**PROFESSOR:** All right. So today we start a new topic, but in the same spirit of PSPACE completeness of puzzles. We're going to talk about a general theory called Constraint Logic. This was the topic of the Ph.D. thesis here by Bob Hearn, which later turned into this book-- *Games*, *Puzzles, and Computation*.

And constraint logic is a big theory. But today we're going to focus on one particular aspect of it called nondeterministic constraint logic. Nondeterministic like NP. And that is specifically about puzzles and PSPACE completeness. So non-deterministic constraint logic-- usually called NCL. And it's a very useful infrastructure for proving PSPACE hardness results, for puzzles, originally motivated by sliding blocks, but we will get there in a moment.

You've seen it very briefly in Lecture 1. I'm going to go through it again more slowly and clearly and define everything. And then we'll see lots of hardness proofs based on that. So we start with the notion of a machine. We won't use this term too much. But the idea is we start with an undirected graph. Think of this as a model of computation. So your computer is an undirected graph of red edges and blue edges. Red edges have weight 1. Blue edges have weight 2.

And then a configuration of that machine-- or a constraint graph in total-- is an orientation of that graph. So it's really a directed graph with red and blue edges. And you have to satisfy the constraint that at every node, the total incoming weight should be at least 2. So either at least one blue edge or at least two red edges.

So this should be a satisfying assignment. This one has two red edges. This one has actually one blue edge and an additional red edge coming in. But in general we have-- so, a constraint graph. It's going to be a directed red/blue graph satisfying, I'll

call it the inflow constraint, that for every vertex, the total incoming weight is greater than or equal to 2 where red is a 1 and a blue is a 2.

And then we're interested in reconfigurations of constraint graphs by reversing one edge at a time. So let's look at a vertex here. This guy has total incoming weight of three-- one from the red, two from the blue.

And a move we're allowed to do is say, reverse an edge like this. So now we have a total incoming weight of 4. 4 is also greater than or equal to 2. So that's a valid move.

Now we could also reverse this edge. We couldn't before. If we had done it in the beginning, there would only be a weight of 1 coming in. But now that we have these two units of weight coming in, we can redirect this-- at least local to this vertex-- we can redirect this guy. It'll only make this vertex happier.

The validity of this move depended on what that vertex looked like. So in general, we start with some constraint graph. We're going to do a sequence of moves. At all times we should have a valid constraint graph, meaning you satisfy the inflow constraint.

So those are the rules of the game. And at each move, or at each time step, you can make any valid move you want. That's the nondeterministic aspect. And that mimics most puzzles. You're not told which move to do. You get to choose which move to do. Guess which move to do, if you like.

So the NCL problem is, I give you some crazy network like this. And I want to know, can I reverse a particular edge? That's one version of the problem. There are actually two decision problems we can think about. That one is the most useful. Or most common, I should say.

Can you reverse a specified edge given the constraint graph? Meaning is there a sequence of moves-- and the last move is reversing the target edge. Another problem would be, can you reach a desired other constraint graph? So I give you an entire configuration and another orientation of the graph that's consistent. And to in

some sense, that gives you more information. I don't just say I want to flip this edge. It will actually be, I want to flip this edge and no other edges is going to be the typical setup.

But in general, I give you one configuration and another. And I want to know, can I get from here to here by a sequence of single-edge reversals. Yeah.

**AUDIENCE:** Does the graph [INAUDIBLE] degree 3?

**PROFESSOR:** It will always be max degree 3. So you can assume that. In fact, we will show both of these problems are PSPACE complete for three regular graphs. Although what I've drawn here is not three regular. It has some vertices in degree 1. You can actually assume max degree 3. And you can assume that there are only two types of vertices-- and those are red, red, blue and blue, blue, blue.

So even just for these two types of vertices, both of these problems are PSPACE complete. Yeah.

**AUDIENCE:** Do degree 1 vertices also have to satisfy the inflow constraint?

**PROFESSOR:** Yeah. This would not be valid. We'll show how to simulate degree 1 vertices with or without the inflow constraint. Yeah.

**AUDIENCE:** Is the problem given the graph, find a valid configuration?

**PROFESSOR:** That's a different problem. Given an undirected graph, find a valid configuration. That problem is NP complete. We'll also prove that.

That's called constraint graph satisfiability. So that's an analog of SAT. So this is, given undirected red/blue graph, find a valid orientation. So that's a good question.

At present, there aren't very many NP hardness reductions that use this problem. But I think maybe there should be more. It's a need.

This was not our original goal with constraint logic. So we sort of forgot about it until recently. We thought it would be cool to do more.

**AUDIENCE:** So the separation suggests that you can't get from any valid configuration to any other valid configuration?

**PROFESSOR:** Right. So in particular, for lots of different pairs of configurations of the same graph, you cannot find a path from one to the other. And deciding whether you can is PSPACE complete. I mean, that implies there are lots of no answers.

**AUDIENCE:** So what prevents you from being able to do it intuitively?

**PROFESSOR:** Hopefully it will become more and more obvious. I don't have a great intuition why it's hard, other than it's hard. I mean, you tend to get nice path connectivity in continuous spaces. This is a very discrete space. And there's a lot of hard constraints that's like rough intuition.

**AUDIENCE:** You'll probably get into this later, but can you give us some brief motivation of why we care about this problem?

**PROFESSOR:** The motivation is a lot of puzzles can easily simulate these kinds of pictures. And in particular, you basically need two gadgets. And we'll also prove that this is hard for planar graphs. So you don't even need a crossover. For planar red/blue graphs, with just these two types of vertices, the whole problem is PSPACE complete. So the ultimate motivation is to prove your problem PSPACE complete.

If it falls into the category of games like this, you just need two gadgets. And you get a proof by two pictures. So that will lead to some very efficient PSPACE completeness proofs. Of course, to get there we first need to prove this theorem. And I'm going to do that first so you see where all this comes from, so you don't have to understand it. I think it's helpful in case you want to generalize it. OK.

So let me tell you about these two types of vertices. This one is called an AND vertex. And this one is called an OR vertex. Because if we think of the two red edges as inputs, and think of the blue edge as output, this picture is supposed to represent that 1 and 1 equals 1. And this picture is supposed to represent that 0 or 0 equals 0.

And if we do some more moves, like if I flip this edge, now I have total incoming weight of 3. That's still not enough to flip this edge, because 0 and 1 is 0 still. And I could flip it back. Maybe I flip the other edge. Also 1 and 0 is 0. Still can't flip this guy.

But if I flip both of them up top, if both of the red guys are incoming, then I can if I want to flip the bottom edge. But I don't have to right away. So I could go from here to here. But this is sort of a slow AND gate. I put in the two 1 inputs. I don't yet have a 1 output, so to speak, where 1 here is represented by 2. You get the idea.

OK. So let me write down the definition to make that slightly cleaner. So I'm going to define the activation of an edge, or an edge being activated.

An input edge is going to be active if it's incoming. And an output edge is active if it's outgoing. This is a symmetric. Incoming and outgoing always clear. Either you're going into the vertex, or you're going out of the vertex.

But just by labeling these inputs, I'm going to say these edges are sort of interesting if they're pointing in. This edge is interesting if it's pointing out. These correspond to the one bits in a Boolean logic. And what we say for an AND gate-- property of an AND gate-- is that the output can activate-- doesn't have to-- only if both inputs are active.

So in this language you can see that it's an AND gate. You need both of the inputs. The AND of the input should be active in order for the output to activate. And so that's why we do this asymmetric thing.

Another reason to do this asymmetric view of inputs and outputs is if you have two vertices and an edge between them. It was asymmetric to begin with, right? This edge from this guy's perspective is outgoing. From this guy's perspective, it's incoming. That's annoying to deal with. I want the edge to either be active or not.

And if this is the output edge of this vertex, and it's the input of this vertex, than this edge is active. And the other direction would be inactive. OK. But it's active from this guy's perspective. And it's active from this guy's perspective. So while this may

seem asymmetric, it actually makes the picture more symmetric. Yeah.

**AUDIENCE:** Can't have a red edge being the output, though. Right?

**PROFESSOR:** That's true. With these gadgets we never have a red edge being output. Imagine red being blue. OK. So that is active. And AND, you have sort of delayed outputs. You can think of this same vertex if you relabel these guys as outputs, and this guy as an input. It's just a perspective change. It's just changing terminology. So it doesn't actually change what happens. But from this perspective. You essentially are splitting, or fanning out, a wire.

So if you have a signal here of true-- so this is an active input now-- that both of these can be active if they want to be. They don't have to be.

But if it's false, then both of these have to be inactive. This is inactive. Both of these have to be inactive. And there are lots of configurations of this guy. So maybe some of them point in.

So the split. That's the same vertex that's a relabeling of who's active and who's inactive. The outputs can activate only if the input is active. And really, I should be saying if and only if. But I'm being a little concise here.

OK. So that's a SPLIT vertex. And then the other vertex type is an OR. So this is really the second vertex type out of three. Third out of three. Whatever.

So this of course looks very symmetric. I mean, there's no clear notion of inputs or outputs here. But if you define two of them to be inputs, and the other one to be an output, then that output is the OR of those two inputs. This edge can activate only if this edge is active, or this edge is active, or both. So it's an inclusive OR.

Probably have some animations. Like if I flip this guy, now this one can choose to activate. I could also have both of them in, or just one of them in. But in order for this one to now go back out and deactivate, this when would first have to deactivate.

So that's ANDs and ORs. You get some sense for why this is a Boolean logic. One

point to make at this point is that I do not have a NOT gadget. And in fact, NOT gadgets are impossible in this universe. Because we're always talking about outputs can activate, but they don't have to, it's up to the non-deterministic player to choose whether to activate an output, or when to do it, NOT is impossible.

Because a NOT would be something like, this edge cannot activate if this other edge is active. This output cannot activate if this input is active. That would be the idea. Maybe output is inactive if the input is active.

And that kind of constraint is impossible to represent with a lower bound on incoming weight. Maybe if you had an upper bound, you could do it. But with a lower bound, you're always happier to have inputs active. So if it worked with the input inactive that the output could activate, then it should also work with the input active. NOT gates are impossible.

So that's a little bit annoying, but it won't be too much trouble. We've in some sense dealt with that in other proofs. Before I get to these hardness results, let me tell you a few other vertices that are helpful, and can be simulated with ANDs and ORs.

So one of them is a choice vertex. This is sort of like an exclusive OR, in a sense. So two of these edges must be incoming-- in the symmetric picture-- two of them must be incoming. At most, one of them can be outgoing.

And this can be simulated by expanding this into a little triangle. So if this guy is outgoing, then both of these must be pointing to it. And once both of these are pointing out, then the blues must be pointing in here, because the red would not be enough to make either of these vertices happy. If this guy's making that guy happy. So if one of them is out, the other two must be in. We'll see how to use this in a second. But it's useful to have red, red, red in addition to red, red, blue; and blue, blue, blue.

OK. Now one issue that arises here, as you're probably guessing, especially when I did this kind of transformation where I said, oh, three red vertices. That's just like this picture, except these are now blue. That's not going to work so well from

whatever they're attached to.

Now luckily, we can deal with this. The more direct version is if we're trying to build some Boolean formula like an AND of ORs, the inputs are in the bottom. Because these are the inputs to the AND. The AND wants its input to be red. But the OR is providing an output which is blue. How come we convert a red edge to a blue edge? Luckily, we can do it.

As long as we have an even number of red to blue conversions, we can fill in this gadget in between, and it works. So basically this is in a forced configuration. I think it can't change at all. At least from this position, it can't change.

What we need at this stage is that these guys can point out-- both of them. So you always get a weight boost of one from both of these vertices. And so now if this guy's pointing in, that's a total weight of 2. And so then this guy can point out. Of course, as this guy is pointing in, this guy can point out. And it's symmetric on the two sides.

So maybe you do try to mess with this gadget. Although I don't think you can. What we need is that you are allowed to leave it like this, and you get a bonus point for each of those vertices, and all is well. And we'll see later how to make an even number of conversions.

Now one thing I should maybe mention at this point also-- this is sort of a technical detail-- but if you used to think of this as a single edge, when I reversed it-- let's say it was pointing up before, and I make it point down-- this vertex will see that effect immediately. This vertex doesn't see it yet. So it's kind of an even more delayed reaction. Because we subdivided an edge into two parts, we can flip one side without having flipped the other side.

The way I would view this is in the original constraint logic graph, we used to have an edge pointing up. We change it into an undirected edge. It has no orientation because it doesn't benefit either side. It's pointing away from both ends. And then we choose to direct it the other way.

So this is an alternative view of a move in a constraint graph that used to be you're reversing an edge. Another view is that you can change a directed edge into an undirected edge. And you can change an undirected edge into a directed edge. So these models are almost identical. In fact, you can prove they're exactly the same power. And they're called asynchronous constraint logic.

So if you were worried about that, you don't need to worry. I won't prove here that they're identical power, but it's not hard. Yeah.

**AUDIENCE:** What was the thing where [INAUDIBLE] that you can't flip those colors?

Cause you can flip the colors, right? But you can flip the two red arrows

**AUDIENCE:** In.

**AUDIENCE:** Inwards. And then you can flip the blue to cover the top node.

**AUDIENCE:** Basically flip all of them.

**AUDIENCE:** So the thing--

[INTERPOSING VOICES]

**AUDIENCE:** And the best you can do is--

**PROFESSOR:** You can't put flip these red guys yet. Are you going to flip this blue guy first? But how?

**AUDIENCE:** Well, you can't do anything first.

**PROFESSOR:** Yeah. Nothing can happen first. There is another configuration. But from here, I think it's rigid.

**AUDIENCE:** Oh.

**PROFESSOR:** There's nothing that can change. You can't flip this guy because this guy would be unhappy. Therefore, you can't flip this guy. Therefore, you can't flip these guys.

Therefore, you can't flip this guy. So I think you get a cycle, and nothing can happen first.

But we don't need that here. We will need it later. That's why I wanted to mention.

OK. So we can convert reds to blues. So now we can chain things together and build a CNF formula. As long as we represent-- so here we're going to use a choice gadget to say-- so right now, nothing is activated. Our goal is to activate that edge at the top that's going to represent that Boolean formula, the satisfiability of that Boolean formula.

We use a choice gadget to say, look, either W can activate, or Wbar can activate. We need some notion of negation. And this is how we're going to get our negation, which is normally called dual real logic. You have one channel representing X, one representing Xbar.

We have this extra straggler, which we basically don't care about. Right now it is taking the weight somewhere. But you can push it up. If we plug-in a suitable terminator down there, which we'll need to talk about.

But then, other than red/blue conversions, then we are basically doing an OR. We have to do an OR of three things, and we only have degree 3. So we do an OR of two things. That gives this output. Then we take an OR of the other thing.

So now this output is the OR of three things. We convert it into a red edge. This is a reference to the gadget we just saw.

And then we can take an AND, here we're taking an AND of three things. In general, it would be an AND of end things. We build a binary tree of those ANDs. And so that edge can reverse if and only if that formula is satisfiable. Assuming we know what to do with degree 1 vertices. So this represents CNF.

Let me tell you what to do with degree 1 vertices. Here are three different gadgets, depending on what you want to do. On the one hand, this is an unconstrained blue terminator. All the vertices are happy even locally. And so this edge can go up or

down. I think that's pretty clear.

This guy already has two incoming edges. Everybody has at least one. And the point here is it's three regular. So this is actually a kind of classic way to make a three regular graph out of a degree 1 vertex.

On the other hand, we can do the same thing with red. And actually, the reason I wanted two incoming edges here is when I recolor them red, that still makes this vertex satisfied. So this guy can go up or down. Now you might say why do I care about a blue terminator separate from a red terminator, because I already have red to blue conversion. Because red/blue conversion-- assume that the number of red/blue conversions was even. And this is how I'm going to make it even.

If it's not even, I'll take any of my degree 1 vertices and add an extra red/blue conversion there. Now it will be even, because it was odd before. But I need either a red or blue terminator, whatever I didn't have before. So this is where we'll get that. And it's easy to do.

In the middle we have a different thing. This forces this edge to be pointed down. That's useful in some scenarios for example. So back to this picture. One thing I've done is we have these floating edges that we want to throw away. We can just use a red terminator unconstrained for that. So that guy can do whatever he wants. That still lets you choose W or Wbar to be true.

If I shrink this picture a little bit more and put here, this is I guess a free terminator. I don't care whether this guy's used also. But I would really like this edge to be directed up. This is, of course, locally invalid. You have to put in a satisfying assignment to make this go up.

So this is how we can prove that constraint graph satisfiability, finding an orientation is NP complete if we use that terminator on the top. In the undirected form, this thing will force this edge to be up, which means it will force this thing to be satisfied, or there won't be a valid orientation, which means there's no satisfying assignment to that formula. So that's one way you could use the terminators. We'll use them in

other ways.

**AUDIENCE:** Can we just use a [INAUDIBLE] red terminator instead of the red-- to do a red [INAUDIBLE], instead of the slightly more complicated thing we actually did?

**AUDIENCE:** [INAUDIBLE] red terminator has a vertex with three red edges.

**PROFESSOR:** That's OK. But maybe that's relevant. So this is going to expand to this. And that requires a red/blue conversion. There we go. That's why. So we're actually already using a red/blue conversion of this style in here.

All right. Let's move on to more interesting things. So that was constraint graph satisfactions NP complete. Let's go to the reconfiguration problems being PSPACE complete.

So now I want to do a sequence of moves and flip one edge, or reach a desired configuration. We'll start with just flipping one edge.

We're going to follow the same kind of prototype that we saw last class, with [? Viglietta's ?] Proof, where we're going to reduce from QSAT. So we have this alternating set of quantifiers, variables. And then we have some Boolean formula. We'll assume it's in CNF form, because we just showed how to do CNF formulas. And we're going to represent variables by two rails, x and x bar. And these things are essentially nested inside each other, and there's some magic mechanisms to glue them together.

So in particular, let's see. At the end here, so suppose these variables, the idea is that one of them will be pointing up and the other will be pointing down. That corresponds to x being true. If y bar is pointing up and y is pointing down, then y bar will be true. In other words, y will be false.

So then this thing will do its computation. And then the output wire is this one. So we'll be pointing out of the CNF logic, if and only if that formula without the quantifiers was satisfied.

OK. Now here we have an AND gate. And we have a signal from this gadget saying

basically, all the variables are set. And here it's called try out. We'll see what that means. Basically I want to check whether this thing is true. And if the try out is pointing out, and the satisfy thing is pointing out, then this guy can point to the left. And so that will tell this gadget that yes, it is satisfied.

So when activated, and this thing has become satisfied, then we will report back to that gadget. In general, each of the quantifier gadgets-- this is an existential one, this is the universal one. Don't look at them in too much detail yet.

But there will be one input here, which is a try in. And there'll be an output, which is a try out, to say, when I'm told by the previous gadget to do something, I'll do something. Then I'll tell the next gadget to do something. And if that gadget reports back with a positive answer, in this case, I just return that positive answer back to my caller. So this is the input and the output.

And same thing for universal. It's going to have a try in. It's going to activate a try out. Then if it gets a satisfied end, stuff will happen. And eventually we might output a satisfied out. So these things are supposed to just chain together left and right.

So let's look at each gadget individually, and in particular, you will see this triangle pattern-- the blue, blue, red triangle-- in a few different places. That's a useful construction by itself. Let's think about that first. It's called a latch. It's like a one-bit memory.

Currently the latch is locked. When this edge out is pointing to the left, then nothing over here can change. There are actually two possible states here. But if this is pointing out, this guy can't flip, which means these two can't flip, which means this one can't flip. OK. I guess this guy could reverse. But that's all.

So this is in a state where A can output something, but B cannot. Now I can flip it to the other symmetric state, where B can output something and A cannot. But to do that, I first must unlock the gadget, unlock the latch by flipping this edge.

If I can flip this edge, then everything becomes free. Now I can flip this edge. And

then I can flip B actually now. Both A and B can output, which is kind of a weird state. But in particular, I can flip this edge. And then maybe put A back. That will let me flip this guy.

And now this guy's happy from this edge. And so I can relock the gadget if I want to. If I click hard enough. So now we're back in a lock state. And now B can output. It could choose to not output. But B can output. A cannot.

So when the gadget is in a lock state, we get at most one signal from A or B. And it can't change. When we unlock, then we can do crazy stuff, have both of them output, whatever. But then when we relock, we know that only one of them is on. And that's our one-bit memory. At this point, it's a nondeterministic memory. You can basically set it to a nondeterministic value, and then lock it down.

OK. This is great for existential quantifiers. That's basically all we need for an existential quantifier. When we're told do something, we're going to convert that blue to a red. So right now this vertex is happy, because we haven't told the next gadget to do anything. This edge is pointed to the left.

So this thing can go up or down, whatever it wants. So basically we are free to unlock this and set it to either value. So where x is pointing out, or x bar is pointing out, or neither. But the direction of this edge will sort of keep track of which thing is being set.

Then the next thing can activate only when this is pointing out, which means the latch is locked. So when we activate the next existential or universal quantifier, or the overall formula, we know both that we're told we're activated from the left, which means all the variables to the left have been locked into their state. And we know from this AND gate that this has been locked into its state.

So it has only x or x bar set. And then we can proceed. And for existential quantifier, that's all you need. That there's some value, some setting for x, that satisfies the rest. And so if you get satisfied from the rest of the formula, you know you are satisfied.

And so here we're using the nondeterminism of the player to make the right call whether to set x or x bar. That's the easy case.

The harder case is the universal quantifier. Here we actually use two latches. And there's a bunch of lines. So what's the idea? This is viewed as a split gadget. So if we activate here, we can do that only with this pointing down. OK. Fine. And this points over.

For the computation to proceed, to do a try out, we need that we are active from the try in. But also, this latch must be locked. This is the locking mechanism of that latch.

So when both of these are incoming, then we will call the rest of the construction. Again, this is just setting x or x bar, just like before. But we're making copies of x and x bar.

So this is a split gadget. If we have this set, we can set both this copy of x and this copy of x. If we have this one set, we can set this copy of x bar and this copy of x bar. So those are splits.

OK. So now, what about this latch? So this is the locking mechanism of that latch. And what we're saying is that if x bar is set, and the formula is satisfied, then we can unlock this latch and set it to something.

Now we will want to set it so that this edge is pointing down. Initially this edge will probably be pointing out. It doesn't have to, though. So we don't really care about this edge. It doesn't bias anything, this edge. Because we're going to have a blue thing coming in, so we don't really care which way this edge goes.

We really want to set this edge. So if you're the nondeterministic player, and suddenly this latch is open, you will want to rearrange it so that it points down. Why? Because we want to satisfy this thing. And this is an AND of this edge and that edge.

So what this is saying is we can turn this latch on, meaning pointing down here, only if for the false setting of x, we get a satisfied thing. Cause here again, we're taking a

satisfied thing and splitting it into two parts. And so satisfied in is true. And x bar is true. Then we can activate this latch and set to the new thing.

Now what do you do? Now we're going to lock the latch, and then roll back the entire computation. Undo everything. In this world, everything's reversible. If you can reverse an edge, you can also put it back.

OK. So now unwind everything. Come back to this place and unlock this latch again. And now set x to true instead of false. And then once this is locked in the true state, we can again activate tryout.

And now we're going to come back through here. We don't care about this latch. We're now going to use this vertex. This is the AND of being satisfied and having x being true.

So if we're satisfied with x true, then this edge will be pointing to the left. And if we already remembered the fact that when we set x to false, we also got true, then this AND will be true. And then we can output that. OK?

So this is why you don't want to reduce from QSAT all the time. There are these annoying gadgets you have to build. But now once you have this, we just need to build ANDs and ORs, which is much easier.

**AUDIENCE:** If we have nondeterminism, do we ever have to roll back things? Can't we just set it to the right one?

**PROFESSOR:** No. With the universal quantifier for all--

**AUDIENCE:** Oh, I see.

**PROFESSOR:** You need to check both of them. Yeah.

**AUDIENCE:** Do you need to build crossovers for this gadget to work?

**PROFESSOR:** Yes. We would need crossovers. But we'll get there. So we're not yet doing planar graphs. At this point, I think we have proved by this construction that for nonplanar

graphs, and only AND and OR gates, we have PSPACE completeness of flipping one edge. OK.

Now what if you wanted to go from one configuration to another? Then I could just put a latch here that's unlocked by this thing. The goal usually is, can I flip this edge. Because if I can flip this edge, that means the whole formula is true.

So all the quantifier constructions. If I put a latch here, and I can activate the latch and then change its state, then I could roll back everything else, and I would back to the original state, except the latch is flipped. And that will be possible only if the formula is true.

So that's how you can predict the entire configuration. Nothing will have changed, except this one edge, two edges. Whatever.

So next question is planarity. But for nonplanar graphs, we're good. Here's a crossover. It's a little bit annoying, I mean, to check all the cases. But this is going to be a blue, blue crossover. So basically these three edges will be in the same orientation, roughly. And these three edges will be-- I mean I can't say they're always in the same orientation. Because there's a transition period. So I think they could both point in. That's like an undirected edge.

But if this guy's pointing to the left, then this guy must point to the left. Why? Cause this points to the left, and both of these are pointing to the left. This is pointing either up or down. Let's say it's pointing up. That will actually have to do with the other edges. But if it's pointing up, and this is pointing left-- oh, notice here I have a degree 4. It's another gadget. Please wait.

But this is the usual constraint. There must be a weight of at least two coming in. And so if both of these are out, then both of these must be in. If this is this way, then this must be to the left.

So if this is to the left, then this must be to the left. And by the same argument, this must be to the left. So that's how you transition horizontally.

And it's a similar thing vertically. If this is pointing up, then both of these must be pointing into that vertex. And this guy's either pointing left or right. That depends on the other edge. But let's say it's pointing to the right. So if that's pointing to the right, both of these are pointing to the right, which means both of these are pointing out, which means both of these must be pointing in. And so if this is pointing out, this is pointing up.

I mean, it depends. If this is pointing up, then this is pointing up. Or this is pointing up. One of them, depending on the state of this edge. But then you can use that again to prove this is pointing up. So it could be they're both pointing in here or here. But that's like an undirected edge. And so if you believe in asynchronous constraint logic, this is simulating the crossover.

So except we have degree 4 red, red, red, red vertices. So we need one more gadget, which is this one. This essentially simulates, of course, the edges here are blue. That's why I've drawn off to the side. Now here we have to be a little careful, because we need that the red/blue transitions do not introduce crossings. Because if we did, we'd have to use a crossover gadget. And we'd get in this infinite recursion.

But you can just stick this on here and turn it red, and connect it to this guy, and similarly over here. And you convert them all the red without any crossings.

And I think I won't go through this gadget, but it simulates the constraint that at least two of these edges must be pointing into the gadget. Cool. So that was a blue, blue crossover. If you have a red, blue crossover, or red, red crossover, again convert red to blue, and then back again.

One more version. So we just did planar graphs, or PSPACE complete. If you draw your planar graph in a grid, naturally you would also want the ability to just take an edge and go left or make a turn.

So do we need turn gadgets and straight gadgets? The answer is no. We can use ANDs and ORs. Actually we can just use ANDs to simulate wires and turns. So

suppose this is one of the gadgets that takes inputs from two adjacent sides and outputs on one of the other adjacent sides.

Then it's always going to look like this pattern, all of the instances of the AND gate are rotations of that. I think no reflections.

So first we build these fillers. These fillers, I guess we'll give them some initial state where we get to choose. So maybe I think I want all these edges to be pointing out.

So I'm going to set everything to 0. It's going to be 0 and 0 equals 0. So that's locally consistent within the cycle. And it means that this edge is free to point out if it wants to.

So in particular, I guess I want these four wires to be zeroes. And so then this one could be 01, and still consistent with the ANDs. And so that means when I put that box here, this is one of those filler gadgets, that I have one unit of flow-- because those are red edges-- pointing out.

And so in particular here, if I have a unit here, I get a free unit from the side, and therefore the AND will be the same as the input. So I put two by two blocks here. I also have a similar two by three block, which I put in these chunks.

And then this will just copy whatever value is here, up there. Again, nondeterministically, we can have both of them pointing in, but this one can point out only if this one can point in. This can activate only if this can activate.

That's a straight. Turn is similar. And if you want to do an actual computation, you put whatever gadget is you want to compute, and use these to copy the data. Question.

**AUDIENCE:** I'm sorry. Do you have [INAUDIBLE]? I'm sorry. The square filler into the top right of the straight grid. Yeah.

**AUDIENCE:** There's a terminator at the input--

**PROFESSOR:** Oh. You mean there's an edge here which doesn't have anything on the side. That's

true. Yes. So that's this little thing.

So in this particular world, this won't work for all proofs, but it's sort of an example of what you could do. There needs to be an input here which you ignore. So it's like a degree 1 vertex.

**AUDIENCE:** Right. And then we can [INAUDIBLE] everything's OK.

**PROFESSOR:** I mean, the point is you have to only build one gadget-- or two gadgets, the AND and the OR. But it has to have this feature that when you stick them on the-- you stick an input to this gate into a non-used neighboring side from this gate, then it just works.

So we'll see an example where that happens. Yeah.

**AUDIENCE:** Can you turn right?

**PROFESSOR:** I'm guessing to turn right, we might actually need the reflected form of the AND. But most gadgets, if it works one way and you reflect it, it also works. So yeah. Good. One more.

**AUDIENCE:** Silly question about what nondeterministic means in this case. Since we've been using nondeterministic to mean you can guess correctly. And to mean that the [INAUDIBLE] aren't being reflected. So what exactly does it mean?

**PROFESSOR:** What was the second version? The edges--

**AUDIENCE:** About the edges taking a while to transfer.

**PROFESSOR:** Oh. No sir, that was asynchronous. Asynchronous is that they take awhile. Nondeterministic is that the player gets to choose which edge to flip. They're related. I mean, asynchrony kind of comes out of this form of nondeterminism that we have.

But they don't have to be. The way I originally set the models, when you flip an edge instantaneously. In both the gadgets that we build, like the actual AND and OR

gates we build for real problems, they will actually take awhile to activate. You'll like pull some things in, and stuff happens before you can pull some things out. So we will naturally get asynchrony.

The way is originally set it up, it was just nondeterministic and having instantaneous flips. But they're the same, so you don't have to worry too much about that distinction.

OK. One more gadget before I go to actual reductions. And this is the definition of a protected OR. I call it protector OR. Cause more ORs is better. But protected OR is the proper name.

Protected OR gadget looks like an OR gadget. So we're going to have two inputs, one output. And these are going to be labeled which are which. I want that at most one input is active at any time.

What I mean here is that if I ever have the state where both of these are active, then the entire universe explodes. And I give no guarantees. I mean, not only does this gadget fail, but all gadgets everywhere could self-destruct. Because we're going to build some gadgets where you really only want one of these two things to be pointing in. If they both point in, then the framework that separates gadgets may shatter. And then gadgets are no longer separate. You can't argue about them. And literally everything falls apart.

So we'll see some examples of that, but this sort of foreshadowing. It turns out we can set things up so that you never have both things pointing into an OR. Because we can build a fully fledged OR out of those protected ORs. So this gadget acts as an OR where these two are the input say, and that's the output. But it does so only using protected ORs.

Now it uses a blue to red transition. So we have to again inspect the blue to red transition. This is where I wanted that this is rigid, that you cannot make any moves in here, because that makes me happy about this OR. This is the only OR in this picture. And I might worry, maybe I could flip one of these edges, and then I have

two inputs activated. But because no edges can flip from this state, in fact, this guy will only have one pointing in, so no matter how you label the inputs and outputs, it is a protected OR. OK. So that part is fine.

And now this simulates an OR. Do we want to go through it? Hm?

**AUDIENCE:** Which of the inputs [INAUDIBLE]?

**PROFESSOR:** So this is one input, and this is the other input. Oh. For these ORs?

**AUDIENCE:** Yeah.

**PROFESSOR:** I'm going to guess-- well, let's find out. I don't remember.

So let's say this guy is pointing in. So if this is also pointing down, then this can point up. But if this is pointing down, this must point up and to the left in order to satisfy this node. So if this is coming in, this one must be pointing out. So I'm going to say that's the output. These are the two inputs.

I'm guessing symmetrically, yeah. If this one is pointing into the vertex, then that must mean both of these red guys are pointing into that vertex, which means this guy must be pointing out of that vertex. So if this one is active for this guy, this guy's inactive. And vice versa.

So this is protected from those two. Similarly over here. But if this is active, and this is down, and this is active, and this is pointing up, then in particular, this can point out. And this guy's satisfied. So then this edge can point to the right. And then this one can point up here. And this can point here. So the AND actually outputs a yes. So if the left is activated, then C can activate as well, and symmetrically.

So this simulates an OR, but it's guaranteed that these two guys are happy. Yeah.

**AUDIENCE:** Sorry. I just want to understand the motivation for this, just to make our gadget-building lives easier?

**PROFESSOR:** Yes. This is just to make our gadget-building lives easier. It's so that I can now say

that deciding whether a planar graph can do a sequence of moves and flip an edge is hard for AND and protected OR. So that's a stronger statement. Now I only need to build a protected OR.

In general, we want to reduce the complexity of our gadgets as much as possible for the hard part, which is actually analyzing a real problem. Yeah.

**AUDIENCE:** Wait. So is protected OR easier to build than OR?

**PROFESSOR:** Protected OR is easier to build, because you don't need to worry about the case for both inputs are active.

**AUDIENCE:** OK.

**PROFESSOR:** So you could have any behavior. You might have the correct OR behavior, or you could have any other behavior, including self-destruction. Or universe destruction, I guess. So it could only be easier to build.

**AUDIENCE:** This is for future reduction?

**PROFESSOR:** Yes. In fact, the very next reduction, I think. Or the one after that. Close enough.

All right. Let's do real reduction, shall we? So all of this stuff that you just saw was built for one problem initially, and then happened to be useful for a lot more. And that's sliding block puzzles.

So these are a bunch of examples of sliding block puzzles. You have blocks-- let's say typically rectangles-- and you're an agent from outside. You can pick any block you want and slide it along some non-colliding path. So I can move this over, and then move this over, and then move this guy down.

In this case, the goal is to move this block to here. Because there's a-- well, I think that's just the goal. These puzzles go back to the '20s. Martin Gardner wrote about them. And in his article, called "Sliding-Block Puzzles," he says these puzzles are in want of a general theory. And there is no general theory because they're PSPACE complete. There's no way to easily tell whether a puzzle's going to be solvable or

not, which is annoying for puzzle design, but there you go.

Here's an actual instance of the puzzle I showed you. Dad's Puzzle, where you take this one. You're supposed to move it here. This optimal solution has 83 moves. And in general, number of moves is going to grow exponentially, because the problem is PSPACE complete.

And here's the proof that it's PSPACE complete, in two pictures. We need an AND gate and an OR gate. And the point is, so here the notion of edges being active is maybe reversed from what you think about. This edge is currently inactive. And if I slide this block out by one, that activates that edge, meaning it points up. So right now the edge is pointing down. If I move this block down, then the edge is pointing up. So it may seem backwards. But if you don't think about it too much, it's actually very clear.

What I mean to say is that in order to move this block down by one, I must first move this block left by one, and this block down by one. So for this guy to be sucked in, this guy must be kicked out, and this guy must be kicked out. Because then I can slide this block over one. And then I can slide this block over and down. Then I can slide this down and then this down. With that sequence of moves, I can move this in. But I needed enough room to do it. So that's an AND gadget, an asynchronous world.

And the OR gadget, this guy will be able to move in if and only if at least one of these guys moves out, because if say, this one moves out, this could move down, this guy can move over, and this can move down. And if the other one happened, this guy can move over too. And then this can move down.

At this point we don't need a protected OR. Now you have to think a little bit about how these gadgets are fit together, but I guess that's the next slide.

Before that, here we have 1 by 2 blocks and 1 by 3 blocks. That's not satisfying because it's not tight. One by one blocks are polynomial, so we can't do one by one blocks. But it's turns out just 1 by 2 blocks are enough. And these were found by a

semi-automatic computer search. And I programmed it [INAUDIBLE] to analyze. In particular, these dots mean that there will always be one of the blocks overlapping that position.

So like even if this guy moves out by one, this is still occupied. And so you could use that analysis to help cut down the search and see that actually all of the gray tiles cannot move at all, assuming that you don't have boundary effects. And only the yellow tiles can move. And that kind of highlights where the action is. Again, if this guy slides out, and this guy slides out, then you get this chain reaction. And this can move down. It needs two units of space. And then this guy can move down.

And similarly, if this one moves out, or this one moves out, this guy has some freedom about how he moves. And there's a couple extra units to gain. Then this guy can move out.

Now this is a protected OR, it turns out. It's not going to be obvious from the picture, but if both of these move out, the whole gadget can fall apart. So this is the genesis of protected ORs. But I won't go into the details here.

I guess I don't have a figure about how the gadgets fit together. But essentially these corners interact. Because here we're vertical and here we're horizontal, you get a little nexus where nothing can move. So as long as the overall boundary is fixed-- the big rectangular box that we're fitting in-- you get this chain reaction that because this point is always occupied, even if this slides up or down one, you transfer the rigidity of this edge to the rigidity of this edge. And so you get that all of the gadget boundaries are rigid, and it's just the action in the yellow tiles, because of these dots, basically.

And I think one thing that might go wrong here is if you could move out by two, then who knows what happens, right? Or is this guy could actually suck in an additional spot equivalently.

OK. So sliding blocks. Now I said one by one blocks are easy. But here's a slight variation where it's hard. In a graph, suppose you have tokens. And a move is to

move a token along an edge. But at all times the set of tokens must be an independent set. You can never have two adjacent tokens.

So this is what you might call reconfiguration independence. And it's a reconfiguration problem in the sense that I'm interested in two solutions to the independence set problem of the same size, in fact. And I want to know, is there a sequence of moves that converts one solution to the other solution. In this case, by moving it into one of the vertices and independence set along an edge.

And here you can very easily simulate an AND. This guy can move down only if this moves out and this moves out, because of these constraints. And an OR, this guy can move down only if this guy can move here or here. And to move here, this guy must move out. And to move here, this guy must move out.

So you can see, you get really short proofs of PSPACE completeness. Just need two pictures.

For fun and somewhat relevant in particular to your p-set. Let me mention another reconfiguration problem. I don't have a figure for it, but this is reconfiguration 3SAT. So suppose I give you two solutions to a 3SAT formula-- two satisfying assignments. And my move is, flip a variable from true to false, or vice versa.

I want at all times to be a satisfying assignment. I claim it is PSPACE complete to find a sequence of such moves. This is originally approved by [? Papadimetriou ?] and a bunch of other people-- [? Gopala, ?] [? Colitis, ?] and [? Mineva. ?]

But there's actually a really easy proof once you have nondeterministic constraint logic, which is-- so we're going to simulate nondeterministic constraint logic using this reconfiguration problem. So in order to represent an edge in our constraint graph, we will just make that into a variable. And true for that variable means the edge is pointed one way, and false means the other way. You just decide economical orientation for each edge.

And then if we have an OR vertex-- call the edges x, y and z-- then the constraints

are that x must be in, or y must be in, or z must be in. That's an OR constraint. And "in" here means I write x or x bar, according to whether in is the positive or the negative orientation-- the true or the false orientation, however you decided this. So that's a 3SAT clause.

And an AND vertex, say, x, y, and z, is going to be two constraints, two clauses. If x is out, then y must be in. And if x is out, then z must be in.

OK. So that's two clauses. We AND them together, we can convert a implies into a not x out or y in, as usual. So these are 2SAT clauses. You take the conjunction of all those clauses, you get a 3SAT formula. And that exactly represents that the configuration is valid, that it satisfies the inflow constraint. And so reconfigurations in the constraint logic are identical to reconfigurations in the 3SAT instance. Very nice simple proof found by Sara.

So now you get the sense that constraint logic is really just a very simplified version of reconfiguration 3SAT. We have some very specific constraints on what we need to worry about, some very specific types of vertices we need to implement. And that's it. So of course you could simulate all of reconfiguration 3SAT, but your life is going to be a lot easier by simulating this special case. And you've seen already a few examples of that.

OK. Next problem. Rush hour. This one we actually covered in lecture one, so I just remind you that here the blocks can only move in the direction but they're oriented. And again, you can build an AND or protected OR. In the setup, this result was proofed previously, but here's a constraint logic proof. Again, we do protected OR.

Now this is 1 by 2 blocks and 1 by 3 blocks. That's as far as we could get. This is a picture here we actually see how things come together at the corners. So this is clearly a locked configuration. That's good.

And then Trump and [? Cidibrasi ?] came up with a purely 1 by 2 car version. It's more complicated. And they had trouble getting just a single gadget. So they ended up doing two AND gates using, I guess, rotational symmetry here. Two AND gates

at once.

But of course, this is really a split viewed from below. So if you just throw away that end, and they have a way to terminate things, then that is just an AND gate.

And similarly, they had to glue two OR gates together. But that also can be used to build an OR gate. They built something related to an OR, but in particular, it is a protected OR. And so that's enough. In fact, these actually act as latches, if you wanted to be more efficient. But you don't worry about that, you can use them as protected oars, and then build latches out of that, and build all the things we've done. So that's cool.

An open problem in the world of Rush Hour is what about 1 by 1 cars. Now this is a little weird, because 1 by 1 cars don't have natural orientations. But if each car is marked whether-- I mean, you can see where the headlights are. So you know whether it's going horizontal or vertical only.

It's open whether this problem is PSPACE complete. In the same paper, they do a computer search to find the hardest 5 by 5 puzzle, and the hardest 6 by 6 puzzle. I think with just one blank space in here, the first move is to move this up, this up, this right, this right, this right, this down, this over. The first chain of moves, so to speak. But it's 199 moves to solve this. This puzzle is 732 moves. And it's drawn here.

That's the longest puzzle that's 6 by 6. So it seems to be growing exponentially, maybe. So maybe it's PSPACE complete, but we don't know.

Triangular Rush Hour. You can do that too. Again, there's an AND gate and an OR gate. Here we had to build a straight. But then everything fits together. Question?

**AUDIENCE:** I was just gonna ask what is the goal in those? Like, there wasn't--

**PROFESSOR:** Oh. Right. What is the goal. I mean, if you put the final edge of the reverse up in the corner somewhere, then there'll be one car which can escape through a little slot in the exit. That's the usual Rush Hour setup. If and only if that edge-- Sorry?

**AUDIENCE:** The pictures didn't specify.

**PROFESSOR:** The pictures did not specify. It's true. Oh, and these ones. I think it's like one of these cars is going to go out through the right. But I forget. I think it might be the second row, or the first row, something like that. OK. So, cool.

Here's another problem. Turns out to be related, though it doesn't seem like the same sort of puzzle. This is called a hinged dissection. It's a chain of blocks that folds from one shape-- in this case, equilateral triangle-- to a unit square. Goes back to even before 1900s. But this one's from 1900s. Early 1900s.

And so that's cool. In particular, there's a motion that avoids self-intersection. And we proved a few years ago that for any two polygons of the same area, there is a finite chain of blocks that can fold one into the other without self-intersection. So great. Problem solved.

But that's good for if you get to design the hinge dissection. But what if you have some existing hinge dissection that someone came up with? So this is earlier work. But it's a chain of these right isosceles triangles-- 128 of them that can fold into any letter of the alphabet and any number and a square.

So you can follow the six into a square into an eight into a nine into a zero. That's cool. Open problem. Can you do it without self-intersection? And it would be great if we had an algorithm to tell us. But it's PSPACE complete.

So if I give you a hinge dissection, and I want to know, can I get from this configuration to that one without collisions, you can simulate essentially Rush Hour. This is a gadget-- the hard part here is to make everything connected in one connected hinge dissection. And it can simulate this block sliding to the right or not in this kind of way, avoiding collisions locally. And so now it's just each of these blocks can slide left or right or up or down. And they have some weird shapes, but it ends up working out. So that's kind of a cool result. This is before we proved hinge dissections actually exist. It was discouraging, maybe. But when you're constructing your own hinge dissections, it's a lot easier to avoid collisions.

All right. Pushing blocks. Remember this table from Lecture 4, I think. So we did a lot of the NP hardness proofs. But there are some PSPACE hardness as well. And I'm going to cover these two.

So Sokoban first, and Push 2F, second. So remember, Sokoban is this puzzle where you want to get the blocks into the target locations. And you are walking around. You only have the strength of one. You can only push one block at a time.

So I think this is Level 5 apparently, in the original classic. So this is a little bit awkward in some ways, because in constraint logic, you're supposed to be able to flip any edge. In Sokoban you have an agent who's walking around. So we're going to cut a lot of tunnels in the walls so that the agent can go anywhere.

And then also, these blocks are exactly where they want to be, meaning there is a target location where that block is. There's only one block that's in the wrong spot. And so we're going to use our configuration to configuration thing to say, I want to solve the QSAT formula, which lets me move one block out of the way.

So I can move one block that's in the wrong spot down to a good spot that's in target location. And then unroll everything. And then all the blocks are back where they wanted to be. And the one guy that we needed to move got to where he needed to go. That's Sokoban.

And the rest is mostly this AND and OR gadget. The idea is, because these guys are where they need to be, if I move the D to touch the A, you're permanently screwed. You can never separate them again. And so you'll never be able to fill some spot, because D in particular, can no longer be useful. But every block needs to be used.

And so let's see, in order to move V down, we must move- or, sorry. That's the other way around. In order to move C to the left, we must first move B and E up one, and A and D left one. And you can check that you can actually do that by moving this out and moving this over, and then pushing E up, and then pushing B up. Then C can move in. And it is reversible if you check carefully.

And for an OR, we just separate these a little bit more. So separately each of them could move. And then this guy could move in and you won't be trapped. If you just tried to move this directly, then you'll never be able to get in here. And so these guys are sort of locked in position. And that's bad news. Or they could only get worse. You could move A in here, but then you're really in trouble.

Now once you do that, you might have some parity issues because of the widths of these gaps. But you can build this kind of extra-long tunnel to change parity. We talked about tunnels. You also need turns. And then you can simulate constraint logic.

This is one of the weirder constraint logic proofs. It's not directly mapping because of the agent issue. But it turns out to work.

OK. That was Sokoban. Next one is Push-2F. I will not cover this one in detail because it's quite a complicated proof.

But I will tell you that we can simplify it, now that we know what we know. This is a fairly old proof. 2002. So it predates last lecture. It predates the [? Viglietta ?] paper.

So what we've constructed in this case is a lock gadget. This is just like the final meta-theorem from last class. We have an unlock traversal where you could come in the U and actually come back out the same place. It unlocks the thing, so you could then go from I to O. Or if you go from LI to LO, you're forced to lock the door. Sorry. It was called a door in the last class.

So that actually has a door. Now we needed a door and a crossover. And you can build a crossover like this. I think at this point we have a crossover.

So left of this line would be a PSPACE completeness proof using the [? Viglietta ?] framework. We didn't have it at the time, so we did this part of the proof also to build an AND and an OR gadget. We're essentially doing the 3SAT thing. And we're just checking that all of the things that need to be true are true for AND and OR.

This reminds me of the reduction from three coloring to one of the other pushing

blocks problems. Push-1X, I think. We'll just check that all the things that need to be true are true. So that's the rough sketch.

But there are a lot of details getting it to work. It'd be great to solve Push-1F in the same way, but that remains an open problem. All right. That was pushing blocks.

Next we have rolling blocks. So this is a class of mazes. So this is how you usually are given the input. What this means, the red things here are fixed. And the green thing is actually a 1 by 1 by 2 block sticking out of the board.

And so it was in this position. And you can roll it over along one of the edges, and it ends up occupying this rectangle up here. And you can keep rolling your goals to get to t sticking up. And it's not so easy. Hence the dot, dot, dot.

So that's the traditional rolling block maze. And if you have a lot of rollable blocks, in this case, all the blocks are rollable. So these red things are actually sticking up. And the green ones are obviously lying in the plan. And the 1 by 1 squares mean that it's sticking up.

Then it's PSPACE complete to roll a particular block, because in order to roll this thing down, you have to make space here, which means this has to have rolled up. Which means-- well, so if this rolls out, and this rolls out, it will be like here and here. Then this can fall over, replacing that rectangle. This can fall over, replacing that rectangle. So that's this and this.

Then this guy can roll down, so it comes here. And then this guy can stand up. And this guy can fall over. So you get that picture. And so it's AND game.

And this one is an OR gate. I think I won't go through all the things, but if either this one comes up, or this one falls over, then this one can come down. And you can show in this case that it's essentially impossible, or never useful, to make any of the red towers fall over, which lets you argue about gadgets independently. Cool. That's a pretty recent result.

Oh, in this proof they also-- it took me awhile staring at I guess these gadgets to

think, oh, do these actually fit together? This fits together. But if I rotate this one 180 degrees, the A's will not match up. And so you need a shift gadget. And you also need a sort of straight gadgets for just indicating information without any stuff going on. OK.

Getting near the end. Plank puzzles. These are ThinkFun puzzles, under that name River Crossing. Although they predate that a little bit. So here you have again, an agent moving around. So we're going to have to do some work to get rid of that.

And you can, for example, walk along a plank. Then you can pick that plank up. And you can only carry one plank at a time. And you can place it, as long as this is a length two plank, as long as there's another dot that's exactly length 2 away, you can drop it down. And then he walked here and walked here.

Then that pickup, that unit-length plank, put it down here. And I get to this picture, then I can walk here, walk here, pick up this plank, put it down. But I also could have walked back to pick up this plank, walk over, drop it somewhere else. But I can only carry one at a time.

You want to cross the shore, cross the river. This is PSPACE complete by nondeterministic constraint logic. And there's stuff going on here. I mean, rough idea, imagine you can sort of teleport to some extent. If this guy can move over here, and this guy can move over here, then let me show you a bigger picture.

This is a bunch of gadgets stuck together. But also what we see are the navigation mechanism. There's a length three guy here, and a length three guy here. The length threes are the ones that go around each gadget. This is one gadget. There's all these length three distances. There are no other length three distances.

What that lets you do is grab this, move it, and basically you can walk around the boundary. Then you can also walk along the boundary, grab this other guy, and move it somewhere else. So you can walk along the boundary and carry an additional length three thing with you.

So that turns out to be helpful in that when you're trying to traverse a gadget, you need to be able to enter from this side, and also enter from that side. And you can set up those planks to make that possible. I'll just wave my hand and say that is possible. It's a little hard to see in these diagrams. But you're getting NCL reduction again.

And I have two examples of non-puzzles. Sort of real world problems reduced from nondeterministic constraint logic. One of them is dynamic map labeling. So there's a whole field of map labeling. One of the typical setups is that you have points, features, cities, whatever that need to be labeled.

Labels are usually model by squares. Because if you scale things, if they were rectangles of fixed size, you'd turn them into squares. And typically you want to have that square, one of its corners should be at the dot. So it's easy to read.

But now, suppose things are changing in your map. Maybe you're adding points, or your zooming out, or zooming in. You'd like to continuously change the labels and you don't want any labels to intersect. So you could think of zooming out on Google Maps. I'm not going to talk about the zooming out, although this paper proves that that is PSPACE complete.

Question is, how many labels can I preserve and be able make a continuous change. Simple example is maybe you add one city. In order to be able to add one city and add a new label, probably something has to move out of the way. And then you can get a nondeterministic constraint logic argument.

I love this figure, because it shows all the gadgets and the ways that they fit together. This is a nondeterministic constraint graph. And an AND gate here. The pinks are the inputs.

So if this square can move over here, then this one can move up to here. And if this one can move over to here, this one can move down to here, then these two can move left. Then this one can move left. It overlaps both of these, so both of these have to move out of the way before this one can move, and that one can move. And

so on.

You can see how to build a wire, which is just a chain of these things. There are these special obstacle blocks which prevent you from doing anything else. cool.

And then OR is actually easier. You just chain these two things together. And it doesn't matter which one is the input. This guy can be in one of three locations according to which one it's directed in, which corresponds to the blocks being away, out of the way.

So continuously changing map labelings is going to be hard pretty much however you slice it.

And finally, this problem is even more complicated, but it's a well-studied problem in computational geometry of searchlights. So there are these dots. And imagine you have a laser pointer. And so you can shoot a ray in any direction of light. And if the laser hits the spy, the spy dies.

OK? Your goal is to eliminate all spies. You have this yellow region, which is where all your treasure is. And you want to make sure there are no spies in the treasure region.

So normally this is a finding-the-spy problem. If you ever see they spy, you can capture him. But the spy can move around really fast while you're moving your lasers around. So how do you sweep your environment with lasers? Turns out it's PSPACE complete to make sure this region is empty. And I won't go through the proof. But it's an orchestration of laser continuous reorientations.

And you end up with ANDs and OR gates, and crossovers-- that's this gadget-- and then the nondeterministic constraint logic's happening down here. You end up clearing all of this space pretty easily. And then there's this little extra spot that will only be cleared if you get an appropriate laser from this guy. Then you could sweep up to there and make sure that it's clear without-- the worry is that there's a bad guy over here, and he somehow sneaks back into the other region. That forces some of the lasers to stay where they are, and ends up simulating nondeterministic

constraint logic. Pretty epic proof. But I will leave it at that. And that's all for today.