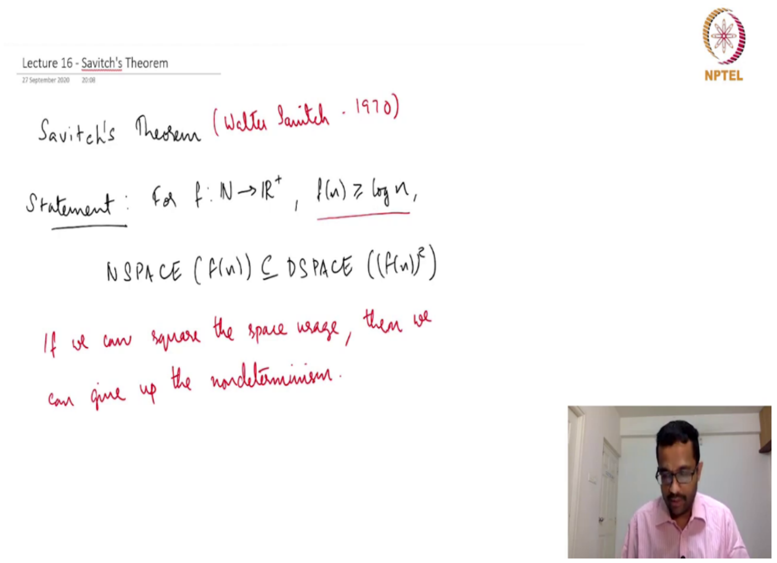
Computational Complexity
Prof. Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad

Lecture -16
Savitch's Theorem

(Refer Slide Time: 00:15)



Lecture 16 - Savitch's Theorem

27 September 2020 20:08

Savitch's Theorem (Walter Savitch - 1970)

Statement: For $f: \mathbb{N} \to \mathbb{R}^+$, $f(n) \geq \log n$,

$$NSPACE(f(n)) \subseteq DSPACE((f(n))^2)$$

If we can square the space usage, then we can give up the nondeterminism.

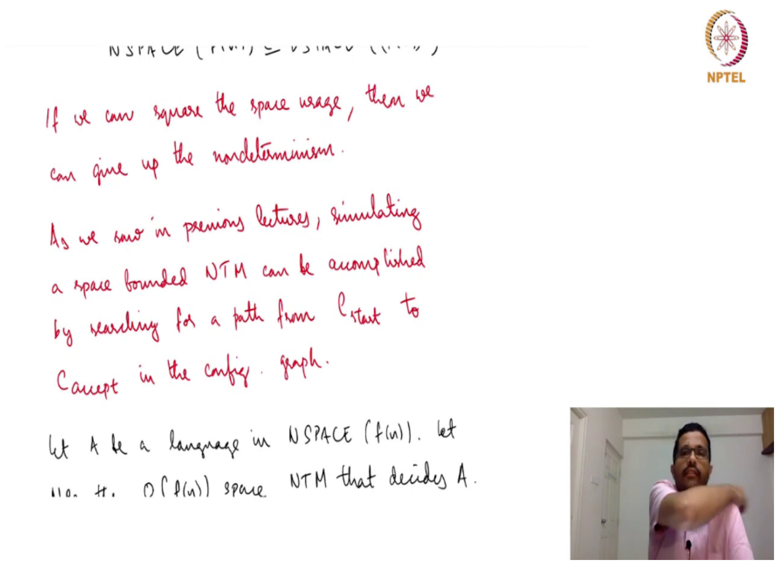
Hello and welcome to lecture 16 of the course computational complexity. In this lecture we will see Savitch's theorem. This was discovered by or proved by Walter Savitch in 1970. In this theorem we see a relation between what is achievable in non-deterministic space bound machine and what is achievable in deterministic space bounded machine. So, in very simple terms what it says is that if there is a non-deterministic space bounded machine whose space usage is $f(n)$ $f(n)$ then there is an equivalent deterministic space bounded machine whose space usage is $(f(n))^2$ $(f(n))^2$.

So, if there is an NSPACE machine which runs in n^2 n^2 space then there is an equivalent DSPACE machine which runs in at most n^4 n^4 space. So, you can go from a non-deterministic space bounded machine to a deterministic space bounded machine by just squaring the space required. So, you can give up non-determinism in exchange for squaring the amount of space that is required and like I said in one of the previous lectures, we have this requirement that $f(n)$ $f(n)$ we always assume to be at least $\log n$ $\log n$.

So, it is bit of a surprising thing because we saw that if a language is an NP such as SAT, we do not know whether it is in P. So, we know that SAT is in probably some NP non-deterministic polynomial time and there is nothing like you can square the time required and get a deterministic machine for that because square of a polynomial is still a polynomial. But in the case of space if a language can be decided in non-deterministic polynomial space $f(n)$ $f(n)$ then it can be decided in deterministic space $(f(n))^2$ $(f(n))^2$.

So, that in that sense it is a bit of a surprise. So, this is one place where we see the differences between space bounded complexity and time bounded complexity. So, just to write it informally if we can square the space usage then we can give up the non-determination. So, basically you can exchange or trade non-determinism by using extra space like squared space.

(Refer Slide Time: 03:14)



NSPACE (f(n)) ⊆ DSPACE (f²(n))

If we can square the space usage, then we can give up the nondeterminism.

As we saw in previous lectures, simulating a space bounded NTM can be accomplished by searching for a path from C_{start} to C_{accept} in the config. graph.

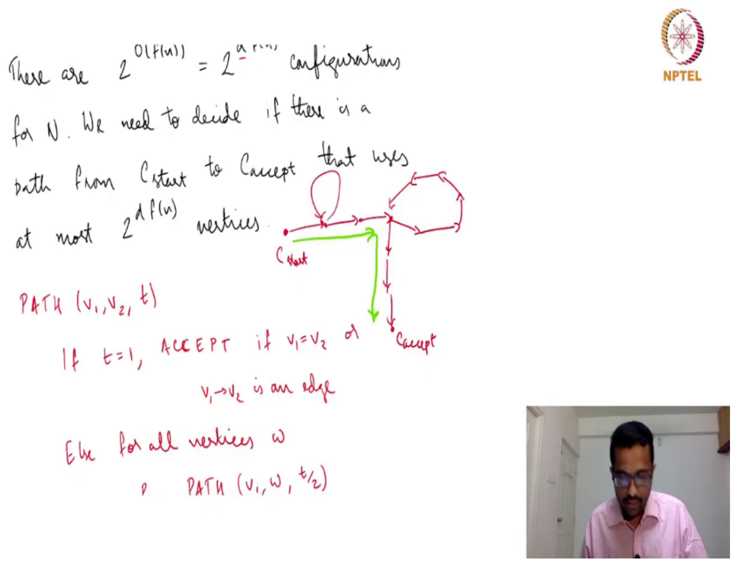
Let A be a language in NSPACE (f(n)). Let M be the O(f(n)) space NTM that decides A.

So, now let us see how this goes. So, this is very, very interesting because we are giving up non-determinism in exchange for a bit more space and we do not have anything like this in the case of time complexity. So, let us see how to prove it and the proof turns out to be surprisingly simple. So, suppose there is a non-deterministic Turing machine or suppose there is a language that is in decidable by an non-deterministic Turing machine.

And we already saw that to simulate this non-deterministic Turing machine all we need to do is check whether there is a path from the starting configuration to the accepting configuration. So, I will just write that down, as we saw in previous lectures at least 2 lectures before, we saw this simulating a space-bounded Turing machine or a non-deterministic Turing machine can be accomplished by searching for a path from c_{start} c_{start} which is notation for starting configuration to c_{accept} c_{accept} which is the accepting configuration in the configuration graph.

So, and we also saw in the previous lecture that given a non-deterministic Turing machine, we can modify it in such a way that there is a unique accepting configuration. So, in the configuration graph all we have to do is search whether there is a path from the c_{start} c_{start} to the c_{accept} c_{accept}.

(Refer Slide Time: 05:25)



There are $2^{O(f(n))} = 2^{\underline{d}\, f(n)}$ configurations for N. We need to decide if there is a path from C_{start} to C_{accept} that uses at most $2^{d f(n)}$ vertices.

```
PATH $(v_1, v_2, t)$
    If $t = 1$, ACCEPT if $v_1 = v_2$ or
                $v_1 \to v_2$ is an edge
    Else for all vertices $w$
            PATH $(v_1, w, t/2)$
```

C_{start}
C_{accept}
NPTEL

So, suppose A is a language that is in NSPACE($f(n)) f(n))$. Now that means there is a non-deterministic Turing machine N that runs in $O(f(n))$ $O(f(n))$ space and decides A. Now how many configurations can N possibly have? again this calculation we performed already in one of the previous lectures.

N can have at most $2^{O(f(n))}$ $2^{O(f(n))}$ configurations. So, instead of working with an asymptotic and O($f(n)) f(n))$, let us fix the constant involved in this O($f(n)). f(n))$. Let us call it $d.f(n)$ $d.f(n)$ where d is some constant. So, in calculations it is always it sometimes helpful to actually work with exact constants instead of the O notation.

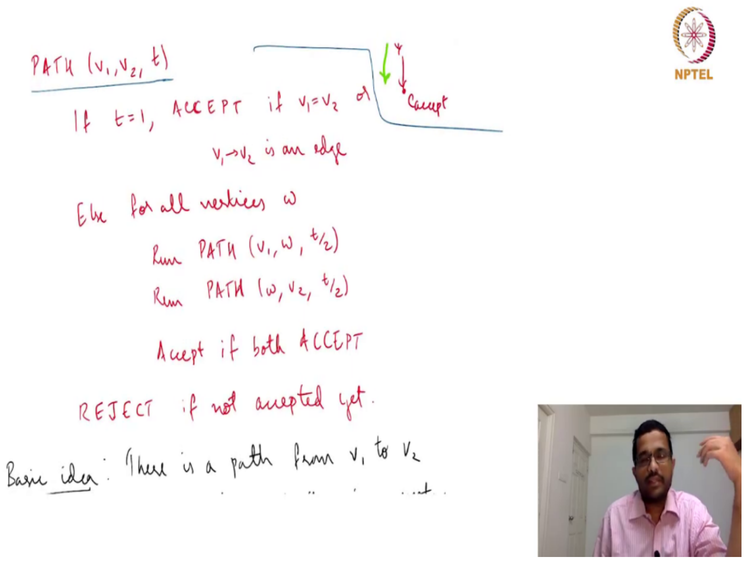
Because sometimes it is risky to work with the O notation because you do not know sometimes, it is possible to lose track of the way the order notation explodes. So, that is why to be extra careful, we use the exact constant here. So, we will just you call it $d.f(n)$ $d.f(n)$. Now we need to decide if there is a path from c_{start} c_{start} to c_{accept} c_{accept} but instead what we will do is we know that there are at most $2^{d.f(n)}$ $2^{d.f(n)}$ configurations. So, we will just check whether there is a path of this length because if there is a path of length longer than this that means that path revisits some vertex again.

So, it can happen something like this. So, let us say this is c_{start} c_{start}. Now if there is a path of longer length it means it may loop and finally it goes to c_{accept} c_{accept}. So, which means actually there is a shorter path you could go without looping from c_{start} c_{start} to c_{accept} c_{accept}.

So, whenever there is a path in this graph which has at most $2^{d.f(n)}$ $2^{d.f(n)}$ vertices, you can remove all these loops that arise. So, perhaps there are loops in multiple locations, perhaps there is a loop big loop here also, but then you can ignore all that and get a simple path of at most $2^{d.f(n)}$ $2^{d.f(n)}$ vertices.

Now again we named this problem of finding whether there is a path from a vertex s to a vertex t in a directed graph as the PATH problem.

(Refer Slide Time: 08:22)



```
PATH ($v_1, v_2, t$)
    If $t=1$, ACCEPT if $v_1 = v_2$ or
              $v_1 \to v_2$ is an edge
    Else for all vertices $w$
          Run PATH ($v_1, w, t/2$)
          Run PATH ($w, v_2, t/2$)
          Accept if both ACCEPT
    REJECT if not accepted yet.
Basic idea: There is a path from $v_1$ to $v_2$
```

t
accept
NPTEL

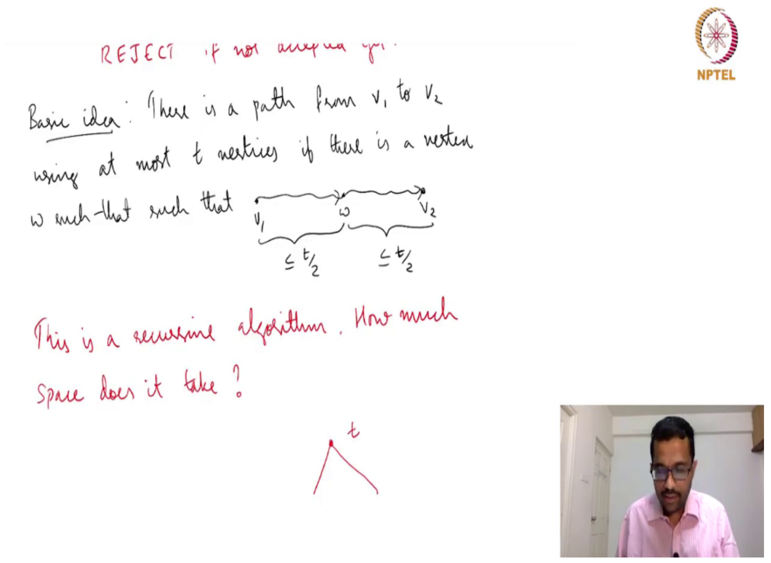
So, now we will make a function call and here this $PATH(v_1, v_2$ $PATH(v_1, v_2$, t) is a program or is a pseudo code that will tell whether there is a path from v_1 to v_2 using at most t vertices or at most t steps. This is what we are trying to do with this sub routine or pseudo code and the reason we named it like this is that we can make recursive calls.

So, I will use this recursively that is why I called it $PATH(v_1, v_2$ $PATH(v_1, v_2$, t) this is asking whether there is a path of length at most t from v_1 to v_2. So, when t is equal to 1, we are checking whether v_1 to v_2 is an edge or $v_1 = v_2$ i.e., both can also be the same vertex. So, if it is not the case what we do is we check for all the vertices w.

So, if there is a path of length at most t that means there is a middle vertex. So, let us say the middle vertex is w. Then there should be a path of length at most $t/2$ $t/2$ from v_1 one to w and there should be a path of length at most $t/2$ $t/2$ from w to v_2.

So, now what is the midpoint w we do not know. So, to have to figure that. Now what we do is we just try out all possible w's. So, remember we are in the space bounded setting and we are trying to get a deterministic space powered machine. So, we just try out all possible w's and call the subroutine $PATH(v_1, w\ PATH(v_1, w$, $t/2$ $t/2$) and $PATH(w, v_2\ PATH(w, v_2$, $t/2$ $t/2$).

(Refer Slide Time: 11:02)



REJECT if not accepted yet.

Basic idea: There is a path from v_1 to v_2 using at most t vertices if there is a vertex w such that such that $\underbrace{v_1 \to w}_{\le t/2} \underbrace{\to v_2}_{\le t/2}$

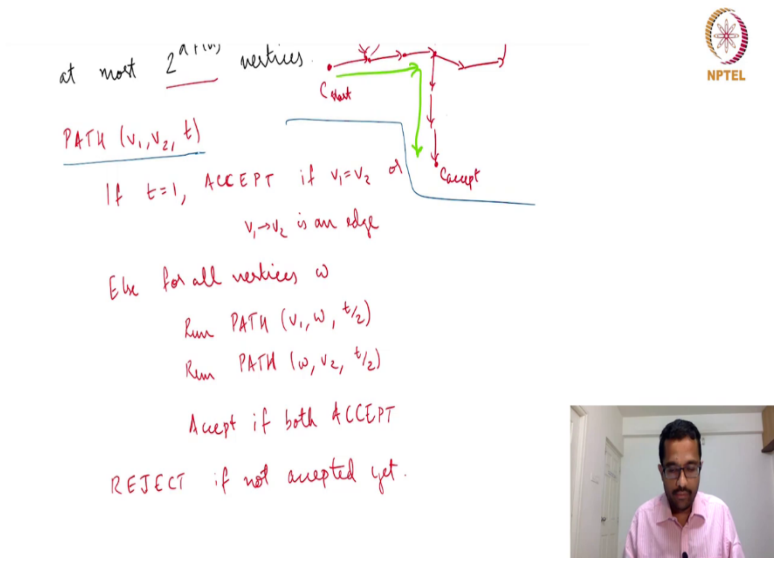
This is a recursive algorithm. How much space does it take?

And if both of these are there then you accept ok. So, notice that if there is a path from v_1 to v_2 of length at most t then it has a midpoint w and there will be a path from v_1 to w of at most $t/2$ $t/2$ and w to v_2 of length at most $t/2$ $t/2$. So, if v_1 to v_2 there is a path of length t we accept the top level call $PATH(v_1, v_2\ PATH(v_1, v_2$, t) if both of these accept and we reject if for even one of them, for no w, this leads to an accept i.e., there is no midpoint vertex w. It means in reject case, there is no path.

So, this is a simple pseudo code using a very simple recursive technique and somehow this is all enough to show the Savitch's theorem. So, you may be surprised how such a simple idea is leading us to such a powerful theorem statement.

So, again this is just what I said there is a path from v_1 to v_2 using at most t vertices if there is a midpoint vertex w such that v_1 to w is $t/2$ $t/2$ vertices and w to v_2 is at most $t/2$ $t/2$ steps. So, now all that remains is to show the analysis of this i.e, to show that this recursive function call uses at most $O\left((f(n))^2\right)$ $O\left((f(n))^2\right)$ space. So, what is t here? t is used for checking whether there is a path from c_{start} to c_{accept} that uses t steps and t is $2^{d\,f(n)}$ $2^{d\,f(n)}$ which is the number of vertices in the configuration graph. The configuration graph has $2^{d\,f(n)}$ $2^{d\,f(n)}$ vertices.

(Refer Slide Time: 13:44)



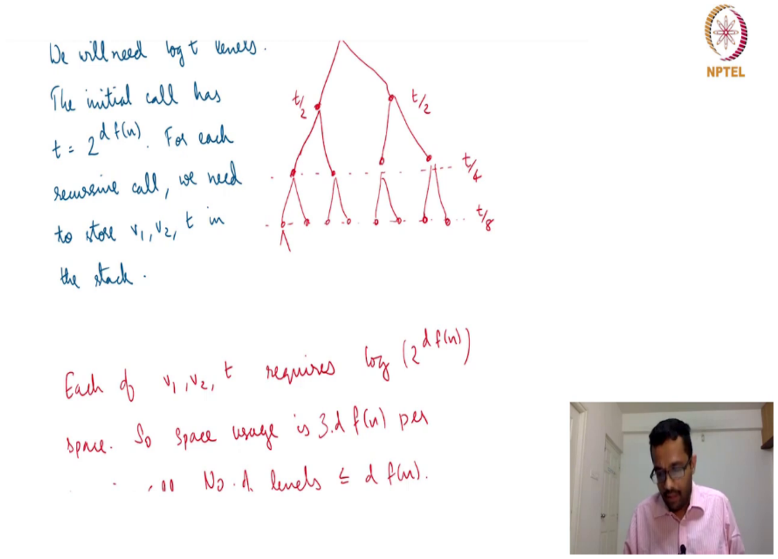
at most $2^{d f(n)}$ vertices

```
PATH (v1, v2, t)
    If t=1, ACCEPT if v1=v2 or
            v1→v2 is an edge
    Else for all vertices w
        Run PATH (v1, w, t/2)
        Run PATH (w, v2, t/2)
        Accept if both ACCEPT
    REJECT if not accepted yet.
```

So, how much space does it take? So, let us see what happens when the recursive call is made all. So, at any level of recursive call you may recall from what you learned in algorithms or data structures that. When a recursive call is made the program needs to store some information in the stack. That is how recursive calls are made and when the recursive call returns the answer and then it needs to resume the computation from wherever it left off.

So, it needs to store some state information and then need to resume the computation. So what is the state information that is required here. So, at the top level we are making the call $PATH(v_1, v_2\ PATH(v_1, v_2$, t). This $v_1, v_2\ v_1, v_2$ and t will be kept in stack. Again when we call $PATH(v_1, w\ PATH(v_1, w$, $^t/_2\ ^t/_2$) we store $v_1, w\ v_1, w$, $^t/_2\ ^t/_2$ in stack and when we call $PATH(w, v_2\ PATH(w, v_2$, $^t/_2\ ^t/_2$), we store $w, v_2\ w, v_2$, $^t/_2\ ^t/_2$ in stack. All the nested calls we make, those many are stored in stack. So, the idea is to measure how much of the stack or how much space is you being used for all these things.

(Refer Slide Time: 15:03)



We will need log t levels. The initial call has $t = 2^{d\,f(n)}$. For each recursive call, we need to store v_1, v_2, t in the stack.

Each of v_1, v_2, t requires $\log(2^{d\,f(n)})$ space. So space usage is $3.d\,f(n)$ per , no. of levels $\leq d\,f(n)$.

So, let us try to understand this. So, let us see how many levels of these recursive calls are required. So, initially at the top level we need the number of steps that are required i.e., we need to check from v_1 to v_2 there is a path of length at most t and in the second call we are checking whether v_1 to w there is a path of length at most $^t/_2\ ^t/_2$ and over here again from w to v_2 we check whether there is a path of length at most $^t/_2\ ^t/_2$ and so, on.

And then each of these calls make further calls and we are where we are checking whether there are parts of length $^t/_4\ ^t/_4$ and further calls are of length $^t/_8\ ^t/_8$. So, now how big is this tree? well at each level the distance between the starting vertex and the destination vertex is becoming half. So, how long does it take for t to become a constant or t to become 1? The answer is simply $\log_2 t$ $\log_2 t$.

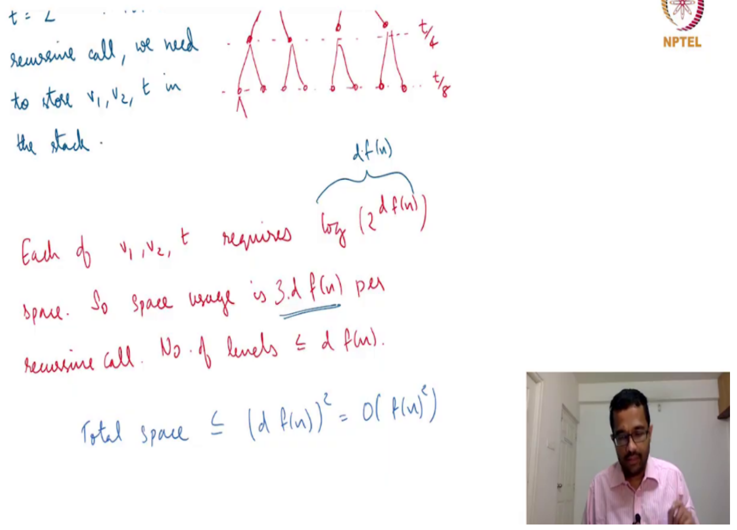
So, we will need $\log_2 t$ $\log_2 t$ levels. I am not sure if I said this said it before but I will say it. Now whenever we say log without specifying the base in this course i.e., if the base is not explicitly mentioned you can assume that the base is 2 and this will be most likely the case for most of the computer science courses. We will need $\log t$ $\log t$ levels where it is $\log_2 t$ $\log_2 t$. The initial call has t = $2^{d\, f(n)}$ $2^{d\, f(n)}$.

So after $\log t$ $\log t$ levels what happens is that then we are just checking whether there is an edge of length 1 or something which is the base case that we already saw in the pseudo code. And for each recursive call we need to store this, what I said. So, we need to store some information in the stack. So, where we called from. So, that is v_1, v_2 and t in the stack. So, when I say v_1, v_2, t, I mean from whichever state of the function we are calling from.

So, at maybe in some interim call it will be some specific vertex starting vertex somewhere specific ending vertex and instead of this $2^{d\, f(n)}$ $2^{d\, f(n)}$ it may be some other value in the stack. And each one of this v_1, v_2, t; v_1 is a is a name of a vertex and there are $2^{d\, f(n)}$ $2^{d\, f(n)}$ vertices v_2 is also the name of a vertex and t is also $2^{d\, f(n)}$ $2^{d\, f(n)}$. And to store this v_1 or v_2 or t we need a bit string of length log of that.

So, to store a number of size 100 we need a space of log 100 which is about 7. So, if there are a 1000 vertices we need a bit string of length around 10. So, if $2^{d\, f(n)}$ $2^{d\, f(n)}$ vertices we need log $2^{d\, f(n)}$ $2^{d\, f(n)}$ = $d\, f(n)$ $d\, f(n)$ because log with base 2 and 2 power cancel and for this is what is required for each of the v_1 , v_2 and t. And hence since this is what is required for each of v_1, v_2 and t, the total space required is $3.\, d.\, f(n)$ $3.\, d.\, f(n)$ per recursive call.

(Refer Slide Time: 19:06)



$t = 2$ ...

recursive call, we need to store v_1, v_2, t in the stack.

Each of v_1, v_2, t requires $\log(\overbrace{2^{d f(n)}}^{d f(n)})$ space. So space usage is $\underline{3.d\, f(n)}$ per recursive call. No. of levels $\le d\, f(n)$.

Total space $\le (d\, f(n))^2 = O(f(n)^2)$

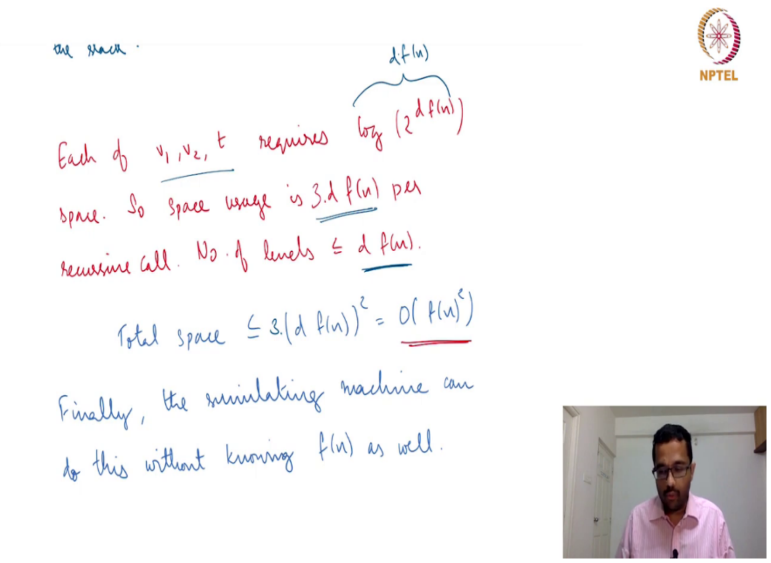
For each recursive call we need to save this so total we need $3.d.f(n)$ $3.d.f(n)$ and what is the number of levels? Again the number of levels is what i said earlier it is $\log t$ $\log t$ it is a number of levels again that is again another $\log t$ $\log t$ is also $d.f(n)$ $d.f(n)$. So, number of levels is also at most $d.f(n)$ $d.f(n)$. So, the total space required is just the number of levels multiplied by the information stored at each level.

So, you have a stack and at each level you need to store this much information and multiplied by the space required to store each information. So, space for one level of information multiplied by number of levels it is $3.d.f(n)$ $3.d.f(n)$ which is simply $d.(f(n))^2$ $d.(f(n))^2$ maybe there is a 3 here but it is still $O((f(n))^2)$ $O((f(n))^2)$. Because d d is a constant and 3 is also a constant.

So, $3d(f(n))^2$ $3d(f(n))^2$ is still $O((f(n))^2)$ $O((f(n))^2)$. This is the total space required and that is it this is what we set out to prove. So, we have a deterministic algorithm. So, notice that it is a deterministic algorithm here else for all vertices w is simply there is no non-determinism here we are just checking one by one by one w and there is no non-determinism guessing going on here and we have a machine that uses a deterministic machine that uses $O\left((f(n))^2\right)$ $O\left((f(n))^2\right)$ space.

So, we started with a non-deterministic machine that uses $O(f(n))$ $O(f(n))$ and we are achieving the same computation by a deterministic machine that uses $O\left((f(n))^2\right)$ $O\left((f(n))^2\right)$ space. And that is it. So, that is the statement of the Savitch's theorem - non-deterministic space $f(n)$ $f(n)$ machine whatever it does can be accomplished by a deterministic space $(f(n))^2$ $(f(n))^2$ machine. Now just one more comment.

(Refer Slide Time: 21:25)



the stack.

Each of $\underline{v_1, v_2, t}$ requires $\log(\overbrace{2^{d f(n)}}^{d \cdot f(n)})$ space. So space usage is $\underline{\underline{3.d\, f(n)}}$ per recursive call. No. of levels $\leq \underline{d\, f(n)}$.

Total space $\leq 3.(d\, f(n))^2 = \underline{\underline{O(f(n)^2)}}$

Finally, the simulating machine can do this without knowing $f(n)$ as well.

So, now to actually do this i.e., to actually devote space and make this call etc. we need to know $f(n)$ $f(n)$ beforehand because otherwise this computation cannot happen because the initial call has to be for $t = 2^{df(n)}$ $t = 2^{df(n)}$. So, we need to know $f(n)$ $f(n)$ beforehand. What if we do not know $f(n)$ $f(n)$ beforehand? We can still do the same thing by simply trying out $f(n) = 1, f(n) = 2, f(n) = 3$ $f(n) = 1, f(n) = 2, f(n) = 3$ and so on.

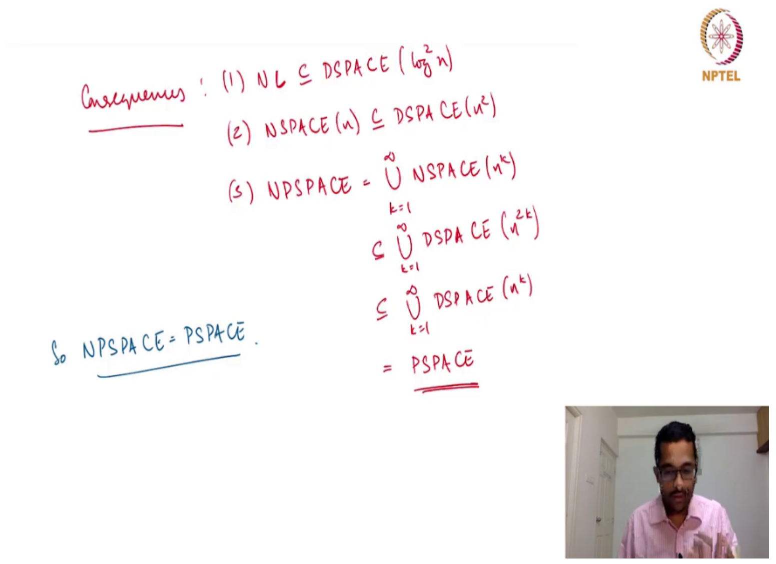
Just that we will eventually be successful in computation in computing the in simulating the non-domestic machine then when we are at $(f(n))^2$ $(f(n))^2$ which is a square of the space used by the non-deterministic Turing machine. So, what I am saying is that if suppose we do not know how much space the non-deterministic Turing machine uses and still we have to accomplish the simulation we can still do that by simply trying out $f(n) = 1,2,3$ $f(n) = 1,2,3$ and so, on.

So, what we can do is that the simulating machine can try out $f(n) = 1,2,3$ $f(n) = 1,2,3$ and so on till it finds a decision. So, what it should try to do is accept reachable when it is limited to $f(n) = 1$ $f(n) = 1$. If the accept is reachable then you accept, if the accept is not reachable, now are we trying to move to configurations of bigger length. If you are not able to move to any bigger length configuration then that means even by trying out bigger space we are not going to reach anywhere.

So, we accept if we reach the accepting state we reject if we are not able to reach any bigger configurations. If accept is reachable, we accept. if no configuration of bigger length is reachable, we reject else we move to next value of $f(n)$ $f(n)$. So, just we continue the computation. This is what we can do. So again this is a minor point that I said at the end. But the fact is that we can continue this we can do this simulation even without knowing $f(n)$ $f(n)$ that is how nice this construction is.

So, once again what this shows is that whatever is accomplished in non-deterministic space $f(n)$ $f(n)$ machine can be accomplished in deterministic $SPACE\left((f(n))^2\right)$ $SPACE\left((f(n))^2\right)$ machine. So, maybe you'll just write down one or 2 consequences and then stop.

(Refer Slide Time: 25:20)



Consequences : (1) $NL \subseteq DSPACE(\log^2 n)$

(2) $NSPACE(n) \subseteq DSPACE(n^2)$

(3) $NPSPACE = \bigcup_{k=1}^{\infty} NSPACE(n^k)$

$\subseteq \bigcup_{k=1}^{\infty} DSPACE(n^{2k})$

$\subseteq \bigcup_{k=1}^{\infty} DSPACE(n^k)$

$= PSPACE$

So $NPSPACE = PSPACE$.

So, consequences maybe. Let us say $NL \subseteq DPACE(log^2 n)$ $NL \subseteq DPACE(log^2 n)$ where NL is non-deterministic log space and another result - $NSPACE(n) \subseteq DPACE(n^2)$ $NSPACE(n) \subseteq DPACE(n^2)$,

$$NSPACE = \cup_{k=1}^{\infty} NSPACE(n^k) \subseteq \cup_{k=1}^{\infty} DSPACE(n^{2k}) = PSPACE$$

$$NSPACE = \cup_{k=1}^{\infty} NSPACE(n^k) \subseteq \cup_{k=1}^{\infty} DSPACE(n^{2k}) = PSPACE$$.

So, what we have shown is that non-deterministic polynomial space is contained in polynomial space deterministic polynomial space. So, when I say contained in the other direction containment is obvious. Deterministic polynomial space is always a sub class of non-deterministic polynomial space. Now we have shown the reverse direction as well. So, $NPSPACE$ $NPSPACE$ is actually equal to PSPACE.

So, this is also something surprising to us because in time complexity, the biggest problem is that whether P equal to NP or P a strict sub-class of NP but in case of space bound complexity we have NP space and P space both are the same. So, today we saw the Savitch's theorem which said that non-deterministic space $f(n)$ $f(n)$ can be simulated by deterministic space $(f(n))^2$

$\left(f(n)\right)^2$. So the proof using a recursive computation and some consequences and that is all, thank you.