constraints. But before I show you that reduction, I need another problem, which is APX-complete. So that problem is APX-complete.

So I need two more problems. One is Max independent set in 3-regular 3-edge colorable graphs. OK. I'm not going to prove this one, because we already did a version of independent set, and it's just tedious to make it-- first, to make it exactly degree three everywhere, and secondly make it 3-edge colorable.

With 3 regular 3-edge color is a nice kind of graph, because every vertex, you've got one edge of each class. So that's kind of cool. And we can use this. This problem is basically equivalent to the actual problem I want, which is a variation of three-dimensional matching.

So remember three-dimensional matching, you have three sets-- A, B, and C. You look at the triples on A, B, and C. And you're given some set of interesting triples among those.

And with 3DM, what we wanted was to choose a set of such triples that covers all the vertices, and no two of them intersect. That's the matching aspect. In this problem, we want to choose as many triples as we can that don't intersect each other. So the problem is choose max subset S prime of S with no duplicate coordinates, I'll say.

So let's assume A, B, and C are disjoint. Then I don't want any element in A union B union C to appear twice in this chosen set S prime. So that's the problem.

Now I'm going to prove that that's hard. It is basically the same as Max independent set, and three regular 3-edge colored graphs, because what I do is I take such a graph, and for each edge color class-- there are three of them-- those are going to be A, B, and C.

So if I have red, green, and blue, all the red edges are going to be elements of A, all the green edges are going to be the elements of B-- B for green. And then all the blue elements are elements of C.

OK. Then a vertex, as I said, has exactly one of each class. So that's going to be my triple. And that's it.

So now, if I want to solve three-dimensional matching among those triples, that's going to correspond to choosing a set of vertices in here, no two of which share a color. No two of which share the same item of A. Let's say A is this color of edge. So that means that the vertices over here are not connected by an edge.

So the cool thing here is that each element of A, B, and C only appears in two different triples. Corresponding to the two ends of the edge. So now we have max three-dimensional matching where every element in ABC appears in exactly two triples. So I guess I can even write E2 if I want to.

OK. That was our sort of homework. Now we have max edge matching puzzles. Again, we're given square tiles. There's different colors on the tiles. Any number of colors. And we would like to lay things out. And I'll tell you the instance here is going to be 2 by N. So it's fairly narrow, unlike the construction we saw in class. And we're reducing from Max 3D M2. That's why I introduced it. And this is a four years ago result.

So the idea is the triple is represented by these three tiles, and some more. But for starters, these three tiles. The u glue is unique-- global unique. So it wants to be on the boundary. And here tiles are not allowed to rotate, so it wants to be on the bottom boundary.

So this ab glues only appear as a single pairs. I guess they'll also appear over there. But not very many of them. So basically a, b, and c have to glue together in sequence like that. And the percent signs are going to be the same on the bottom row. So nothing else.

This is basically forced to do this. We'll actually have to do it a few times, but you have to build this bottom structure. And then the question is what do you build on top. And the idea is there are exactly one each of these three tiles which just communicate dollar sign left to right, and have a, b, c on the bottom.

So those are cool. And if you want to put a triple into your three-dimensional matching, then you put those in sequence. No mismatches. This is great. You can take a whole bunch of these, stick them next to each other, everything will match.

No errors. So you're getting some constant number of points for each of these.

But you will have to build more-- at least two copies of this bottom structure. And there's only one copy of this top thing. So that's the annoying part. But there are some variations of these tiles which look like something like this-- I'll show you all of them in a moment-- which have exactly one mismatch. So you don't get quite as many points. You get, I don't know, 15 instead of 16 points, or whatever.

Bottom structure looks the same. And the point of this is we know a appears in two different places. So we need two versions of the a tile. But we only want one of them to be happy and give you all the points, because you should only be able to choose the a thing once.

So yet this triple will still exist. adc will still be floating around there. You want to still be buildable, but at a cost of negative 1. So this part's still built. Then you have these sort of filler tiles. Your goal is then just get rid of all the stuff and pay a penalty. But you want to minimize the number of times you do this, or maximize the number of times you do this, and then it will be simulating Max 3DM.

There'll be some additive consistent cost, which is the cost of all the unpicked triples. And then this will be an L-reduction.

So I have some more slides. It's a bit complicated to do all of the details, but this is a fully worked-out example with two triples. We have a, b, c and a, d, c. And because they share a, we don't want them both to be picked. So the same as what I showed you just in the previous slide.

But then there are all these other tiles that are floating around in order to make all the combinations possible. And there's all these tiles to basically allow them to get thrown away.

And so that's not so clear. This is the overall construction. For every triple, you're going to have exactly these three tiles that we saw. It got rotated relative to the previous picture. Maybe rotations are allowed.

And then for every variable, here they're called x, y, z instead of a, b, c. But the same thing. For every a thing we'll have some constant set of tiles that includes the really good one. Sorry. The good one has two dollar signs. This is the one you really like.

And then there's all this stuff to make sure things can get consumed. And you can get rid of the triples and pay exactly one per unpicked triple. So I don't want to go through the details, but once you have that, you get an L-reduction from Max 3DN2. Questions? All right.

So I want to go up the hierarchy. We've been focusing on constant factor, approximable problems that have no PTASses. I will mention there before we go on that there are some constant factor approximable problems that are not, that have no PTAS, and yet are not APX-complete. So APX-complete is not all of APX minus PTAS. So there are APX minus PTAS problems that are not APX-complete.

So these are still useful from a reduction standpoint. You can use them to show that your problem has no PTAS. But you have to state them differently.

And they're somewhat familiar problems. One of them is bin packing. This is you're moving out of your house. You have a bunch of objects. You live in a one-dimensional universe. So each box is exactly the same size. It's one-dimensional in size.

And you have a bunch of items which are one-dimensional. And you want to pack as many as you can into each box-- but overall use the minimum number of boxes. It's a minimization problem. This has no constant factor approximation.

But you can find what's called a asymptotic PTAS, where you can get a PTAS-style result-- 1 plus epsilon times OPT plus 1. So an additive error. And so in particular, distinguishing between two bins and three bins is weakly NP-complete. That's like partition, right, between two bins and three bins. So you need this sort of additive one. You can't get a PTAS without the additive one. So it's not as hard as all constant factor inapproximable problems, but somewhere in between. APX-intermediate is the technical term.

Some other ones are minimum.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Oh, this is all assuming P does not equal NP. Yes. If P equals NP, then I think all these things are equal. So, thank you.

Another problem I've seen in some situations is you want to find the spanning tree in a graph that minimizes the maximum degree. This is also APX-intermediate. There's a constant factor approximation. No PTAS, but not as hard as all of APX.

And another one is min edge coloring, which is quite a bit easier than vertex coloring. So these are problems to watch out for. They're the only ones I know of that are APX-intermediate. There may be more known.

OK. So unless there are questions, I want to go up to log factor approximation.

Surprisingly, in the CSP universe, we didn't get any log approximation as the right answer. But there are problems where log is the right answer. Again, there's probably intermediate problems. But here are some problems that are actually complete over all log approximable problems. So there's a log lower-bound and upper-bound on their approximability. I've mentioned two of them-- set cover and dominating set.

First thing I'd like to show is that these two problems are the same. I'm not going to try to prove lower bounds on them-- at least for now. But let me show that you could L-reduce one to the other.

So the easy direction is L-reducing dominating set to set cover, because dominating set says, well, if I choose this vertex, then I cover these vertices. OK.

So let's call this vertex V, and then maybe a, b, c, d. I can represent that by a set-- namely v, a, b, c, d. If I choose that set, it covers those elements, just like when I choose this vertex it covers those vertices. OK. So that's a strict reduction from dominating set to set cover. In some sense, the bipartite version gives you more control. OK. This is the non-bipartite version of set cover.

So what about the other reduction-- reducing set cover to dominating set? So this is a little more fun. We need to build a graph dominating set that somehow has two very different types of vertices. We want to represent sets, and we want to represent elements.

So here's what we're going to do. We build a clique representing the sets. So there are nodes in this clique-- one for every set. And then we're going to have an independent set over here that will represent the elements. And then whenever a set over here contains an element over there, we will add an edge.

So in general, an element may appear in several sets, and the set is going to consist of many elements. But over here, there's not going to be any edges between these elements. These are independent. And over here, all of the edges exist.

So the intent is you choose a set of these vertices corresponding to sets in order to cover those vertices. And that's going to work, because these vertices are super easy to cover in the dominating set. You choose any of them, you cover all of them.

These guys, you never want to put them in a dominating set. Why would you put this in a dominating set, when you could just follow one of these edges and put this in instead? That vertex will cover this one, and it will cover all of these. And the only edges from here are to over here.

So if you choose a set, you'll cover all the sets and that one element. If you choose the element, you'll cover the element and some of the sets. So in any optimal solution, if this ever appears, you can keep it optimal and move over here. That is sort of arguments we've been doing over and over. So there is an optimal solution where you only choose vertices on the left, and then that is a set cover. Again, it's a strict reduction. No loss.

Cool? So that is why these two problems are equivalent. Now we're just going to take on faith for now that they are log inapproximable. And you've probably seen that this one is log approximable. So now you know that this is log approximable.

I would say most of the literature I see for inapproximability is either APX hardness, or what people usually call set cover hardness. I mean, the fact that set covers log APX-complete, that is complete for that class-- not just a log lower-bound-- is fairly recent. So people usually have called it set cover hardness. Now you can call it log APX-hardness.

So let me show you one example. There are a lot of both out there, and I'm actually just showing you sort of a small sampling, because there's so much.

So here's a fun problem. It's called token reconfiguration. And the idea is you're doing some kind of motion planning in a graph. So something like pushing blocks, except you have a bunch of robots, which here are represented-- well, you have a graph. And each vertex can either have a robot or not.

In some, you're given an initial configuration of how the robots are placed, and you're given a final configuration of how you want the robots to be placed. And they have the same number of robots, because you can't eat robots, or create them yet. So when robots can create robots, that will be another problem. So here you have robot conservation.

So in a configuration, there are three types of vertices in that situation. It could be you have a vertex that currently has a robot-- here they're called tokens, to be a little more generic. It could have a robot, but not be a place that should have a robot. So in the initial configuration, it has a robot, but in the final configuration it does not.

It could be you have some robots that are basically where they want to be. They are robot and also in the target configuration, there's a robot there. Or I guess there's four cases, but in this case we'll only have three. Or it could be that you want to have robot there, but currently you do not.

So this is an instance that simulates set cover. And this is a situation where robots are all treated identically. So you don't care which robot goes where. So you've got these robots over here, which don't want to be here. They want to be over there. I mean, if you measure this length, it's the same as this length.

And these robots don't want to move, but they're going to have to, because they're in the way. In this tripartite graph, they're in the way from here to there. I didn't tell you a move in this scenario is that you can take a robot and follow any empty path, OK So you can make a sequence of moves all at a cost of one, as long as it doesn't hit any other robots. So, a collision-free path. You follow it, then you can pick up another robot, move it along a collision-free path, pick up another robot, and so on.

So if you want to move all these guys over here, you're going to have to move some of these out of the way. How many? Set cover many. Here's the set cover instance in this bipartite graph.

So what you can do is take this robot, move it out of the way, move it to one of these elements, and then for the remainder of this set, which are these two nodes, you can take this guy and move it there in one step, take this guy and move it there in one step. The length of this doesn't matter, because you can follow a long path. And you just drain out this thing one at a time-- except for this guy, who you moved out of the way. You move one of these to fill his spot.

And if you can cover all the elements over here with only k of these guys moving, then the number of moves will be k plus A. So that's what's written here. OPT is, this is a fixed added of cost plus the set cover. And this is going to be an L-reduction, provided this is a linear in A, which is easy enough to arrange. So that's the unlabeled case.

You can also solve the labeled case. Maybe you want robot one to go to position one, and you want robot two to go to position two. Same thing, but here these robots are going to have to go back where they started. So you just add a little vertex so they can get out of the way. Everything can move where they want to. Again, choose a set cover, move those over, and then move them back. So you end up paying two times the set cover. But just a constant factor loss. Still an L-reduction.

And this problem is motivated, it's sort of a generalization of the 15 puzzle. You have a little 4 by 4 grid. You've got movable tiles. You can only move one at a time in that case, because there's only a single gap. This is sort of a generalized form of that, where you have various tiles. You want to get them into the right spots, but you can't have collisions during that motion. So that's where this problem came from.

15 puzzle, by the way, in the generalized n by n form is NP-hard and in APX, but I think it's open whether it's APX-complete. I would show the proof, but it's very complicated, so, I won't. Cool.

Well, in the last little bit, I wanted to tell you about the super high end. So we went to log approximation. There are other things known, but not a lot of completeness results. So we're going to get to other kinds of interapproximability next class. For now, I want to stick to something APX-complete. And the most studied class above log is poly, which is like n to the 1 minus epsilon.

And my main goal here is to tell you about some problems that you should, if you think your problem is like Poly-APX-hard, these are the standard problems to start from. There are two of them. And I've mentioned them, but not quite in this context.

They are clique and independent set. These are really the same problem. One is the complement graph of the other. Both maximization problems. And those are the standard ones. I'll leave it at that.

I'm going to keep going up. The next level most studied is Exp-APX-complete. So for these problems, the best approximation is n divided by log squared n. And there's a lower bound of n to the 1 minus epsilon. So there is a gap in terms of their approximability. But what we know is that they are the hardest problems that have any n to the ce approximation. They're all reducible to each other via PTAS reductions. So, fairly preserving.

So our next class up is APX-complete, things, problems approximable in exponential and n approximation factors. How would that happen? This is kind of funny. And the canonical problem here is the basic reason is numbers.

We take the traveling salesman problem. And every edge can have a weight. Let's say it's integer weights. But any integer weight that can be expressible in n bits is fair game, which means the actual value of that edge is going to be exponential in n. And from that, you can get a very easy lower bound. And in fact, all problems that are approximable in exponential APX can be reduced to general TSP, where you're just given a bunch of distances between pairs of vertices. It doesn't satisfy triangle inequality. That's the non-metric aspect.

The triangle inequality TSP, which is what normally happens, there is a constant factor. It's APX complete. But for general waits between pairs of vertices, non-metric, it's Exp-APX-complete, because you can basically make a graph and solve Hamiltonicity by saying all the edges in the graph have weight one or zero, and all of the edges-- I guess one would be a little bit more legitimate. And all the non-edges in the graph are going to give weight infinity. Infinity is the largest expressible number which is 1, 1, 1, 1, n bits long. And so either you use one of those edges or you don't. And there's an exponential gap between them.

So even if we disallow zeros being an output, then we get exponential separation. That doesn't prove completeness, but it proves that you can't hope for better than exponential approximation there.

OK. Two more even crazier classes. Now we did see these classes come up with the characterization theorem. But these are probably how these results were proved.

So you might think, well, double the exponential. I don't know. What's next? Next, you could define that. But what seems to appear most often is this is the ultimate class among all NP optimization problems, you could imagine being complete against all of them. And this is with respect to AP-reductions, one of the ones we saw.

And I'm going to define a very closely related class, which is NPO PB, NPO polynomially bounded. OK. So these are the hardest problems to approximate. This is basically the problems that have numbers in them, and this is the problem that have no numbers, or if they have numbers they are polynomially bounded, like the polynomial situation. So non-metric TSP, well, it's not as hard as NPO-complete, but it's more in this category.

AUDIENCE: Is there a notion of strongness, weakness in these kind of things?

PROFESSOR: That's funny. This is a stronger result. So there's not quite an analog. But you can do exponential tricks and give yourself a hard time over here. And here you're just not allowed to use. Everything's polynomial. So a three-partition is sort of more in this universe.

But in this situation, if you sort of have three partitions, but with exponential numbers, then you get this harder class. So this is not the analog of weak.

You could maybe imagine-- well, in some sense, weak is a modifier in the problem, where you say I want to restrict all the numbers to a polynomial size. So when you do something like three partition, it's sort of a weak problem, or it's a polynomially bounded problem. Strong NP hardness means that that is NP-complete. Anyway vague analog, but not quite. It's possible some of these, you could add a weak modifier, and it would mean something, but I don't know.

All right. So I just want to give you some sample problems on both of these sides. Maybe let's start with this side, which is a little more interesting, because you get some kind of familiar problems, and they're super hard. Minimum independent dominating set.

We've seen independent set. We've seen dominating set. Independent set is already hard to approximate. But this problem is worse, because even finding an independent dominating set is NP-complete, whereas finding an independent set, I can choose nothing. But if I want to simultaneously be dominating an independent, that's NP. Hard to find any solution.

In general in NPO PB problems, NPO PB-complete problems, it's always NP-complete to find a feasible solution. But it's worse than that.

So the first level would be to find a feasible solution. And this is saying on top of that you want to minimize the size. I think Max would also be hard. But I think there's a general theorem, that if you're hard in the min case, you're also hard in the max case. But it depends on the exact set-up.

So this is sort of an optimization version that makes it even harder than NP-complete. So I think this is NP-complete, and this is kind of even worse. It's sort of stating the stronger thing about when you're trying to optimize over a space of solutions, that it's NP-complete to decide.

Notice that's still an NPO problem. We define that solutions need to be recognizable in polynomial time. But we didn't say that you can generate one in polynomial time. So it could be NP-complete to find a single solution, like here. All of these problems will have that property.

Another fun problem is shortest computation. This is sort of the most intuitive one at a certain level. If you know Turing machines, and you have a non-deterministic Turing machine, which could take non-deterministic branches, you want to find the computation in such a machine that terminates the earliest using the fewest steps. So you might think of that as canonical NPO PB problem. There's no numbers in it, but as you can imagine, that's super hard to do.

Here's some more graph theoretic ones. Quite natural problems, but super hard. Longest induced path. Induced means, there are no other edges between the chosen vertices. So this is sort of longest path is one thing. That's quite hard to approximate-- like, I think, n to the 1 minus epsilon. That's sort of the analog of Hamiltonicity. Along this induced path is worse. Even finding an induced path of length k, finding a feasible solution, finding an induced path is hard.

Another fun one is longest path with forbidden pairs. So there are pairs of edges that you're not allowed to choose together, and subject to those constraints you want to find the longest path. So these are all NPO PB complete. No numbers in any of them.

Now let me give you some number problems. So Ones was you want to maximize the number of true variables. Now we're going to add weights. So we want to maximize the sum of the weights of the true variables-- and while satisfying a Boolean formula.

So again, finding a feasible solution is hard. That's not surprising. Here, the weights can be exponential in value, because we allow n bits for the weights. And that pushes you into NPO completeness. If you say the weights have to be polynomially bounded, then this problem is NPO PB complete. And that's sort of the starting problem that they used to prove all of these are hard. So they're reductions from this with polynomial weights to these guys.

AUDIENCE: [INAUDIBLE]?

PROFESSOR: 3SAT. I don't know whether you could go down to 2SAT is interesting. Here they say, I think, probably 3SAT or CNFSAT. Those reductions definitely still work. Whether you could put the 2SAT into the Max aspect, I don't know. But this could be fun to look at. There aren't a ton of papers about these two classes, but there are a few before they nailed down any interesting problems.

Here's another interesting problem. Suppose you want to do integer linear programming. To keep it simple, we'll assume that the variables are zero or one, and then that is equally hard. Here it's a little, unless you know a lot about linear programming, it's not so obvious that finding a feasible solution here is hard.

But in general, linear programing-- at least in the non-integer case-- you could reduce optimization to feasibility. So I think the same thing applies here. If you're not familiar with linear programming, it's basically a bunch of inequality constraints, linear inequality constraints. And now this is a bunch of integers. These are both given integer matrices and vectors. And they can have exponential value. Question?

AUDIENCE: For the max/min weighted ones, for polynomial bounded, is it still hard if you just do ones and minus ones?

PROFESSOR: I think min or max ones without weights is NPO PB-complete. I should double-check. I didn't actually mention, but this characterization theorem works for weighted problems also. For every single case, they show that weighted and unweighted are the same complexity, except for this one. In the min ones case, if all the variables' true, satisfy it, you get Poly-APX-completeness if you're unweighted.

If you're weighted, then you can't find any approximation. It's NP-hard to find any factor, which I think, this is, I think, before the introduction or popularization of these classes. So that may be distinguishing between Poly-APX-complete, which is definitely smaller than NPO PB-complete. This might be NPO PB-completeness. Unclear. But it's definitely worse than Poly-APX. Yeah?

AUDIENCE: How is it that distinguished from PXP? Because I'm just confused how you would ever get anything worse than this, because, that's like the biggest that you [INAUDIBLE].

PROFESSOR: So this problem is exponential APX-hard if you forbid zero. If you allow zero, then you can't get any approximation. Here, I think even when you allow zero, or even when you forbid zero, you still can't get an approximation. I think that's the idea here.

Here, these problems generally you can get, depending on your set-up, these problems you can all get like a factor, n approximation. Well, maybe not in polynomial time. This is hard to find. Some of these you can. Longest induced path, just have a path of length 1. That will be induced. So that gives you a factor n approximation.

There is a lower bound on this situation, n to the 1 minus epsilon inapproximability. I think morally it should be a factor n, but this is the best result I found.

So it's funny. This is only for number problems. So I presented this is as in between. But this is actually in some sense lower than Exp-APX-completeness. It's sort of a harder version of Poly-APX. This is a slightly harder version of Exp-APX. I think it's a small difference, but it's good to know there is this difference. Other questions?

All right. So this ends what I plan to say about L-reduction-style proofs, which are all about preserving approximability. The next class, we're going to look at a different take on inapproximability, which is called gaps, and gap preserving reductions, where you can set up a problem that either it has a great solution, or the next solution below that is way lower. And there's a gap between the best and the next to best.

And whenever you have such a gap, you also have an inapproximability gap, because you know there's this solution out there, but finding it, if it's NP-complete to find this, to solve it exactly, and so the next level down you lose some factor. And whatever that gap is is your inapproximability bound.

It doesn't give you completeness results like this in general-- not always. But it tends to give you really get inapproximability bounds.

Here I've completely ignored what the constant factors are. Most of them are not so great. Like when you prove APX-hardness, usually you get a 1 plus 1 over 1,000 kind of lower bound on the possibility factor. But the best upper bound is like 2, or 1.5. And what we'll talk about next time, you can get much closer-- sometimes exact bounds between upper and lower. But that will be next week.