PROFESSOR: All right. Today, more games. Last time in general, we're going to look at this categorization of different types of games. So far we've been focusing on the one-player category, which normally think of as puzzles or not games at all. But today we're going to branch out in the other directions. And we're going to talk about zero-player games, which arguably are also not games, and two-player games. But to fill out this space, we probably won't get out here yet to EXPTIME and Undecidable games. But we will stick to this chunk.

We're going to start with zero-player games, also simulations-- Game of Life is the classic example here. And these can be very hard, despite being very simple and not really having any choices. So as a funny equality here, the sliding block puzzles and things that we looked at last time where you have agency, you have a choice of which move to make, will remain PSPACE complete even when you don't have that choice.

This category is not so interesting. If you a polynomially bounded zero-player game, that is a computer. And it can just compute something for polynomial time. And recall, this polynomially bounded number of moves. And this is an exponentially bounded number of moves.

Although you can actually go a little bit higher. And we'll talk about that and get even harder problems when you have potentially infinitely many moves. And then we'll go into two-player, in particular, two-player bounded, where we get another PSPACE completeness result.

As you may recall from time ago, all of these versions of games have constraint logic variations. So we've focused on this one. But I'll mention this one, and this one,

and this one today. And they're all complete. So we can use them to reduce things.

But before we get to constraint logic, I'm going to be talking about both more classical ways of proving hardness in these classes, and then also the constraint logic way. And we'll start with the Game of Life. Let me write down a definition of the game in case you haven't seen it, or if you've forgotten.

So this was invented by Conway. '70s. So you have two types of cells-- living cells, which are the black cells; and dead cells, those are the white cells. And you have this iteration-- every cell in parallel updates. In general, this is called a cellular automaton.

And if we have a living cell, it will continue to live if and only if it has two or three living neighbors. This is among the eight neighbors in the 8-way adjacency. And then if we have a dead cell, then it becomes living-- it's born, if you will-- if and only if it has exactly three living neighbors.

And so those are the rules. We don't really need to know the rules except for verifying that very small gadgets work. Most of what I'll talk about are higher level than specifics of these rules.

But some things you should know is sometimes you get a periodic pattern like this pulsar. Sometimes you get a static pattern. These things don't change because for example, on the 2 by 2 square, each of the living cells has exactly three neighbors. And so it lives. And each of the dead cells has not exactly three neighbors. And so I guess one or two. And so they remain dead. And so nothing changes. That's called still life. There's tons of puns in this area.

And so that's well and good. And so one question you might ask is, given a pattern, is it a still life? Well, that's easy to check. Is it periodic? That's harder to check. Does it ever die out and become nothing? That's actually kind of common. And that, we will prove, is very hard to check. But all these questions are related, other than still life, which is easy.

So some other cool things you should know about are gliders. Gliders are a kind of

periodic pattern, but with a diagonal offset. So they go off to infinity in a diagonal direction, unless they hit something. Then something might happen. But in the absence of anything else they'll just keep going on for infinity.

So this is a little different from most of our games. Most of our games we have a finite board. There's some rectangular thing. Sometimes Life is played on a torus. So when you go off the south edge, you end up on the north edge, and so on. In that setting, Life is going to be PSPACE complete.

So in general for zero-player games, if we have a polynomial number of moves, then of course we're going to be in polynomial time. If we have an unbounded number of moves but a polynomial space, basically, then we're going to get PSPACE. OK. No surprise.

So the finite versions of Life will be like that. But if we have infinite space, normally we can't think about infinite space. But with Life it's meaningful to have an infinite board. But initially only a finite rectangle has any stuff in it.

So if you just encode all of the positions of the living cells and say, well everything outside of that is dead, then you can think about infinite space. And here we will get that the problem is undecidable. So there's no finite algorithm to solve it. So that's cool.

And I'm going to tell you a little bit about both of these. Spaceship is like a glider, but it goes in one dimension instead of diagonally. So usually horizontally. There are many spaceships out there. This is one particularly efficient one.

And one of the coolest things is this thing called Gosper glider gun, invented by Bill Gosper, I think while he was at MIT. It's this construction up here. And it creates gliders. And so they just go off to infinity. And you just keep making more, and more, and more of them. So in some infinite sense, this thing is periodic, I guess. But at any finite time you have maybe n of these coming out. And you just keep making more and more. Gliders and glider guns are the key things you should know about for hardness proofs.

First I'm just going to show you a couple of pictures of fairly recent hardness proofs. This is an explicit construction of a particular Turing machine. I think it is a Turing machine that given a number represented in unary doubles that number. So it's a fairly clean machine, which is in here.

And there's two stacks. So you can think of a Turing machine, normally you have a tape. And you have your head at some cell in the tape. You can think of that as a stack going to your left, and a stack going to your right. Each time you can pop off an item from the stack and push it onto the other side. That would be the same as moving to the left or moving to the right.

So this is implementing what's called a pushdown automaton with two stacks, but basically a Turing machine with one tape. And you can do that. And this is done algorithmically. So he also applied it to a universal Turing machine, if you're more a fan of that. That's the Turing machine that takes a Turing machine as input and simulates it. So that's the code in here. It's much bigger.

Now this construction relies on the tape being finite. So it proves PSPACE completeness for the polynomial space version. It will remain in this rectangle, as big as it is. And it will run the machine. If it ran out of tape, then the machine might crash or not do the thing that it was supposed to do if it was a Turing machine.

So that's fine for PSPACE hardness, because simulating a Turing machine is hard. I mentioned that last time. But not good for undecidability.

Same guy a year later came up with a Turing machine that grows itself automatically. So it's a little hard to see, but this is the original thing. And then as you run it longer and longer, this machine moves along this dotted line. So it's moved here. This is like the dirt that it leaves behind. And simultaneously it's been building larger and larger versions of its stacks, so by the time it needs to use it, it will be available. Because space is bounded by time.

So limit on the rate of growth. Very complicated, obviously. But extremely powerful.

This establishes that some version of Life like, will the thing ever stop growing, is undecidable. Cause if the machine halts, you could make this thing just stop.

It will non answer the question of, given a life setup, will everything eventually disappear? At least not easily. So I'm going to talk about that version and a somewhat older proof, partly because I like old proofs. And it's a very epic proof.

So this one, I don't understand the details of. But they are online if you want to study this thing. There is another proof by Berlekamp, Conway, and Guy in their series of books called *Winning Ways*, which is a very cool set of books. Interested in two-player games in particular.

But they have this chapter about life being undecidable. At a high level it's very simple. It's just if you have a glider, that's going to represent a 1 bit. If you don't have a glider in a particular area, that's going to represent a zero bit. And so you can think of streams of bits and doing stuff with them over time.

And so that's cool. Everything's going to be oriented diagonally, of course. Now one useful structure. That's going to be our wire.

We can also build a terminator. They call these eaters. So this is a particular construction where-- and this is frame-by-frame-- if the glider comes in here, it basically gets eaten by the eater, and the eater remains. The notation, by the way, is that the dots are going to be newly born positions in the next step. And the holes are going to die in the next step. So you can see the transition.

Stuff happens. These are just things you need to check. And boom, it ends up just the eater remains. The eater is otherwise still life. So it will just hang out.

So you can send gliders into an eater, and they'll just disappear. One thing to note is that the exact offset of this stream versus the eater is very critical. You do something a little different, eater might explode or do something else weird. So if you're going to build these, be careful. We'll talk a little bit more about parity and things in a moment.

OK. So here's a fun thing you can do with a glider gun up top, and a stream of bits. So if you have a stream of bits coming in here. And you have a glider gun shooting one, one, one, one down, then if you set up your gliders with the exact right offsets-- so I've got to shift that gun to be precise-- and then when gliders hit each other, they annihilate each other. Nothing is left with this particular offset.

So what that means is in some sense you're negating the signal. If a 1 comes in here, it'll hit another 1. And then 0 will come out here.

But in particular, if there's a 0 here, there will be nothing to stop this guy from going through, so you get a 1 out. So this isn't really a negation, because whenever it's horizontal, it's going to be one way. And whenever it's negative, it's going to be inverted.

But this is really a turn. So if I turn this and then turn it back, which I think is the next image, I can now turn a wire. So this is progress, but already a little challenging. So we have this input wire gets turned by the gun. Turn back, turn back, turn back. So sufficiently zoomed out, you can do arbitrary turns. You can use this to delay signals, which is useful, because timing is really sensitive here with the wires and when there's a one bit or a zero bit.

Another thing you can do with this is get a parity offset shift. Because there are actually many different offsets of the gliders that all lead to evaporation and into nothing. So you have to believe this a little bit, but there are enough different offsets here that by doing a bunch of turns like in this picture, with slightly different offsets of each of these guns, you can shift your signal to be anywhere you want on the grid. That's good, because all the gadgets will require very careful alignment.

So, that's a shift. So now here's how we can do AND and OR gates pretty easily. So on the one hand, if we have two streams A, B, and we want to make the end of them. We're going to use a single glider gun to kind of merge them together. So a 1 is going to get through here if and only if there was a 0 here. And in that case, it will prevent any A from getting through.

So if B was false, there's a 0, then there'll be a glider here. And it will annihilate whatever was an A, so the AND will be 0. On the other hand, maybe both of them are 0. Then the glider will just go up here to the eater and disappear. So that's also fine. And if A is zero, then there will be nothing over there, obviously.

And if they're both one, then A will get through, because B will prevent the gun here from getting there. So A and B have to be sort of offset in time, because it takes a little bit longer for the signal to get here, but you get the And.

And similarly for the OR, if you want to send up a gun here, and then it's essentially getting negated on the right. So for there to be an output in the OR, there must be nothing here. And so for a glider to be blocked, either A blocks it or B blocks it. So then the output is the OR of A and B. And because everything here is horizontal, we're not negating anything.

That's actually annoying. It would be better if we had an AND and an OR. Or an AND or an OR, cause then it would be done. So we still need negation. Yeah, question.

AUDIENCE: If you go back into this line, you are going to have this A and B output there as well, to get the terminator. In the right-hand diagram.

PROFESSOR: Right. Right. Sorry. So there should also be an eater here, because this is not fully occupied. Good. But because this is fully occupied, we don't need an eater there. Cause this is fully occupied, we don't need and eater here. You could put them in for good measure. But, cool.

So before I show you the next gadget, I need another fun tool called the kick back. So this is of somewhat weirder offset between two gliders. So one's coming in this way, the other is coming in this way.

And stuff happens. And then this glider goes backwards. Returns along its original spot. So this is really cool. It means if you're sending a glider, if you can kind of head it off at the pass with the perpendicular glider, you can cause that glider to come back the way it was going. Which I think exactly the same offset, even.

So this is useful for a lot of things. In particular, it lets us build a crossover. So the idea with a crossover is that instead of every-- a glider gun produces gliders at a certain rate. Call that rate one.

When it's so dense, if you cross two streams, you're guaranteed collisions and something unintentional will happen. But if I could just thin out the stream so that every 10th position is a glider, and then we only look at the bits, you know, modular 10, then there will be no problem to do crossings. You just have to set up the parity so that the gap happens whenever the other guy happens. And vice versa. So they'll never collide with each other. Even if it's all ones.

So here's how you thin out a stream. We have a single glider here, which is just going to be kicked back and forth, just hanging out. And on the other hand, we have these guns-- G1, G2, G3. So this is going to replace a single gun.

So this is the notation for the kickback minute mechanism. So mostly these guys are just going to shoot off. But they're set up so that if this glider comes here, he gets kicked back. And also we lose one glider here, but they're going to eat it all up anyway, so it doesn't matter.

And same thing over here. So this guy is mostly ones, but every once in awhile this glider will come in and get kickback, and consume one of the gliders down this G2 stream. So there will be a couple holes here. And then if you basically negate that stream, so you send another glider down through here, then only every once in awhile will you have a glider. And by controlling this width, you can control the rate. And I think in this proof, you just need all of them the same rate. Sufficiently large to allow crossovers.

OK. Cool. So that's thinning and crossover.

AUDIENCE: Quick question.

PROFESSOR: Yeah.

AUDIENCE: Are glider gun constructions that we know about, or as least know about it at the time [INAUDIBLE] enough that you can't just set the rate of fire from the gun construction use.

AUDIENCE: There's a gun for every period either greater than or equal to 14.

PROFESSOR: Oh, cool. According to--

AUDIENCE: Wikipedia.

PROFESSOR: Wikipedia. Must be true.

I'll make a weak claim to the extent to which I know glider guns, these are not like easily variable constructions. They're kind of magical, and they just work. There are probably many different glider guns, each with a different rate. But I don't think there's a general purpose glider gun with an arbitrary rate.

Now there might be one by now that has a slow enough rate that we're OK. But I don't know. So this certainly guarantees that yeah, you could do it. It's a little bit messy, but maybe there's a simpler one.

Life is still actively studied so there are many more guns and there were in the early '80s so I think I won't go through this construction in detail, but using orders and some kickbacks and some crazy ideas with guns, you could build a split and a not. Is that OK?

In particular, we're using this idea of rate limiting, so there's a lot of zeroes here. And this is, whatever, 10 spots-- there's only one every 10 spots. And there's two cases, depending on whether A is there or not. And by ORing it together with something, and then having all these chain reactions, we end up with a copy of A here, and a negated copy of A hear with different offsets. Yeah.

AUDIENCE: So for NOT, isn't there the super easy gadget of A equal gun, and then-- sorry-- the gun gets through if A is not there.

PROFESSOR: That's what we had for the turn gadget, but it always turns the signal.

AUDIENCE: And that's bad?

PROFESSOR: That's bad because it means if you think of your gadgets having inputs horizontally, then you need to negate and still be horizontal. So this negates and still is horizontal. In their writeup, they actually call the turn a NOT gadget. But I think that's not NOT. It depends on your perspective. You definitely need a direction-preserving NOT. OK.

So that's cool. And at this point, they say just give this to the engineers and they'll build a computer. Once you have AND or NOT, you can build binary logic, so you can do CIRCUIT-SAT style things. And you could make re-entrant circuits. And so you can compute on stuff and build a Turing machine with a finite size, or any machine of finite size, bounded size. So that's enough to prove PSPACE hardness.

But if you want to prove undecidability, you need some way to grow up to infinity. And their idea is to have a small number of registers. Two are enough. This is called a Minsky machine, is the two-counter version. And Minsky proved that you can simulate a Turing machine with two counters.

What's a counter? It stores a non-negative integer. And you have three operations-- increment, decrement, and check whether it's 0. So it's a little crazy. But that's enough to simulate an entire Turing machine with infinite tape.

So the rough idea is to store a bit onto the tape. You double it and possibly add one. And if you want to remove something from the stack, then you divide by 2.

So with four registers, you can use this one to represent one stack, and this one to represent the other stack. Dot, dot, dot. It's not too hard.

So how are we going to represent an arbitrary non-negative integer? We're going to have a little 2 by 2 box-- this is drawn diagonally-- somewhere along this ray. And then we're going to be able to increment or decrement the box by sending a wave of gliders using these constructions. Sorry, next slide we'll have the constructions for incrementing.

Here actually is a glider that's going to test whether this guy's currently zero. And he happens to be. And so here's how you test. I mean, you just send a glider through the box.

And there may be a cleaner test. This one is a destructive test. It destroys the box. But in particular, it destroys the glider.

So then you can have a gate on this side that says, well, if I didn't get a glider, then I know that I'm actually 0, and I can feed that back to my logic circuit. If I do get a glider, then I know this was empty. So that's pretty easy.

The only catch is then you've destroyed the box. You have to create a new box. And you can do that by sending two gliders with a slightly different offset. This looks like this same as all the others, but in this one you end up with a box at the end. So if you send like another one here and another one timed just right here, you'll recreate the box after you've discovered that it was there. So that's cool.

And then this is a swarm. I think it's actually drawn somewhat accurately to push this block forward by 1. I think it's this stream. So there are two constructions here. One is a wave of gliders that pulls it a block back by 3, another one pushes it forward by 1. Together you could use this to increment or decrement by 1. So I mean, lots of details to check here. This one is a little bit cleaner. We have one glider here, a second glide here. And it ends up moving. It goes in this order and then back. And we end up moving the box up to here, back three spaces.

I assume these are found by trial and error. And then to push a block forward 1, this is pretty crazy. Their second wave of five gliders. Where's the first wave?

AUDIENCE: Turning the block into a honey farm.

PROFESSOR: That's right. Turning the block into a honey farm, which is this particular still life pattern. Anyway. Dot, dot, dot. You check the results. It works. It's actually a lot of fun to implement these in a life simulator and check that it works. But for now I'm showing you the ancient diagrams, hand-drawn diagrams.

OK. So there's like a catch with this. In order to generate these gliders in this very specific pattern, currently the only way we have of making gliders is with the glider gun.

Glider guns are large objects. And you can't put them right next to each other. And even worse, if you have a glider gun, it's spitting out gliders in the middle of it. You can't put a glider gun shifted over a little bit even if you slide it this way or that way.

If you want to produce two streams very close to each other, one of them will go through the other glider gun and destroy the glider gun. So you can't actually build this kind of pattern with glider guns ultimately. But there's this trick using kickbacks to get a glider to go out far away from any guns.

And this is to-- so here's your original gun. And this one has some holes in it. So you can get a glider through here. And then get this glider to bounce back and forth until you have the right timing. Then this glider will escape through this little gap. And then the result is, you send the glider basically offset to the right from this gun.

So using that, you can have a bunch of glider guns sort of around where you need it. Get all the gliders to come up. And now you're very narrow. You could have two gliders right next to each other without any trouble.

So you do a bunch of these constructions at various heights. So this is the picture of two glider guns, one shooting a glider through the other. That's bad. So instead, we'll put these away from it, and send the gliders offset to be like that, using two of these constructions.

OK. So now we can increment and decrement, and test for 0. With this similar trick we can do some fun stuff in sending gliders forward and backward. So this is cool. Remember I said that you could send a glider and cause it to come back using the kickback.

So normally if you want to send the glider and have it come back, you need another glider to be shot this way. So that's OK in some cases. But here's a cool construction where all the gliders are way down here, and still I can make the glider

go up and then come back at some desired time. And it's the same kind of idea. You use this construction to make this glider go back and forth, spend lots of time. And then eventually hit this guy at just the right orientation to send it back. And if you have some more stuff, you could actually get it to go back and forth many times and come over farther to the right.

Why do I care about this? For self-destruction. So the idea is if the Turing machine says yes, if it stops, then you want the entire thing to collapse to nothing.

And so when you get to this magical state, you're going to send out a whole bunch of gliders, and then turn them around using the mechanism I just showed-- the boomerang-- and get them to hit every gadget in the construction at exactly the right offset to make them disappear. This is how you kill an eater from going from slightly wrong orientation. This is how you destroy a gun. It ends up with nothing.

Oh. Sorry. This is a gun. This is a how you destroy a square. We already saw that.

So all of that infrastructure being able to precisely place gliders exactly where you need them, lots of details to check. But you should be able to get them all come out and come back. And you always hit them in the backside of the guns.

I think first you kill all the guns so they stop generating gliders. You let the gliders get eaten by eaters. And then you go out and destroy all the eaters and squares and so on.

So, pretty epic. But in the end we get undecidability of life in an infinite board. So zero-player is more interesting than you might expect. This is certainly one of the more epic proofs in that direction.

Next thing I want to tell you about is zero-player constraint logic. This is called deterministic constraint logic, as opposed to nondeterministic constraint logic, which we talked about last time. So let me define it for you.

So in deterministic constraint logic we have a little bit more state. Before it's just in the state of a machine is just the orientations of all the edges. Now we're also going

to have-- it's a little hard to see on this projector-- but there's some highlighted edges which are active. So I'll describe what they are in a moment. So even if you can't see them, they are there.

Active literally means that you just flipped the edge in the previous step. So deterministic constraint logic, we're going to be flipping multiple edges at once in one round. And in the next round, those edges are all called active. And the rest are inactive. Now that's for edges.

We're going to define a vertex to be active basically if that reversal did something useful. So if its active incoming edges have total weight greater or equal to 2. So if the just flipped edges satisfy that vertex possibly in a new way, then we call the vertex active. It's been activated by what just got flipped.

And then here's what we're going to do in each round. We are going to reverse inactive edges pointing to active vertices. And we're going to reverse active edges pointing to inactive vertices. And then these reversed edges are the new active edges. OK.

So let's look at an example. So here's a little gadget. We're actually going to use this gadget in proving PSPACE completeness of this model. And initially just this edge is active. And if that's all that happens-- now there's some inputs and outputs here-- if that's all that happens, this edge is active. It alone does not satisfy this vertex. So this vertex is inactive. And in the inactive case, if the vertex is inactive, we're supposed to reverse the active edge.

This is basically bouncing back. If you try to satisfy a vertex, you didn't satisfy it, you just undo what you did. You're going to reverse the edge.

So what's going to happen here is this guy will just flip back and forth. First he's gonna go this way. Says, oh, I didn't satisfy this one. So I'll reverse this edge. Now it points into here. Still doesn't satisfy that vertex. It's inactive. So this guy will just flip back and forth forever until here we're drawing the picture where A reverses, so because of some gadget below it.

Suppose A reverses at the same time this one is reversing. So now this edge is active. And this edge is active. Still this guy didn't satisfy what he wants. So he's going to reverse again in the next step.

But this guy did satisfy the vertex because he has weight 2. So now it alone satisfies this vertex. So now both of these are going to reverse. This is the forward case. We have a active edge satisfying-- so now this vertex is active.

We're going to reverse these inactive edges which point to the active vertex. So we end up flipping those. Now they're pointing out of the vertex. This one remains pointing in. It's no longer active, because we didn't just flip it. So we propagate from this being active to these two being active. This is a splitter. We have the signal and we split it into two parts. So it's acting like the splitter that we know and love from last lecture.

OK. Meanwhile, this guy just reversed again. But now these two are simultaneously here. So together, those two active edges satisfy the vertex. So this vertex is active, which will flip that edge up there.

Yes. I should say this vertex is a little bit weird. You could think of there being a red to blue conversion here. This guy only has a desired weight of 1. This is basically a red edge, but it's subdivided to fix the timing. So we'll see how to get rid of that subdivision.

But you can think of this as a red to blue. And there's two blue edges here. But this one edge will actually satisfy this vertex. And so then this guy flips in the next stage. And then it's just going to hang out here, flipping back and forth along this path. And meanwhile, this will go off and do something until it comes back and reverses the edge.

And I should say at this point-- because there are a lot of timing issues here, we're going to set up all the gadgets so that they only get an input edge reversed at times divisible by 4. So this one happened at time 0. And then this one we know will happen at a certain parity relative to 4.

And so we know when this guy has gotten flipped, this one will also have just been flipped. And then stuff happens. And then this edge ends up getting reversed. And then more stuff happens. Then this edge gets reversed again. And then the A gets reversed again.

So this gadget has sort of three cycles. When you reverse A, first B will reverse, then C will reverse, then B will reverse, then A will reverse. It's a little bit overkill. But we're going to use this to check two options. We're just going to check one of them twice because that's what we can build easily.

OK. So now we want to prove PSPACE completeness of this model. The decision question is again, does a particular edge ever get reversed?

And so we're going to reduce in the usual way from quantified CNF-SAT. We have our blocks which represent quantifiers. They're going to produce variable settings. We're going to have some CNF logic. And then in the end, this is actually not quite accurate. We're going to get some satisfied signal which is piped into these guys.

So the first thing that has to change are the quantifier gadgets. In particular, the existential quantifier needs to change because before we let the player decide whether to choose x or x bar. So in this case, we're going to use the gadget you just saw, the little wheel, which will try x being false. Then it will try x being true, then it will try x being false. And all we need is that one of them is true.

So if at any point this formula is satisfied, we'll just output that formula satisfied. We have four edges here because everything needs to work modular 4 in timing. So there's a little bit more going on here, which I will get into in a moment.

On the other hand, universal quantifier is similar. I think I will leave it at that. There's this generator to try all the options. This is going to act in some sense like the one bit of memory, the latch, that we had before. So it's similar. But I think the details are not worth getting into. If you look at the appendix of *Games, Puzzles, Computation*, you'll see an explicit execution of this thing. I mean, it's like life. You just run it through and see if it works. And it does.

So what's going on up here is that when we set x bar to be false, we're going to flip this edge. And we're basically telling the circuit, hey, x is false. And then at some point it's going to acknowledge and say, OK. I understand that x is false.

But that's going to be on a different channel. So let me illustrate the issue. So this is how we used to do CNF logic. We did ANDs of ORs using OR vertices and AND vertices. And that was cool if we had a signal for what was true and what was false, we could throw away signals we didn't need.

Then this would propagate and produce a true answer. Nondeterministically it would. But deterministically there's a lot of timing issues here. Everything has to arrive at the inputs at exactly the same time. And AND will only work if this edge and this edge simultaneously reverse.

So OK. Maybe you could subdivide enough edges to fix the timing issue. But it's worse than that. If you're doing an AND of two things, and you discover that the answer is false, then all bets are off. So normally that would be OK, because we can always undo what we did. But we need to guarantee that the deterministic constraint machine will exactly undo what it was supposed to do. So it's an issue that if one of these comes in true and the other's false, this one will bounce back and then chaos will happen. We'll end up getting edges just randomly reversing it at annoying times.

So that seems messy. And so instead we build this much safer version called AND prime, OR prime, and split prime, I guess. There should be a prime there where each input is represented by two things. First the signal, and then the acknowledgement.

And so there's some details to check here. The OR is particularly messy. But these are just deterministic constraint logic machines. You see this gadget yet again to try various things.

But in this case, if input 1 comes at sometime before input 2, then it will essentially-- so this comes up. The signal gets split out here. And so we send off the

acknowledgement. And then this edge will, I think, just be flipping back and forth. So it's basically holding the input for awhile. And later, the second input could come in. And it will trigger this to happen, and also trigger the output, and also trigger the acknowledgement, cause it does all the things.

So that is roughly how that works. So I think the details are not worth spending more time on. This is the final thing after you have all the CNF logic come out, then this is how you end up sending it to satisfy out to the quantifiers, which are here.

So it's a bit messy. But the main point is that this works. And so hopefully we can use deterministic constraint logic to prove lots of interesting zero-player games are hard. We don't know very many interesting zero-player games. So please help in finding them.

There's one detail I'd like to get rid of though. These degree 2 vertices, it would be nicer to avoid those. So this is a bunch of reductions to get rid of degree 2 vertices and keep things mod 4. So we're going to take every edge and subdivide it into a path of length 4, because then if we have two red edges in the original graph together, instead of replacing it by two red paths of length 4, we're going to replace it by two paths here, with the first and last edge being red, but everything else being blue. The motivation for that is now we just have blue, blue edges and red, blue edges. No red, reds.

To get a blue, blue, we just add on a thing like this that's forced to be out. And this will be satisfied if and only if one of these is in. So we've done that kind of trick before. And red, blue edges, we've explicitly done before. So we know how to deal with those. And so we do all the subdivision in order to get rid of the red, reds. Also where these guys have a smaller weight constraint, just one of them has to be in.

So now we have no degree 2 vertices. The other thing is, we have a non-planar graph. It would be nice to get rid of the crossings.

AUDIENCE: Don't you still have a timing issue?

PROFESSOR: This should preserve the fact that everything happens at time 0 mod 4. So we'll slow

everything down by a factor of 4. But it shouldn't be a timing issue. And because these things are rigid, they don't have any timing.

OK. So next we get rid of crossings. This actually is a little easier for once. This was how we implemented a red, red, red, red vertex in the crossover gadget for NCL. But here it's actually enough as a crossover, because we have the luxury of timing in some sense. There are a lot of things that could've happened undeterministically that just can't happen deterministically.

So does it work? Here I pasted in the figure. You run it through and you check that it works in both cases. If they don't activate at the same time, then it works. Activated at the same time, I don't know what would happen. But we can offset all the timing so there are no collisions. And boom, we get crossovers.

So the conclusion is deterministic constraint logic is PSPACE complete for a planar and OR graphs. I think I actually want to also say split. So normally we think of AND and OR as the same thing. But it is helpful to distinguish them based on their initial orientations. An AND would probably be this way. Because you want to activate this, and you can only do that if both of these have been activated.

And so this is the output. For a split, this is the input. So it's initially not active. And then it wold look something like that.

So if you want to distinguish between these two vertex-- sorry. This should be the other way. If you want to distinguish what the initial orientations are, then you also need to list split here. So up to you whether you feel like doing that.

OK. That's all I want to say about zero-player games. Any questions?

AUDIENCE: What should happen in the end? Everything is--

PROFESSOR: In the end, if you satisfy the whole formula then you will get here, this edge will reverse. And so the decision question is, does this edge reverse? Either it will or it won't. In both cases, I think the behavior's periodic. Because it's a funny thing. Yeah.

AUDIENCE: So there were two decision problems for nondeterministic constraint logic. One of them was can you reach this configuration. Another one was can you fit this edge? Do you know anything about--

PROFESSOR: Yes. OK. If you want to solve configuration to configuration for DCL, I think you can basically add some red edges here. So if this thing activates-- well, you need to do a little bit more of a construction here. But I want to basically get an edge to start flipping over here. Whereas before it was not flipping. I think you can get an edge flipping, and then everything inverts. And then you will be in the original configuration, except this guy will be flipping. If you set your parity right, then he'll be flipped in a state where everything else is in its original state. So, pretty sure. I forget whether that's in the book. But I think configuration to configuration, also PSPACE complete for DCL. Good.

So let's move on to multiplayer games. First I'd like to reduce to the two-player case. So there's a big difference between one and two players. But there's not so big a difference between two and three and four and five and six players.

At least the usual decision problem is, does the first player have a winning strategy. Can the first player win, basically. Sometimes this is called a forced win. If a player forces a win.

If you think of it this way, and all the other players are free agents, then the worst case is when they all collude to try to make you not win. So that would be the opposite of a forced win. If it's not possible for you to force a win, that means the players can somehow work together in order to prevent you from winning. So in that sense, they are all essentially one player. You can thinking of them as one hive mind.

Now in the mechanics of the game, of course there are differences between two players and three players, because they have more power in some sense. They can do three things for every time you do one thing. But at some level that is just a two-player game again, where you are relatively impoverished compared to your

opponent.

So the asymmetric, but still two-player game from a complexity standpoint. So we just need to think about two-player games and their complexity. Later we'll add some variations that make this not true. But in a perfect information deterministic game, this is true, this reduction is true.

So for a two players, we're going to have-- at least for the duration of this lecture-- the two players will be called white and black. In Winning Ways for example, they're called red and blue. But this is problematic because we have red and blue edges already. And I want red, white edges and blue, black edges and so on.

So the players will be white and black as in chess, checkers-- not checkers. Go, whatever. But color renaming.

OK. And I'm going to focus today on games that have a polynomial number of moves in any execution, which places the problem into PSPACE. So again, we're going to aim for PSPACE completeness.

It's PSPACE because you can think of this question as being equivalent to, do I have a move such that no matter how the opponent responds to that move-- so every move for the opponent, which I will call response-- I have another move such that no matter what the opponent does dot, dot, dot through the polynomial number of moves. If there's only a polynomial in many moves, then there's only a polynomial number of quantifiers here.

And then in the formula I'm going to write whatever the rules of the game are, and then AND it together with I win. I think you believe almost all games you can write as some Boolean formula to do that. Some polynomial-sized Boolean formula saying at each step you satisfy what you need.

And so this is a QSAT problem. And therefore, any two-player game where you can write moves and responses succinctly and have a polynomial number of moves is in PSPACE, because you can reduce it to QSAT. OK.

So that's why PSPACE is the class we care about for polynomially bounded games. So now we want to prove hardness. And first I want to give you some hard problems in this setting starting with SAT-style games.

This is from another paper by Shaefer, same guy who did Shaefer Dichotomy Theorem. He also did some nice stuff on games. So first game is just your favorite flavor of QSAT. You can think of it as a game. Because QSAT is exactly of this form, you can think of the game as-- in your first move you're going to choose this variable. In your second move, your opponent is going to choose this variable. In the third move you choose this variable. And so on.

Because play alternates, you are following exactly an alternating quantifier kind of thing. And the goal of player one is to satisfy this formula. The goal player two is to have it not satisfied in the end. And so player one will win if and only if this formula is true.

So in the game, you think of it as there are a sequence of variables-- x1, x2, x3. Play alternates between assigning xi and xi plus 1. Player one's always assigning the odd variables. Player two is always assigning the even variables. And then if the formula is satisfied, then player one wins.

OK. So this is what you might call the game version of a satisfiable formula is that player one wins if and only if the formula is satisfied in the end.

But there's some other goal conditions you might consider that Schaefer defines. One is called Seek, which is the first player to satisfy the formula wins. And so in general, if you're choosing some variables, you have a set of variables, some of them have been assigned 0 or 1. Some of them haven't been assigned yet. So when I say satisfy here, I mean that unassigned things are 0. So if you think of everything as initially 0, whoever satisfies the formula first wins.

Then the opposite version is Void. So the first person to satisfy the formula loses. So the formula starts false. And the first person to set it to true loses the game. Other person wins. OK. So those are some rule variations of the goal state. Now let

me tell you about the moves.

So other than QSAT, we're going to have two types of moves. One is impartial moves. Impartial means that both players have the same moves.

So an impartial move is going to be whoever's playing sets an unassigned variable. So on all of these games, you can only assign a variable once. If you can assign it multiple times, then you're no longer polynomially bounded, and you get x time completeness.

But for the polynomially bounded games, we're going to define each variable can only be assigned once. And in a partial game, any player can assign any variable that's not yet assigned. OK. So whereas in QSAT, you had a fixed ordering. First x1, then x2, then x3.

An impartial game version of SAT, players take turns setting variables, any one they want. It's not yet assigned. And in the end, if the formula is satisfied, player one wins. That would be impartial game. But I could also do impartial seek, impartial void.

And then by the way, I made up these names. So they don't match what Shaefer calls them. But these are more modern. These terms are in *Winning Ways*, for example. Seek and Avoid he used. But the other ones I made up.

OK. Partisan version is that there are white variables and there are black variables. And the white player can only set white variables, black player can only set black variables. There's 50% of each. And so they balance if you play all the way to the end, like in the game version, then it's OK to have the white player play any unsatisfied white variable and set it to true or false.

I think that defines all the games that I want. Now let me tell you which combinations are PSPACE complete. A bunch of them. So impartial. In addition to this characterization, there's also all the different versions of SAT that we know.

So we have impartial game positive 11-SAT. So this is a version of CNF. Instead of

3-SAT, I have 11-SAT. There are no negative literals. And it's impartial. So anyone could set any variable. And it's the game version, so player one wants to set it true. Player two wants the formula to be false. That's PSPACE complete.

Also, if we do 11-DNF-SAT, so a DNF formula with positive. This is also as hard. I mean, that essentially corresponds to flipping all the quantifiers. So that doesn't make a big difference.

So this is cool because positive SAT is usually pretty easy with all existential quantifiers. With alternating quantifiers, not so easy. Yeah.

AUDIENCE: Is impartial game positive 10-SAT [INAUDIBLE]?

PROFESSOR: We don't know the exact cutoff. There is a paper, I think, proving six is enough. But the positive disappears. So I think that's an open problem, how low you could go.

It's hard to do the usual kind of reductions from many occurrences to fewer, because games are very sensitive about who's playing win, timing, and so on.

So next one is partisan game CNF-SAT. Here we don't have positive. And we don't have the bound 11. But we changed impartial to partisan. That still is hard.

And next we have a whole bunch of games that work both for impartial and partisan moves. OK. So we have avoid positive 2-DNF-SAT. We have seek positive 3-DNF-SAT. We have avoid positive CNF-SAT. And we have seek positive CNF-SAT.

So the main point is, most combinations are hard for some version of SAT. And these are the tightest ones that Shaefer could show. Doesn't necessarily mean that they're tight. And there's no dichotomy theorem here, so we don't know exactly which problems are hard. But this is a ton of games to start from. And in particular, we're going to use at least one of them.

OK. Let's maybe go over here. OK. A couple more problems which are in Shaefer paper. One's called Kayles, one is called Geography.

Kayles is essentially the two-player version of independent set. So there are a

couple of versions. There is node Kayles and bipartite node Kayles. These are the traditional names. You could also think of this is impartial Kayles and this as partisan Kayles.

So here the moves are all the same. The moves are here. Pick any vertex, add to an independent set. That's a move. So players take turns adding. At all times, you must add a vertex that keeps it independent set. So you're effectively removing all of its neighbors and itself. Last player to move wins. You can't move, you lose.

Bipartite node Kayles, you have a bipartite graph. And one side you call white, one side you call black. A white player can only choose from the white side. A black player can only choose from the black side. So these are both PSPACE complete as well, via reductions from these problems. But I don't have those reductions with me.

OK. So that's independent set. Geography is essentially a two-player version of Longest Path. It's inspired by this real-life game where one person names a city or country or pick your favorite class of objects. And then the other player must name a word whose first letter equals the last letter of the previous word. So you're forming chains of words. And you can think of that as just having a graph where vertices are the words, the edges are valid moves between words. Do they have some common property.

So in general, you're given a directed graph and you're given a start node. And you have a token at that start node. A move in the game is to follow an edge, to move the token along an edge.

But then there's some non-repetition condition. So we have node geography where you're not allowed to repeat any notes. That would be the usual longest path. And again, the last player to move wins. If you can't make a move, you lose the game, just like the real game.

And on the other hand, we have edge geography, which is more like an [INAUDIBLE] path. The longest edge destroying path. So edge geography you're not allowed to repeat edges. And for each one, you could think of a directed graph,

you have to follow directed edges or an undirected graph.

And here undirected, no geography is NP.

Directed, no geography if PSPACE complete. And for edge geography, surprisingly, both are PSPACE complete. So this is a useful problem. I've seen a ton of PSPACE completeness proofs based on geography. It's kind of conceptually clean. And I will show you one now.

Everyone play Othello, Reversi. In general, you have these pieces which are black on one side, white on the other. The black side is the one that you play if you're the black player. But then you can, for example, if you place a new black stone here, given this configuration, because there's a black string of whites and then a black here, you end up flipping that one over. Because this is black and then a bunch of whites, or black and then a space, nothing else happens.

In this picture though, if you place this black thing, this is black, a bunch of whites, then black, black, bunch of whites, black, black, bunch of whites, black. So all of those flip over. And your goal is to maximize your color. You want more more stones of your color than the opponent's. OK. Cool game.

Bounded number of moves, polynomial number of moves, because every time you add a new stone, you've occupied one of the squares on the board. And you'll never get that back. So n by n board would be exactly n squared minus 4 moves, because four is the original number. So this is PSPACE complete, given a particular configuration to decide whether black has a winning strategy, say, is PSPACE complete.

And most of the construction is in here. This box is actually this picture. And that's going to simulate geography.

But let me tell you about this other part. So right now white has a huge amount of area. This is actually really big. So I guess this is very thin, so there's not much action here.

Here you've got a huge amount of white area. So white will win unless black can take this part. And what happens is there's a gap here. And if black can play any of these spots, now black won't be able to play here if there's some corresponding black thing over here. So that's what these long horizontal channels will be. So if black can play over here, because there's a black stone and long white channels here, then this will flip to black.

And then in this picture is this. So if one of these stones up here is now black, that means you can play at position alpha. And if I play at position alpha, black plays there. Then I get black all the way here.

White can't really respond much to that. And then in the next black move, black can go here and flip that one. And then black can go here in the corner. Corners are very important in Othello. So once you take this corner and flip all these things to be black, then I can play here, here, here, here, here, here, here and take all of these.

And because this is the edge, white won't be able to take anything back. So that's the threat. Each of these lines is going to be a threat that if any of these become black over here, white has to turn them back to white. Otherwise, in the very next move, black will win the game. OK.

A lot of two-player game proofs have this constant notion of threat. One player's basically just slogging through trying to put out fires. And the other player is constantly causing fires in order to force the other player to do what they want. So it almost reduces to a one-player game. But there's still some choices involved.

Well, these are not terribly exciting. But let me show you the typical setup. This is just a turn gadget, or it could be a degree 2 vertex.

I should mention we're reducing from directed no geography, where this is directed no geography. Also max degree 3. So that means you might have a degree 2 vertex, or you might have a degree 3 vertex. There are two types of degree 3 directed vertices merging and branching.

So we need gadgets for each of these. Also, bipartite. Bipartite is useful because in geography you're always moving along an edge, which means you'll always be jumping from the left side of the bipartition to the right side. So you can color the edges black and white. And black player will only play in the black side, and white player will only play in the white side. So this is good. You can predict who's going to be playing in each gadget.

So in a degree 2 gadget, it works something like this. If white has played here, that will be because of the other gadget. This is the trigger. Then now this is black, a bunch of whites, and now black can play here, reversing all of these. This is a threat line. And there's now a black square, which means white must immediately play here in order to reverse all of those stones back to white.

But now this is a white stone, which lets this thing trigger. So now black can play at the end of this thing, et cetera. So this just propagates a signal, does a turn. It's a degree 2 vertex. Whatever. All those things.

And this is slightly different version for a different orientation, because these things have to go to the left. This looks a little different. OK.

More interesting are the degree 3 vertices. This is the type where I have two incoming edges and one outgoing. Key thing here is we need to check that we do not visit this vertex more than once. We came in here and then came in again. We want something bad to happen.

Now what should happen bad depends on whether it was the white player moving or the black player moving. if it was the white player moving and they double visit, it means white should lose. If it was the black player moving, and they double visit, it means black should lose.

Because of the bipartiteness, we know which gadget to use. They're almost the same gadget. Just these two dots have been added.

So something like, if this activates, and it's symmetric, but let's say this one activates by white playing here, then black can play here, flip all those, which has this threat

line, which means white must play here. Flip them all back to white. But now this B1 position is white, which enables black to play here, flip all those to black. This is the threat line, therefore white must play here. Can't play there. And turn all these back to white. Now this is white. And now this activates down there. So that's just following the vertex. No big deal.

But if you came along a second time and activated this as white-- now this is already white. So when you play black, you get a black here. And white can't do anything. There's no white stone up there. Yeah.

Well it's entirely black. If there was a white on either side, white could flip it and get them back. But you can't. And so you actually have two threat lines activated. And then black wins. So this is the version where black wins.

If instead, we have a white thing here, everything else is the same. So this was white, this was white. Black now plays here. Now white can go here and completely make this thing white, and then black doesn't have any moves anymore. And that's the way that white's going to win is to prevent black from playing anymore. Then the area that was white is white, so we're done. There are lots of details here to check, but that's how it works.

There's one more gadget basically that says this is for one incoming edge and two outgoing edges. So here you want the players to have a choice, either I go this way, or I go this way.

And if it's a white vertex, you want white to make that choice. If it's a black vertex, you want black to make that choice. But black is still kind of doing all the action. So it can be done. When this activates, then black plays here, flipping all these things. Then white plays here, immediately flipping it back.

That's the trouble, is always black is in control. Now black plays here, flipping that one guy. And white has two choices. It can either play here and flip just this one, or play here and flip those guys. And either put a white here or here, and that will end up activating for black either this path, or this path. So white made the choice. And

in this version, black makes the choice. That's a little easier. Yeah.

AUDIENCE: Are these gadgets constructable?

PROFESSOR: In an 8 by 8?

AUDIENCE: No. Like from a starting configuration. Can you get to a configuration of these gadgets?

PROFESSOR: Oh, I see. I think so. I didn't read that as [INAUDIBLE], but it wouldn't surprise me if it's in the paper. It's traditional in constructing these generalized games to actually show that you can reach this position from the initial state. Although the problem makes sense even if you can't.

It's more interesting to say, well we played like crazy. And then we ended up with this thing. Can you finish it off for me? That's basically the problem we'd like to solve.

Can white win? It's like the commentator problem. People are playing. And now I want to know who's going to win at this point. But I don't know for sure for this reduction. Usually they can.

OK. So that's a sketch of Othello PSPACE hardness reduction from geography. So the last thing I want to talk about, back to this picture, is the two-player setup for constraint logic. So polynomially bounded, two-player game should be PSPACE complete.

First I'm going to talk briefly about what does bounded mean in general for constraint logic. So in particular, for one player which we know, the bounded NCL means each edge can flip only once. That's our definition. And that game is clearly in NP, cause one you flip all the edges, you're done.

And in the same way, we prove that just finding and satisfying orientation of a constraint graph is NP complete, we can show just I changed the very top gadget, that bounded NCL, you just want to flip this one edge is NP complete. Because you basically have to make a choice for each of these vertices, which way to set them.

And then you can propagate up. We don't have to flip any edges for this very simple proof. So NP completeness for a bounded NCL.

We do for this proof need a choice gadget. So all three red edges. It doesn't work to just blow it up like we could before in the bounded case. OK. And there's a crossover gadget in this setting as well. It's actually pretty simple because a bounded you can't revisit.

OK. So now bounded two-player constraint logic is called 2CL. This is PSPACE complete by pretty much the same proof structure. So what's the game? I have white edges and black edges. Each edge is white or black, and it's also red or blue. Exclusive ORs.

So they're drawn as-- these are all white edges. This is an example of a black edge. So the fill is white or black, the outline is red or blue, as before. White players can only flip white edges, black players can only flip black edges. And otherwise it's the same constraints. I mean, the inflow constraint doesn't care whether you're black or white. It's politically correct vertices, I guess.

So we just need one new type of vertex. Everything else is going to be done by the white player. There's one new type or vertex, which is an incoming black edge and an incoming white edge. Call this a variable. The idea is that wherever gets there first, white or black can flip it, preventing the other player from flipping their edge.

So that's what we're going to use for variable settings. And I'm reducing from-- remember the terminology here-- impartial game positive CNF-SAT. The very first one here. Impartial game positive CNF-SAT.

So player one's goal-- white is the player one. Player one's goal is to satisfy the formula and flip their edge. And it's the impartial version, meaning anyone can set any variable. So if black plays this edge, that's going to correspond to setting x2 false. If white plays the edge, it's going to correspond to x being true.

Because this is a positive formula, every time white choose a variable, they're

gonna want to set it to true. And every time black sets a variable, they're going to set it to false, because they want to prevent satisfaction of the formula.

So that's why it's OK to just represent it this way. And then the rest just propagates. So if there are n variables, there's going to be n over 2 rounds where all the variables get set. And then black can't do anything else. Just sitting there. This one is not going to be flippable because of this gadget.

So white's just going to fill in the formula. And if it happens to be come out true, and white can flip their edge, then they win the game. And that will correspond to the formula being true. And otherwise, not. Yeah.

AUDIENCE: So is pass in that you can do? Or they're just like--

PROFESSOR: In this game you can pass. I think you could probably avoid that by just adding a bunch of floating black edges that you can flip, or just flip repeatedly until white either wins, or white doesn't win. The decision question is does white win?

Now we end up with a tie in the case white doesn't win. You can also change that by having a long path of blacks. And black is going to sit there flipping, trying to get to flipping their edge. And the length of the wire is exactly how long it takes to fill in all these things. Yeah.

AUDIENCE: The reason why they have that gadget there at the end is you need to set exactly a certain number of them true? Before black can, and so--

PROFESSOR: Oh. I see. Right. Yeah. That's a little more subtle.

OK. Zero minutes remaining. The same crossover works. And you can also build a protected OR. Protected OR is where you never have both of these coming in.

If you allow me the notion of a free edge, which is just floating there and can be reversed whenever you want, this is a white edge, then this will act as a OR, because this choice gadget can only go one way or the other. It's only one of the inputs can actually flip one of these edges, and then the OR will just take it. So we

can build predicted ORs.

And then I have a bunch of PSPACE hardness proofs based on bounded constraint logic, but I suggest we wait til Thursday to see this. Because they're kind of fun. And I'd rather spend a little time going through the proof details.

So you'll have to wait in suspense before we get to Amazons, which is a very fun-- you should play Amazons meanwhile, if you get a chance. It's very fun if you have a chessboard around.

And Konane, which is an ancient, at least 300-year-old game from Hawaii, and Cross Purposes, which is a modern game. These are all three bounded two- player games. And they're all PSPACE complete by a reduction from bounded two-player constraint logic. Cool.