UNIT- VI

COMPUTABILITY

8

COMPUTABILITY THEORY

After going through this chapter, you should be able to understand :

- Chomsky hierarchy of Languages
- Linear Bounded Automata and CSLs
- LR (0) Grammar
- Decidability of problems
- UTM and PCP
- P and NP problems

8.1 CHOMSKY HIERARCHY OF LANGUAGES

Chomsky has classified all grammars in four categories (type 0 to type 3) based on the right hand side forms of the productions.

(a) Type 0

These types of grammars are also known as phrase structured grammars, and RHS of these are free from any restriction. All grammars are type 0 grammars.

Example : productions of types $AS \rightarrow aS$, $SB \rightarrow Sb, S \rightarrow \in$ are type 0 production.

(b) Type 1

We apply some restrictions on type 0 grammars and these restricted grammars are known as type 1 or **context - sensitive grammars** (CSGs). Suppose a type 0 production $\gamma\alpha\delta \rightarrow \gamma\beta\delta$ and the production $\alpha \rightarrow \beta$ is restricted such that $|\alpha| \leq |\beta|$ and $\beta \neq \in$. Then these type of productions is known as type 1 production. If all productions of a grammar are of type 1 production, then grammar is known as type 1 grammar. The language generated by a context - sensitive grammar is called context - sensitive language (CSL).

In CSG, there is left context or right context or both. For example, consider the production $\alpha A\beta \rightarrow \alpha a\beta$. In this, α is left context and β is right context of A and A is the variable which is replaced.

The production of type $S \rightarrow \in$ is allowed in type 1 if $\in$ is in L(G), but S should not appear on right hand side of any production.

Example : productions $S \rightarrow AB, S \rightarrow \in, A \rightarrow c$ are type 1 productions, but the production of type $A \rightarrow Sc$ is not allowed . Almost every language can be thought as CSL.

Note : If left or right context is missing then we assume that $\in$ is the context.

(c) Type 2

We apply some more restrictions on RHS of type 1 productions and these productions are known as type 2 or context - free productions. A production of the form $\alpha \rightarrow \beta$, where $\alpha, \beta \in (V \cup \Sigma)^*$ is known as type 2 production. A grammar whose productions are type 2 production is known as type 2 or context - free grammar (CFG) and the languages generated by this type of grammars is called context - free languages (CFL).

Example : $S \rightarrow S + S, S \rightarrow S * S, S \rightarrow id$ are type 2 productions.

(d) Type 3

This is the most restricted type. Productions of types $A \rightarrow a$ or $A \rightarrow aB|Ba$, where $A, B \in V$, and $a \in \Sigma$ are known as type 3 or regular grammar productions. A production of type $S \rightarrow \in$ is also allowed, if $\in$ is in generated language.

Example : productions $S \rightarrow aS, S \rightarrow a$ are type 3 productions.

Left - linear production : A production of type $A \rightarrow Ba$ is called left - linear production.

Right - linear production : A production of type $A \rightarrow aB$ is called right - linear production. A left - linear or right - linear grammar is called regular grammar. The language generated by a regular grammar is known as regular language.

A production of type $A \to w$ or $A \to wB$ or $A \to Bw$, where $w \in \Sigma^*$ can be converted into the forms $A \to a$ or $A \to aB$ or $A \to Ba$, where $A, B \in V$ and $a \in \Sigma$.

Example : $A \to 10$ A can be replaced by productions $A \to 1B$, where B is a new variable and $B \to 0A$.

In general, if $A \to a_1a_2a_3 \ldots\ldots a_na_{n+1}B$, then this production can replaced by the following productions.

$$A \to a_1\ B_1,$$
$$B_1 \to a_2\ B_2,$$
$$B_2 \to a_3\ B_3,$$
$$\ldots$$
$$B_n \to a_{n+1}\ B$$

Similar result is obtained for left - linear grammars also.

8.1.1 Hierarchy of grammars

Type 0 or Phrase structured grammar

$\Downarrow$ Restrictions applied

Type 1 or Context - sensitive grammar

$\Downarrow$ Restrictions applied

Type 2 or Context - free grammar

$\Downarrow$ Restrictions applied

Type 3 or Regular grammar

Example : Consider the following and find the type of the grammar.

(a) $S \to Aa,\ A \to c \mid Ba,\ B \to abc$

(b) $S \to aSa \mid c$

(c) $S \to aAS \mid SBb,\ AS \to aAS|aS,\ SB \to Sb|SBb$

Solution :

(a)

Production			Type
S	→	Aa	Type 3
A	→	c	Type 3
A	→	Ba	Type 3
B	→	abc	Type 3

So, given productions are of type 3 and hence grammar is regular.

(b)

S	→	aSa	Type 2
S	→	c	Type 3

So, given productions are of type 2 and hence grammar is CFG.

Note : We select the higher type and higher type between type 3 and type 2 is type 2).

(c)

S	→	aAS	Type 2
S	→	SBb	Type 2
AS	→	aAS	Type 1
AS	→	aS	Type 1
SB	→	Sb	Type 1
SB	→	SBb	Type 1

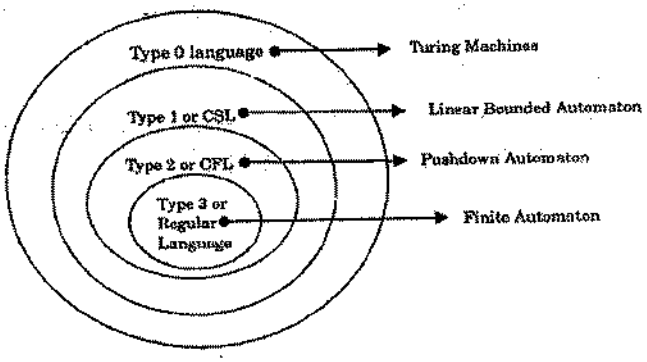
So, given productions are of type 1 and hence grammar is CSG.

8.1.2 Relation Among Grammars and Languages

Type 0 is the super set and type 1 is contained in type 0, type 2 is contained in type 1, and type 3 is contained in type 2.

$$\text{Type } 0 \subseteq \text{Type } 1 \subseteq \text{Type } 2 \subseteq \text{Type } 3$$

8.1.3 Languages and Their Related Automaton



FIGIURE : Languages and their related Automaton

8.2 LINEAR BOUNDED AUTOMATA

The Linear Bounded Automata (LBA) is a model which was originally developed as a model for actual computers rather than model for computational process. A linear bounded automaton is a restricted form of a non deterministic Turing machine.

A linear bounded automaton is a multitrack Turing machine which has only one tape and this tape is exactly of same length as that of input.

The linear bounded automaton (LBA) accepts the string in the similar manner as that of Turing machine does. For LBA halting means accepting. In LBA computation is restricted to an area bounded by length of the input. This is very much similar to programming environment where size of variable is bounded by its data type.

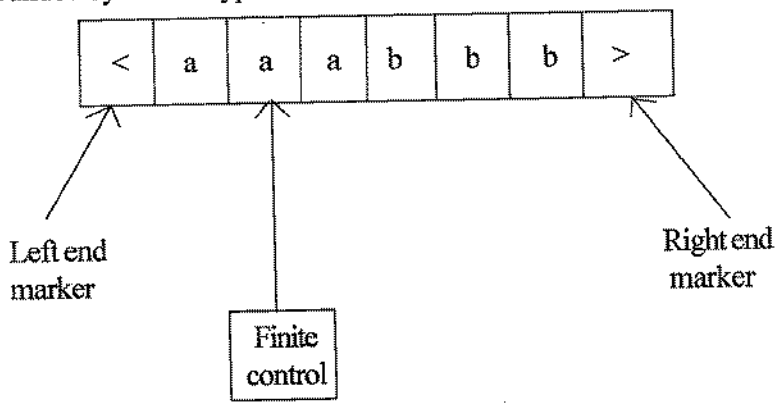


FIGURE : Linear bounded automaton

The LBA is powerful than NPDA but less powerful than Turing machine. The input is placed on the input tape with beginning and end markers. In the above figure the input is bounded by < and >.

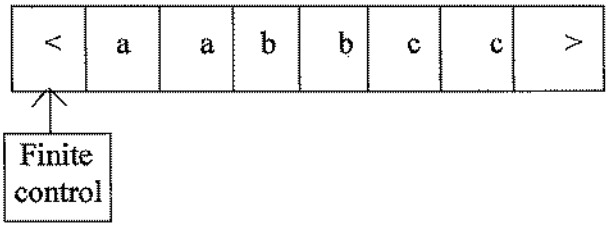
A linear bounded automata can be formally defined as :

LBA is 7 - tuple on deterministic Turing machine with

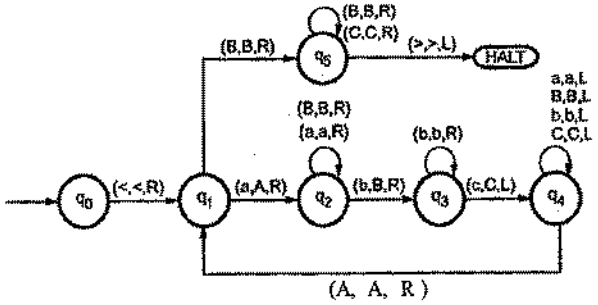
$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}) \text{ having}$$

1. Two extra symbols of left end marker and right end marker which are not elements of Γ.
2. The input lies between these end markers.
3. The TM cannot replace < or > with anything else nor move the tape head left of < or right of >.

Example : We can construct a language $L = \{a^n\ b^n\ c^n \mid n \geq 1\}$ using LBA as follows.



The input is placed on the input tape which is enclosed within left end marker and right end marker. We will apply the simple logic as : when we read 'a' convert it to A then move right by skipping all a's. On encountering first 'b' we will convert it to B. Then move right by skipping all b's. On receiving first c convert it to C. Move in left direction unless you get A. Repeat the above procedure and convert equal number of a's, b's, and c's to corresponding A's, B's and C's. Finally move completely to the rightmost symbol if it is '>' a right end marker,then HALT. The machine will be :



Simulation : Consider input aabbcc

< aabbcc >	Move right.
↑	
< aabbcc >	Convert to A, move right.
↑	
< Aabbcc >	Move right.
↑	
< Aabbcc >	Convert to B, move right.
↑	
< AaBbcc >	Move right.
↑	

<AaBbcc>	Convert to C, move left.
<AaBbCc>	Move left
<AaBbCc>	Move left.
<AaBbCc>	Move left.
<AaBbCc>	Move right.
<AaBbCc>	Convert to A, Move right.
<AABbCc>	Move right.
<AABbCc>	Convert to B, Move right.
<AABBCc>	Move right.
<AABBCc>	Convert to C, Move left.
<AABBCC>	Move left continuously by skipping B's.
<AABBCC>	Move right.
<AABBCC>	If we get B, we will move right to check whether all b's and c's are converted to B and C.
<AABBCC>	If we get right end marker '>' then we HALT by accepting the input aabbcc.

Thus in LBA the length of tape exactly equal to the input string and tape head can not move left of '<' right of '>'.

8.3 CONTEXT SENSITIVE LANGUAGES (CSLs)

The context sensitive languages are the languages which are accepted by linear bounded automata. These type of languages are defined by context sensitive grammar. In this grammar more than one terminal or non terminal symbol may appear on the left hand side of the production rule. Along with it, the context sensitive grammar follows following rules :

i. The number of symbols on the left hand side must not exceed number of symbols on the right hand side.
ii. The rule of the form $A \rightarrow \in$ is not allowed unless A is a start symbol. It does not occur on the right hand side of any rule.

The classic example of context sensitive language is $L = \{a^n\ b^n\ c^n \mid n \geq 1\}$. The context sensitive grammar can be written as :

S	→	aBC
S	→	SABC
CA	→	AC
BA	→	AB
CB	→	BC
aA	→	aa
aB	→	ab
bB	→	bb
bC	→	bc
cC	→	cc

Now to derive the string aabbcc we will start from start symbol :

S	rule S →	SABC
SABC	rule S →	aBC
aBCABC	rule CA →	AC
aBACBC	rule CB →	BC
aBABCC	rule BA →	AB
aABBCC	rule aA →	aa
aaBBCC	rule aB →	ab
aabBCC	rule bB →	bb
aabbCC	rule bC →	bc
aabbcC	rule cC →	cc
aabbcc		

Note : The language $a^n\ b^n\ c^n$ where $n \geq 1$ is represented by context sensitive grammar but it can not be represented by context free grammar.

Every context sensitive language can be represented by LBA.

8.4 LR (k) GRAMMARS

Before going to the topic of LR (k) grammar, let us discuss about some concepts which will be helpful understanding it.

In the unit of context free grammars you have seen that to check whether a particular string is accepted by a particular grammar or not we try to derive that sentence using rightmost derivation or leftmost derivation. If that string is derived we say that it is a valid string.

Example :

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow id \mid (E)$$

Suppose we want to check validity of a string id + id * id . Its rightmost derivation is

$$\begin{aligned} E &\Rightarrow E + T \\ &\Rightarrow E + T * F \\ &\Rightarrow E + T * id \\ &\Rightarrow E + F * id \\ &\Rightarrow E + id * id \\ &\Rightarrow T + id * id \\ &\Rightarrow F + id * id \\ &\Rightarrow id + id * id \end{aligned}$$

FIGURE(a) : Rightmost Derivation of id + id * id

Since this sentence is derivable using the given grammar. It is a valid string. Here we have checked the validity of string using process known as derivation.

The validity of a sentence can be checked using reverse process known as reduction. In this method for a given x, inorder to know whether it is valid sentence of a grammar or not, we start with x and replace a substring x_1 with variable A if $A \rightarrow X_1$ is a production. We repeat this process until we get starting state.

Consider the grammar,

$$E \rightarrow E + T \mid T$$
$$E \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

Let us check the validity of string id + id * id.

F + id * id Replaced F with id since F → id is a production
T + id * id Replaced F with T using production T → F
E + id * id Replaced T with E using production E → T
E + F * id Replaced id with F using production F → id
E + T * id Replaced F with using production T → F
E + T * F Replaced id with F using production F → id
E + T Replaced T * F with T using production T → T * F
E Replaced E + T with E using production E → E + T

FIGURE(b) : Reduction of id + id * id

Here since we are able to reduce to starting state E, so that id + id * id is accepted by the given grammar.

Note : There may be different ways of selecting as substring in sentential form. In our reduction we have used reverse of rightmost derivation shown in Figure(a).

The substring in right sentential form which causes reduction to starting state is known as handle and corresponding production is known as handle production. For example, in right sentential form E + T * id of Figure(b) we can either replace substring T with F using T → F or replace id with F using F → id. If we use the first reduction, the sentential form will become E + F * id. This will not lead to starting state. Hence here F is not handle. Where as if we reduce, the sentential form will be E + T * F which can be reduced to starting state using subsequent reductions. Hence here F is a handle and F → id is handle production.

In reduction process we have seen that we repeat the process of substitution until we get starting state. But some times several choices may be available for replacement. In this case we have to backtrack and try some other substring . For certain grammars it is possible to carry out the process in deterministic. (i. e., having only one choice at each time). LR grammars form one such subclass of context free grammars. Depending on the number of look ahead symbolized to determine whether a substring must be replaced by a non terminal or not, they are classified as LR(0) , LR(1).... and in general LR(k) grammars.

LR(k) stands for left to right scanning of input string using rightmost derivation in reverse order (we say reverse order because we use reduction which is reverse of derivation) using look ahead of k symbols.

8.4.1 LR(0) Grammar

LR(0) stands for left to right scanning of input string using rightmost derivation in reverse order using 0 look ahead symbols.

Before defining LR(0) grammars, let us know about few terms.

Prefix Property : A language L is said to have prefix property if whenever w in L, no proper prefix of w is in L. By introducing marker symbol we can convert any DCFL to DCFL with prefix property. Hence $L\$ = \{ w\$ \mid w \in L \}$ is a DCFL with prefix property whenever w is in L.

Example : Consider a language L = { cat, cart, bat, art, car } . Here, we can see that sentence cart is in L and its one of the prefixes car is also is in L. Hence, it is not satisfying property. But L$ = { cat $, cart $, bat $, art $, car $ }

Here, cart $ is in L$ but its prefix cart or car are not present in L$. Similarly no proper prefix is present in L$. Hence, it is satisfying prefix property.

Note : LR(0) grammar generates DCFL and every DCFL with prefix property has a LR(0) grammar.

LR Items

An item for a CFG is a production with dot any where in right side including beginning or end. In case of $\in$ production, suppose $A \to \in$, $A \to .$ is an item.

Example :

Consider the grammar,

$S' \rightarrow S$

$S \rightarrow cAd$

$A \rightarrow a \mid \in$

The items for this grammar are ,

$S' \rightarrow .S$

$S' \rightarrow S.$

$S \rightarrow .cAd$

$S \rightarrow c.Ad$

$S \rightarrow cA.d$

$S \rightarrow cAd.$

$A \rightarrow .a$

$A \rightarrow a.$

$A \rightarrow .$

An item indicates how much of a production we have seen at a given point in parsing process.

Valid Item : We say in item $A \rightarrow \alpha . \beta$ is valid for a viable prefix (i. e., most possible prefix) γ there is a rightmost derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta \alpha \beta w$ and $\delta\alpha = \gamma$.

Example :

$S \rightarrow cAt$

$A \rightarrow ar$

The sentence cart belongs to this grammar,

$S \overset{*}{\Rightarrow} CAt \Rightarrow cart$

The possible or viable prefixes for cart are { c, ca, car, cart } for the prefix ca $A \rightarrow a.r.$ is valid item and for viable prefix car $A \Rightarrow ar$ is valid item.

Computing Valid Item Sets

The main idea here is to construct from a given grammar a deterministic finite automata to recognize viable prefixes. We group items together into sets which give to states of DFA. The items may be viewed as states of NFA and grouped items may be viewed as states of DFA obtained using subset construction algorithm.

To compute valid set of items we use two operations goto and closure.

Closure Operation

It I is a set of items for a grammar G, then closure (I) is the set of items constructed from I by two rules.

1. Initially, every item I is added to closure (I).
2. If $A \rightarrow \alpha.B\beta$ is in closure (I) and $B \rightarrow \delta$ is production then add item $B \rightarrow \delta$ to I, if it is not already there. We apply this rule until no more new items can be added to closure (I).

Example : For the grammar,

$$S' \rightarrow S$$
$$S \rightarrow cAd$$
$$A \rightarrow a$$

If $S' \rightarrow S$ is set of one item in state I then closure of I is,

$$I_1 : S' \rightarrow .s$$
$$S \rightarrow .cAD$$

The first item is added using rule 1 and $S \rightarrow .cAd$ is added using rule 2. Because '.' is followed by nonterminal S we add items having S in LHS. In $S \rightarrow .cAd$ '.' is followed by terminal so no new item is added.

Goto Function : It is written as goto (I, X) where I is set of items and X is grammar symbol.

If $A \rightarrow \alpha.X\beta$ is in some item set I then goto (I, X) will be closure of set of all item $A \rightarrow \alpha.X.\beta$.

For example,

goto (I_1, c)

closure $(S \rightarrow c.Ad)$

i. e., $S \rightarrow c.Ad$

$A \rightarrow a$

now let us see how all the valid sets of items are computed for the given grammar in example 1.

Initially I_0 will be the starting state. It contains only the item $S' \rightarrow .S$ we find its closure to find set of items in this state for each state I_1 and symbol β after '.' we apply goto (I_1, β), goto (I_0, S) and find its closure. This constitutes next state I_j. We continue this process goto (I_0, a) until no new states are obtained.

$I_0 : S' \rightarrow .S$

$S \rightarrow .Ad$

$I_1 : S' \rightarrow S.$

$I_2 : S \rightarrow c.Ad$

$A \rightarrow .a$

goto (I_2, A)

$I_3 : S \rightarrow cA.d$

goto (I_2, a)

$I_4 : A \rightarrow a.$

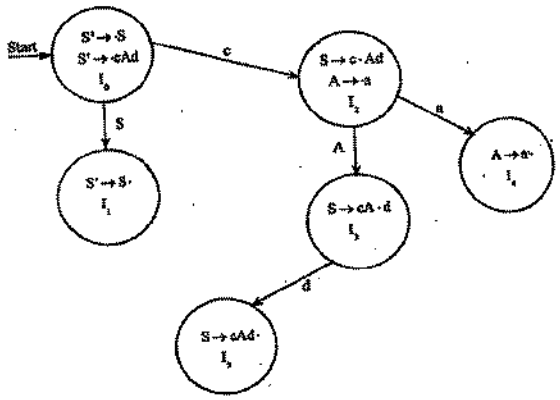
goto (I_3, d)

$I_5 : S \rightarrow cAd.$

This process is stopped because all possible complete items are obtained. A complete item is the one which has dot in rightmost position.

Ecah item set corresponds to a state of DFA. Hence, the DFA for given grammar will have six states corresponding to I_0 *to* I_5.

DFA :



FIGURE(a) : DFA whose States are the Sets of Valid Items

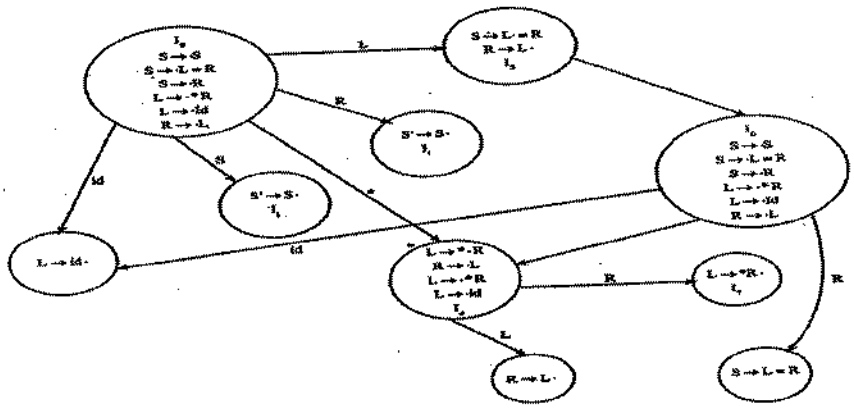
Definition of LR(0) Grammar : We say G is an LR (0) grammar if,

1. Its start symbol does not appear on the right hand side of any production and
2. For every viable prefix γ of G, whenever $A \rightarrow \alpha$ is a complete item valid for γ, then no other complete item nor any item with terminal to the right of the dot is valid for γ.

Condition 1 : For a grammar to be LR(0) it should satisfy both the conditions. The first condition can be made to satisfy by all grammars by introduction of a new production $S' \rightarrow S$ is known augmented grammar.

Condition 2 : For the DFA shown in Figure(a), the second condition is also satisfied because in the item sets I_1, I_4 and I_5 each containing a complete item, there are no other complete items nor any other conflict.

Example : Consider the DFA given in figure(b).



FIGURE(b) : DFA for the given Grammar

DFA for grammar,

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

i. The first condition of LR(0) grammar is satisfied.

ii. Consider state I_2 and viable prefixes of L = R { L, L = and L = R } for prefix LR $\rightarrow$ L. is a complete item and there is another item having the prefix L i. e., S $\rightarrow$ L . = R followed by terminal. Hence, violating second rule. So it is not LR(0) grammar.

8.5 DECIDABILITY OF PROBLEMS

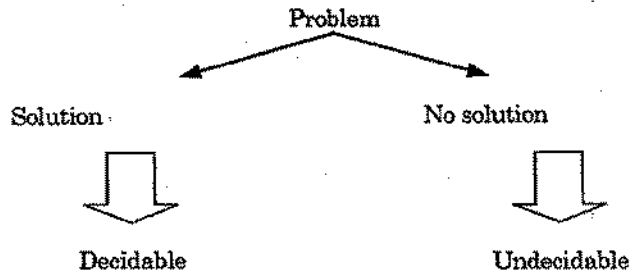
In our general life, we have several problems and some of these have solution also, but some have not. Simply, we say a problem is decidable if there is a solution otherwise undecidable.

Example : consider following problems and their possible answers.

1. Does the sun rise in the east ? (YES)
2. Does the earth move around the sun ? (YES)
3. What is your name ? (FLAT)
4. Will tomorrow be a rainy day ? (No answer)

We have solutions (answers) for all problems except the last. We can not answer the last problem, because we have no way to tell about the weather of tomorrow, but to some extent we can only predict. So, the last problem is undecidable and remaining problems are decidable.

So, if a problem can be solved or answered based on some algorithm then it is decidable otherwise undecidable.



Each problem P is a pair consisting of a set and a question, where the question can be applied to each element in the set. The set is called the domain of the problem, and its elements are called the instances of the problem.

Example :

Domain = { All regular languages over some alphabet Σ } ,
Instance : L = { w : w is a word over Σ ending in abb} ,
Question : Is union of two regular languages regular ?

8.5.1 Decidable and Undecidable Problems

A problem is said to be decidable if
1. Its language is recursive, or
2. It has solution

Other problems which do not satisfy the above are undecidable. We restrict the answer of decidable problems to " YES" or "NO" . If there is some algorithm exists for the problem, then outcome of the algorithm is either "YES" or "NO" but not both. Restricting the answers to only "YES" or "NO" we may not be able to cover the whole problems, still we can cover a lot of problems. One question here. Why we are restricting our answers to only "YES" or "NO"? The answer is very simple ; we want the answers as simple as possible.

Now, we say " If for a problem, there exists an algorithm which tells that the answer is either "YES" or "NO" then problem is decidable."

If for a problem both the answers are possible ; some times "YES" and sometimes "NO", then problem is undecidable.

8.5.2 Decidable Problems for FA, Regular Grammars and Regular Languages

Some decidable problems are mentioned below :
1. Does FA accept regular language ?
2. Is the power of NFA and DFA same ?
3. L_1 and L_2 are two regular languages. Are these closed under following :
   (a) Union
   (b) Concatenation
   (c) Intersection
   (d) Complement

(e) Transpose
(f) Kleene Closure (positive transitive closure)

4. For a given FA M and string w over alphabet Σ, is $w \in L(M)$? This is decidable problem.
5. For a given FM, is $L(M) = \phi$? This is a decidable problem.
6. For a given FA M and alphabet Σ, is $L(M) = \Sigma^*$? This is a decidable problem.
7. For given two FA M_1 and M_2, $L(M_1), L(M_2) \in \Sigma^*$, is $L(M_1) = L(M_2)$? This is a decidable problem.
8. For given two regular languages L_1 and L_2 over some alphabet Σ, is $L_1 \subset L_2$? This is a decidable problem.

8.5.3 Decidable And Undecidable Problems About CFLs, And CFGs

Decidable Problems

Some decidable problems about CFLs and CFGs are given below.

1. If L_1 and L_2 are two CFLs over some alphabet Σ, then $L_1 \cup L_2$ is CFL.
2. If L_1 and L_2 are two CFLs over some alphabet Σ, then $L_1 L_2$ is CFL.
3. If L is a CFL over some alphabet Σ, then L* is a CFL.
4. If L_1 is a regular language, L_2 is a CFL then $L_1 \cup L_2$ is CFL.
5. If L_1 is a regular language, L_2 is a CFL over some alphabet Σ, then $L_1 \cap L_2$ is CFL.
6. For a given CFG G, is $L(G) = \phi$ or not ?
7. For a given CFG G, finding whether L(G) is finite or not, is decidable.
8. For a given CFG G and a string w over Σ, checking whether $w \in L(G)$ or not is decidable.

Undecidable Problems

Following are some undecidable problems about CFGs and CFLs :

1. For two given CFLs L_1 and L_2, whether $L_1 \cap L_2$ is CFL or not, is undecidable.
2. For a given CFL L over some alphabet Σ, whether complement of L i. e. E * - L is CFL or not, is undecidable.
3. For a given CFG G, is L(G) ambiguous ? This is undecidable problem.
4. For two arbitrary CFGs G_1 and G_2, deciding $L(G_1) \cap L(G_2) = \phi$ or not, is undecidable.
5. For two arbitrary CFGs G_1 and G_2, deciding $L(G_1) \subseteq L(G_2)$ or not, is undecidable.

8.5.4 Decidability and Undecidability About TM

We have considered TM as a most powerful machine that can compute anything, which can recognize any language. So, from where undecidability comes and why ? These questions are really interesting. According to Church - Turing Thesis, we have considered TM as an algorithm and an algorithm as a TM . So, for a problem, if there is an algorithm (solution to find answer) then problem is decidable and TM can solve that problem. We have several problems related to computation and recognization that have no solution and these problems are undecidable.

Partial Decidable and Decidable Problems

A TM M is said to partially solve a given problem P if it provides the answer for each instance of the problem and the problem is said to be partially solvable. If all the computations of the TM are halting computations for P, then the problem P is said to solvable.

A TM is said to partially decide a problem if the following two conditions are satisfied.

(a) The problem is a decision problem, and

(b) The TM accepts a given input if and only if the problem has an answer "YES" for the input, that is the TM accepts the language L = { x : x is an instance of the problem, and the problem has the answer "YES" for x }.

A TM is said to decide a problem if it partially decides the problem and all its computations are halting computations.

The main difference between a TM M_1 that partially solves (partially decides) a problem and a TM M_2 that solves (decides) the same problem is that M_1 might reject an input by a non - halting computation, whereas M_2 can reject the input only by a halting computation.

A problem is said to be unsolvable if no algorithm can solve it, and a problem is said to be undecidable if it is a decision problem and no algorithm can decide it.

Decidable Problems about Recursive and Recursive Enumerable Languages

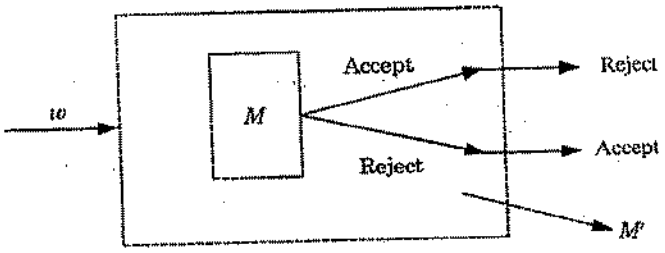
As we have discussed earlier that if a problem has a solution then it is decidable. In this section, we will discuss some decidable problems about recursive and recursive enumerable languages.

1. The complement of a recursive language L over some alphabet Σ is recursive.

Proof : We will discuss a constructive algorithm to prove that complement of a recursive language is also recursive i. e. recursive languages are closed under complementation.

As we know that for all strings $w \in L$, a TM always halts and rejects those strings that are not in L. So, " for all strings $w \in L$ " is always decidable.

We construct a TM M, which recognizes the language L. We construct another TM M' based on M such that M' accepts those strings which are rejected by M. It means, if M accepts then M' does not. M' rejects those strings that are accepted by M. It means, all strings $x \notin L$ are accepted by M' and for all strings $w \in L$ are rejected . So, M' also follows same kind of algorithm to decide whether a string $w \in L$ or not. Hence, complement of recursive language L i. e. $\Sigma^* - L$ is also recursive. The logic diagram of M' is shown in Figure(a).

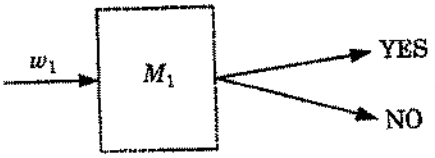


Figure(a)

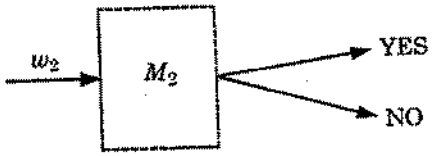
In general, recursive languages are closed under complement operation.

2. The union of two recursive languages is recursive.

Proof : Let L_1 and L_2 be two recursive languages and Turing machines M_1 and M_2 recognize L_1 and L_2 respectively shown in Figure(b) and Figure(c).

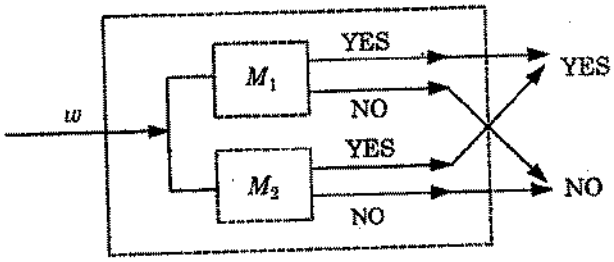


Figure(b)



Figure(c)

We construct a third TM M_3, which follows either M_1 or M_2 as shown in figure(d).



Figure(d)

TM M_3 accepts if either M_1 accepts or M_2 accepts and rejects if either M_1 rejects or M_2 rejects. Since, M_1 and M_2 are based on algorithms, so M_3 is also based on the same kind of algorithm. Therefore, union of two recursive languages L_1 and L_2 is also recursive. In general, recursive languages are closed under union operation.

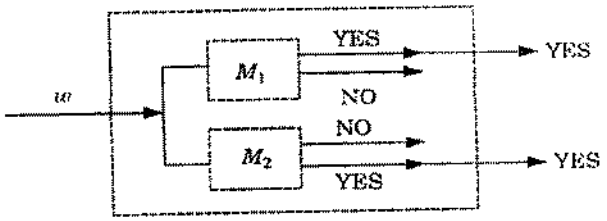
3. A language is recursive if and only if its complement is recursive.

4. The union of two recursively enumerable languages is recursive enumerable.

Proof : Let L_1 and L_2 be two recursively enumerable languages and recognized by M_1 and M_2 Turing machines. We construct another TM M_3, which accepts either L_1 or L_2. Now, as we know the problem about recursive enumerable languages that if w is not in L_1 and L_2, then M_3 can not decide. So, the problem of recursive languages is persistent with M_3 also. So, $N(M_3)$ is recursive enumerable language and hence $L_1 \cup L_2$ is recursive enumerable languages. In general, recursive enumerable languages are closed under union operation.

5. If a language L over some alphabet Σ and its complement $\overline{L} = \Sigma^* - L$ is recursive enumerable, then L and $\overline{L}$ are recursive languages.

Proof : We construct two Turing machines M_1 for L and M_2 for $\overline{L}$. Now, we construct a third TM M_3 based on M_1 and M_2 as shown in figure(e). TM M_3 accepts w if TM M_1 accepts and rejects w if M_2 accepts. It means, if $w \in L$, then w is accepted and if $w \notin L$ then it is rejected. Since, for all w, either w is accepted or rejected. Hence, M_3 is based on algorithm and produces either "YES" or "NO" for input string w, but not both. It means, M_3 decides all the strings over Σ. Hence, L is recursive. As we know that complement of a recursive language is also recursive and hence $\overline{L}$ is also recursive.



Figure(e)

6. We have following co - theorem based on above discussion for recursive enumerable and recursive languages.

Let L and $\overline{L}$ are two languages, where $\overline{L}$ the complement of L, then one of the following is true :

(a) Both L and $\overline{L}$ are recursive languages,

(b) Neither L nor $\overline{L}$ is recursive languages,

(c) If L is recursive enumerable but not recursive, then $\overline{L}$ is not recursive enumerable and vice versa.

Undecidable Problems about Turing Machines

In this section, we will first discuss about halting problem in general and then about TM.

Halting Problem (HP)

The **halting problem** is a decision problem which is informally stated as follows :

"Given a description of an algorithm and a description of its initial arguments, determine whether the algorithm, when executed with these arguments, ever halts. The alternative is that a given algorithm runs forever without halting."

Alan Turing proved in 1936 that there is no general method or algorithm which can solve the halting problem for all possible inputs. An algorithm may contain loops which may be infinite or finite in length depending on the input and behaviour of the algorithm . The amount of work done in an algorithm usually depends on the input size. Algorithms may consist of various number of loops, nested or in sequence. The HP asks the question :

Given a program and an input to the program, determine if the program will eventually stop when it is given that input ?

One thing we can do here to find the solution of HP. Let the program run with the given input and if the program stops and we conclude that problem is solved. But, if the program doesn't stop in a reasonable amount of time, we can not conclude that it won't stop. The question is : " how long we can wait .... ?" . The waiting time may be long enough to exhaust whole life. So, we can not take it as easier as it seems to be. We want specific answer, either "YES" or "NO", and hence some algorithm to decide the answer.

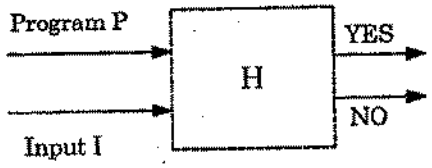
The importance of the halting problem lies in the fact that it is the first problem which was proved undecidable. Subsequently, many other such problems have been described.

Theorem : HP is undecidable.

Proof : This proof was devised by Alan Turing in 1936. Initially, we assume that HP is decidable and the algorithm (solution) for HP is H. The halting problem solution H takes two inputs :

1. Description of TM M i. e. program P and
2. Input I for the program P.

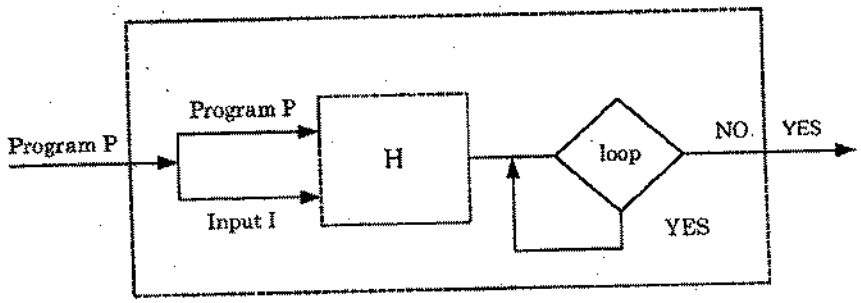
H generates an output "YES" if H determines that P stops on input I or it outputs "NO" if H determines that P loops as shown in figure(a).



Figure(a)

Note : When an algorithm is coded, it is expressed as a string of characters . Input is also coded into the same format. So, after coding, a program and data have no difference in their format of representation and so, a program can be treated a data sometimes and a data can be treated as a program sometimes.

So, now H can be modified to take P as both inputs (the program and its input) and H should be able to determine if P will halt on P as it's input shown in figure(b).



Figure(b)

Let us construct a new, simple algorithm Q that takes output of H as its input and does the following:

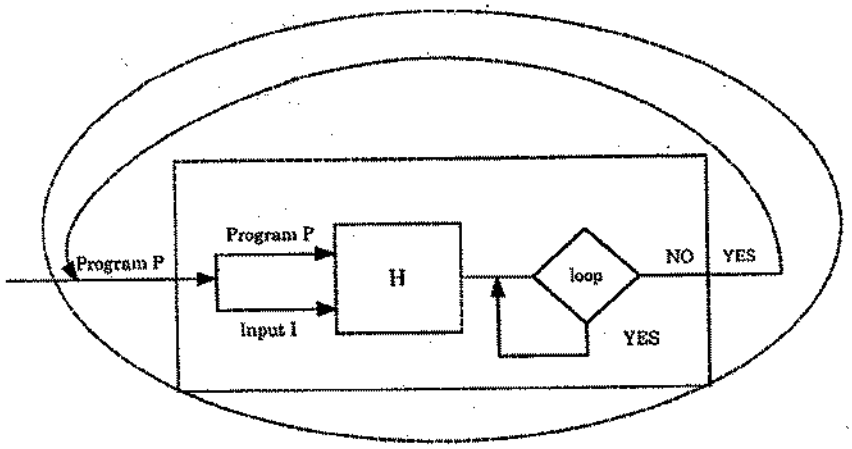
1. If H outputs "NO" then Q outputs "YES" and halts.
2. Otherwise H's output "YES" causes Q to loop forever.

If means, Q does the opposite what H does.

We define Q as follows :

```
Function Q( )
{
   if ( Function H ( ) = " NO")
   {
         return ("YES");
   }
   else
   {
         while (1);              // Loop for ever
   }
}        // End of the function Q
```

Since, Q is a program, now let us use Q as the input to itself as shown in figure(c).



Figure(c)

Now, we analyse the following :

1. If H outputs "YES" and says that Q halts then Q itself would loop (that's how we constructed it).
2. If H outputs "NO" and says that Q loops then Q outputs "YES" and will halts.

Since , in either case H gives the wrong answer for Q. Therefore, H cannot work in all cases and hence can't answer right for all the inputs. This contradicts our assumption made earlier for HP. Hence, HP is undecidable.

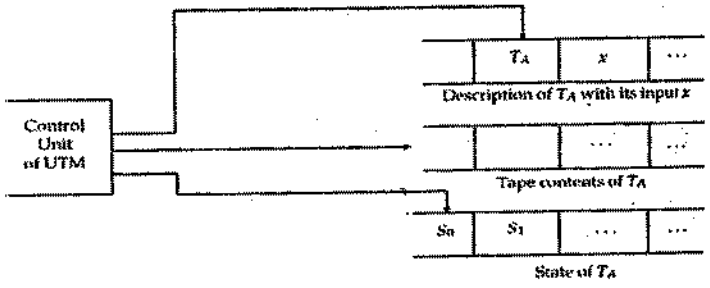
Theorem : HP of TM is undecidable.
Proof : HP of TM means to decide whether or not a TM halts for some input w. We can prove this following the similar steps discussed in above theorem.

8.6 UNIVERSAL TURING MACHINE

The Church - Turing thesis conjectured that anything that can be done on any existing digital computer can also be done by a TM. To prove this conjecture. A. M. Turing was able to construct a single TM which is the theoretical analogue of a general purpose digital computer. This machine is called a Universal Turing Machine (UTM). He showed that the UTM is capable of initiating the operation of any other TM, that is, it is a reprogrammable TM. We can define this machine in more formal way as follows :

Definition : A Universal Turing Machine (denoted as UTM) is a TM that can take as input an arbitrary TM T_A with an arbitrary input for T_A and then perform the execution of T_A on its input.

What Turing thus showed that a single TM can acts like a general purpose computer that stores a program and its data in memory and then executes the program. We can describe UTM as a 3 - tape TM where the description of TM, T_A and its input string $x \in A^*$ are stored initially on the first tape, t_1. The second tape, t_2 used to hold the simulated tape of T_A, using the same format as used for describing the TM, T_A. The third tape , t_3 holds the state of T_A



To construct a UTM, we thus require three essentials, viz.,

(i) a uniform method to describe or encode any TM into a string over a finite symbol set, I.
(ii) a similar method of encoding any input string for a TM into a string over I, and
(iii) a set of TM programs (i. e., a set of instructions for any TM) that describe the TMs basic cycle of operations.

Encoding an arbitrary TM

Since a TM can have only a finite number of configurations defined by $\langle s, a, b, s', d \rangle$, we can describe or encode any TM in terms of fixed symbols of universal Turing machine.

Let the internal states of a TM, T_A, is given by

$$S = \{S_0, S_1, S_2, ..., S_{n-1}, S_n\}$$

where S_0 is the initial state, and $S_n = H$, halting state.

Also, let the set of tape symbols be $A = \{a_0, a_1, ..., a_{m-1}\}$, where $a_0 = B$, a blank character.

We define the encoding for T_As configurations as follows :

Original	Code
S_i	1^{i+1}
a_j	1^{i+j}
R	R
L	L
N	N

We use the symbol '0' as a separator between each encoded symbol of a configuration.

For example, in the TM for parity checking, we have

$$S = \{S_0, S_1, S_2, H\}, \text{ and}$$
$$A = \{B, 0, 1, E, D,\}$$

therefore, the encoding for the configuration $\langle S_1, B, D, H, N \rangle$ will be 1101011111011110N.

Now, suppose that a Turing machine, T_A, is consisting of a finite number of configurations, denoted by, $c_0, c_1, c_2, \ldots, c_p$ and let $\overline{c}_0, \overline{c}_1, \overline{c}_2, \ldots, \overline{c}_p$ represent the encoding of them. Then, we can define the encoding of T_A as follows :

$$* \overline{c}_0 \# \overline{c}_1 \# \overline{c}_2 \# \ldots\ldots \# \overline{c}_p *$$

Here, * and # are used only as separators, and cannot appear elsewhere. We use a pair of *'s to enclose the encoding of each configuration of TM, T_A.

The case where $\delta(s,a)$ is undefined can be encoded as follows :

$$\# \overline{s}\ 0\overline{a}\ 0\overline{B} \#$$

where the symbols $\overline{s}$, $\overline{a}$ and $\overline{B}$ stand for the encoding of symbols, s, a and B (Blank character), respectively.

Working of UTM

Given a description of a TM, T_A and its inputs representation on the UTM tape, t_1 and the starting symbol on tape , t_3, the UTM starts executing the quintuples of the encoded TM as follows :

1. The UTM gets the current state from tape, t_3 and the current input symbol from tape t_2.
2. then, it matches the current state - symbol pair to the state symbol pairs in the program listed on tape, t_1.
3. if no match occurs, the UTM halts, otherwise it copies the next state into the current state cell of tape, t_3, and perform the corresponding write and move operations on tape, t_2.
4. if the current state on tape, t_3 is the halt state, then the UTM halts, otherwise the UTM goes back to step 2.

8.7 POST'S CORRESPONDENCE PROBLEM (PCP)

Post's correspondence problem is a combinatorial problem formulated by Emil Post in 1946. This problem has many applications in the field theory of formal languages.

Definition :

A correspondence system P is a finite set of ordered pairs of nonempty strings over some alphabet.

Let Σ be an alphabet, then P is finite subset of $\Sigma^+ \times \Sigma^+$. A match or solution of P is any string $w \in \Sigma$ such that pairs $(u_1, v_1), (u_2, v_2), ..., (u_n, v_n) \in P$ and $w = u_1 u_2 .... u_n = v_1 v_2 .... v_n$ for some n > 0. The selected pairs $(u_1, v_1), (u_2, v_2), ..., (u_n, v_n)$ are not necessarily distinct.

Let strings $u_1, u_2, ..., u_m$ are in U and strings $v_1, v_2, ..., v_m$ are in V, then

$U = \{ u_1, u_2, ..., u_m \}$ and $V = \{ v_1, v_2, ..., v_m \}$ for some m > 0 .

PCP is to determine whether there is any match or not for a given correspondence system.

Theorem : PCP is undecidable

PCP is undecidable just like the HP of Turing machine.

Example 1: A correspondence system $P = \{ (b, a), (ba, ba), (bab^3, b^3) \}$.

Is there any solution for P ?

Solution : We represent the P as follows :

i	u_j	v_i
1	b	a
2	ba	ba
3	bab^3	b^3

Here , $u_1 = b$, $u_2 = ba$, $u_3 = bab^3$, $v_1 = a$, $v_2 = ba$, $v_3 = b^3$.

We have a solution $w = u_3 u_1 u_1 u_2 = v_2 v_3 v_3 v_2 = bab^3 b^3 a$.

Example 2 : Consider a correspondence system P = { (b, ca), (a, ab), (ca, a), (abc, c) }. Find a match (if any).

Solution : We represent the P as follows :

i	u_j	v_i
1	b	ca
2	a	ab
3	abc	c

Here, $u_1 = b,\ u_2 = a,\ u_3 = abc,\ v_1 = ca,\ v_2 = ab,\ v_3 = c$.

We have a solution $w = u_3\ u_2 = v_2\ v_1 = abca$.

8.8 TURING REDUCIBILITY

Reduction is a technique in which if a problem A is reduced to problem B then any solution of B solves A. In general, if we have an algorithm to convert some instance of problem A to some instance of problem B that have the same answer then it is called A reduces to B.

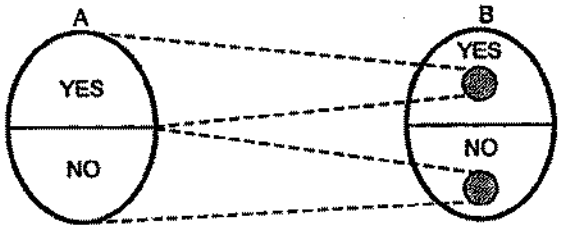


FIGURE: Reduction

Definition : Let A and B be the two sets such that $A, B \subseteq N$ of natural numbers. Then A is Turing reducible to B and denoted as $A \leq_T B$.

If there is an oracle machine that computes the characteristic function of A when it is executed with oracle machine for B.

This is also called as A is B - recursive and B - computable. The oracle machine is an abstract machine used to study decision problem. It is also called as **Turing machine** with **black box.** We say that A is Turing equivalent to B and write $A \equiv_T B$ if $A \leq_T B$ and $B \leq_T A$.

Properties :

1. Every set is Turing equivalent to its complement.
2. Every computable set is Turing equivalent to every other computable set.
3. If $A \leq_T B$ and $B \leq_T C$ then $A \leq_T B$.

8.9 DEFINITION OF P AND NP PROBLEMS

A problem is said to be solvable if it has an algorithm to solve it. Problems can be categorized into two groups depending on time taken for their execution.

1. The problems whose solution times are bounded by polynomials of small degree. **Example:** bubble sort algorithm obtains n numbers in sorted order in polynomial time $P(n) = n^2 - 2n + 1$ where n is the length of input. Hence, it comes under this group.

2. Second group is made up of problems whose best known algorithm are non polynomial example, travelling salesman problem has complexity of $O(n^2\ 2^n)$ which is exponential. Hence, it comes under this group.

A problem can be solved if there is an algorithm to solve the given problem and time required is expressed as a polynomial p(n), n being length of input string. The problems of first group are of this kind.

The problems of second group require large amount of time to execute and even require moderate size so these problems are difficult to solve. Hence, problems of first kind are tractable or easy and problems of second kind are intractable or hard.

8.9.1 P - Problem

P stands for deterministic polynomial time. A deterministic machine at each time executes an instruction. Depending on instruction, it then goes to next state which is unique.

Hence, time complexity of deterministic TM is the maximum number of moves made by M is processing any input string of length n, taken over all inputs of length n.

Definition : A language L is said to be in class P if there exists a (deterministic) TM M such that M is of time complexity P(n) for some polynomial P and M accepts L.
Class P consists of those problem that are solvable in polynomial time by DTM.

8.9.2 NP - Problem

NP stands for nondeterministic polynomial time.

The class NP consists of those problems that are verifiable in polynomial time. What we mean here is that if we are given certificate of a solution then we can verify that the certificate is correct in polynomial time in size of input problem.

Example :

Hamiltonian circuit problem. Given a directed graph $G = < V, E >$, a certificate would be a sequence $< V_1, V_2, V_3, ... V_n >$ of $|V|$ vertices. It is easy to verify in polynomial time that $(V_i + V_j + 1) \in E$ for i = 1, 2, ... |V| -and $(V_{|V|}) \in E$ as well using a nondeterministic algorithm. Hence it is in class NP. There doesnot appear any deterministic algorithms to recognize those graphs with Hamiltonian circuit. Hence it is not in class P.

A nondeterministic machine has a choice of next steps. It is free to choose any move that it wishes and if the problem has a solution one of these steps will lead to solution.

Definition : A language L is in class NP if there is a nondeterministic TM such that M is of time complexity P(n) for some polynomial P and M accepts L.

The difference between P and NP problems is analogous to difference between efficiently finding a proof of a statement (such as "This graph has Hamiltonian circuit") and efficiently verifying a proof of a statement ("i. e., checking a particular circuit is Hamiltonian"). It is easier to check a proof than finding a one.

In other words class NP consists of problems for which solution are verified quickly. P consists of problems which can be solved quickly.

Any problem in P is also in NP, but it is not yet known that P = NP. Hence, commonly believed relationship between P and NP is,

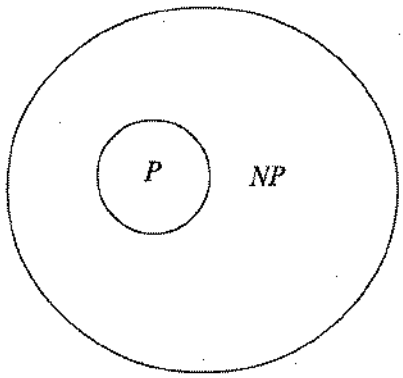


FIGURE: Relationship between P and NP Problems

8.10 NP - COMPLETE AND NP - HARD PROBLEMS

A problem S is said to be NP- Complete problem if it satisfies the following two conditions.

1. $S \in NP$, and
2. For every other problems $S_i \in NP$ for some $i = 1, 2, n$, there is polynomial - time transformation from S_i *to* S i. e. every problem in NP class polynomial - time reducible to S.

We conclude one thing here that if S_i is NP - complete then S is also NP - Complete.

As a consequence, if we could find a polynomial time algorithm for S, then we can solve all NP problems in polynomial time, because all problems in NP class are polynomial - time reducible to each other.

"A problem P is said to be NP - Hard if it satisfies the second condition as NP - Complete, but not necessarily the first condition .".

The notion of NP - hardness plays an important role in the discussion about the relationship between the complexity classes P and NP. It is also often used to define the complexity class NP - Complete which is the intersection of NP and NP - Hard. Consequently, the class NP - Hard can be understood as the class of problems that are NP - complete or harder.

Example : An NP - Hard problem is the decision problem SUBSET - SUM which is as follows.

" Given a set of integers, do any non empty subset of them add up to zero? This is a yes / no question, and happens to be NP - complete ".

There are also decision problems that are NP - Hard but not NP - Complete , for example, the halting problem of Turing machine. It is easy to prove that the halting problem is NP - Hard but not NP - Complete. It is also easy to see that halting problem is not in NP since all problems in NP are decidable but the halting problem is not (voilating the condition first given for NP - complete languages).

In Complexity theory, the **NP - complete** problems are the hardest problems in NP class, in the sense that they are the ones most likely not to be in P class. The reason is that if we could find a way to solve any NP - complete problem quickly, then you could use that algorithm to solve all NP problems quickly.

At present time, all known algorithms for NP - complete problems require time which is exponential in the input size. It is unknown whether there are any faster algorithms for these are not.

S. A. Cook in 1971 proved that the Boolean satisfiability problem is NP - Complete. After Cook's original results, thousands of other problems have been shown to be NP - complete by reductions from other problems previously shown to be NP - complete.

Example : Consider an interesting problem in graph theory known as " Graph isomorphism". Two graphs are isomorphic if one can be transformed into the other simply by renaming vertices. Consider these two problems given as follows :

Graph Isomorphism : Is graph G_1 isomorphic to graph G_2 ?

Subgraph Isomorphism : Is graph G_1 isomorphic to a subgraph of graph G_2 ?

The " Subgraph Isomorphism" problem is NP - complete, but the " Graph Isomorphism" problem is suspected to be neither in P nor in NP - Complete, though it is obviously in NP. This is an example of a problem that is thought to be hard, but it is not thought to be NP - Complete.

Following are some other NP - complete and NP - Hard problems :

(1) The Boolean Satisfiability Problem (SAT)

In mathematics, a formula of propositional logic is said to be satisfiable if truth - values can be assigned to its free variables in such a way that this assignment makes the formula true. The class of satisfiable propositional formulae is NP - Complete problem.

Consider the logical operators defined as follows :

And : This is denoted by $\wedge$ and $0 \wedge 1 = 1 \wedge 0 = 0$, $0 \wedge 0 = 0, 1 \wedge 1 = 1$,

OR : This is denoted by $\vee$ and $0 \vee 1 = 1 \vee 0 = 1$, $1 \vee 1 = 1$, $0 \vee 0 = 0$, and

NOT : This is denoted by ' and $0' = 1$, $1' = 0$.
Now, consider the expressions

(a) $E_1 = x' \vee y$, where x, y are variables ; either 0 or 1 So, $E_1 = 1$ if $x = 0$ or $y = 1$
Therefore, E_1 is satisfiable for $x = 0$ or $y = 1$.

(b) $E_2 = (x \vee y) \wedge x' \wedge y'$ is not satisfiable because every assignment for the variables x and y will make the value of $E_2 = 0$.

(2) The Travelling Salesman or Salesperson Problem

The problem is defined as follows .

"Given a number of cities and the cost of travelling from one to the other, what is the cheapest roundtrip route that visits each city and then returns to the starting city ?"

The most direct answer would be to try all the combinations and see which one is cheapest, but given that the number of combinations of cities is n ! (factorial n), this solution becomes impractical for larger n, where n is the number of cities.

How fast are the best known deterministic algorithms ?

This problem has been shown to be NP - Hard, and the decision version of it which is given below :

" Given the costs of routes between cities and a number N, decide whether there exists a tour program for salesman to visit all the cities so that the total cost is less than or equal to N."

The above version of salesman problem is NP - Complete problem.

(3) The Hamiltonian Cycle or Hamiltonian Circuit Problem

This problem is in graph theory to find a path through a given graph which starts and ends at the same vertex and includes each vertex exactly once.

This is a special case of the travelling salesman problem obtained by setting the distance between two cities to unity if they are adjacent and infinity otherwise. Like the traveling salesman problem, the Hamiltonian cycle problem is NP - Complete.

(4) The Vertex Cover Problem

This problem is stated as follows .

" Given a graph G and a natural number K, does there exist a vertex covering for G with K vertices."

This is NP - complete problem.