


Theory of Computation
Professor Subrahmanyam Kalyanasundaram
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Computation Histories

(Refer Slide Time: 0:17)




Computation Histories

This is an approach to show that some languages are undecidable. For instance, the undecidability of the **Hilbert tenth problem** uses this approach.

Computation History of a TM on an input is simply the sequence of configurations that the TM goes through while it processes the input.

It is represented as $C_1 \# C_2 \# \dots \# C_n$ where C_1, C_2, \dots, C_n are configurations.

Def 9.5: Let M be a TM and w be an input string. An accepting computation history of M on w is a sequence of configurations C_1, C_2, \dots, C_n such that




C_1, C_2, \dots, C_n are configurations.

Def 9.5: Let M be a TM and w be an input string. An **accepting computation history** of M on w is a sequence of configurations C_1, C_2, \dots, C_n such that C_1 is the starting configuration of M on w , C_n is an accepting configuration, and each C_i legally follows from C_{i-1} , according to the rules of M .

A **rejecting CH** is defined the same way, but C_n is a rejecting configuration.

$$ALL_{CFG} = \{ \langle n \rangle \mid n \text{ is a CFG, } L(n) = \Sigma^+ \}.$$

Theorem 9.13: ALL_{CFG} is undecidable.



Hello and welcome to lecture 42 of the course theory of computation, in the previous lecture we saw Rice's theorem and we have also seen other undecidable languages so Rice's theorem was a technique or a theorem to show a decidability of a certain type of languages, we had also seen other undecidable languages by using reductions. So, in this lecture we will see computation histories, computation history is an approach to show undecidability of some languages.

In particular we will use this approach to show reduction from A_{TM} to this language so since A_{TM} if you can show that A_{TM} reduces to a certain language then the language that we reduce A_{TM} to is also undecidable, so computation history is a certain technique or approach to perform the reduction from A_{TM} to the language that we want because in some cases we are able to use this approach.

For instance one of the, one of the interesting cases is the Hilbert's tenth problem that we discussed at the end of chapter 3, this that that was asked by David Hilbert in the ICM this was shown to be undecidable using this approach, so what is computation history, so computation history is just like a history of computing a certain string by a certain TM, so as we already know configuration consists of the tape content, head position and the state of the Turing machine at a certain point of time.

So, computation history is given let us say computation history of a Turing machine M on a string W is nothing but the sequence of configurations that M goes through while computing on W , so it is written as a series of configurations like this like $C1 \# C2 \# \dots \# C_l$ where $C1$ to C_l are the configurations and these hashes are simply delimiters like to separate one configuration from the other.

$$C1\#C2\#C3\#\dots\#C_l$$

So, there are two definitions one is accepting computation history and one is rejecting computation history, accepting computation history is nothing but sorry, is nothing but computation history that leads to an acceptance of a string, so accepting computational history is a sequence of configuration $C1$ up to C_l where the sequence of configurations lead to the string getting accepted, meaning as per the rules of what when does this thing get accepted $C1$ must be the starting configuration of M on W meaning $C1$ should be the starting state followed by the input string W $W1, W$ the individual symbols of W and C_l must be an accepting configuration only then it gets accepted.

So, the last configuration must be an accepting configuration and everything else must be a valid successor of the previous configuration so $C2$ should be a valid successor of $C1$, $C3$ should be a varied successor of $C2$, $C4$ should be a valid successor of $C3$ and so on and so on till C_l should be a valid successor of C_l minus 1, so all the moves should be valid moves such a sequence of configurations is called an accepting computation history, so I want each configuration to be a valid successor of the previous one the starting configuration should be

the to be the correct starting configuration and the accepting configuration should be an accepting configuration.

Similarly, we can define rejecting configuration computational history sorry, where everything is the same, rejecting computational history is defined exactly the same way just that instead of CI being an accepting configuration we want CI to be a rejecting configuration so these are the two definitions, what we will just now for our purposes of this lecture we only be using accepting computation history.

(Refer Slide Time: 4:34)

$ALL_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG, } L(G) = \Sigma^* \}$.

Theorem 5.13: ALL_{CFG} is undecidable.


Proof Sketch: We reduce $A_{TM} \leq_m ALL_{CFG}$ using computation histories.


Given M and w , we will construct a CFG/PDA that generates all the strings that are not an accepting CH of M on w .

M accepts $w \Rightarrow$ CFG generates all but one string.

M does not accept $w \Rightarrow$ CFG generates all strings.

$ACH_{M,w} = \{ z \in \Delta^* \mid z \text{ is an accepting} \}$





So, what we will show in this lecture or at least show the outline of in this lecture is a language that we have referred to couple of times in the past, the language is ALL_{CFG} , ALL_{CFG} is asking given a context free grammar does this grammar generate all possible strings, does it generate all the strings over the terminal set Σ , so this problem happens to be undecidable, so you may recall we showed that, we assumed ALL_{CFG} is undecidable and we showed that EQ_{CFG} is also undecidable because you could reduce ALL_{CFG} to EQ_{CFG} but we had never, we had just stated that ALL_{CFG} is undecidable and now we will see the kind of sketch of the proof that ALL_{CFG} is undecidable.

So, it is an undecidable problem given a grammar, given a context free grammar to determine whether this generates the set of all strings, all strings possible over the terminal set, let us see why, so we will reduce A_{TM} to all ALL_{CFG} complement, we will reduce, so if I reduce A_{TM} to ALL_{CFG} complement and if I reduce A_{TM} to ALL_{CFG} complement this implies that ALL_{CFG} complement is undecidable but that also implies that ALL_{CFG} is undecidable because if a

language endless undecidable its complement is also undecidable, if L complement is undecidable then it is also undecidable because if you can decide, if you can decide the language L complement you can just flip the output yes to no and no to yes to get a decider for L so if ALL_{CFG} is undecidable ALL_{CFG} complement is undecidable there is no way for ALL_{CFG} to be decidable because you can flip the decider's output to get a decider for ALL_{CFG} complement.

So, this implies that ALL_{CFG} is undecidable as well and the reduction from A_{TM} to ALL_{CFG} will use computation histories, so while let us see how it uses, so given M and w the A_{TM} instances does M accept w that is what we have to decide, what we will do is given the A_{TM} instance M and w we will construct a context free grammar or equivalently a PDA, context-free grammars and PDA, they are equivalent, they both recognize the class of context-free languages this grammar or PDA will generate all the strings which are not an accepting computation history of M on w which are not that is also important which are not an accepting computation history of M on w .

So, it will generate everything which is not an accepting computation history, so which means so it is a very it is slightly twisted suppose M accepts w then it will not generate the accepting computation history of M on w , it will generate all the other strings, the grammar will generate all the other strings, suppose M does not accept w , so if M does not accept w there is no way for it to accept w which means there is no accepting computation history at all because only if M accepts w can we have an accepting computation history because otherwise it cannot be valid.

So, if M does not accept w the context free grammar will generate all possible strings, it will generate everything because there is no accepting computation history so the by design it is supposed to generate all strings that are not an accepting computational history, so if M does not accept w it will generate all the strings if M accepts w it will generate all the strings with the exception of the accepting computation history. So, when M accepts w the grammar will not be able to generate all the strings but when M does not accept w the grammar is going to be able to generate all the strings.

(Refer Slide Time: 9:15)

M does not accept $w \Rightarrow$ CFG generates all strings.
 $ACH_{M,w} = \{z \in \Delta^* \mid z \text{ is an accepting CH of } M \text{ on } w\}$
 $\langle M, w \rangle \in A_{TM} \Leftrightarrow M \text{ accepts } w$
 $\Leftrightarrow ACH_{M,w} \neq \emptyset$
 $\Leftrightarrow \overline{ACH_{M,w}} \neq \Delta^*$
 The grammar that generates $ACH_{M,w} \Leftrightarrow ACH_{M,w} \notin ALL_{CFG}$.
 $\Leftrightarrow \overline{ACH_{M,w}} \in ALL_{CFG}$
 To complete the reduction $A_{TM} \leq_m ALL_{CFG}$, we need to show that there is a CFG that

So, let me define formally $ACH_{M,w}$, $ACH_{M,w}$ is the set of all accepting computation histories of M on w , so with the set of all strings Z where Z is in Δ^* so Δ means Δ includes the input string γ , it also includes the state tape string it also includes States it also includes hash symbols every everything that that comes in the tapes or in the computation history so any string that is formed out of Δ which is an accepting computation history so ACH contains all the strings that are an accepting computer in history of M on w .

So, this same is what I just said $\langle M, w \rangle \in A_{TM}$ if and only if M accepts w now if M accepts w the accepting computation history is not empty if it accepts then there is an accepting computational history, if it does not accept there is no such thing so accepting computation history is not empty means the complement is not everything so the complement of accepting computation history is not the set of all strings possible which means the grammar that generates this $ACH_{M,w}$ so maybe to be more precise the grammar that generates this maybe I will just, the grammar that generates this $ACH_{M,w}$ should not be in ALL_{CFG} .

The grammar that generates ACH complement which means the grammar again here also the same thing the grammar that generates this should not be in ALL_{CFG} means that it should be in the complement of ALL_{CFG} so that is the reduction so maybe I will edit this that is the reduction if $\langle M, w \rangle \in A_{TM}$, M accepts w which means there is an accepting computation history which means the complement of $ACH_{M,w}$ does not contain everything and hence it is not part of ALL_{CFG} .

So, and if M does not accept w then there is no accepting computation history and the complement of $ACH_{M,w}$ is all the setups all strings and hence the grammar that generates it is part of ALL_{CFG} , so this gives a reduction from A_{TM} , M from given M w we are constructing the grammar, so it is a reduction from A_{TM} to ALL_{CFG} complement given an instance of A_{TM} we are giving a instance of ALL_{CFG} complement or given a yes instance of A_{TM} we are giving a yes instance of ALL_{CFG} complement which is a no instance of ALL_{CFG} but yeah.

So, now what we need to show is to construct a grammar that generates $ACH_{M,w}$ complement, this grammar the grammar that we refer to here we need to build to complete the reduction we need to build this, so just to summarise we want to reduce A_{TM} to ALL_{CFG} complement the we use computation histories if M accepts w so we build a grammar that generates all the strings that are not accepting computation history of M on w we build a grammar that generates all the strings that are not accepting computation histories.

So, if M accepts w then the grammar cannot generate all the strings because there is an accepting computation history if M does not accept w then there is no accepting computation history which means the complement of ACH will, can generate potentially everything so it is a member of ALL_{CFG} that is the reduction.

(Refer Slide Time: 14:02)


$\Leftrightarrow ACH_{M,w} \in ALL_{CFG}$


To complete the reduction $A_{TM} \leq_m ALL_{CFG}$, we need to show that there is a CFG that generates $ACH_{M,w}$.

Given $z \in \Sigma^*$, there are four possibilities on why $z \in ACH_{M,w}$ (or why z is not an accepting CH of M on w). The CFG can nondeterministically choose one of the following.

- z is not well formed. Maybe does not start or end with $\#$. Either no state or more than one state between two $\#$'s.

- This is a regular language and can be checked using a DFA.







Computation History of a TM on an input is simply the sequence of configurations that the TM goes through while it processes the input.



It is represented as $\#C_1\#C_2\#\dots\#C_n\#$ where C_1, C_2, \dots, C_n are configurations.

Def 9.5: Let M be a TM and w be an input string. An **accepting computation history** of M on w is a sequence of configurations C_1, C_2, \dots, C_n such that C_1 is the starting configuration of M on w , C_n is an accepting configuration, and each C_i legally follows from C_{i-1} , according to the rules of M .

A **rejecting CH** is defined the same way, but C_n is a rejecting configuration.

- z is not well formed. Maybe does not start or end with $\#$. Either no state or more than one state between two $\#$'s.
 - This is a regular language and can be checked using a DFA.
- z does not start correctly. The first config. is not the starting config. of M on w , i.e., $q_0 w_1 w_2 \dots w_n$.
 - Regular. Can be checked using a DFA.
- z does not end correctly. The state between the last two $\#$ symbols is not the accepting state.

And now what remains is to show the grammar, show the grammar that generates the complement of accepting computation histories of M on w , so this part is where we are not going to get into full details because the details may be quite technical so just I just want to go to the high level points and the details is not that hard it can be worked out but it may be too boring or too detailed to kind of go through all of the details.

So, we want a grammar to generate strings in the complement of $ACH_{M,w}$, so how can a string not be an accepting computation history all such things we need to generate so there are four possibilities first the string is not even of this, so we want the computation history to be denoted like this, this is our desired form for the computation history configuration starting from the starting configurations to the accepting configurations separated through valid transition and separated by delimiters hash symbols.

So, perhaps the input itself is not given in this format it is not a properly formed input the string is not a well-formed computation history, it cannot be read or interpreted as one in that case it is not an accepting computation history, so this maybe it does not have a hash symbol or maybe it does not end with the hash symbols or maybe between two hash symbol there is no state it is not a valid configuration or maybe there are two states so all of this is an example of not being well formed.

This is actually something that can be checked by a DFA or a regular language. It is a regular language because you are just checking whether every hash follows between every two hashes there is a state and all that whether it starts and ends with the hash it is actually a regular language.

So, again I am not getting into the detail of how you can check it but it should be straightforward to see that it is a regular language, so in that case it is not an accepting computation history, the second case is maybe it is well formed meaning it starts with hash ends with hash and it has $C_1 C_2$ etc. but maybe I just missed a small thing and it starts with hash I think we need this, it has to be this form I think, maybe it does look like a proper computation history but let us say it does not start correctly meaning the first configuration is not the starting configuration of M on w it is a configuration but it is something else.

So, the starting configuration of M on w should be looking like this where Q_s is the starting config starting state and $w_1 w_2$ etcetera is the string w , so is the first configuration equal to this, if not that is another way why z is not an accepting computation history, this is also easy to check all we are asking is the first configuration equal to something, this also can be checked using a DFA.

(Refer Slide Time: 17:28)

- Regular. Can be checked using a DFA.



3. z does not end correctly. The state between the last two # symbols is not the accepting state.

- Regular. Can be checked using a DFA.

4. There is an i such that C_{i+1} is not a valid successor of C_i , i.e., $C_i \# C_{i+1}$ is not valid.

- Non-deterministically guess i such that C_{i+1} is not a successor of C_i .
- Use a PDA to check that $C_i \# C_{i+1}$ is not valid.

Try problem 2.22. Show that C is context-free.



M end with #. Even no state or more than one state between two #'s.

- This is a regular language and can be checked using a DFA.



2. z does not start correctly. The first config. is not the starting config. of M on w , i.e., $q_0 w_1 w_2 \dots w_n$.

- Regular. Can be checked using a DFA.

3. z does not end correctly. The state between the last two # symbols is not the accepting state.

- Regular. Can be checked using a DFA.

4. There is an i such that C_{i+1} is not a valid



4. There is an i such that C_{i+1} is not a valid successor of C_i , i.e., $C_i \# C_{i+1}$ is not valid.

- Non deterministically guess i such that C_{i+1} is not a successor of C_i .
- Use a PDA to check that $C_i \# C_{i+1}$ is not valid.

Try problem 2.22. Show that C is context-free.

$$C = \{ x \# y \mid x, y \in \{0,1\}^* \text{ and } x \neq y \}$$

Main ideas for checking that $C_i \# C_{i+1}$ is not valid.

abcabcagb...	#	
↓		
abcabc#ac		$\delta(q_i, b) = (q_i, c, L)$

Third possibility is that it is a properly formed computation history but it does not end correctly meaning the last configuration is not an accepting configuration meaning the state between the last two hash symbols is not the accepting state so it is not the accepting state this also can be checked easily using a DFA because we just have to go to the last two hashes and check whether you remember the last states seen and check whether the last state seen is an accepting state this also can be checked using a DFA.

So, this is also easy not that difficult to see then, so we have covered the string being well formed in sense of a conflict computation history, it is starting correctly meaning C_1 is a starting configuration, ending correctly meaning the last configuration is an accepting configuration then the third, the final way in which the computation history may not be a valid computation history is if at some point, so we have $C_1 C_2 C_3$ this is a sequence of configurations at some point there is some i such that C_i plus 1 is not a valid successor of C_i there is an i such that C_i is not a valid successor of C_i plus 1 is not a valid successor of C_i meaning there is some part where this sub part C_i, C_i plus 1, C_i hash C_i plus 1 is not a valid successor.

So, if there is such a thing then it is again not a valid accepting computation history because somewhere we started correctly ended correctly but somewhere the it is not a valid successor so how do we do this, we non deterministically guess an i so we have a PDA so far we just said that everything else can be done by a regular or DFA so this part requires a PDA or a context free grammar we can non deterministically guess an i for which C_i plus 1 is not a valid successor of C_i and then we check whether this does not happen right.

And then we use a PDA like once we guess an i we use a PDA to check whether this is not a valid pair of configurations meaning what do I mean by not valid, C_i plus 1 is not a varied successor of C_i this can be checked using a PDA, so again details I am not getting into the details that is why I called it a proof sketch to begin with.

But the idea is this so one good exercise is there is a problem 2.22 in the book so I am reproducing the problem here so set of all strings like it is a set it is a language called C which is of the form $x \neq y$, strings of the form $x \neq y$ where x and y are from $\{0, 1\}^*$ such that $x \neq y$.

So, I will tell you the idea for C , so one way to do it is you let us you guess that if x and y are not equal you can guess that they may differ in the k th symbol, there are three possibilities so one is that x is or two possibilities one is that x and y are of two different lengths then they are immediately not equal so maybe there are three possibilities, one is that x is longer than y , two is that y is longer than x , three is that x and y are of the same length but at some point some symbol differs meaning maybe some k th symbol of x is not the k th symbol of y .

So, what you can do is you can guess some K , I am just addressing the third part so you can non-deterministically decide which path to pursue, so you guess K , so to guess K you just keep pushing symbols into the stack, so at some point let us say after pushing let us say K symbols maybe K is 10, 10 symbols you look at the 11th symbol of x you remember it and you go to the hash and then you start popping out the stack when you are reading the symbols of y .

So, that you pushed 10 things into the stack, so the first 10 symbols of y are going to be passed by just pushing it, we are not going to compare or anything, this is just used as a counter and then when we come to the 11th symbol of y we will see that the stack is at empty you can put a dollar or something to identify that and then you try to remember what you saw earlier and see whether the 11th symbol of x is the same as 11th symbol of y .

So, you guess which position to so you non-deterministically decide which position to compare till that point you push things and push things into the stack and use that as a counter to get to the same point at y , so this is how and then you accept if the strings are different if the 11th symbol is different or the $K + 1$ symbol is different. The same ideas kind of so maybe this is something that you can try working out, this problem C .

(Refer Slide Time: 23:07)

Try problem 2.22. Show that C is context-free.

$$C = \{ x \# y \mid x, y \in \{0,1\}^* \text{ and } x \neq y \}.$$

Main ideas for checking that $C_i \neq C_{i+1}$ is not valid.



C_i a b c a b c a y b ... #
↑
 C_{i+1} a b c a b c # a c

$\delta(q, b) = (r, c, L)$

→ We have to nondeterministically choose the position where C_{i+1} makes an "error", say k .

→ Use stack to keep track of k .

→ Need to remember the previous two symbols in case the state (tape head) is nearby.

And the ideas for checking C_i hash C_i hash C_i plus 1 is not valid similar just that now that we are not dealing with x not equal to y we are dealing with two configurations where C_i plus 1 is not a successor of C_i , so earlier here we said x should not be y , here we are saying C_i plus 1 should not be a successor of C_i so earlier there was equality here we are saying not a valid successor.

So, if you write the let us say this is C_i , this is C_i , this is C_i plus 1 so if you write it like you have some strings and it will, so if you write two successive configurations, you see that configurations do not change all that much only the part around the head changes, so the head is here a q b because q indicates a state and this means the head is pointing to b here that is replaced by r a c so only these symbols around the head changes.

So, you can check you may have to remember more details because two or three symbols may change and you do not know where the head is going to be well processing, so you have to not only determine which position K that you are going to check but you also have to remember couple more states, couple more symbols not just the, so in the case of C we said we remember just the K plus 1 symbol here we mean it we may need to remember more symbols than just K plus 1, so that is the main idea here, we need to remember more symbols because this transition that I have written here this happens when $\delta(q, b) = (r, c, L)$ meaning here the status queue it is, the head is pointing at b and the move here is that state goes to r , b is overwritten by the symbol c and then the head moves one step to the left.

So, now this a has moved over here so this you can see, this a has moved over here, the state has more one step to the left and b is changed to C so that is the result of this move if it was a

move the head moved to the it would be slightly different but the point is that whatever is the move corresponding to the particular configuration the move, the next step we will have to we will have to identify that and make amends in the or make changes in the or will have to construct the context free grammar appropriately in order to track that. So, that is the main idea of the proof that ALL_{CFG} is undecidable.

(Refer Slide Time: 26:21)

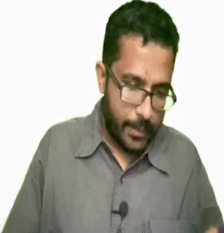

Computation Histories

This is an approach to show that some languages are undecidable. For instance, the undecidability of the **Hilbert tenth problem** uses this approach.

Computation History of a TM on an input is simply the sequence of configurations that the TM goes through while it processes the input.

It is represented as $\#C_1\#C_2\#\dots\#C_t\#$ where C_1, C_2, \dots, C_t are configurations.

Def 9.5: Let M be a TM and w be an input string. An **accepting computation history** of M on w is a sequence of configurations C_1, C_2, \dots, C_t such that C_1 is the starting configuration of M on w , C_t is





Def 9.5: Let M be a TM and w be an input string. An **accepting computation history** of M on w is a sequence of configurations C_1, C_2, \dots, C_t such that C_1 is the starting configuration of M on w , C_t is an accepting configuration, and each C_i legally follows from C_{i-1} , according to the rules of M .

A **rejecting CH** is defined the same way, but C_t is a rejecting configuration.

$ALL_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG, } L(G) = \Sigma^+ \}$.

Theorem 9.13: ALL_{CFG} is undecidable.

Proof Sketch: We reduce $A_{TM} \leq_m ALL_{CFG}$



of Σ^* .

A rejecting CH is defined the same way, but C_0 is a rejecting configuration.

$$ALLCF_n = \{ \langle G \rangle \mid G \text{ is a CFG, } L(G) = \Sigma^* \}$$

Theorem 5.13: $ALLCF_n$ is undecidable.

Proof Sketch: We reduce $A_{TM} \leq_m \overline{ALLCF_n}$ using computation histories.

Given M and w , we will construct a CFG/PDA that generates all the strings that are not an accepting CH of M on w .

M accepts $w \Rightarrow$ CFG generates all but one string.

String.

M does not accept $w \Rightarrow$ CFG generates all strings.

$$ACH_{M,w} = \{ z \in \Sigma^* \mid z \text{ is an accepting CH of } M \text{ on } w \}$$

$$\langle M, w \rangle \in A_{TM} \Leftrightarrow M \text{ accepts } w$$

$$\Leftrightarrow ACH_{M,w} \neq \emptyset$$

$$\Leftrightarrow \overline{ACH_{M,w}} \neq \Sigma^*$$

The grammar that generates $\overline{ACH_{M,w}} \Leftrightarrow \overline{ACH_{M,w}} \in \overline{ALLCF_n}$.

$$\Leftrightarrow ACH_{M,w} \in ALLCF_n$$

To complete the reduction $A_{TM} \leq_m \overline{ALLCF_n}$, we need to check that there is a CFG that

$z \in ACH_{M,w}$ (or why z is not an accepting CH of M on w). The CFG can nondeterministically choose one of the following.

1. z is not well formed. Maybe does not start or end with $\#$. Either no state or more than one state between two $\#$'s.

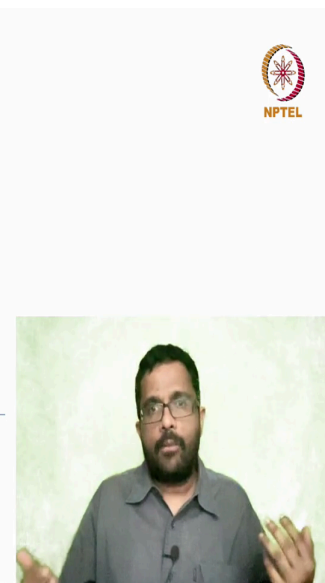
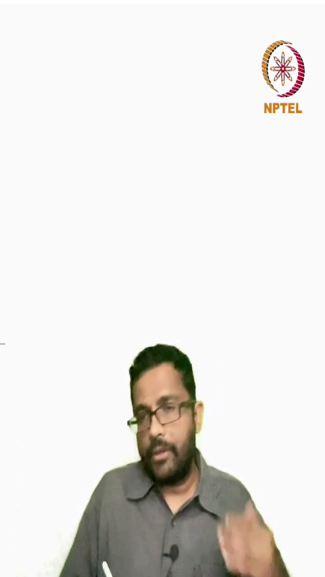
- This is a regular language and can be checked using a DFA.

2. z does not start correctly. The first config. is not the starting config. of M on w , i.e.,

$$q_0 w_1 w_2 \dots w_n$$

- Regular. Can be checked using a DFA.

3. z does not end correctly. The state between



3. z does not end correctly. The state between the last two # symbols is not the accepting state.



- Regular. can be checked using a DFA.

4. There is an i such that C_{i+1} is not a valid successor of C_i , i.e., $C_i \# C_{i+1}$ is not valid.

- Non-deterministically guess i such that C_{i+1} is not a successor of C_i .

- Use a PDA to check that $C_i \# C_{i+1}$ is not valid.

Try problem 2.22. Show that C is context-free.

$$C = \{ x \# y \mid x, y \in \{0,1\}^* \text{ and } x \neq y \}.$$



Try problem 2.22. Show that C is context-free.

$$C = \{ x \# y \mid x, y \in \{0,1\}^* \text{ and } x \neq y \}.$$

Main ideas for checking that $C_i \# C_{i+1}$ is not valid.

C_i abcabcagb... #



\downarrow \downarrow
 C_{i+1} abcabc+ac

$\delta(q, b) = (r, c, L)$

→ We have to nondeterministically choose the position where C_{i+1} makes an "error", say k .

→ Use stack to keep track of k .

→ Need to remember the previous two symbols in case the state (tape head) is nearby.

Maybe I will just quickly summarise so an accepting computation history is a sequence of configurations starting with starting configuration ending at an accepting configuration where each transition is valid, so to show that ALL_{CFG} is undecidable we reduce A_{TM} to ALL_{CFG} , so given an A_{TM} instance which is a pair Mw we will build a grammar that generates all these strings that are not an accepting computation history and so if M accepts w this grammar cannot generate all the strings because there is a valid computation history.

If it does not accept w it can generate everything, so Mw is in A_{TM} if and only if the grammar does not generate all the strings or if and only if the grammar is in the complement of ALL_{CFG} that is how we get the reduction of A_{TM} from ALL_{CFG} and now how do you build such a grammar?

There are four possibilities the first one being the we want to generate ill-formed strings which is not a configuration, second one is it does not start correctly, third one is it does not end correctly meaning it is not accepting the last configuration is not an accepting configuration the final thing is what we required some work and that too we just gave the high level overview and not the entire proof that is there is some i for which C_i plus 1, the i plus 1 configuration is not a valid successor of C_i .

And that we said that we can do it by using a PDA but I am kind of alternatively interchangeably using PDA and CFG, so if there is a PDA we can use a CFG as well so we keep track of some like where to compare using the stack and then you remember the last couple of symbols read so that you know how to adjust yourself and if you read the book this part of the proof is written slightly differently, this last part.

They use, the book uses something where alternate configurations are written in the reverse manner so you have C_1 then C_2 in reverse then C_3 then C_4 in reverse or something like that so that is also correct this is a this is slightly different but this is also a valid approach so that is how this, proves that we can build a CFG that generates all the strings that are not an accepting computation history and that completes the reduction from ATM to the complement of ALL_{CFG} . This means the complement of ALL_{CFG} is undecidable which also means that ALL_{CFG} is undecidable and that is the main thing that I want to cover in this lecture.

(Refer Slide Time: 29:18)

Post Correspondence Problem (PCP)

This is a simple undecidable problem.

Given dominos $\left\{ \begin{bmatrix} t_1 \\ b_1 \end{bmatrix}, \begin{bmatrix} t_2 \\ b_2 \end{bmatrix}, \dots, \begin{bmatrix} t_n \\ b_n \end{bmatrix} \right\}$


can we arrange them (repeats allowed) so that
top string = bottom string?


Example: $\left\{ \begin{bmatrix} ab \\ labal \end{bmatrix}, \begin{bmatrix} ba \\ labb \end{bmatrix}, \begin{bmatrix} b \\ lab \end{bmatrix}, \begin{bmatrix} abb \\ b \end{bmatrix}, \begin{bmatrix} a \\ lab \end{bmatrix} \right\}$

$\begin{bmatrix} ab \\ labal \end{bmatrix} \begin{bmatrix} a \\ lab \end{bmatrix} \begin{bmatrix} ba \\ labb \end{bmatrix} \begin{bmatrix} b \\ lab \end{bmatrix} \begin{bmatrix} abb \\ b \end{bmatrix} \begin{bmatrix} abb \\ b \end{bmatrix} \begin{bmatrix} b \\ lab \end{bmatrix} \begin{bmatrix} abb \\ ab \end{bmatrix}$

↑
This is a match.

PCP is an undecidable problem. How about it next week.







C_1, C_2, \dots, C_n are configurations.

Def 9.5: Let M be a TM and w be an input string. An accepting computation history of M on w is a sequence of configurations C_1, C_2, \dots, C_n such that C_1 is the starting configuration of M on w , C_n is an accepting configuration, and each C_i legally follows from C_{i-1} , according to the rules of M .

A rejecting CH is defined the same way, but C_n is a rejecting configuration.

$ALL_{CFR} = \{ \langle n \rangle \mid n \text{ is a CFR, } L(n) = \Sigma^* \}$.

Theorem 9.13: ALL_{CFR} is undecidable.

I will just briefly describe one problem before the next lecture but then this will be the last lecture for this week this is going to be a rather lighter week but I just want you to want to give you something to think about so I will just state the main thing that I will the first thing that I will discuss next week.

So, this problem is called post correspondence problem and the reason for discussing this problem is that so far the problems that we have seen the undecidable problems that we have seen are of the type like is there a Turing machine that accepts this, is there a Turing machine that recognizes like it is a very abstract kind of situation so for the languages that we have seen that are undecidable are rather abstract.

This is a very simple problem that is very easy to describe you can even tell it to some third standard, fourth standard school going child and the problem is so easy to describe so I just want to describe the problem, so we will see the proof in the coming week, so given some dominoes, so the problem is this, given some dominoes we want to see whether can we list them in such a way there is a match, match means can we list them in some such a way that the top string and the bottom string are the same?

So, let maybe it is best described through an example so let us see this, so there are five dominoes, first one is a b in the top and a b a in the bottom then ba in the top and a b b in the bottom, b in the top and a b in the bottom, a b b in the top and b in the bottom, a in the top and b a b in the bottom, now I claim that the following listing here this constitutes a match so as you can see that I am using some dominoes more than once and it looks like I am using all

the dominoes at least once but even that is not necessary, we do not need to use all the dominoes and I am allowed to use dominoes more than once, yeah.

So, let us see so if you read the top string it is a b a b a b so here if you see a b a b a b so it is the same string then the rest is a b b a b b and from here it is a b b a b b so now the we are up to here in the bottom and we are up to here in the top the rest is b a b b b oh no I think there is a small error it is a b b divided the bottom is b so let us do it again so a b a b a b the first four dominoes top part is a b a b a b first two dominoes bottom part is a b a b a b so now up to fourth in the top up to second in the bottom.

Then a b b a b b up to sixth in the top a b b a b b up to fifth in the bottom, so now the rest is b a b b and b a b b so this con, so if you read the entire top string it is the same as the bottom string and this is what we mean by a match, so the question is this, given an instance an instance like this, is there some way or can we tell whether there is such a match or not that is a problem.

Given an instance is there a match or not, meaning match means is there a way to write the dominoes in some order with possible repeats we do not have to have repeats but we can have repeats in such a way that we get a match, this is the problem, so it is a very simple problem very easy to, simple meaning simple to describe, however this problem turns out to be undecidable, such a simple question you can tell it to some school going child but this problem turns out to be undecidable.

So, this is an sharp contrast with what we have seen before, the other problems is it A_{TM} does this Turing machine accept does this halt, is this regular and like Rice's Theorem and these are they all seem very abstract things but here we are we have something which is very clearly easily described and very concrete problem and surprisingly this problem turns out to be undecidable.

So, maybe you can just try, try to see to come up with strategies and you would see that you will not be able to create an algorithm for this so that may that may give you some indication of why this is indeed so there is no general algorithm that is what it means to be undecidable of course in some cases by inspection you can figure out a match in some cases by inspection you can rule out a match but there is no standard clearly well-defined approach that will lead us to telling lead us to clearly saying yes this is, this has a match or this does not have a

match. So, the question is, given this instance we have to figure out whether there is a match or not, yes or no, but it is an undecidable problem.

So, that completes this lecture, lecture number 42. This also completes week 8 lectures so in week 8 we have seen reductions, we have seen reductions we define reductions, we saw undecidable languages using reductions, then we saw Rice's theorem which was a template result that could show that many languages are undecidable like many languages that fall in that framework.

Then we saw computation histories and saw the high level overview that ALL_{CFG} or not high level or not that high level but somewhere medium level overview that ALL_{CFG} is undecidable and then I stated the post correspondence problem so it is called Post correspondence problem because it was invented by a mathematician called Post, so Post means that and I said that we will see the undecidability of post correspondence problem so also sometimes abbreviated as PCP next weekend also maybe applications for this and that is all I have in lecture 42 and also week 8. So, see you next week.