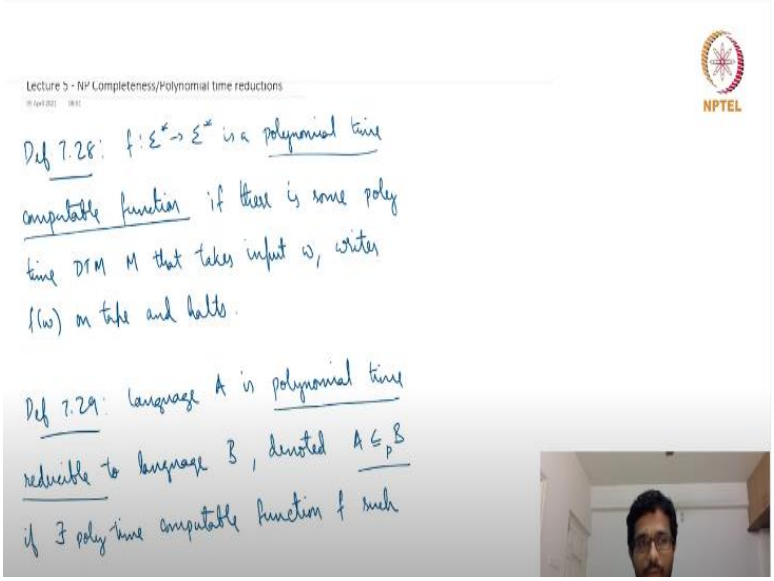
Computational Complexity
Prof. Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Lecture -05
Polynomial Time Reductions

(Refer Slide Time: 00:16)

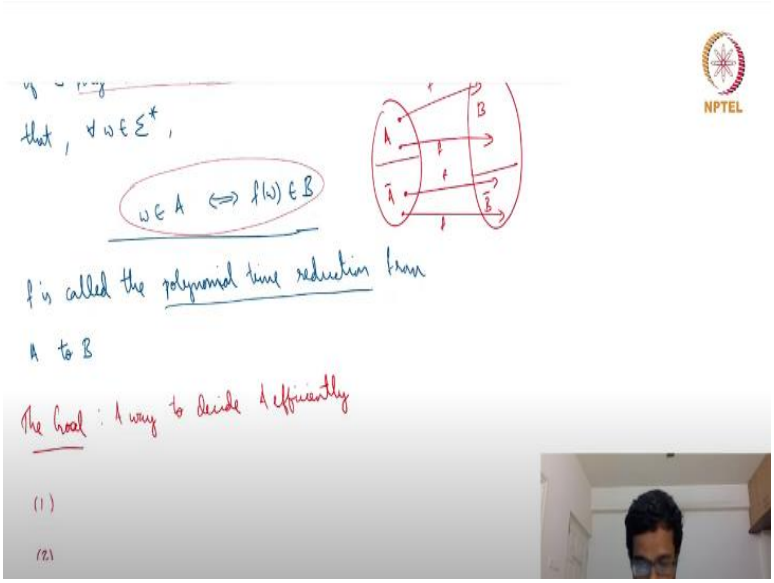


Hello and welcome to lecture five of computational complexity. So far we have seen the classes SP, NP and we have seen the language SAT. So today we are going to see the next important concept that we will see in this course is that of NP completeness. So on the way to NP completeness we first need to understand what is a polynomial time reduction. So what is a polynomial time reduction?

So first of all a polynomial time computable function is a function that can be computed by a deterministic polynomial time turing machine. So this is a formal definition that is written here if sigma from any alphabet to any other alphabet, or the same alphabet is a polynomial time computable function. If there is a turing machine that let us say given x computes f of x and writes f of x on the tape and stops. So here it is written W and then writes f of w on the tape and halts. So it is a turing machine that does not decide but rather computes a function.

So given w it will compute f of w and we say f of w f of f is a polynomial time computable function if there is a deterministic turing machine that runs in polynomial time that is capable of computing f.

(Refer Slide Time: 01:49)



And a language A is polynomial time reducible to language B it has this notation A less than or equal to B with subscript p. If there is a polynomial time computable function so this part is important polynomial time computable function f such that for all w in the alphabet, if w is an A f of w is in B. If w is in A, f of w is in B. So just to illustrate that there are two requirements here. First is that there should be a polynomial time computable function f such that f should be polynomial time computable.

That is requirement number 1. Two is that every w in A should be mapped to f of w and B if and only this means that everything that is not in A should also be mapped to not in B. So this is an if and only if it means that so just to illustrate suppose A maybe the sigma star, this half is A and this half is A compliment. And let us see this is B and this is B complement. So everything we are here in the top half should not be mapped to the top half.

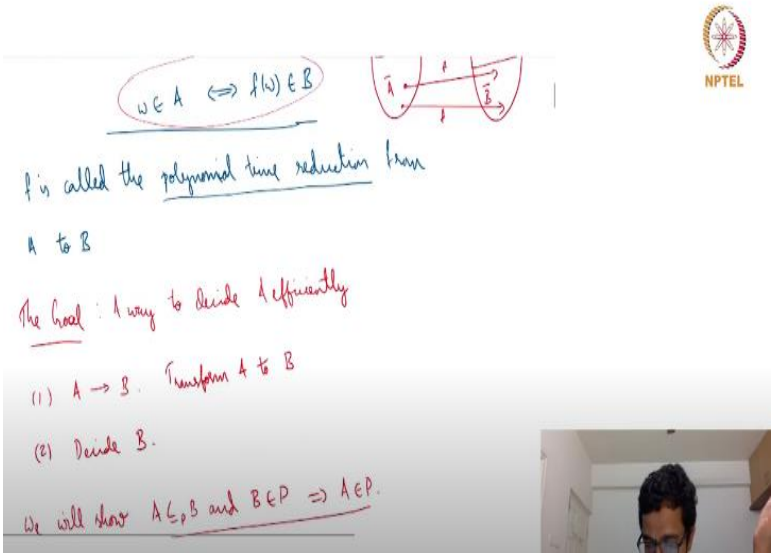
And everything in the bottom half should be mapped to the bottom half by f. So this is what these are the two conditions of the polynomial time reduction. So the point is that this if and only if condition means that if w is in A f of w should be in B which is here. And if f of w is in B that

should necessarily imply that w is in A. So which means which is the reverse direction which also means that something in A complement cannot be mapped to B.

Because if something in A compliment maps to B then that means the reverse implication does not hold good. So this is another way to think about it. Everything in A should go to B and everything in A complement should go to B complement. And so the two requirements again 1 is that f should be polynomial time computable and two is that everything in A should be mapped to B and everything not in A should be mapped to not in B which is written like this w in A if and only if f of w is in B.

So this line over here and this figure over here mean the same thing. And further if A is reducible to B in polynomial time, we say that the function that produces A to B is called the polynomial time reduction from A to B. So what is the point of this?

(Refer Slide Time: 04:57)



What is the point of doing a reduction or what is the point of doing a polynomial time reduction? The point is this is another way to decide A. So one way to decide A is to take A and transform it to B and then you decide B. So you first take, so you are given w and you have to decide whether w is in A or not. But you do not seem to know a way to decide whether w is in A or not. So you do not know whether w is in A or A compliment.

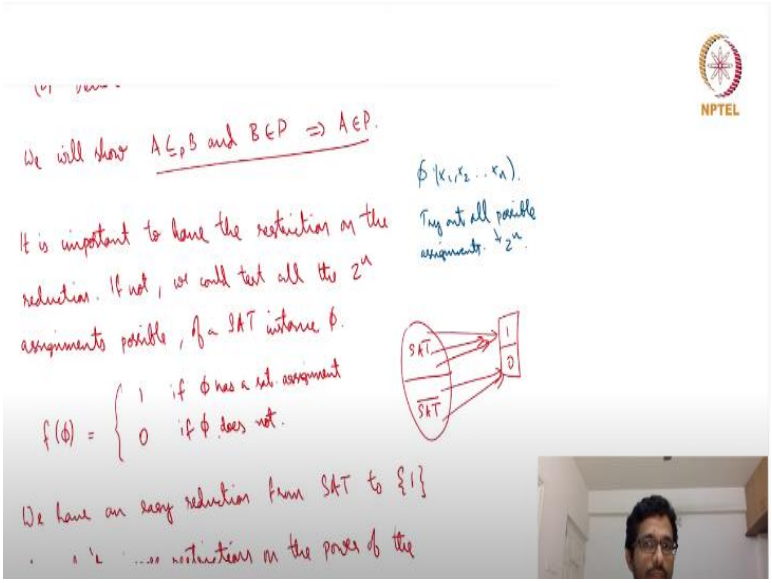
So what you can do is let us say you are given a w. Let us say for instance w is in A, but you do not know that and you transform it you compute f of w. So we know that f is polynomial time computable. So it will be taken to something in B. And now suppose we know a way to decide whether some given string is in B, some given y is in B. And now we can do that. You can say that the decision algorithm can be applied to the f of w that we computed.

So you can use the B's decision algorithm, decision algorithm for B. You run it on f of w and use that to decide A. So this is the goal or this is one of the goals for doing reduction. So from this what we infer is that if there is a way to reduce A to B, solving A or deciding A is not too much harder than deciding B. Because you can convert A to B and then decide B. So the difficulty level of deciding A is not that much higher compared to the difficulty level of deciding B.

And that also kind of explains this notation A less than or equal to B which means that the way one should think about it is that we are reducing A to B. So which means let us say B is not that hard, let us say B is rather easy. Then A is also not that hard because you can always transform A to B and then solve B. So this is the and the subscript p is to denote that the reduction uses a polynomial time reduction.

So we also see other reductions you may have seen in theory of computation you may have seen mapping reductions where we had you may have seen a subscript of m. In the future we will see log space reductions where with the subscript. So the principle is the same. But the constraint on the reduction machine will be different. So here it is polynomial time there it will be something else. And so we will soon see that if you can reduce A to B in polynomial time and B itself is decidable in polynomial time, then A is also decidable in polynomial time.

(Refer Slide Time: 08:10)



And it is important. So you may ask why do you restrict the reduction to be a polynomial time computable function, what is the point of this. Like say I compute, I can just transform A to B and then solve B. What is the problem is that the transformation is difficult to complete. The point is that then it may be possible to brushes like do the bulk of the work in transformation. And that is not very good because that does not give you an idea of the difficulty level of A.

So the goal here is to understand the difficulty level of different problems and compare it with each other. So if you are not really if you give the reduction operation too much power what may end up happening is that you may not get meaningful relationships between the problems. So that is why it is important to limit the power of the reductions. So for instance, we saw SAT. We have not said this already, but SAT is a problem that we know.

It is a problem in NP. We saw that in the previous lecture. However, we do not know a polynomial time algorithm for SAT. So which means we do not know of any efficient I mean the deterministic polynomial time. So there is no efficient deterministic solution for SAT as of now. However, it is an NP. So it is one of the problems that is in NP but not known to be NP and it is widely believed that it is not NP.

So as I said P versus NP this is one of the problems. People think it is P is not equal to NP and P is not equal to NP and SAT is one of the problems that is in NP but that is not known to be NP.

So this problem now I will show you a way to if you do not limit the power of the reduction we can solve this problem. So what you can do is let us say the reduction can use more time than polynomial time deterministic polynomial time. Let us say the reduction can use deterministic exponential time.

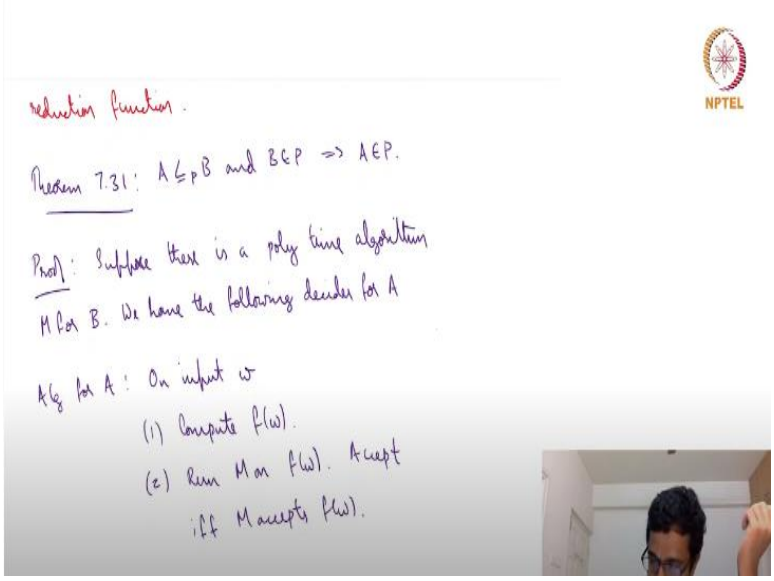
So one thing I can do is to give a Boolean formula, I can try out all the let us say there are n variables x 1, x 2, etcetera up to x n. I could give a Boolean formula phi, let us say it is in variables x 1, x 2 up to x n. I could try out all possible assignments. So there are 2 power n possible assignments for this. I could try them all out and check each one of them whether it leads to a true, evaluates phi to true. So there is something I can do.

And then basically I am deciding whether phi is satisfiable or not. And then I evaluate f to be 1 if phi has a satisfying assignment and 0 if phi does not. So now this process is not in polynomial time because I am trying out all the possible 2 power n assignments. But I have a reduction at the end of it I have a reduction. So there is a set of satisfiable Boolean formulas and there is a set of not satisfiable Boolean formulas.

So all the satisfiable Boolean formulas so now I have a set of two elements, 1 here and 0 here. Now all the satisfiable formulas are mapped to 1 and all the unsatisfiable ones are mapped to 0. So now this is something that is easily decidable. You have basically I am saying, I am giving you a string, you check whether it is 1 or 0. It is a trivial thing to check. So the map solution is something that is easy to check.

But we are reducing SAT to an easy language. However, the reduction process is time consuming. So now the SAT is considered to be a difficult problem. But we have reduced it to a seemingly simple problem or we have reduced it to a simple problem. But this reduction is not really meaningful because the reduction has too much power. So for us to get a better understanding or better comparison across problems it is important to limit the power of reductions.

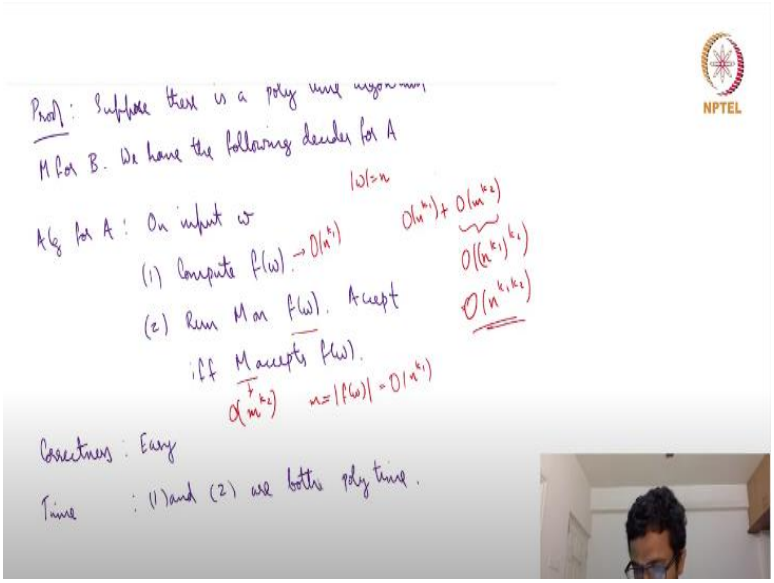
(Refer Slide Time: 13:11)



So that is a pointer imposing restrictions on the power of reduction. So now there is something that I already mentioned, but I will just quickly go through the details. If A is reducible to B in polynomial time and B is decidable in polynomial time, B has a deterministic polynomial time algorithm, then it follows that A also has a deterministic polynomial time algorithm. Why? So it is already what I outlined. Given w which we have to decide whether it is in A or not.

So you compute, so you know it is reducible to B and you know the reduction function. So given w you compute f of w and then decide run the decider for B. This is what it is.

(Refer Slide Time: 14:05)



So given w you compute f of w and run M on f of w where M is the decider for B. So you accept w if and only if M accepts f of w. So basically you test whether f of w is in B and use that to decide whether w is in A. If w is in A f of w is in B. If w is not in A f of w is not in B. This is something that we know because that is the property of the reduction. And so the correctness is just what I said right now.

And the other thing that is to be checked is so the claim here is that if A is in P. So now if this process if you combine this process, does it give a polynomial time algorithm for A? So let us see. So the claim is that both these operations number 1 and number 2 converting w to f of w, computing f of w and to decide whether f of w is in B, both of them run in polynomial time. Let us see. Why is that? This is true because let us say w is of length n and let us say this computing f of w takes polynomial time.

So let me say it is n power k 1 time algorithm. And the decision let us say M runs in let me write m power k 2 time where m is the input length. So I am just using a different input length small m here just to differentiate between this M and this m. So the running time of f is order n power k 1 and the running time of M is order n power k 2. So when you run what you do is you compute f of w. So this takes order n power k 1 time and then you run M on f of w.

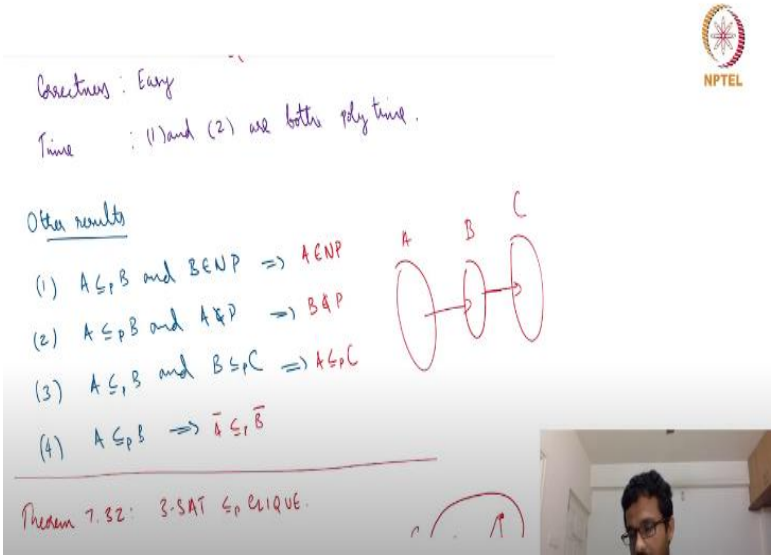
So this complexity of m power k 2 is on the length of the input. So the complexity is usually a function of the length of the input but the length of input is the length of f of w. So what is the length? So where m is the length of f of w, but what is the length of f of w? The reduction does not tell us what is the length of f of w. It only tells us how much time it takes. Well, if it takes 100 steps to write down f of w then f of w cannot be longer than 100 bits or 100 symbols.

Because writing each symbol takes one time step at least. So since we know that the input length is n and the running time of the reduction machine is n power k 1 the maximum length possible for f of w is order n power k 1. It cannot be longer than that because if it is bigger than that then you will not be able to write this much down in that much time in that many number of steps. So M is order n power k 1. So together how much time is it?

It is order n power k 1 plus order m power k 2 but M itself is order n power k 1. So which is order n power k 1 whole power k 2. So which is order n power k 1 k 2. So this dominates this. So eventually what remains is this. But anyways, k 1 is a constant, k 2 is a constant. So together we have a n power constant which is a polynomial time thing. So the combined algorithm of transformation and then running the decider for B is also a polynomial time algorithm.

When I say polynomial time algorithm I am always the baseline is the input length. So it is polynomial in the length of the input instance. So the input instance here is w. So the length of the w is n. So it is polynomial in n because it is n power some constant. Even though it is a bigger constant than either 1, so it is k 1 here and k 2 here. So it is k 1 k 2 which is potentially bigger. But it is a constant. So what we have seen is that if A reduces to B in polynomial time and B is in P then A is also in P.

(Refer Slide Time: 19:05)



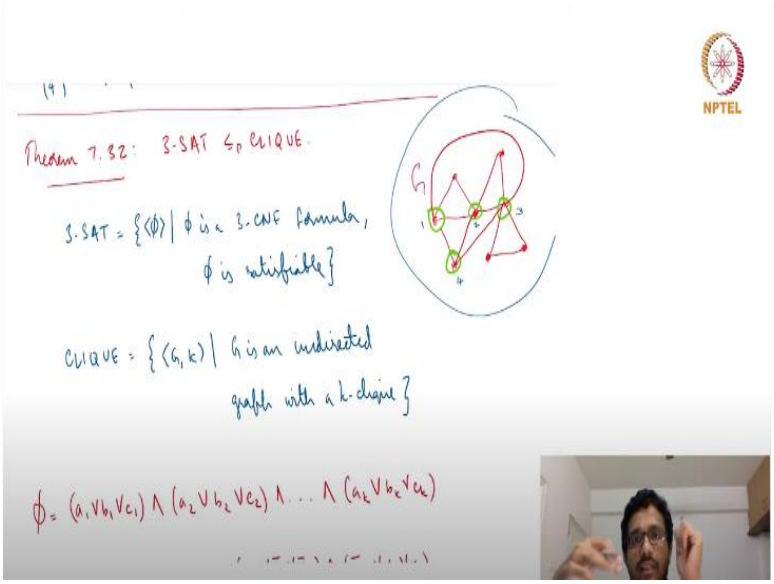
And some other results it is not so difficult to see. I will just state them you could think about how we get these things. So one is that if A is reducible so this result was that if A is reducible to B and B is in polynomial time then A is in polynomial time. So instead if B was in NP then we can infer that A is in NP by exactly the same kind of arguments. The next point is that if A is not in P and A is reducible to P then B is also not in P.

So this is here notice that we are using the reverse direction to argue. Why is this? Suppose B was in P then we could just use what I stated here. If A is reducible to B and B is in P we could use that to infer A is in P. But by assumption A is not in P so B also cannot be in P. If B was in P then this would imply that A was in P. The next thing is that if A is reducible to B and B is reducible to C then A is reducible to C in polynomial time.

This is because you could just compose the two transformations. Whatever function reduces A to B you could compose that with so basically you get an instance A, B and C. You reduce from A to B and then reduce from B to C you could just combine these functions to get a combined function that reduces A to C. And finally A reduces to B implies that A complement reduces to B complement. Because if you just look at this definition of reduction the same function is used.

So just because the roles of A complement and the roles of B and B complement interchange with each other. So now we will just say w is in A compliment if and only if f of w is in B complement but that is logically equivalent to saying w is in A if and only if f of w is in B. Hence if A reduces to B equivalently we could say A complement reduces to B complement. So we saw what is reduction and we saw some basic understanding of what is the reduction?

(Refer Slide Time: 21:46)



Now let me just show you one interesting reduction. So this is theorem 7.32 from Sipser. So again whatever I have been talking about in the time complexity part so far is there in the

introduction to theory of computation by Michael Sipser, chapter 7, the chapter on time complexity. So the statement here is that 3 SAT is reducible to CLIQUE. So what is 3 SAT? phi is a 3 CNF formula that is satisfiable. CLIQUE is given G and a number k, g is a graph.

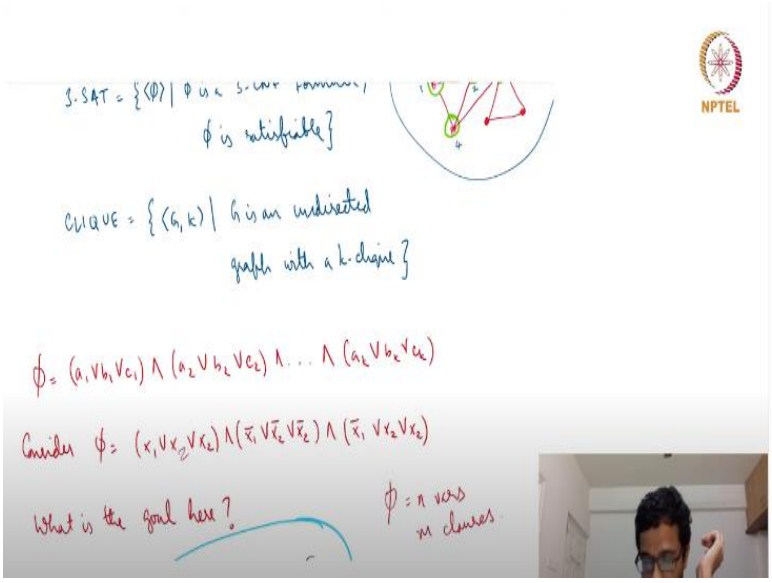
G has a graph with a k clique. So this is an example over here. So G, 4 would be an instance because there is a 4 clique here. The 4 circle vertices form a clique. This vertex, vertex 1 is connected to vertex 2, vertex 2 is connected to vertex 3, vertex 3 is connected to vertex 4, vertex 4 is connected to 1 and then 1 is connected to 3 and 2 is connected to 4. However there is no 5 clique here. There is no five vertices that are all adjacent to each other.

So clique is a set of vertices that are all adjacent to each other. Just as an aside, the fact that we are able to draw this graph without the edges crossing each other. That itself is an indication that there is no 5 clique. So there is a result not really related to complexity theory. In graph theory it states that if you have a 5 clique, there is something that you can just try out not in complexity theory. You try connecting all the vertices to each other.

So like this 1, 2, 3, 4, 1, 2, 3, 4, I do not know. So you will not be able to connect everything to everything else. That is the point. So I think this vertex 1 is connected to everything else, vertex 2 is connected to everything else, vertex 4 is connected to everything else. But the edge between 3 and 5 is missing. Anyways this is an aside. This area is something you can try out for fun. So anyway, this is a no instance if you give k equal to 5 and this graph.

So this may seem like two different problems. So one is some Boolean formula and checking whether it is satisfiable and two is given a graph is there a certain pattern between the vertices. Interestingly you can reduce one to another. So that is what we will see now.

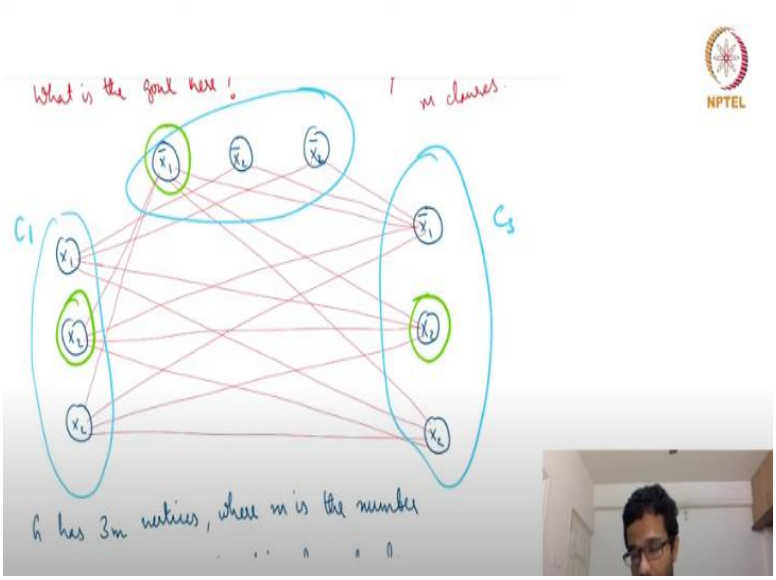
(Refer Slide Time: 24:47)



So 3 SAT means it is a 3 CNF formula which means it is an AND of clauses where each clause is an OR of three literals. So let us say the clauses are a 1 or b 1 or c 1 and a 2, b 2, c 2 and so on a k, b k, c k. So there are k clauses that each of a 1, b 1, c 1 etcetera can be literal. So each a 1 could be x 1 or x 1 complement, x 2 or x 2 complement. Same thing for b 1. It could be x 2 or x 2 complement, x 1 or x 1 complement and so on.

Same thing for a 1, b 1, c 1 and a 2, b 2, c 2 and everything. So for instance, one example is consider phi to be the following.

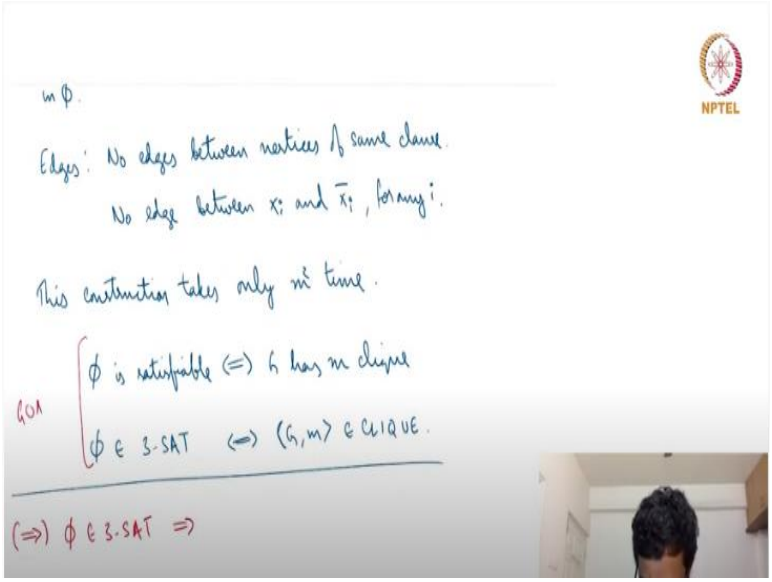
(Refer Slide Time: 25:29)



So we will be giving you a pictorial depiction of this example also. This example is also there in Sipser. So consider this 3 SAT formula there. So there are some repetitions because deliberately or intentionally we have kept the number of variables to two because if the number of variables blew up then the example does not look very small or it gets very messy with too many lines criss-crossing. So the formula is x 1 OR x 2 OR x 2 which is kind of trivial.

Because x 2 OR x 2 is the same as x 2 itself. And x 1 complement OR x 2 complement OR x 2 complement AND x 1 complement OR x 2 OR x 2. So there are two variables here and three clauses. And we need to show that 3 SAT reduces to CLIQUE. So this is an example 3 SAT instance a Boolean formula in the 3 CNF form. The goal is to draw a clique instance. So come up with a graph G and the number k such that maybe I will show here.

(Refer Slide Time: 26:54)



This is the goal. Given a formula there should be a function or a process to convert the formula into a graph and a number k such that the formula should be true or satisfiable if and only if the graph has a clique. If the formula has a satisfying assignment then the graph should have a k clique and if the formula does not have a satisfying assignment then the graph should not have a k clique. This is the goal. So what is the way in which you can convert a graph formula into a graph and then number k.

So now we will explain the construction. So first I will explain the construction then I will explain why the construction is correct. And the fact that the construction is in polynomial time. So remember for reduction we need two things. One is that the function should be polynomial time computable and two the function should satisfy this. w is in A if and only if f of w is in B. That is the correctness of the reduction. So both we have to show.

So in this case it will be easy to see the time taken by the reduction. The characters is a reduction we will explain after this. So what I have done here is I made 3 sets of 3 vertices each. So you could see that this set corresponds to the clause 1, x 1, x 2, x 2. The top set corresponds to the clause 2, x 1 bar, x 2 bar, x 2 bar and the right side set corresponds to the clause 3, x 1 bar, x 2, x 2.

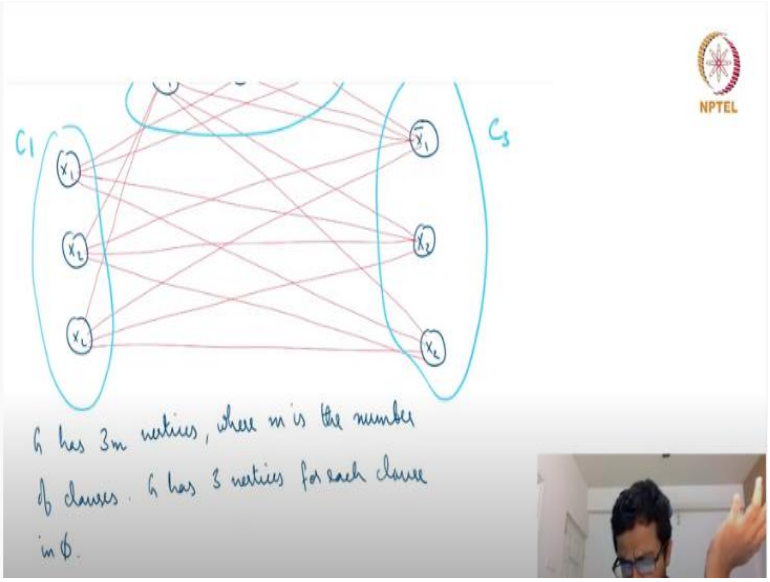
So basically one vertex for each one group for each clause and within one group 3 vertices one for each literal in the clause. This is the graph. So there are k clauses or n clauses there will be 3 m vertices will be there. And then now let us talk about edges. So I have already drawn for convenience the edges. So within a clause there are no edges. So you will see within c 1 there is none, within c 2 there is none and within c 3 there are none.

There is no edge connecting two vertices in the same clause. And across groups there are edges. But I do not put an edge between a literal and its complement later. So between x 1 and x 1 complement there is no edge. So you see between x 1 and x 2 complement there is between x 1 and x 2 there is between x 2 and x 1 complement there is everything there. But between x 1 and x 1 complement there is none.

Between c 1 from c 1 to C 2 also and from c 1 to c 3 also. And the same thing between x 2 and x 2 complement it is not there. But between x 2 and x 1 complement it is there and between x 2 and x 1 is also there. So you make 2 vertices adjacent if they are, one they are in two different groups and two they are not complements of each other. So no edges between vertices of the same clause. And two no edges between x i and its complement for any i.

These are the only rules and you connect everything else. So you can check in this figure and everything else is connected.

(Refer Slide Time: 30:46)

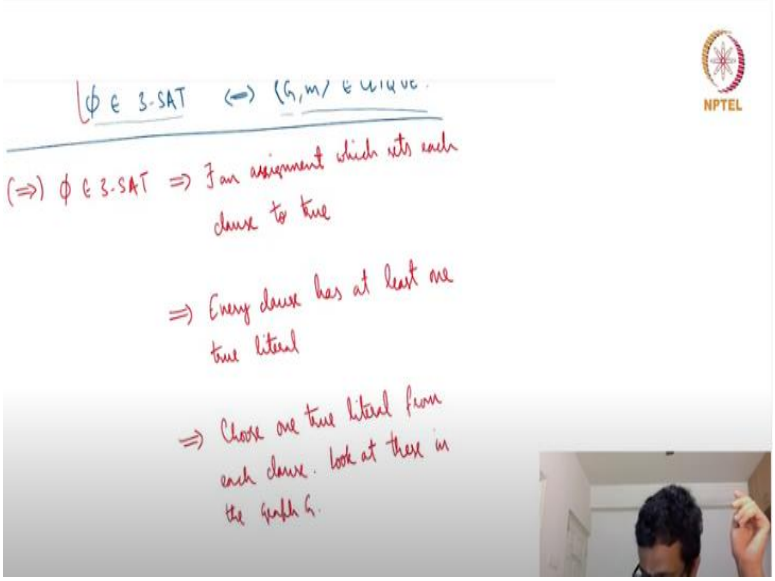


There are 3m vertices and between 3m vertices maximum you can have 3m square or 3m choose 2 edges. So the number of edges is some order of m squared edges. That is the maximum that you can have. So the construction is simple. So the construction takes m squared time I have written so it is actually order m squared. Because the rules are very straightforward you could just have some for loop to compute this graph.

So given a formula you can compute this graph. This is a reasonably straightforward procedure and it is clear that the procedure will be in polynomial time. It is not that difficult to see. If it is not immediately clear to you just think about it. So how you will write an algorithm given a formula that comes up in this graph. Now let us see why this construction is correct meaning the main part of the reduction or in this case the main part of the reduction that the Boolean formula is satisfiable if and only if the graph has a k clique.

So here by our choosing instead of k we will have m itself. So the graph of the Boolean formula will be satisfiable if and only if this has a m clique. Let us see why?

(Refer Slide Time: 32:09)



So we will have two directions because it is an if and only if. So first we will show that if the formula is satisfiable G has an m clique and then we will show the opposite, if the graph has an m clique then the formula is satisfiable. So what does it mean to say the formula is satisfiable? It means that there exists an assignment which sets each clause to true. There is a way to assign true false to the variables so that each clause is true.

Because it is an AND of the clauses and what does it mean to say each clause has to be true? Every clause has at least one true literal. So because each clause is an OR of literals so it has at least one true literal. It could have more but it has at least one. Now what you do is you choose one true literal. So some clauses may have more than one but you choose one true literal from each clause. So till now we are talking about only the formula.

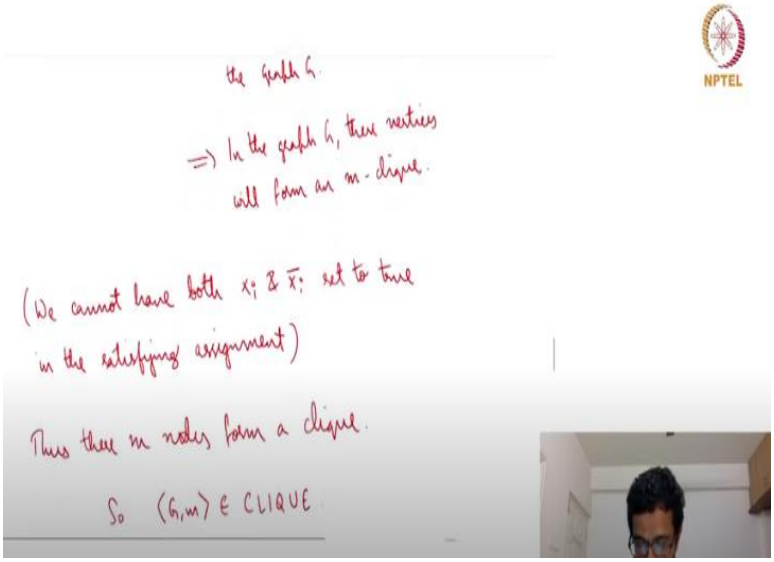
We are not at all talking about the graph. So now choose one true literal from each clause and look at this in the graph G. So assume there are m clauses and you can choose one true literal from this clause. So let us say is this satisfying? Is this Boolean formula satisfiable? I believe yes. If you set x 1 to false and x 2 to true everything is satisfied. x 2 is true, x 2 is true, here x 1 is false. So what is the true literal in each clause?

Here x 2 is true literal, so I will use maybe green colour. Here x 1 complement is a true literal because x 1 is false. And here x 2 is a true literal. So these three green vertices correspond to two

literals. So now these three literals are in three different clauses. And none of them are complements of each other. Why? Because if they are in complements of each other they cannot be both true. So if x 2 is true x 2 complement is false.

So you cannot pick both of them to be true. So these are the only conditions that we wanted to have edges between them. They should be in different groups and they should not contain any i such that x i and x i complement is there.

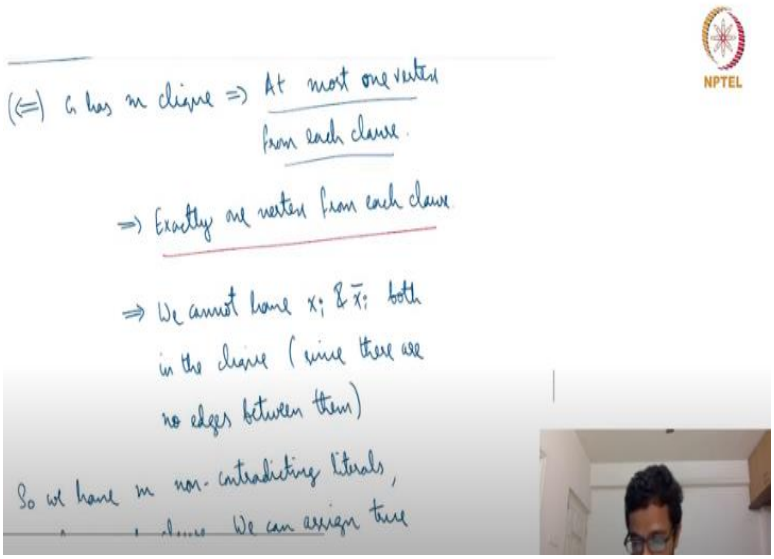
(Refer Slide Time: 35:24)



So in the graph G these vertices will form an m-clique, why? Because we started from the assignment and we picked one literal from each clause so that automatically places the literals in each of these clauses. One in each clause and second these are the true literals that were set to true. So if x i was set to true in some clause x i complement cannot be set to true in another clause. So x i and x i complement cannot appear together.

So what we will have is a bunch of true literals in separate clauses and with no conflicts. So all of them will be chosen. And all of them will have edges between each other. So because there are m clauses there will be exactly m vertices in the set and they will form an m clique. So these m notes form a clique. So the graph has an m clique. So if the formula has a satisfying assignment then it has an m clique. Now we will show the reverse direction.

(Refer Slide Time: 36:57)

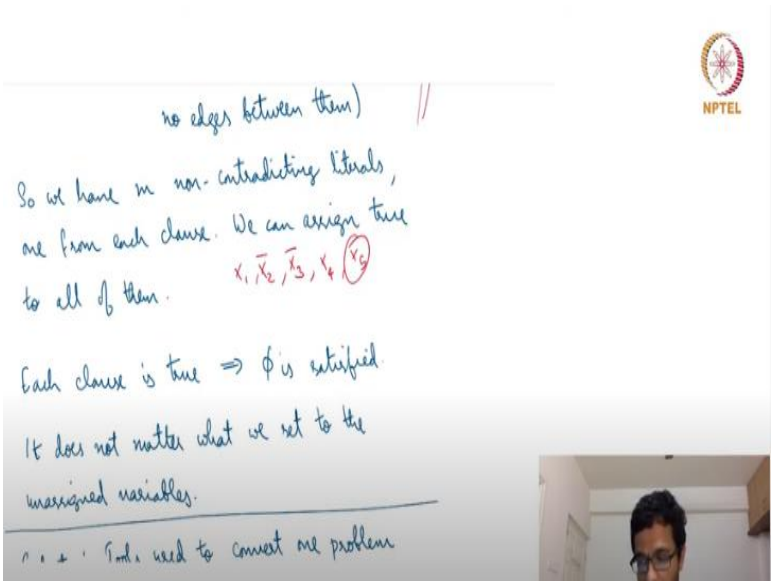


If the graph has an m clique then notice that this graph has no two vertices within the same group are adjacent to each other. So you cannot have two vertices in the m clique from the same group or the same clause. So we will have at most one vertex from each clause. So there are m cliques so m vertices and we can have at most one from each clause which means we should have exactly one from each clause. We cannot have more than one but we have to have m of them.

So there has to be exactly one for each. And by the rules we cannot have x i and x i complement both in the clique. Because if x i and x i complement they are both in the clique then these two will not be adjacent. So they cannot be part of the clique. So for instance in this figure, in a clique x 1 and x 1 complement cannot be both there. Because these two are not adjacent. So this clique that we picked here x 1 complement, x 2 and x 2, they form a clique because they are all in different groups and they are all not like x i and x i complement.

So we cannot have both x i and x i complement for any i in the clique. So we have m literals, one from each clause. So till now we are talking about the graph. The graph has an m clique which means exactly one vertex comes from each clause. When it is a clause I mean the clause part in the graph and we know that no two of x i and x i compliment or together can come.

(Refer Slide Time: 39:20)



So that means we have m non contradicting literals one from each clause. So now let us look at them in the Boolean formula phi. And we will make all of them true. So suppose x 1 is there, x 2 complement is there, x 3 complement is there and x 4 is there. So we will assign true to x 1, false to x 2, false to x 3 and true to x 4. And this will certainly create a satisfying assignment for the Boolean formula. Hence each clause will be set to true and the formula will be satisfied.

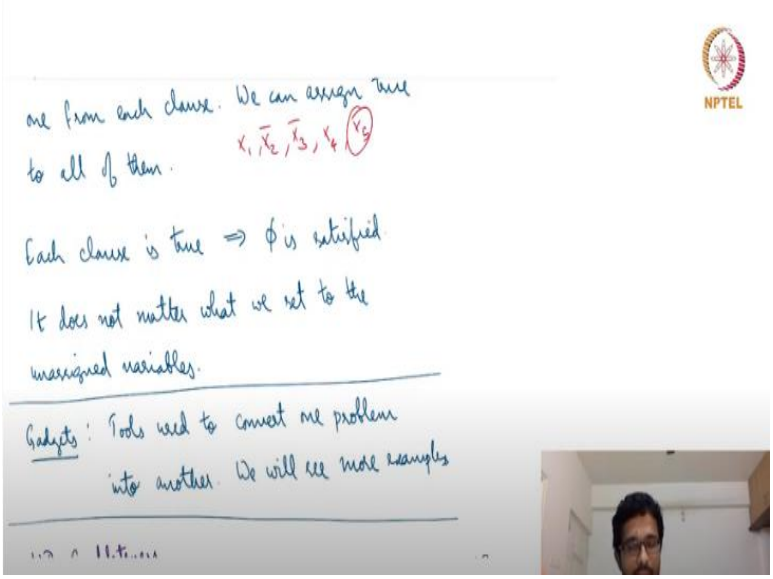
And perhaps maybe there is the Boolean formula also has a variable x 5 which is not set to true or false. But it does not matter what we set to that because from the set x 2, x 3 and x 4 every clause will be set to true. So this way now we have shown that if the graph has an m clique then the Boolean formula is a satisfying assignment. So we have shown both ways. The formula is satisfiable if and only if the graph has an m clique.

So this is an example of a reduction. So we converted satisfiability to the clique problem. So given an instance of satisfiability we constructed a graph such that the formula is satisfiable if and only if the graph has a certain size clique. And then we showed that is indeed the case. And we also showed that the transformation that is used is polynomial time. So this completes the proof that there is a reduction between satisfiability and clique.

Again this shows only one direction. We can reduce satisfiability to clique. This does not show that the other direction holds. It is not clear if you give a graph the clique this corresponds to a

Bollean formula. Not all the graphs with cliques have to be in this form. This is something for you to think about. Just one more small comment before I stop this lecture. I think I am overtime.

(Refer Slide Time: 42:01)



So we used a certain type of structure to convert satisfiability to clique, we used a certain type of graph. So sometimes these specific types of structures are called gadgets. So gadgets is a fancy name for these structures that are used to transform one problem into another. And we will see more examples of this. And with that I conclude this lecture 5. So what we have seen is what is a reduction, what is a polynomial time reduction, two things, one is the time required for the reduction and w is in A if and only if f of w is in B and some consequences of reductions and the fact that 3 SAT is reducible to clique in polynomial time. With that I end this lecture. Thank you.