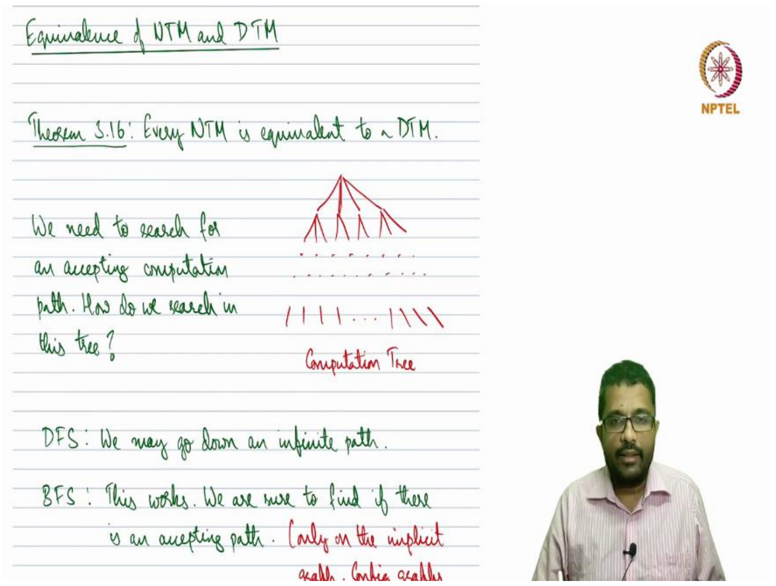
Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Equivalence of Deterministic and Nondeterministic TM

(Refer Slide Time: 00:15)



Equivalence of NTM and DTM

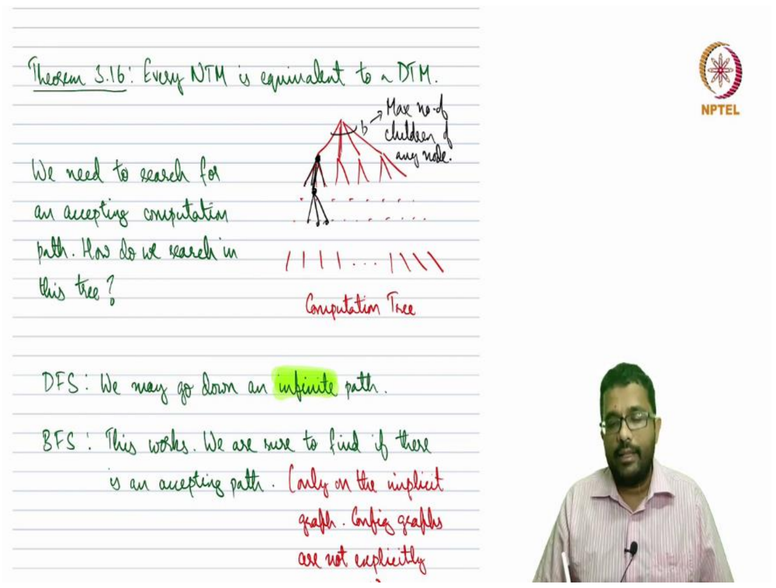
Theorem 3.16: Every NTM is equivalent to a DTM.

We need to search for an accepting computation path. How do we search in this tree?

Computation Tree

DFS: We may go down an infinite path.

BFS: This works. We are sure to find if there is an accepting path. (only on the implicit graph. Config graphs



Theorem 3.16: Every NTM is equivalent to a DTM.

b → Max no. of children of any node.

We need to search for an accepting computation path. How do we search in this tree?

Computation Tree

DFS: We may go down an infinite path.

BFS: This works. We are sure to find if there is an accepting path. (only on the implicit graph. Config graphs are not explicitly

Hello and welcome to Lecture 31 of the course Theory of Computation. So far, we have seen Turing machines and we saw the multi tape version and then we saw the non-deterministic variant of the Turing machine. In this lecture, we will start by seeing the equivalence of the non-deterministic and the deterministic Turing machine model. Clearly, the non-deterministic Turing machine has access to seemingly more powers.

So, anything a deterministic Turing machine can do a non-deterministic Turing machine can also do in the sense that a deterministic Turing machine can only decide and make the next step once we fix the state and the symbol that is being read. Whereas the non-deterministic Turing machine may have zero options, one option or more options. So, since one option is also a possibility, this means that a deterministic Turing machine itself can be viewed as a non-deterministic Turing machine.

So, clearly non-deterministic Turing machines are at least as powerful as deterministic Turing machine. So, what we are going to show here is that anything a non-deterministic Turing machine can do a deterministic Turing machine can also do. Meaning if a language can be recognized by a non-deterministic Turing machine there is a deterministic Turing machine that recognizes the same language.

So, in terms of the class of languages recognized, they are the same, non-deterministic as well as deterministic Turing machines. So, there are other things like the time it takes which may differ from non-deterministic to deterministic. But for now, we are not focusing on the resources used. All that we are caring about is whether there is an equivalent deterministic Turing machine given a non-deterministic Turing machine and this theorem says that there is always one.

So, the key thing in the non-deterministic Turing machine is that we have possibly multiple options at each stage. And what we are looking for is, is there an accepting configuration that can be reached from the starting configuration? So, there is a tree that is formed, the computation tree. And you want to know whether there is some accepting configuration that can be reached starting from the starting configuration somewhere in the stream. So, the answer is not that difficult.

So, it is a tree and if you learnt data structures or algorithms you may have seen graph traversal approaches. So, how to traverse a tree? Two standard approaches are depth first search and breadth first search. So, in depth first search, you start from the root of the tree and then go all the way deeply to the end of a path. And once you reach the bottom of the path, you retract back and then you try to go the next path and so on.

The problem with this approach in depth first search is that the first path that you try to go through or one of the paths that you try to go through may be an infinite path. So, it may be an unending path and you may just get stuck on that path. So, you will never be able to get back and try some other paths, and perhaps the accepting computation is in one of those paths.

So, this is an issue with depth first search. So, we may get stuck on an infinite path. Whereas in breadth first search, what we do is we first go level by level, so, we first try to go through all the first level, then the second level points in the tree, and then the third level points in the tree and so on. So, you go level by level, so, you cannot get stuck on an infinite path, because you are not going deep.

You are just going to the first level, then the second level, then the third level and so on. So, that is why breadth first search works. And one comment that is here. So, breadth first search works and finally there is some accepting path somewhere. So, you will encounter the accepting path, whichever happens the earliest. Once you see that accepting path you can accept. Because that is what we need to do.

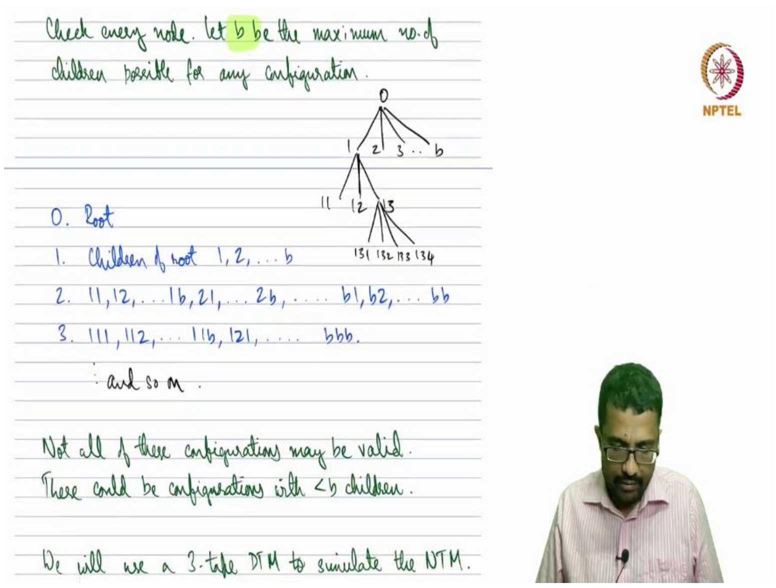
To accept the computation, you just need to find one accepting path. One point I want to mention here as we have been talking about this computation tree. It is not like we build this entire tree, because this entire tree itself is so huge, even if the time taken is T, the branching factor of this tree could be like big. It could be like, let us say it is b. Then that means that we could have b^T possible paths starting from the root node to the leaf nodes and that is a huge number.

So, we will not be constructing this computation tree explicitly. Rather what we will do is, if you are at the configuration, we will see what are the next possibilities. If you are at this configuration, then we will calculate these three successor possibilities then let us say we come to this configuration, so this one the middle one, and then again we will calculate the successor configurations there and so on.

So, we will do it on the go. We will not explicitly construct this graph and maintain the adjacency or anything like that. So, this will be implicit. This will be an implicit graph. So, let us try to see what the process is again. We have explained in a very high-level manner. So, the idea is to do breadth first search. So, starting from the root node, then the nodes at level one, then the nodes at level two, then the nodes at level three and so on.

Finally, whenever we get an accepting computation, we accept and stop the computation. So, one assumption is that, once the machine is fixed, the rules are fixed, let b be the branching factor, which is a max number of children of any node. In other words, it is the maximum number of successor configurations that follows any given configuration. And this is entirely dependent on the Turing machine alone and is independent of the input that is given.

(Refer Slide Time: 07:43)

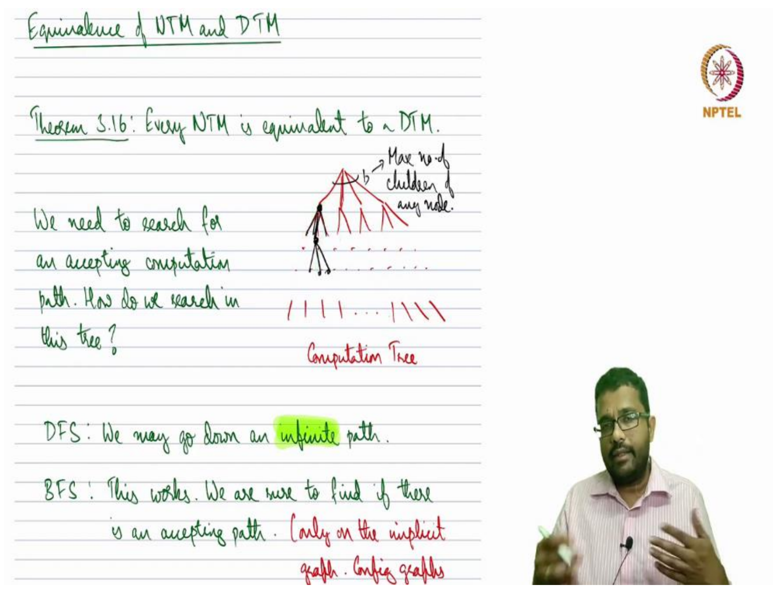


Check every node. Let b be the maximum no. of children possible for any configuration.

0. Root
1. Children of root 1, 2, ... b
2. 11, 12, ... 1b, 21, ... 2b, ... b1, b2, ... bb
3. 111, 112, ... 11b, 121, ... bbb.
⋮ and so on.

Not all of these configurations may be valid.
There could be configurations with < b children.

We will use a 3-tape DTM to simulate the NTM.



Equivalence of NTM and DTM

Theorem 3.16: Every NTM is equivalent to a DTM.

We need to search for an accepting computation path. How do we search in this tree?

DFS: We may go down an infinite path.

BFS: This works. We are sure to find if there is an accepting path. (only on the implicit graph. Config graphs

So, let b be the maximum number of children possible for any configuration or the successors possible for any configuration. So, at 0-th level, we go to the root, then the children of the root, that is, I am labeling it 1, 2, up to b. So, the reason is this. So there is a root. And let us say 1, 2, 3, and let us say b, and let us say the node 1 has three children, I will call them 11, 12, 13. Let us say 13 has four children, I will call them 131, 132, 133, and 134 and so on, this is how we label it.

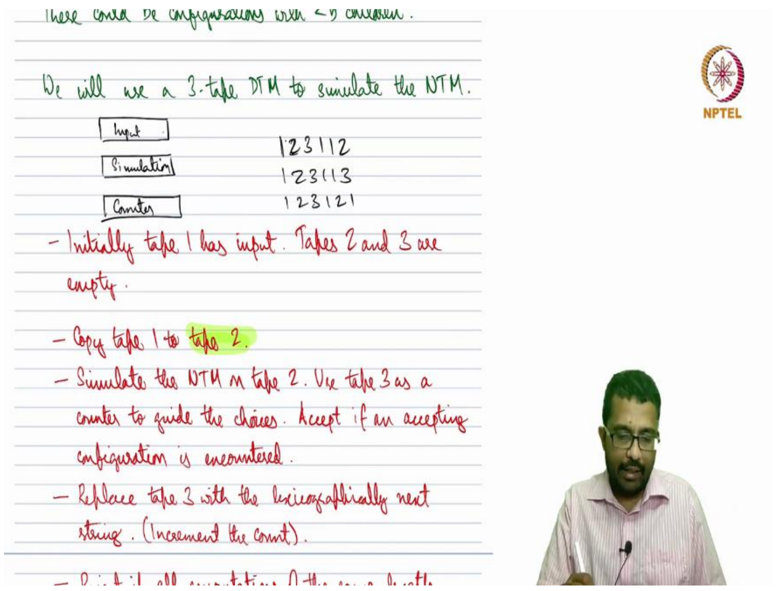
So, even though I said b is the maximum number of children possible, it is just the maximum number. So, it is possible that some nodes may not have b children, they may not have any children also. So, here in the way I have drawn it, 1 has only three children 11, 12, and 13, and 13, has only four children 131, up to 134. So, just to avoid clutter, I am not drawing the children of 11, or 2, or 3 or b or whatever, but this is just some part of the tree that I have drawn just to convey what the labeling is.

So, children have the root, 1, 2, 3, up to b. At the second level, we have 11, 12, up to 1b, 21, 22, up to 2b and so on. Finally, b1, b2, up to bb. Third level, we have 111, 112 and so on up to bbb, and so on. And we may not have all these configurations. So, for instance, here I do not have let us say b is 4, we do not have 14 because, so what we do is we try out paths in this manner. So, we go to 1, 2, 3, up to b then 11, 12, 13 then 131, 132.

So, all the levels, all the nodes, all the configurations at a certain level we check in the respective step. And whenever we get an accepting configuration, we accept. So, we will explain how to, so, the goal again. Let me just state the goal. We have to find in this tree, is there any path that leads to an accept. But I do not have access to non-determinism and I have to do it deterministically.

So, when I am doing it non-deterministically I will automatically or magically be able to find the path. But when I am doing deterministically, I will have to actually go through the paths in some way and find out and that is why we said breadth first search.

(Refer Slide Time: 10:47)

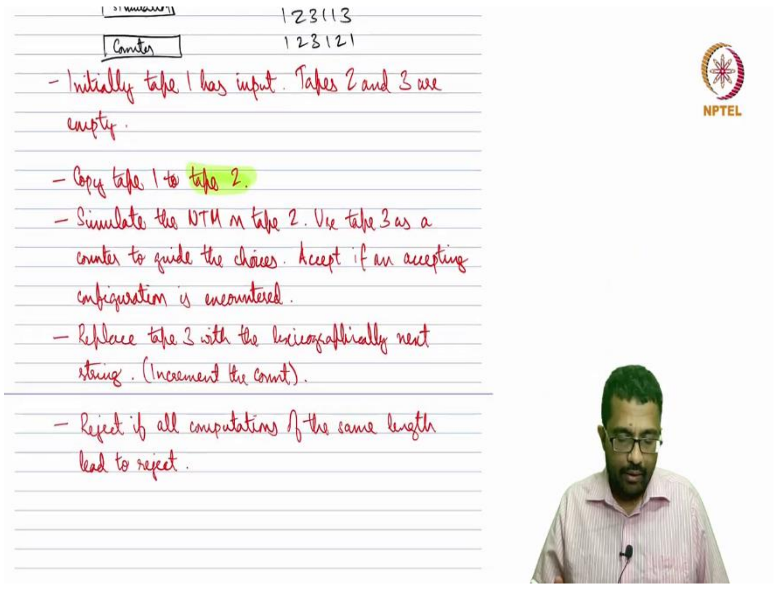


There could be configurations with < b children.

We will use a 3-tape DTM to simulate the NTM.

Input	123112
Simulation	123113
Counter	123121

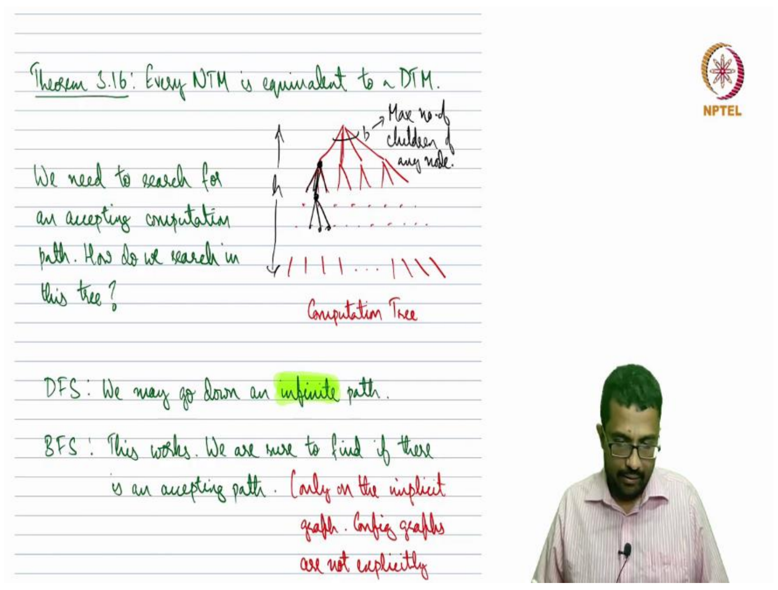
- Initially tape 1 has input. Tapes 2 and 3 are empty.
- Copy tape 1 to tape 2.
- Simulate the NTM on tape 2. Use tape 3 as a counter to guide the choices. Accept if an accepting configuration is encountered.
- Replace tape 3 with the lexicographically next string. (Increment the count).
- Reject if all computations of the same length



Simulation	123113
Counter	123121

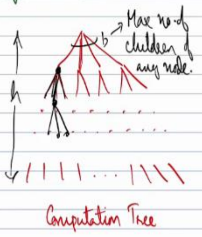
- Initially tape 1 has input. Tapes 2 and 3 are empty.
- Copy tape 1 to tape 2.
- Simulate the NTM on tape 2. Use tape 3 as a counter to guide the choices. Accept if an accepting configuration is encountered.
- Replace tape 3 with the lexicographically next string. (Increment the count).

- Reject if all computations of the same length lead to reject.



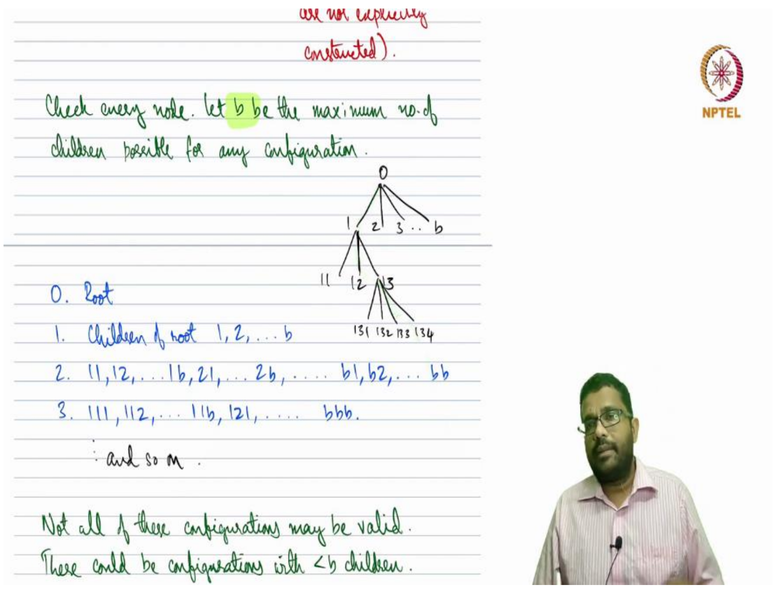
Theorem 3.16: Every NTM is equivalent to a DTM.

We need to search for an accepting computation path. How do we search in this tree?



DFS: We may go down an infinite path.

BFS: This works. We are sure to find if there is an accepting path. (only on the implicit graph. Config graphs are not explicitly



are not explicitly constructed).

Check every node. Let b be the maximum no. of children possible for any configuration.

0. Root

1. Children of root 1, 2, ... b

2. 11, 12, ... 1b, 21, ... 2b, ... b1, b2, ... bb

3. 111, 112, ... 11b, 121, ... bbb.

⋮ and so on.

Not all of these configurations may be valid.
There could be configurations with < b children.

So, what we do is we use a 3-tape DTM, a 3-tape deterministic Turing machine where, tape 1 has input. So, we want to decide whether a particular input is accepted by a non-deterministic Turing machine. Tape 1 has the input. We use tape 2 for the simulation. This we will use it for the simulation of the non-deterministic Turing machine. And tape 3 has the counter.

So, when I say counter, I mean we are trying to generate all these things root then 1, 2, up to b, then all the two-digit codes then the three-digit codes and so on. So, that is generated in the third tape. So, initially, we have the input in tape 1 and we copy the tape 1 content, which is the input to tape 2. And in the tape 2, we simulate the non-deterministic Turing machine move, but we do not have non-determinism.

So, we cannot automatically try out all the possible moves at once. Instead, we will, so let us say if the counter in tape 3 is 1 2 3 1 1 2 something. Then we will follow this path. So, whenever we have to make a choice between let us say multiple successor configurations, the first time we choose the first successor, second time we choose a second successor, third time the third successor, fourth time first successor, fifth time first successor, sixth time second successor and so on.

So, the counter will guide us which of the non-deterministic choices to exercise. And when we see an accepting configuration, we accept. So, the counter tells us which path we should take in this tree. So, the counter may tell us to take this path. So, it may tell us to take some path and we will follow that path. And finally, we accept when we reach an accepting state.

And after each time, when we reach the end of that path, we increment the counter. So, after let us say, 1, 2, let us say the counter says 1 2 3 1 1 2 now, next time, it will say 1 2 3 1 1 3.

And next time, let us say b is 3, 1 2 3 1 2 1 because now there is no 114. In that case, it will be 121 and so on. Let us say 113 is not there, if let us say that 1 2 3 1 1 does not have a third sign, then in that case we will not take that path because that path itself is not there.

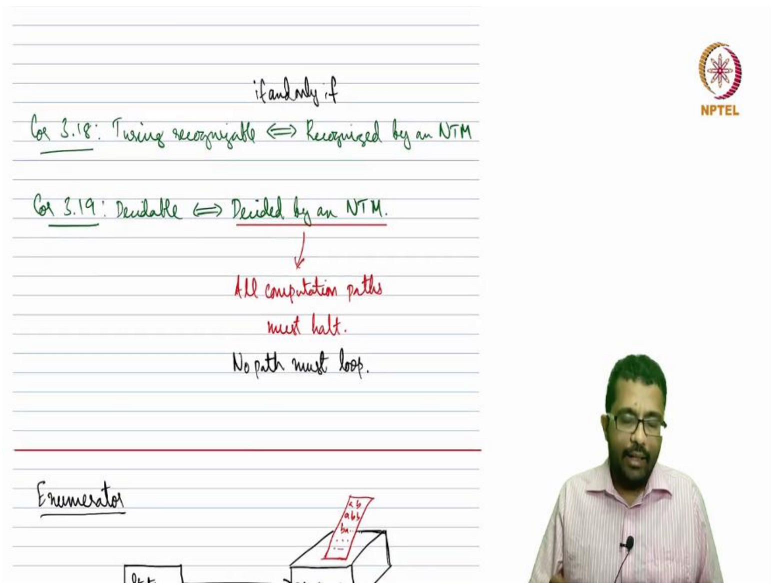
We will not accept, we will not reject, but we will just continue with the next option. And we reject the computation if let us say we come to these level by level and we reach a level. Let us say, at this level, all paths are rejecting. If all the paths at a certain level are rejecting, then we know that every path stops there. And in that case, we can stop. Otherwise, if there is at least one path continuing, then we will have to go to the next level.

So, that is the way to simulate a non-deterministic Turing machine using a deterministic Turing machine. So, the key thing is that a deterministic Turing machine cannot magically find the path, so it has to manually try to find out the accepting computation from all the possibilities. So, this actually takes more time because in non-deterministic Turing machine if the height is let us say h, a non-deterministic Turing machine takes only h time, whereas the deterministic Turing machine now has to traverse this entire tree of height b^h which is exponential in the height.

So, we are losing out on time, but as I said earlier, right now we are not focusing on the resources used. We are only seeing whether the language that we could recognize in a non-deterministic Turing machine can it be recognized by a deterministic Turing machine. So, the answer is yes. We do this process and if there is an accepting computation path, we will find it.

So, that is the algorithm by which if there is a language that is recognized by a non-deterministic Turing machine you can build an equivalent deterministic Turing machine. So, this means that deterministic and non-deterministic Turing machines are equivalent in power in terms of at least the class of languages that are recognized.

(Refer Slide Time: 16:00)



if and only if

Cor 3.18: Turing recognizable $\Longleftrightarrow$ Recognized by an NTM

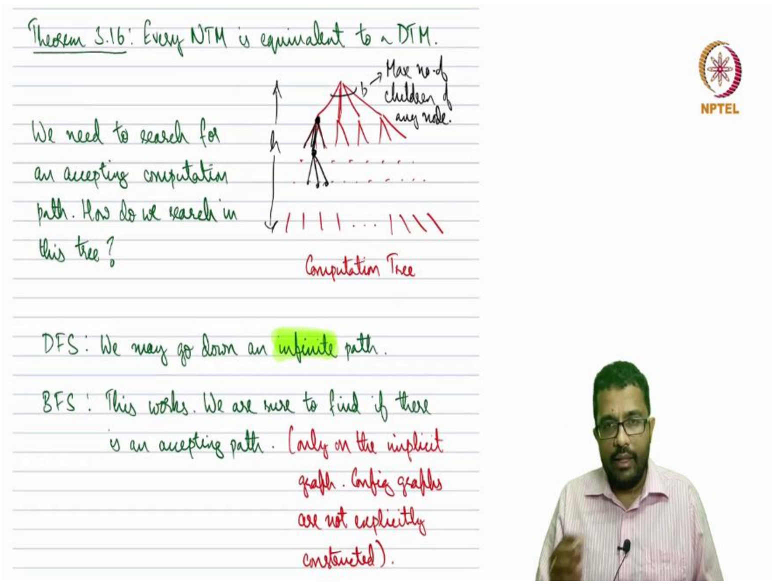
Cor 3.19: Decidable $\Longleftrightarrow$ Decided by an NTM.

$\downarrow$

All computation paths must halt.

No path must loop.

Enumerator



Theorem 3.16: Every NTM is equivalent to a DTM.

We need to search for an accepting computation path. How do we search in this tree?

DFS: We may go down an infinite path.

BFS: This works. We are sure to find if there is an accepting path. (only on the implicit graph. Config graphs are not explicitly constructed).

So, now, we can say that if a language is Turing recognizable if and only if it is recognized by a non-deterministic Turing machine. So, this double-sided arrow means if and only. And similarly, a language is decidable if and only if it is decided by a non-deterministic Turing machine. Because, non-deterministic and deterministic Turing machines have the same computation power.

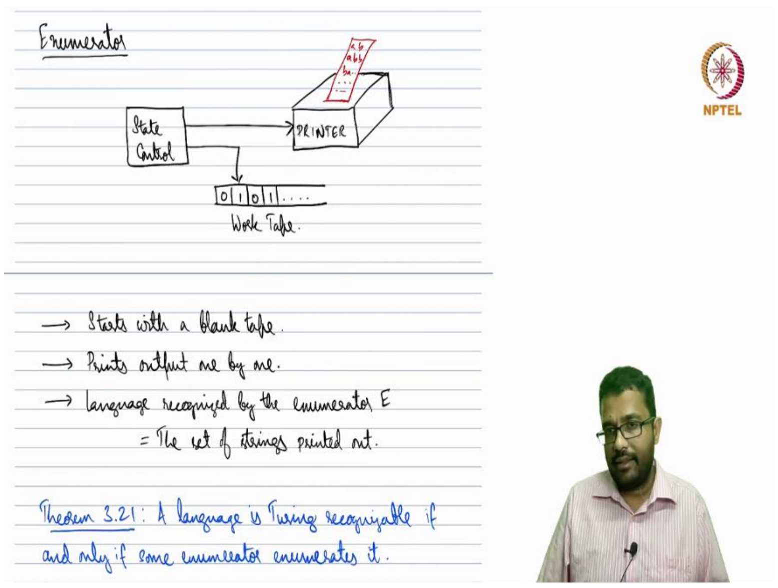
If you give me a non-deterministic Turing machine that accepts some strings, rejects some strings and loops on some strings, I can make a deterministic Turing machine that accepts exactly the same strings, rejects exactly the same strings and loops on exactly the same strings. So, if you are asked to show that some language is Turing recognizable it is enough to build a non-deterministic Turing machine that recognizes that language. It is not necessary that you have to build a deterministic Turing machine.

Building a deterministic Turing machine is also okay but even a non-deterministic Turing machine is okay. If you have to show that a language is decidable you can build a non-deterministic decider. Just that one point I want to mention about non-deterministic decider. All the computation paths must halt in a non-deterministic decider.

So, we said that a decider is something where for every string the computation halts. There is no looping. But in a non-deterministic Turing machine even for one string there are many, many computation paths. So, when do I say a string is decided by a non-deterministic Turing machine? When all the paths are decisive meaning if you feed a string into a non-deterministic Turing machine all the computation paths halt. All the paths either accept or reject. No path loops. No path must loop. So, that is the thing.

So, what we have seen is that every non-deterministic Turing machine has an equivalent deterministic Turing machine, and hence, the class of Turing recognizable languages are the same as the class of languages recognized by a non-deterministic Turing machine and the class of decidable languages are the same as the class of languages decided by any non-deterministic Turing machine.

(Refer Slide Time: 18:30)



Enumerator

State Control

PRINTER

0 1 0 1 ....

Work Tape.

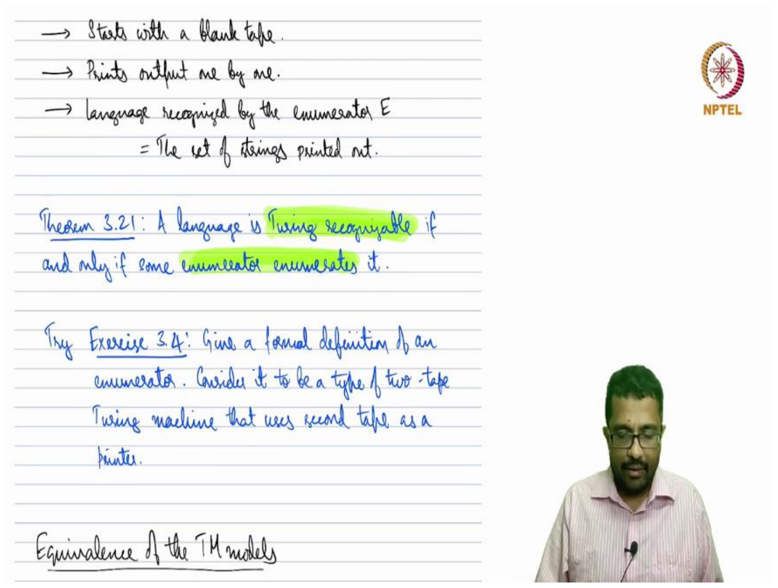
→ Starts with a blank tape.

→ Prints output one by one.

→ language recognized by the enumerator E

= The set of strings printed out.

Theorem 3.21: A language is Turing recognizable if and only if some enumerator enumerates it.



→ Starts with a blank tape.

→ Prints output one by one.

→ language recognized by the enumerator E

= The set of strings printed out.

Theorem 3.21: A language is Turing recognizable if and only if some enumerator enumerates it.

Try Exercise 3.4: Give a formal definition of an enumerator. Consider it to be a type of two-tape Turing machine that uses second tape as a printer.

Equivalence of the TM models

Finally, I just want to quickly point out one more model which is there the textbook which I will not spend much time upon. But I want to just briefly mention it. It is called an enumerator. So, enumerator, unlike a Turing machine, does not accept or reject. It is a machine which has a state control and a work tape. There is no input that is given. Instead, it has an output device which is a printer. So, this is an output device.

So, there is a work tape and a printer, these are the two things. Initially the tape is blank no input is given. So, there is no input and it works in a different way. It is not like a Turing machine where you give an input it accepts or rejects. Initially the work tape is blank, and then it starts working, you just switch on the machine it starts working. And it keeps printing strings in the printer.

So, the language recognized by the enumerator is a set of strings that are printed out. It prints out a bunch of strings and maybe an infinite list. So, the language recognized by the enumerator is a set of strings printed out. So, either a string is printed or not printed. There is no infinite loop kind of situation happening here. However, we do not know what order it prints, so it is possible that the string is not printed now, but it is an infinite list, it may come later.

So, there is this theorem which I will just state, which I will not get into the proof of. A language is Turing recognizable if there is some enumerator that enumerates it. So, when I say enumerates it, it means it prints it out. So, a language is Turing recognizable if there is some enumerator that enumerates it. The proof is not that difficult, you can look up the book.

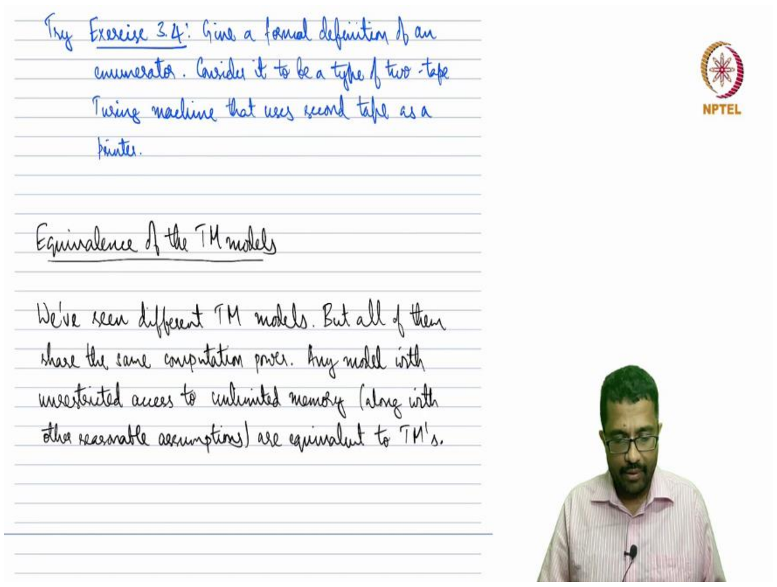
So, if there is some enumerator enumerating a language, before that it is if and only if. It is not if, if and only if. So, if an enumerator enumerates a language, then I can have a Turing machine that kind of simulates the enumerator and checks whether the given input string is printed out. So, that is one direction of the proof. So, the language recognized by the Turing machine will be the same language.

The other way is if there is a Turing machine and you want to build an equivalent enumerator what you can do is to, so basically you want the enumerator to print out all the possible strings that the Turing machine accepts. So, you can make the enumerator simulate the Turing machine on all possible strings. But if you try all possible strings in serial order that can get into an infinite loop. So, you have to do it kind of parallelly.

So, something similar to the breadth first search approach. So, you simulate it for one step together, then two steps and so on. And any string that is accepted by the Turing machine will eventually be printed out, that is the high-level idea. For the details, please have a look at the textbook. And there is also this exercise 3.4 which is the end of the chapter to give a formal definition for an enumerator.

So, you can write it like, when I say formal definition, I mean like we had a 5 tuple, 6 tuple and so on. So, you can write a formal definition and say what is the transition etcetera. So, the suggestion is, you can consider it to be 2-tape Turing machine and the second tape is the printer and the first tape is the work tape. So, we have seen enumerator, we have seen non-deterministic Turing machine and the equivalence of non-deterministic Turing machine.

(Refer Slide Time: 22:42)



Try Exercise 3.4: Give a formal definition of an enumerator. Consider it to be a type of two-tape Turing machine that uses second tape as a printer.

Equivalence of the TM models

We've seen different TM models. But all of them share the same computation power. Any model with unrestricted access to unlimited memory (along with other reasonable assumptions) are equivalent to TM's.

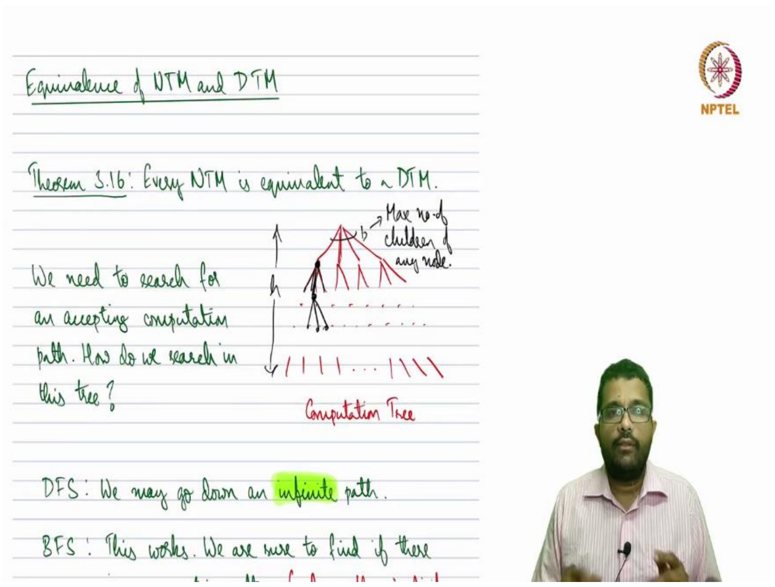
Finally, I just want to say one small point while ending this lecture. So, we have seen different models of Turing machine. So, we have seen deterministic, multi-tape deterministic and non-deterministic and all of them seem to have the same computation power. Meaning if a language is recognized by one model it is recognized by all three models, if it is decided by one model it is decided by all the three models.

So, in fact, the thing is that the Turing Machine model itself is fairly robust that these changes like from determinism to non-determinism or the increase in the number of tapes, it does not really affect what can be recognized or decided. So, the key things that we want are unlimited memory and unrestricted access to this memory. So, if you consider the DFA it had finite memory, it had limited memory. If you had considered an NFA again it had finite memory.

If you consider the PDA it had a stack which is infinite memory, but it had only restricted access to that stack. So, we did not have unrestricted access to this memory. But any model that has unrestricted access to an unlimited memory along with some reasonable other assumptions, which I do not want to get into. But if a model has unrestricted access to unlimited memory, then it is likely that this model of computation will be equivalent to Turing machines.

So, in that sense the model of Turing machines is kind of robust. Even slight changes that you make, for instance when we defined the Turing machine, we said that the tape is infinite to one side, but not to the other side there was a left end but infinity to the right. Instead, what if we had it extending to infinity on both sides, even this is equivalent to the Turing Machine model. So, this is just another example. And I think that is all that I have in Lecture number 31.

(Refer Slide Time: 24:43)



Equivalence of NTM and DTM

Theorem 3.16: Every NTM is equivalent to a DTM.

We need to search for an accepting computation path. How do we search in this tree?

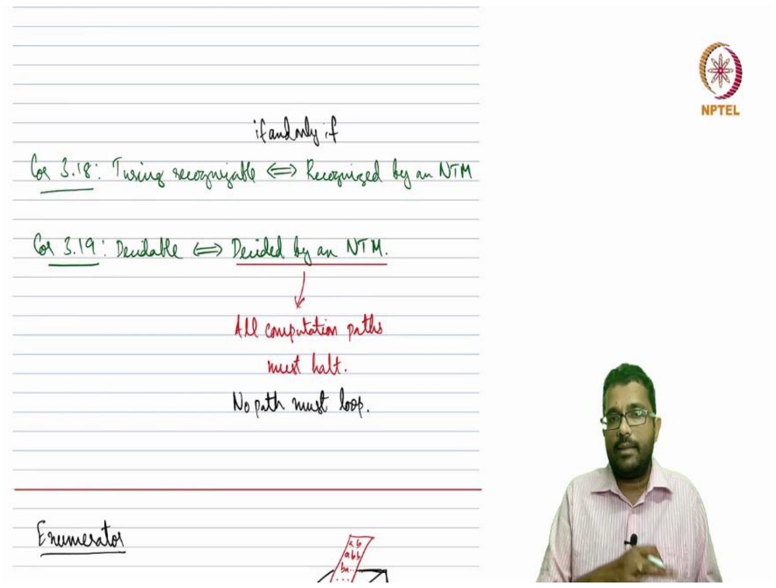
DFS: We may go down an infinite path.

BFS: This works. We are sure to find if there

So, we first talked about the equivalence of non-deterministic and deterministic Turing machines. We had to simulate a non-deterministic Turing machine with a deterministic Turing machine. So, we said depth first search does not work and we tried the breadth first search approach and we formalized it.

So, in this simulation we mentioned that we may be losing out on time right because in non-deterministic Turing machine may take only h time to compute this tree, whereas a deterministic Turing machine will take b^h time to try out all the possibilities, but that is okay. We are not bothered about resources now, perhaps later in the time complexities chapter we will see how much resources this takes.

(Refer Slide Time: 25:33)



if and only if

Cor 3.18: Turing recognizable $\Longleftrightarrow$ Recognized by an NTM

Cor 3.19: Decidable $\Longleftrightarrow$ Decided by an NTM.

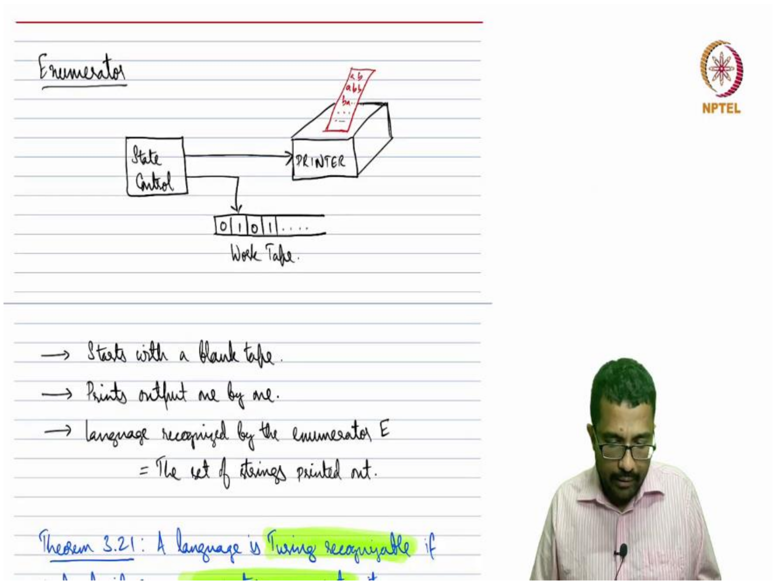
All computation paths must halt.

No path must loop.

Enumerator

And then we define, then we said that the class of Turing recognizable languages are the class of languages recognized by a non-deterministic Turing machine, if and only if, and the class of a language is decidable if and only if it is decided by an NTM. When I say decided by non-deterministic Turing machine, I want all the strings to halt and all computation paths of all the strings to halt. Even for one specific string, I want all the computation paths to halt, either accept or reject.

(Refer Slide Time: 26:05)



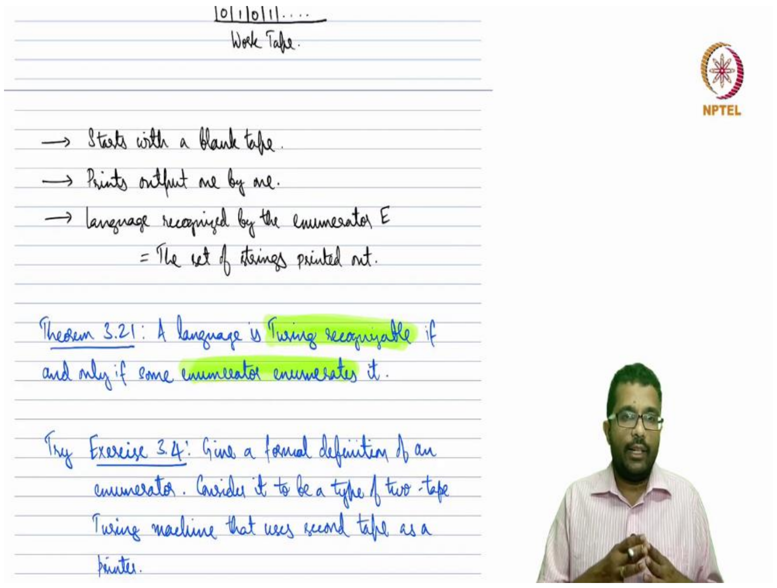
Enumerator

→ Starts with a blank tape.

→ Prints output one by one.

→ language recognized by the enumerator E = The set of strings printed out.

Theorem 3.21: A language is Turing recognizable if



|0|1|0|1|1|....

Work Tape.

→ Starts with a blank tape.

→ Prints output one by one.

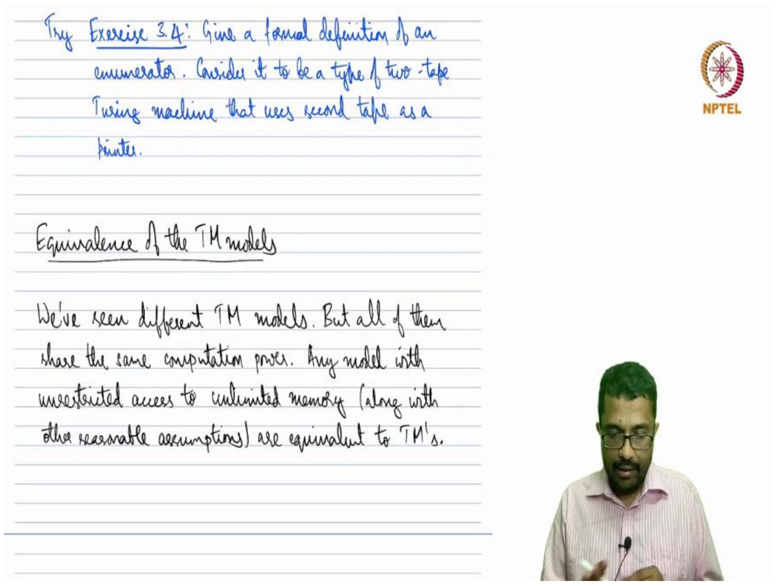
→ language recognized by the enumerator E
= The set of strings printed out.

Theorem 3.21: A language is Turing recognizable if and only if some enumerator enumerates it.

Try Exercise 3.4: Give a formal definition of an enumerator. Consider it to be a type of two-tape Turing machine that uses second tape as a printer.

We then saw the enumerator which was just basically a printer with a work tape. And we said that the language enumerated by an enumerator is the same as the class of Turing recognizable languages.

(Refer Slide Time: 26:22)



Try Exercise 3.4: Give a formal definition of an enumerator. Consider it to be a type of two-tape Turing machine that uses second tape as a printer.

Equivalence of the TM models

We've seen different TM models. But all of them share the same computation power. Any model with unrestricted access to unlimited memory (along with other reasonable assumptions) are equivalent to TM's.

Finally we conclude by saying that any models of computation which have unlimited memory and unrestricted access to the unlimited memory are all equivalent to Turing machines. And I think that is all I have in this lecture, Lecture number 31. And we will conclude this chapter, chapter number 3 in the next lecture. So, see you in the next lecture. Thank you.