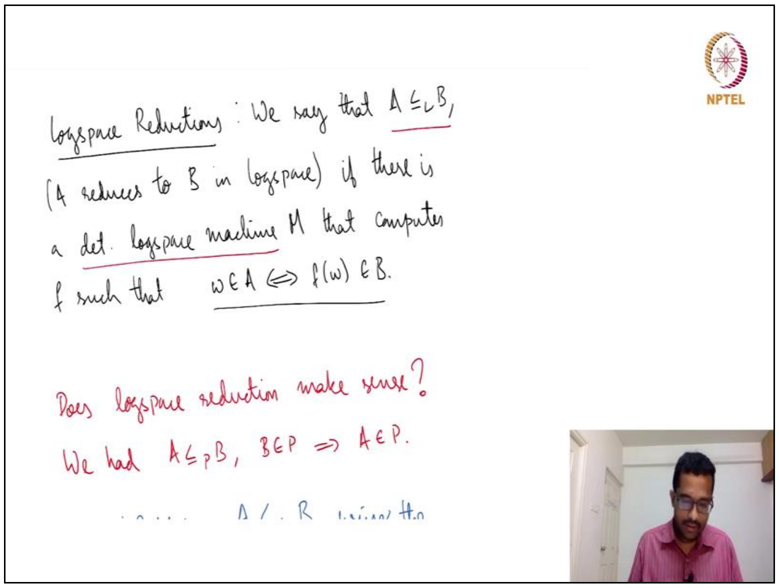
Computational Complexity

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

Lecture -15
NL-Completeness

(Refer Slide Time: 00:15)



Logspace Reductions : We say that $A \leq_L B$, (A reduces to B in logspace) if there is a det. logspace machine M that computes f such that $w \in A \Leftrightarrow f(w) \in B$.

Does logspace reduction make sense?

We had $A \leq_P B$, $B \in P \Rightarrow A \in P$.

Hello and welcome to lecture 15 of the course com computational complexity. So, in the last lecture we saw space bounded complexity classes like D space fn and N space fn, L, NL and so on. In this lecture we will see a specific a notion of completeness much like NP completeness called NL completeness. So, even though the definition is going to be similar the way these NL completeness behaves or the way this space bounded completeness behaves is a bit different.

So, the details we will see maybe in the next few lectures but in this lecture we will define and see some basics. So, to have a notion of completeness we should have a notion a reduction a property deduction with us and we cannot use. So, when we reduce let us say when we when you reduce a language A to a language B and we are talking about we are trying to compare like which class these are in etcetera.

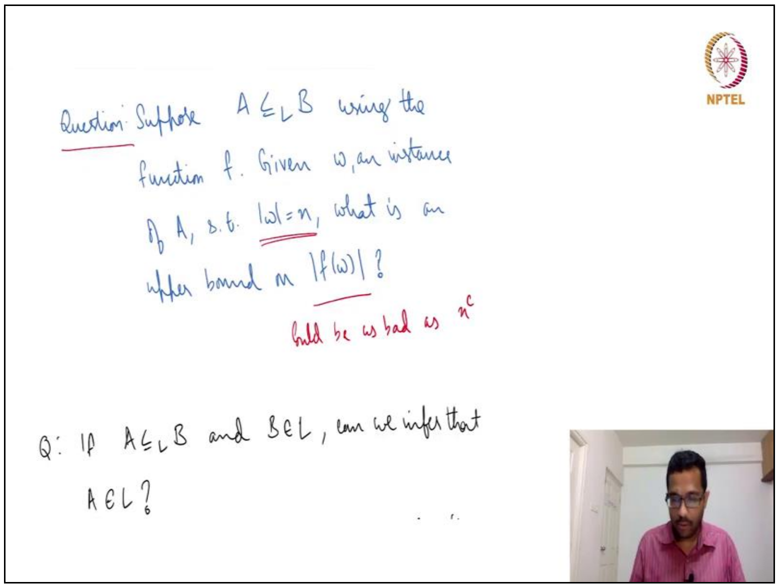
The resources available for the reduction must be strictly lesser than the classes. So, suppose we are doing NL completeness. So, we will be dealing with languages or problems in NL as we already saw in the previous lecture NL is contained in P. So, if we allow polynomial time reductions then we could take a language and decide it in the polynomial time because all these languages are in NL.

So, we need a more limited notion of reduction which is not polynomial time reduction. And since we are studying space bounded complexity classes we could we may as well use a space bounded reduction as well the reduction that we will use is log space reduction. So, what does it mean? So, A is reducible to B in log space denoted by A less than or equal to B with a subscript L as against a subscript of P.

In other words A reduces to B in log space if there is a deterministic logarithmic space machine M given x or given w in it computes f of w it computes a function f such that w is in A if and only if f of w is in B. So, everything is very similar to polynomial time reductions the only difference here is that this is the reduction this is the difference. In polynomial time reductions we the machine which computed was restricted to polynomial time.

Here the machine that computing the reduction that is computing the direction is restricted to deterministic log space and this is the w is in A if and only if f of w is in B is exactly like what we had in polynomial time reductions or even in the theory of computation course mapping reductions. So, now let us try to see whether this notion of reduction makes sense so we had this neat thing in polynomial time reductions.

(Refer Slide Time: 03:43)



Question: Suppose $A \leq_L B$ using the function f. Given w, an instance of A, s.t. $|w| = n$, what is an upper bound on $|f(w)|$?

Could be as bad as n^c

Q: If $A \leq_L B$ and $B \in L$, can we infer that $A \in L$?

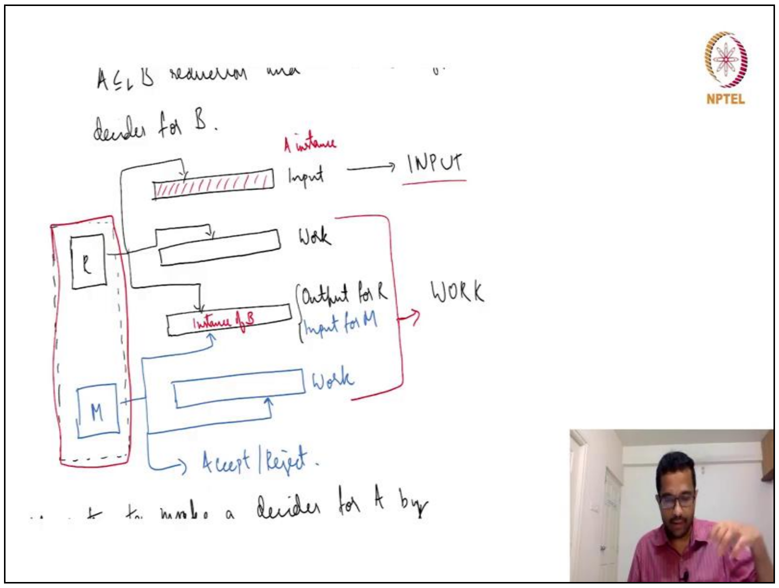
When A is in reducible to B in polynomial time and B is itself in polynomial time then we could combine the reduction with the decider for B to infer that A is also in polynomial time. So, can we do the same thing suppose A is reducible to B in logarithmic space then if we have a decider for B can we get a decider for A. So let us try to understand. Suppose a reducible to B in logarithmic space using a function f all now given let us say w is an instance of a suppose length of w is n what is the upper bound on the length of f of w?

So, how much space does this reduction take the reduction takes order log n space because it is a logarithmic boundary detection how much time can it take? It can take a lot of time because as we saw in the previous lecture D space of fn just to show the previous lecture notes D space of fn is contained in D time of 2 power order fn. So, anything that is possible in D space of fn requires the time of 2 power order fn because there could be that many configurations 2 power order fn configurations.

So, here the number of configurations for this machine could be 2 power order log n which is polynomial. So, it could take up to polynomial time to compute the reduction. So, in polynomial time f of w is written down in polynomial time. So, recall a log space bounded machine we are only charging for the work tape and if f of w is being computed this computation the result of this computation will be written down in the output tape.

So, the space usage in the output tape is not going to be accounted for. So, f of w could potentially every time step you could be writing one symbol over there. So, this could be as bad as f of length of f of w could be as bad as or as long as n power or n power constant or polynomial space or polynomial space. And so, now A reduced to B now the instance of the length of the instance of reduced instance is polynomial space.

(Refer Slide Time: 06:35)



Now the key question again like we said here over here a reducible to B in polynomial time B in polynomial time implies A in polynomial time. So, can we say the same thing for log space reductions A reduces to B in log space suppose B itself is in log space is A also in log space. So, let us try to set up the same kind of framework how we decided how we saw that A reducible to B.

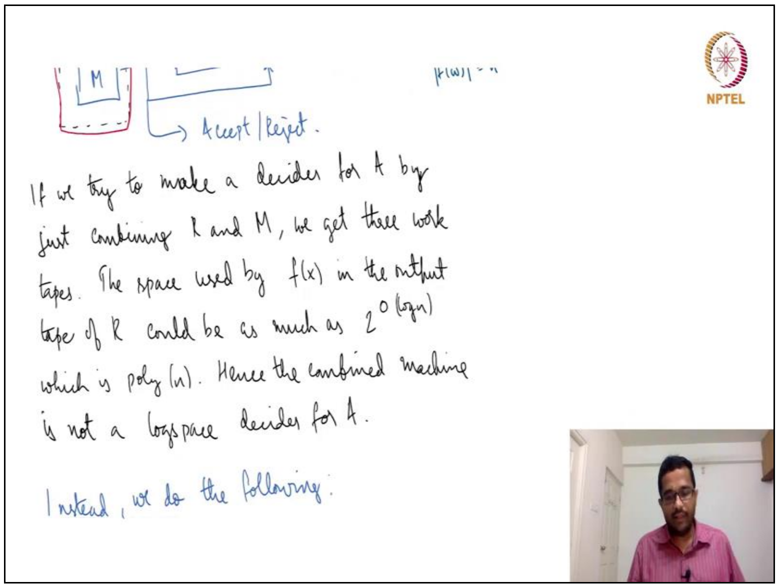
How we saw it in polynomial time can we do the same for polynomial logarithmic space. So, let R be the reduction machine that takes the instance of A as input . So, R is a reduction machine. So, this takes A instance is input it computes some work and it writes a instance of B as output instance of B. And let M be the decider for B so, N, M takes the whatever the instance of B that is given by R and then it also has a work tape.

So, the M is shown as blue R is shown as black and then it decides accept or reject. So, now if we combine the reduction machine and sorry if you combine the reduction machine and the decider for B as one machine. This input tape of R will be the sole input tape all for the combined machine this will be the sole input. And the work tape for R and the work tape for M and even the output type for R which is the input type for M will have all of this will together form the work tape for this combined R and M machine.

Because as far as if you see the 2 machines together this work tape these 2 are work types which are read right and this output for R and input for M is being written to by R and read by m. So, this is actually effectively a read write machine once you combine this. So, the work tape contains all of these 3 things and as we already saw suppose the A instance is of length n. So, suppose we already saw that if length of w is n then the length of f of w could be for polynomial in the length of w.

So, that means the space usage just by the length of the B instance that itself will be polynomial what we are shooting for was the logarithmic in n. So, this itself is polynomial.

(Refer Slide Time: 09:51)



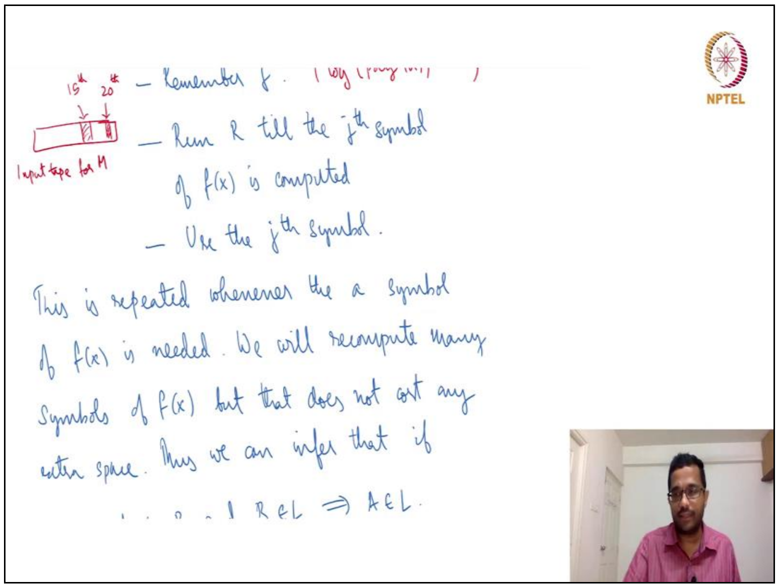
If we try to make a decider for A by just combining R and M, we get three work tapes. The space used by f(x) in the output tape of R could be as much as $2^{O(\log n)}$ which is poly(n). Hence the combined machine is not a logspace decider for A.

Instead, we do the following:

So, this does not work. So, again I have written down this in detail if you try to make it decided for a by just combining R and M the space used by the by the output type of R which the input tape for M will be polynomial in n it because it comes from this like we saw in the previous lecture. So, the combined machine, so, this is not a log space decider because it is using more space than it is using polynomial space log space means we can only use o log n space.

However this question that we asked if A reduces to B in logarithmic space and B is in logarithmic space itself can A also be decided in logarithmic space this question is still answered yes this simply that this approach was wrong . So, that is what is the correct approach?

(Refer Slide Time: 10:53)



15th 20th

Input tape for M

— Remember j. [illegible]

— Run R till the j^{th} symbol of $f(x)$ is computed

— Use the j^{th} symbol.

This is repeated whenever the a symbol of $f(x)$ is needed. We will recompute many symbols of $f(x)$ but that does not cost any extra space. Thus we can infer that if

[illegible] $R \in L \Rightarrow A \in L$.

So, this approach was wrong where we try to merge or combine R and M together this approach did not work. So, instead let us we have to take a different approach because we cannot afford. So, what happened here is that we had to write the instance of B entirely and this is something we cannot afford. So, we will now try to do something where we do not have to write this down.

So, what we will do let us say we want to decide A. So, we start. So, this is where we are we have we have w which is an a instance a instance means it is we have to decide whether w is in A or not. So, now we start the decider for B. So, this may seem strange because decider of B needs the input of B. We just started now let us say. So, now the A instance is in my nodes I have written it x. So, the not w, so, we have x which is the A instance.

And we start the deciders for B with without any fx, f of x is not there but at some point the decider for B will start looking at the input which is the in instance of B which is f of x. So, let us say when it where it wants a symbol let us say it first looks at the first bit of f of x or first symbol of f of x when that happens then we remember that it needs the first symbol of f of f of x and then we start running the reduction machine.

We start running R and once R outputs the first symbol then we can continue running M. So, in general let us say we require M requires the jth symbol f of x. So, M needs some input. So, this is this is let us say this is M input for M input tape for M. So, it may not require each one of them in sequence. So, maybe let us say it requires the 15th symbol. So, then we start running M sorry R the reduction machine till.

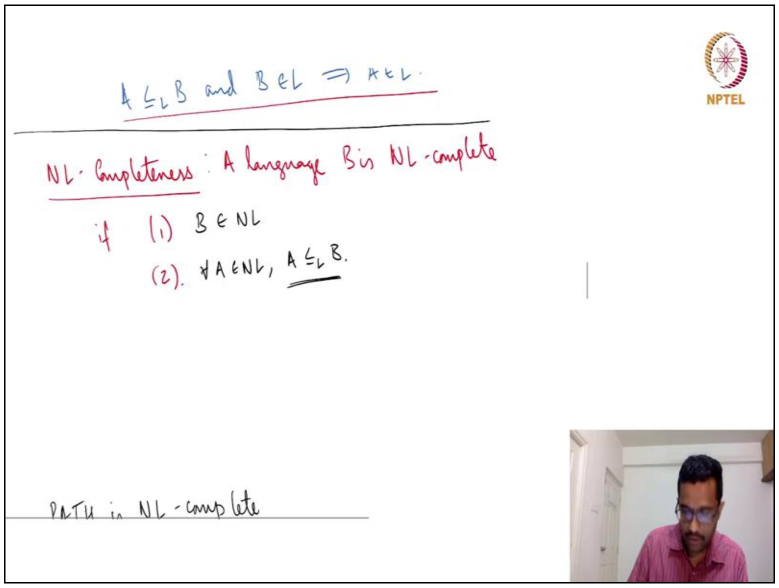
So, if we ignore everything like first bit second bit. So, on till it writes the 15 symbol, once it writes a 15 symbol we just remember the 15 symbol and M continues the computation. So, now M is purely running what is taken by M from R is only the 15 symbol whatever it required for now. So, like that whenever it requires a certain symbol let us say next it requires 20th this thing. So, again now M is sorry R is started from scratch till it computes the 20th symbol.

Now again let us say it requires seventh symbol. So, again R is started from scratch till it computes seven symbol again let us say 15 symbol is required by M again started from scratch and again computed 15 symbol even though it was already computed. Remember M cannot afford to remember the entire f of x because just remembering the entire f of x requires polynomial space and this is something we do not have all that we have is logarithmic space.

So, every time we require a certain symbol of f of x R recomputes it from the beginning or R computes it or recomputes it. So, there may be a lot of recomputing of certain symbols of of f of x but then that is because now we are not using up the entire space to write down f of x. So, f of x is never written down as a whole. We are just looking for let us say 10th is required 10 symbol f of x is required we just write down the 10 symbol 15th or 100 symbol is required we just write down the 100th symbol and that is it.

So, we are never writing down the entire f of x at any time. So, as and when they are required they are generated on the fly and then M continues running.

(Refer Slide Time: 15:21)


A ≤_L B and B ∈ L ⇒ A ∈ L.

NL-Completeness: A language B is NL-complete

if (1) B ∈ NL

(2). ∀A ∈ NL, A ≤_L B.

PATH in NL-complete

So, M just continues running let us say it requires a certain segment and that symbol is produced and given to it. So, what the problem in the in this earlier picture was that this output tape for R and input f for M used to take a lot of space. So, now we are avoiding that problem by just re-running R as and when each symbol is required. So, in some in us in a way we are sacrificing time by re-running again and again and again for space.

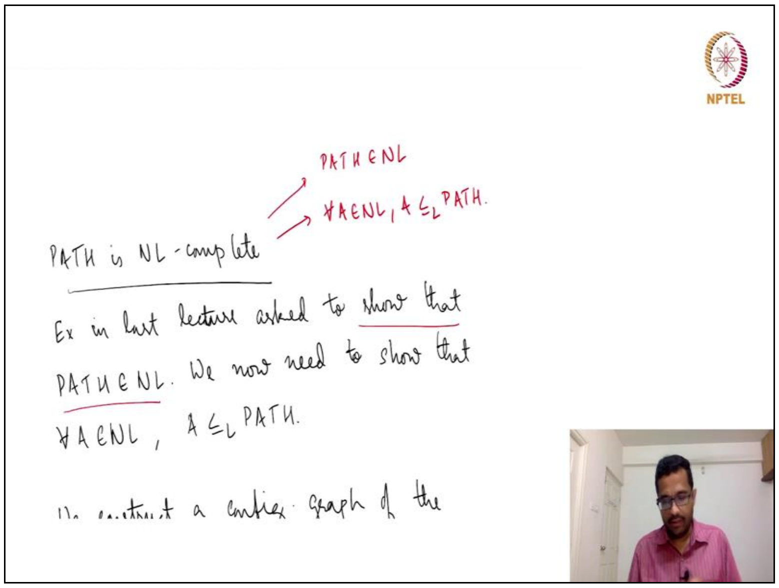
So, we are in and we are saving on the space because we are never writing down the f of x entirely in this step which is ok because in this model time is never measured what is measured is a space. So, now M can complete the computation and M can decide whether f of x is in B or not. And by the properties of the reduction we have that f of x is in B if and only if x is in A. Hence we are able to decide A using the decider for B and the reduction from A to B.

So, we can infer that A reduces to B and B is in L implies that A is in L. So, the whole setup here is log space because M requires log space and the reduction machine also requires log space but we never write down the output of the reduction machine as a whole. We just write down what we just pass on whatever symbols are needed at that point of time on the fly. We end up doing a lot of recomputation but that is because we are saving on the space.

So, that is why this reduction makes sense this is what we would like to have and that property is indeed there. So, now let us define NL completeness having seen the reduction logarithmic space reduction log space reduction and the fact that this reduction is sensible. So, the NL completeness is very similar to NP completeness just that we use polynomial log space reduction instead of polynomial time reduction.

So, a language B is an complete if first B is in NL and 2 for all A in L, A reduces to B in logarithmic space not log not polynomial time logarithmic space. This is the notation for the reduction is less than or equal to with a L subscript to note that to denote that it is logarithmic space bounded reduction.

(Refer Slide Time: 18:00)



PATH is NL-complete

$\nearrow$ PATH $\in$ NL

$\nearrow$ $\forall A \in NL, A \leq_L$ PATH.

Ex in last lecture asked to show that PATH $\in$ NL. We now need to show that $\forall A \in NL$, $A \leq_L$ PATH.

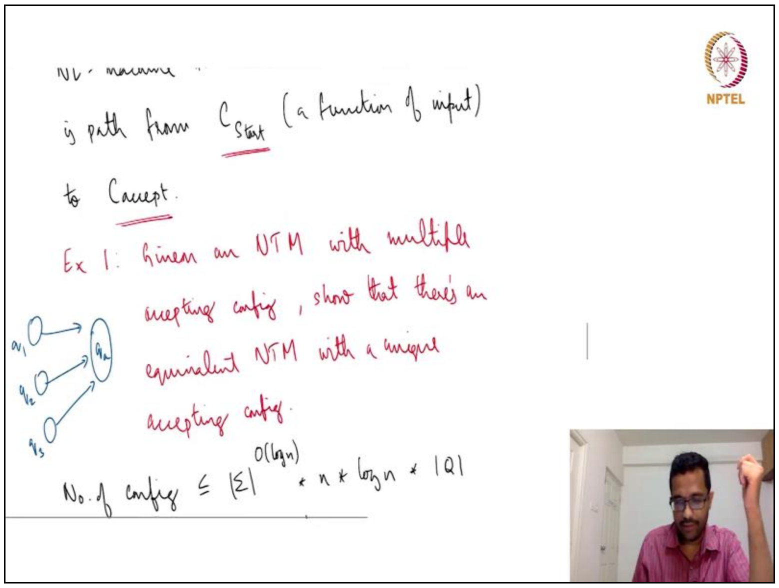
We construct a config. graph of the

So, this is the definition of angle completeness. So, one language that is NL complete. So, just like we had SAT that was the first language that was NP complete here we will our corresponding language will be PATH. So, we already saw PATH in the previous lecture and in fact we even asked I even mentioned to show that PATH is in NL . So, to show that path is in NL complete we need to do 2 things one is that path is in NL and for all a in NL a reduces to path in logarithmic space.

So, path in NL I already mentioned in the previous lecture but just I will just give a very brief high level thing we cannot run BFS or DFS but however we have the non-determinism. So, to decide whether s is or t is reachable from s one way to do it is to just guess one path just guess a sequence of vertices let us say s V 1, V 2 something t. Now you just check whether this path takes you to t is it a valid path. If there is a valid path that takes you from a that takes you from s to t then you will guess it and that is enough.

If there is no valid path whatever you guess it will lead to reject. So, that is a high level idea. So, the details you can work out. Now we need to show that for all A in NL A reduces to path which is the main part of the reduction.

(Refer Slide Time: 20:08)



NL · machine

is path from $\underline{C_{start}}$ (a function of input)

to $\underline{C_{accept}}$.

Ex 1: Given an NTM with multiple accepting config, show that there's an equivalent NTM with a unique accepting config.

$$\begin{tikzcd} q_1 \arrow[dr] & \\ q_2 \arrow[r] & q_a \\ q_3 \arrow[ur] & \end{tikzcd}$$

No. of config $\leq |\Sigma|^{O(\log n)} * n * \log n * |Q|$

So, if you have paid attention in the previous lecture you may you may guess what is going to come. We can construct a configuration graph for A. So, the A in NL we can construct a configuration graph for A. And in this configuration graph we can check whether there is a path from the starting configuration to the accepting configuration. So, C start is what we used to denote the starting configuration and C accept is the accepting configuration.

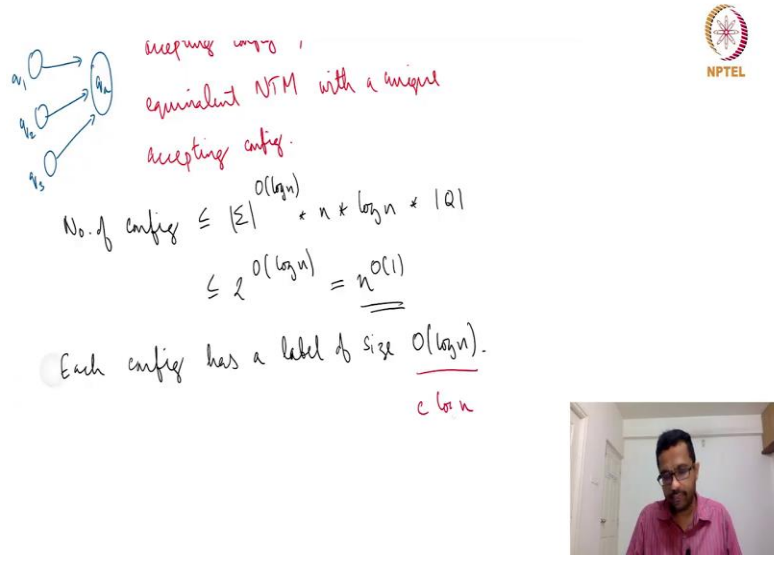
So, I am lying slightly here see accept is not a unique accepting configuration because you could be in different you could have different configurations. So, you could have different tape contents head positions and so, on. So, there is no unique accepting configurations but that is what I say here in the next exercise that you can think. If there is a non-deterministic turing machine with multiple accepting configuration show that you can you can get an equivalent non-deterministic turing machine with a unique accepting configuration.

So, it is very simple I will just very briefly highlight the idea if there are many multiple accepting configurations what you could do is you could map them. So, this is all this is a q 1, q 2, q 3 are all accepting configurations. And now I want to have a unique accepting configuration how can we do that one way is to sorry. So, the point is here is that how do we have the why do we have the different accepting configurations because because the tape contents are different and the head positions are different.

So we could program the turing machine to erase the tape content and move the head all the way back to the left most position. So, if we do that there will be only and the acceptor reject should happen only after that. Again this is a bit of turing machine stuff and which you can verify easily. So, once you do that there will be unique accepting configuration. So, now in this modified setup all we need to do is to check whether there is a path from the starting configuration to the accepting configuration.

The starting configuration will depend will be dependent on the input. So, if for a different input will have a different configuration obviously because the tape contents will be different type content at the start is contains the input.

(Refer Slide Time: 23:02)



$$\begin{tikzcd} q_1 \arrow[dr] & \\ q_2 \arrow[r] & q_a \\ q_3 \arrow[ur] & \end{tikzcd}$$

accepting config.

equivalent NTM with a unique accepting config.

$$\text{No. of config} \leq |\Sigma|^{O(\log n)} * n * \log n * |Q|$$

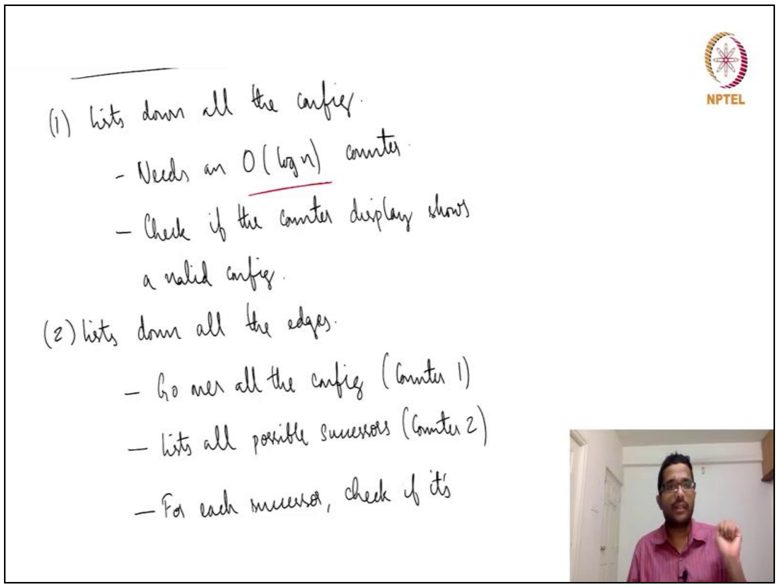
$$\leq 2^{O(\log n)} = \underline{\underline{n^{O(1)}}}$$

Each config has a label of size $\underset{c \log n}{\underline{O(\log n)}}$.

So, now we have this configuration graph and we just need to check whether there is a path from this vertex to that vertex which is exactly the problem that path is. Path is just given a graph G is there a path from s to t but maybe I should just mention here G, s, t; G is a directed graph and has a directed path from s to t. So, that is that is kind of the idea for the reduction. So, one point here is that we already saw this in the previous lectures just repeating the number of configurations that the machine has that the machine for NL machine for A has is 2 power constant times log n which is polynomial in the input length n power order 1.

This is something that we did already. So, which means to write down to represent a polynomial like if to represent a configuration we can have a label of length order login because it is just. So, to represent something we need logarithmically many bits. So, log of 2 power order login is simply order log n. So each configuration has a name of length order login. So, some con maybe c log n or something for some constant.

(Refer Slide Time: 25:02)



(1) Lists down all the config.
- Needs an $\underline{O(\log n)}$ counter.
- Check if the counter display shows a valid config.

(2) Lists down all the edges.
- Go over all the config (Counter 1)
- Lists all possible successors (Counter 2)
- For each successor, check if it's

This is going to be important. So, now we have shown that we have shown that given a machine A in or given A language a in NL you take the decider non NL decider for it and this gives you the problem of the equivalent path problem. So, you have the configuration graph is a graph starting vertex is a starting configuration s is the starting configuration t is the accepting configuration. So, the equivalence is clear but how is the reduction performed.

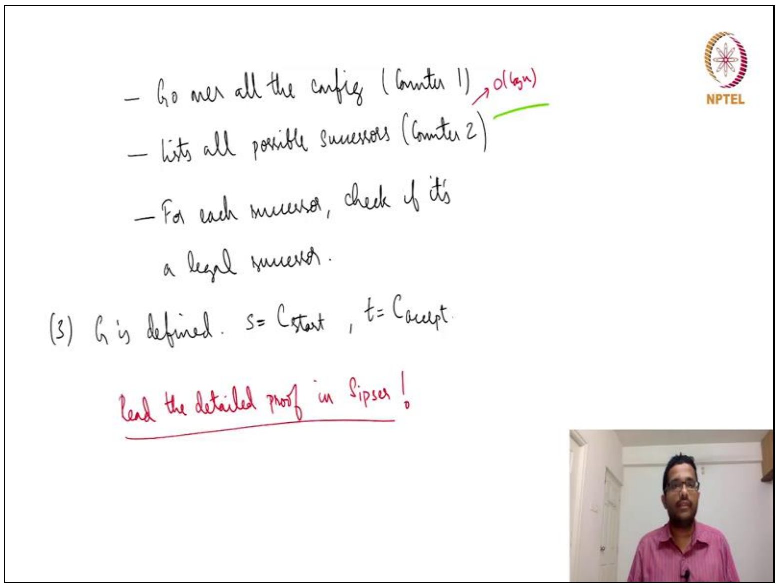
So, recall that this reduction also must be performed in log space. So, we have to be careful in doing this because we cannot we cannot use more space but then we are dealing with a configuration graph that is of size polynomial it has polynomially many vertices. But it is ok to write down the answer in the work tape but sorry not in the work type the answer in the output tape but in the work type we have we are constrained for space.

So, what do we do we one is that we list down all the configurations. So, we are what we are going to do is write down the configuration graph followed by s and t. So, s and t are easy to write down in the output table. So, how do we write down the configuration graph we first discover all the vertices followed by all the edges. So, what we do is we list down all the configurations. So, this could be done by a counter. So, you could just have a counter and whether it is a valid configuration this needs to be checked.

And as I already said the length of representing a certain configuration is order log n. So, the counter can be of order log n length. And at each point of the counter we check whether it is a valid configuration because you could have strings that do not make sense you could have multiple states. So, a configuration has only one state. So, you could have no state. So, these are all illegal strings that are not correspond to the that do not correspond to a configuration.

If it is a legal configuration you write it down. Now, that after these processes all the configurations would have been written down.

(Refer Slide Time: 27:38)



- Go over all the config (Counter 1) $\nearrow O(\log n)$
- Lists all possible successors (Counter 2)
- For each successor, check if it's a legal successor.

(3) G is defined. $s = C_{start}$, $t = C_{accept}$.

Read the detailed proof in Sipser!

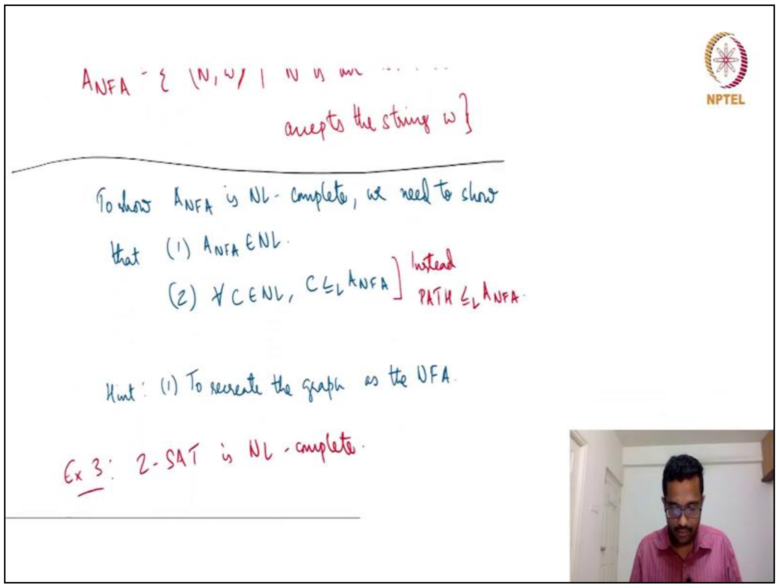
Now let us come to the edges in the configuration graph. So what you do is once again you go over all the configurations you one by one you go over all the configurations. So, that is counter one again this requires order login space. Now for each valid configuration generated by counter one we generate all the possible successors. So, again it is 2 nested loops everything running both the nested loops running over all the possible configurations.

So when one is fixed at a configuration 2 is again checking for all the possible configurations we check whether it is a valid configuration and then we check whether this is successor of the entry in the counter 1. So, if it is a valid successor of the entry in counter 1 you listed down as a possible edge of the outgoing edge of the out neighbours of the configuration counter one and so, you get an adjacency list of sorts.

So, this 3 counters o log encounter here here and o log and counter here and o log encounter here and maybe a bit of extra space to check whether the configuration is successful successor etcetera. So, all that we need is this much for the work tape. So, the work tape requires order log n space so, and it is deterministic. So, the reduction happens in order login space and we have a equivalent reduction.

So, any given any language in NL we could we could convert it to an instance of path using a log space reduction. You can see start and s c start and t c accept this is easy to note. This is also there in Sipser you can have a look they have a more paragraph writing style. So, it may be a bit different but it will also help you to see different styles of writing.

(Refer Slide Time: 29:50)



$A_{NFA} = \{ \langle N, w \rangle \mid N$ is an … … … accepts the string $w \}$

To show A_{NFA} is NL-complete, we need to show that (1) $A_{NFA} \in NL$.

(2) $\forall C \in NL, C \leq_L A_{NFA}$] instead $PATH \leq_L A_{NFA}$.

Hint: (1) To recreate the graph as the NFA.

Ex 3: 2-SAT is NL-complete.

So, again this is an instance is an example of a case where the reduction is in log space but the output size is polynomial. So, now just one exercise which is which you can try out show that A NFA is NL complete, so, A NFA is a language that you have you may have seen in theory of computation NFA stands for non-deterministic finite automaton. Given a description of a non-deterministic finite automaton N and a string w does N accept w that is a question.

And it turns out that this language is also NL complete. So, I will just give you one hint or maybe 2 hints sorry. So, the hint is to recreate the configuration graph has the NFA. So, again one point here now you can draw analogues of whatever we set for NP completeness. So, to show it that a language is angle complete, so, to show that A NFA is NL complete it is enough. So, maybe I will just say that again sorry it is.

So, what I want to say here is that to show A NFA is NL complete, we need to show that one NFA is in NL this is not that difficult and 2 for all B sorry for all let us say B or maybe c in NL, c reduces to A NFA. So, instead of this what we will do is since we already have an NL complete language we will do which is path we will just do path reduces to A NFA. This is exactly the kind of thing that we did to show that let us say clique is an NP complete.

So, now that we have in a complete language we can reduce that linear complete language to whatever languages we desire. So, we just need to show path reduces to A NFA. So, step one is going to be easy step 2 is the main thing. So, there we recreate the configuration graph or we recreate not the configuration graph we recreate the graph there is no configuration. We recreate the graph as the NFA.

So, the NFA can have the like basically the transitions will be defined by the graph and there are 2 things one point again I will just give you one hint there are 2 ways to do it as far as I know. One is by a unary language one is by an empty language and you can you could do it either way you can think about it. And another language that I will just mention that is NL complete 2 SAT is NL complete.

So, 2 SAT is when where the it is an hand of clauses and each clause is a or of 2 literals. So, it turns out this is also incomplete you can think about this. Just to summarize the lecture we defined the log space reductions we saw why it makes sense we defined NL completeness we saw that path is NL complete. So, what does it mean to say that path is NL complete, so, just maybe a bit more.

So, given a language a and given a string x is an a right we can build an instance of path say G, s, t such that or given a string x rather not x is in A, x is in A if and only if G, s, t is in path. So, the idea is G is a configuration graph of the NL decider for A, s is a starting configuration and t is the unique accepting configuration. So, x is accepted by the machine if and only if there is a path from the starting configuration to the accepting configuration.

So, that is exactly what we check in path and we also verified that this reduction can be performed in a logarithmic space and then we conclude with a couple of exercises that is all, thank you.