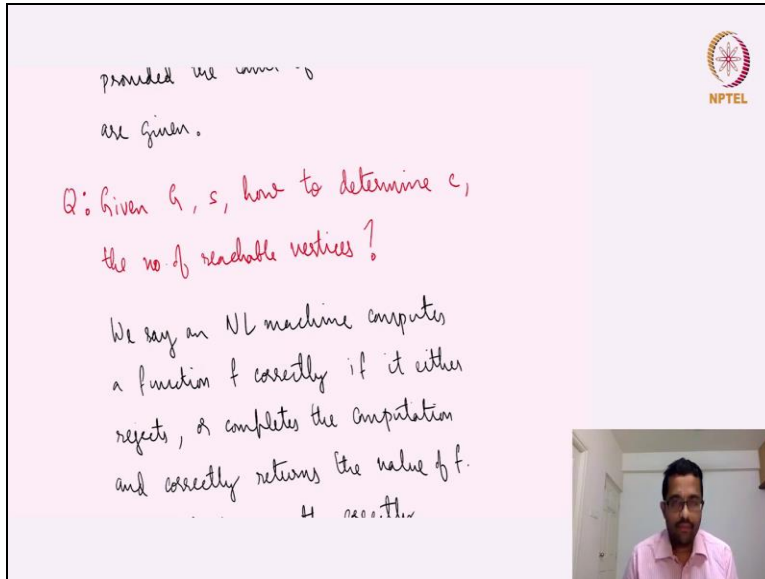


Computational Complexity
Prof. Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Lecture -18
NL co-NL_ Part 2

(Refer Slide Time: 00:15)



The slide contains handwritten text in black and red ink. At the top right is the NPTEL logo. The text reads: "provided the number of vertices are given." followed by a red question: "Q: Given G, s , how to determine c , the no. of reachable vertices?". Below this, it says: "We say an NL machine computes a function f correctly if it either rejects, or completes the computation and correctly returns the value of f ." A small video inset in the bottom right shows the professor speaking.

Hello and welcome to lecture 18 of the course computational complexity. In the previous lecture we were seeing the proof of the Immerman and Szelepcsini theorem that was that NL is equal to co-NL. So, we saw how given the count of c ; so, we wanted to show that path complement is in NL that is given a graph G and s and t we should be able to verify or we should be able to show using an NL machine that there is no path from s to t .

So, we saw how to do this given the count of reachable vertices from s . So, given the count of reachable vertices from s let us say 10 or 20 or whatever we could guess 20 vertices reachable from s none of which being t and we can verify that these 20 vertices are indeed reachable from s . So, now we have to see how to get this count using the NL machine. So, the count of reachable vertices from s . So, that part we did not see in the previous lecture. So, let us see that.

(Refer Slide Time: 01:25)

$\downarrow \downarrow \downarrow \downarrow$
 $R \ R \ R \ 100$ a function f always
 rejects, or completes the computation
 and correctly returns the value of f .
 And at least one path correctly
 computes f .

$\downarrow \downarrow \downarrow \downarrow$
 $R \ 100 \ R \ 100$ ✓

$\downarrow \downarrow \downarrow \downarrow$
 $R \ R \ R \ R$

To compute c , we recursively compute
 $c_i = |A_i|$ for $i=0, 1, 2, \dots, |V|$.
 where $A_i =$ set of vertices reachable in
 i steps or less.

So, first step we need to see what does it mean. So, now how does what does it mean to say an NL machine non-deterministic machine computes a function. So, here we are trying to see how a non-deterministic turing machine computes the number of reachable vertices. So, we say that an NL machine or non-deterministic machine correctly computes a function if on each computation path it either rejects the computation path or correctly estimates the function.

So, let us say the function value is let us say 100. So, let us say if the machine has a 4 computation paths from root to all of them. So, it could reject every path and one computation path has to come at least one computation path has to output the correct value. So, it is not possible it is not to have. So, this is ok. So, even this is ok, like you could have something like this R 100, R 100 this is however it you cannot have this that I am drawing on the right side you cannot have R 100 R let us say 200 this is not ok because the correct value is 100.

So, in no computation path could you have the wrong answer. You can even this also does not is not useful as you may guess if all the paths rejected it is also not useful. So, it should have at least one path which goes to the which outputs the correct function value it could have more than one path that outputs the correct value but everything should be reject or the correct value. This is the meaning of a not this is how a non-deterministic machine should compute a function.

Each path should either be a reject or estimate the correct function value plus at least one path should give us the correct function value. If all of them are rejected it is of no use.

(Refer Slide Time: 03:43)

$c_i = |A_i|$ for $i = 0, 1, \dots$
 where $A_i =$ set of vertices reachable in i steps or less.
 A_0, A_1, A_2, \dots
 $A_0 = \{s\}$, $A_1 = \{s, \text{outneighbors of } s\}$
 We will calculate c_{i+1} from c_i successively till we get $c_{|V|} = c$.
 For each $v \in V$, the NTM guesses and verifies if $v \in A_{i+1}$. For each ... non ... to ... to the no elements

The slide includes a diagram showing a central node 's' with arrows pointing to nodes 'a', 'b', and 'c'. Node 'a' is labeled 'A1', and 's' is labeled 'A0'. The NPTEL logo is in the top right corner. A video inset in the bottom right shows a man in a pink shirt speaking.

So, now to compute c . So, this is the definition of what we are trying to attain. So, we want an optimistic log space bounded machine that has at least one computation path that leads us to the number of reachable vertices from s . So, how does how do we compute the number of reachable vertices from s . So, what we will do is to have this kind of step by step process. So, we will define a sequence of sets called A_0, A_1, A_2 and so, on.

A_0 is nothing but the number of vertices or the set of vertices reachable from s in 0 steps A_1 is a set of vertices reachable from s in one step A_2 is a set of words as reachable from s in 2 or smaller than 2 steps and so, on. So, A_0 is nothing but the set s itself right because if it if you are not allowed any steps then only the only vertex that we can reach from s is s itself. We can reach in zero steps. A_1 let us say this is s and s has 3 vertices or three neighbours then A_1 is all of this this is A_1 .

So, A_1 has 4 vertices whereas A_0 has only one vertex right and A_2 has perhaps more vertices. So, A_0, A_1 etcetera are kind of A_0 is contained in A_1, A_1 is contained in A_2 because each one we say it is the number of A_i is the number of word is a set of vertices reachable from s in i

steps or lower and corresponding to $A_0 A_1 A_2$ we could also have $C_0 C_1 C_2$ and so, on which is the size of each of these.

So, C_i is size of A_i and what we want is a total number of vertices reachable from s . So, if any vertex is reachable from s there is a there is a path of length at most V where V is the number of vertices it cannot be longer than V overall we have V vertices. So, any path has to be of length at most V in fact $V - 1$ would do the number of steps cannot be more than $V - 1$. So, A subscript or C subscript V will give us the actual number of vertices reachable from s .

So, our goal is to compute C subscript V which is the num which will be the count C . So, and again so, we do not compute C subscript V in one shot what we will do is. So, we know C_0 which is 1 and C_1 can be computed from C_0 , C_2 can be computed from C_1 , C_3 can be coupled from C_2 and so, on that is the approach that we will follow. So, let us see how using a non deterministic log space machine we can compute C_{i+1} from C_i . So, we know C_0 is 1 then we will see how to compute C_{i+1} from C_i . So, for each of the vertices V , so, now we are going to see how to compute C_{i+1} from C_i .

(Refer Slide Time: 07:12)

The slide contains handwritten text and diagrams. The text reads: "We will calculate C_{i+1} from C_i successively till we get $C_V = C$. For each $v \in V$, the NTM guesses and verifies if $v \in A_{i+1}$. For each v , the NTM reconstructs the C_i elements of A_i and check if $\exists u \in A_i$ such that $(u,v) \in \text{edge}$, or $u=v$. But A_i cannot be guessed and written down. So no matter v that is guessed to be in". There are two diagrams: one on the left showing a set C_i with elements u and v and arrows between them, and one on the right showing a set A_i inside a larger set A_{i+1} with an element v and an arrow from u to v . An NPTEL logo is in the top right corner, and a video inset of a man speaking is in the bottom right corner.

So, for each of the not vertices V the non-deterministic turing machine it verifies if it can be reachable in $i + 1$ steps or less. So if we can guess a path that is that that of $i + 1$ of length $i + 1$ or less than good we know that is reachable in $i + 1$ steps or less. But how do we verify that a vertex

is not reachable in $i + 1$ steps because maybe just we just guess the wrong path. Here again we will use the trick that we saw in the previous lecture.

So, notice that when we compute or when we attempt to compute C_{i+1} we already know C_i . So, if a certain vertex if ever let us say this is A_i and. So, let us say this is A_{i+1} . So, all the vertices let us say all the vertices that are in A_{i+1} but not in A_i there will be a neighbour or there will be a vertex let us say u is or V is in A_{i+1} but not in A_i there will be some use in A_i such that $u V$ is an edge.

Because if otherwise it like if V can be reached in $i + 1$ steps then it has to have a predecessor that can be reached in i steps. So, in order to verify that let us say some vertex w cannot be reached in $i + 1$ steps how do we verify that it cannot be these $i + 1$ steps. So, what we will do is to actually show that we have explored all the vertices in A_i . We have explored all the vertices in A_i .

Let us say we know that A_i has 100 vertices we will say that we have x we have identified all these 100 vertices in A_i and none of these vertices in A_i were found to have an edge to w . So, this edge did was not found. So, we know that there are hundred words and we found the 100 vertices none of them had an edge to w . So, that means w is not in A_{i+1} . This is how we will establish that a vertex is not in A_{i+1} .

So, for that we will have to verify all the vertices in A_i . So, this is the way we will do it. So, for each vertex each we need to either verify that it is reachable or we need to verify that it is not reachable. So, how do we verify that is not reachable we need to actually reconstruct the entire set A_i and show that none of these vertices has an edge. So, for each V we have to check that we have to check the all the elements that are in A_i .

And check whether there is a predecessor in A_i such that either the predecessor had an edge to V or the predecessor was V itself. So, we could be in A_i also.

(Refer Slide Time: 10:39)

NPTEL

and verified again


Algorithm 3 To compute c_i , given G, s

- 1: Let $c_0 = 1$.
- 2: for $i = 0$ to $|V| - 1$ do
- 3: Let $c_{i+1} = 1$.
- 4: for each node $v \neq s$ in G do
- 5: Let $d = 0$.
- 6: for each node u in G do
- 7: Nondeterministically either perform or skip the following steps.
- 8: Call the function CHECKPATH(G, s, u, i).
- 9: $d \leftarrow d + 1$.
- 10: if $(u, v) \in E(G)$ then *OR (u=v)*
- 11: $c_{i+1} \leftarrow c_{i+1} + 1$
- 12: Go to Stage 5 with the next v .
- 13: if $d \neq c_i$ then
- 14: Reject.
- 15: return c_i

Need to store i, d, u, v, c_i, c_{i+1} .

We don't store all the c_i 's. This

function returns $c = c_{i+1}$ which is used



So, the challenge here again is that we cannot guess the entire set A_i or we cannot guess the entire set and write it down somewhere because A_i could have let say $n/2$ elements and that storing that that requires order N space or even at least order N space. So, we cannot guess the entire set A_i and write it down somewhere. So, we have to kind of like we have seen before in the case of space boundary classes we have to make these guesses in a serial fashion by remembering only the little things that are necessary.

So, whatever is necessary just that is what we will remember. So, here what we will remember is only the count and we will see how to using the count we can verify that. So, this is the algorithm more formally more formally. So, let us see what the algorithm is. So, first we say C_0 is equal to 1 which is 1 vertex is reachable from s in 0 steps or less. Now we are going to see how for each i we are going to compute C_{i+1} from C_i .

So, for each what for each i this loop the for loop in step number two is saying how to verify how to compute C_{i+1} from C_i . So, we start with C_{i+1} being equal to 1 because s is always in all these A_{i+1} . So, remember A_{i+1} is a set of all words is reachable in $i+1$ steps are less and C_{i+1} is the size of A_{i+1} . So, always s is a member of A_i or A_{i+1} for all i . So, we can assume C_{i+1} and start from s start from 1. Now for each node each vertex that is not equal to s we check whether there is a path from s to u .

So, what we do is we guess a predecessor sorry we do not guess a predecessor we for each u in the graph we check whether u can be reached in i steps or less and for that we use check path which we defined in the previous lecture which is just verifying whether s to u is reachable in i steps. So, we are trying to build the set but we are not building the set and saving it somewhere we are just going one by one and verifying the count.

And at each point whenever $A u$ is found to be reachable from s and i steps we are checking whether u has V as its out neighbor or $V u$ is also. So, there is a small typo here or u equal to V . So, if one of these things happen then we know that V is also reachable in $i + 1$ steps if V is not reachable in $i + 1$ steps we actually need to make sure that we have covered all our possibilities that all the vertices that were supposed to be reachable in i steps or have actually been identified.

So, whatever I said over here we have indeed computed all the vertices in $A i$. So, that is done in step number 13 that have we checked that uh. So, we are we are keeping a count of for each vertex that was identified to be in $A i$ that running count is d and we are checking whether d was equal to $C i$. So, we already know from the previous loop that $C i$ was computed $C i$ is a count of vertices that are reachable in i steps or less.

So, we are checking whether we found d or is a running count d is actually equal to the actual count of the vertices $C i$ or the number of word is a $C i$ if it is not equal that means some things were not reached some count did not get checked. So, we reject the computation path entirely but there will be a computation path that will actually go through the identify the correct number of vertices and we can using in that computation we will correctly identify that V was not reachable or V was reachable.

So, suppose we found a vertex V to be reachable then again we start from step 5 which is that d equal to 0 which means we again start from zero for each we are again estimating each vertex in $A i$. So, every time we are for each vertex V we are re-estimating or recounting the the set $A i$. So, we are doing a lot of computation again and again redoing a lot of computation but that is the only way out because our space is very limited and we are not bounded on time.

So, we can afford to do the recomputation because we are only paying by space in this case. So, we start from. So, in this loop starting from line 2 what we are doing is that with the assumption that we have C_i using that and assuming that it is correct we are counting or we are estimating C_{i+1} . So, if we correctly identify the set A_i say this is A_i and then we know exactly what are the vertices that are reachable in one more step from A_i . So, we can correctly estimate C_{i+1} .

To check that we have correctly counted correctly accounted A_i all that we need is the. So, we do not keep track of A_i we do not save the set A_i anywhere instead we just save the count. And every time we regenerate A_i and we check whether the count is correct? If the count is correct we can be assured that this computation is correct. So, there will be a lot of parts leading to reject but that is ok because there will be some path at least one path that will lead to the correct computation path meaning the computations.

And this loop runs till $V - 1$. So, which means that at the end we will have correctly computed V , c_V which is equal to; so, c_V is equal to the actual count of reachable vertices from s if we have not rejected. If in any stage we if we return a value that will be correct and what are the things that we need to store here? We need to store i we need to store C_i we need to store C_{i+1} we need to store V we need to store u , C_0 not C_0, i, C_{i+1}, d, u, V and we also need C_i . These are the things that we require.

And notice that just to store C_0, C_1, C_2 up to C_V let us say the number of vertices is n these itself are n values. So, we cannot even store these n values because n values itself are n things we do not have in order N space. So, instead what we will do is that once we compute C_0 or once we compute C_0, C_1, C_2 and then C_0 can be forgotten and C_1 can be forgotten only we only need C_2 to compute C_3 and once we have C_3, C_2 can be forgotten and once we have C_3, C_4 C_3 can be forgotten. So, we only need the previous value C_i to compute C_{i+1} and once we have C_{i+1} we can forget C_i .

(Refer Slide Time: 18:41)

NL-complete, ...
 $\overline{\text{PATH}} \in \text{NL}$, this implies $\text{co-NL} \subseteq \text{NL}$.
 $\overline{\text{PATH}} \in \text{NL} \Rightarrow \text{PATH} \in \text{co-NL} \Rightarrow \text{NL} \subseteq \text{co-NL}$.
 Thus we get $\text{NL} = \text{co-NL}$
 What we need to show is $\overline{\text{PATH}} \in \text{NL}$.
 $\overline{\text{PATH}} = \{ \langle G, s, t \rangle \mid G \text{ has undirected } s-t \text{ path} \}$.

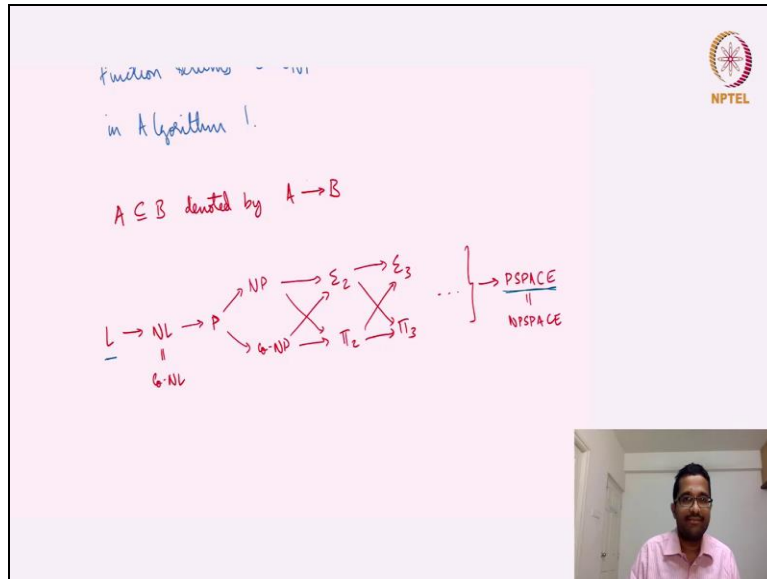
So, we do not store all the C i's together we only need the latest the previous value and the current running value and each one of these numbers are either a vertex or a count that is no more than n. So, each one of them requires only order log n space. So, finally the function returns C V if it has not if it has not rejected it will correctly compute C V. Again this algorithm, so, that is the com that is pretty much the Immerman and Szelepcsinihi proof.

This proof is not so, straightforward. So, because it it has this intricate use of non-determinism and reusing recomputing again and again in order to save space this idea is you being used again and again. So, there are two or three key idea ideas. One is computing again and again to save space and then non-deterministically guessing a sequence of vertices right and then for each one choosing whether it is in the set or not in the set or non deterministically guessing a sequence to find a path from ever s to t.

So, for each vertex in the path is guessed one by one. So, these are the tricks that are used in this proof but it is very, very instructive to understand how non-determinism works and also how space bounded computation works. So, what I am saying is that if there are parts that you cannot follow please do not feel disheartened it is only natural because this proof is a bit involved. Please you can you can read Sipser or this notes that I will share and also I had also typed up a pdf sometime back. So, I can also share that pdf um.

So, the result is that NL is equal to co-NL uh. So, we do that by it is called Immerman and Szelepcsinihi theorem we do that by showing that path complement is in NL which is to show that we should be able to decide using an NL machine that you cannot reach t from s.

(Refer Slide Time: 21:07)



So, first we saw how to show this using an NL machine given the count of what is reachable from s and in this lecture we saw how to come up with that count. So, we came up with that count by by have having these counts of the successive sets $A_0 A_1 A_2$ etcetera where A_0 is a set of words is reachable in zero steps from s, A_1 is set of course is reachable in one step from s, A_2 set of versus reaching two steps from s and so, on.

And most of the algorithm was about how to compute A_{i+1} from A_i and we need to store a bunch of variables maybe 5 or 6 variables and each of one of them being taking at most log order log n space maybe it can be erased. So, we have seen some complexity classes. So, just uh. So, we could have a short recap let us say let us say a or let us say a subset of B this is denoted by; so, I am using a right arrow to denote containment. So, let me draw the status of the complexity classes as we know it.. Now in the course, so, we know that L is contained in NL. So, I am using the right arrow to denote containment and we know that NL is contained in P. We do not know whether LN is a strict subset of L is a strict subset of NL we do not know if NL is a strict subset of B and we saw P is contained in NP.

We saw P is contained in $co-NP$ but between NP and $co-NP$ we do not know how they interrelate we saw NP is contained in Σ_2 sorry Σ_2 , $co-NP$ is contained in Π_2 , $co-NP$ is contained in Σ_2 , NP is contained in Σ_2 Π_2 Σ_3 Π_3 . So, notice that if P is in NP and NP is in Σ_2 it automatically implies that P is in Σ_2 and so, on. So, you can write Π_4 Σ_4 everything and so, on.

And we will see that all of these are contained NP space which is polynomial space D space of polynomial and we already saw in the lecture where we talked about Savitch's theorem that in the case of space complexity P space and NP space are the same. So, P space is same as this and another point is that NL in this lecture we saw that NL is equal to $co-NL$. So, this is what we know as of now and none of these arrows that have drawn here we do not know whether any of them are strict or equal.

So, L equal to NL or L is a strict subset we do not know NL equal to P or a strict subset we do not know P strict subset of NP or equal we do not know NP Σ_2 we do not know and so, on anything NP Π_2 we do not know. The only equal equality that we know is P space is NP space NL is equal to $co-NL$. The only other thing that we know is that the left endpoint P space sorry left end point is L and the right end point being P space.

These two are not the same because L uses log space and P space uses polynomial space. So, just like we had time hierarchy theorem we will also see a space hierarchy theorem in the next week's lectures. Where we will show that if there are two like if two sufficiently separated functions D space of the first function is a strict subset of D space of the second function as long as they are sufficiently separated.

So, log space is a strict subset of P space. So, we know that L is not equal to P space but then none of these individual arrows we know whether they are strict or not. This is just food for thought there is so, much that we have learned. So, far but even in the world of; even the most up-to-date state-of-the-art research we do not know so, many things about these complexity classes and that is what makes this an interesting field.

There are so, many things that are still to be known and still being studied and understood with time. And with that thought I will stop this lecture I will end this lecture 18 and thank you.