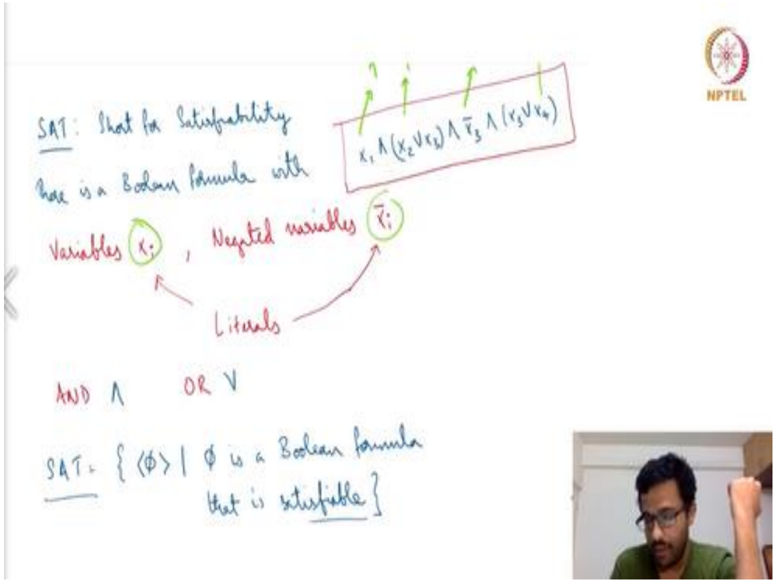
Computational Complexity

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

Lecture -04
The Class NP- Alternate Definition

Welcome to lecture 4 of the course. In the last lecture we saw a brief review of what is non-deterministic Turing machine and the clauses N time tn followed by some example problems and then we define NP, then we saw some example problems in NP followed by a brief description of the P versus NP problem which is a big open problem in computer science.

(Refer Slide time: 00:45)



SAT: Short for Satisfiability

Here is a Boolean formula with

$x_1 \wedge (x_2 \vee x_3) \wedge \bar{x}_3 \wedge (x_3 \vee x_4)$

Variables x_i , Negated variables $\bar{x}_i$

Literals

AND $\wedge$ OR $\vee$

SAT = $\{\langle \phi \rangle \mid \phi$ is a Boolean formula that is satisfiable$\}$

So, today we will see one more example or one more canonical example of a problem in NP followed by another characterization of NP. So, we realized or we learned that NP is a class of all languages that can be decided by non-deterministic polynomial time Turing machines. So, today we will see the characterization for that. So, first let me define what is called SAT, the problem called SAT. SAT is short for the satisfiability the word satisfiability or sometimes it is also called as a Boolean satisfiability problem.

So, you are given a Boolean formula, say maybe you are given something like this x 1 AND x 2 OR x 3. So, these kinds of things that kind of look like an inverted V are called wedges and they stand for AND the Boolean and the ones that look like V are called are stands for the Boolean OR. Now these are Boolean variables, x 1, x 2, x 3, x 4 and now you are asking, so what is the satisfiability problem?

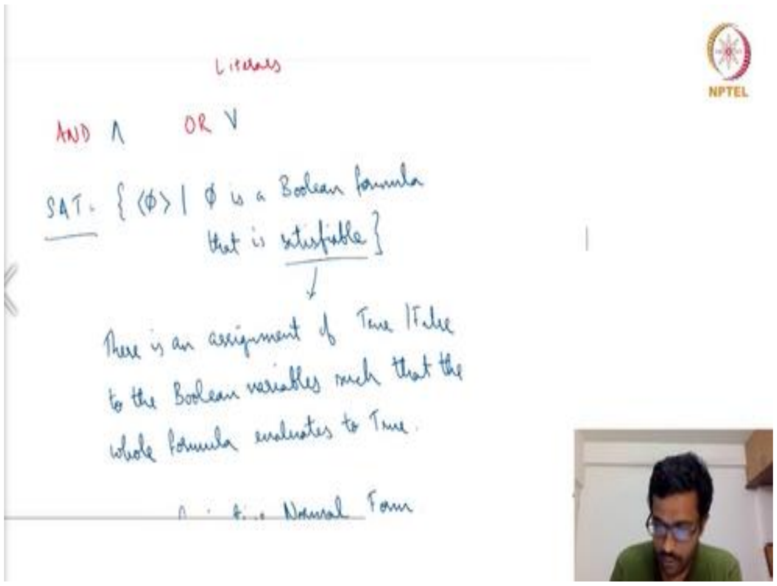
Given a formula like this now, is there any way to assign true or false values to x 1, x 2, x 3 and x 4 such that this formula evaluates to true. Let us see, so in this formula is there any such way. So, now, this is the AND of four things x 1 AND x 2 OR x 3 AND x 3 complement this bar at the top means complements AND x 3 OR x 4. So, these four things need to be true. So, this means that x 1 has to be true and this means that and it is AND of four things all these four things must be true.

So, this means that x 3 also must be x 3 must be false, because x 3 complement must be true. And now because x 1 must be true AND x 3 compliment must be true so, we got the inference that x 3 must be false, now since x 3 must be false now look at the last part, which says x 3 or x 4. Here x 3 has to be false, but x 4 has to be true for this part to be true so x 4 has to be true, and likewise x 2 also has to be true.

So, one way to satisfy this Boolean formula is for x 1 x 2 x 4 to be true, AND x 3 to be false and it looks like the only way for this Boolean formula to be satisfied. However, it does not matter whether it is the only way or if it is one among many ways, the point is that there is one way for it to be satisfied. So, again just to formally define it consists of Boolean formula a consists of Boolean variables like x i negated variables, which is a complement denoted by x i with a bar above it and sometimes both x i and x i complement are together called literals.

So, both are two different forms of x i and they connected by AND and OR, so AND is denoted by an inverted V and OR is denoted by a V and of course, they are also categorized or organized with parenthesis.

(Refer Slide Time: 04:47)



Literals

AND $\wedge$ OR $\vee$

SAT: $\{\langle\phi\rangle \mid \phi$ is a Boolean formula that is satisfiable$\}$

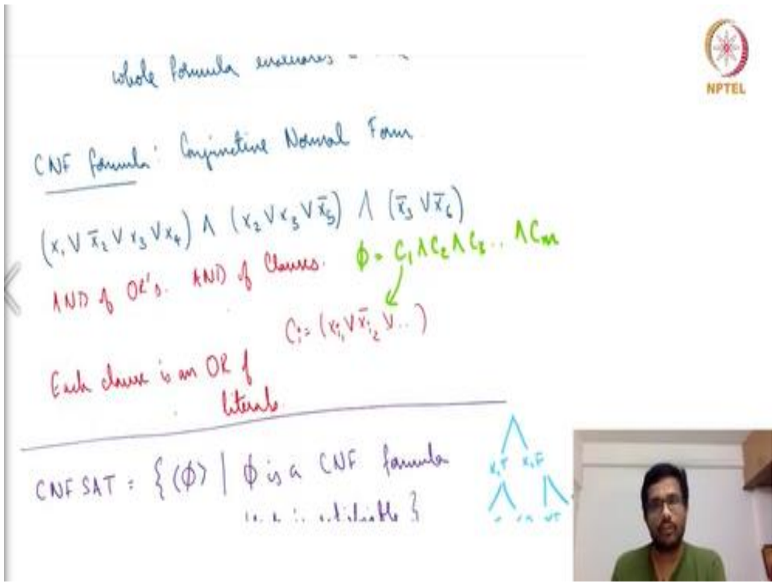
There is an assignment of True/False to the Boolean variables such that the whole formula evaluates to True.

[illegible] Normal Form

And formally SAT is a collection of or is a set of all Boolean formula such that they are satisfiable meaning, there is some way to assign the variable in there in such a way that the Boolean formula evaluates to true. So, this Boolean formula that will be written over here, this one is it is satisfiable this is in SAT. Because we found a way of assigning true false values such that this entire formula evaluates to true.

So, satisfiable means there is an assignment of true false to the Boolean variables that the whole formula evaluates to true.

(Refer Slide Time: 05:31)



whole formula evaluates [illegible]

CNF formula: Conjunctive Normal Form

$(x_1 \vee \bar{x}_2 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee \bar{x}_5) \wedge (\bar{x}_3 \vee \bar{x}_6)$

AND of OR's. AND of Clauses. $\phi = C_1 \wedge C_2 \wedge C_3 \ldots \wedge C_m$

$C_i = (x_{i_1} \vee \bar{x}_{i_2} \vee \ldots)$

Each clause is an OR of literals

CNF SAT: $\{\langle\phi\rangle \mid \phi$ is a CNF formula that is satisfiable$\}$

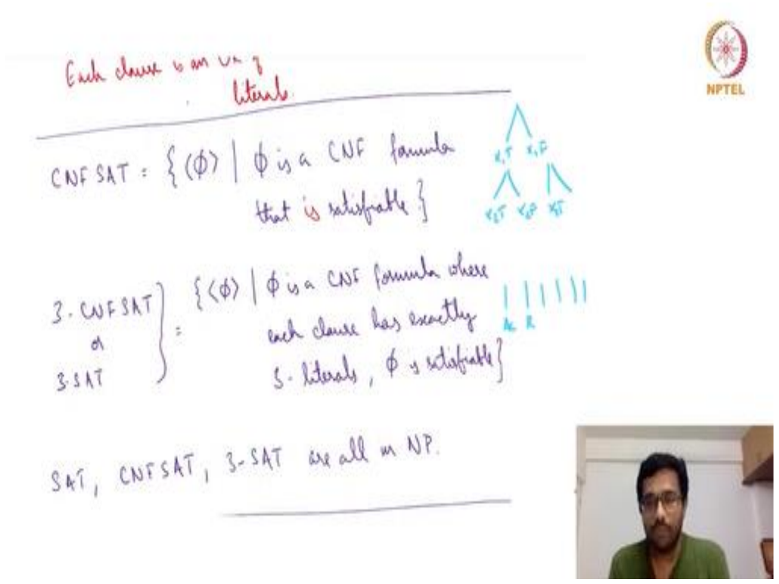
Now some minor examples of some special subclasses of the SAT. So, that is the class of all Boolean formula such that, that are satisfiable. Now, we will learn something called Conjunctive Normal Form. So, Conjunctive Normal Form means it is AND of OR's, so what you will have here is an AND of clauses the whole formula will be AND of clauses. So, something like this, let us say the whole formula phi, phi is just one way to call it a variable in algebra as x.

So, like the Boolean formula is sometimes called phi there are also other notations like psi. So, phi is C 1 AND C 2 AND so on till C m. That is an AND of m clauses. With each clauses of the form, C i is of the form something some x i 1 AND not AND, OR x i 2, sometimes some of them will be complemented by something like this. So, what we have here about in the blue is a conjunctive normal form, because we have three classes.

So, the full formula is an AND of three classes and each class is an OR of literals. So, this is why it is a Conjunctive Normal Form, so each clause is in OR of literals which is what is happening here. In fact, even the first formula that we talked about here, this formula that we said is inside even that is after CNF for a conjunctive normal form. Because it is the AND of four clauses, the first clause turns out to be have just one variable, so it is really an OR of variables.

So, in the second clause two variables are two literals and third clauses one literal and fourth clauses two literal. So, CNF SAT, so CNF form is this form, where each variable AND of clauses and each clause is an OR of literals.

(Refer Slide Time: 08:07)



Each clause is an OR of literals

CNF SAT = $\{\langle\phi\rangle \mid \phi$ is a CNF formula that is satisfiable$\}$

3-CNF SAT or 3-SAT $= \{\langle\phi\rangle \mid \phi$ is a CNF formula where each clause has exactly 3-literals, ϕ is satisfiable$\}$

SAT, CNF SAT, 3-SAT are all in NP.

So, the CNF SAT is a subclass of satisfiability consisting of all Boolean formulas in the CNF form which are satisfiable. Again, this big formula over here is certainly satisfiable. You can see there are too many variables x 1, x 2, x 3, x 4, x 5, x 6 and just three clauses to be satisfied. So, you can easily figure out ways to just x 1 is true and x 2 is true and x 3 is false, for instance satisfies this. CNF SAT is a clause of all Boolean formulas in CNF forms that is satisfiable.

You can also write CNF forms that are not satisfiable. So, in fact if you had here in the first formula over here. If you had an AND of let us, say, if you had another clause over here, and x 4 complement x 2 complement this. If you take all these five clauses together, you cannot be it cannot be satisfied simply because we already learned that with the four clauses to be satisfied x 2 AND x 4 have to be both true. Now, once x 2 AND x 4 have to be both true in the last clause x 4 AND x 2 have to be both false to make the last clause true.

So, this is an example of a Boolean formula that is not satisfied with a CNF Boolean formula that is not satisfiable. And another subclass of CNF SAT is what is called three CNF SAT, where it is obvious CNF form something like this. But in addition to it being in the CNF form, we are also insisting that each clause has exactly three literals. So, here the first clauses x 4 second clause x 3 and third clause x 2. But three CNF SAT or three SAT for shot, we want each clause should be in the CNF form and each clause should have exactly three literals.

And the formula has to be satisfied and interestingly perhaps not, so it is difficult for you to see at this point all the three languages SAT set up all satisfiable Boolean formulas three SAT or CNF SAT that have seen a formula that is satisfiable and three SAT which have three CNF formulas that are satisfiable are all in N P. How it is not so have to see. First you could have a non-deterministic Turing machine like this first decide x 1 to be true like what should x 1 be should actually be set to true or x 1 should be set to false.

Basically it is non-deterministically choosing whether each variable has to be true or false. And after x 1 true false status has to be chosen, then you decide whether x 2 has to be true or false and so on. So, when the tree let us say the n variables at after n levels, you would have 2 power n leaves at each leaf would correspond to a certain assignment like there could be n leaf where all the variables are true it could be n leaf where all the variables are false there could be n leave where x 1 is true x 2 is false x 3 is false x 4 is false.

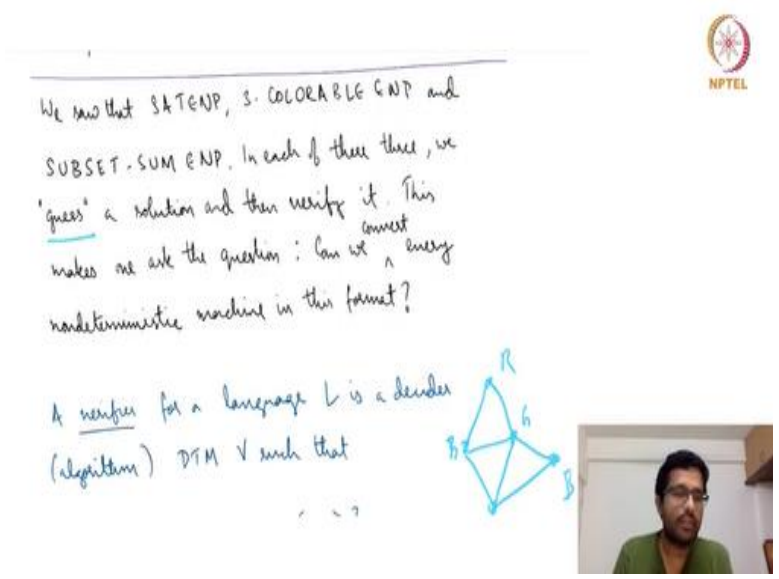
Whatever order or whatever way you want it all this all these assignments will be appearance in each leaf. So, this is the non-deterministic part where we non-deterministically choose the variables that the true false assignment for each of these variables and then all that you have to do is evaluate the formula at this assignment. So, given a certain assignment one has to evaluate the formula.

This could be done via a simple algorithm in polynomial time in deterministic polynomial time. So, once I tell you x 1 is true x 2 is false etcetera in this formula for instance. x 1 is true x 2 is false x 3 is for true something then it is very easy for one to evaluate whether for this assignment does this formula evaluated for this assignment is for my evaluate OR to. So, the non-deterministic part guesses all the possible assignments, and if any assignment leads to make this formula being true, then you will find it in the 2 powers n choices.

If all the if this formula is not satisfiable all whatever assignment you assign like in this case, this particular case, whatever assignment you assign none of these assignments are going to satisfy it because whatever you assign, there will be some clause or the other which is not true. So, in which case, this whole evaluation will all the parts will lead to a reject. Whereas if there is some satisfying assignment, then the path corresponding to that satisfying assignment will lead to an accept.

So, the formula will be accepted, because it has at least one accepting computation path. Hence, in the whole whatever I described works, whether it is a Boolean formula, it is a CNF specific Boolean formula like CNF formula, or a three CNF formula, so they are all in NP.

(Refer Slide Time: 13:58)



We saw that SAT $\in$ NP, 3-COLORABLE $\in$ NP and SUBSET-SUM $\in$ NP. In each of these three, we 'guess' a solution and then verify it. This makes one ask the question: Can we $\overset{\text{convert}}{\wedge}$ every nondeterministic machine in this format?

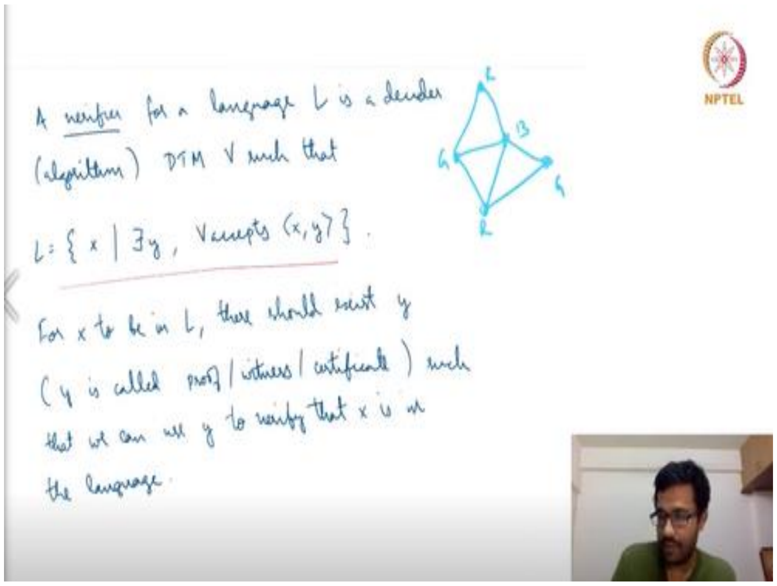
A verifier for a language L is a decider (algorithm) DTM V such that

So, one thing that you may want to observe here is that we saw three colourable, we saw satisfiability we saw subsets some. In each of these cases, we kind of non-deterministically choose a solution and then verify that this solution is a valid solution. So, there is an initial part which is non-deterministically choosing something or one might say guessing something and then followed by verifying whether this guess that we made is a valid solution.

So, now one might wonder, is it just coincidence that these three computational problems there in NP, because of a certain type of solution. First you guess and then you verify or is it that every problem in NP has a guess and verifies our solution. So, that is one question and the answer turns out to be yes, any problem in NP has a guess and verify solution. So, I explained this in the previous lecture as well.

So, we will see the proof of that. So, first let us formalize this notion of guessing and verifying. So, first let me define a verifier. Again, once again, I just want to say there is a guessing part which is non-deterministically guessing something or not determining choosing something followed by a verifying part, which is completely deterministic, there is not determinism going on.

(Refer Slide Time: 15:39)



A verifier for a language L is a decider (algorithm) DTM V such that

$L = \{ x \mid \exists y, V \text{ accepts } \langle x, y \rangle \}$.

For x to be in L, there should exist y (y is called proof / witness / certificate) such that we can use y to verify that x is in the language.

So, a verifier is the are trying to model this latter part the verification part, so a verifier is a decider for a deterministic decider or you can think of it as an algorithm a deterministic algorithm such that verifier for a language is a deterministic algorithm or a deterministic Turing machine such that for all the members of the language x. There is some way that the verifier can verify that can be used to verify that x is in the language.

In other words, if x is part of the language, so for instance maybe it is easier explained with an example. So, consider this graph now, is it a three colourable graph. So, the problem the language is the class of all three colourable graphs. So, maybe I will just erase this and now you are trying to determine whether this graph is three colourable. So, one way to do it is to guess a colouring assignment, so maybe you guess red, red, green, green blue.

And the verifier simply checks whether this coloring assignment is a proper colouring. Of course, the answer is no, because this edge has two reds on either side, this edge has two greens on either side. But if you consider another assignment, red, green, blue may be red over here this corresponds with added three colouring. And now is this graph three colourable? Yes, now if I give you the three colouring, the verifier can verify that this is an actual three colouring as a proper three colouring.

So, with the help of this assignment, a verifier can verify that this graph is three colourable. So, I want to decide or I want to verify whether this graph is three colourable. Sure, you can do that, because, if I give you the three colouring, you can verify it. So, whenever x to be in L there should exist y. The y is the information that we need to supply to the verifier and sometimes it is called proof, sometimes it is called witness and sometimes it is called certificate.

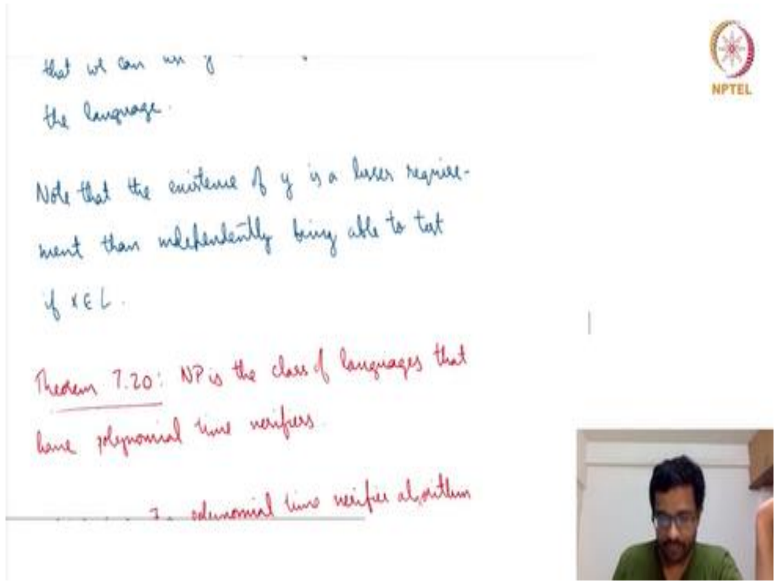
So, you can think of it. Now, I want to say that this graph is three-colourable. Now, how do I convince you that these are three colourable? So, now you are asking me, give me a certificate or give me proof that this is indeed three colourable. So, now I tell you look at this particular three colouring, it is a valid three colouring for this graph. Now, once you get the valid three colouring you can verify this edge is fine, this edge is fine this edge is fine and so on.

Then you are convinced. So, that is why it is called a proof for a witness or a certificate. It is a witness to the fact that it is three colourable. Such that we can use this y to verify that x is in language. In the case of a subset sum I would give you the subset that sums to the target. Now, suppose this graph was not three colourable. Now, whatever we were saying three colours it is not going to be a proper colouring.

So, there is no way this graph is this you cannot produce a certificate for this to be part of the three colouring like whatever certificate you give, it will not be verifiable because the verifier cannot clear it. So, all that we need is one certificate, sometimes there could be multiple certificates so now, if I change for instance, if I change here maybe I do not know maybe in this case, it is not there is not much scope for changing much, but we could do something like this.

Now, let us say this is blue, this is green and this is blue. So, this is different from the fact that there are multiple certificates this is different from the earlier colouring. So, it does not need multiple certificates, but there should be at least one certificate for so if x is in the language, there should be some certificate y such that we should be able to verify that x is in the language with the help of y and y is called proof or witness or a certificate.

(Refer Slide Time: 20:47)



that we can use y

the language.

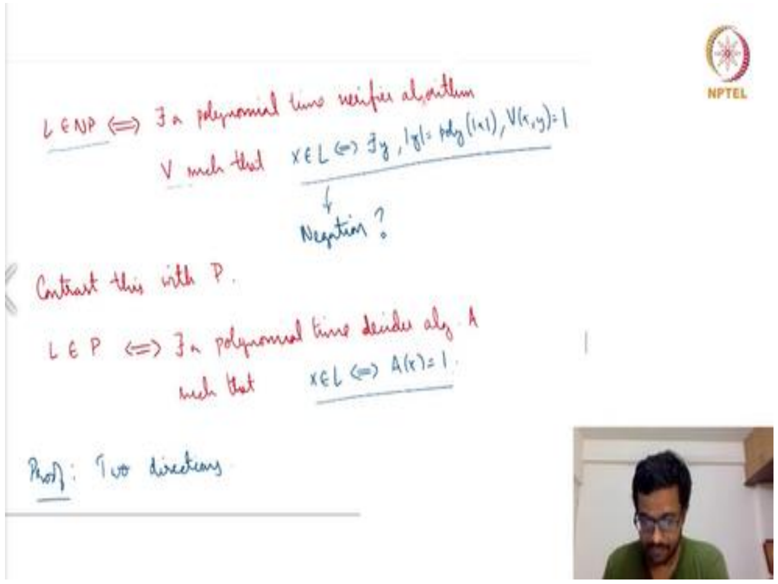
Note that the existence of y is a lesser requirement than independently being able to test if $x \in L$.

Theorem 7.20: NP is the class of languages that have polynomial time verifiers

$\exists$ a polynomial time verifier algorithm

And again, I said this in the previous lecture, verifying the certificate is something seemingly simpler than deciding whether the x is in the language or not. So, testing whether x is in the language is seemingly harder than verifying x is in the language with the help of the certificate. So, this is something that is where non-determinism helps us.

(Refer Slide Time: 21:20)



$L \in NP \iff \exists$ a polynomial time verifier algorithm V such that $x \in L \iff \exists y, |y| = poly(|x|), V(x,y) = 1$

$\downarrow$

Negation?

Contrast this with P.

$L \in P \iff \exists$ a polynomial time decider alg A such that $x \in L \iff A(x) = 1$.

Proof: Two directions

So, now let us use this we will go on to the next characterization that I was referring to, that is NP is the class of languages that have polynomial time verifiers. We already defined NP to be the class of languages that have non-deterministic polynomial time. That can be decided by non-deterministic polynomial time Turing machines. Now I am giving another definition and saying that this is also a valid definition for NP. NP is a class of languages that have polynomial time verifiers. So, I already defined what a verifier.

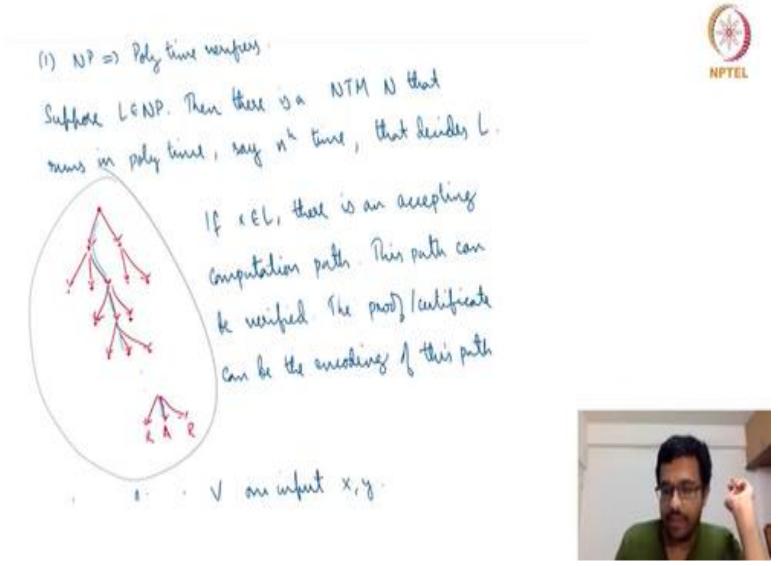
Now, let me formally write it, a language L is in NP, if there exists a polynomial time verifier algorithm be such that, x is an L if and only if the x is y and the length of y has should be polynomial in the length of x such that the verifier accepts x with the help of y. So, this is the formerly again this is the same definition that I made over here, but here it is more mathematical notation, there exists a polynomial time verifier algorithm v such that x is an L if and only if there is a verify there is a certificate y or a proof y which is of short length, which is of polynomial length.

So, why do we want it to be part of a polynomial length, because if it is, if the certificate itself is too long, then the verifier is going to take too much time verifying this. So, we want the verification process to be efficient. Such that with the help of y the verifier machine we can verify that x is in the language. So, if x is not in the language, there does not exist, such y which means for all the possible y is that you can think of, so think about it.

So, what would be the negation of this? For all the possible y if x is not in the language, for all possible y, the verifier should not accept x for all the ways. Whereas if x is in the language, the need to x is only one y, there could be more but they need to access only one y. And what is P? Again, we already defined P but just to contrast we will write it in this form. P is the class of all languages that have a polynomial time decided.

So, in other words, let us say the decider algorithm is A if x is an L, then A should accept x, which are x is equal to one. So, there is a simpler definition over here, but not N P is a more slightly more involved definition. If x is an L if and only if there is a verifier machine such that and this is a certificate such that the verifier can accept x with the help of the certificate.

(Refer Slide Time: 24:40)



(1) NP ⇒ Poly time verifiers

Suppose $L \in NP$. Then there is a NTM N that runs in poly time, say n^k time, that decides L.

If $x \in L$, there is an accepting computation path. This path can be verified. The proof/certificate can be the encoding of this path

V on input x, y

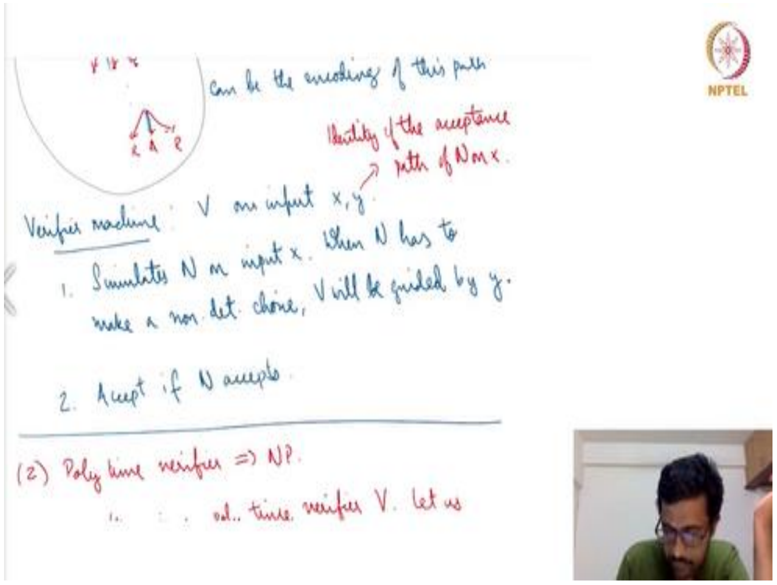
So now, we stated this definition that NP is a class of languages that have polynomial time verifiers. Now let us try to complete this proof. There are two directions here we have to show that NP is the definition that we already saw and NP is a class of all languages that can be decided by non-deterministic polynomial time Turing machines. This means that the language has polynomial time verifiers and the second prediction is that if it has polynomial time verifiers, then it is an NP.

So, suppose L is in NP, suppose the language is in NP, which means there is non-deterministic Turing machine that runs in polynomial time, say n power k time that decides L which means we can look at the computation tree of the non-deterministic Turing machine running on x. So, which means this is the starting configuration at the top and then there is some tree. Suppose x is in the language then we must be able to find we must be able to trace out in a path that leads to an accepting state.

So, there could be multiple accepting states, there could be multiple rejects, but we should be able to find at least one path. Now, one way to make this into a verifier guessing verifying model is to give the identity of this path. So, what is this path? So, I first go left then I go right most of then I go middle so, maybe I will say you I can label the children of these configurations and they will say first of all the leftmost than go right most than go middle then go middle again and so on till you reach the accepting path.

So, in order to receive a Turing machine that has the capability to identify an accepting computation path if it exists were a deterministic verifier cannot do this. But, if I tell you like you first go left then go right then go middle then go right whatever then, it is only the matter of verifying just one path from the root to the leaf and this is easier to do, we do not need to traverse the entire graph, we just need to traverse one small path the blue path that I have drawn in this figure. Hence, it is easier to verify this.

(Refer Slide Time: 27:29)



Can be the encoding of this path

Identity of the acceptance path of N on x.

Verifier machine: V on input x, y.

1. Simulates N on input x. When N has to make a non det. choice, V will be guided by y.

2. Accept if N accepts.

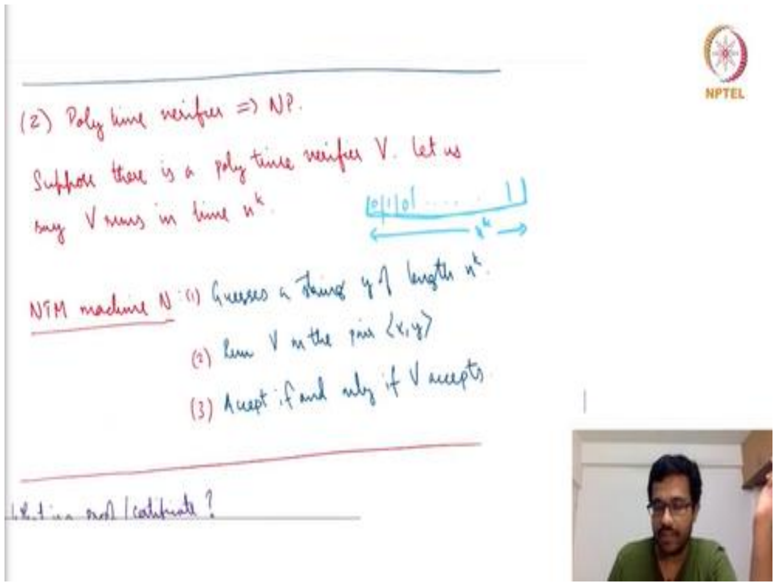
(2) Poly time verifier ⇒ NP.

[illegible] time verifier V. Let us

So, what does the verify machine do on input to y input x and y, so what is y here? So, y is the value of another colour y is the identity of the acceptance path of N on x. So, this is the certificate. So, if x is in the language there is some accepting path and that accepting path is a certificate y or the name of the accepting path or the identity of the same. So, what does a verifier machine do? It simulates input x and when N has to make a non-deterministic choice V will be guided by y.

So, basically it runs V it runs as if N is running but N makes non-deterministic choices and what will V do? V will follow the path as guided by the string y and the y will tell you which is the accepting path. So, if there is an accepting path, y can encode that and V can verify that. If x is not in the language all the paths will go to reject. So, whatever y you give it we cannot be convinced to accept x because the y's are all bad, all the y's lead to rejected and accept if N accept.

(Refer Slide Time: 29:45)



(2) Poly time verifier ⇒ NP.

Suppose there is a poly time verifier V. Let us say V runs in time n^k.

NTM machine N : (1) Guesses a string y of length n^k.

(2) Run V on the pair $\langle x, y \rangle$

(3) Accept if and only if V accepts

So, now, this is one direction, the other direction is that if a language is a polynomial time verifier, then it is an N P. Suppose there is a language that is a polynomial time verifier, suppose in this language that is a polynomial time verifier let us say V is a verifier now, assume that V runs in time n power K. So, this means that we need to come up with a non-deterministic Turing machine for the language.

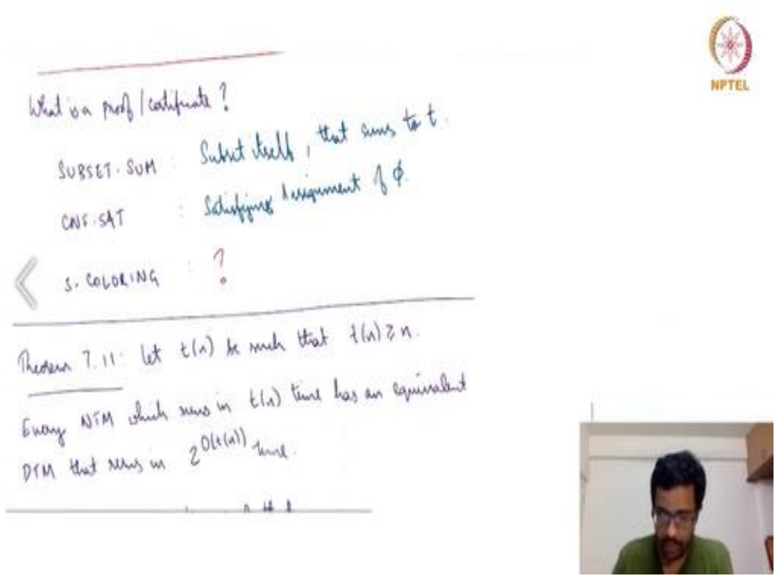
So, what V do? If a certificate is supplied V will verify x with the help of the certificate y. Now, what does the non-deterministic Turing machine do? So, not the missing Turing machine what it does is it guesses is string y of length n power k, so as I will draw here. Basically, the reason is if the verifier runs in time and power k then the certificate cannot be longer than that. So, let the non-deterministic Turing machine get a certificate of this length and then run.

So now, it gives us a string of length y, so this is this process we have seen before it can guess the first bit then the second bit and so on. Then run V so, now it has a certificate now, you run V on the on the pair input in the certificate accept if and only if V accepts. So, now the non-determining guess could be all the possible combinations of y's if x is in the language that x is a y that will lead to its verification and hence, that will correspond to a path in the non-deterministic Turing machine that will lead to an accept.

If x is not in the language there is no such y and whatever part then automatic Turing machine takes it will lead to rejection. Hence, we saw this, so if there is a non-deterministic Turing machine polynomial time Turing machine that accepts the language that decides the language, then the language is a polynomial time verifier and if the language is a polynomial time verifier, then it can also be decided by a non-deterministic polynomial time Turing machine.

So, this is an alternate characterization or an alternate definition of the class NP, NP is a class of all languages that have polynomial time verifiers.

(Refer Slide Time 32:48)



What is a proof / certificate ?

SUBSET-SUM : Subset itself, that sums to t.

CNF-SAT : Satisfying assignment of ϕ.

3-COLORING : ?

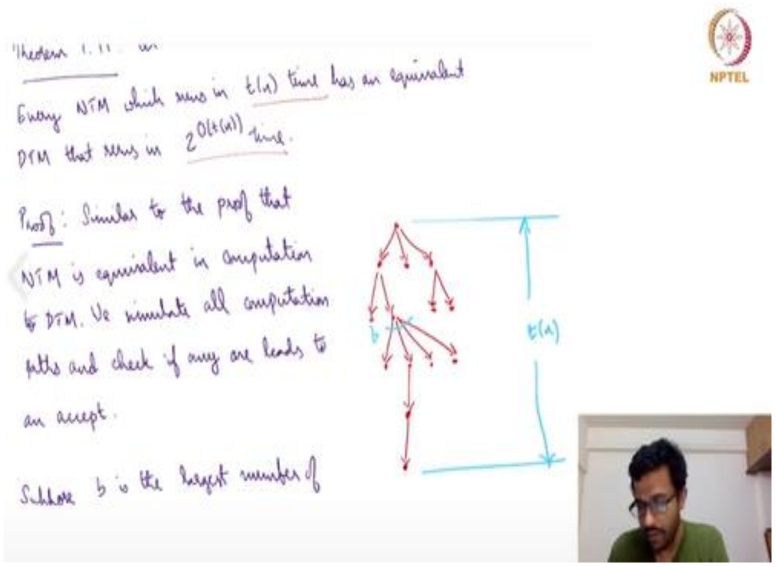
Theorem 7.11: Let $t(n)$ be such that $t(n) \geq n$.

Every NTM which runs in $t(n)$ time has an equivalent DTM that runs in $2^{O(t(n))}$ time.

So, just to summarize, what are the proofs and certificates that we have seen so far? In the case of subset sum, it was the certificate that was the subset itself that sums to the target some t. In the case of CNF SAT, it was a satisfying assignment using the Boolean formula. And the case of three curving I am not going to say if you can you can think over it what was the proof or certificate that the given graph is three-colorable.

So, just to summarize, we have seen a SAT and we have seen a CNF SAT, then we saw the alternate characterization of NP which is that it is a class of all languages that have polynomial time verifiers and then we saw why this definition is equivalent to the previous definition.

(Refer Slide Time: 34:00)



Theorem 7.11

Every NTM which runs in $t(n)$ time has an equivalent DTM that runs in $2^{O(t(n))}$ time.

Proof: Similar to the proof that NTM is equivalent in computation to DTM. We simulate all computation paths and check if any one leads to an accept.

Suppose b is the largest number of

Now, I just want to conclude and before I conclude, I just want to say two small things. I believe we will be able to share these notes with you and you can read the proofs detailed proofs, but I just share it I just explained in brief now. Again, this is theorem 7.11 these numbers are from Sipser. I think the second edition is what I have perhaps the third edition is the time complexity chapter. What it says is that?

If there is no deterministic Turing machine that transcends tn time deciding a certain language. Then there is an equivalent deterministic Turing machine that runs in the exponential of tn time, so to power order tn time that decides the same language. So, equivalent means they both decide the same language. And if you recall, t o c proof that a non-deterministic Turing machine is equivalent to a deterministic Turing machine.

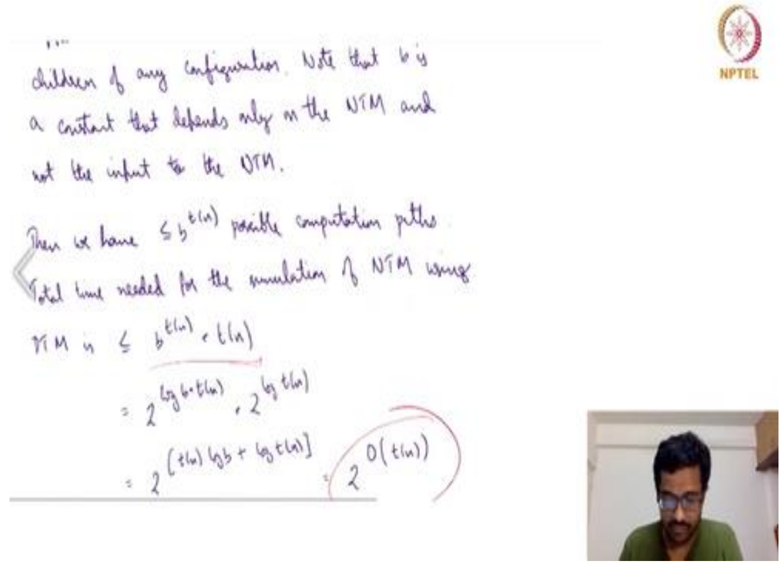
This is kind of the same proof what we do is we simulate basically the non-deterministic Turing machine has to decide, if there is at least one accepting path. So, but then non-deterministic Turing machine can automatically kind of find the path if it accesses, but the deterministic Turing machine does not have the power to non-deterministically search for an accepting path.

So, it has to kind of do a depth first search or breadth first search in this graph in the big configuration graph.

So, essentially that is the proof. So, you want to search in this graph which is a problem of usual traversal like depth first search or breadth first search. So, what is the maximum number of configurations that can be there that is the number of vertices in this graph or a tree. And to estimate the number of configurations or a to estimate an upper bound on the number of configurations, we make it assumption that not we assume that the maximum branching of any configuration is some constant and we call that constant b.

So, the maximum branching will be always limited by some number, because it is just a function of the Turing machine and so, it will be a constant. So, let us assume it is b.

(Refer Slide Time: 36:25)



children of any configuration. Note that b is a constant that depends only on the NTM and not the input to the NTM.

Then we have $\leq b^{t(n)}$ possible computation paths

Total time needed for the simulation of NTM using TM is $\leq b^{t(n)} \cdot t(n)$

$$= 2^{\log b \cdot t(n)} \cdot 2^{\log t(n)}$$

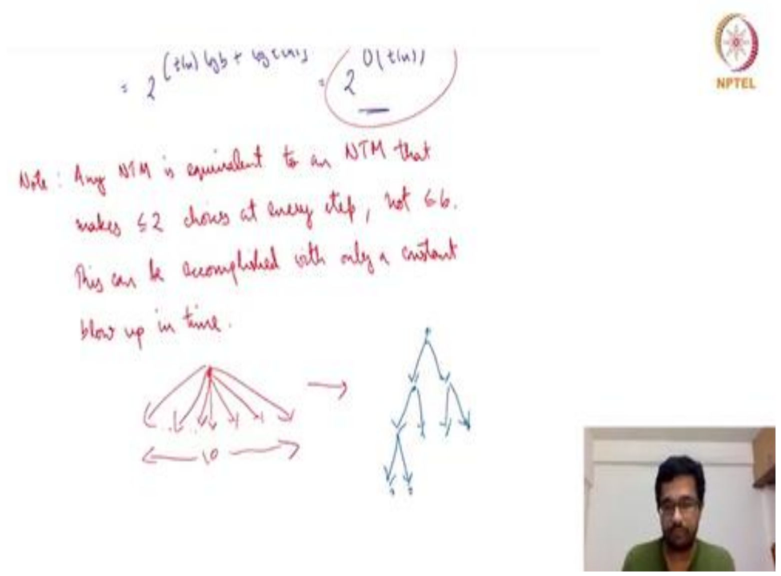
$$= 2^{[t(n)\log b + \log t(n)]} = 2^{O(t(n))}$$

And once it is bounded and it will be bounded for once the Turing machine is fixed. And it does not, vary as per the input length, it is just fixed based on the Turing machine. Now, I teach at level one there is one node C level 0 there is 1 configuration level one there could be at most b configurations, maybe not as much. Second level two there could be squired, maybe not as much and so on.

So, we have and the number of computation paths will be b power t n, t n is the total number of steps required for an optimistic Turing machine. And using simple calculation we can see that this the number of states in this graph is this one b power t and in two multiplied by t n and we can see that this is upper bounded by 2 power order tn, which is a claim. So, any non-deterministic Turing machine has an equivalent Turing deterministic Turing machine with only an exponential or with an exponential blow up in time;

Maybe I should not say only because the exponent the blow up is huge. So, if there is a knowledge of the Turing machine that runs in order n time the corresponding equivalent Turing machine constructed using this procedure that is into power order n time. So, in specific cases you may be able to find simpler algorithms, but this is a general proof and this gives you a general bound which is kind of very bad.

(Refer Slide Time: 38:07)



$$= 2^{(t(n)\log b + \log t(n))} = \boxed{2^{O(t(n))}}$$

Note: Any NTM is equivalent to an NTM that makes ≤ 2 choices at every step, not 66. This can be accomplished with only a constant blow up in time.

And this part you can just read it out when I share the notes. All it is saying is that the branching factor if it is large, we can replace it with a subtree of a branching factor at most two at the expense of the number of levels in the tree, you can just have a look and that is it for this lecture. So, just to summarize it subsides three SAT CNF SAT characterization for NP and two simple results which say that any non-deterministic Turing machine the runs in time tn has an equivalent deterministic Turing machine that runs in two power order tn times. In the next lecture, we will see NP completeness. See you there. Thank you.