

Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Verifiability and NP

(Refer Slide Time: 0:17)

Verifier Model for NP

We saw that SAT ∈ NP, 3-COLORABLE ∈ NP and SUBSET-SUM ∈ NP. In each of these proofs, we "guess" a solution and then deterministically verify it. The natural question, at this point, is:

Can we construct every nondeterministic machine to this format?

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 v to verify that x is in



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.

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.



Hello and welcome to lecture 47 of the course, theory of computation. This is also the beginning of week 10. In this week, we will see more about the class NP and we will also see NP completeness. So, we will first begin with a different characterization of the class NP.

So, in the previous week, in week 9, we saw that NP stands for all the computational problems or all the languages that can be decided by a non-deterministic Turing machine in polynomial

time. So, that has a non-deterministic polynomial time decider. So, in this lecture, we will see a different characterization for it, one that is based on verifiers.

So, to understand this characterization, let us take some examples. So, in the previous lecture, we saw a couple of languages that are in NP. So, one was SAT, one was three colorability and one was subset sum. So, if you recall the proof that these three languages were in NP, so in the case of SAT, what we did was to guess an assignment of the variables involved.

And then check that this assignment is a satisfying assignment. In case of subset sum, we guessed a subset of the given set and checked what is the sum of the given subset. And in three colorability, we guessed a three coloring or we non-deterministically came up with a three coloring and checked whether this is a proper three coloring.

So, just to give you a bit more detail on one of these, let us say three colorability, suppose there was a valid three coloring, then the valid three coloring will be one of the possible guesses that we will make and hence we will accept the non-deterministic machine will accept. If the graph is not three colorable, whatever be the guesses, none of them will accept.

So, what I am saying is that the algorithms that I have described is a proper correct non-deterministic algorithm and the running time is all polynomial time, it is easy to see. So, the point that I am coming to is that in all these three cases, there is a common framework. We guess something which is completely non-deterministic and then we do some verification which is completely deterministic.

So, we have some non-deterministic choices made at the beginning followed by an entirely deterministic process. So, this is called guess and verify, where the guess stands for the non-deterministic part and the verify stands for the deterministic part.

So, now the very very natural question to ask at this point is, is it the case that any non-deterministic Turing machine or any NP machine can be converted into something that does this, like guess and verify. So, in these three cases it was a guess and verify situation, but perhaps we could have non-deterministic Turing machine that makes some non-deterministic choices, then does something deterministic or make and then again make some non-deterministic choices or maybe it makes non-deterministic choices throughout.

So, now can any non-deterministic Turing machine be converted into something where all the non-deterministic choices are made at the beginning followed by an entirely deterministic process. So, that is the question that we are trying to answer through this verifier model. So, the answer is that yes, any non-deterministic Turing machine can be converted into a guess and verify non-deterministic Turing machine and in particular we will see this guess and verify model specific to the class NP.

So, we will first formalize this notion of guessing and verifying and then see the characterization. So, first we will define what is a verifier. A verifier for a language is an algorithm V or a deterministic Turing machine V such that the language is a set of all X whose membership can be verified by the deterministic Turing machine.

So, what do I mean by whose membership can be verified? So, the language X , the L consists of all X whose membership in L can be verified by V along with a string Y , meaning, so this Y helps in the verification that X is in the language. So, right now it may seem a bit abstract but it is easy to see once we have the example. Suppose X is a Boolean formula and L is satisfiability, the class of or the set of all satisfiable Boolean formulas.

So now, let us say Y is a satisfying assignment, so now given a satisfying assignment I can just check, I substitute this satisfying assignment into the formula. So, satisfying assignment means X_1 is true, X_2 is false some assignment which satisfies the formula. So, now given a satisfying assignment it is easy for us to check that this formula is satisfiable.

If we were not given a satisfying assignment we may have to do some other thing. So, V in this case could be an algorithm or a decider that just substitutes the given assignment which is given by Y sorry, given assignment which is given by Y into the formula X and then accepts if it is a satisfying assignment, if X evaluates true on Y .

In case of 3 colorability, X is a graph that is 3 colorable and Y is an actual valid 3 coloring. So, now and the verifier is a decider that actually assigns the proposed 3 coloring and then verifies whether it is a proper 3 coloring. So, we will accept if the coloring given by Y is an actual 3 coloring of X , otherwise it will reject. So, this is the role of Y .

So, a verifier is a device or a decider or a deterministic Turing machine that accepts X and with the support of Y . So, Y is something that helps us verify that X is a member of the language and a verifier is something just that conducts the verification. So, verifier is entirely

deterministic, it just has to check something that, in the case of satisfiability, does this assignment satisfies this formula.

In case of 3 coloring it checks that this coloring is an actual 3 coloring of this graph. So, Y is called, Y is sometimes called a proof or a witness or a certificate. So, the thing is why is it called proof or witness or certificate because Y is in some sense an evidence or a proof that X is in the language.

So, you want to know whether this graph is 3 colorable, so an actual 3 coloring is something that you can use to certify that this graph X is in the class of all 3 colorable graphs or we can say the 3 coloring is a witness that this graph is a member of all the 3 colorable graphs or a certificate of the same thing.

So, this is called a verifier. A verifier is a Turing machine that is able to verify the membership of a string X in the language with the help of a proof or a witness string or a certificate string Y . So, the language is just consisting of X , the language is just the set of all satisfiable Boolean formula or the set of all 3 colorable graphs. Y is a proof or a witness or a certificate that helps us verify that the graph is 3 colorable or the formula is satisfiable.

So, why is this whole thing interesting? This is because to actually determine whether a given formula is satisfiable is hard because right now it is not clear how you will go about doing it. Maybe the only thing that you can do is brute force by checking all the possible assignments. This is one possibility. There may be slightly better algorithm but not significantly better. There is no efficient algorithm. So, we do not know of any polynomial time algorithm.

But suppose I give a satisfying assignment and then I ask you to verify that this formula is satisfiable. You can just substitute this assignment and see for yourself that it is indeed a satisfying assignment. So, now in the second case you are just asked to verify that this formula or this assignment satisfies this formula.

This is something much simpler to do than to search the whole space of assignments and see if there is some assignment that satisfies this. So, given an assignment to verify that this is a satisfying assignment is much much simpler than deciding whether it is satisfiable. Similarly for 3 colorability. Given a graph to decide whether it is 3 colorable one has to check all possible 3 coloring assignments.

But given a 3 coloring it is easy to verify that this is a valid 3 coloring. Given if it is a valid 3 coloring it is easy to verify. So what the point I am making is that it is much easier to verify something. So, the existence of a y which is existence of a proof and the requirement to verify that x is in the language with y is much simpler than without being given anything to test if x is in the language.

Verification of x being in the language with the help of y is much simpler. So, that is the situation. So, y is a proof or witness or certificate that helps us determine that x is in the language.

(Refer Slide Time: 10:58)



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.

$L \in \text{NP} \iff \exists$ a polynomial time verifier algorithm V such that

$$x \in L \iff \exists y, |y| = \text{poly}(|x|), \forall (x, y) = 1$$

Contrast this with P.



And what is NP? NP is the class of languages that have polynomial time verifiers. This is the characterization of NP using the guess and verify model. So, we earlier saw that NP is a class of languages that have a polynomial time non-deterministic decider. Now I am saying it is a class of languages that have polynomial time verifiers.

In other words, what do I mean by polynomial time verifier? It is what we just discussed. So a language is an NP. So L is an NP if there is a polynomial time verifier algorithm for the language L . So again just to elaborate it a bit more. So what do we want to have? What property do we want the verifier to satisfy? If x is in the language, there should be a y such that the verifier can verify the membership of x with the help of y .

And in addition, so this is what we discussed as verifiers earlier. And in addition now, I want everything to be polynomial time. So earlier in the definition of verifier, I did not say anything

about polynomial time. But now that I am talking about NP, I want it to be polynomial time. So now, which means the verifier should be running in polynomial time.

And also the length of y should be also in polynomial in the length of x because if y is much longer, then the whole thing does not make sense because the verifier runs polynomial in polynomial time in the length of x and y . So if y is much longer than x , then the verifier could take a long time which again defeats the purpose. So L is in NP if there is a polynomial time verifier for L and what should the verifier, what properties do we want the verifier to satisfy?

For any x that is in L , there should be a y which can be verified polynomially by the verifier, meaning the y should have a length polynomial. So length of y should be polynomial in the length of x and V should also be polynomial time verifier, meaning a polynomial time deterministic Turing machine that outputs 1 when x is in the language. So 1 is like accept and as again 0. So think of this 1 as a Boolean output.

So just to understand what is the negation of this? If x is in the language this is the case, if x is not in the language, if x is not in the language, so if a graph is 3 colorable there should exist a proper 3 coloring. If the graph is not 3 colorable, no matter what 3 coloring you try, none of them will be a proper 3 coloring.

So for all the possible certificate strings or for all the possible 3 colorings which are of polynomial length, $V(x, y)$ should be 0, meaning whatever 3 coloring you assign should not be a proper 3 coloring, if the graph is not 3 colorable. So is the negation. If x is in L , there should exist a proper 3 coloring, if L being the language of all 3 colorable graphs. If the graph is not 3 colorable, then no matter what 3 coloring you assign, it is not going to work.

(Refer Slide Time: 14:44)

that have polynomial time verifiers.

$LNP \Leftrightarrow \exists$ a polynomial time verifier algorithm V such that

$x \in L \Leftrightarrow \exists y, |y| = poly(|x|), V(x, y) = 1$



$x \notin L \Rightarrow \forall y, |y| = poly(|x|), V(x, y) = 0$

Contrast this with P.

$LP \Leftrightarrow \exists$ a polynomial time decider algorithm A such that

$x \in L \Leftrightarrow A(x) = 1$

P. N. T. ...



So just one exercise, one easy thing to do is to just compare with what is the definition of P. P is the class of all languages that have polynomial time deterministic algorithms. So if L is in P, then there is a polynomial time decider algorithm A such that whenever x is in the language, A(x) should be equal to 1. So if x is not in the language, A(x) will be equal to 0.

$$X \in L \Leftrightarrow A(x) = 1$$

$$X \notin L \Leftrightarrow A(x) = 0$$

So notice this, this is much simpler because P is a much simpler class to understand. So if x is in the language, the decider will accept and if x is not in the language, the decider will reject. Whereas in the case of NP, if x is in the language, there is a certificate string that can be used by the verifier to verify that x is in the language.

If x is not in the language, no matter what certificate string you try, it should not work. So otherwise it does not make sense. Because let us say if the graph is not 3 colorable, then whatever 3 coloring you try should not lead to a proper 3 coloring. So we saw two definitions. One was that NP is a class of all languages that can be decided by non-deterministic polynomial time Turing machines.

And now this verifier model, NP is a class of languages that have polynomial time verifiers. So let us see the proof of this.

(Refer Slide Time: 16:23)

$x \in L \Rightarrow \exists y$

Proof: Two directions

(1) NP \Rightarrow Poly time verifier.

Suppose L \in NP. Then there is an 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.

Verifier machine: V on input x, y .

$L \subseteq NP$



$x \in L \Rightarrow \exists y$

Proof: Two directions

(1) NP \Rightarrow Poly time verifier.

Suppose L \in NP. Then there is an 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.

Verifier machine: V on input x, y .

$L \subseteq NP$



$x \in L \Rightarrow \exists y$

Proof: Two directions

(1) NP \Rightarrow Poly time verifier.

Suppose L \in NP. Then there is an 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.

Verifier machine: V on input x, y .

$L \subseteq NP$



So suppose L is in NP, meaning L has a non-deterministic polynomial time decider. Now we will show that it has polynomial time verifiers. So both directions of the proof are not that hard, both of them are fairly straightforward. So suppose L is in NP, which means there is a non-deterministic Turing machine N for L and the non-deterministic Turing machine runs in n^k time and that decides that.

So suppose x is in the language, then that means that when x is input to the non-deterministic Turing machine, there is a computation tree. If x is in the language, there is at least one accepting path. So let us say this is the accepting computation. So I am trying to highlight it here and finally till here. So by the new color, the green or whatever.

And so here we take the first option, here we take the third option, here we take the second option, here we take the second option and finally we take the second option. So the point is if x is in the language, there is an accepting computation path. There could be multiple accepting computation paths, but there is at least one accepting computation path. Now we want a polynomial time verifier for the language. So the verifier can do something.

So if I give you a non-deterministic Turing machine and tell you that there is a specific path, this is the path, you first take the first choice, basically I tell you this path, the path that we just marked here, I just tell you this path and then ask you to verify, instead of asking you to explore all possible paths, I ask you to verify, just traverse this particular path and see if it leads to acceptance.

And that is one way to convince you that the string x is accepted by the Turing machine N . So again the setting is that you have a deterministic Turing machine and I want you to verify that this non-deterministic Turing machine accepts x , but you do not have the bandwidth or the time to test all the possible computation paths and simulate the entire NTM N using your deterministic Turing machine.

But I am telling you do not simulate the entire thing, instead just walk through this path and check that it accepts. And if once you do that, then you can convince yourself, you can be sure of yourself that this string is accepted. So the only thing that I need to do is tell you the way to which path leads to acceptance. And this is the idea that we can use to prove this direction. So whatever I just said is what is the verifier algorithm.

(Refer Slide Time: 19:36)

Verifier machine: V on input x, y .

1. Simulates N on input x . When N has to make a non-det. choice, V is guided by y .
2. Accept iff N accepts.

(2) Poly time verifier \Rightarrow NP

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

NTM decider N :

- (1) Guesses a string y of length n^k .
- (2) Run V on $\langle x, y \rangle$.
- (3) Accept if and only if V accepts.



So what is the verifier algorithm? The verifier algorithm simulates the non-deterministic Turing machine on the input x . But then the verifier is deterministic, it cannot simulate the entire non-deterministic Turing machine because then it will not be polynomial. So instead it does not guess all the paths, nor does it try to exhaustively enumerate all the paths.

It just walks through one path. This path or the path that it walks through is specified by the Turing machine y , sorry the string y . So y could be the string that says something like y could be the string that says 1 followed by 3 followed by 2 etc, etc. So by this we are trying to encode that you first take the first option, then you take this third option, then you take the second and so on.

So the verifier Turing machine which is deterministic just simulates N but does not entirely simulate N , it only simulates one computation path of N which is given by the string y . So y dictates which or y is telling which path to simulate and then if it leads to an acceptance, the verifier accepts. So given the non-deterministic Turing machine, there are many possible computation paths and x is accepted if at least one of them accepts.

So the correct witness string is the identity of that path which leads to acceptance. If x is not accepted by the non-deterministic Turing machine, whatever y you give it cannot be verified. So the verifier machine is simply simulator of n on the input and whenever there is a non-deterministic choice, it looks up y to decide which path to take. That shows that if L is in NP, it has a polynomial time verifier.

So clearly the verifier runs in polynomial time because L is also polynomial time. Verifier just walks through one path of n and verifier is deterministic. So that is one direction. The other direction is also not that difficult. It says that if a language is a polynomial time verifier, then the language is in NP.

(Refer Slide Time: 22:03)

2. Accept iff N accepts.

(2) Poly time verifier \Rightarrow NP

Suppose there is a poly time verifier V for L .

Let us say V runs in time n^k .

NTM decider N :

- (1) Guess a string y of length n^k .
- (2) Run V on $\langle x, y \rangle$.
- (3) Accept if and only if V accepts.

What is the proof/certificate?

SUBSET-SUM : The subset itself.



So suppose there is a polynomial time verifier for the language L . Let us say the polynomial time verifier is V and let us say V runs in time n^k . Now we need a non-deterministic Turing machine for, so now to show this characterization that polynomial time verifier implies that the language is in NP. NP now is that, it is a class of languages that have deciders that run in non-deterministic polynomial time.

So we have to come up with a non-deterministic polynomial time decider. So what is a decider? The decider is a non-deterministic Turing machine that runs the verifier, but with what witness string? Because there is no witness string that we are telling. So what it does is, it guesses the witness string.

So the non-deterministic Turing machine just guesses a random or guesses a non-deterministically a witness string y of the appropriate length and then initially it guesses the witness string and then runs or simulates the verifier on the input x and the witness string as the witness string y . And then if V accepts the pair x, y then the non-deterministic Turing machine accepts.

So suppose x is a string in the language then that means that there is some witness string that would lead to the acceptance of x . Suppose if x is a three colorable graph there is at least one proper three coloring which can be used to verify that the graph is three colorable. So if x is in the language there is a y that will help us verify that x is in the language.

Now the non-deterministic Turing machine one of its many guesses would be the correct y and so it will lead to acceptance. If the graph is not three colorable no y is going to make the verifier accept and hence all the non-deterministic Turing machine paths will lead to reject. So again the non-deterministic Turing machine is very simple, it guesses the witness string and then runs V the verifier Turing machine on the input string x and the guessed witness string y and accepts if and only if V accepts. S

So basically here the Turing machine is basically guessing y , so many possible guesses of y and then followed by running of so here y equal to something 0000 and till y equal to 1111 and then followed by running of the verifier. So then now V , V and so on and then based on that you accept or reject.

So this shows that if the language has a polynomial time verifier then the language is in NP. So that completes the proof that so we have shown both directions if L is in NP then it has a polynomial time verifier and that if L has a polynomial time verifier then it is in NP. So now this gives us a now after the completion of this proof it gives a another characterization for NP. NP is a class of all languages that have polynomial time verifiers.

So now if you want to show something is in NP instead of showing that it has a non-deterministic polynomial time decider I can also show that it has a deterministic verifier. You can verify the membership of a particular string in the language.

(Refer Slide Time: 25:54)

NTM decider N :

- (1) Guess a string y of length n^k .
- (2) Run V on $\langle x, y \rangle$.
- (3) Accept if and only if V accepts.

What is the proof/certificate?

SUBSET-SUM : The subset itself
 CNF-SAT : The satisfying assignment.
 3-COLORING : The valid 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 now we have another characterization for NP. Now just to kind of recap or just to kind of revise so what is the proof or certificate or witness? So y is the string, the string y is the proof or the certificate or the witness and in different problems what was it? In the case of subset sum, the subset that adds up to t was the that adds up to the target sum itself was the proof.

In the case of three colouring the proper three colouring was the proof. In the case of SAT or CNF SAT this assignment that satisfies the formula was the proof. So these are all proofs that the given string so in the case of subset sum in the given subset or given instance is a yes instance; in the case of satisfiability it is a proof that the given formula is satisfiable; in the case of three colourability it is a proof that the given graph is three colourable.

So the proof here is something that is quite straight forward. In the case of subset sum the correct subset sum that adds to the target sum. In the case of CNF SAT it is a satisfying assignment, in case of three colouring it is a proper three colouring. So that completes the characterization that characterization of NP that NP is a class of languages that have polynomial time verifiers. So now this is another way to think of NP.

(Refer Slide Time: 27:31)

SUBSET-SUM : The subset itself
CAF-SAT : The satisfying assignment.
3-COLORING : The valid 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

Proof: Quantifying the proof of Theorem 3.16
seen in lecture 31, where we showed that
every NTM has an equivalent DTM.
We simulate all computation paths and check
if any one leads to an accept.

Suppose b is the largest number

Now I want to just quickly state and explain the proof of the fact that so you may recall that in chapter 3 we said that every non-deterministic Turing machine has an equivalent deterministic Turing machine. So now one question that we can ask now that we are bothered about time that it takes how much time does it take for a deterministic Turing machine to run in non-deterministic Turing machine or simulate a non-deterministic Turing machine?

So it turns out that the answer is if the non-deterministic Turing machine runs in t time, time $t(n)$, the deterministic simulator takes time $2^{O(t(n))}$. So it is exponential time in $t(n)$. So that is one of the bounds. There are slightly better bounds in the literature but this is a simple result. So how do we get this result? So turns out that the proof is something that we have already seen, is the same proof that we saw in chapter 3.

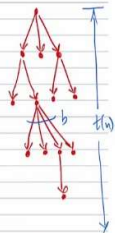
To be precise we saw it in lecture 31 where we said that every NTM has an equivalent DTM. Now all that we are doing is we are just going through the same proof and kind of computing the time taken for the simulation. So earlier we told the proof without really bothering about the time. So now we are going to measure the time of the simulation described in the same proof.

(Refer Slide Time: 29:03)

every NTM has an equivalent DTM.

We simulate all computation paths and check if any one leads to an accept.

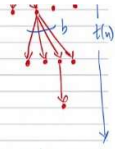
Suppose b is the largest number of children of any configuration. Note that b is a constant that depends only on the NTM and not on the input to the NTM.



Then we have $\leq b^{t(n)}$ computation paths. Total time needed to check all these paths using the DTM is upper bounded by

$$\leq b^{t(n)} + t(n)$$
$$\rightarrow \log b + t(n) \rightarrow \log t(n)$$


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



Then we have $\leq b^{t(n)}$ computation paths. Total time needed to check all these paths using the DTM is upper bounded by

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

Note: Any NTM is equivalent to an NTM that makes ≤ 2 choices at every step, not $\leq b$



So just to describe the proof we wanted to simulate a non-deterministic Turing machine by a deterministic Turing machine. So what we do is we kind of consider the computation tree and look at all possible computation paths and we accept if we find an accepting computation paths. And the thing was that we do this in a breadth first search manner that is what we said in the chapter 3. We simulate everything to one level then second level and so on.

So anyway that particular detail is not very critical here because it is all deciders but let us see the time complexity of this process. So before that let B be the maximum number of children any configuration can have in this computation tree. So in this particular partial tree that I have drawn the maximum children that any configuration has is this the one that I have marked here it has there is a configuration with four children.

So B let us say it is four and if a non-deterministic Turing machine is defined the constant B only depends on the Turing machine not on the input. So it does not really depend on the input, it just depends on the Turing machine. So this is something that one has to understand. So now we know that the non-deterministic Turing machine runs in time $t(n)$ which means this whole computation from start to accept from root to the leaf that is farthest is of height $t(n)$.

And since every configuration can have at most B children there could be at most $B^{t(n)}$ computation paths. So the maximum that can happen is from the root there are B children from the every B children there are further B children. So the second level there are B, the third level there is B square, fourth level there is B cube and so on.

So finally at the level $t(n)$ we have $B^{t(n)}$ possible leafs. So we have at most $B^{t(n)}$ computation paths and each computation path is of length $t(n)$ or at most $t(n)$. Each computation path is of length at most $t(n)$. So to go through all these paths and to check all these paths we need to spend $B^{t(n)} * t(n)$ time.

And what follows is a very standard calculation $B^{t(n)}$ we can write it as $2^{\log(b)*t(n)}$ and $t(n)$ we can write as $2^{\log(t(n))}$ and by adding the exponent we get this the third equality

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

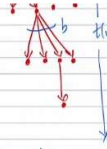
and since $t(n) * \log(b)$ dominates $\log(t(n))$ we can just write it as $2^{O(t(n))}$.

So the proof is very simple, the simulation of the NTM by the DTM is just by traversing the computation tree. The tree has height at most $t(n)$ because that is the running time of the NTM and at each level of the tree it has B children. So the maximum number of computation paths to consider is $B^{t(n)}$. So the running time is $B^{t(n)} * t(n)$ which simplifies to $2^{O(t(n))}$ or which is upper bounded by $2^{O(t(n))}$.

Hence the upper bound on the running time of the simulation is $2^{O(t(n))}$. So any non-deterministic Turing machine that runs in $t(n)$ time that determines the simulation takes exponential time of that. If the non-deterministic Turing machine runs in n time the deterministic equivalent runs in $2^{O(n)}$ time which is quite bad. So it is significantly worse. That is another reason why we prefer to have deterministic polynomial time algorithms.

(Refer Slide Time: 33:37)

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



Then we have $\leq b^{t(n)}$ computation paths.
Total time needed to check all these paths using the DTM is upper bounded by

$$\begin{aligned} &\leq b^{t(n)} \cdot t(n) \\ &= 2^{\log b \cdot t(n)} \cdot 2^{\log t(n)} \\ &= 2^{\left[\log b \cdot t(n) + \log t(n) \right]} \\ &= 2^{O(t(n))} \end{aligned}$$

Note: Any NTM is equivalent to an NTM that makes ≤ 2 choices at every step, not $\leq b$.



And finally I just want to say one more thing that this may feature in some of the proof we may be assuming this, that is why I am saying it at this point. So we mentioned this branching factor B which is the maximum number of children any specific configuration can have. Suppose the branching factor is some number like 10. Now the point is that we can without loss of generality we can assume that every configuration has at most 2 children.

At every stage we have at most 2 choices and not B choices. Why is that? That is because we can like a node has let us say 10 children, we can replace that one node with 10 children with a small tree. So where this node corresponds to this node here and the small tree will have 10 leaves and these 10 leaves correspond to the 10 children here.

So basically the point is that if the non-deterministic Turing machine has many like takes up to 10 choices we can replace it every non-deterministic step with a step that takes at most 2 choices. So sometimes this is convenient in some proofs. We can assume that B is equal to 2 and it does not increase the time too much because if B is usually considered to be a constant. So now to build a tree with 10 leaves we can be the height is height required is 4.

So the height required will be 4. So here I think I have marked 1 2 3 4 5 leaves. So maybe one more level we need to mark. So something. So height required is 4 which is again a constant. So it is just a constant blow up for the running time of the NTM. So initially if it took $t(n)$ time, now maybe it is some $t(n)$ multiplied by $\log(B)$ time, $\log(B)$ to the base 2. So asymptotically it is the same. So that is all that I have to say in lecture 47.

(Refer Slide Time: 36:10)

Verifier Model for NP

We saw that $\text{SAT} \in \text{NP}$, $3\text{-COLORABLE} \in \text{NP}$ and $\text{SUBSET-SUM} \in \text{NP}$. In each of these proofs, we "guess" a solution and then deterministically verify it. The natural question, at this point, is:

Can we connect every nondeterministic machine to this format?

A verifier for a language L is a decider (algorithm) D 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.

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.



So first we saw the verifier model for NP where we saw that the every non-deterministic Turing machine or every language that can be decided by a non-deterministic polynomial time Turing machine has a polynomial time verifier and if a language has a polynomial time verifier then it can be considered to be NP, meaning it has a non-deterministic polynomial time decider. So meaning what does it mean when I say that L has polynomial time verifier?

There is a machine deterministic Turing machine V which runs in polynomial time which can verify the membership of any string x in the language with the help of a proof string or a witness string y . So that is the meaning that NP is a class of languages that have polynomial time verifier. So for instance if L is a class of 3 colorable graphs and x is a 3 colorable graph then y is a 3 coloring that helps us verify that the graph is 3 colorable.

If L is a class of satisfying Boolean formula then x is a specific satisfiable Boolean formula then y is the satisfying assignment that we can use to verify that x is in the group of or x is in the class of satisfiable Boolean formulas. And the two directions were both straight forward.

(Refer Slide Time: 37:43)

3-COLORING : The valid 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.

Proof: Quantifying the proof of Theorem 3.16 seen in lecture 31, where we showed that every NTM has an equivalent DTM.

We simulate all computation paths and check if any one leads to an accept.

Suppose b is the largest number of children of any configuration.



And then we saw that a non-deterministic Turing machine that runs in $t(n)$ time can be verified can be simulated by a deterministic Turing machine in $2^{O(t(n))}$ time.

(Refer Slide Time: 37:57)

$$\begin{aligned} &\leq b^{t(n)} \cdot t(n) \\ &= 2^{\log b \cdot t(n)} \cdot 2^{\log t(n)} \\ &= 2^{[t(n) \log b + \log t(n)]} \\ &= 2^{O(t(n))} \end{aligned}$$

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



And then finally we said this small fact about if you have a branching of big number then we can replace it with a small tree where the branching is at most 2. And that is all I have in lecture

47. In lecture 48 we will see NP completeness and we will see reductions and we will start moving towards NP completeness. So see you in 48. Thank you.