Unit - IV

Hierarchy of formal Lang & automata:

Recursive and recursive enumerable languages - unrestricted grammars - context sensitive grammars and languages - chomsky hierarchy.

Limit of algorithmic computation:

Problems that can't be solved by TM - undecidable problems for recursively enumerable languages - post correspondence problem - undecidable problems for CFL.

An overview of computational complexity:

Turing machine models and complexity - language families and complexity classes - complexity classes P and NP - some NP Problems - Polynomial time reduction - NP - Completeness.

Recursive and Recursive Enumerable languages.

Definition:

A Lang L is said to be RE if there exist a TM that accepts it. such that for every $w \in L$.

$$q_0 w \vdash_M x_1 q_f x_2 \quad q_f \in F$$

- TM - halt - string accepted

$w \notin L \rightarrow$ TM halts in a non-final state / never halts

L goes to infinite loop.

Definition: Recursive

* A Lang L on Σ is said to be recursive if there exist a TM M that accepts L and that halts on every w in Σ^+.

* A Lang: is recursive iff there exist a membership algo: for it.

→ Consider a TM M – that determines membership in a recursive Lang: L.

Consider another TM $\hat{M}$ – generates all strings in Σ^+ in proper order, consider

$w_1, w_2, \ldots$

$\hat{M}$ generates string $w_1, w_2 \ldots$

↳ input to M

↓

writes input to tape

if they are in L. ie. $w \in L$.

some $w_j \notin L$, machine M, started with w_j on its tape, may never halt & w never get in to L. that follow w_j in the enumeration.

$\hat{M}$ generate w_1 – M execute one move on it.

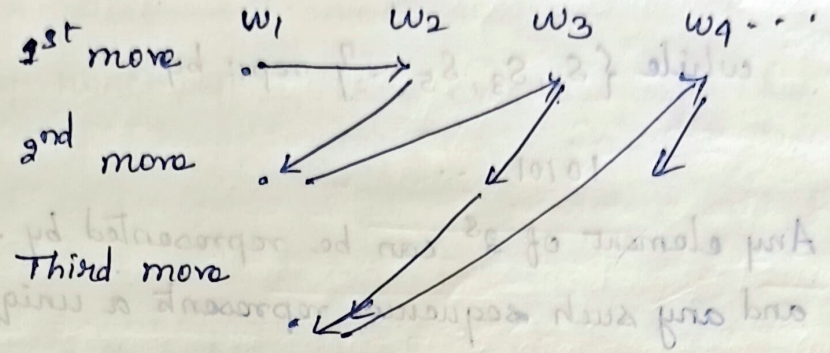
$\hat{M}$ " w_2 – M second move on w_1.

w_2 – one step on w_2

$w_n \ldots w_3 w_2 w_1$

M never get into an infinite loop.

* since $w \in L$ is generated by $\hat{M}$ & accepted by M in a finite number of steps



* Every Lang for which an enumeration procedure exists is RE.

* if $w \in L$, will eventually get a match, & the process can be terminated

$$\hat{M} = w_1 \; - - - \; w_n$$

$$enu = \hat{w}_1 \; - - - \; \hat{w}_n$$

Languages that are not RE:

Theorem:

Let S be an infinite countable set. Then its power-set 2^S is not countable.

Proof:

* Let $S = \{S_1, S_2, \ldots\}$

Then any element 't' of 2^S can be represented by a sequence of 0's and 1's, with a '1' in position 'i' iff S_i is in 't'.

Eg.

The set $\{S_2, S_3, S_6\}$ repn: by

01100100

while $\{S_1, S_3, S_5 \ldots\}$ repn: by

10101 ...

* Any element of 2^S can be represented by such a sequence, and any such sequence represent a unique element of 2^S.
* Suppose 2^S were countable

↳ its elements could be written in some order, say $t_1, t_2 \ldots$ and we could enter these into a table

t_1	(1)	0	0	0	0	...
t_2	1	(1)	0	0	0	...
t_3	1	1	(0)	1	0	...
t_4	1	1	0	(0)	1	...

RE { D → not acc ; ↳ $\bar{C}$ — acc

* In this table, take the elements in the main diagonal, and complement each entry, that is, replace 0 with 1, and vice versa.
* The elements are 1100, as the result.
* The new sequence along the diagonal repn: some element of 2^S say t_i for some i.
* But it cannot be t_1, because it differs from t_1 through S_1. For some reason it cannot be t_2, t_3 or any other entry in the enumeration.

* It involves the manipulation of the diagonal elements of a table, is called diagonalization.

Theorem:

For any nonempty Σ, there exist languages that are not RE.

Σ^* – no: of subset → every subset is a lang.
↳ not countable
↳ but enumeration is possible.
↳ RE
because Σ^* is infinite
otherwise it is not RE

Proof:

* A Lang. is a subset of Σ^*, and every such subset is a Lang:.
* Therefore, the set of all Lang: are exactly 2^{Σ^*}. since Σ^* is infinite.
* The set of all Lang: on Σ is not countable. But, the set of all TM can be enumerated, so the set of all RE Lang: is countable.
* There must be some lang: on Σ that are not RE.

A Lang: is not RE:

* A Lang: that can be described in a direct algorithmic fashion can be accepted by a TM → is RE
* The description of a Lang: that is not RE must be indirect.
↳ diagonalization theme.

Theorem:

There exists a RE Language whose complement is not RE.

Proof:

* Let $\Sigma = \{a\}$ – consider set of all TM with this input alphabet.
* A set is countable – its elements $M_1, M_2 \ldots$
* For each M_i – there is an associated RE lang: $L(M_i)$
* For each RE
  ↳ there is some TM accepts it.
* Consider a new lang: L:

  for each $i \geq 1$, the string a^i is in L iff $a^i \in L(M_i)$

  → The lang: L is well defined, since $a^i \in L(M_i)$ hence $a^i \in L$, must be either true or false.
* Next consider the complement of L;

  $$\bar{L} = \{a^i : a^i \notin L(M_i)\}$$

  it is well defined but, is not RE.
* Starting from the assumption that $\bar{L}$ is RE.
* If there is so, then there must be some TM, say M_k such that

  $$\bar{L} = L(M_k)$$

* Now consider the string a^k
  is it in L or $\bar{L}$?

  Suppose $a^k \in \bar{L}$

  $$a^k \in L(M_k)$$

  But $a^k \notin \bar{L}$

$101 \in L(M_i)$ – RE
$010 \notin L(M_i)$ – not RE
$a^i \in L(M_i)$ – RE
$\bar{L} = L(M_k)$ – not RE

* Alternatively, assume that a^k is in L,

Then $a^k \notin \bar{L}$ & implies that $a^k \notin L(M_k)$

We get that $a^k \in \bar{L}$

Conclude that $\bar{L}$ is not RE.

* To complete the proof, L is RE.

Use the known enumeration procedure for TM, given a^i, first find 'i' by counting the number of a's.

Consider the universal TM M_u - simulates the action of M on a^i

* If a^i is in L, the computation carried out by M_u will eventually halt.
* The combined effect of this is a TM that accepts every $a^i \in L$. Therefore, L is RE.
* This proof explicitly exhibits, a well defined lang: is not RE.

A Language that is RE but not recursive:

Theorem:

If a Lang: L and its complement $\bar{L}$ are both RE, then both Lang: are recursive. If L is recursive, then $\bar{L}$ is also recursive, and consequently both are RE.

Proof:

* If L and $\bar{L}$ are both RE, then there exist TM M and $\hat{M}$ that serve as enumeration procedures for L and $\bar{L}$, respectively.

L, $\bar{L}$ } RE — TM M, TM $\hat{M}$

* The first will produce

$w_1, w_2 \ldots$ — in L — generated & accepted by M

$\hat{w}_1, \hat{w}_2 \ldots$ — in $\bar{L}$ — accepted by $\hat{M}$

* Suppose we are given any $w \in \Sigma^+$.

  → First M generate W_1 and compare it with W.

* → If they are not the same, $\hat{M}$ generate W_2

  Then $\hat{M}$ generate $\hat{W}_2$, and so on.

* Any $w \in \Sigma^+$ will generated by either M or $\hat{M}$, will get a match.

* If the matching string is produced by M, W belongs to L, otherwise it is in $\bar{L}$.

* The process is a membership algorithm for both L and $\bar{L}$, so they are both recursive.

* Assume that L is recursive. Then there exists a membership algorithm for it.

* But this becomes a membership algorithm for $\bar{L}$ by simply complementing its conclusion.

* Therefore, $\bar{L}$ is recursive. Since any recursive lang: is RE, the proof is completed.

* We conclude that the family of RE lang: and the family of recursive lang: are not identical.

Unrestricted Grammars.

* The production rules were allowed to take any form, but various restrictions were later made to get specific grammar types.

* We take the general form and impose no restrictions we get unrestricted grammars.

Definition:

A grammar $G=(V, T, S, P)$ is called unrestricted if all the productions are of the form

$$u \rightarrow v,$$

where u is in $(V \cup T)^+$ and v is in $(V \cup T)^*$

$S \rightarrow aAbX$
$Ab \rightarrow a \mid \lambda$

* In an unrestricted grammar, essentially no conditions are imposed on the productions.
* Any number of variables and terminals can be on the left or right, and these can occur in any order.
* There is only one restriction: λ is not allowed as the left side of a production.

$\lambda \rightarrow aA$ ↳ no meaning & impossible

* Unrestricted grammars are much more powerful than restricted forms like the regular & CFG.
* Unrestricted grammars correspond to the largest family of langs, that is, unrestricted grammars generate exactly the family of RE langs.

Theorem:

The langs generated by an unrestricted grammar is RE.

Proof:

* The grammar in effect defines a procedure for enumerating all strings in the lang systematically.

eg. $S \Rightarrow w$, w is derived in one step.

* The set of productions of the grammar is finite, there will be finite number of such string.
* Next list all 'w' in L that can be derived in two steps

$$S \Rightarrow x \Rightarrow w.$$

* We can simulate these derivations on a TM and therefore, have an enumeration procedure for the lang:

* Hence it is RE

* The grammar generate strings by a well-defined algorithmic process, so the derivations can be done on a TM.

* Given a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ & want to produce a grammar G such that $L(G) = L(M)$.

* The computation of the TM can be described by the sequence of ID

$$q_0 w \vdash^* x q_f y \quad \text{— TM Derivation}$$

$$\Downarrow$$

$$q_0 w \Rightarrow x q_f y \quad \text{— Grammar derivation.}$$

* To construct a grammar the following properties are considered.

1. S can derive $q_0 w$ for all $w \in \Sigma^+$
2. $q_0 w \overset{*}{\Rightarrow} x q_f y$ is possible iff $q_0 w \vdash x q_f y$ holds
3. when a string $x q_f y$ with $q_f \in F$ is generated, the grammar generate/ transforms this string into the original w.

modify step 2 ↑ Trouble one.

* The complete sequence of derivations is then

$$S \Rightarrow q_0 w \Rightarrow x q_f y \Rightarrow w.$$

* The third step is troublesome one.

The grammar 'w' if it is modified during the second step?

$w \leftarrow x \leftarrow S$

→ The coded version originally has two copies of w.

The first is saved. (1)

Second is used in $q_0 w \Rightarrow x q_f y$. (2)

→ When a final configuration is entered, the grammar erases everything except the saved w.

→ To produce two copies of 'w' and to handle the state symbol of M, introduce the variables V_{ab} and V_{aib} for all $a \in \Sigma \cup \{\square\}$, $b \in \Gamma$, and all 'i' such that $q_0 \in Q$.

($a \in \Sigma \cup \{\square\}$; $V_{ab} \rightarrow \Gamma$; $V_{aib} \rightarrow$ state)

→ V_{ab} encodes the two symbols a and b,
V_{aib} encodes a and b, as well as state q_i.

* The first step can be achieved in the encoded form by

$$S \rightarrow V_{\square\square} S \mid S V_{\square\square} \mid T$$
$$T \rightarrow T V_{aa} \mid V_{aqa} \quad \text{for all } a \in \Sigma.$$

* These productions allow the grammar to generate an encoded version of any string $q_0 w$ with an arbitrary number of leading and trailing blanks.

* For the second step, for each transition of M,

$$\delta(q_i, c) = (q_j, d, R)$$

$$\Downarrow$$

$$V_{aic} V_{pq} \Rightarrow V_{ad} V_{pjq}$$

(i state, j state, d)

for all $a, p \in \Sigma \cup \{\square\}$, $q \in \Gamma$.

For each M,

$$\delta(q_i, c) = (q_j, d, L)$$

$V_{aic} V_{pq} \Rightarrow V_{pjq} V_{ad}$

$V_{pq} V_{aic} = V_{pjq} V_{ad}$ (V_{ad})

We include in G

$V_{pq} V_{aic} \rightarrow V_{pjq} V_{ad}$ for all $a, p \in \Sigma \cup \{\square\}$, $q \in \Gamma$.

* In the second step, M enters a final state, the grammar must then get free of everything except w, which is saved in the first indices of the V's

* For every $q_j \in F$, include

$$V_{ajb} \rightarrow a,$$

* For all $a, p \in \Sigma \cup \{\square\}$, $b \in \Gamma$.

This creates the first terminal in the string,

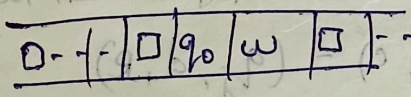
$$cV_{ab} \rightarrow ca,$$
$$V_{ab}c \rightarrow ac,$$

For all $a, c \in \Sigma \cup \{\square\}$, $b \in \Gamma$.

We need one more special productions

$$\square \rightarrow \lambda.$$

This last productions take care of the case when M moves outside that part of the tape occupied by the input w.



Representing all the tape region used. The extraneous blanks are removed at the end.

Example:

Let TM M contains

$Q = \{q_0, q_1\}$

$\Gamma = \{a, b, \square\}$

$\Sigma = \{a, b\}$

$F = \{q_1\}$

and $\left.\begin{array}{l}\delta(q_0, a) = (q_0, a, R) \\ \delta(q_0, \square) = (q_1, \square, L)\end{array}\right\}$ $L(aa^*)$

↳ $w = aa$

This machine accepts $L(aa^*)$

$w = aa$ $\quad q_0aa \vdash aq_0a\square \vdash aaq_0\square \vdash aq_1a\square$.

* To derive this string with G_1,

$S \Rightarrow SV_{\square\square} \Rightarrow TV_{\square\square} \Rightarrow TV_{aa}V_{\square\square} \Rightarrow V_{a0a}V_{aa}V_{\square\square}$

$q_0 \; aa\square\square$

* The last sentential form is the starting point for the part of the derivation that mimics the computation of the TM.

Right → $V_{a0a}V_{aa} \rightarrow V_{aa}V_{a0a}$

and

$V_{a0a}V_{\square\square} \rightarrow V_{aa}V_{\square0\square}$

which are specific instances

left at final | $V_{aa}V_{\square0\square} \rightarrow V_{a1a}V_{\square\square}$

} unrestricted grammar → TH

Then

$V_{a0a}V_{aa}V_{\square\square} \Rightarrow V_{aa}V_{a0a}V_{\square\square} \rightarrow V_{aa}V_{aa}V_{\square0\square} \Rightarrow V_{aa}V_{a1a}V_{\square\square}$

The last step

$V_{aa}V_{a1a}V_{\square\square} \Rightarrow V_{aa}aV_{\square\square} \Rightarrow V_{aa}a\square \Rightarrow aa\square \Rightarrow aa$

$V_{ab} \rightarrow a$, $a \le a$, $\square \rightarrow a$

$V_{aa} \rightarrow c$, $V_{ab} \rightarrow c$, $\square \rightarrow \square$

ac

Theorem:

For every RE L, there exists an unrestricted grammar G_1, such that $L = L(G_1)$.

Proof:

The construction described guarantees that

$x \vdash y$ then

$e(x) \Rightarrow e(y)$

$e(x)$ – the encoding of a string according to the given convention

$e(q_0 w) \Rightarrow e(y)$ iff

$q_0 w \vdash y$

Context Sensitive grammars and Languages.

Definition

A grammar $G = (V, T, S, P)$ is said to be context sensitive if all productions are of the form

$x \rightarrow y$

where $x, y \in (V \cup T)^+$ and

$|x| \leq |y|$

* The length of the successive sentential forms can never decrease.
* All productions are of the form

$xAy \rightarrow xvy$

This is equivalent to

$A \rightarrow v$

Content-sensitive Lang: & LBA:

Definition

A Lang: is said to be content sensitive if there exists a CSG G_1, such that

$$L = L(G_1) \text{ or}$$

$$L = L(G_1) \cup \{\lambda\}$$

* A CSG can never generate a lang: containing the empty string. eg. $L = \{a, b, \ldots\}$
* Every CFL without λ can be generated by a special case of a CSG.
* The family of CFL is a subset of the family of CS-lang.

$$\underset{(3)}{RG} \subset \underset{(2)}{CFG} \subset \underset{(1)}{CSG} \subset \underset{(0)}{URG}$$

Example:

The Lang: $L = \{a^n b^n c^n : n \geq 1\}$ is a CSL. show this by exhibiting a CSG for the lang:

$$S \to \underset{n=1}{abc} \mid a\underset{n>1}{A}bc$$

$$Ab \to bA$$

$$Ac \to Bbcc$$

$$bB \to Bb,$$

$$aB \to aa \mid aaA$$

$$w = a^3b^3c^3 = aaabbbccc$$

$$S \Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc$$

$$\Rightarrow aBbbcc \Rightarrow aaAbbcc \Rightarrow aabAbcc$$

$$\Rightarrow aabbAcc \Rightarrow aabbBbccc$$

$$\Rightarrow aabBbbccc \Rightarrow aaBbbbccc$$

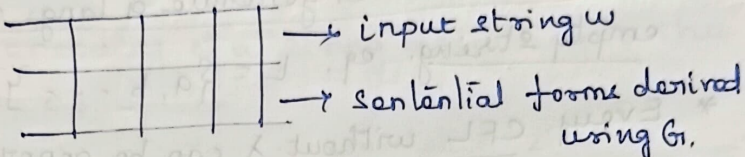
$$\Rightarrow aaabbbccc.$$

Theorem:

For every context-sensitive lang: L not including λ, there exists some LBA M such that $L = L(M)$.

Proof:

* If L is context sensitive, then there exists a CSG for $L - \{\lambda\}$.
* Derivations in this grammar simulated by a LBA.
* The LBA have two tracks



* No possible sentential form can have length greater than $|w|$.
* A LBA is nondeterministic, the correct production can always be guessed and that no unproductive alternatives have to be pursued.
* The computation can be carried out using space except that originally occupied by w, is done by LBA.

Theorem:

If a lang: L is accepted by some LBA M, then there exists a CSG that generates L.

Proof:

$$\square \longrightarrow \lambda$$

* Generally, this production can be omitted. But, it is necessary only when the TM moves outside the bounds of the original input.
* The grammar obtained by the construction without this unnecessary production is non-contracting, completing the argument.

Relation between Recursive and content sensitive lang:

Theorem:

Every content sensitive lang. L is recursive.

Proof:

* consider the CSL L with an associated CSG 'G' at a derivation of w

$$S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_n \Rightarrow w.$$

* All sentential forms in a single derivation tree different, that is $x_i \neq x_j$ for all $i \neq j$.
* The number of steps in any derivation is a bounded function of $|w|$.

$$|x_j| \leq |x_{j+1}|$$

x y $|x| \leq |y|$

because G is non-contracting.

* To add is that there exist some m, depending only on G and w, such that

$$|x_j| < |x_{j+m}|$$

* For all j, with $m = m(|w|)$ a bounded function of $|V \cup T|$ and $|w|$.

↳ implies that there are only a finite number of strings of a given length. The length of a derivation of $w \in L$ is at most $|w|\, m(|w|$

* This gives us immediately a membership algorithm for L. we check all derivations of length upto $|w|\, m(|w|)$.
* Since the set of productions of G is finite, there are only a finite number of these.
* If any of them give w, then $w \in L$, otherwise it is n

Theorem:

There exists a recursive lang: that is not context sensitive

Proof:

* Consider the set of all CSG on $T = \{a, b\}$
* Each grammar has a variable set of the form

$$V = \{V_0, V_1, V_2, \ldots\}$$

* Every CSG is completely specified by its productions

$$x_1 \to y_1$$
$$x_2 \to y_2$$
$$\vdots$$
$$x_m \to y_m$$

* Now apply the homomorphism

$$h(a) = 010,$$
$$h(b) = 01^20$$
$$h(\to) = 01^30$$
$$h(;) = 01^40$$
$$h(V_i) = 01^{i+5}0$$

* Let us introduce a proper ordering on $\{0, i\}^+$, we can write strings in the order w_1, w_2 etc.
* A given string w_j may not define a CSG, if it does, call the grammar G_j.

$$L = \{w_i : w_i \text{ defines CSG } G_i \text{ and } w_i \notin L(G_i)\}$$

* L is well defined and is in fact recursive.
* Given w_i check if it is CSG G_i, if not, then $w_i \notin L$.

* If the string does define a grammar, then $L(G_i)$ is recursive, & use the membership algo: to find if $w_i \in L(G_i)$. If it is not, then w_i belongs to L.

* If we assume that $w_j \in L(G_j)$, then w_j is not in L. But $L = L(G_j)$ so we have a contradiction.

* If we assume that $w_j \notin L(G_j)$ then by definition $w_j \in L$ and we have another contradiction.

* We must therefore conclude that L is not context sensitive.

Example:

Find CSG for the following lang $L = \{a^m b^n c^m d^n \mid n, m \geq 1\}$

Answer:

$S \rightarrow aABB \mid aB$

$A \rightarrow aAX \mid aX$

$B \rightarrow bBd \mid bYd$

$Xb \rightarrow bX$

$XY \rightarrow Yc$

$Y \rightarrow c$

$m = 2 \quad n = 3$

$w = aabbbccddd$

$S \Rightarrow aAB \Rightarrow aaXB \Rightarrow aaXbBd \Rightarrow aaXbbBdd \Rightarrow$

$aaXbbbYddd \Rightarrow aaXbbYddd \Rightarrow aabXbbYddd$

$\Rightarrow aabbXbYddd \Rightarrow aabbbXYddd \Rightarrow aabbbYcdd$

$\Rightarrow aabbbccddd.$

Find CSG for the following languages

$$L = \{ a^{n+1} b^n c^{n-1} \mid n \geq 1 \}$$

Answer

$$S \rightarrow \underset{n>1}{aaAbbc} \mid \underset{n=1}{aab}$$

$$Ab \rightarrow bA$$

$$Ac \rightarrow Bcc$$

$$bB \rightarrow Bb$$

$$aB \rightarrow aaAb$$

$$aA \rightarrow aa.$$

$$w = aaabbc$$

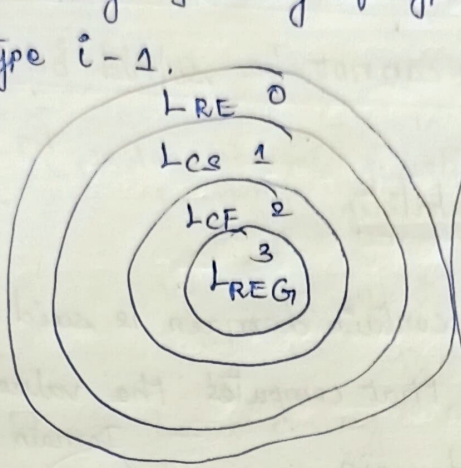
$$S \Rightarrow aaAbbc \Rightarrow aaabbc$$

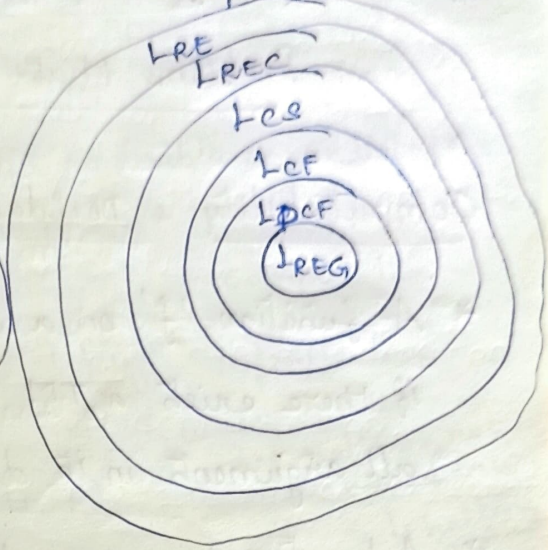
Chomsky Hierarchy.

* The relationship b/w lang: exhibiting by the chomsky Hierarchy.
* chomsky a founder of formal lang:, provided an initial classification into four lang: types type 0 to 3.

Type of Lang:	Grammar	Language.
i= 0 0	unrestricted grammar	Recursive enumerable lang (TM & LBA)
i= 1 1	Content-sensitive (CSG)	Content-sensitive Lang. (TM & LBA)
i=2 2	content free grammar (CFG)	CFL (PDA)
i=3 3.	Regular grammar	regular lang. (FA)

* Each lang: family of type 'i' is a proper subset of type i-1.

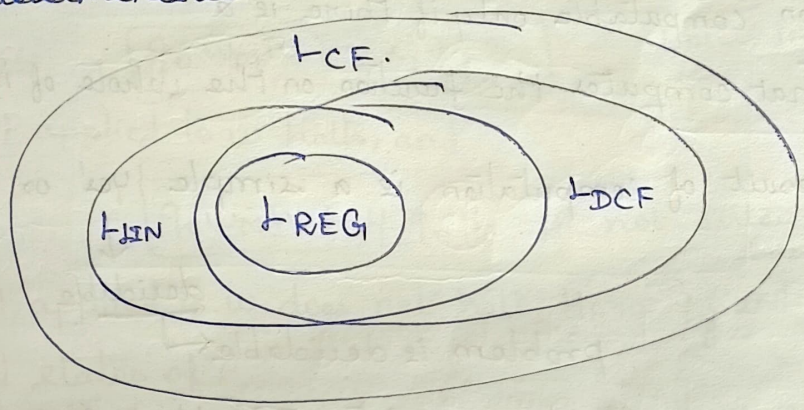




Example:

* The CFL $L = \{w : n_a(w) = n_b(w)\}$ show that it is deterministic, but not linear.

* The Lang: $L = \{a^n b^n\} \cup \{a^n b^{2n}\}$ is linear, but not deterministic



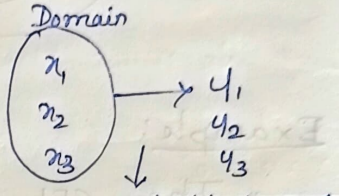
Limitations of Algorithmic Computation

Problems that cannot be solved by TM

Computability & Decidability

* A function 'f' on a certain domain is said to be computable if there exists a TM that computes the value of 'f' for all arguments in its domain.

Eg



* A function is uncomputable if no such TM exists.

solvable by TM & computable
otherwise uncomputable.

* There may be a TM that can computation 'f' on part of its domain, but call the function computable only if there is a TM that computes the function on the whole of its domain.

* The result of computation is a simple 'yes' or 'no'.

↓ decidable ↓ undecidabl

problem is decidable ←

if there exists a TM that gives the correct answer for every statement in the domain of the problem.

* Problem may be decidable on some domain but not on another.

TM Halting Problem:

Given the description of a TM M and an input w, does M, when started in the initial configuration q_0w, perform a computation that eventually halts?

Consider TM — $M_1 \dots M_n$) apply M on W.
String — $w_1 \dots w_n$ ↙

TM - M ↳ solve w from q_0 - initial state ↳ reach final & halt.

unable to predict the length of the computation ↓ infinite loop.

W ↓ $w_1 w_2 \dots w_n$ M_1 $M_2 \dots M_n$ if any one not enter into halts

↳ unable to predict TM halts or not based on w. ↳ halting problem.

Definition:

Let W_M — be a string

$TM = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$

Let W be a string in M's alphabet.

W_M and W are encoded as a string of 0's and 1's.

* A solution to a halting problem is a TM H, which for any W_M and W performs the computation

$$q_0 w_M w \vdash x_1 q_f x_2$$ — Enter into halt

* If M applied to w halts, and

$$q_0 w_M w \vdash y_1 q_n y_2$$ — not enter into halt but q_n is final.

* If M applied to w does not halt. Here q_y and q_n are both final states of H.

Theorem:

(*) The halting problem is an undecidable problem.

Proof:

* Assume that there exists an algorithm and some TM H, that solves the halting problem.
* The input to H will be the string $W_H W$.
* The requirement is then given any $W_H W$, the TM H will halt with either a yes or no answer.
* This can be achieve that by H halt in one of two corresponding final states say q_y or q_n.

* If H is started q_0 with input $w_M w$, it will eventually halt in state q_y or q_n.

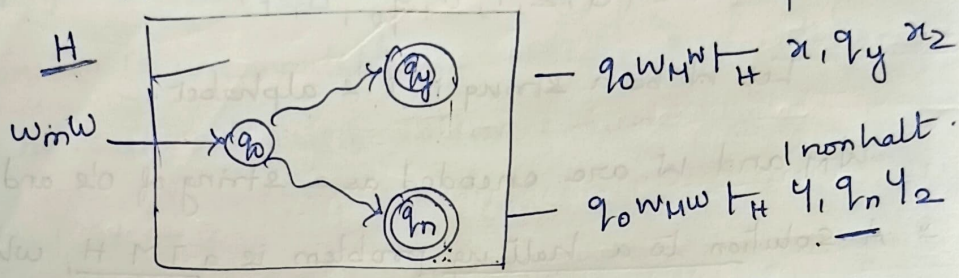
* H is operated by the following rules

$$q_0 w_M w \vdash_H x_1 q_y x_2$$

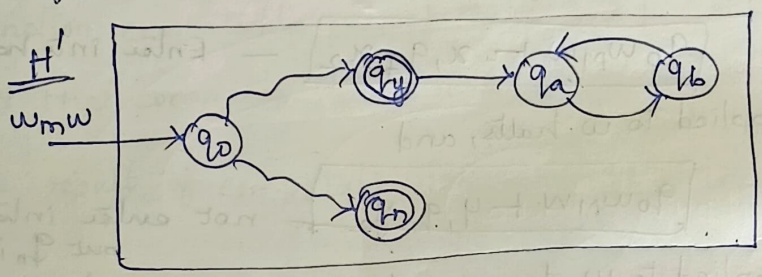
* If M applied to w halts and

$$q_0 w_M w \vdash_H y_1 q_n y_2$$

* If M applied to w does not halt.



* Modify H to produce a TM H' with the structure as



* With the added states to convey that the transitions between state q_y and the new states q_a and q_b are to be made, regardless of the tape symbol remains unchanged

* Comparing H and H' where H reaches q_y and halts, the modified machine H' will enter an infinite loop.

* The action of H'

$$q_0 w_M w \vdash_{H'} \infty$$

* If M applied to w halts, and

$$q_0 w_M w \vdash_{H'} y_1 q_n y_2$$

* If M applied to w does not halt.

* Construct

$$H \rightarrow H' \rightarrow \hat{H}$$

$$q_0 w_M \vdash_{\hat{H}} q_0 w_M w_M \vdash_{\hat{H}} \infty$$

$$H \rightarrow H \rightarrow H' \rightarrow \hat{H}$$
$$\searrow NH$$

If M applied to w_M halts, and

$$q_0 w_M \vdash_{\hat{H}} q_0 w_M w_M \vdash_{\hat{H}} y_1 q_n y_2.$$

M applied to w does not halt.

Identifying M with $\hat{H}$

$$q_0 \hat{w} \vdash_{\hat{H}} \infty$$

If $\hat{H}$ applied to $\hat{w}$ halts

$$q_0 \hat{w} \vdash_{\hat{H}} y_1 q_n y_2$$

If $\hat{H}$ applied to $\hat{w}$ does not halt.

→ To solve the halting problem, H had to start and end in every specific configuration.

Theorem:

If the halting problem were decidable, then every RE Lang: would be recursive. Consequently, the halting problem is undecidable.

Proof:

* Let L be a RE lang on Σ

  M be TM that accept L.

* Let H be the TM that ~~halts~~ solves the halting problem.

* We construct from the following procedure.

1. Apply H to $w_M w$.

   If H = No, $w \notin L$

2. If H = Yes, apply M to w.

   But M must halt, so it tell $w \in L$ or $w \notin L$.

   ↳ its says L is recursive

   but already known that L is R.E, not recursive

   ↓

   Says H is not exist, hence halting problem is undecidable.

Reducing one undecidable problem to another:

Problem A → reduced to problem B.

↓

if its decidable, A is also decidable.

If A is undecidable, B is also undecidable.

State Entry Problem:

Given any TM M and any $q \in Q$, $w \in \Sigma^*$, decide whether or not the state q is ever entered when M is applied to w. This problem is undecidable.

* To reduce the halting problem to the state entry prob

↓

Solved by an algorithm A.

↓

Use this algo. to solve the halting problem

eg. consider

M & w

↳ its modified TM is $\hat{M}$

↳ halts in state q iff M halts.

if M halts ← because we modify (q_i, a)

To get $\hat{M}$, change every undefined δ to

$$\delta(q_i, a) = (q, a, R)$$

↳ final state.

Apply state-entry algo: A to $(q, \hat{M}, w)$

↳ if A = Yes – state q entered halts (M, w)

A = No – (M, w) not halt.

If state-entry problem – decidable.

↳ halting problem also decidable.

But halting problem is undecidable.

↳ state-entry problem is undecidable.

Example:

Show that there is no algorithm for deciding if any two TM M_1 and M_2 accept the same lang.

Answer:

* Given (M, w) → its enter into halts when $\hat{M}$ accept some simple lang;
  ↳ modify to $\hat{M}$,

* M checking the input and remembering whether the input was a.

  ↓

  then M carryout normal computations when it halts – check if the input was 'a' – accept otherwise reject it.

  ↙

  when M halts $\hat{M}$ accepts {a} then construct a new m/c M_1,

& check $L(\hat{M}) = L(M_1)$ → if $L(\hat{M}) = L(M_1)$ then M hal

if $L(\hat{M}) \neq L(M_1)$ → M does not halt

Example:

Is the halting problem solvable for dpda, that is, given a pda can always predict whether or not the automaton will halt on input w?

Answer:

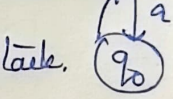
* Yes, this is decidable.

The only way to enter into infinite loop is λ.

* find out if there is a sequence of λ transitions that

  (i) Eventually produce the same state & stack top

  (ii) Do not decrease the length of the stack.



↳ if there is no such sequence, then dpda must halt.

if such sequence is there.

↳ finds its reachable or not.

↳ needs only finite number of moves.

↓

its predictable.

Undecidable Problems for RE languages.

orem:

Let G be an unrestricted grammar. Then the pro
ermining that $L(G) = \phi$ is undecidable.

f:

* Modify the TM M as follows.

(i) M first save its input on some special part of its tape. Then, whenever it enters a final state, it checks its saved input and accept it iff it is w.

(ii) By changing δ

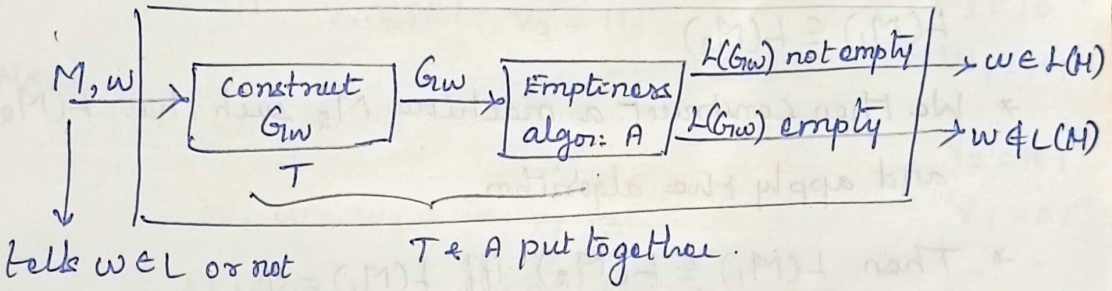
Creating for each w a machine M_w such that $L(M_w) = L(M) \cap \{w\}$

↓

Then construct its corresponding grammar G_{Mw}.

$L(G_{Mw})$ – is non empty iff $w \in L(M)$

* Assume that there exists an algorithm A for deciding whether or not $L(G_1) = \phi$



↳ if such a TM exists we have an membership algo: for RE lang:

$\therefore L(G_1) = \phi$ is not decidable.

$L(aa^*b^*)$

$L(ba^*)$ { b, ba, baa, baaa

Example:

Show that for an arbitrary TM M with $\Sigma = \{a, b\}$ the problem "L(M) contains two different strings of the same length" is undecidable.

Proof: Answer

* When $\hat{M}$ reaches a halting configuration, it will be modified to accept the two strings 'a' and 'b'.

* For this the initial input is saved and at the end of the computation compared with 'a' and 'b', accepting only these two strings.

* Thus, if (M, w) halts, $\hat{M}$ will accept two strings of equal length, otherwise $\hat{M}$ will accept nothing.

Example:

Let M_1 and M_2 be arbitrary TM. show that the problem $L(M_1) \subseteq L(M_2)$ is undecidable.

Answer:

* Suppose we had an algorithm to decide whether or not $L(M_1) \subseteq L(M_2)$
* We then construct a machine M_2 such that $L(M_2) = \phi$ and apply the algorithm.
* Then $L(M_1) \subseteq L(M_2)$ iff $L(M_1) = \phi$.

Post Correspondence Problem. (PCP)

* In a halting problem, many instances work with the halting problem directly, and it is convenient to establish some intermediate results that bridge the gap between the halting problem and other problems.
* These intermediate results follow from the undecidability of the halting problem. One such intermediate result is the PCP.

* The PCP can be stated as follows:

Given two sequences of n strings on some alphabet Σ, say

$$A = w_1, w_2 \dots w_n$$

and

$$B = v_1, v_2 \dots v_n$$

* There exist a PC solution for pair (A, B) if there is a nonempty sequence of integers $i, j \dots k$ such that

$$\boxed{w_i w_j \dots w_k = v_i v_j \dots v_k}$$

Example.

Take A and B

$w_1 = 11$, $w_2 = 100$, $w_3 = 111$

$v_1 = 111$, $v_2 = 001$, $v_3 = 11$

Now

PCP solution is

$$w_1 w_2 w_3 = v_1 v_2 v_3$$

$$\underline{11100}\,\underline{111} = \underline{111}\,\underline{00}\,\underline{11}$$

8 8

Now

$w_1 = 00$ $w_2 = 001$ $w_3 = 1000$

$v_1 = 0$ $v_2 = 11$ $v_3 = 011$

$$w_1 w_2 w_3 = v_1 v_2 v_3$$

$$\underline{00\,001\,1000} \neq \underline{0\,11\,011}$$

9 $\neq$ 6

Hence no PCP solution.

string composed of A is longer than B.

$$\underline{10\,101} \quad \underline{101\,01}$$
$$w_2 w_1 = v_2 v_1$$
$$\text{I/P} \qquad \text{O/P}$$

PC Solution

(2, 1)

$w_2 = 10$

$w_1 = 101$

$v_2 = 101$

$v_1 = 01$

1, 2, 3

2, 1, 3

3, 1, 2

$\boxed{k_1} w_1 w_2 = \boxed{Y} v_1 v_2$

* In specific instances, we may be able to show by explicit construction that a pair (A, B) permits a PC solution or may be able to argue, that no such solution can exist.
* No algo for deciding this question under all circumstances. Hence PCP is undecidable.
* Break the process into two parts, modified PCP.

The pair (A, B) has a modified PCP (MPC solution) if there exists a sequence of integers $i, j \cdots k$ such that

$$w_1 w_i w_j \cdots w_k = v_1 v_i v_j \cdots v_k$$

* In a MPC problem, the first element of the sequence A and B play a special role.
* An MPC solution must start with w_1 on the left side and v_1 on the right side.
* If there exists an MPC solution, then there is also a PC solution, but the converse is not true.
* This MPC solution is also an undecidable problem.

Procedure unrestricted grammar to string w.

A (w)	B (v)	
FS⇒	F	F is a symbol not in VUT
a	a	$a \in T$
V_i	V_i	$V_i \in V$
E	⇒wE	E is a symbol not in VUT
y_i	x_i	$x_i \to y_i$ in P
⇒	⇒	

Example:

Take an unrestricted grammar

$S \rightarrow aABb \mid Bbb$

$Bb \rightarrow C$

$AC \rightarrow aac$

Take $w = aaac$.

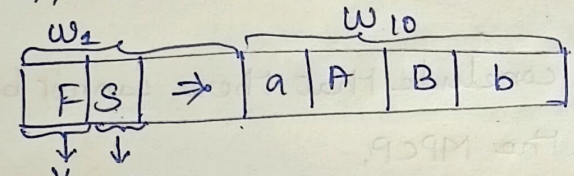
follow the above construction to generate a string

$w = aaac$ is in $L(G_1)$

$S \Rightarrow aABb \Rightarrow aAC \Rightarrow aaac.$

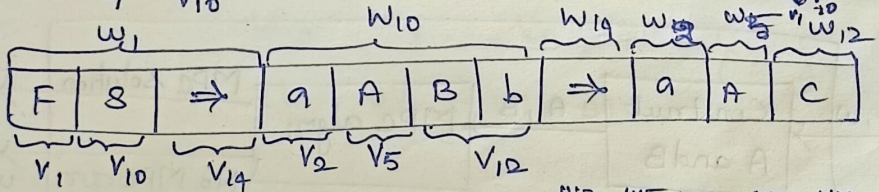
i	w_i	v_i
1	FS⇒	F
2	a	a
3	b	b
4	c	c
5	A	A
6	B	B
7	C	C
8	S	S
9	E	⇒aaacE
10	aABb	S
11	Bbb	S
12	C	Bb
13	aac	AC
14	⇒	⇒

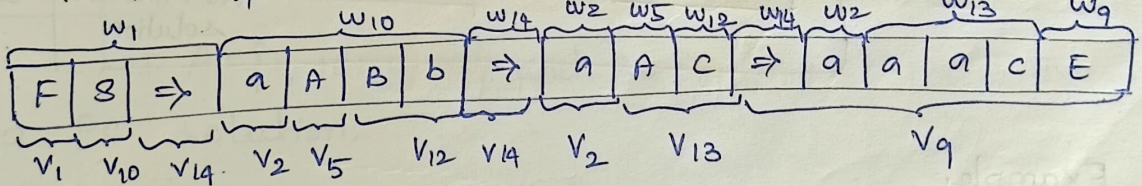
To construct an MPC solution, we must start with w_1, that is FS⇒,



$w_1 w_{10} = v_1 v_{10}$

FS⇒ aABb





* The string contains S, so to match it have to use v_{10} or v_{11}. In this instance, use v_{10} this brings in w_{10}, leading us to the second string in the partial derivation.
* After several more steps, that the string $w_1 w_i w_j \cdots w_k$ is always longer than the corresponding string $v_1 v_i v_j \cdots v_k$ and the first is exactly one step ahead in the derivation

Theorem:

The MPCP is undecidable.

Proof:

* Given any unrestricted grammar $G=(V, T, S, P)$

  & $w \in T^+$

  Construct the sets A & B

  The pair (A, B) has an MPC solution

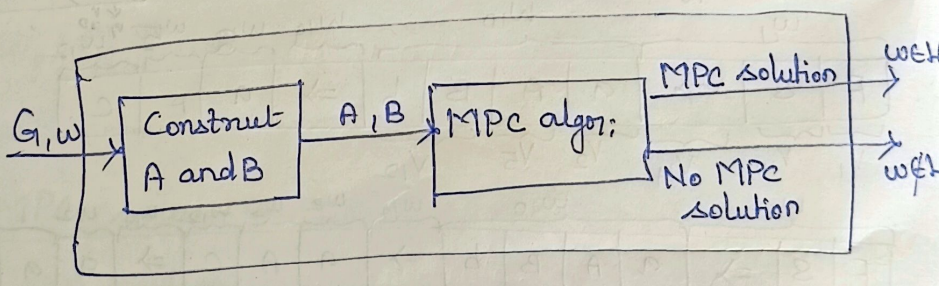
  iff $w \in L(G)$.

* Suppose the MPC problem is decidable.

  We can then construct an algo: for the membership problem of G.

* An algorithm for constructing A from B from G and w clearly exists, but a membership algo: for G and w does not.

* We must therefore conclude that there cannot be any algo: for deciding the MPCP.



Example:

Let $A = \{001, 0011, 11, 101\}$ and $B = \{01, 111, 111, 010\}$. Does the pair (A, B) have a PC solution? Does it have an MPC solution?

Answer:

* A PC solution is $w_3 w_4 w_1 = v_3 v_4 v_1$

  $(3, 4, 1)$ – PC solution

* There is no MPC solution because one string could have a prefix 001 and the other 01.

$$\underline{111}\ \underline{01}\ \underline{001} = \underline{111}\ \underline{01}\ \underline{001}$$
$$w_3\ w_4\ w_1 \quad v_3\ v_4\ v_1$$

$$\boxed{w_1}\ w_3\ w_4\ w_1 = \boxed{v_1}\ v_3\ v_4\ v_1$$
$$001 \quad \times \quad 01$$

Example:

Let $A=\{10, 00, 11, 01\}$ and $B=\{0, 001, 1, 101\}$ There exists a PC solution.

$$10\ \ 00\ \ 11\ \ 01 \qquad 0\ \ 001\ \ 1\ \ 101$$
$$w_1\ \ w_2\ \ w_3\ \ w_4 \qquad v_1\ \ v_2\ \ v_3\ \ v_4$$

Answer

$$w_2\ \ w_3 = v_2\ \ v_3 \quad (2,3)$$
$$00\ \ 11 = 00\ \ 11$$

$$w_3\ \ w_4 = v_3\ \ v_4$$
$$11\ \ 01 = 11\ \ 01 \quad (3,4)$$

Example:

Let $A=\{1, 10111, 10\}$ and $B=\{111, 10, 0\}$ There exists a PC solution.

Answer:

$$w_2\ \ w_1\ \ w_1\ \ w_3 = v_2\ \ v_1\ \ v_1\ \ v_3$$
$$10\ \ 111\ \ 1\ \ 10 = 10\ \ 111\ \ 1\ \ 10$$

$$(2, 1, 1, 3)$$

Example:

Let $A = \{10, 011, 101\}$

$B = \{101, 11, 011\}$ There exists a PC solution or not.

Answer:

$w_1\ w_2\ w_3 \neq v_1\ v_2\ v_3$

$10\ 0\ 11\ 101 = 101\ 11\ 011$ } does not have the PC solution.

$w_1\ w_3\ w_2 \neq v_1\ v_3\ v_2$

Example:

Let $A = \{10, 110, 11\}$ and $B = \{10, 11, 011\}$ There exists a MPC solution.

Answer:

$w_1\ w_i \cdots w_n = v_1\ v_i \cdots v_n$.

$w_1\ w_2\ w_3 = v_1\ v_2\ v_3$

$10\ 110\ 11 = 10\ 11\ 011$

Theorem:

The PC problem is undecidable.

Proof:

* If the PC problem were decidable, then the MPC would be decidable.
* The given sequences

$A = w_1\ w_2\ w_3 \cdots w_n$

$B = v_1\ v_2\ v_3 \cdots v_n$ on some alphabet Σ.

* Then introduce new symbols $\aleph$ and $\S$ and the new sequences

$$C = y_0 y_1 y_2 \cdots y_{n+1}$$

$$D = z_0 z_1 z_2 \cdots z_{n+1}$$

Defined as follows.

For $i = 1, 2 \ldots n$

$$\begin{cases} y_i = w_{i1} \aleph w_{i2} \aleph \cdots w_{im} \aleph \\ z_i = \aleph v_{i1} \aleph v_{i2} \cdots v_{ir} \end{cases}$$

where

w_{ij} } j^{th} letter of w_i and v_i
v_{ij} }

$$m_i = |w_i|$$

$$r_i = |v_i|$$

$\aleph$ – aleph

y_i is created from w_i by appending $\aleph$ to each character

z_i – obtained by prefixing each char. of v_i with $\aleph$.

* To complete the definition of C and D,

take

$$\begin{cases} y_0 = \S y_1 \\ y_{n+1} = \S \\ z_0 = z_1 \\ z_{n+1} = \S \end{cases}$$

$W_1 \cdots = V_1$

$y_i = w_{i1} \aleph w_{i2} \cdots w_{im} \aleph$

$z_i = \aleph v_{i1} \aleph v_{i2} \cdots v_{ir}$

* Consider a pair (C, D) and suppose it has a PC solution.

* Because of the placement of $\aleph$ and $\S$ such a solution must have y_0 on the left and y_{n+1} on the right end.

$w_{i_1} w_{i_2} \cdots w_{i_j} \cdots w_{i_k} \cdots = v_{i_1} v_{i_2} \cdots v_{i_j} v_{i_k} \cdots$

ignoring the symbols.

$w_i w_j \cdots w_k = v_i v_j \cdots v_k$.

$\therefore$ PC $\Rightarrow$ MPC is there.

so that the pair (A, B) permit an MPC solution.

Undecidable Problems for CFL.

* The PC problem is a convenient tool for studying undecidable questions for CFL.

Theorem:

There exist no algorithm for deciding whether any given CFG is ambiguous.

Proof:

* Consider the sequence of strings

$$A = (w_1, w_2 \cdots w_n)$$
$$B = (v_1, v_2 \cdots v_n)$$ } over Σ.

* Consider new set of distinct symbols $a_1, a_2 \cdots a_n$

$$\{a_1, a_2 \cdots a_n\} \cap \Sigma = \phi$$

and consider the two lang:

$$L_A = \{w_i w_j \cdots w_k w_l a_k a_l \cdots a_j a_i\}$$

& $$L_B = \{v_i v_j \cdots v_k v_l a_k a_l \cdots a_j a_i\}$$

* Now look at the CFG.

$$G_1 = (\underbrace{\{S, S_A, S_B\}}_{V}, \underbrace{\Sigma \cup \{a_1, a_2 \ldots a_n\}}_{T}, P, S)$$

Production P is the union of two subsets

P_A : $S \to S_A$

$S_A \to w_i S_A a_i \mid w_i a_i \quad i = 1, 2 \ldots n$

P_B : $S \to S_B$ $\qquad a S_a \mid a a$

$S_B \to v_i S_B a_i \mid v_i a_i \qquad i = 1, 2, \ldots n$

Take

$$G_{1A} = (\{S, S_A\}, \Sigma \cup \{a_1, a_2 \ldots a_n\}, P_A, S)$$

$$G_{1B} = (\{S, S_B\}, \Sigma \cup \{a_1, a_2 \ldots a_n\}, P_B, S);$$

Then

$$\left.\begin{aligned} L_A &= L(G_{1A}) \\ L_B &= L(G_{1B}) \end{aligned}\right\} \quad L_{G_1} = L_A \cup L_B$$

$$L(G_1) = L_A \cup L_B \qquad L_{AB} = S \to S_A \mid S_B$$

G_{1A} & G_{1B} are themselves unambiguous

* If a given string in $L(G_1)$ ends with a_i then its derivation with grammar G_{1A} must have started with

$$S \Rightarrow w_i S a_i.$$

* Thus, if G_1 is ambiguous it must be because there is a 'w' for which there are two derivations

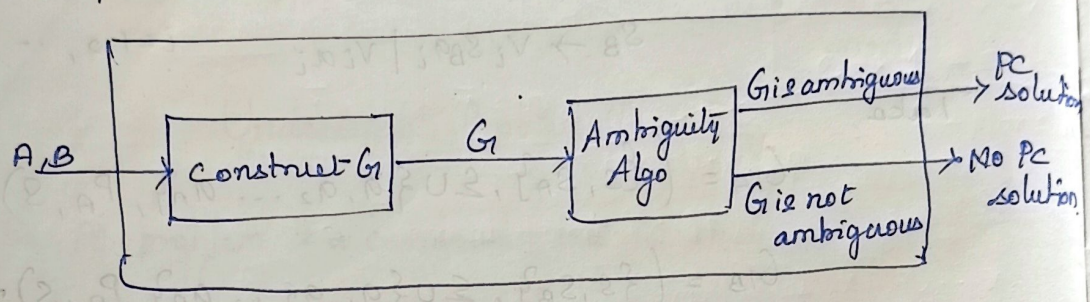
$$S \Rightarrow S_A \Rightarrow w_i S_A a_i \Rightarrow w_i w_j \ldots w_k a_k \ldots a_j a_i = w$$

$$S \Rightarrow S_B \Rightarrow v_i S_B a_i \Rightarrow v_i v_j \ldots v_k a_k \ldots a_j a_i = w.$$

* If G is ambiguous, then the PC problem with the pair (A, B) has a solution.

* If G is unambiguous, then the PC problem cannot have a solution.

* If there existed an algorithm for solving the ambiguity problem, we could adapt it to solve the PC problem.



* But, there is no algo: for the PC problem, we conclude that the ambiguity problem is undecidable.

Theorem:

There exists no algo: for deciding whether or not $L(G_1) \cap L(G_2) = \phi$ for arbitrary CFG G_1 and G_2.

Proof:

Take G_1 as G_A

G_2 as G_B

Suppose that $L(G_A)$ and $L(G_B)$ have a common element, that is

$$S_A \overset{*}{\Rightarrow} w_i w_j \cdots w_k a_k \cdots a_j a_i$$

$$S_B \overset{*}{\Rightarrow} v_i v_j \cdots v_k a_k \cdots a_j a_i$$

* Then the pair (A, B) has a PC solution.

If the pair does not have a PC solution, then $L(G_{1A})$ and $L(G_{1B})$ cannot have a common element.

* Conclude that $L(G_{1A}) \cap L(G_{1B})$ is non empty iff (A, B) has a PC solution. This reduction proves the theorem.

Some known undecidable problems are.

1) if G_1 & G_2 CFG, $L(G_1) = L(G_2)$?
2) if G_1 & G_2 CFG, $L(G_1) \subseteq L(G_2)$?
3) if G_1 is CFG, $L(G_1)$ regular?
4) if G_1 is CFG, G_2 is regular, $L(G_2) \subseteq L(G_1)$?
5) if G_1 is CFG, G_2 is regular, $L(G_1) \cap L(G_2) = \phi$?

Turing Machine
Models and complexity

* The efficiency of a computation can be affected by
  1. Number of tapes of the machine
  2. Whether the m/c is deterministic or not.
* Nondeterministic solutions are more efficient than deterministic solutions.

Example:

The satisfiability problem (SAT) plays an important role in complexity theory.

* A Logic or boolean constant or variable is take an exactly two values, true(1) or false(0).

* Boolean operators are used to combine boolean constants and variables into Boolean expressions.

* The simplest boolean operators are (∨) and (∧)

$$0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1$$

$$0 \vee 0 = 0$$

and the ∧

$$0 \wedge 0 = 0 \wedge 1 = 1 \wedge 0 = 0$$

$$1 \wedge 1 = 1$$

negation denoted by bar

$$\bar{0} = 1 \quad \& \quad \bar{1} = 0$$

T(1), F(0)

∨, ∧, ¬, (,)

e ⇒ 1 - satie

0 - Nonsati

Now consider the Boolean expressions in conjunctive normal form (CNF)

* In this CNF expression created by variables $x_1, x_2 \ldots x_n$ starting with

$$e = t_i \wedge t_j \wedge \ldots \wedge t_k$$

E → T

Term $t_i, t_j \ldots t_k$ are created by or-ing together variables and their negation

$$t_i = s_l \vee s_m \vee \ldots \vee s_p$$

E → T

t_1, t_2

$t_1 \wedge t_2$

$s_l \vee s_m$

* Now the SAT problem is stated as:

Given a satisfiable expression 'e' in CNF, find an assignment of values to the variables $x_1, x_2 \ldots x_n$ that will make the value of 'e' True

* For a specific case, look at

$$e_1 = \underbrace{(\overbrace{\bar{x}_1}^{s_1} \vee \overbrace{x_2}^{s_2})}_{t_1} \wedge \underbrace{(\overbrace{x_1}^{s_1} \vee \overbrace{x_3}^{s_3})}_{t_2}$$

3SAT

$$e_1 = (\bar{0} \vee 1) \wedge (0 \vee 1)$$

$$= (1 \vee 1) \wedge (0 \vee 1) = 1 \wedge 1 = 1 = \text{true}$$

The assignment $x_1 = 0,\ x_2 = 1,\ x_3 = 1$ makes e_1 true. So that this expression is satisfiable.

* On the other hand,

$$e_2 = (x_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2$$

$0 \vee 1$
$1 \wedge 1 \wedge 0 = 0$

$(0 \vee 1) \wedge 1 \wedge 0 \Rightarrow 1 \wedge 1 \wedge 0 = 0$ false.

It is not satisfiable because every argument for the variables x_1 and x_2 will make e_2 false.

* A deterministic algo is easy to process but

Take all possible values of the variables $x_1, x_2 \ldots x_n$ and for each evaluate the expression.

There are 2^n - possibilities.

↳ its exhaustive search & takes exponential time complexity.

* The non-deterministic simplifies, if 'e' is satisfiable, we guess the value of each x_i and then evaluate e.

↳ its $O(n)$ algorithm.

∴ Deterministic takes 2^n time

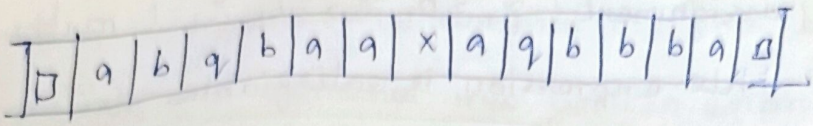
& Non-deterministic take $O(n)$ time.

Theorem:

Suppose that a two-tape machine carryout a computation in 'n' steps. Then this computation can be simulated by a std: TM in $O(n^2)$ steps.

Proof:

* For the simulation of the computation on the two-tape machine, the std: m/c keeps the ID of the two-tape m/c on its tape.



* To simulate one move, the std: m/c needs to search the entire active area of its tape
* But, one move of the two-tape machine can extend the active area by at most two cells, after 'n' moves the active area has a length of at most $O(n)$.
* Therefore, the entire simulation takes $O(n^2)$ moves.

Theorem:

Suppose that a nondeterministic TM M can carry out a computation of 'n' steps. Then the std: TM carry out the same computation in $O(k^{an})$ steps, where k and 'a' are independent of n.

Proof:

* A std: TM can simulate nondeterministic m/c by keeping track of all possible configuration, continually searching and updating the entire active area.
* If 'k' is the max: branching factor for the nondeterminism then after n steps there are at most k^n possible configurations
* Since at most one symbol can be added to each configuration by a single move, the length of one configuration after n moves is $O(n)$.
* ∴, to simulate one move, the std m/c must search an active area of length $O(nk^n)$, leading to the desired result

Language Families and Complexity classes.

* Each class of automata is defined by the nature of its temporary storage.
* Another way to classifying langs: is to a TM & its time complexity.

Definition:

* A TM M decides a lang: in time $T(n)$ if every $w \in L$ with $|w| = n$ is decided in $T(n)$ moves.
* If m is nondeterministic, then every $w \in L$,

  there is at least one sequence of moves of length less than or equal to $T(|w|)$ that leads to acceptance, and the TM halts on all inputs in time $T(|w|)$

Definition:

* A Lang L is said to be a member of the class $DTIME(T(n))$, if there exists a deterministic multitape TM that decides L in time $O(T(n))$.
* A Lang: L is said to be a member of the class $NTIME(T(n))$ if there exist a nondeterministic multitape TM that decides L in time $O(T(n))$.
* Some relations between these complexity classes, such as

$$DTIME(T(n)) \subseteq NTIME(T(n)).$$

and

$$T_1(n) = O(T_2(n))$$

implies

$$DTIME(T_1(n)) \subseteq DTIME(T_2(n))$$

Theorem:

There is no total Turing computable function $f(n)$ such that every recursive lang: can be decided in time $f(n)$, where 'n' is the length of the input string.

Proof:

* Consider the alphabet $\Sigma = \{0, 1\}$ with all strings in Σ^+ arranged in proper order $w_1, w_2 \ldots$
* Also proper ordering of the TM is $M_1, M_2 \ldots$
* Then the function $f(n)$ is

$$L = \{ w_i : M_i \text{ does not decide } w_i \text{ in } f(|w_i|) \text{ steps} \}$$

* Now consider L is recursive

Any $w \in L$ & compute first $f(|w|)$

if 'f' is total Turing computable fn: this is possible. $w_1 - M_1, w_2 - M_2 \ldots$

Then find the position 'i' of w in the sequence $w_1, w_2 \ldots$

This is also possible because the sequence is in proper order,

Take any 'i' & find M_i

M_1 M_2 ... M_n

w_i → to find in $f(|w|)$ space

↓

its tell $w \in L$ or not

↓

recursive

But $w \in L$ or not is an undecidable in $f(n)$ space.

↓

By assuming the diagonalization lang.

↓

is an undecidable.

∴ Total Turing computable function is false.

Example:

The non-context free lang (RL, CSL, URL) $L = \{ww : w \in \{a,b\}^*\}$ is in NTIME(n). This is recognize the string by the following algo:

1. copy the input from the input file to tape 1.
   Nondeterministically guess the middle of this string.
2. Copy the second part to tape 2.
3. Compare the symbols on tape 1 and tape 2 one by one.

* All the steps can be done in $O(|w|)$ time, so $L \in NTIME(n)$.
* We can show that $L \in DTIME(n)$ if we can devise an algo. for finding the middle of a string in $O(n)$ time.

* This can be done:

Look at each symbol on tape 1, keeping a count on tape 2, but counting only every second symbol.

Example:

- Consider the family of CSL.

$$L_{CF} \subseteq DTIME(n^3)$$

and $L_{CF} \subseteq NTIME(n)$.

* Exhaustive search parsing is possible here. Since only a limited number of productions are applicable at each step.
* The max: number of sentential form is

$$N = |P| + |P|^2 + \cdots + |P|^{cn}$$

$$= O(|P|^{cn+1})$$

we cannot claim that one in

$$L_{CS} \subseteq DTIME(|P|^{cn+1})$$

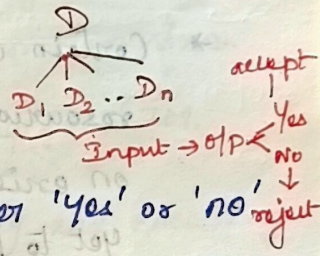
* As $T(n)$ increases, more and more of the families L_{REG}, L_{CF}, L_{CS} are covered.

The Complexity classes P and NP

1. There exist an infinite number of properly nested complexity classes $DTIME(n^k)$, $k = 1, 2 \cdots$ Chomsky model is not clear.
2. TM is the best model of an actual computer is not clear. So ananlysis not depend on any particular type of TM
3. For some lang: nondeterministic TM is efficient & some lang: deterministic is efficient.

Definition of P classes:

* set of decision problem is solvable in polynomial time on in the class P.
* If there exist an algorithm A such that
  - → 'A' takes instance of D as input:
  - → 'A' always output the correct answer 'yes' or 'no'
  - → There exist a polynomial P such that the execution of A on ~~output~~ input of size 'n' always terminates in P(n) or fewer steps.



eg. 2CNF

Decision { Yes – Accept / No – Reject }

Nondeterministic polynomial Time algorithm:

* It consist of those problem that are 'verifiable' in polynomial time.
  - → Decision problems solvable in non-deterministic polynomial time.

eg. 3-SAT, 3CNF, TSP.

NP Problem ↓

Polynomial time { Problem identification time + verification time

Consider

$$P = \bigcup_{i \geq 1} DTIME(n^i)$$

* This class includes all langs: that are accepted by some deterministic TM in polynomial time, without any regard to the degree of the polynomial.
* L_{REG} and L_{CF} are in P.
* Similarly

$$NP = \bigcup_{i \geq 1} NTIME(n^i)$$

$$P \subseteq NP$$

* In the class P — realistic & unrealistic computations.
* Certain computations are theoretically possible & high resource requirement, they must by rejected as unrealistic on existing computers, as well as on supercomputers yet to be designed.

↳ such problem also called as intractable

↓

There no realistic hope of a practical algo:

* The intractable problem definition, is called cook-karp thesis
* In the cook-karp thesis, a problem that is in P is called tractable, and one that is not is said to be intractable.

Some NP Problems

Problems that can be solved nondeterministically in polynomial time.

NP - Nondeterministic polynomial time algo:

↓

solution identification time + verification time } Polynomial time.

Example:

Consider the SAT problem. We made some rudimentary argument to claim that this problem can be solved efficiently by a nondeterministic TM and, rather inefficiently, by a brute-force exponential search.

* suppose that a CNF expression has length n, with m different literals. $m \leq n$. Take 'n' as the problem size.
* Encode the CNF expression as a string for a TM.
* By taking $\Sigma = \{x, v, \wedge, (,), -, 0, 1\}$ and encoding the subscript of x as a binary number.
* The CNF exp. $(x_1 \vee \bar{x}_2) \wedge (x_3 \vee x_4)$ is encoded as

$x_1 = x1$ $x_4 = x100$
$x_2 = x10$
$x_3 = x11$

$$(x1 \vee x\bar{1}0) \wedge (x11 \vee x100)$$

* The subscript cannot be larger than m, the max. length of any subscript is $\log_2 m$.
* As a consequence the maximum encoded length of an n-symbol CNF is $O(n \lg n)$.
* Next generate the trial solution for the variables. it can be done non-deterministic time O(n) time
* The trial solution is then substituted into the input string - O(n) time.

* This can be done in $O(n^2 \log n)$ time. The entire process therefore can be done in $O(n^2 \log n)$ or $O(n^3)$ time and SAT $\in$ NP.

* There are large number of graph problems are known to be in NP.

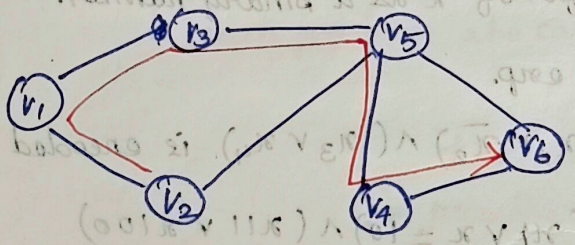
Example:

The Hamiltonian path problem.

* Given an undirected graph, with vertices $v_1, v_2 .. v_n$ a Hamiltonian path is a simple path that passes through all the vertices.

* The graph has a hamiltonian path

$(v_2, v_1), (v_1, v_3), (v_3, v_5), (v_5, v_4), (v_4, v_6)$.



$v_2 \rightarrow v_1 \rightarrow v_3 \rightarrow v_5 \rightarrow v_4 \rightarrow v_6$

Deterministic
↳ n! - permutati
↳ Brute-force

Non-deterministic
↳ adjacency matrix

* Deterministic algo: is easily found, since the Hamiltonian path is a permutation of the vertices $v_1, v_2 .. v_n$.

* There are n! of such permutations, and a brute-force search of all of them will give the answer.

* To explore the non-deterministic solution, we must first find a way to represent a graph by a string using the adjacency matrix.

* For a directed graph with vertices $V_1, V_2, V_3 \cdots V_n$ and edge set E, an adjacency matrix is an $n \times n$ array in which $a(i,j)$ the entry in the i^{th} row & j^{th} column satisfies

$$a(i,j) = 1 \quad \text{if } (v_i, v_j) \in E,$$
$$= 0 \quad \text{if } (v_i, v_j) \notin E.$$

* An undirected graph edge can be considered two separate edges in opposite directions. The array.

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \Rightarrow O(n^2)$$

* The graph with 'n' vertices then requires a string of length n^2 for its representation.

* For an undirected graph the matrix is symmetric, so the storage requirement can be reduced to $\frac{(n+1)n}{2}$ but in any case, the input string will have length $O(n^2)$.

* Generate, non-deterministically, a permutation of the vertices. This can be done in $O(n^3)$ time.

* Finally check the permutation to see if it constitutes a path. A time $O(n^2)$ is sufficient for this.

Therefore, HAMPATH $\in$ NP.

string length - $O(n^2)$

permutation - $O(n)$

Check the permutation - $O(n)$

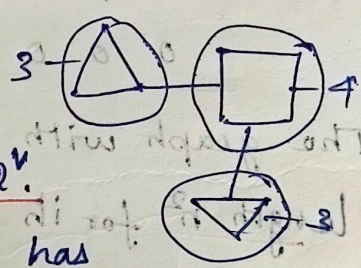
$O(n^4)$

The Clique Problem:

* Let G be an undirected graph with vertices $v_1, v_2, \cdots v_n$.

* A k-clique is a subset $V_k \subseteq 2^V$, such that there is an edge between every pair of vertices $v_i, v_j \in V_k$.

* The clique problem is to decide, for a given k, G has a k-clique.



* A deterministic search can examine all the elements of 2^V. This is straightforward, but has exponential time complexity.

deter - exp.
nondeter - linear.

* A non-deterministic algo: just guesses the correct subset.

* The clique problem is similar to Hamiltonian path problem, hence can be solved in $O(n^4)$ time that CLIQ $\in$ NP

The common characteristics

1. All problems are in NP and have simple non-deterministic solutions.
2. All problems have deterministic solutions with exponential time complexity, but it is not known if they are tractable.

Polynomial Time Reduction

* If we can reduce them to each other, in the sense that if one is tractable, the others will be tractable also.

Definition:

A lang: L_1 is said to be polynomial-time reducible to another lang: L_2 if there exists a deterministic TM by which any w_1 in the alphabet of L_1 can be transformed in polynomial time to a w_2 in the alphabet of L_2 in such a way that $w_1 \in L_1$ iff $w_2 \in L_2$.

$w_2 \in L_2$

$w_1 \in L_1$

$L_1 \longrightarrow L_2$

tractable tractable.

try to find solution.

Example:

In a SAT problem, we put no restriction on the length of a clause. A restricted type of satisfiability is the three-satisfiability problem (3SAT) in which each clause can have at most three literals. The SAT problem is polynomial time reducible to 3SAT.

2SAT - 2 - x_1 & x_2

3SAT - 3 - x_1, x_2 & x_3

4SAT - 4 - x_1, x_2, x_3 & x_4.

* Illustrate the reduction with the simple 4-literal expression

$x_1 = 0$, $x_2 = 1$, $x_3 = 1$, $x_4 = 1$

$$e_1 = (x_1 \vee x_2 \vee x_3 \vee x_4)$$
$$= (0 \vee 1 \vee 1 \vee 1) = 1$$

* Introduce a new variable z & construct

$$e_2 = (x_1 \vee x_2 \vee z) \wedge (x_3 \vee x_4 \vee \bar{z})$$
$$(0 \vee 1 \vee 0) \wedge (1 \vee 1 \vee 1)$$

equivalent

* If e_1 is true, $1 \wedge 1 =$ true

one of the x_1, x_2, x_3, x_4 must be true.

if $x_1 \vee x_2$ — true

choose $z = 0$ & e_2 = true.

if $x_3 \vee x_4 = 1$

choose $z = 1$ to satisfy e_2

$$e_2 = \underbrace{(x_1 \vee x_2)}_{\text{true}} \vee 0) \wedge \underbrace{((x_3 \vee x_4)}_{1} \vee 1)$$

true $\wedge$ true

* If e_2 is true, e_1 must also be true, so for satisfiability, e_1 and e_2 are equivalent.

Example:

The 3SAT problem is polynomial reducible to CLIQUE.

* We assume that in any 3SAT expression each clause has exactly three literals.

* If in some expression that is not the case, we just add extra literals that do not change the satisfiability.

eg.

$(x_1 \vee x_2)$ is equivalent to $(x_1 \vee x_1 \vee x_2)$.

$e = (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3)$ $\Rightarrow 1$

$\wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$.

* Draw a graph in which each clause is represented by a group of three vertices and each literal is associated with one of the vertices.

* Draw an edge from $(\bar{x}_2)_1$ to $(x_3)_2$ and from $(\bar{x}_2)_1$ to $(x_3)_3$ but not from $(\bar{x}_2)_1$ to $(x_3)_2$ — not possible.

* The sub-graph with vertices $(\bar{x}_2)_1, (x_3)_2, (x_3)_3$ and $(x_1)_4$ is a 4-clique and that

$\bar{x}_2 = x_3 = x_1 = 1$

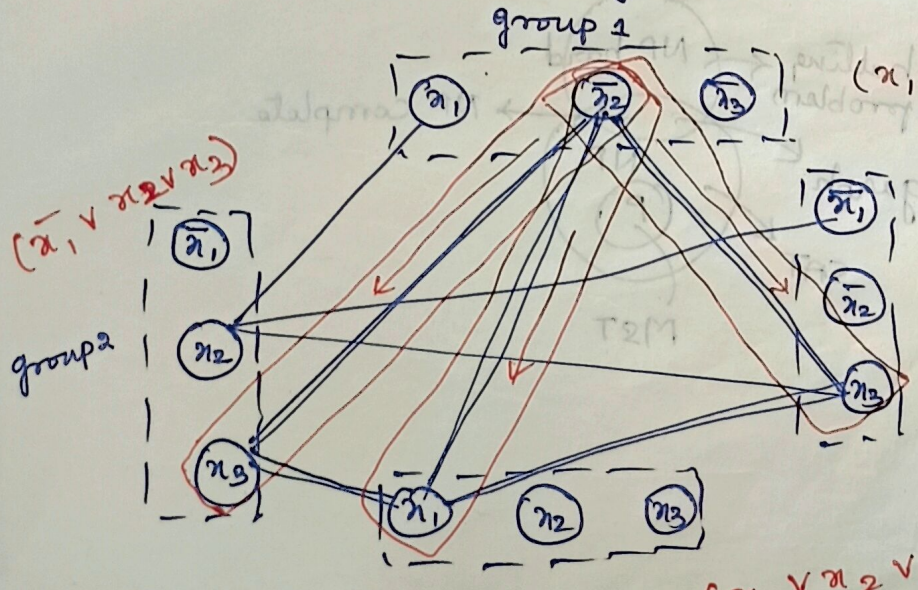
$(\bar{x}_2 \vee x_3 \vee x_1)$ $(0 \vee 1 \vee 1) = 1$

$x_1 = 0$ $(0 \vee 0 \vee 1) = 1$

$x_2 = 1$

$x_3 = 1$

is a variable assignment that satisfies e.



* The 3SAT problem can be satisfied iff the associated graph has a k-clique.

* The transformation from the 3SAT to the graph can be done deterministically in polynomial time.

NP Completeness.

Definition:

A Lang: L is said to be NP-complete if $L \in NP$ & every $L_1 \in NP$ is polynomial time reducible to L.

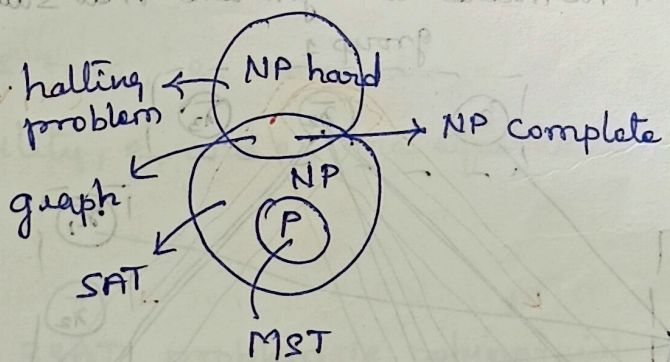
NP com $L \rightarrow L_1$ by NP com

* If L is NP-complete and polynomial time reducible to L_1, then L_1 is also NP-complete.

* If we can find one deterministic polynomial-time algo: for any NP-complete lang: then every lang in NP is also in P, that is,

$$P = NP.$$

* NP-complete problems is intractable, then many interesting problems are not practically solvable.



Theorem:

The SAT is NP-complete

Proof:

Every configuration sequence of a TM one can construct a CNF expression that is satisfiable iff there is a sequence of configurations leading to acceptance.

* SAT can be reduced to 3SAT,
  3SAT can be reduced to CLIQUE
  ∴ 3SAT & CLIQUE are both NP-complete.

———— × ————.