PROFESSOR: All right. Welcome back to 6.890. Today we're going to look at a bunch of different graph problems, that is, their unifying feature is they're problems on graphs. We're going to look at vertex cover, we're going to look at vertex coloring of graphs, and we're going to look at some ordering problems on graphs, and I think, one more, orientations of graphs.

So these are all going to be some kind of constraint that we place on the graph or something we want to do with the graph. This is sort of a miscellaneous lecture. We've spent a lot of time talking about 3SAT, most recently Hamiltonicity, and before all that, 3-partition, which are the most popular, most useful, I would say, ways of doing NP hardness reductions. But there are few others that are good to know that are sometimes relevant, and so this is that. Each one is a little bit smaller in content, so we're just going to lump them all together.

So we'll start with vertex cover. We saw this slide from lecture four, I think, from planar 3SAT. Lichtenstein proved planar vertex covers hard. Vertex cover, remember, is this problem.

You want to choose a set of vertices, let's say k vertices, to hit all the edges. A different way of thinking about this problem-- and a lot of these problems, you can think of in a more logical context, formulas and so on-- this is essentially a form of 2SAT with k true variables. It's even a positive 2SAT.

Because essentially, you could think of there being a variable for each vertex, whether you choose it, and then the edge is just stating the constraint that you should choose one or the other endpoints and so that's a 2SAT constraint. And 2SAT is easy, but when you say, I want to solve 2SAT with only setting k of the

variables to be true, then it becomes this NP hard problem vertex cover.

And so we have-- This was a reduction from planar 3SAT to planar vertex cover, so we know this problem is hard. Let's use it for some things, and let's also prove some even more related versions are hard. So this one was already maximum degree 3, if we have some unused copies, we'll get degree 2 vertices, but certainly every vertex is at most degree 3. And it was planar, so that's already cool.

Planar max degree 3 is hard. Some polynomial versions to be careful of if you're using vertex cover as a starting point. One is what I call exact vertex cover. Each edge is incident to exactly one chosen vertex. So the vertex cover could be one or the other. If you have exclusively, or between those two, it's easy.

And another version, sort of the duel between vertices and edges, one might call it edge cover, which would be choose k edges to hit all vertices.

AUDIENCE: Doesn't exact edge cover perfect matching?

PROFESSOR: Exact vertex cover is perfect matching, thank you. This is also essentially maximum cardinality matching because the more you can be a matching, the more efficient you're going to be, and then every isolated vertex after you have a maximum cardinality matching, you just have to cover those with one more. So both of these reduce to matching.

So it's obvious in hindsight, but be careful, if you're ever doing vertex cover, not to accidentally do one of these. OK, so this morning's. Here's a cool version of vertex cover. It can be useful. We'll use it in a moment to prove a particular problem hard, connected vertex cover. So usually, with a vertex cover, you imagine you're just grabbing vertices from all over the graph. With connected vertex cover, you require that the chosen vertices induce a connected subgraph.

So this is not, obviously, easier or harder as a problem. It's more restrictive on the cover, but we can prove that it's NP hard by reduction from this problem. And so in particular, we'll get that it's NP hard for planar max degree 4 graphs.

So here's the reduction. I think this reduction is quite cool, because it uses planarity. It may not be necessary to use planarity, but this reduction definitely uses planarity in a rich way. So even if you were trying to prove this without planarity, it would be more awkward. So, suppose you have a planar graph and you want to find a minimum vertex cover in this graph. We're going to modify the graph in this way.

This is yet another Garey and Johnson paper. So you can see the original faces of the graph, and then for each face, we're going to add in sort of a copy of the face. So also the outside face gets this kind of curvy copy. And so in particular, at every vertex there's now going to be five copies of the vertex. One on the one incident face, one on the other incident face, and then for every one of these we're going to have an extra leaf hanging off.

Why leaves? Leaves are really cool, or really annoying, or whatever. They're very forceful in the case of vertex cover. If you look at a leaf-- So here's a leaf. It's connected to some vertex that's presumably not a leaf, otherwise there'd be an isolated component. It's never useful to put this in the vertex cover. If you decide to put this in the vertex cover, you might as well choose this guy instead, because this covers more. It covers all the edges that this one did, namely that one edge, and some other edges.

Maybe it was already in the vertex cover. Then it wasn't minimum. Then you got smaller. But you can assume there is an optimal solution where you never put leaves in the vertex cover. What that means is, wherever we add a leaf like this, it basically forces you-- It lets you know that you might as well have this in the vertex cover. There is an optimal solution where this is in the vertex cover, because the only other option is that this one is, and then you can move it over.

So these vertices on the copies of the faces, those are all forced to be in the cover, which means this entire copy of the face, the inset copy the face, is completely covered already, plus these edges are covered connecting the inner copy of the face to the original face.

So there's still stuff to cover. So in particular, it's still interesting to put one of the

original vertices in the cover. That would cover these three, now every edge got cut into three parts. So this would cover 1/3 of the edge. The idea is, if you put this in the cover and you cover those three guys, we still have to cover these two, and we'll do that by putting this vertex in the cover.

Because one of these two has to be in in order to cover this middle edge, and the idea is that you'll be able to just put one of them in. You'll be able to put exactly one of these two on every subdivided edge, if and only if there is a vertex cover of the right size. So if I do the arithmetic here, the claim is we added exactly 5 times the original number of edges to the optimal vertex cover.

5 because for every original edge we added 2 here, 2 here, and then 1 of these too. So there were 6 vertices around this edge that we added, and if we can cover, say, this edge using that vertex or cover this edge using that vertex, which is the same as saying that this original edge is covered by one side or the other, then we only need one of these two guys and then we'll get away with just 5 per edge.

So the original thing has a vertex cover of size k if and only if this new thing has a vertex cover size of k plus 5 times the number of edges. Question?

AUDIENCE: What stops you from taking one of the--

PROFESSOR: Taking both of these, for example?

AUDIENCE: Yeah.

PROFESSOR: Yeah. So that's a good thing to worry about. Maybe you take both and then you don't have to choose either one, so it doesn't look like a vertex cover, but this is a similar argument to the leaves. If you choose both of these guys, because you know that the other four are there, you can always move one of them to the vertex. So you'll get a recover vertex cover.

AUDIENCE: [INAUDIBLE] fives times the edges plus the number for vertices because you also have to cover the leaves off the vertices?

PROFESSOR: These leaves?

AUDIENCE: No, the one with the vertices.

PROFESSOR: Oh, these ones. Whoops. Oh, so it looks like they're always increasing the degree to 4. Connected vertex, however, is actually an important problem. People think about it a lot. For some kind of networking applications, you want to build some backbone that can reach everybody, you need a connected network.

But Garey and Johnson's original motivation for introducing that problem is rectilinear Steiner tree, also an important networking problem. Steiner tree is about usually you imagine you're given some space where you can build a network, and you're given some things that you want to interconnect.

So maybe you're building a new city around some existing houses, you're given some points in the plane that represent houses. Now you want to build roads to connect them all together. You want to minimize the amount of roads you have to build, so you're going to build a tree. And Steiner tree means that you can add intersections wherever you want, as opposed to a minimum spanning tree, where you can only turn at the given vertices.

If you have some points like this in the plane, the best way to connect them together in the Euclidean metric is to have-- I didn't draw it super well-- to have 120 degree angles at all the intersections, something you can prove.

Here, we're thinking about rectilinear Steiner tree, which is the Manhattan version, where you're only allowed to draw orthogonal connections, like that. So you can still add extra vertices if you want to minimize the total length of these connections given endpoints in the plane. This problem is NP hard, and you can prove it in a very simple way.

This is the reduction from the previous problem. So the first step is-- And the reason they care about max degree 4 in the previous problem is to draw the graph in the plane in orthogonal drawings. So every vertex becomes a point, every edge become some orthogonal path connecting two vertices. We've done that in previous

proofs.

So now everything's drawn in the plane, something like this. And that each vertex, we're going to erase a little circle of radius 2, and then that leaves a bunch of segments, and the segments connected together, and now we're just going to place along those segments a whole bunch of points. And so the idea is that the Steiner tree should connect all those together, and then it's left with a choice. And those are all going to be distance 1 apart. Then it's left with a choice of where to connect things together, which vertices.

So a little bit more formally, each of these edges is going to be scaled up by a huge factor, 4 times n squared, where n, I think, is the dimension of the n by n grid. So these are the original grids, the dash lines. Originally, when you draw it on a graph, on a grid, you imagine probably these are unit length.

Scale it up to the length 4n squared and then the rule is, wherever you have an integer point along that line, add a dot. And so they will be spaced with distances of 1. Over here, we have distances of 2 between the vertex, which is not actually a point in the set, and here. And so distance of 4 between any pairs of those points.

Because these distances are so huge, you never want to connect from here to anywhere else that's not at this local intersection. OK? And there's an argument about that, looking at these regions of where you could possibly want to go, and there's only those endpoints in there and these are really cheap to connect. So it really only pays to connect things in between.

So basically, you show that you're forced to do all the unit length connections, and now you just have to make the thing connected. Question?

AUDIENCE: Why 2?

PROFESSOR: Because 2 is bigger than 1. Yeah?

AUDIENCE: I guess with the vertex covers that you'd want to fill in that whole cross? But maybe there's a fear that you would just connect--

PROFESSOR: No. You won't fill in the whole cross in the vertex cover. Let me tell what you're going to do. This proof is-- I mean, the reduction is super simple, but the argument is confusing. So here's the idea.

So there's essentially two things going on, but let's first think about the case where I give you a connected vertex cover of some size. What I'm going to do is first-- or you can think of it in either order-- I'm, in particular, going to connect all the vertices together. Now, there's no particular reason to connect the vertices, but I'm just going to do it anyway, or think about doing it anyway, by a spanning tree.

So there's lots of redundant connectivity here. I don't need to connect this, to this, to this all the way around. I can drop one of those connections because I just need to be connected, I just need a spanning tree. And in general, there are v vertices to connect together, and each one of them costs 2, in a certain sense. To connect a vertex to an incident edge costs 2.

And so there are v minus 1 edges in the spanning tree, so I'm going to pay that. On the other hand, I also knew that every edge is connected to some vertex. That's the vertex cover constraint, and so every edge is going to pay 2 in order to connect to a vertex. So the weird thing here is the involvement of the vertices, even though there's no point there. That's funny.

Maybe we could add a point there and make it a little clearer, but instead of thinking about connections from edges to edges, which would cost 4, that's sort of the wrong way to think about it, because there's two things going on. One is just to get the edges to connect to something, and then there's getting all that something's connected together. So this is sort of a spanning tree thing, and this is more of the actual vertex cover constraint.

Together, they give the connector vertex cover constraint. It would be impossible for a Steiner tree to do one without the other. But essentially-- I mean, I'm going to wave my hands a little bit here, but you work it out. In all cases, the number of connections you need is exactly this if there's a vertex cover, and this will have to go up if there isn't a vertex cover. That was vertex cover.

Let's do coloring. This will be, I think, more fun. So first, let's prove that coloring is hard. So in general, in the coloring problem, also called chromatic number, you're given a graph and you're given a number k. You want to color the vertices of the graph using k different colors, so that's a mapping from the vertices to the colors such that no edge is monochromatic.

So you want a mapping, let's call it c, from the vertices to 1 up to k such that for every edge, let me call it vw, the color of v does not equal to color of w. So that's the no monochromatic edge constraint. It's usual coloring for vertex coloring. You could also talk about edge coloring and so on, and this problem is easy.

If k is 1, then you better not have any edges. It's easy if k is 2. That's bipartiteness testing. You can just greedily color and you can never make a mistake unless the graph isn't bipartite. But it becomes hard when k equals 3, so that's a fun transition. And so here is why vertex 3-coloring is hard as proved by Garey, Johnson, Stockmeyer.

Reduction from 3SAT. So we have, on the one hand, variable gadget. We're going to represent XI and XI bar like this. I mean, coloring should feel a lot like SAT. In fact, you might think of this as xor 2SAT, if you think of xor as the not equals operator. But it's over ternary logic. So this gets back to a question from class. What about ternary logic? You can think of 3-colorings like ternary logic, just like on the problem set.

So what we're going to do is we're going to have one copy of this gadget. I call it a colors gadget. It's just a triangle, and so all three colors must appear on that triangle. We don't know in what order, but we don't really care. We can just define the one. The color that this guy's assigned, we'll call it red. It could be 1, 2, or 3, but it doesn't matter.

Call the color that this guy's assigned green, and this one blue, and hopefully you're not sufficiently colorblind to be unable to distinguish those. But I did check with a color blind tester, they do seem at least different, but it might be hard to know the

names. All right.

So this green vertex is connected to both XI and XI bar for each of the variables, which means they can either be red or blue. And I want to red to mean false, blue to mean true. So that's cool, that's nice, regular binary logic. And then we're going to combine those variables with this clause gadget. Clause gadget also has one node out at the end here connected to both green and red, which forces it to be blue. But otherwise, it's kind of free.

So we have instances of literals. These don't have to be the positive forms, this could be XI bar, XJ bar, whatever. And now we're going to think about coloring. So let me show you a couple of possible colorings. Here's a valid coloring. It's valid because at least-- This is going to be 3SAT, so at least one of these should be true, true means blue.

This guy is blue, and in general, what we're doing is kind of taking an or of this pair and then an or of that pair with this one variable, one literal, I should say. And because this is not red, I can put red here. Again, all three colors must appear. So in general, what I want to do-- you'll see why in a moment-- is push the reds as far back as possible.

So if I can put a red or red here or here, I'm happy, and I put some other color over there. I guess I'll put-- It doesn't matter which one is blue or green here. As long as I can put red back here, I can also put red back here. And it could be if this one is blue, I could also put the red guy here, and I have the flexibility. But as long as I can put red either here or here, this one will not be red.

And furthermore, I can make it not green, and that's what these constraints tell me. This vertex should not be red or green. That's satisfied here. And you can show, you can check all cases, or just sort of go through that argument. The only bad case is when they're all red, because then, looking at this triangle, the red one has to be pushed forward.

And then, because this one is red and this one is red, again, this red has to be

pushed forward, but then we have a red-red adjacency and that's not allowed. So that is 3SAT to vertex 3-coloring. Cool. Now, this does not preserve planarity, because the colors gadget is connected to pretty much everything, and it does not preserve bounded degree. Question.

AUDIENCE: Yeah. The variable gadget doesn't seem to be connected to the clause gadget here.

PROFESSOR: Sorry. I mean, when I sit right XI here, I mean it's the same vertex as one of these. Yeah. So if there's n variables, n clauses, it's going to be 2n of these vertices and then they're shared among the n clauses. It's hard to draw, because actually they're identified as opposed to connected by an edge. Other questions? All right. So I think first we make it planar.

We have a new crossover gadget for specific to 3-coloring. Planar 3SAT doesn't seem to help immediately, so we're just going to, because we have all of these connections from colors gadget to everybody, plug this in whenever we have an intersection. And the idea, locally, is that whatever color is assigned to this vertex, x, must be the same as the color assigned to this vertex, x prime, and similarly, y, and y prime.

And they're free of each other, you can do any assignment to x, any assignment to y, and this will be satisfiable. I'll give you some colorings to give you-- I mean, this is essentially two cases, which is either x and x prime have the same color as y and y prime or they have different colors. So here's the same color case, you get this nice rotational symmetry.

In general, you've got this wheel pattern of four triangles, and you have some color here, and then that forced these guys to alternate in the other two colors available around that center, and then it essentially communicates the information you need. It's hard to do sort of a straight line argument about why this is the case, other than to just try all the possibilities.

But there's, again, lots of triangles. So once you know this is red, you know one of

these is green, and one of them is blue. Could go one way or the other at this point. I think you could actually do it one way or the other and just flip all the greens with blues, and vice versa, because green and blue is, in this case, local to the gadget. And, anyway, you end up with, once these two are set red, these two are forced to be set red by casework.

Here is the other case, when xx prime is different from yy prime, or you could say x and y are different, and then, again, it forces x to propagate through, y to propagate through. We still get alternation here, but now the unused color is in the center, whereas before, this picture, we had the center color was the color used by all three, or all four of them. On the outside here, it's the color that's not any of those two. And, again, it's forced by playing around.

OK. So that means we have a figure here about how you actually use this crossover gadget because there's this issue of identification, which is little bit subtle. So if you have an edge that's crossed by a bunch of edges, you intuitively want to stick this into each of the crossings.

But because this is copying the value here to here, it's really just like taking this vertex and pushing it to the other side of the edge. So when you throw in this crossover, you want to identify the left vertex of the crossover with the original vertex on the left side, but not identify it on the right side. Therefore, overall, there's still one edge connecting x and y because this is essentially a copy of x, but you still need that edge to connect to y.

So you don't want to identify both sides, you don't want to identify on neither side, because that would be two edges. Identify on one side. And it's like a vertex cover, but you just pick one side for each edge arbitrarily. And that is planar vertex 3-coloring not bound to degree yet. OK?

Next reduction. This is in the same paper. Here's how to simulate high degree. I mean, it's pretty intuitive. Once you have the ability to copy color, you can use it to get high degree. OK, I'll talk about the actual degree bound in the moment, but let's say we're aiming for max degree 4. This one, you can actually argue in a very

simple way. So here's a little gadget. I claim it makes three copies of this color, or two copies of the color at x.

So you've got these three vertices. One of them's blue, one of them's green. Doesn't matter which is which, but then this vertex must be red because of that triangle, and then this vertex must be green because of that triangle, then this vertex must be blue because of that triangle, this one must be red, and this one must be red. So that's a really easy one to argue.

And so, this is not very interesting, because we made two copies. This will simulate a degree 3 vertex, which we don't worry about. But where it gets interesting, if you just string a bunch of these together here, we end up with five copies of a single color, and so you can connect with a single edge out here, let's say. I guess you could even afford two edges there.

In particular, we can use this to simulate one vertex of degree 5 and we will end up with max degree 4. Degree 4 Because some of these vertices have degree 4. Actually, most of them do.

AUDIENCE: Could you have just used the crossover gadget as a high degree gadget?

PROFESSOR: Oh, in a circle. Yeah, that would also work, I think. Do you have a problem?

AUDIENCE: There are degree 7 vertices when you pit together two of these? You've moved--

[INTERPOSING VOICES]

PROFESSOR: Degree 6. Yeah. Well, OK. So that will give you max degree 6, and this gives you max degree 4. Yeah. If we do is after the crossover gadget, then the crossover gadget will become happy. So what's the conclusion? Planar max degree 4, 3-coloring is hard. But be careful, max degree 3, 3-coloring is easy polynomial time unless your graph is k4.

So one counter example. This is always possible. This is called Burke's theorem from 1941. So in general, if you have max degree delta, there's a delta coloring unless a couple of bad things happen. Odd cycle or a complete graph. So that's

cool.

AUDIENCE: What do you mean if it's k4 and then it doesn't work?

PROFESSOR: k4 requires four colors.

AUDIENCE: But then when finding whether or not there's--

PROFESSOR: But k4 has max degree 3.

AUDIENCE: [INAUDIBLE] polynomial figure out if there's a 3 colorable?

PROFESSOR: It's polynomial in all cases, but I'm saying every max degree 3 graph is 3 colorable except for k4. So the decision problem is, am I k4. Or I guess, am I not k4 would be the 3-coloring problem.

AUDIENCE: The word planar isn't up there.

PROFESSOR: Right, even without planar, it's polynomial. Don't need planarity for that algorithm for testing for k4ness. And I didn't check, but I'm pretty sure. Usually once the decision problem is easy, also the actual coloring algorithm is easy, but I didn't check. I assume there's a polynomial coloring algorithm, not just a decision algorithm, but we should double check before you cite that result.

AUDIENCE: You can [INAUDIBLE], make yourself a [INAUDIBLE] gadget and then try probing things.

PROFESSOR: Oh, right. You could reduce the decision problem to the actual coloring problem. You can test whether two guys have the same color by a bunch of probes. OK, so what? Why graph coloring? There aren't a ton of proofs that use coloring, because usually 3SAT is simpler because it only has binary values, but there are situations where coloring is helpful. I have one here that we used in the context of pushing blocks.

We covered Push-* is hard, we covered Push-1 is hard, we covered these two proofs in that lecture, and then implied PushPush, but we didn't talk about Push-X.

Push-X was the version where you're not allowed to revisit the same square twice. Like every time you leave a square, it falls down into the abyss behind you, so you can never step on that square again.

So our hardness proof for that uses coloring, and I think it's instructive not because I care especially about Push-1X but it seems like a general approach to representing color. So good to see the gadgets. First, a simple idea is that, if I have some planar graph I want to take an Euler tour-- a tour that visits every edge exactly twice-- and I want to do that in a planar way.

So the idea is, I don't want my tour to come down this way through a vertex and then later come through this way in a vertex. That's a meaningful thing because I know that planar coloring is hard, so I'm going to reduce from planar max degree 4, 3-coloring.

So I've got a planar graph, I draw it in the plane, and then relative to that drawing, I want to make sure there's no crossings in my tour that visits every edge exactly twice. These always exist, simple inductive proof, start with one vertex, visit the star around it, and then just start gluing these things together. In the inductive way, you will get a planar Eulerian tour, standard trick.

Now we're going to use that tour. So here we see an actual graph in the dashed lines and then we see the Euler tour in the red lines, and the red path is essentially the tour taken by the robot that's pushing the blocks around. So it's walking around in some direction somewhere-- I think not drawn here. Maybe that.

I'm going to break this apart and I'm going to say, the robot starts here, there's an obstacle here, and the goal is to get here. So your sole purpose in this puzzle is to start here and get all the way around the loop, and you're just going to be able to go along the red path. But there's some interactions. There's the blue arrows, and then there's the green, wiggly lines and that's all you'll need.

One of them is an equal constraint, and one of them is not equal constraint. And the idea is that when you visit a vertex-- so let's say you start here at u-- I'm going to

pick a color-- 1, 2, or 3, and there will actually be three red paths here.

Then those three red paths will interact with these three red paths to force equality. The wiggly lines mean equality. So I want that whatever color I've chosen here is the same as the color I've chosen here, because I want u to have one color, I'm only allowed to assign one color to u. So we're going to look at how to do that equality constraint.

And then we have these types of constraints, which say that the colors are different, non-equal, because I want the color assigned to u to be different than the color assigned to v. And this path is coming from v so at this point the color that you're on, which of the red rails you're on, says which color is assigned to v. You want that color to be different from the color assigned to u.

Then you transition and say, OK. Now I'm going to start over and pick another color between 1 and 3. And then here, it's constrained to be equal to this one. So I think the arrows are delineating the transitions from the color. Here it should all be color v. Well, that's maybe not so [INAUDIBLE], but somewhere along this edge, you're going to switch to thinking about the color v to thinking about the color of u.

You only need one color at any moment, so there will only have to be three parallel tracks for each of these things. That's the high level picture. Let me tell you actually do it. So here's this left part expanded into a slightly more detailed picture. Still a bunch of details to be filled in, but we see still the non-equal and the equal gadgets, just like before, but here I've explicitly shown the three tracks.

So we were looking at this edge before, so let's say here. Here's where you choose the color for u. So we're going to use what we call a fork gadget. We've maybe seen something like this before. When you come in, you can choose one of these three paths. Once you choose, you can't unchoose. It's a one-way gadget.

So let's say you choose path one, that means that this vertex u is colored 1. Then you're going to have some equal gadget-- these three paths are connected over here-- but you'll be forced that this path is the same as the one chosen over here.

We'll get to that one later. Then you go over here, you're forced that among these three paths, you're different from the three paths over here, because that's going to be the color v.

And then, you now want to switch from the u color to the v color. So I'm just going to have some one-way gadgets here that coalesce these three wires into one, and then whatever's next-- Let's say this is next, v was a leaf. Then you have a fork, again, to choose the color for v. So it's kind of weird, you get to choose the color for vertex several times.

Here we're choosing it for u, here were choosing it again for u, but they're combined together with this equal gadget, so those two choices are forced to be the same. That's the idea. And then the non-equal gadgets are doing the actual coloring constraint of no monochromatic edges. So it looks complicated, but all you really need are these things. Well, and the fork and the one-way.

Let's do the fork and the one-way, because those are easy. We've basically seen them before. Pretty sure we did this one-way gadget. You can go from a to b, but not from b to a. Cool. So that means when I come through here, I can't back up along some other path, that would be bad.

And the fork gadget, this is a two-way fork. You come in from a and you can either push this in and then choose to go to c, but then you'll never be able to go to b, or you can push this down and then go to b and you'll never be able to go to c. So when you make this choice, you can't undo the choice. You just chain two of these together to make a three-way choice. And the one-way is to prevent you from going back along one of these paths.

So that's the easy part. Then what about the equal and the non-equal gadgets? These are complicated, but in the end, they reduce to some very simple gadgets. So let's start with the non-equal gadget. So we have three possible color choices for one edge, for one vertex, and we have three possible choices down here for the other vertex.

Together, those vertices should form the edge e, and this is written as e going one way or e going the other way and coming from one vertex and coming from the other vertex end. And what we want is to forbid blue-blue, or to forbid red-red, or to forbid orange-orange. And because this paper was also printed in black and white, the dash patterns also duplicate the color information. So even if you lost all color, you can distinguish these types of lines.

So we need this gadget, which we called a manned gadget, but you should not have both of these. And there's a couple different cases, depending on the orientation. But, for example, if you go from a to b, you have to push this down. Which, because this is Push-1, you only have strength one, you're not able to push up.

You're not able to traverse cd anymore if you do a to b. This is the symmetric version where, if you first do b to a, then that prevents cd traversal. And its symmetric, so vice versa, OK? So that's the gadget that plugs in here.

And so now we know that, if we follow the path along 3, the blue path for this vertex, we won't be able to traverse the blue path. And so whatever choice you made here at the fork has to be different from this choice.

It's kind of a fun, non-local effect. And it doesn't matter, whoever makes the choice first will block the choice for the other guy. If you had multiple robots simultaneously doing things, then it would get tricky when there's two robots right here at the same time, but probably even then, it would work as long as everything eventually gets traversed. OK.

So then we also need some limited kinds of crossovers to make this happen, because we need to take this orange path. I don't know why it's orange. It should be green, but there you go. Bring it down here and then bring it back up. So that's going to require the orange path to cross the red path and the blue path. Good news. We know that only one of these will be traversed, because the fork gadget has that property.

If you end up following the one path, you know that 2 will not have to be traversed,

so this is what we called an xor crossover back when we were doing pushing blocks. It's a crossover that works as long as you only are visiting it once, one way or the other way. So, for example, if you come-- And it's also uni-directional. So if you come from a, you can push that down, but then you won't be able to go to c or d, and then you can leave through b.

And from c, you can push this over, go to d, and those are the only cases we need with various reflections and rotations. We either go from up top to down here, or we go from right to left here through that xor crossover. So you just plug those crossovers in and you can get each of these paths to where you need them to be, and you know that they'll work because you won't have to do both of those traversals, and there you go.

So in general, as long as you have a NAND gadget and an xor gadget, then you can do this to make a not equal gadget. So there's a lot of pieces here, but in the end, it reduces to very few things. We had a one-way, a fork, an xor, and a NAND. If you have those things, you can simulate 3-coloring in this planar way.

Now, I didn't cover one more gadget, which is the equal gadget. It's just a more complicated version of the non-equal gadget. So you need to prevent this one from being blue when the bottom one is orange, you need to prevent this one from being blue when the bottom one is red, and you need to prevent this one from being red when that one's orange, you need to prevent this one from being red when that one's blue. And all the pairwise things you don't want to have happen, just make them not happen.

So you can imagine, of course, much more general gadgets than this. We're probably doing much more than 3-coloring, but again, all we need are the xors and the NAND. So that proves Push-1X is NP complete for free, and this approach has been used by a couple of papers.

So here is another one, Push-1G. This is pushing blocks with gravity. So imagine-- this happens in a lot of games-- when you're pushing a block, the block will fall if it ever has a hole below it. Let's say that you don't fall though, or you could do lots of

jumps, or flying, or whatever. You can do something to avoid. I don't think we'll need any big jumps for this to work.

So there are lots of video games that follow these kinds of roles. Here's a one-way in that model, you just push this block over, it will fall. So you can't push it the other way, but once you push it forward, it's open and we're going to denote that with an arrow.

This is Eric Friedman. And here is an xor crossover. This is kind of fun. If I come in this way, I push this guy over, it falls down blocking that path, but I can still go through here, and if I push this one over, the block falls, and I can go this way, but I'm blocked from going this way or that way. So that, again, works as long as I'm only doing one of the two traversals. We have a fork gadget, which is familiar.

If you're coming out from here, you can push this over and then you'd be prevented from going the other way because you, again, only have strength 1, and symmetrically the other side. Then the NAND ga-- Yeah?

AUDIENCE: For the xor crossover, you said you couldn't-- When you're going from the bottom one, you push that over and it falls, and then you can't go back up because you couldn't push that block?

PROFESSOR: Yeah. So I maybe need a wiggle here.

AUDIENCE: The arrow is already representing one-way gadgets?

PROFESSOR: Good. That's already here because the arrow is a one-way. I forgot the notation. Yeah. It's not the input and the output. This is an actual one-way. Good. OK. And here's a fun thing I learned from reading this paper, you don't even need a NAND gadget, because you can simulate a NAND gadget with xor crossovers.

This is kind of like Jason's idea of using the crossover gadget to make copies. If you traverse through this way, and through this way, we know that's possible, and if you have a set up where, by going through a crossover you block the other traversal-- so that would be a true xor crossover-- Once I've pushed this one over, if I tried to

go through this way down to here, I would hit the block here that had fallen from there and vice versa.

So if it actually prevents the second traversal, and in Push-1X, it also prevented the second traversal because we had non-crossing paths. We weren't allowed to revisit the same square, so there it was really trivial. Here with gravity, it's a little more subtle, because the second time you come through maybe you could go back that way.

But the key thing is that the second time you go through, you won't be able to go through in the regular crossover way, and then you just string two of them together and you can either traverse this way or traverse this way, and each one will block the other. So in the end, you just need a one-way, fork, an xor crossover, and some kind of notion of sequential traversal and you can simulate 3-coloring.

So this is fun. I haven't seen this technique used too much, but in a few papers and maybe we can use it for more. Cool. I didn't mention, but the Push-1 proofs that we saw before, lecture four or whatever, revisit the same square many times. So we can't use those proofs for Push-1X. All the gadgets break.

The next problem I want to talk about is a little different, another graph problem. It's called graph orientation. Kind of like edge coloring, but the colors differ on each side. So a graph orientation. This is a pretty recent problem. It was introduced in 2012, but I think it's very cool and deserves much more study.

So you're given an undirected graph and you want to find an orientation-- orientation means for every edge, you give it a direction-- satisfying certain vertex constraints, and there are three types of vertices, three types of vertex constraints. They are 1-in-3, 2-in-3, and 0 or 3. They all end in 3.

I didn't mention, this is a 3 regular graph. Every vertex has degree 3, and so if you look at an orientation, every vertex either has 3 in, 0 out, 2 in, 1 out, 1 in, 2 out, or 0 in, 3 out. And this problem is NP hard. Here is a simple reduction from 3SAT. That's right, 1 in 3SAT.

So we're going to have a variable gadget, which is just this loop of a cycle in the graph. And for each of these vertices that's a solid black circle is a 0 or 3. So that means in any solution-- here's a solution-- you're going to alternate between all in and all out, and then all in, all out. So this is going to represent x and x-bar.

What the parity of that cycle is is up to you, you could either make all the x's all in or all the x-bars all in, and that corresponds to x being true or false. I think in means false in this case. And then for the clauses, we're going to use a 1 in 3 gadget.

It's because that's a 1 in 3SAT clause. We want exactly one of these three things to be in coming to the clause. That gives us exactly one of them being set to true. Now, here we're allowing negations. We know that's not necessary for 1 in 3SAT, but we can in this proof, so they're drawn here anyway.

But to make it a little bit weird, one issue in this style, or in this reduction, really I just want a whole bunch of copies of x. But I get all these copies of x-bar, and I need to put them somewhere. Every vertex has to have degree 3, so this has to go to something. And so their solution for doing that is for every clause, we also build the anti-clause and make it false.

So we want exactly two of the negated versions of these variables to be set to true. That's the same thing as exactly one of the positive forms of them being true. Yes. So we just negate everything in the clause, and then wherever we use x, we also use x-bar and vice versa. So that guarantees that we use up all of these instances.

If there are k occurrences of x, then we'll make k occurrences of x-bar and make them all used by the corresponding anti-clauses. So this is why we need those three types of gadgets. If we're doing a problem in the plane, we'll also need a crossover for this to work. But at this point, it's just a graph problem. OK? So that's cool. And this problem was introduced in order to solve a packing problem.

A little bit of history, some time ago, I think the '70s, there was a paper about, if I give you polygon some orthogonal polygon shape with holes in it, and I want to pack as many, say, 3 by 3 squares in the polygon, that's NP hard. If I want to pack as

many 2 by 2 squares, that's NP hard. That was a later paper, and so how much smaller of a square can you make?

Well, a 1 by 1 square, that's pretty easy, in a grid polygon. How many 1 by 1 squares can I pack? The area. In a grid polygon, how many 2 by 1 rectangles can I pack this way or this way? That's maximum cardinality matching. So the next thing left is trominoes. Three squares, and they could be in an L-shape or they could be in an I-shape. Here, we're thinking about both problems separately.

So suppose you have a whole bunch of L-shaped packages that you want to fit into this weirdly shaped warehouse, that is NP hard. It's even hard in the exact packing case. There will be no holes-- Sorry, there will be no gaps in this packing. Every unit square will be filled, and I think that's quite particular.

Those other proofs of packing, the 2 by 2 and the 3 by 3 squared into a polygon, I should have shown them, but they're from 3SAT, planar 3SAT. They leave gaps all over the place. Here, you don't leave gaps and I think it's somehow fundamental to this graph orientation business.

So here is the idea of for an edge in that graph orientation problem. It's basically a rectangle with a bump every other square. So this is a big hole. You're not allowed to put anything down there. And the idea is you can either have the L's all pointing to the right or all pointing to the left.

And so you think of these as kind of the communication position. One of these will correspond to the edge pointing to the right, one will correspond to the edges pointing to the left. And you can build a turn gadget. It works pretty cleanly. It doesn't matter whether this is covered by that guy or covered by that guy. It behaves the same as a regular wire. So again, it's either this or this is occupied, exactly one of them. That tells you the orientation of the edge.

So that's an edge. You can also build a crossover. This is a little more casework. But again, this is occupied if and only if this is not occupied, and this is occupied if and only if this is not occupied, and all combinations are possible. We just need to

check that there is a valid packing no matter which of those choices you want. Either left and top being empty, or left and bottom being empty, or top and right being empty, or bottom and right being empty. Those are the four cases. Cool. Crossover.

Now we need the 0 and 3, 1 and 3, and 2 and 3. So instead of a single 0 or 3, they built two in a row. I don't know if this is required, but it's fine for the proof, because in the proof it was always XI, XI-Bar, so there are always 2, 0, or 3's in a row, and there it is. So in this setting, essentially either the left two will have bumps and the right two will not have bombs, or vice versa, the right two will have bumps.

This corresponds to these guys all being out in all three directions, and these guys all being in all three directions, or vice versa, and these are all out and these are all in. So that's the x or x-bar. And this gadget does it. Again, check the cases. We have exactly 1-in-3. If exactly 1 of these is in, then this will be packable, and otherwise it won't. That's the part that's hard to show but you check it.

And if you want a 2-in-3 gadget, you just add a couple more blocks right there and it works. So that's it. Once you have all those gadgets, you can do graph orientation and then you're NP hard. That was for L-trominoes. Do the same thing for I-trominoes. This is hard to see where the bold lines are, but if you look and if you download the slide, it's probably a little easier to see, but, again, you can build.

Here, an edge gadget is fairly straightforward. Just wiggle a path and you'll either have it-- These wiggles are in order to guarantee-- There's actually three possible parities you could have. Sticking out by 1, or sticking out by 2 with an I-gadget. This will force it to be just 1 or 0, because we don't want it to be 2. And then the crossover is ugly, but it works.

And the 0, 3, and the 1-in-3 are not too hard. And do I have the 2 in? I think this is a 2-in-3, I didn't label it. So, presumably, for any polynomial shape you want, you can build such gadgets so that we don't have such a theorem, but I think this is neat. It's a little different from a lot of the other proofs, similar to one of the problem set problems where we had some constraints on the neighboring things.

But instead of about constraints on the neighboring colors of the vertices or whatever, or the truth assignments in the vertices, here, it's an edge direction. So the edge directions are interesting because, first of all, there's only two choices, but also that what you see on one side is the opposite of what you see on the other side. Questions?

So that is graph orientation, and I have one more set of problems I want to talk about. They have many names, but the general family is usually called something like linear layout of a graph, and it's a bijection-- let's call it f-- from the vertices to 1 up to the number of vertices.

So in other words, I'm thinking of a one dimensional diagram and the coordinates I have are 1, 2, up to the number of vertices and I just want to put exactly one vertex at each of these spots. So I basically want a permutation of the vertices, and then I slap them down in that order, and then I measure something about the quality of that layout based on the edges. We haven't involved edges yet.

And there are many different measures you might consider. Here are many of them, but without definitions. So let me tell you some definitions. So the first one, of the earliest to be considered, is called bandwidth. Bandwidth is, if you look at an edge, one endpoint of the edge gets mapped to one coordinate and the other endpoint gets mapped to another coordinate.

You can measure the length of that edge in the embedding. I shouldn't call it embedding because it's non-crossing, but in the layout. So for some edge vw, you can measure the length of the edge. If I want to minimize the maximum length of any edge, that is bandwidth. Why is it called bandwidth? Anyone know? No one does matrices anymore, or numerical linear algebra, so I'll tell you why.

If you have a matrix, let's say, of 0's and 1's-- that's a graph, of course-- and all of the non-zero elements are in that band, then we call this bandwidth w. If it's 0 out here and 0 out here. Why are these interesting? Because then if you run Gaussian elimination, you always stay within the band. So this is some of the early approaches to solving sparse linear systems.

If you can get all of the 0's into the corners, then you can focus here, especially if your tri-diagonal is a common case. Anyway, that's called bandwidth, and what this problem is saying is I'm allowed to permute the rows and columns of my matrix in order to minimize how many diagonals I need to use. So that is permutation to minimize bandwidth. It would be great, except it's NP hard.

It's NP hard even on trees of maximum degree 3. It's NP hard even on caterpillars. Almost caterpillars. A caterpillar is a graph, something like this. I think this is caterpillar with hair length at most 3, so these may be paths of a length 3, maybe they even branch a little bit, but every vertex is within distance 3 of a single path.

So even for such graphs, when mapped into matrices, this problem is NP hard. Lots of cases are NP hard, even for good graphs. You might think of that as more general, but it's not immediately implied. OK. Cool. That was bandwidth. Next one, which I've seen used in a few different hardness proofs, is minimum linear arrangement.

Almost the same problem, but instead of taking the maximum edge length and trying to minimize that, take the sum of the edge lengths and try to minimize that. That's minimum linear arrangement. We will see a reduction from that in a moment. It's NP hard, even for my bipartite graphs.

Cut width. This is, you draw all the edges as horizontal segments and then I come in with a vertical line and see how many edges can I cross. I want to find a permutation, so I minimize the maximums, you might call it stabbing width. The maximum number of edges that cross from the left side to the right side where I take the maximum over all notions of side.

I take the maximum over all choices of this x-coordinate, and I want to minimize that maximum. That is cut width. Why do I want to minimize the maximum as opposed to, say, minimizing the sum of those cut values? Because that's the same as minimum linear arrangement. If I minimize the sum of all of these cuts, that's the same thing as minimizing the sum of the lengths of the edges. So that's the same

problem.

So, OK. We got rid of one. That was cut width. It's hard for planar graphs, max degree 3, good graphs, lots of things. I'm going to skip mod cut. That's just a slight different variation on that definition. Next one is vertex separation. OK. This is a different way of counting. So here, I was counting how many edges cross, but maybe many of those edges come from the same vertex.

I don't want to double count those, let's say. I just want to count how many vertices on the left side have at least one edge that goes to the right side, and only count it once instead of three times in this picture. OK? Otherwise the same problem as cut width. That is also hard.

That problem is different if you look at the sum versions. You can account for every partition point how many vertices in the left have an edge to the right side, sum that over all of these x values, and then that is sum cut. All of these have been considered in various contexts.

Last one is edge by section. This is where you only look at cutting in the middle at v/2. So you want to balance partition all the things on the left, you want to have very few edges to things on the right. Or the vertex version where you want to minimize number vertices on the left and edges to the right half, but exactly half. v/2. v/2.

A ton of problems. I mentioned them so that if you ever run into a problem about ordering vertices on a line, you should look at all of those and see which one is the most useful. Sort of like we have 3SAT and 1 and 3SAT, and not-all-equal-SAT Choose the one that's easiest for you.

If you have some kind of ordering problem, choose the one that's easiest for you. Good to know that these are out there. They come from various applications. Cut width is studied a lot in graph theory, graph minor stuff. It's closely related to path width. Some of these problem-- The bisection comes from numerical linear algebra. Minimum linear arrangement comes, I think, originally from VLSI layout, chip design.

It's like a very simple version. If you just have a bunch of ports on the bottom and

you know that certain things need to be connected by wires, that's your graph, you want to minimize the total amount of wire stuff you have to-- minimize the total wire lengths. So that's minimum linear arrangement. Very simple version of some kind of VLSI layout problem. And there's this survey if you want to see all these problems.

There's one problem not on the survey because it's not about a graph, it's about a hypergraph, but it's a useful one. Among ordering problems, it's the one I know the best. I've tried to use it a couple times, but rarely have I succeeded in getting an actual NP hardness proof from it, but I will show you one, and it is called betweenness.

So in this case, I'm given a bunch of triples, a set of pairs of things-- it's not a graph-- of the form y is between x and z. And what that means is either it's between x and z in that sense, or it's between x and z in the other sense, where either x is to the left of z or z is to the left of x, but y is always between. And here, of course, I really mean f of x and f of y. I mean them in the linear embedding.

OK. So you're given a bunch of triples like this, and then you want to find a linear layout of your letters. So again, that's a bijection from 1 to n, and such that these all hold. So there's no objective function to minimize here. It's just you want each of these things to be true. So this is nice, because it's a pretty clean constraint and yet, it's hard.

If I, for example, was just giving you-- If I gave you a bunch of inequality constraints, like x is less than y. That's easy. That's sorting a partial order. But here, you have a little bit of ambiguity. You don't know how x and z relate. You just know that y is in between them. That's enough to get hardness. So I'm not going to prove any of those problems hard, but I will show you two examples of hardness proofs.

First one is going to be from the minimum linear arrangement. In case you haven't already memorized all the problems I just described, let me remind you. Minimum linear arrangement was minimize the sum of the edge lengths. That was the second problem I described, that was like the VLSI layout. Minimize the sum of all the red lines. Find the permutation that does that.

So we're going to reduce it first to a problem called bipartite crossing number. It's a bit of a weird problem. It's mostly a stop gap on the way to another problem, which is crossing number. Suppose you're given bipartite a graph, which is hard to see in this picture. Given a bipartite graph, and you want to draw in the plane. So I have some bipartite graph, I want to draw it in the plane in a special way.

I want all of the vertices in one side of bipartition to be on a horizontal line. I want all the vertices in the other side of the bipartition to be on a parallel horizontal line. And all the edges are in between. They're straight lines. And I want to minimize the number of crossings. So it's this very specific kind of graph layout problem, but minimizing number of crossing is clearly good. We want to draw as planar as possible.

So here is a reduction from minimum linear arrangement to bipartite crossing number. So in this problem, we're given a general graph, not necessarily bipartite. We want to convert it into a bipartite graph. How do we do that? Make two copies of every vertex.

OK, so for every vertex in the minimum linear arrangement problem, we're going to make two copies called bottom one and top one, top two and bottom two, bottom two and top two, and so on. So there's n top vertices and bottom vertices, and I'm going to do two things. One is connect a whole bunch of edges. And a whole bunch means e squared, and between bottom I and top I. I just do that.

That will basically force this kind of layout where, basically, the order on the top has to be identical to the order on the bottom. Because if ever any of these two bundles of edges crossed, you would get e to the fourth crossings. And so if you ever want to get less than e to the fourth crossings-- and that's what we will hopefully do-- then these must appear same order on top and bottom.

But we don't know what the order is. You can still permute the bottom, just correspondingly permute the top, and all be well as far as these edge bundles go. So that's good, because it has exactly the flexibility, we have exactly one

permutation on n things. That's what we want to represent with minimum linear arrangement.

Then the only other thing we do is add in the edges of the graph. But in the minimum linear arrangement problem, the edges are like from vertex I to vertex J. We're going to make that connection from bottom I to top J. There's this choice, but it doesn't matter which is which. So the idea is, then, that edge will cross a bunch of bundles. The number of bundles it crosses is the length of the edge minus 1, I think. I'll ignore these additive constants.

You have to be careful to make sure everything adds up the right way. If you have a link 0 edge-- We never have the link 0 edge, because vertices map to different places. If you have a length 1 edge, you won't cross anything, so 0 crossings. If you have a length 2 edge, you will cross exactly one bundle, and you pay e squared for that.

In general, it will be something like e squared times the sum of the lengths of the edges. Not exactly. You have to subtract off some things, but you just compute what that is. It will always be basically some fixed constant times the minimum linear arrangement cost, which was the sum of the lengths of the edges plus some fixed constant.

And so you can solve the bipartite crossing number with a given specified number of crossings if and only if you can solve minimum linear arrangement with a specified sum of edge lengths. Question.

AUDIENCE: What about the crossing between edges?

PROFESSOR: Yes. There's also crossings between edges, and you have to count them.

AUDIENCE: [INAUDIBLE] because you can't get more than e squared with this in the normal case?

PROFESSOR: Yes. So that will be in the noise. The bulk of the number of crossings will be from crossing the bundles with the single edges. You don't want to have bundle-bundle

crossings. Those, you can never afford. So you're basically counting bundles versus single edges. The total number of single edge crossings will be strictly less than e squared, and so it will be strictly less than a single guy crossing a bundle.

So you have to inflate. It's not an exact counting, because you don't know how many of those single edge crossings you're going to get. So you have to add almost e squared. I have the exact count here. I don't know how interesting it is. But what they wrote, it's e squared times k minus e, plus 1 minus 1 is the exact number in the paper. k here is the sum of the lengths of the edges in that problem.

This is, I think, the minus 1 per edge, that gets multiplied by e squared, and then we're basically adding e squared minus 1 at the end. So almost e squared to allow for any number of crossings between the single guys. It does make a lot of sense. OK. So that was bipartite crossing number, but the more natural problem, I would say, is I give you a graph, I want to draw on the plane with fewest crossings. That is, crossing number, and it's a reduction from the previous problem.

So basically, you can force these vertices to be on a horizontal line, and these vertices to be on another horizontal line, and to only have edges between here and here by adding huge bundles out here to basically prevent anything from going out there,

Now we're given a bipartite graph. We want to draw it in this kind of way, minimizing number of crossings in between. And so this will turn that into a general graph. It's actually still bipartite, but now the planar embedding is forced, more or less. I mean, it's not an embedding. I keep using that word. The planar drawing is more or less forced.

You can show these guys have to be in this kind of topology, and then there's some crossings in here, but none of these edges could ever cross this, because this is way more than the number of crossings in the input graph. Yeah?

AUDIENCE: What about not multi-graphs?

PROFESSOR: Not multi-graphs? Good question. I assume you can split these things up, or maybe

subdivide the edges, or some trick, but I have to be very careful. I don't know for sure. Definitely not mentioned in this paper. OK. So that was crossing number, bipartite, and unconstraints.

And I have one more sketch of a proof, which is mostly for fun, and I get to use my favorite phrase, how to kill log n birds with one stone, or order one stones. So suppose you have a Rubik's cube, but 3, by 3, by 3, that's easy. So you have an n, by n, by n Rubik's cube, like this 7, by 7, by 7, v cube, and moves are-- I assume you all know how a Rubik's cube works.

You can rotate in each of these directions-- and I hope not to mess it up too much-- and your goal is to get to the state where it's all solved. Now, usually, someone's mean, and they mix it all up, and they just give it you like, OK solve it. I don't just want to solve it, I want to solve it with the fewest moves, because it's polynomial time to solve it at all. I want to solve it in the fewest moves.

So if I'm given this position, I want to know it's only one move away from solved. We do not know the complexity of that problem. Let me first tell you a nice way to think about this problem is actually in the 2D case. So these are not built super large, but, again, I can rotate either a row or a column.

So at a high level, you can think of a picture like this. You have a red side and a blue side, and there's a certain-- If you think about where this square goes, it can go to this position, to this position, and this position. In general, a little cube can only go to four different spots in 2D. On the 3D cube, there's 24 spots it can go to, because there are 24 automorphisms on the cube. It's just a lot harder to see, but it's essentially the same thing going on.

So here's a sort of thing you could do. If I flip this column, these guys go over to here, and they flip upside down. So whatever was red here becomes blue down here, and it gets reflected across this line. So when I do this move, I get this pattern for that row, that column. If I also do it at this column, I get that pattern. OK.

So now maybe I do these two columns. They completely flip, become all blue. Now

maybe I do these two columns and I get this picture because these red spots become blue up here, and so on. And then if I flip these two rows, hey, I solved the puzzle.

And in general, if you look at a [? cubee ?] and the four other [? cubees ?] that it can go to, they have some pattern. There's a constant number patterns they can have. For 2D it's-- I forget-- like 10 or 20 different patterns. In 3D, it's, like, a lot more. Billions or something. Anyway. But it's constant, even for an n, by n, by n cube.

So you can characterize for each such pattern what it needs to be solved. So these [? cubees ?], for example, need a column, row, column, row. That's what we showed. We flipped its column, then we flipped its row, then we flipped its column, then we flipped its row. That solved it. That's exactly what they need. And in minimal solution, you will do that.

But what you see here is, suppose I had a big grid of them. I could do all the columns, then all the rows containing them, then all of the columns and all the rows. If they were in the same initial pattern, I get a big savings in how quickly I can solve it. If I have an x by y grid of identically oriented [? cubees ?], I can solve it in about x plus y moves instead of x times y moves.

And this is something we used to prove that you can solve an n, by n, by n Rubik's cube and n squared divided by log n moves, in the worst case. So you can kill log n birds with one stone. There's always such a grid of area roughly log n. But here, I want to use that idea for hardness. In some sense, that was to give you some intuition.

Sadly, we don't know whether this problem is NP hard, minimizing the number of moves, but what we do know is that if some of the stickers fell off your cube, then it's NP hard. So the white things here don't care. You don't care what state they end up in. They're sort of wild cards, so it could be the sticker came off or maybe it just changes its color to whatever's correct at the moment.

But some of the stickers are still on and they have to be solved, and this is a reduction from betweenness. This gadget will be solvable in a certain number of moves if and only if the first time you make the x2 column move is between the first time you make the x1 column move and the x3 column move. I think these are in the situation that they want to do column, row, column, row, something like that.

So each one's going to get used twice. It's a matter of how you intersperse those orders. So it's an ordering problem, and this ends up being betweenness. I will not go through the proof. It's quite tedious. So this column is going to get used many times in the reduction. Basically, you just work in the upper left corner of the picture because the other quarters move similarly.

You introduce some extra rows and columns that are specific to this betweenness gadget, and if you want to have more betweenness gadgets, you add more such columns and rows in this pattern. As long as they're sort of off diagonal from each other, they won't interact, hence you end up with a big betweenness reduction. So that's it for today.