[SQEAKING]

[RUSTLING]

[CLICKING]

MICHAEL SIPSER: Welcome, everyone, back to theory of computation. So we are now at lecture number 15. And this is an important lecture. Well, all of our lectures are important, but this is going to introduce one of the major topics that we're going to see going on in various forms through the rest of the semester, which is the notion of a complete problem. In this case, it's going to be an NP complete problem.

And I think some of you have probably heard of that concept already. Maybe you've seen it in some courses or other, but we're going to do that in a rather careful and formal way over the next couple of lectures. So we are following up on our previous discussions about complexity, time complexity. We defined the time and not deterministic time complexity classes, as you may remember, the classes P and NP, talked about the P versus NP problem, looked at some interesting algorithms for showing problems in P called dynamic programming.

And we started to move toward our discussion of NP completeness with the introduction of polynomial time reducibility, which is related to some of the earlier reducibility notions that we discussed in the computability section. Quick review-- what it means for one problem to be polynomial time reducible to another-- this follows our pattern of reducibility concepts, where if a problem is reducible to another problem, and that other problem is solvable, then the first problem is solvable. So if a problem is reducible to an easy problem, that problem becomes [INAUDIBLE].

And the kind of reduction that we're going to be looking at here are the mapping reductions, but now where the reductions are computable in polynomial time. And so we had this result we mentioned last time, that, if A is polynomial time reducible to B, and B is in P, then A is also in P. So that's going to be critically important for the whole of discussion that's coming.

Our intuition about P and NP, repeating that here-- the NP problems are the ones where you can verify membership easily, as opposed to being able to test membership easily. Those are the problems that are in P. The verification typically requires some kind of a certificate that establishes a proof that the input is a member of that language.

And the big question of the field, which remains an unsolved problem, is whether P equals NP. It's called the P versus NP question. And we don't know the answer-- so whether there are problems that are in NP solvable in non-deterministic polynomial time-- typically these are problems that involve searching-- whether they can be solved in polynomial time, typically without the searching.

And if P were equal to NP, then you could always eliminate searching. And if P were different from NP, then there were cases where you need to search. And we don't know the answer to that. So in the direction of exploring this question and its ramifications, we introduced this problem called SAT. These are the Boolean formulas, which have an assignment that makes them evaluate to true. So we call those satisfiable formulas.

And we mentioned, but have not yet proven-- and we will not prove until the next lecture on Tuesday-- that there is this theorem, a very remarkable theorem that says that, if you have-- take the satisfiability problem, and if it is solvable quickly, then all of the NP problems are solvable quickly. So if SAT is in P, then P equals NP.

So in a sense, SAT is kind of no the-- it's sort of the super NP problem, in the sense that all of the difficulty of any NP problem is embedded within SAT. So if SAT becomes easy, then all of the other NP problems become easy. And we'll eventually prove that, but right now we're setting up the terminology to allow us to do that. So anyway, we'll get there.

So the key ingredient for proving this Cook-Levin theorem is polynomial time reducibility. What we're going to show is that every problem in NP can be polynomial time reduced to SAT. So every NP problem can be converted into a SAT problem. And so working toward that-- because when you think about it, there are infinitely many primes in NP, and being able to show that all of them are reducible to SAT is, in a sense, kind of-- you have to prove a higher level theorem.

It's not just a single reduction. We're exhibiting a schema of reductions, which shows that all of these problems can be reduced to SAT. And developing our intuition toward that direction, we're going to look at some specific polynomial time reductions today. And we'll start out with a reduction between two problems we have not yet seen-- one of them called 3SAT and the other one called clique. So on this slide, we're going to introduce those two problems.

So remembering, again, about Boolean formulas, we're going to consider a special class of Boolean formulas, restricted form of Boolean form formulas called conjunctive normal form. And so this formula here in particular is in that conjunctive normal form. So just remember-- I was explaining some of this to someone else earlier this morning, showing some of these slides, and it was pointed out that not all of you may be familiar with these the Boolean operations of and and or.

So this is the or symbol here. This is the and symbol here. Hopefully you've seen just the concept of Booelan and and or. Or is, in a sense, a little bit like a union. And it's like an intersection. And the symbols here are a little bit similar to the union and the intersection-- union and intersection symbols themselves. The V shape is a little bit like a pointy union symbol an the upside down V shape is a little like a pointy intersection symbol. If you haven't seen those-- seen that connection before, maybe it's interesting to observe that.

But anyway, what makes this formula be in conjunctive normal form? Well, conjunctive normal formulas are organized in a certain way. They have these groups called clauses within the parentheses that are anded together, that are connected by these and operations. And within those groups, which are called clauses, the elements are ORed together. Those elements are going to be either variables or variables with negation, negated variables.

So just to repeat that, these-- well, just to state, those variables or negated variables are going to be called literals, and the ORs of a bunch of literals are going to be called clauses. This is just the standard terminology in the field. OK? So a literal is a variable or a negated variable and a clause is an or of literals-- just a definition. All right, so we have a bunch of these clauses here. Each of the clauses has an or of literals in it.

Now, a Chomsky-- Chomsky-- conjunctive normal form formula-- I guess they're both CNFs-- but conjunctive normal form formula is one that's written as an and of these clauses. So we take these clauses, which themselves are ORs-- we and them together, we get a formula that's conjunctive normal form. Conjunctive stands for and, I think. Conjunction is an and.

And then a 3CNF is a CNF where each clause has exactly three literals. So for example, this particular formula is a CNF, but it's not a 3CNF, because it has at least two clauses which violate the condition of three literals per clause. This one is OK, obviously-- this first clause. And a 3SAT problem-- so this is the first of the two problems we're going to be talking about-- the 3SAT problem is the satisfiability problem restricted to these 3CNF formulas.

So this is the collection of-- set of 3CNF formulas which are satisfiable. So you can think of it as kind of a special case of the SAT problem, where we only care about formulas of the special kind. This being a special case, you might imagine that this might be an easier problem to solve, but in general, it turns out not to be, as we will see. Solving this special case is just as hard as hard as solving the general case for satisfiability.

Now let's turn to the second of these two languages up in the headline, the clique problem. So for that we-- going to turn to graph theory. So we're going to consider graphs, points and lines-- connecting them. And we will say that a clique in a graph is a collection of nodes-- collection of the points that are all pairwise connected by lines. And a k-clique is one where you have k such nodes.

So here we have a three clique, four clique, and a five clique. And then the clique problem is to try to find cliques of a certain specified size embedded within a given graph. You're going to be given a graph, and it targets clique sized k, and you want to know, is there a subset of the nodes in the graph of size k that are all connected to one another? That's the clique problem.

So obviously, the clique problem, just as with the satisfiability problem, is a decidable problem. You can just try every possible subset of k nodes and see whether it constitutes a clique. But that's, in general, going to be an exponential algorithm. If you have-- k is a large value-- it might be half the size of the graph-- you might be looking for a very large clique.

Then you have to try many, many subsets in order to see whether any one of them is a clique. It's also going to be and-- I hope you get this intuition-- this is a problem that's in NP. The clique problem is an NP problem, because you can easily verify that a graph has a k-clique just by exhibiting the clique. OK.

Now, so the language here is you've given a graph and a k, and you want to know, does the graph contain a k-clique? And what we're going to show is that these two problems are connected, that the 3SAT problem on the clique problem are related, in that you can reduce in polynomial time 3SAT to clique.

Just standing back at it-- back for a minute, it seems kind of surprising. There's no real reason-- no obvious reason why there should be a connection between three 3SAT and clique. They look very different from one another. But we do give that-- we will give that reduction.

That implies that, if you can find a way of solving the clique problem quickly, that'll give you a way of solving the 3SAT problem quickly. And that's going to be the whole point of this. So we're going to show this over the next slide or two, this polynomial time reduction. I'm going to walk through it slowly, but this is one where it's really important to try to get a sense of how it's working, because this is the kind of thing that we're going to be doing a lot of, and you're going to be asked to do it also on the homework and possibly on the final exam.

I hate to use that as the motivating force here, but at least it might be one motivation for some of you, that you have to understand how to do this kind of a reduction. I think, at a high level, what's going on is we're showing how to recode a Boolean formula satisfiability problem into the problem of testing whether a graph has a clique. Oh, let's see if we have any questions here that are coming up in the chat.

OK, so this is an interesting question here. Can we always convert a Boolean formula into conjunctive normal form, first of all? Yes. The answer is you can always convert a Boolean formula into an equivalent one in CNF. But in general, that might make the formula exponentially larger.

So just the mere fact that you can convert formulas into conjunctive normal form doesn't mean that solving conjunctive normal form formulas for testing satisfiability of the CNF formulas is going to be as hard as testing the general case, because just the conversion might be exponential. There's something more-- little bit more complicated going on than that. Let me just see here.

If phi is a satisfiable with formula which is not in CNF, can be-- so a similar question-- so the questions that I'm getting are about converting formulas to CNF. So yeah, you can do it, but not in polynomial time, in general, because the resulting formula you get might be much larger-- if you're looking for an equivalent formula. If you're not looking for an equivalent formula, then-- of course, then, depending upon what you're looking for, you might be able to find something smaller.

This is, I guess, a good basic question. Why is clique in NP? Doesn't verifying that you have a clique require going through all the possible cliques? You have to understand what verifying means. Verifying means you can verify something if you're given a certificate. In the case of a clique-- the clique problem-- the certificate is the clique. So once you have the certificate, you can do the verification in polynomial time.

Finding the certificate, of course, might be difficult. So you only think of NP in the context of having that certificate. So in a case for compositeness, the certificate might be the factor. There are problems sometimes where what the certificate is is not necessarily obvious. But there can be a certificate for showing that the input is in the language. In the ones that we've done so far, maybe you can argue that the certificate is sort of an obvious thing, but it's not always an obvious thing.

OK. Why don't we move on then? So let's see, how do we do this polynomial time reduction from 3SAT to clique? OK, here we go. So I'm just going to give a reduction. That's what the definition means. I'm going to give a way of converting formulas to pairs, a G and a k, where the formula's going to be satisfiability if and only if the graph has a k-clique.

OK, so let's a little bit do it by example. And in order to do that-- so here's going to be a formula now. It's in 3CNF. That's what I need in order to be doing this reduction. I'm converting 3CNF formulas into clique problems. We to have a little bit understand what it means when we say-- we talk about the satisfiability of a formula like this, because it's going to be helpful in doing the reduction.

Obviously, satisfiability means that the-- you can find an assignment to the variables. So you're going to set each of these variables-- A, B, and C, and so on-- to true or false, and you want to make the whole formula evaluate the true. But what does that actually mean?

It means that, because of the structure of the formula, that making this formula true corresponds to making each clause true-- because the clauses are all anded together, so the only way for the formula to be true is to make each clause true. And to make a clause true, you have to make at least one of the literals true.

So it's another way of thinking about satisfying this formula. Satisfying these [INAUDIBLE] satisfying assignment makes at least one true literal in every clause. It's really important to think about it that way, because that's what's going to be the basis for doing this reduction and all of the reductions. It's what makes 3SAT easy to think about, in terms of its satisfiability.

If you had a general satisfiability problem and you had a satisfiability both formula, there's no obvious way of seeing what the satisfying assignment looks like, but here we understand what it looks like. It has that very special form, making one true literal in every clause-- at least one true literal in every clause. So now we're going to do the reduction. So I'm going to take from this formula-- I know, for some of you, you're going to be chafing. Why am I going slowly? But I want to make sure that we're all together and understanding what the rules of the game are and what we're trying to do.

We're trying to convert this formula into a graph and a number. So right now my job is, to do this reduction, is to exhibit that graph. So I'm going to do that and two steps. First, I'm going to tell you what the nodes of the graph are. Then I'll tell you what the edges of that graph [AUDIO OUT]

Finally, I'll tell you what the number k is. That's the way this polynomial time reduction is going to work. And we have to also observe at the very end that the reduction that I'm given-- giving you, this procedure for building this graph can be done in polynomial time, but that, I think-- you'll see, once I'm done, that that's pretty obvious.

OK, so first, as I promised, the nodes-- so the nodes of this graph are going to correspond to the literals of the formula. Every literal is going to become a node in the graph, and it's going to be labeled with the name of that literal. Every node is going to be labeled an a, a b, or c bar, and so on. So here it goes. So those are the nodes of the graph G, one for each literal in the formula, labeled as promised.

Now I have to tell you what the edges look like. I'm going to tell you what the edges are by first telling you what they are not. I'm going to first explain to you what the forbidden edges are, what edges I'm going to promise not to include.

And then the ones that I will include are going to be all the others. So what are going to be the forbidden edges? First of all, the forbidden edges are going to be of two types. One is the edges between nodes that come from literals in the same clause. So I would just call that no edges within a clause.

So for example, these three nodes will not be connected to one another. And I'm going to indicate that by writing red dashed lines, which means there's not an edge there. Those are forbidden from having an edge. So these three have no edge, and the same thing for every other triple of three nodes that come from clauses. Those are no edges there.

And there's one other category of edge that I'm forbidding, and that are edges that can-- that go between inconsistent labels, and the nodes with inconsistent labels. So for example, between a and a bar-- those are inconsistent. Nothing's wrong with a going a to d. Those are not inconsistent.

Those are just different labels. Or going from a to a-- that's OK. But from a to a bar-- that's not allowed. So that's going to be another forbidden edge-- or for example, a to a bar here, a bar to a, or for example, from this c bar to c-- forbidden. OK? So you imagine, you're going to write down all of those forbidden edges.

Those are all of the forbidden edges, and then, after taking those away, you're going to be putting in all the other edges possible. So for example-- let me just gray those out so they don't interfere with the picture-- from a to d, there's going to be an edge, because those are not forbidden. a to a-- not forbidden, because they-- they're in different clauses and they're not inconsistent. They're consistent with one another.

So here are a whole bunch of others. I'm not showing them all. It becomes very messy. But here are all of the other edges between nodes which are-- where they're not forbidden. OK? And that's the graph. That's G. G is all those nodes and those edges which are not forbidden. And I just have to tell you what k is.

k is going to be the number of clauses. That's going to be the size of the clique I'm looking for in this graph G that I just spelled out for you. And I'm going to claim that this graph here that I just described will have a k-clique. k is the number of clauses exactly when phi was satisfied. It's kind of cool. If phi is satisfiability, then there will be a k-clique here. And if phi was not satisfiability-- no k-clique. We're going to prove that as a claim on the next slide.

OK, so k is the number of clauses. All right? Any questions on that construction? So I've done with the construction. What's left is to argue why the construction works. So far, so good-- let's move on. OK. So here is that very same construction. I eliminated the red forbidden edges. These are the ones that were remaining, plus anything else that was not forbidden.

That was the same formula that I had from the previous slide. Now I want to claim that that formula is satisfiability exactly when G has a k-clique. Now, why in the world is that? So this is an if and only if. It's proved in both directions. And this is going to be the typical kind of thing that you're going to want to do when you're exhibiting a-- one of these reductions, which is you're going to have an opportunity to do that. And we'll do that too in our examples.

OK, so now what I want to show is that, if phi is satisfiable-- so let's prove the forward direction-- if phi is satisfiable, then G has a k-clique. OK, so first of all, if phi is satisfiable, that means it has a satisfying assignment. Now, here's a common confusion. I'm not sure whether it's helpful to sprinkle the confusions alone with the discussion, but in case you're worried, you might ask, well, how do I find that satisfying assignment? I thought that was exponentially hard to do.

This is a proof. This is not an algorithm. We're just trying to argue the correctness of this construction. So there's no longer any concern about how to find an assignment. I'm just saying, if there is an assignment-- if there is a satisfying assignment, then something will happen something. Then something will happen-- in particular, then we will show that G has a k-clique.

So let's say phi is satisfiable, so it has some assignment. Let's take that satisfying assignment. And remember that, in any satisfying assignment to a formula in CNF, that makes at least one literal true in every clause. So let's just pick one of those. There might be several clauses which have multiple true literals. In that case, just pick one of them arbitrarily.

I don't have this indicated on this diagram here, but imagine maybe, in the very first clause, a was true; in the second clause, b was true; in the third clause, e complement-- e bar was true, not e-- e bar was true, which means e itself was false, but e bar was true. And that's the way each of those clauses, in turn, got to be true in this particular satisfying assignment-- because you're going to pick one true literal every clause.

And now, from that choice of literals-- one per clause-- I'm going to look at the corresponding nodes in G, and I'm going to claim that those nodes taken together form a k-clique. OK, so first of all, do they have the right number of nodes? Well, sure, because I'm picking one node per clique. I already said k is the number of cliques, so I'm getting exactly k nodes. So I have at least the right number of nodes.

But how do I know they're all connected to one another, those nodes that I just picked? Well, they're all connected to each other because I'm going to say-- well, because there were no forbidden edges among them. And remember that we put in all possible edges except for the forbidden ones. So how do I know there are no forbidden edges?

Well, what were the rules for being forbidden? It means they-- two nodes in the same clique-- in the same clause. Well, I'm picking one node per clause, so I can never be having two nodes from the same clause. So I'm never going to run into trouble with the first condition of being forbidden.

So what's the second possibility for being forbidden, is that I'm picking two nodes which are inconsistent-- well, how do I know that I didn't end up with two nodes with inconsistent labels? That would be bad, because then they would have a forbidden edge, and what-- my result would not be a clique. How do I know I didn't end up picking-- in this group I pick a, and in this group I pick a bar?

Well, because they all came from the same assignment. They were all true liberals in clauses. It's not possible that a was true in this clause and a bar is true in that clause, because if a is true here, a bar's got to be false. It can't be the true literal in this clause. So I cannot have any inconsistent nodes appearing among the nodes of my clique.

And so they're not in the same clause. They're not inconsistent, so the edge has to be there. And that's going to be true for every pair of nodes in that clique-- in that group of nodes, and that's why it's a clique. So that proves one direction. Now we still have to prove the other direction, because we have to say, well, if G has a k-clique, how do we know that phi is satisfied? So that's the reverse. So let's just take any k-clique that's in G.

And how do I know I can get from that, a satisfying assignment to the formula? Good-- getting some good questions here in the chat. But let me just move on. So proving the reverse-- the reverse direction, assuming we have a k-clique-- take any such k-clique. Now, first of all, you observe that it's got to have one node in every clause. It can't have two nodes in the same clause or zero nodes in a clause.

First of all, it can't have two nodes in the same clause. Why? Well, because those nodes are never connected to one another. So they cannot be both in the same clique, because all the nodes in the inner clique are connected to one another. By the way we constructed G, nodes from the same clause are not connected, so they cannot appear in a clique together. So there's going to be at most one node from every clause appearing in this clique, but-- appearing in this clique.

But we also have-- we know we have k nodes, so that means, if there is at most one per clause, and we only have k clauses, that means every clause has got to have one. If some clause is missing a node, then there's got to be some other clause that has two. So we know that every-- that this clique that G has exactly one node in every clause.

So how do we get from that satisfying assignment? So what we're going to do is take the corresponding literals that correspond to that one node in that was in the clique, take those liberals and set those literals to be true, which means setting the variable to be true or setting-- if the literal was a bar, then setting a to be false, because you want a bar to be true. You want to set the literals to be true.

Well, now we're setting-- there's one node in each clause-- we're setting one literal-- the corresponding literal-- true, so that's going to set one true literal in every clause. So that means it's going to be satisfying. There's one thing that one has to really be careful here to double check-- to make sure that we're doing this consistently, that we're not being asked to set a true and also a bar true, because then that would not be possible.

But a and a bar are not connected, so it's-- we're never going to be trying to set both a and a bar true. If we're going to be trying to set a true, we're never going to try to set a bar true, because those are not-- a and a bar cannot be in the same clique. They're not connected to each other. OK? So that is the argument.

And lastly, I claim-- we're not going to look at this in detail-- that it's kind of obvious that this reduction can be done in polynomial time. Namely, if I give you that formula, you can write out that graph in-- pretty easily. There's no hard work to be done in writing out that graph or counting up the number of clauses. So that's the proof that I can reduce 3SAT to clique.

And it tells you also that, if I could solve clique quickly, then I can solve 3SAT quickly. And this is the whole point, because I can convert now clique problems to 3SAT problems. No. I'm sorry. I got it backwards-- convert 3SAT problems to clique problems. So if I can solve clique easily, then I can solve 3SAT easily.

I just showed a way of converting these formulas to graphs. So that says, if I can solve the graphs easily, then I can solve the formulas easily. OK, why don't we turn to some-- oh, question, check-in. But we can now also-- I'll launch a check-in, but we'll-- let's just see. I'm seeing many, many questions here, which are actually what my check-in is about. I'll turn it back to you guys.

OK, so where did we use the fact that we have three literals per clause? Does this thing just work even if we had any number of literals per clause? What do you think? We got a tight race here. Truth is losing out, unfortunately. All right. Oh, it's really close, but still-- OK.

One more-- OK, it's neck and neck. OK, almost done-- 10 seconds-- are we done here? OK, that's it-- ending polling. Yes, truth has won out. Thank God. Yeah, it works for any size clause. We didn't use the fact that it's a 3CNF. It could have been any number here.

Well, we've got a-- I guess it's a plurality here, though, not a majority. Where did we use 3 in any of these argument? I didn't mention it. Maybe you were imagining that it's going to be part of it, but 3 does not come into this discussion at all. If you think about how it's-- why it's working, even if we had-- one of these clauses had 10 literals in it, as long as k is the number of clauses, and we don't connect any variable-- any literals internal to a clause, this whole argument is going to still work.

So please check that make. Sure you understand what's going on, because I can see that a good chunk of you have not got this right. So I got a good question here. What if it's only just one big clause-- so it's like the whole formula's just one big OR. So what does happen in that case? Good question-- so in that case-- suppose there's one big clause with 100 literals. That's the whole formula.

It's just a big OR. So we know the formula's going to be satisfiable, by the way, obviously. So if you look at the corresponding G, it's going to have 100 nodes, because there's one for every literal. None of them are going to be connected to each other, because they're all in the same clause. So it looks like we're going to be in tough shape trying to find a clique there, because there's no edges at all. Everything's forbidden.

But what is k? k is going to be 1 in that case, because there's only one clause. And kind of a degenerate case, but a clique with just one-- just a single node is-- counts as a one clique, because it's just one node. There's no need for any edges at all. It still counts as a clique. So it'll still work out. Let's see. What else here? That was kind of a fun question that you asked me. Thank you.

There are a lot of questions. I think we're actually pretty near the break also. We'll just see. Yeah. Oh, many, many questions here-- so I think I'm going to start the clock going down for our, coffee break and I will take some of these questions. I'll try to answer some of these questions-- oops-- try to answer some of these questions afterward.

All right. Good-- all right. So I got a question about making sure why nodes don't have inconsistent labels, if I understand the question correctly. So I never put edges between nodes with inconsistent labels. That's the rule. That's my construction. I get to say how the construction works. So those two nodes with inconsistent labels can never be in the same clique, because there's no edge between them.

So I'm not sure that answered the question there, but that was-- let's see. So we're getting another question here. Since any Boolean formula can be converted to CNF, does that mean-- is SAT polynomial time reducible to clique as w-- [INAUDIBLE] time reducible to clique, but not for the reason that any Boolean formula can be converted to CNF, because that conversion is going to be an exponential conversion in general-- the conversion to CNF. So you have to be careful about what we mean by converting a formula to CNF.

So also getting question-- what happens in the case of 1SAT? Well, we didn't really talk about 1SAT. So the question is, suppose there's only a single literal per clause. That's an interesting-- another sort of edge case here. What happens under those circumstances? So there's only a single literal per clause.

So you have to work out what happens. But in that case, all the clauses just have one literal in them, and now they have-- when you have a 1CNF, so there's only one literally per clause, the only way that can be satisfiable is you have to make each literal true, because there's no ORing anymore. So every literal has to be true in the assignment.

So that means you can never have any consistent literals. And that's the only case when you would not have an edge, because you're not-- the forbidden condition won't happen because you don't have more than one node per clause. Every clause is going to have just a single node in it. So it really comes down to whether or not you have an inconsistently labeled-- inconsistent clauses. I didn't explain that super well, but it's-- the argument still works, though. You should check it for yourself.

So this is a basic question. How do we see that F is polynomial time? I'm not sure I want to spend a lot of discussion on that. The conversion from the formula to the graph is kind of a one-to-one-- every literal becomes a node. The rule for when edges are present-- it's a very simple rule. I'm not sure what more to say. It seems pretty clear that the conversion-- that reduction is polynomial time computable. If you wrote a program to do that, it would operate very quickly.

Is it possible for G to have a k plus n clique, where n is greater than 0? Does that matter? If you think about it, the biggest possible clique that this graph G could have is k. It could never have more than a bigger clique, because two nodes in the same clause are never connected. You only have k clauses. So the answer is no, you cannot have a-- bigger than a k-clique-- just not going to happen.

Does each clause need to have the same number of literals? I don't see why. So question is, why are we worrying about 3SAT, since it didn't seem to matter here? There are other examples where it does matter. I'm actually not sure if we'll see one or not, but there are cases where it does matter. Let me just see if there's any quick questions here.

Somebody's saying, does-- do we have to worry about there being a polynomial number of literals? So you have to think about what that question means. The size of the input, which includes all of the literals, is n. So there's going to be, at most, n literals, because that's the size of the input, at most, in literals appearing. So you don't have to worry about that being polynomial, because-- by the way, we're defining the size of the input is going to be, at most, n literals. I hope that's clear.

OK. OK, so this is a good question. When we talk about polynomial time, it's polynomial in the representation of the input, which is-- if you want to think about it in terms of bits, that's fine-- or whatever. It's not going to matter if you use some larger alphabet. But the number of symbols you need in your fixed size alphabet, the number of symbols you need to write down that input in whatever your reasonable encoding is is going to be n. And it's going to be polynomial in n, polynomial in that length of the input.

So another question is, does SAT reduce the 3SAT? Yes. That we will see. SAT actually does polynomial time reduce to 3SAT. Not by converting to an equivalent formula, but by some more-- somewhat more involved argument than that. OK, why don't we move on? Some of these are going to come out anyway in the lecture.

OK, so now let's talk about NP completeness, because we've kind of set things up. We're not going to prove that the basic theorem, the Cook-Levin theorem about NP completeness, but at least we'll be able to make the definition. OK, here is our definition of what it means for a problem to be NP complete.

So a language B is called the NP complete if it has two properties. Number one is that it has to be a member of NP, and number two-- every language in NP has the polynomial time reduce to that language-- to that NP complete language in order for it to be NP complete.

So simple picture-- has to be in NP and everything an NP reduces to it. And so that's kind of the magical property that we claim that SAT has. sat, for one thing, is obviously in NP. And as we-- the Cook-Levin theorem shows-- or will show-- everything in NP is reducible to SAT, so SAT's going to be our first example of an NP complete problem.

And we're going to get what we claimed also for SAT-- that, if SAT or any other NP complete problem turns out to be solvable in polynomial time, then every NP problem is solvable in polynomial time. And that's immediate, because everything is reducible in polynomial time to the NP complete problem. So if you can do it easily, you can do everything easily just by going through the reduction.

OK. So the Cook-Levin theorem, as I mentioned, is that SAT is NP complete. And we're going to actually prove it next lecture, but let's assume for the remainder of this lecture that we know it to be true. So I'll use the terminology of problems being NP complete, assuming that we know-- that we have SAT as NP complete. OK? So we're going to be using some of the things that we're proving next lecture just in the terminology that we're going to be talking today.

OK, so here's the picture. Here's the class NP. And everything in NP is polynomial time reducible to SAT. SAT itself is a member of NP, but I didn't want to show it that way because it makes the picture kind of hard to display. So just from the perspective of the reduction, everything in NP is polynomial time reducible to SAT. We'll show that next lecture.

Another thing that we'll show next lecture is that SAT, in turn, is polynomial time reducible to 3SAT. So 3SAT, as you remember, are just those problems that are in conjunct-- are in 3CNF. And then what we show today is that 3SAT is polynomial time reducible to clique. So now, taking the assumption that SAT is NP complete-- so everything is polynomial time reduce both the SAT, which is, in turn, polynomial time reducible to 3SAT, and in turn, reducible to clique.

These reductions, as we've seen before, composed. You can just apply one reduction function after the next. If each one individually is polynomial, the whole thing as a combination is going to be polynomial. So now we know that 3SAT is going to be also NP complete, because we can reduce anything in NP to SAT, and then to 3SAT, and then we get a reduction directly to 3SAT by composing those two reductions-- and then, furthermore, at the clique.

So now we're-- have several NP complete problems. And moving beyond that, we have the HAMPATH problem, which we are going to talk about next. And we'll show another reduction in addition to the one we just showed to clique, now one going from 3SAT to HAMPATH. OK. So in general, I think the takeaway message is that, to show some language is NP complete, you want to show that 3SAT is polynomial time reducible to it.

OK, some good questions coming in-- I'll try to answer those. So you're going to take 3SAT and reduce to C. That's the most typical case. There's going to be other examples too, we might start with another problem that you've already shown to be in NP complete, and reduce it to your language. So it doesn't have to start with 3SAT, though often, it does.

Why is this concept important? So I would say there are two reasons. And this is going to get a little bit at some of the chat questions. First of all, if you're faced with some new computational problem-- you've got some robotics problem that you want to solve in your thesis, and you need some algorithm about whether the robot arm can do such-- move in a certain such a way, and involves searching-- possibly searching through a space of different kinds of motions, and you want to know-- I'd like to find a polynomial algorithm to solve that problem.

I'm using this as an example because this actually did happen to one of the former students from this class, who was working in robotics, and they actually end up proving that the problem that they were trying to solve is NP complete. So that's useful information, because even though knowing a problem is NP complete doesn't guarantee that it's not in P-- because conceivably, P equals NP-- what it does tell you-- if you have a problem that's NP complete and it does turn out to be in P, then P equals NP.

So there would be tremendous surprising consequences of your problem, if it was known to be NP complete, ending up in P. So generally, people take a proof of NP completeness as powerful evidence that the problem is not in P. Even though it's not quite a proof, it's powerful evidence that it's not in P. And so you might as well give up working on trying to find a polynomial algorithm for it, because if you do, you don't have to worry about robotics anymore.

You're going to become famous as a So I wouldn't worry about-- so if you throw problem's NP complete, you can pretty much assume-- almost certainly not in P. Now, there's another reason related to-- for the theorists to be-- care about NP completeness, and that is, if you're trying to prove P is different from NP-- or P equals NP, as one of the chat questions is raising-- order to prove P different from NP, the most likely approach is you pick some [? pre ?] problem in NP and show it's not in P.

That's what it would mean for them to be different. You're going to pick some NP problem and show it's not a P problem. Well, one thing would be terrible is possibly, P is different from NP, and you pick the wrong problem. Suppose I'd spent all my time trying to show-- back 20 years ago, I'm working really hard trying to show composites is not in P, which is-- would have been perfectly reasonable to do, because composites is an NP, and it was not known to be in P 20 years ago.

And then I invested tons of effort to try to prove-- because I like number theory or God knows what-- to prove that composites is not in P. And then, turns out, composites was in P. It was the wrong problem to pick, even though P might be different from NP. But what NP complete is guarantees is that, if you work on a problem, which is NP complete, you can't pick the wrong problem, because if any problem is in NP, and not in P, an NP complete problem is going to be an example of that-- because if the NP complete problems in P, everything in NP is in P.

So if NP is different from P, you know the complete problems are not in P, so you might as well work on one of those. So those are two ways in which NP completeness has turned out to be-- is an important concept. OK, so here's a check-in. You guys are getting-- maybe you're getting-- starting to think the way I think. You're getting-- some-- at least one of you asked this question in the chat.

Which language are we've probably seen is most analogous to SAT-- ATM, ETM, or 0 to the k, 1 to the k? Obviously, this is maybe subjective. You may have your own interpretation of what that means. There's, in a sense, perhaps no right answer, but what do you think? OK, that's it. I don't know how in the world that you could see this problem as analogous to 0 to the k, 1 to the k, but OK. I'm sure you have your reason.

Yes, this is a lot like ATM. Why? well, because for one thing, we showed in a homework problem that all Turing recognizable languages are reducible to ATM-- mapping reducible to ATM. So that's a little bit like the notion of completeness that we have for satisfiability, because all NP problems are going to be reducible to the SAT.

And the other thing too is that, once we start-- we want to show other problems are undecidable, we reduced ATM to them. And that's also very similar. We're going to be reducing SAT or 3SAT-- so it's indirectly from SAT-- to other problems in order to show that there shouldn't NP complete, so that they're hard-- that they're hard, in a sense. And so in both those ways, ATM and SAT are kind of playing similar roles.

One key difference, however, between ATM and SAT is in that for ATM, we can prove that it's undecidable, but for SAT, we don't know how to prove it's outside of P. Those would be the analogous situations. And so the story for SAT, which is easily solved by a diagonalization argument for ATM-- there's reasons to believe-- that we will see later-- that the diagonalization is not going to work to prove SAT outside of P. And besides that, we don't really have any good methods.

So anyway, let's move on. Why is ETM less analogous? Because the ATM was the first problem we showed undecidable, and SAT is the first problem that we're going to be showing NP complete. I guess that would be my answer. OK, let's continue. So let's show now that HAMPATH is NP complete, assuming that we know SAT or 3SAT is NP complete. So we're going to give a reduction from 3SAT to HAMPATH.

That's what this is about. It's just like what we did for clique, but now for HAMPATH. And this is going to be very typical. In these reductions, typically what happens is that you're trying to simulate a formula-- Boolean formula for satisfiability-- from the satisfiability perspective. You're trying to simulate that formula with some sort of structures inside the target language, which would be HAMPATH. The lingo that people use is that you're going to build gadgets to simulate the structures in the formula-- namely, the variables, the literals, and the clauses.

OK, these are going to be substructures of the graph, in this case, that you're building. We'll see what that means. So let's take a formula here and let's, again, try to imagine how we would reduce that to the HAMPATH problem. So the reduction would produce a graph-- no, it would produce an HAMPATH instance-- so a G, s, and t. Want to know, is there a Hamiltonian path from s to t in the graph?

And this is going to be not the whole graph, but this is going to be a substructure in that graph. The next slide is going to have the global structure of the graph. But here, this is going to be a key element, and we're going to call that the variable gadget. OK, what does it look like? I don't know if you can see it on your-- clearly enough, but these edges-- so there are four outside nodes here. The edges connecting them are all kind of pointed downward. And then there were these horizontal nodes here, and there are edges connecting them both left to right and to left. There's a row of these horizontal nodes.

OK, so you get the picture of what this looks like? You have to look carefully to see the arrowheads. I maybe should have made those a little bigger. Whoops-- so now, if we're trying to get from this node to that node-- imagine now you're trying to build a Hamiltonian path, because this is going to be a part of that graph G that I'm constructing.

Now, remember, for them-- for there to be a Hamiltonian graph, that means you have to go through every node in the graph. So if I want to get from s to t, the only way I'm going to be able to go-- to hit these horizontal nodes here is by picking them up as I go from s to t. So the only possibility is-- if you think about it, is-- would be for the sequence to go like this, if you can-- if that comes through for you.

So the path would go from s to this node, and then through these hops of-- along these horizontal nodes, and then down to the bottom node here. So that's one way that you can get from here down to there and pick up all the other nodes along the way, which is-- they're not going to have any other possibility-- possible ways of getting to them. So I'm going to call that a zig-zag. OK?

But there's another way to get from s to the bottom node, which is going to be by doing sort of the dual-- going to the right, then going out to the left, and then down to the bottom. I'm going to call that a zig-zig-- and with a little diagram here just to summarize what it means.

And this is the classic thing for a variable gadget, because it's a structure that when you're trying to think about how the object that you're asking whether it exists or not-- the Hamiltonian path-- how it relates to that object, it's going to have two possibilities, which are going to correspond to the variable being set true or false in the formula.

So we're showing how to set the-- simulate the variable in this HAMPATH instance. So setting the variable is going to correspond to constructing the path. OK? Now, we also have to make sure not only that the variable gets set to true or false, but that it gets a set of true and false in a way that makes this a satisfying assignment-- namely, that we get one true literal in every clause. So I'm going to add another gadget here called a clause gadget, which is just a single node.

And visiting that node here is going to correspond to satisfying that clause, to having a true literal in that clause. I'm going to have [? enabled ?] a detour from these horizontal nodes out to this clause gadget. So here it is. Here's that detour. As I'm going from left to right, I can-- instead of doing a single jump here, I could branch out and visit that clause node, and then come back, and pick up my left to right path, as I was doing before.

I hope you're seeing the big picture, because this has to be a Hamiltonian path. It has to hit every node. That's one of the requirements. This is going to be one of the nodes in my graph. The path has got to hit that node. The only way it's going to be able to hit that node is by taking a detour off of this horizontal path here.

But notice-- and this is the key-- if I'm doing a zig-zag, then I can make the detour and visit that clause gadget-- that clause node. But if I'm doing a zag-zig, the way this detour is structured does not allow me to visit that node, because if I'm going from to right left here, by the time I get to this outgoing edge-- now I want to come back-- that note has already been taken.

It only works if I'm going zag-zig, if I'm going from right to left-- left to right. If I'm going from left to right, then I can do it, but if I'm going right to left, no. If you think of left-- the left to right direction as true, that's going to correspond to that variable appearing in the positive way in that clause, but not in the negative way. The negative way-- I would reverse in and out of the detour, flip that around so now I could only do the detour when I'm going right to left, instead of left to right.

I hope you get the picture. This is how the structure is working. Now what's only left for me to do is put it all together. But this slide contains the guts of what's happening. Is there any question that I can answer for you on this? Let's move on to the next slide, and then, as questions come up, you can be typing them in, and we can maybe answer it there. OK, so here is the big picture.

So imagine we have that formula that we're start-- we're reducing that formula. And let's say it has m variables and k clauses. I'm going to call them clause c1, c2, up to ck-- here, the m variables appearing either positively or negated in that formula. And this is the way the global structure of G is going to look like. Now, I'm getting a question here. What do those horizontal nodes-- what role do they play?

Those nodes are there to allow me to visit those clause nodes, the nodes which represent the clause gadgets, which I'm going to place over here. This is almost a whole of G. I'm just missing a few edges, but these are all the nodes of G. So remember, I'm trying to find out, is there a Hamiltonian path from s to t?

Now, if I didn't have to worry about these nodes, the answer would be just yes. In fact, there would be many Hamiltonian paths from s to t, because I can do a zig-zag or a zig-zag through each of these variable gadgets, and that would take me from s to t, and I'd be good. It's just the c nodes, these nodes-- these ci nodes-- I have to hit them too. So they're going to be-- visiting them is going to be enabled by detours from here.

So let me just try to show you what that looks like. So I'm going to magnify little pieces here from these gadgets here and show you how these guys are connected up. So here is the x1 gadget. So x1 appears positively in c1. Here's x1 and c1. And so that means, when I'm going left to right, I'm going to be a possibility of visiting c1. But now, let's look what happens with-- which was the next one I had here? OK.

Right. So the next one is-- oh, yeah. So x1 appears positively in c1. That's why I have the connection like this. Now, x1 appears negated in c2, so I only want to do-- enable the detour to visit c2 when I'm going to left, as opposed to left to right. So this set of horizontal nodes is only going to allow me to take a detour either out to c1 or out to c2, but not to both, because the Hamiltonian path is either going to go left or right or to left. It can't do both, when it's going through the x1 gadget.

You need to-- [INAUDIBLE] I'll try to help you through it, but you have to try to think about why it's working. Let's think together. So x2 also appears in c1, but now it appears negated.

So I'm going to have edges from this x2 gadget-- oops-- the x2 gadget, but now look at the-- look at how I've arranged that detour. I can leave on the left leftward side and return on the right side, which means I can only do that detour when I'm going to left. And that's because x2 is negated in c1.

Maybe it's a lot here, if you're not quite getting it, but the point is, let's say, try to quickly prove-- so that's the whole construction. You just do that for every single appearance of the literal in a clause. You're going to add these detours, which allow you possibly to go visit the clause. So the forward direction is-- why is this true? So you take any satisfying assignment, as I suggested, make the corresponding zig-zags or zag-zigs through the variable gadgets from s to t, and then take the detours to visit the clause nodes.

The reverse direction actually is slightly trickier. We're not going to have time to go through the subtlety of it, but what you want to make sure here is that you don't have a weird path occurring, because I'm going to start with a Hamiltonian path now and build an assignment. And we want to make sure that the path that I'm constructing doesn't go from one-- from this horizontal nodes to a closed node, and then back to somebody else's horizontal nodes, and is kind of a hodgepodge of things which don't make any sense in trying to reconstruct a satisfying assignment.

What you really want to have happen is the Hamiltonian path should have clear zig-zags and zags-zigs that allow you to decide how to set the variables. And so that's the role of these little nodes here. These are spacers that separate the detours from one another, and that force a visit to the clause node to come back to the same place from which it left. Otherwise, you would never be able to visit those spacer nodes.

You have to look in the book for that, but there is a little bit of a detail that you have to go through. You have to show it must be zig-zags and zag-zigs, and then you get the corresponding truth assignment, and it must satisfy phi for all paths. OK. Again, the reduction is polynomial time computable. I'm not going to say more about that. We're a little bit low on time.

Last check-in-- would this construction still work if G was undirected? Suppose I just eliminated all the directions from the edges, made them lines, instead of arrows. Would that now show that the undirected Hamiltonian path problem is NP complete?

Let me see. OK, what do the c nodes represent here? There's one c node for every clause. So there are k clauses named c1 to ck, and there are k c nodes. These are the so-called clause gadgets, which are going to force there to be one true literal in every clause for the satisfying assignment. You have to look at it, or maybe we can spend a little bit more time explaining it, but that's what the purpose is.

Does that mean we need only two inside nodes? So the horizontal nodes-- do we only need two of them? You won't be able to reuse these nodes for multiple detours. For one thing, once you've gone to a detour, you come back to the node next over, and so you better not overlay multiple detours. And also, you need to keep them separated from each other. Don't forget, this node x1 can appear in many, many different clauses, so you would need to have possibly many of these horizontal nodes.

So someone now says 2k inside nodes would suffice. Probably 2k-- I would say 3k, just to be safe for the spacer nodes. You need to look carefully at the argument, which is laid out in the textbook. You may not actually need the spacer nodes, but then it makes the argument just more ugly. So the way the construction is done is you have 3k inside nodes.

OK. Again, several questions like that-- the graph would start looking messy if x9 was in c1. Yeah, if x9 down here was in c1? Yeah, it would be messy. It's OK. Messy is allowed. All right, I think we're-- let's end the polling. Are you all in? All right-- share results. Yes. The answer's no. The construction depends on this being directed.

You can see that all over the place, but for one thing, the whole point of these detours is the directions of the edges. And so without that, this construction is going to be just a bunch of-- is not going to mean anything. It's not going to prove anything. It's probably always going to be-- have a Hamiltonian path without the directions.

So I think we're out of time. A quick review-- these are the topics we've covered. I think we're out of time, so I should let you go. But I'll stick around for a few minutes, in case any of you have any questions. But I need to run off at 4:00 myself for another meeting, so I don't have much that much time.

Clarify my comment about picking the wrong problem to tackle P versus NP, when I used composites as an example-- if I worked hard to prove that composites is not in P as a way of proving that there is some NP language which is not in P, that would have been a mistake, because composites is in P. I would have been working hard to prove something which we now know was false.

So we don't want to spend time working on the wrong language. But the nice thing about NP complete languages is that we have a guarantee that, if P's different from NP, that that language is not in P. Are there problems that are not in P that are in NP, but are not NP complete? Oh, that's a good question. Are there problems in between P and NP complete?

So NP complete is sort of like the hardest problems in NP, and the P problems are obviously the easy problems that are in NP. Is everything either NP complete or in P? So for one thing, there are problems that are not known to be in either category. So we'll discuss some of those in due course, but one of them is the graph isomorphism problem, testing two graphs-- if they're really just permutations of one another. It's clearly a problem in NP, but not known to be in P. So there are problems that are not known to be either NP complete or in P, so there are problems that might be in between.

But then there was another theorem out there, which says that, if you assume that P is differ from NP, then you can construct problems which are in between, which are neither NP complete nor in P. They're NP, problems but they're not NP complete, not in P. So those problems themselves are perhaps somewhat artificial, but they at least prove the point that it is possible to have these intermediate problems.

Oh, so somebody's asking, isn't factorization one. Not [? known-- ?] for the case of factorization-- or you have to make a language out of that, by the way. But it's a because factorization's a function, so we won't really want to be talking about languages. As the homework suggests, NP and P are classes of languages, but OK, that's a separate note there.

Factorization is not known. Factorization could be in P and it could be NP complete. Both of those are not ruled out. So I think most people would probably venture to guess that it's a problem that's in the in-between state, that's neither P nor NP complete, but not known.

Who first thought of this reduction from 3SAT to HAMPATH? It's so clever. Well, it wasn't me. I think that is due to Dick Karp, who was one of my professors at Berkeley, where I was a graduate student. That was done before I got there. It was around 1971 when-- so there were two famous papers. There's the Cook paper. There's also the Levin paper. That was in Russian. That took a while for people to discover out here in the West.

But the Cook paper was 1971, and very quickly followed after-- he just showed SAT is NP complete, but after that, Karp was-- he had a paper called "Reducibility Among Combinatorial Problems," and he had a list of about 20 problems that he showed were NP complete-- by reduction from SAT-- include clique, include HAMPATH, and a bunch of other things. And that was also a very famous paper. Both of those are-- people often talk about Cook-Karp as, together, they really show the importance of NP completeness and the whole notion of NP completeness.

Yeah, 21 problems, so-- yeah, so Karp proved 21 when problems were NP complete in 1972. So that was shortly after Cook showed that SAT was NP complete. Incidentally, I think the terminology NP complete wasn't around until a little later. And that might have been-- might be due to Knuth. I'm not sure.

I remember he did a big poll of people about what should be the right language to use for that term, and I think he came up with it. All right, I'm going to head off, guys. Nice seeing you all-- so until Thursday-- oh, until Tuesday. Bye bye.