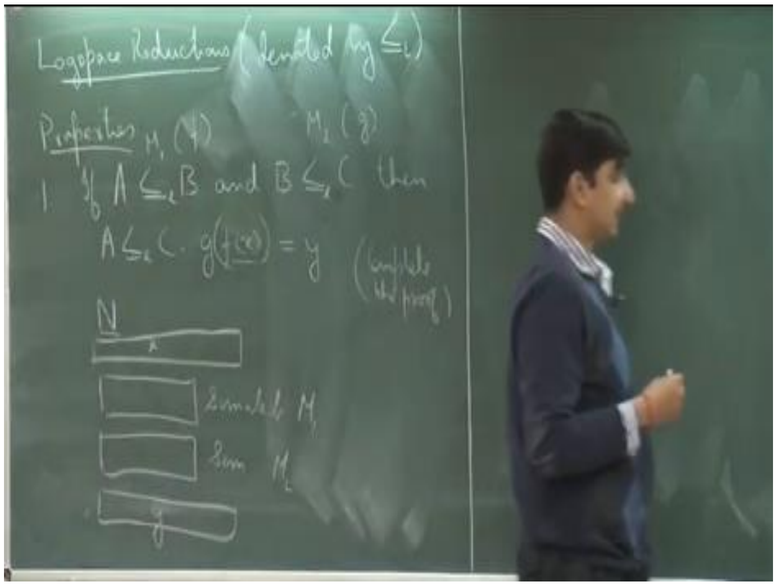
Computational Complexity Theory

Prof. Raghunath Tewari
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture -08
Introduction

So last time we discussed, we ended our class with the definition of logspace reductions. So, logspace reductions just to recall, logspace reductions are reductions which computes functions using a Turing machine which has a logspace bound on its work tape and its output tape is just a write only type. So, it just goes from left to right. So, that is logspace reduction. So, let us look at some properties of logspace reduction.

(Refer Slide Time: 01:03)


Logspace Reductions (denoted by ≤_L)
Properties M_1 (f) M_2 (g)
1 If A ≤_L B and B ≤_L C then
A ≤_L C. g(f(x)) = y
(complete the proof)
N
x
Simulate M_1
Sim M_2
y

And these are not very difficult to prove. So, the first property is a closure property. So, these reductions are closed under transitivity, so that notation is as follows. So, we have a L at as the subscript to denote it is a logspace reduction. So, the first property is if a language A reduces to B in logspace and B reduces to C then A reduces to C. So how do we prove this? So, suppose you have a Turing machine for this reduction.

So, let us say you have a Turing machine for this, let me call this M 1 and you have a Turing machine for this M 2. So, you have to be little bit careful over there. So, that is okay. So, suppose we want to build a new Turing machine N which computes this reduction and it simulates let us say M 1 on one tape and M 2 on the other tape. But you have to be little bit careful because we do not have the input of M 2.

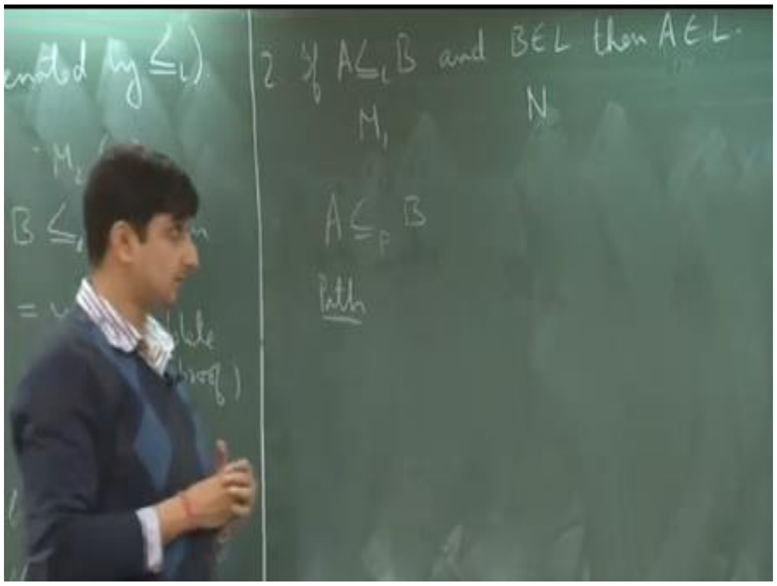
So, you are given some input x which belongs to this language A and you have to compute an instance y which belongs to language C. So, suppose this reduction computes the function f and this computes the function g. So, what we want to compute is f of x, I mean g applied on f of x. So, this is what we want, this is our y. So, we do not explicitly have f of x, so I just cannot simulate M 2 and if I try to just write down f of x that will create problem.

Because where do I store it because f of x is not guaranteed to be bounded by logspace. So, the idea is that we do not care about time. So, this is something that we discussed some lectures earlier. We do not care about the time taken to compute this reduction. We do not care about the time taken by this machine in what we do is we go ahead and simulate the machine M 2 that is we try to compute this function g.

And whenever it needs a bit of, f of x, so suppose it needs some jth bit of, f of x which is good look at the other tape as you said which contains the description of machine M 1. And it has the input x on its input tape, we just simulate M 1 on x and we find out what that jth bit of, f of x is and we carry on with our computation of g. So, there is so there are two tapes which we will imagine.

So, we will use the first tape to simulate the computation of M1 and will use the second tape to simulate the computation of M 2 and whenever M 2 requires a particular bit of, f of x, we will just go back to the first step will simulate M1 on the input which is x get that bit and proceed with the computation of M 2 and then finally on the output tape we will write g of f of x which is y. So, that is the idea, and this is again possible in log space. So, just you guys can complete the proof essentially you just follow this idea.

(Refer Slide Time: 06:13)



So, let us look at another property. If a language A reduces to another language B and B is contained in L then A belongs to L as well. Again, this is using similar ideas. So, we have a logspace machine for B whenever that logspace machine requires an input of the language B, we just use this reduction. So, basically, I want to show that a belongs to l so given an input, so I will just use this reduction and this logspace machine together and decide whether A belongs to L or not.

So, another reason why we need logspace reductions is because if we did, I mean, if it do not consider logspace reductions then this property is not true. In other words, if you are looking at only polynomial time reductions let us say and we have some language A reduces to B in polynomial time and B is contained in L. It does not imply that A is contained in L. So, we have to look at reductions which are of computation power that is not greater than the given classes so in this case L.

So, again the idea is the same. So, suppose you have a machine, you have a logspace machine M 1 which computes this reduction and you have a machine let us say N which decides if a string d is in L or not. So, what we do is given an input x, I want to decide if x belongs to A or not. So first what we do is we reduce x to an instance of B and then use this machine to check whether B that instance belongs to L or not.

Again, we have to be careful because when we are reducing, we do not want to write out the entire thing. Because then that would I mean, we do not have sufficient space for that. So, we will again just write out as and when or will run this machine M 1 as and when we need a bit of an instance of B. This basically the same idea, you mean to say that if A is reducible to B in polynomial time see then the problem is that, so basically what you can pick is, suppose you pick A to be some, I mean basically by the same thing that I said last time.

So, suppose A and B are both languages which are smaller than P. So, let us say that, they are both so B is a language which is in L and let us say A is a language which is a NL. So, I can always say that A reduces to B in polynomial time by doing the following, I pick two instances of B, a yes instance and a no instance and since I am already assuming that A is contained in NL. Let us say that A is the path problem.

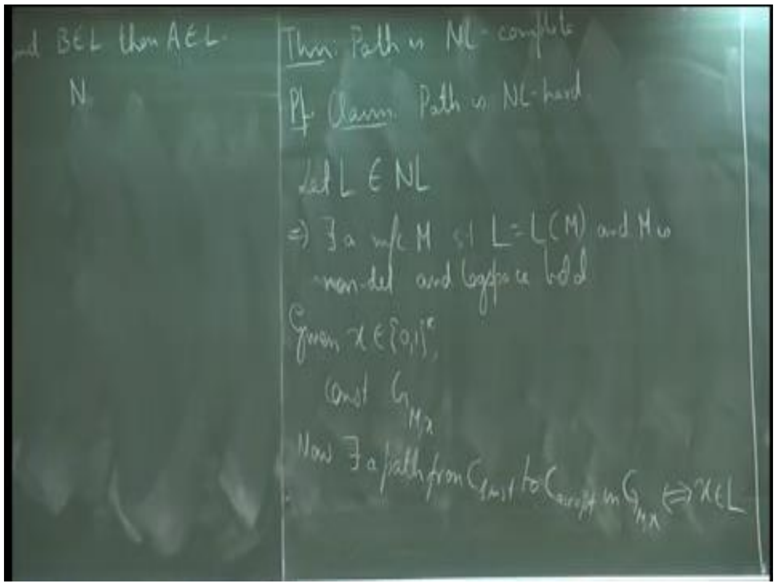
So, there is a polynomial time algorithm for A, so I just go ahead and compute A, right away. So, I will just use DFS or BFS and given any instance of A, I will just check whether it is a yes instance or no instance and depending on whether what answer I get, I will map it to the corresponding instance of B. So, if it is a yes instance, I will map it to the yes instance of B and if it is a no instance, I will map it to the no instance of B.

So, it is a very trivial kind of a reduction that I am doing. Basically, I am abusing the power of polynomial time reductions that I am given. So, instead of doing and then it will not imply that A belongs to L. Because A is a language which is not known to be contained in L. So, basically the idea of reductions is that they should translate the structure of one problem to the structure of one another problem. I mean the reduction should not be so powerful so that it can actually compute the problem that is given to us.

Because if it is actually computing the problem, we are abusing the power of reduction that is given to us, so is that clear. So, let me know Ii mean if it is not clear I mean, I can say this again but just let me know. What we are assuming here is that so suppose A is the problem path. So, we do not know if A belongs to L, in fact that is an open problem. But since we are using polynomial, the power of polynomial time reduction I can actually go ahead and compute A and then map it to A yes or no instance of B appropriately.

So, I am misusing the power of reduction that is given to us to actually solve the problem which we are not supposed to do. So, that is why it is important that we consider reductions which are at most as powerful as the languages that are the classes that we are considering.

(Refer Slide Time: 12:49)



…d $B \in L$ then $A \in L$.

N

Thm: Path is NL-complete

Pf Claim: Path is NL-hard

Let $L \in NL$

$\Rightarrow \exists$ a m/c M s.t. $L = L(M)$ and M is non-det and logspace bdd

Given $x \in \{0,1\}^*$,

Const $G_{M,x}$

Now $\exists$ a path from C_{start} to C_{accept} in $G_{M,x} \Leftrightarrow x \in L$

So, let us move on, so the next theorem that we will see is actually we have already seen the idea behind this theorem but nevertheless I will go ahead and state it that the problem path is NL complete. So, we have already seen that path is in NL, so you just guess a path from S to T and if your guess succeeds you output yes otherwise you output no. So, the question is how do we prove that path is NL hard?

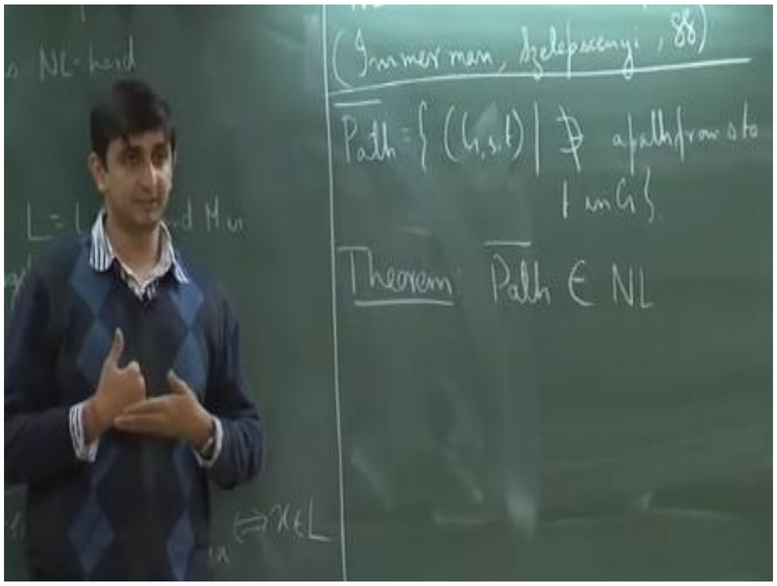
The claim is that path is not only in NL, it is NL hard as well. So, how do we argue this? Exactly, so it is basically the same ideas that we have seen in our earlier classes. So since so how do we prove path is NL hard. So, let us take any language L in NL. So, this means that there is some non deterministic machine which is deciding L and which is logspace bounded. So there exists a machine M such that L is equal to L of M and M is non deterministic and logspace boundary, that is a definition.

So, now since we know that such a machine exists, so given any input x, what we go ahead and do is we construct the configuration graph of the machine M on the given input x. And together with the vertices C start and C accept that gives an instance of the path problem. So, now there exist a path from C start to C accept in G m, x, if and only if what, if and only if this the given input x belongs to our language L. So that is the reduction.

So, it is basically just taking the machine M, taking the input x and outputting all the configurations basically have to output all the edges of G m, x. So, basically for all pair of configurations C and C prime by looking at the transition function of M, you check whether C prime is reachable from C in one step. If it is you just say that C, C prime is an edge in G m x and you output that edge.

So, you output the list of all edges and then you output the start configuration and the accepting configuration. So that is a so if x belongs to L then this is a yes instance of the path problem and if x does not belong to L this is a no instance. So, there is nothing dramatic happening over here it is just basically just follows, from the definitions, any questions? So, let us move on so we will see a very important result about space bounded complexity and this is one of the breakthrough results in complexity theory.

(Refer Slide Time: 17:44)


NL-hard
(Immerman, Szelepcsenyi, 88)
$\overline{Path} = \{ (G,s,t) \mid \nexists$ a path from s to t in G$\}$
Theorem $\overline{Path} \in NL$
$\Rightarrow x \in L$

So, basically what we will show is that, so we will show that NL is closed under complement. Basically, NL is equal to CO NL, and the reason why this is so very important is that, so recall that when we were discussing the class NP and we for the class CO NP, we saw that how difficult it is to prove that NP is equal to CO NP. In fact, what people generally believe is that these two classes are not the same.

And what is the reason for that. So, the reason for that is the following, so what does it mean to say that let us say NP is equal to CO NP, it means that even for nonexistence of a certain instance there is a small sized certificate. So, let us say if you want to say this with respect to the SAT problem. So given a formula phi, it is very easy to check existence of a satisfying assignment.

You just guess an assignment and that is a small sized assignment. But how do you prove that it is not satisfiable how can you come up with a short certificate which proves that phi is not satisfying. So, that is something which is very difficult. Because well there are exponentially many assignments and well one way to get a certificate is to basically try out all those ones and that is an exponential sized certificate.

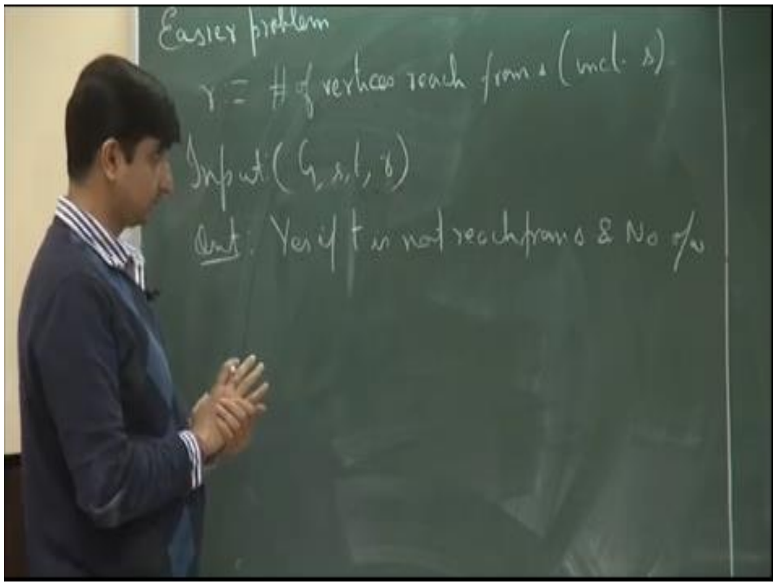
So, that is the fundamental reason why for non deterministic classes closure under complement is so very difficult. So, this result was shown independently by two people in 1988 and this was something which was not believed. I mean although we are looking at a different model, we are looking at space bounded complexity and in particular machines which can use only logarithmic amount of memory.

But still this was something which was not believed to be true. So this was very counter intuitive that how can non deterministic classes be closed under complement. So this was shown independently by two guys Immerman and Neelimerman and Robert Zelazny, I hope I have the spelling correct, in 88. So Zelazny was in Russia and Emmerman was in America. So, there was no communication between them. But so nevertheless this was a very important result.

So, we look at the proof of this theorem today. So, what we will show is that the language path complement. So, path complement is the set of all instances G, s, t such that there does not exist a path from s to t in G and path complement is a complete problem for CO NL. By again you can prove this by the same argument. So, what we will prove today is that for the rest of the lecture is path complement belongs to NL.

So, what do we want to show that given such a instance if s is not reachable, I mean if t is not reachable from s then we can answer that using an NL machine. So, let us look at an easier problem first. So, I just want to motivate, I mean how this proof was constructed. So, instead of looking at the actual algorithm, let us look at a easier problem.

(Refer Slide Time: 22:27)

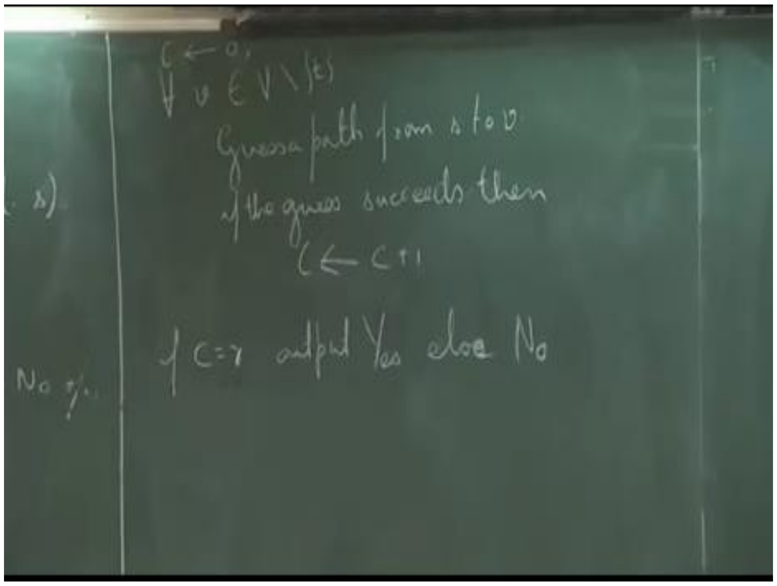

Easier problem
r = # of vertices reach from s (incl. s)
Input (G, s, t, r)
Out: Yes if t is not reach from s & No o/w

So, suppose somebody gives you the following number, so together with the input G, s and t he also provides you a number r which is the set of all vertices, I mean the not the set but the number of all vertices that are reachable from s including s. So, suppose you are given this a number r which is defined as the number of vertices reachable from s, of course including s, because s is trivially reachable from itself.

So, can you now give me an NL algorithm. So, what you are given as an input is you are given G, s, t, r together with the promise that r is the number of vertices that are reachable from s and what you have to output is, you have to output let us say yes, if s is not reachable or if t is not reachable from s and no otherwise of course in NL, how to do this? So, basically what the idea is that we will use this as a check. So, this number r will use it as a check.

So, let me give the algorithm and maybe we can discuss then why it is correct and how much time it is taking. So, the algorithm is as follows.

(Refer Slide Time: 25:11)



```
$C \leftarrow 0$,
$\forall v \in V \setminus \{t\}$
    Guess a path from s to v
    if the guess succeeds then
        $C \leftarrow C+1$

if $C = r$ output Yes else No
```

So, for all vertices v belonging to the vertex set of G minus t let us say. I guess a path from s to v. So, if the guess succeeds that is if I am actually able to correctly guess a path. Then well I need to keep a counter. So, I will keep a counter over here, C which will initialize to 0, then you increment the counter and you carry on. So, now when do we accept can we complete the rest of the algorithm and when do we reject?

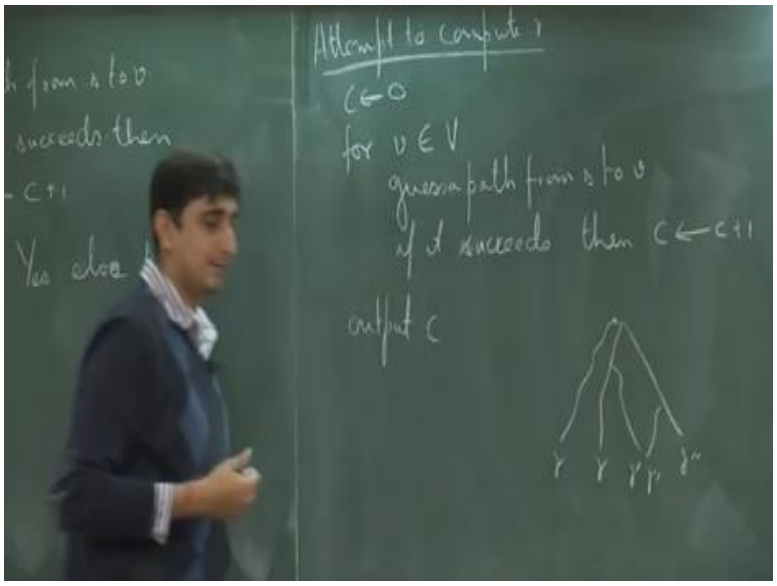
So, at the end of this loop if C equals r, what do we do? What can we say? We output Yes else No, so this is the correct algorithm and why is this correct? So, we are looking at all vertices in the graph other than t. So, if C actually if I am able to count, I mean actually if the counter reaches r it means that the number of vertices that are reachable from s it does not include t. Because if you look at the definition if you look at the way this algorithm is proceeding the counter can only be at most as large as this number r.

Because I never increment this counter for a vertex which is not reachable from s. So, there can be many computation paths along which this counter is strictly less than s where I mean there was a vertex which was reachable from s but I was not able to guess a path properly. So in which case we do not care? But there will be at least one computation path along which I will be able to correctly guess all the vertices that are reachable from s.

In other words where the counter will be exactly equal to r and of course t is not reachable from s which is why I will output Yes and otherwise I will output No. So, is this clear to everybody and this again it is easy to see this is in log space. Because I am using just few variables, I am using a variable here and maybe to guess these paths I will use one or two variables. So, just by using some constant number of variables, I can run this algorithm and each variable takes log n bits.

So, now the problem is how do I get this value r, how do I get the set of all vertices and the count of all vertices that are reachable from s. So, what if we do the following? So, what if we attempt the following?

(Refer Slide Time: 29:11)



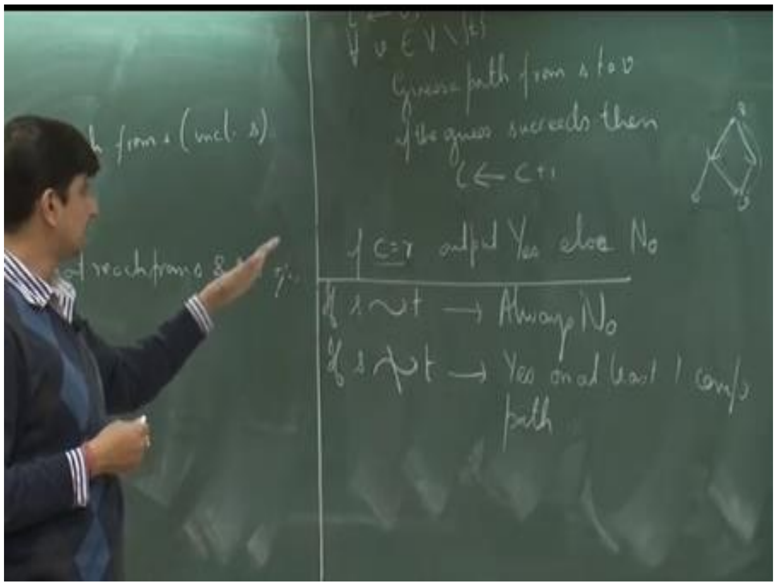
So, basically what we do is, so let us say we keep a counter C, so let me write this as attempt to compute r. So, what we do is we keep a vertex C as the vertex that we will use for counting and for all vertices v in V, I guess a path from s to v. If it succeeds is if I correctly reach the vertex V, then I increment the counter and I run this loop and then finally at the end I just output C. So, is this correct, so this is a non-deterministic algorithm, because I am doing some guesswork.

So, the first question is does it actually give the correct value of r along some computation path? It does right. So, does it give an incorrect value along some computation path can it give, it can right. In particular, here right here so suppose you have a vertex v that is reachable from s. But you are not able to correctly guess a path, you guess an incorrect path in which case you are not incrementing the counter.

So, in which case the final value that you output is strictly less than r. So, this is a non-deterministic algorithm. So, starting from the beginning, there are several computation paths along some paths maybe you output the correct value r and along some path you output some r prime, r double prime may be some a double prime again which are strictly less than r. Sorry, these values are strictly greater than r.

So, you are right, so if you are not able to reach. So, suppose so what do I want, what I want is that if t is not reachable from s there should be some accepting path, along some path out I should output Yes and if t is irreachable from s along all paths I should output No. So, let us look at the first case here.

(Refer Slide Time: 32:20)


from s (incl. s)
reach from s
∀ v ∈ V \ {s}
Guess path from s to v
if the guess succeeds then
C ← C+1
if C = r output Yes else No
If s ⇝ t → Always No
If s ⇝̸ t → Yes on at least 1 comp path

So, maybe I can analyze the correctness of the first algorithm. So, if there is a path from s to t. So, then note that since I am not including t in the set of vertices that I am considering this counter will never reach the value r along any path. Because there is a path from s to t so t is counted within this value r. And since I am not including t the counter c will always strictly be less than r.

So, therefore I always output No. Similarly, if t is not reachable from s, it can happen as you said it can happen that I incorrectly guess a path for which I do not and therefore I do not increment C and therefore when I come to this step C is less than r and I output No, that can happen. But the point is that there will be at least one computation path along which I will correctly guess all the vertices that are reachable from s and therefore c will be equal to r and therefore I will output Yes. So, it is yes on at least one computation path.

What do you mean by always output Yes? That is what I am saying but it can be that there is a path but I am not able to guess a path. Suppose you have a graph suppose you have a simple graph, and I am trying to guess let us say this is my vertex v and this is my vertex s. So, we know that there is a path from s to v, so let us say when I start guessing whether there is a path from s to v, I start from here.

So, I non-deterministically guess which way I should go. So, let us say if I am going left then I reach this vertex then again, I non deterministically guess which way I should go and then I can reach either this vertex or this vertex. So, along if I am guessing this path then I will never be able to find v, if I am guessing this path then I am able to find v also if I am guessing this path, I will be able to find v.

So, there is a sequence of non-deterministic choices along which I will find v and there can also be sequence of non-deterministic choices along which I will not find v. So, that is what I am saying that if the guess succeeds. So, even if a vertex is reachable it can it is possible that my guess does not succeed. Because I mean what does it mean to guess I mean, it just means that when I am at this vertex, I am just choosing which way I should go either this way or this way.

So, if a vertex is reachable then the claim is that there is at least one path which will lead me to the correct destination. So, if I am using this approach to compute r what can happen is that, along certain computation paths I will get a value which is, no, actually values which are smaller than r. So, this was correct, so r prime and r double prime they are strictly small smaller than r and therefore if i am using these values on top of this subroutines, I will get incorrect answers.

So, the point is that how do I compute the correct value of r? So, in fact this is the most crucial part of this algorithm. Any questions before we proceed. All possible paths, how do you know that the answer of every computation is No? That you do not know. Because the moment you proceed along a particular computation path you do not know what is happening in the other paths.

So, computation is proceeding only along one path. So, it is only the global property of the machine that you use to decide whether a string is acceptable or not. But locally within a computation path they are totally ignorant about what is happening along other parts. See that is the same problem with NP and CO NP as well. So, you can say that I mean if you want to use the same logic you can say that well I will guess a satisfying assignment and if along all paths I am not able to find a satisfying assignment then I will accept.

Let us say if I want to accept SATBAR that is not possible. Because I do not know what is happening, I mean I cannot get information about the other computation paths. You mean for the SAT problem. No, this problem. What do you mean by satisfying assignment? Basically, as you find a path, so if you find a path you flip that bit to one. So, however you test. How are you testing all the paths?

So, you do not know I mean you cannot keep a count on the number of paths. I mean you cannot index all the paths. So maybe the next time that you are guessing you will try guessing the same path and if you try to index all the paths that you cannot do it in log space. Because there are exponentially many paths it will take at least some super logarithmic number of bits. We have loops here, now I have a loop here as well as here.

So, here it is still deterministic I mean up to this point it is deterministic, but the moment I try to guess a path that is where I am introducing non-determinism so that will give me different computation paths. It is actually a loop so this is a I mean this is giving rise to a loop and the same thing here this is giving rise to a loop anything else. There we have the guarantee that if the string belongs to the language, there is some computation path which will give me a Yes.

Guessing is not the root of the problem. Basically, guessing is the same as non-determinism using non-determinism, because what is non-determinism? It means that from one configuration I can go to multiple configurations. Guessing a state in the middle of the no so that is what i was saying when I answered his question that you do not care. So, maybe there are sequence of guesses which will give you to a No.

So, suppose you have a instance x which does belong to a language. But there is a sequence of guesses which will lead you to a No. For example, here so v is reachable from s, but there is a sequence of guess in particular, this particular sequence which does lead you to No. But the promise is on the global structure of the machine.

Globally it has the property that there is some computation path which leads to Yes. So, what is the definition of non determinism? I mean, if even if you go back to first time when we saw non-determinism. So, non-deterministic machine is one where from one configuration you can go to multiple configurations. And the thing is that if at the end, so finally at the end of your computation you can be at many different configurations at the same time. And if at least one of those configurations contains an accepting state then you accept.

There can be many configurations which contain a rejecting state at the end of your computation. But you do not bother about them.

(Refer Slide Time: 43:21)

So, basically you have a machine. So, suppose you have a machine M which is given a string x. So, the question is how do you answer yes. You answered yes if this machine at the end of its computation, it can be at many different configurations. Because since it is non deterministic, it has this property that it branches away. So, at the end of its computation it can be, so let us assume that M is a halting machine. M always halts some x.

So, therefore when it holds it can be at many different configurations at the same time. It is the case in these algorithms. So, here we are treating our accepting state to be those states where we are at the vertex v. For example, here so, I am saying that suppose if i want to so let us maybe I can explain this a little bit more. So, suppose I am given a graph G and I want to test if there is a path from s to t. So, let us look at what the non-deterministic algorithm exactly is.

So, what I will do is from s at any vertex, so if I am at any vertex p, I look at all the edges that are going out of p and for all those edges, so suppose there are many edges q, there are many vertices q that are reachable from p in one step. For all those vertices I will go to a different configuration of the machine which is trying to decide if t is reachable from s and then I set q to be my current vertex and I keep on doing this.

Now finally when do I accept if at the end of n steps if I am at a configuration which contains the vertex t as its current vertex that is when I go ahead and accept. So, I define my accepting state to be that particular state where it is reading t as its current vertex. So there can be many things, there can be many different paths from s to t, there can be paths which lead to other vertices and there can be many ways in which that can happen.

So there are many ways in which I can land up at different vertices in this graph. But the point is that if there is a path from s to t there is at least one path which will land me up at t. So that is the thing. So the way to think about non-determinism is that I mean, you have to share the idea I mean you have to I mean you cannot think in terms of a deterministic machine. You do not have power to decide to look at all these computation paths.

So locally you can only look at one computation path and there can be computation paths which reject a particular input x although it belongs to the language. But we say that the machine correctly decides a language L if for all instances x that belong to the language. There is at least one final configuration of the machine which has an accepting state. So, when you say that you are guessing it means that if there is a path there will always be a correct sequence of guess.

So, the way I will define my non-deterministic machine is as follows. So, what is a non-deterministic machine? It means that the transition function can go to multiple configurations. So, I will define my transition function as, so if I am at a vertex v and it has many vertices that I mean if it has many outgoing edges from it. I mean the machine simultaneously will go to all these vertices.

So, my transition function will take the machine simultaneously to all these possible vertices. So, when I am saying, I am guessing a path it means that I am simultaneously trying to look at all possible paths. So, if you have any other questions, I think we can take them offline and we will complete the proof next time.