PROFESSOR: All right. Welcome back to 6890. Today we start a series of lectures on satisfiability. We saw a little bit about satisfiability in the first lecture. Now we're going to do it right, and do it more intensely, and see a lot of examples. Today we'll just see a few examples of using SAT to prevent NP-completeness or NP-hardness. But over the next lecture or two we will see many more examples.

And SAT is really the most common problem that everyone uses to prove NP-hardness. So this is the moment you've all been waiting for, I suppose. This is one of the motivations for organizing this whole class, so we can tell you all about SAT, it's many variations, so you get to learn them all, and then see lots of different ways you can SAT to represent your problems. So without further ado, let's get started.

Today I'm going to give you a whole bunch of problems and definitions, and then we will do reductions afterwards. So there's going to be a lot of problems here. You probably will have trouble remembering them all, there are so many. But in particular, the notes serve as a useful reference, like, here's all the versions you might care about-- here's which ones are hard, which ones are easy. So with that in mind, get ready for the ride.

So the original SAT problem is you're given a Boolean formula, which is like over the operators AND, OR, and NOT. And in case you look at the literature, usually AND is the wedge, OR is the v, and NOT is the neg? I don't know the name of that symbol. And it's over n variables x1, x2, xn. So usually, you're not given true or false so that you can construct them by saying, x1 and not x1. That's never true. So that's false, for example. And then you could negate that and get true. And the question is, can you set the variables to make the formula true?

So that's the general problem. This is the very first problem proved to be NP-complete by Cook and then by Levin. So it's usually called the Cook-Levin theorem, the satisfiability is NP-complete. And I was looking at the paper the other day. It doesn't actually mention the notion of NP in that paper. I think that came later. But the notion is there.

So I'll give you another version of the same problem, essentially, called circuit SAT. This is a useful perspective which we will see probably not till next class. But another way to think about it. Also makes it the NP-completeness a little bit more intuitive.

OK. This is kind of the algebraic way of thinking about things. You have variables and you write operators, and you use parentheses and that sort of thing. But if you're more graphically inclined, you could imagine you have your excise as wires, they're connected to gates-- this is an and gate, this is an or gate, this is a negation. So you can imagine something like this.

So you can, generally, copy your data and do various things. And then this is the output. And so this is what you might call the formula. It is the AND of the negation of the OR of x3 and the AND of x1 and x2. And then, pop off, it's also ANDed with this thing again.

So the one advantage of circuit SAT is you can reuse complex computations, just by copying the signal. Normally, in a formula, you'd have to copy and paste that chunk of the formula. So you could imagine this lets you write formulas slightly more efficiently. I think it doesn't. If you reuse things, you can always write them as a variable over here. Question.

AUDIENCE: What about feedback?

PROFESSOR: Feedback is forbidden. So this should be a directed acyclic graph. Yeah. Good question. So it's an acyclic circuit. That would be another problem, which is harder than an NP. So circuit SAT is NP-complete when it's acyclic and you can convert between one and the other. And it's a little bit more intuitive that you can write arbitrary computations as a circuit. And then this is a question of whether you can-- the existence of some setting of the AIs is the same thing as saying, is there some set of guesses that will lead me to a yes answer?

So that is intuitively like NP and that's roughly how you prove these problems are NP-complete. You write the computation-- the checking computation-- to see whether your certificate is a valid certificate as a circuit or as a formula. And then the existential quantifiers on the xi's let you do all the guessing to see whether they're-- and this is biased, right? It's trying to find a true. And that's exactly what NP does. It's trying to find a true path that ends up answering yes. So yes and true are symmetric here.

So that's the extent of the complexity theory we'll do today. I'm going to give you more versions of SAT. Next one is CNF SAT. CNF stands for conjunctive normal form. I hope you all know prepositional logic. Because we're going to be doing a bunch today.

So ANDs are also called conjunctions, ORs are called disjunctions. This is old terminology. So conjunctive normal form means that your formula is an AND of clauses. What's a clause? Well a clause is going to be an OR of literals. What's a literal? A literal is going to be xi or not xi. So these are variables. So variables are possible literals and the negations of variables and literals, and that's it. And that is CNF SAT.

So it's a special case of SAT where your formula happens to have this picture. And in general, you can convert arbitrary formulas into conjunctive normal form. It's a normal form, meaning it's essentially unique. A minimal CNF formula is unique. So there's a known transformation to do that, polynomial time. That's how you prove this is hard.

Another view in this picture. We'll use CNF SAT a lot, usually in even more specialized form. But already, one useful view-- which we saw a little bit in lecture one, we'll see it again today-- is you can view such an input as a bipartite graph. You have, let's say, variables on the one side and clauses on the other side. And then you have two types of connections, a clause-- in general, a clause is going to have degree three. And let's say the dashed edges are negated and the solid edges are not negated.

So this is a clause that involves x1 or x2 or not x3. And in general, each of these clauses is going to involve some number of variables over here. I said three, but I haven't actually gotten to three yet. That is 3SAT.

So the most common form of CNF SAT we use is called 3SAT, where it's CNF SAT in the special case where the clause is an or three literals. And you can assume it's exactly three or at most three. So that's like saying the degree of each of the clause notes here is exactly three. And that's the problem with it we used for proving Super Mario Brothers was hard in the first lecture.

So in my notes I have things nicely indented. So we had CNF SAT, which is a special case of SAT, and then 3SAT, which is a special case of CNF SAT. A special case of 3SAT that's also hard is called 3SAT-5. I don't know that this is super standard, but I found at least one paper that gives it this name. This says that each variable occurs in less than or equal to five clauses, either in it's positive or negated form. So this is sometimes called max five occurrence 3-SAT. And I think you can even make them exactly five occurrances if you want. But, at most, five occurrances is usually what you want. Question.

AUDIENCE: Is that tight as 3SAT-4 NP?

PROFESSOR: That's a good question. 3SAT-4? I don't know. I would guess it's tight. Because there are a lot of people that mention five, but I haven't seen a mention that four is polynomial. All right? So we should figure that out. Other questions? Or it's probably the same one. All right.

Here's another special case of 3SAT. Monotone 3SAT. This is where each clause is all positive or all negative. So of course, if every clause is all positive, then you could set all the variables to true and you'll satisfy. If all the classes are negative, you set all the variables to false and you satisfy. But if half the clauses are all positive, half the clauses are all negative, then that's hard. That's called monotone 3SAT. I've actually not seen this used, but I imagine it is helpful in a few situations.

This is definitely the most common. Almost every proof starts with 3SAT. But it's really good to know all the extra things you can assume about your 3SAT problem and it still be hard. I have more hard versions, but before we get there I'm going to tell you about some easy versions. And question.

AUDIENCE: Is monotone 3SAT-5?

PROFESSOR: Another good question. Monotone 3SAT-5, is that NP complete? I don't know. Partly these were done at different times. Monotone 3SAT is mentioned in Garey and Johnson as done in 1978, presumably for a particular hardness proof. 3SAT-5 is probably a very old idea. Probably this idea of reducing occurances goes to some logic thing. But the earliest reference I found was 1998, and most people had forgotten about monotone 3SAT by then. So it probably just hasn't been considered, but it might be easy. Yeah.

AUDIENCE: The construction to limit to five variables-- should I believe work with the monotone construction as well?

PROFESSOR: OK. Conjecture. Yes, it's hard. OK. But stay tuned for a certain answer. So let me tell you some polynomial time versions of SAT. The three is tight if you have a clause being an or of two literals, but otherwise they're just like 3SAT-- or just like CNF SAT, I guess-- then this is polynomial time. And let me give you a rough sketch why.

So if you have an or of two literals, that's something like x or y. And the little bit of propositional logic you should know is the meaning of, let's say, a implies b. Saying this should hold-- and a and b are either true or false-- is the same as saying not a or b. Because if a is false, then the implication tells you nothing. If a is true, then it better be the case that b is true. So either a is false, in which case you don't worry about something, or a is true, and then b better be true, as well. So these are the same-- maybe write triple equals, for these are equivalent logical statements.

So we can apply that here and say x or y is the same thing as not x implies y, or not y implies x. It's symmetric. But this is now just a simple implication. If we ever set x to be false, then y must be true. And every clause can be converted into such an implication. You can build a graph of all such implications.

And then, turns out, to solve 2SAT, you can just pick your favorite variable, xi, set it to true, follow all implications, see whether you get a contradiction. If you don't, then the claim is there is a satisfying assignment where xi equals true. So you can try that with xi true, xi false. If there's any hope, then one of them should say no contradiction. Then just run with that. And you can prove by induction that won't get stuck. It won't make any impossible assignments, unless there was no assignment to begin with. So that's why 2SAT is easy and one of the situations you should be careful about.

On the other hand, MAX 2SAT is hard. So MAX 2SAT is you're given a 2SAT formula-- you're given a 2CNF formula-- and normally, we ask for an assignment to the variables that satisfies all the clauses, but if you just want to satisfy k of them-- so satisfy k of the clauses.

So usually you want to maximize k, but let's say, for a decision problem, I'll give you k and I want to know, can you satisfy k of the clauses? That problem is NP hard. So that can be useful. If you can only represent 2SAT clauses, but you can somehow get a maximization thing in, then you're golden again. But 2SAT alone is not enough.

There are some other easy-to-solve versions. This is essentially a generalization of 2SAT is Horn SAT. It's maybe a little bit more surprising the first time you see it. Again, it's a special case of CNF SAT, which you can tell by my indentation.

Each clause has at most one negative literal. So that means a clause is going to look something like-- has one negative? Sorry. Sorry. Horn is the other way around. We will get-- that problem is also solvable. But the one that's called Horn SAT is-- there's at most one positive literal. So that means you're form is going to look something like not x or not y are not z or w. So there's one positive. The rest are all negated.

And we can do some more prepositional logic. So use De Morgan's theorem. This is the same thing as the negation of the AND. OK? And then we can apply this helpful rule. And say this is the same thing as if x and y and z are true, then w better be true.

And so you can use essentially the same algorithm that you're-- I mean, slightly harder to check. If all three of these things are true, then this one better be true. But it's always a guarantee. You know this thing must happen. Just like in 2-SAT, if x happened to be set to false, then you know y must be true.

So you can just follow these implication chains. If you get a contradiction, you know you're in trouble. If you don't get a contradiction, again, you can prove by induction that all will be well. And so you just make sure every time you assign a variable you don't get a contradiction and you can satisfy any Horn formula. So that's cool.

So I didn't write it on the board, but this is polynomially solvable. There's a symmetric version, which is called Dual Horn SAT. So this is the same thing, but at most one negative literal in each clause. And this is also solvable in polynomial time because you can just negate all the variables in your formula. And then when you get an answer, you can negate all the variables again to get the solution to the original problem. So because you can solve Horn SAT, you can solve Dual Horn SAT.

Cool. One more bad case I'll mention now is DNF SAT. You might say, well, why do we make things ANDs of ORs? What about ORs of ANDs? So DNF is disjunctive normal form, meaning the disjunctions are on the outside.

So the formula is an AND of OR-- sorry.

AUDIENCE: It's an OR of ANDs.

PROFESSOR: Other way. OR of ANDs of literals. We use some shorthand, not-defined clauses here because we don't really use this problem. Because it's polynomial time. Why is it polynomial time?

AUDIENCE: You can just evaluate one of the ANDs.

PROFESSOR: Just evaluate one of the ANDs. If you--

AUDIENCE: If it's true.

PROFESSOR: Right. It's true if any one of these is possible. So you can just check for obvious contradictions, like xi and not xi. If that happens, then that cause is impossible. Throw it away. If any clause has no internal contradictions, then just satisfy the clause. OK? It's basically-- the answer is yes whenever there is a clause. It could be of the empty formula, no clauses.

So writing-- you can also write any formula into DNF. It's like an enumeration of all the true possibilities. But it takes exponential time to do so. So it's a funny asymmetry between AND and OR. That's life. Yeah.

AUDIENCE: I just want to say, for Horn SAT, even if you don't have Horn SAT or Dual Horn SAT, you might have a formula where some renaming-- not necessarily all of the literals-- but renaming just some of the literals will put it in Horn SAT. And that's also in--

PROFESSOR: Oh. Just negating some of them.

AUDIENCE: Yeah. Like, each time a variable occurs, you have to negate all the instances of that variable. But these are called renamable Horn formulas. And it's also-- finding the renaming is linear time.

PROFESSOR: OK. So some kind of renamable Horn. And by renaming, you just mean negating, right?

AUDIENCE: Yeah. The term used in the literature is renamable Horn.

PROFESSOR: There exists a negation of, let's say, some subset of the variable's x such that Horn. OK? That's the very concise version. So you get to choose some of the variables to negate and make it a Horn clause that's also polynomial time. Cool. Thank you.

You might be wondering at this point, how much-- I mean, do I have to remember all of these? And sometimes the answer is yes. But there is actually a dichotomy theorem that will tell you which versions of SAT are polynomial time and which versions are NP hard, and we'll cover that in one more page. But I'm-- well, all of these are involved in the statement of that dichotomy. Not quite all, but most of them. So it's not exactly a shortcut.

I would say a lot of the time, the problem you're working with does not naturally map onto ANDs and ORs. It sort of involves bits of some sort-- there's a 0 and a 1 notion-- but they may not really correspond to logical notions of true or false. And the operations you can do on them may not correspond to AND or OR, or anything nice like that. So the next two versions of SAT are in that spirit.

So this is usually, these days, called 1-in3 SAT, but originally it was called exactly-1 3SAT. So this is going to be a little bit weirder to write down, but like CNF SAT, the formula is the end of a bunch of clauses. So that part's the same. But now we're going to make a clause to be a relation on three variables, which is that exactly one of, let's say, xi, xj, and xk is true. So this means it could be true-false-false, xi is true, but the other two are false, it could be xj is true, the other two are false, or it could be that xk is true and the other two are false. But those are the only happy assignments. Question.

AUDIENCE: Is it exactly one of three variables or three literals?

PROFESSOR: Good question. The original statement is its variables, so that's how I wrote it. This is usually called monotone. These days it's usually called monotone 1-in-3SAT. I don't know how usually exactly. Sometimes it's called all positive 1-in-3SAT.

So let's say literals equal variables. You could, of course, if you want, consider a more general version where you can have negations. But you don't need to, so why bother. That fact is usually forgotten in most proofs. So you'll see in the literature a reduction from 1-in-3SAT with negations, and they have a negation gadget. It's like, you don't need to have a negation gadget. So why not skip it? But there you go.

Now I'll just mention I'm not a fan of the word monotone here because here we have monotone to mean all positive or all negative. Here we mean all positive. Not ideal reuse of terminology. I think that's why sometimes this is all positive 1-in-3SAT. Anyway, it's a bit of a mess. But that is the state of the literature. So you get it all.

All right. Here's another problem. Monotone not-exactly-1 3SAT. I should not have any suspense here. This is NP. OK? This is NP-complete. This is also NP-complete. But not-exactly-1 3SAT is polynomial. So I think you know what it means. A clause specifies that-- again, we take an and of clauses. And we want zero, two, or three of three variables are true. In other words, exactly one of them is false. No. I don't mean that.

Exactly one of them being false would be 1-in-3SAT again, just by negating everything, which we're allowed to do if we want to. But this is different. This is saying, it could be everything's false, or it could be one thing is false, or it could be zero things are false. But not two things are false. OK? This turns out to be polynomial. And do I have-- oh. There's one funny thing here, which is if all your clauses look like this, you can set all your variables to false. So this is sort of a trivial problem.

But to make it more interesting, you can say x1 equals true, just to get you started. So there's no trivial solution then and it still turns out this is easy. Because if you think about this long enough, as I did yesterday, this will look something like-- if you have three variables, either they're all false-- then fine-- or if one of them is true, then you better have another one true. That's a way of saying if there's at least one of them, there better be at least two. That's what we want. This has to be true for all shifts of i, j, k. So for each of i, j, k, if one of them is true, you want to imply the OR of the other.

And this is the same as NOT xi or xj or xk. Don't need the parentheses because it's associative communicative. And that is a dual horn clause. And that's why this is polynomial. Question.

AUDIENCE: I think I'm confused about the definition. Why can't they just all be true then?

PROFESSOR: Good question. Let's say x2 is false. I should double check. I don't remember that in the statement of the problem. We do not allow negations here. Once you allow negations, this trick won't work. But if these appear all in positive form, then we can convert into the single negative and get dual horn. Question.

AUDIENCE: Are you allowed to mix the zero, two, and threes? Or does it have to be all the clauses have to be--

PROFESSOR: All causes look like-- all clauses say, zero, two, or three of these three variables must be true. You can't have a clause that says zero of these are true and two of these are true.

AUDIENCE: No. But if you have two of these clauses that give you all those choices, can you choose one to be zero and then have another one be two, or whatever.

PROFESSOR: For each clause, it's an independent choice whether you have zero, two, or three of the variables true. Yeah. So this OR is local to the clause. Other questions? So it's still an AND of things that-- it's just we have a weirder relation. Instead of just taking the OR of a bunch of things, which would be saying at least one of them is true, now we allow zero or two or three of them to be true.

OK. One more version, then we'll get to-- well, one and a half more versions-- then we'll get to the dichotomy theorem. So x1 is Not-All-Equal-3SAT. I feel like that's about all I need to write down, other than the fact that it is NP-complete. But just in case, what this means is a clause is something like Not-All-Equal of three variables again. And this is what I'm defining is going to be the monotone Not-All-Equal-3SAT-- which is also hard-- where these are variables, not just literals. So no negations in monotone Not-All-Equal-3SAT.

Again, the original proof already had monotonicity in there, so there's no work to be done. Cool. So not all equal just means that they're not all the same values. So that means not all true and not all false. Not TTT and not FFF. I really like this version of 3SAT because it's completely symmetric between true and false. I mean, not at the clause level. Every clause has to be satisfied. Those are ANDed together in the logical level. But the xi's are treated completely symmetrically between true and false. You could just call them red and blue. There's no reason to think one is true one is false. You just can't have them all be the same color within a clause.

OK. So you could think of it as a problem on hypergraphs. Three uniform hypergraphs, you have all these triples of things. You just want them to not all be colored the same. So it means two of one, one of the other two trues and one false, two reds and one blue, two falses, one true. They're all this-- those are all good cases and these are the bad cases. Cool.

AUDIENCE: So this is one or two in 3SAT?

PROFESSOR: Yeah. You can think of this exactly one or two in 3SAT, if you want to phrase it in this style. OK. So ideally, you should remember all of these. But I'll tell you the most important ones are regular 3SAT-- that's at least one of each thing is true-- exactly 1 3SAT, or 1-in-3SAT-- where exactly one of the things is true, and adding more things breaks it-- and Not-All-Equal-3SAT. Those are the three important ones to know from a lower bounds perspective. These others are to, like, be careful that you don't fall into one of these things that is polynomial.

So occasionally MAX 2SAT is the one other that would be useful here. But remember these guys. They're super handy. Because what will happen when you're proving hardness is you fool around and you try to find-- you build a gadget that has two truths-- two possible ways to satisfy it. Call one red and one blue or one true and one false. And then you try different ways to combine three of them.

And you're trying to get-- you need some other things-- but you're trying to get clause gadgets. Trying to get them to-- when you combine three wires into a little gadget, you want them to be constrained somehow. That in order to be globally OK, something must hold locally at those three things. And it might end up being a Not-All-Equal constraint, it might end up being an exactly 1 constraint, or it might end up being a 3SAT constraint. With some negations to make it happy, it should be one of those to be hard. If you fall into something like this, then that's not good. Question.

AUDIENCE: So let's say, since this Not-All-Equal thing-- let's say you call it red and blue. What if you add green? Then would there be-- would Not-All-Equal 3SAT be NP-hard, or would you need 4SAT or something?

PROFESSOR: Yeah. So what about ternary truth? I-- there might be a problem on that in the PSET. But in general, you would have to go through the work to check which problems. I think those are pretty uncommon. So usually what you do is if you have a gadget that can be solved not two ways, but four ways, is you call two of them true and two of them false and hope they behave more or less identically.

So that's the most common answer, practically, to what we do. But it certainly is plausible with three different values. Some of these are going to be hard but I don't know which ones. Hopefully all them, but you have to be careful. And definitely the next theorem I'm going to talk about-- the dichotomy theorem-- would get more complicated with three colors. Nice question.

So let's do Schaefer's Dichotomy Theorem. This is about which versions of SAT are polynomial and which versions are NP-complete. With the right set up, every problem you can think of is either polynomial or NP-complete. There's no things in between. These are called NP-intermediate problems. So it's always going to be one extreme or the other, as I'm about to set it up.

And this theorem is by Schaefer. And in the very same paper he proves Not-All-Equal 3SAT and 1-in-3SAT are hard. Those are the original proofs. So it's a great paper. I have looked at it many times. It's from 1978. So long time ago. But still quite readable.

So I don't know how much-- all the last problems we've stated have this property, but I'll make it explicit again. That's your formula is going to be an AND of clauses. And now we're going to allow general kinds of clauses. A clause is just going to be any relation on some number of variables. So there won't be any notion of literal here because you can put that in the relation. I'll call this a general clause and say relation on some variables.

So relation is something-- I give you a set of truth values for those variables and we'll say yes or no. That's valid or it's invalid. You can think of a relation as the set of all assignments, their variables, and make it true. But you don't have to specify that, per se. You just sort of know what it is.

OK. So I mean, in particular, it could be the OR of three variables, then we get 3SAT. Or it could be the Not-All-Equal constraint on three variables, then it's Not-All-Equal 3SAT, and so on. OK. We are going to-- so I guess, sorry. The relationship should be given to you as a Boolean formula. So it could be an OR, or you can write 1-in-3SAT as a Boolean formula. It's just little tedious. You could say, well it could be this, or it could be this, or it could be this.

In general, I'm going to assume that they're given to you in CNF form. Sorry, that's redundant. CNF has form in it. Because any formula can be made into CNF. So now CNF is an AND of ORs, so this is going to be an AND of what we might normally call clauses, but we're already in a clause, so I'm going to call this subclauses. Starting to sound like legalese. I made up this word. It's not in the literature.

So in general, your formula is an AND of clauses, each one is your sum relation, which we're going to think of as an AND of subclauses. Of course, it's really just an AND of all the things. But this is trying to be general. Because we're going to have constraints in the clauses, in particular.

So then claim is SAT-- on these types of formulas-- so here's the difference, I guess. To define the problem, you specify what kind of relations that you allow. So in 3SAT, we say, OK, so OR of three things. In CNF SAT, it's an OR of k things for any k. In Not-All-Equal 3SAT, it's not all equal of three things, and so on. So we give that up front and then the decision problem is well, I have n variables and I can combine them with these clauses however I want. So we need some kind of infinity, right? If I gave you a specific problem, then it's not going to be NP-hard. Like, with these 10 variables, that's never going to be interesting.

So I give you the notion of what causes are allowed, what relations are permitted, and then I want to consider the class of all possible formulas you can build with clauses of that type. So you can think of this as really more of a clause type, like Not-All-Equal, just to be precise here. And then we get a version of SAT. And it's going to be polynomial if one of four cases happen. At least one. Any one of these will make it easy. We have seen almost all of these.

So first one is setting all variables true satisfies the formula. Well not just the formula, but all formulas of this type. Or all variables false satisfies all clauses. OK. This is a statement over all formulas with this clause type, right? So, a statement about the clause types. And it's one of the issues we were having with not exactly 1 3SAT, because there the clause type allowed everything to be false and it also had everything to be true, so it was doubly bad.

But of course is if you have clauses where this is true for all the-- you could allow different types of clauses. You could have Not-All-Equal plus 1-in-3SAT, that will also be hard, of course. But if all of your clauses have this property, then of course, you just globally set the variables and you're done.

OK. That was the first case. All right. So the next one is that it could be the subclauses are all Horn, or all dual Horn. So those are two happy cases we saw before. I mean, we can think of the overall problem as an AND of the clauses, which are ANDs of subclauses, so if everything is a Horn or dual Horn thing, then we're happy.

And next case is the relations are all 2-CNF. So this would be the 2SAT case. If all the relations you're working with are in 2-CNF, then when we AND them together, you still have a 2-CNF formula, so you can solve it by 2SAT.

So these are all things we've seen. There's one more case we haven't seen. Is there a question? Yeah.

AUDIENCE: Isn't 2SAT just a subcase of the Horn and dual Horn thing? Because you're always going to have--

PROFESSOR: It's true. 2SAT is a special case of Horn.

[INTERPOSING VOICES]

AUDIENCE: You could have one with two positives and another one with two negatives.

AUDIENCE: Yeah, that's true. Yeah.

PROFESSOR: Ah, right, right, right. So I see. So 2-CNF, some of the clauses are going to be Horn and some of them are dual Horn, I think. So it doesn't fall into this because it's not all one or the other. In general, those combinations are bad. But 2SAT is always OK. Good question.

OK Last case is some linear algebra. This is one other easy case of SAT which doesn't come up very often, so I didn't write it as a separate one. So imagine equations like this. I take some number variables, I x OR them together and I say, that should be 0, or similar thing, I set it equal to 1. Those are what I would call linear equation over Zmod2. Because in Zmod2-- the finite field in two elements-- addition becomes x OR and there's no multiplication here because it's a linear system.

So we can solve these things because z2 is a finite field. We can use Gaussian elimination if-- even if I have a whole bunch of these equations, I can solve them all using Gaussian elimination. Or determine that they're unsolvable. So that's another easy case for SAT to be careful about.

And the theorem is if you have one of these situations-- so you can't mix these. If you have one clause of this type and another clause of this type, your problem will be NP-hard. So in general, you say, otherwise SAT is NP-hard. I guess it will actually be NP-complete here, the way we've set it up. Well, assuming the relations are checkable.

So these are the only cases. This is an easy case, this is an easy case, this is an easy case. It could be that multiple of these things are true. Maybe your 2CNF and your all Horn. That will also be polynomial, of course. But if none of these individually hold, then your problem is NP-hard. Question.

AUDIENCE: So how does this generalize from non-Boolean fields? I'm sure the last one is also still true.

PROFESSOR: Yeah. So we can go back to your question about three colors, and the answer is I don't know. As far as I know, there's no theorem of that type. But there might be one. It's been 30 years. So it wouldn't be surprising. Certainly, you can-- some of these positive results will generalize. But I think even this one would be a little tricky.

AUDIENCE: So MAX 2SAT doesn't fall in this?

PROFESSOR: Right. So here the goal is always to satisfy all of the clauses. It's always the AND of all the clauses. You could imagine a MAX 2SAT-like theorem. My guess is most problems will be hard. But as far as I know, there's no such theorem. Yeah.

AUDIENCE: Is there any way to understand this theorem as making a geometric statement about the relation being convex and hypercube or something? Like, is there any sort of convexity property encoded in this?

PROFESSOR: I don't know. I would guess no. I know there is a more modern take on this that is more algebraic. So it's more like, if you start with these things and you can build up in this way, anything you can build up in this way are the polynomial solvable versions of SAT. Anything you can't build up in this way is NP-hard. So if you're interested in that, check the Wikipedia page for Schaefer's Dichotomy Theorem. But I don't think there's a geometric interpretation. This one, obviously, has a geometric interpretation. But I think the others not, would be my guess. Yeah.

AUDIENCE: So does this say something about the complexity of recognizing the clauses if you interpret them as a language?

PROFESSOR: If I give you the formula that specifies the types of clauses, can you determine whether any of these is the case. I would guess yes, but I don't know of such a theorem. So another good question. So many questions to think about here. It's definitely not explicitly mentioned, that I saw, in the Schaefer paper. But it's been around for a while so people may have thought about that more. It definitely can be a little tricky to check which things are of this type, practically. So it would be nice if there was an algorithm.

I would say-- so again, practically speaking, there was one hardness proof I was trying to generate gadgets computationally. So just enumerate all possible gadgets of a certain size for my problem, and then see what formula they were representing. And then we would take that formula, do a Karnaugh map-- if you've ever done digital logic stuff-- and then from that, you get a nice, minimal form. And then we would-- usually we could just look at the map and say, oh, that's just equals, or that's not equals or something. We were hoping for-- we were dreaming for-- one of these things or 3SAT. We never got the gadget we wanted.

So I think, with a Karnaugh map, you could do this. But that's exponential time.

AUDIENCE: Yeah.

PROFESSOR: So I don't know for sure. I should probably check the algebraic view. That might--

AUDIENCE: That's a clever approach though.

PROFESSOR: Yeah.

AUDIENCE: To do things computationally.

PROFESSOR: It's definitely helpful. Because of course, computationally, you can only look at small gadgets. But hopefully there is a small gadget and then a nice proof. So why do the hard work of generating them yourself when the computer could do it for you? It doesn't work for all problems. Your problems need to be locally isolatable. To not worry about the big picture. Other questions? Cool.

Well that's all the versions of SAT you need to know. Because here we have the universality theorem. There will be another one or two that we bump into but these are the things you should all know. It's really helpful when doing a proof to not have to worry about which version 3SAT you even are going to use, and just know that these are all out here. So that when you find a gadget that happens to match one, and then you say, oh, well I meant to do a reduction from Not-All-Equal 3SAT.

That's why I wanted to tell you all these, although I know it's a lot to take in all at once.

Let's do some reductions, finally. NP hardcore time. So the first one, I had actually never seen before, but it's in Schaefer's paper. So I thought it'd be fun to cover. Here's a problem, which is NP-hard, and we will actually prove this one NP-hard. 2-colorable perfect matching. Let's say you're given a planar 3-regular graph, every vertex has degree 3. And what you'd like to do is 2-color the vertices, a red and blue, such that every vertex has exactly one neighbor of the same color.

OK. So if you look at a vertex and it has three neighbors, then-- let's say we color this guy red, there should be exactly one neighbor that's red. And so you can think of this edge as being red, and then the red edges will form a perfect matching in the graph. Every vertex will be [? instant to ?] exactly one edge. So that's the 2-colorable perfect matching. It's kind of-- well, sorry. The red edges form perfect matching on the red nodes, and the black-- or the white edges, I guess-- form a perfect matching on the white nodes, black nodes, whatever. So it's like two perfect matchings, one in each color class.

So this is a nice problem. You can think of it as SAT, in a sense. It's, again, just a local constraint on the notes. And so you can think of this as being a clause involving those four guys. I think, unless I did something wrong, you can think of it as 2SAT and 4SAT, or a special version of 2SAT and 4SAT. Because-- is that right? Yeah.

So let's say red is true. So what we're saying is if this guy is true, among these four nodes there should be exactly one other one that is red. On the other hand, if this is black, there should be exactly one of them that is black. And so the other two should be red. So in all cases, it's exactly two and four of them are red. And it's symmetric between red and black, so that seems good. So this is a special case of 2SAT and 4SAT. In case you were wondering whether 2SAT and 4SAT is hard, here it is. And I have the original reduction by Schaefer here. Do I have any notes? No, that would be too easy.

So here's a gadget and here's another gadget, and then they're pasted together. And I should mention-- so Schaefer claims that if you have a planar 3-regular graph, this problem is hard. But he doesn't prove it. He just proves it for general graphs, so I'm only going to prove it for general graphs. Maybe we can think about planar 3-regular case, but not right here. So this will just make a graph instance to that problem.

So this gadget, this is a k4 on-- and we're only distinguishing x, y, and z. And it has to form-- there's going to be one red edge and one black edge. So maybe like this, and like this. Or like this, or like this. It's going to be a rotation of one of those assignments. So I believe the claim is x, y, and z, just looking at those three vertices, should be not-all-equal. OK? If two of them are red, by symmetry-- there's lots of rotational symmetry here-- so maybe two of them are red and one is black. Then this guy can be set black and you're happy. It's actually forced for that guy to be black.

If two of them are black, then these two must be red. And if all three of these are black, you're toast. Because you should have two of each. And if all three of them are red, you're in trouble. So this is a not-all-equal clause gadget for this problem.

So we're going to reduce from Not-All-Equal 3SAT to 2-colorable perfect matching. So we're representing a Not-All-Equal clause like this. And now what we need is the ability to copy data, right? So these are three variables that-- at the moment, yeah, they can be red or blue. But what we need is that the same xi can appear in multiple clauses.

Because we have a bipartite graph. If every variable appeared in only in one clause, the problem would be really easy. So that's what this gadget does. The claim is this gadget copies a value. And this, I think, requires-- so it says that these two guys must have the same color.

And so what you do is you just have-- for each Not-All-Equal clause-- you have one of these Not-All-Equal gadgets. And then whenever you have two variables that are supposed to be the same thing here, it's x and x. In our terminology, xi and xi. Then you're just going to connect them with this gadget. And that will force them to be equal.

Or over here we have y and y. So r here is Not-All-Equal. In our terminology, it'd be NAE, xxy, and yzu. This would represent that formula.

So the thing to check, which I will leave as an exercise, because it seems-- at least, I couldn't find a clean way to do it. It seems a little bit tedious. That this forces equality between the two ends, not providing any other constraints.

So that was a simple proof. One of few simple proofs. Still some cases to check. Yeah.

AUDIENCE: Wait, you said those general graphs, they don't [? tap in? ?]

PROFESSOR: Right. This is general graphs. So you might say, what about planar 3-regular graphs? Planar Not-All-Equal 3SAT-- when this graph is planar-- when the bipartite graph is planar-- it's actually easy to solve, polynomial. So you can't reduce from planar Not-All-Equal 3SAT because it's easy.

But I would guess that in this situation-- and we just proved this is a more general problem than Not-All-Equal 3SAT-- what we would need at this point is a crossover gadget. So that when-- and this thing is going to end up with crossings-- if there's a gadget that just communicates the information across the crossover, without any other constraints, then we can just plug that in and get rid of all crossings. Then we have a planar graph and that would prove this part.

And then the other part is that we have high degree nodes here. And so I'm guessing Schaefer had in mind the gadget that takes a high degree node and splits it up into lots of little lower degree nodes-- degree 3 nodes-- that simulates the same effect. But I don't know either gadget. But that would be my guess on how to-- that would be the obvious approach of how to proceed to get that theorem. Yeah.

AUDIENCE: The other gadget is just [? 1b ?] split into degree 3 copies connected by that gadget.

PROFESSOR: OK. Good. One gadget down. Now we just need a crossover. Other questions?

All right. I want to talk about two families of problems-- a proof hardness for two families of problems-- next. One is called pushing blocks. These come up in lots of different video games.

One of the first, I think, is called *Sokoban.* This goes back to 1984 and this is, I believe, the original CGA graphics for *Sokoban.* And so you may have played *Sokoban.* I think it's in Emacs. It's all over the place. Tons of implementations on it.

You are at a warehouseman, that is what sokoban means literally in Japanese. And you have these boxes. They're all one by one boxes. You are one by one. There are some bricks which cannot be moved.

You have some target locations and your poor job is to move all these boxes into the target locations. Or rather, every target location must be covered by exactly one box. Boxes can overlap each other, you can push a box, but you can only push one at a time because you're not that strong. So you could not, for example, push left here.

But in general, your inputs are up, down, left, and right. Up, up, down, down, left, right, left, right. And you can only push one block at a time. So if you push a block into a corner, it's never going to move again. So often you have to hit reset. There's a lot of ways to get stuck. But this is a solvable instance. I think it's level five in the original game.

So this got started a huge family of problems. There are tons of other video games that have pushing blocks problems. This is one, in *Legend of Zelda: Minish Cap*, which I think is a Gameboy Advance game. It's been a while since I played this one.

But here is a level where you have-- it's in perspective a little bit, but this is really a 2D problem. Everything is one by one. Believe me. You have one by one blocks. You're on ice.

So if you ever push a block, it will slide off to infinity unless it hits something else, like a block. And your goal is to get a block here. OK? Anyone see how to do it?

AUDIENCE: It's like *Ricochet Robots.*

AUDIENCE: Can you walk without sliding?

PROFESSOR: It's a lot like *Ricochet Robots*. Yeah. You can walk wherever you want. There is no block. So there's lots of free space in this case.

So you can push this guy down and then push it over and then push it up. It's actually not that hard. At some point I was thinking, well maybe I should stack up three here so it ends up-- but no. It's just three pushes and you're done.

And if you watch the YouTube video that's linked from this, you'll see him spend like 30 minutes until he finds the right solution. In the game, it's frustrating because once you push them off, they disappear and you have to leave and come back. Anyway.

So out of all these games, we've defined a bunch of models. Here's one aspect of the model. So we have the fixed blocks, which are in red, the movable blocks, which are in cyan here, and the robot is blue. Everything's one by one.

And in one model, which we call Push, when you push a block, it moves one step. That's the normal model. And that's like *Sokoban.* And I'll talk about the two in a second.

In PushPush-- this is like the on-ice version. So you're not on ice. You can still kind of move and then kind of counteract physics and just move one step. But the blocks, they'll just fly off until they hit something. So here, this block will fly until it goes there.

In PushPushPush, everything is so slippery that if you hit a block and it hits another block that's movable, they will all just keep going until they hit an immovable block.

AUDIENCE: But you don't slip.

PROFESSOR: You still don't slip in any of these models. So there's another version where you slip, which has not been considered so much. Although *Ricochet Robots* has been considered and that's in that genre. Yeah?

AUDIENCE: Is there a model where the block you first pushed stops and transfers its momentum to the other--

PROFESSOR: Oh, cool. Yeah. Conservation momentum. So this one hits then the next one goes. That's probably--

AUDIENCE: PushStopPush.

PROFESSOR: PushStopPush. I would guess that's also hard because this ends up being the same proof. I'm guessing that works, but we would need to check. That's a good-- yeah. The push-- push ricochet push.

OK. One other thing here is the number two. You'll notice only two blocks are moving. And normally here, two is the strength of the robot, meaning if there are up to two in a row, you can push two blocks, but once there are three in a row, you can't push at all.

The idea here is the same thing happens. So they're just too heavy. After you get up to three of them, then they'll stop sliding. And same with PushPushPush.

AUDIENCE: So *Sokoban* is actually Push-1.

PROFESSOR: *Sokoban* is like Push-1. It's a little bit more complicated, as illustrated in this table. So *Sokoban* is here and you can only push one block at a time. There are fixed blocks.

And in general, the models that have fixed blocks are highlighted in pink here. The slide thing is trying to capture whether it just moves one step or all the way until it can't anymore. That's the max.

And there's only a couple of slide versions. I haven't put PushPushPush here because that would be the only difference-- is what max means. But everything else you just move one step.

Then the other issue is, what is the goal? I mentioned in *Sokoban* you have to cover every storage space with a box. That is the only problem with that flavor, in this list anyway. All the ones that are called Push-- the goal is just to get the robot to a destination, like in *Mario* or *Zelda* or something.

So that's the difference between *Sokoban* and Push-1. Well Push-1-f, I suppose, is pushing one thing at a time with fixed blocks. That's identical to *Sokoban* except for this issue of what the goal is. Either just to get from s to t-- the robot-- or to get all the blocks into a particular configuration.

OK. Well there are some other things here. We've talked about Push-k. Push-star is when k is infinity. So when you push, you can-- your arbitrary strength-- you can push as many blocks in a row as you want. PushPush-k, PushPush-star, then.

We've talked about f. f is when you have fixed blocks. Push-1-f, Push-k-f, and Push-star-f. And then there's one other variation here that's been considered, which is the x. This is when the path that the robot takes must not cross itself-- must not revisit a square.

This is-- there a lot of video games where after you leave a square, that square disappears. It falls down into the abyss and so you're scared. So to represent those games-- there's another reason we did that, but I'll get to that in a moment-- there's Push-k-x and Push-star-x.

OK. Now let's talk about complexity. That's the right two columns here. The reference and the complexity. So all of these problems are NP-hard. OK? But there's this issue-- are they in NP or are they PSPACE-Complete? Question.

AUDIENCE: Sorry. What did you say a fixed block is?

PROFESSOR: A fixed block is a block that cannot be pushed. So it's just glued to the ground.

AUDIENCE: So this is just whether or not there are some blocks that are fixed.

PROFESSOR: Right. In these problems, this is a more general version than this game. Here, you allow some blocks to be specified as fixed. Here, everything is potentially movable.

Although if you have more than k in a row, it's like a fixed block. If you have a k plus 1 by k plus 1 block, that's fixed, effectively. But you have a resolution issue that you can't make tiny fixed blocks in this model.

And in this model there are no fixed blocks whatsoever. But here you can specify some of them are fixed. A lot of versions.

AUDIENCE: So push-star has a boundary? Otherwise you could just--

PROFESSOR: Push-star does live in a rectangular box. Yeah. That is the one.

AUDIENCE: Is it fixed?

PROFESSOR: You could think of that as fixed blocks or not. If nothing is fixed, then you can just walk off to infinity. Move all the blocks away and then come back and find your destination.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Right. So they're all NP-hard. There's a few that are known to be PSPACE-Complete. Push-Push, with a fixed strength, is PSPACE-Complete. Or with fixed blocks, I think, should also be PSPACE-Complete. That's not written here.

But Push-Push-Star, all we know is NP-hardness. Push-k-f, where k is at least 2, is known to be PSPACE-Complete. But Push-1-f or Push-2, without the f, are both open.

And the reason we were interested in a non-crossing path is that forces the problem to be an NP. Because then you know the solution path has polynomial length. You can visit each [? grant at ?] most once.

So the hard part here was to prove that it's still NP-hard, even with non-crossing paths. We won't try to prove that today. So those, we actually have tight bounds of

NP-complete. And *Sokoban*-- there's a relatively old result-- 1998-- that it's PSPACE-Complete.

OK. Let's do some reductions. So this first reduction is amazingly cool. It's by Michael Hoffman, 2000. This is Push-star with a rectangular box.

OK. So you're here. The outlined regions are the blank space. Everything else is filled with a block. But every block is movable and you can push arbitrarily strongly.

So this is going to be variables. You're going to make some choices here. These are the connections between variables and clauses.

This is a bipartite graph encoded in binary in the obvious way. There's variables here, clauses here. There's a hole exactly when that literal appears in that clause. OK? There's actually two rows per variable, the true and the false.

And then there's this gadget here to connect the things. And then these are the clauses where we're going to check that they were all set correctly. So let's go through it. This is the schematic diagram of what I just said. We're going to start here in the variable block and walk through each of the variables. Here's what the variables look like.

You have two rows, the xi row and the xi complement row. That's another way to write NOT xi. And you're going to count, how many times does xi appear in any clauses? In the positive form, negative form. That's called ni and ni bar.

And you're going to measure out these lengths. You can negate variables to make sure that ni is always bigger than ni bar. So do that.

So in this case, ni is this big, ni bar is this big. Measure it out. This is your blank space.

And what you're allowed-- what we will show you you're allowed to do-- is either move up here and move all the way over here and then up. Or to move here and all the way over and then up. And this star is so that the one block that's here could fit.

And then you can go over to this x. In general, in these gadgets, the x is your target. x marks the spot. All right? For the gadget. And then there's a global x, which will be at the end of the clauses.

So what does this do? Well if you think about this connection block-- this is the bipartite graph encoded as a matrix-- the number of blank spaces over here is exactly ni. So this is why, after you push ni steps or ni bar steps, you'll have to stop. Because there are no blank spaces to the right of you. Then the only thing you can do is go up because there are no blank spaces below you.

So in general, these gadgets are super tight because above you there's nothing. It's all blocks. And below you and left of you, it's all blocks. There's nothing you can do.

You put the next variable gadget here, so then that remains true. In your row, and in your columns, you're completely packed. So the only choice you have is to do this or to do this. And you will fill the row that you choose. Exactly. OK? So that's the variable gadget.

AUDIENCE: What prevents you from choosing both?

PROFESSOR: You could choose both. Or you could do a little bit of one and then do the other. That's true. But as we'll see, that only makes your life worse.

AUDIENCE: OK.

PROFESSOR: Yeah. Good question. So next, we enter the bridge gadget, which is these two pictures and it looks like this. This is basically a locking mechanism.

So you start here-- I'll just tell you're supposed to do. You walk over through this blank space, then you push all of these things down to here. So this basically prevents you from going back to where you were. So you push all that down.

And then you go over here and you tunnel down-- so I think you're moving these blocks over to here. That's another kind of lock. And then you're pushing this stuff down to here. And then you get there.

So when all is said and done, this will be down here, this will be down here, this will be over here. And so when you're in the clause block-- again, you have full rows to your left, full columns above you. So there's nothing-- you can't go up and you can't go left.

That's the purpose of this gadget-- is to connect this thing to this thing. If there was a teleporter, it would be much easier. We could just leave these all filled. But we want to get up to here, but make sure this is all filled at the end. So that's the sole purpose of this gadget.

OK. Now a clause. How do we do a clause again? So you have-- there's three spaces here because this is 3SAT. You can move down one here, and then move over two. These two blocks go here. And then you can get to the x.

Or you can go down here and over and get to the x. Or you can go here and down, over. And this is possible if there's a hole below you. This is possible if there's a hole below you, and this is possible if there's a hole below you. These are aligned with these things.

So with exact-- I didn't quite align them, but these three columns are these three columns. And if one of these is unfilled, you'll be able to get from here to here. If they're all filled, you won't. Because it will be full columns all the way down. So that's a clause gadget because things were filled when you chose that thing to be-- maybe I-- did I get it backwards?

AUDIENCE: So you're really choosing the opposite.

PROFESSOR: Right. You're choosing the thing to be not true and leaving the other one to be as true as possible by not pushing it. So it leaves the hole so that later you can traverse--

AUDIENCE: So that's why you wouldn't want to choose both.

PROFESSOR: Right. If you chose both, that would be like making them both of them not true, and so you don't get any of the benefits. OK? That's Push-star. Cool?

OK. Let's do Push-Push-1 in 3D. This is really easy. This is almost like the *Super Mario Brothers* proof. So it's just old drawing style.

And we've drawn sort of the dual graphs, so these paths are little width-1 tunnels that you can walk down. So you start up here and you can either push this thing this way or this way. And so you're choosing which way you'll be able to traverse.

Either you can go the true way or the false way-- the opposite of wherever you push that block. And so that cuts off one of the things. Then that path is going to be connected to all of the clauses that it satisfies. That that literal choice satisfies.

Then there's this gadget to prevent you from wrapping around to the other side. Whichever one you come down, you will block off the other path. And then you do that again for the next variable, the next variable, and so on. At the end of the last variable, you run through all the clauses.

So how do the clauses work? Very simple. If any of the literals that satisfy the clause were visitable, then you could push this block over. And then later, when you visit the clause-- at the end, when you visit the clause and try to traverse it-- if there's nothing here when you push this block down, it will go all the way to the bottom and you're trapped, never to get to the finish line down here.

But if at least one of these was in, then it will block this guy and you can-- if that's true for all the clauses-- then you can get to the destination. So these are just clause checking. And that's a very straightforward 3SAT proof.

In fact, the previous proof is based on this one. And the next proof is based on this one. And the Nintendo proofs are all based on this one. This is sort of the prototype. Yeah.

AUDIENCE: Do you have the crossover gadget that--

PROFESSOR: So this is 3D.

AUDIENCE: Oh, oh.

PROFESSOR: So there's no crossover. But in 2D, we want to get a crossover. So here's how we do 2D. And this will work for both Push and Push-1-- Push and Push-Push. The only place we're using Push-Push was this clause gadget, so let's first get rid of that aspect.

So here's something called a lock, and you're going to have to believe a little bit here. Your goal, let's say, is to get from a to b. And this is what happens if you try to go from a to b directly.

There isn't much that you can move-- well, you can maybe move i down, j left, and like this, but-- yes. You can move i down, j left, e down, f left, b down, but not c left, because d is in the way. OK?

So you can't get from a to b. But if you visit from u, and you push this block out of the way, then you'd come back through a, then you can do--

AUDIENCE: Push them over.

PROFESSOR: You can do these things. Push them all over. And then you have room to push c over, and then a can go all the way down, and you're through. So this unlocks the lock. Then allowing a to be traversal later.

AUDIENCE: And going backwards from v to u would lock it again?

PROFESSOR: No. This is not a reversible gadget. It only works once.

AUDIENCE: [INAUDIBLE].

PROFESSOR: You can't unlock. Once l's down, it's permanently there. So now we're going to use this in a clause as follows. So here's the lock and then there's this schematic above it.

So we have three possible entries. This is xi or NOT xj or xk. And we're going to use this gadget to say, well if you come down the true path from here, you're going to have to push this block down, which prevents you from using the other half of the gadget.

So either you move y or you move x. From then on, you can only do west to south traversal or only north to east traversal. So it blocks the other path. So this is-- because I don't want to come down here, unlock the lock, and then go back on a different path.

Because that's not something that's necessarily true. So as you come down one of these, you force these gadgets to be in a particular state that will only let you come back the way you came-- go back the way you came. And then you can unlock the lock and then this just connects back. OK?

Then later, when we're checking the clauses-- when we come through all the clauses to make sure that they're true-- we're going to route those from a to b, to a to b, to a to b for all the locks. And if they're all unlocked, then we'll be able to traverse them, and otherwise, not. That's the idea.

OK. Then we have the issue of a crossover, if we want to go into 2D. So we have-- here's a basic crossover in the Push-1 model, or Push-Push-1. If we're going to go from north to south, then we will go-- OK. We can push this down. But we won't be able to go to the east then. OK?

And we're not able to go to the west because if we push this block, it gets stuck. So we can go north to south, that's fine. And we can also go west to east, by symmetry, essentially.

But you can't go from west to north or west to south, or any of the other combinations. This works as long as you do one or the other. Once you do west to east, you can't even do north to south. So that's why we call it an XOR crossover. It's not what we want.

But if we combine things in this way, we get a uni-directional crossover. This is one where-- I better check my notes-- you can do with one of three things. You can do north to south, if you want. And then you can do west to east.

So you could just do west to east. Or you could do north to south, then west to east.

Or you could just do north to south. Those are all possible here.

So when we do north to south, we prevent this particular thing from being traversed. But we unlock this gate, which later, when we go west to east, will allow us to do it. So we go here and then these are called no-reverse gadgets.

So once you push this block back, you can never go-- this gadget becomes untraversible. So you come here, you push this, push that, push that, and now you can never use this gadget again. So it's like a single-use thing.

So you can come here. If you tried to do that, you would get stuck because this is a lock and it hasn't been unlocked yet. So instead you've got to go over here, go through this thing, permanently destroy it, and then unlock this gadget, and then you exit. So that was north to south traversal. We unlocked this gate and we unlocked this gate.

So if you're coming west to east, it could be this has been done or not. So going west to east, maybe you haven't visited this gadget. Then you can just go through here, block off a later north to south traversal, that's OK. Then come over here and leave. And if you try to go here, nothing happens. OK?

Or it could be this has already been done-- north to south-- and then when you're coming from the west, this has been unlocked, and so you can open this gate and come through here. This has already been used, so you can't go that way. So instead you use this one. This has already been unlocked, so you can get through. And then you get out.

So with just a little bit of checking, those are the only things you can do. So when we have this diagram-- I think I have one here-- variables and clauses, and you connect all the variables to all the clauses they're involved in-- this is a slide from lecture one, but it's the same outline. You know the order in which these crossings happen. Because you know I'm going to visit variable i before I visit variable j greater than i.

So I know how to order-- and each of these paths is really two paths, one and then the other. And so I know whether I'm going to do north to south before west to east. If not, I'd transpose the gadget and exchange north-south with west-east.

And then I don't need that they're both visited. I don't know which ones are going to be visited, but I could do one or the other. Or if I do both, I know one will come before the other. So that kind of crossover gadget is enough for Push-1 and Push-Push-1.

And it's exactly mimicking what we did with *Super Mario Brothers*, we just had a different variable choice and a different clause and a different crossover gadget. But other than that, it was exactly the same. The gadgets were different, but the proof structure was the same. And that's our beginning of 3SAT reductions. We will do many more next class.