4AID4-06: Theory Of Computation

Credit: 3 **Max. Marks: 100(IA:30, ETE:70)**

3L+0T+0P **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:** Objective, scope and outcome of the course.	1
2	Finite Automata & Regular Expression: Basic machine, Finite state machine, Transition graph, Transition matrix, Deterministic and non-deterministic finite automation, Equivalence of DFA and NDFA, Decision properties, minimization of finite automata, Mealy & Moore machines. Alphabet, words, Operations, Regular sets, relationship and conversion between Finite automata and regular expression and vice versa, designing regular expressions, closure properties of regular sets, Pumping lemma and regular sets, Myhill- Nerode theorem , Application of pumping lemma, Power of the languages.	7
3	Context Free Grammars (CFG), Derivations and Languages, Relationship between derivation and derivation trees, leftmost and rightmost derivation, sentential forms, parsing and ambiguity, simplification of CFG, normal forms, Greibach and Chomsky Normal form , Problems related to CNF and GNF including membership problem.	8
4	Nondeterministic PDA, Definitions, PDA and CFL, CFG for PDA, Deterministic PDA, and Deterministic PDA and Deterministic CFL , The pumping lemma for CFL's, Closure Properties and Decision properties for CFL, Deciding properties of CFL.	8
5	Turing Machines: Introduction, Definition of Turing Machine, TM as language Acceptors and Transducers, Computable Languages and functions, Universal TM & Other modification, multiple tracks Turing Machine. Hierarchy of Formal languages: Recursive & recursively enumerable languages, Properties of RL and REL, Introduction of Context sensitive grammars and languages, The Chomsky Hierarchy.	8
6	Tractable and Untractable Problems: P, NP, NP complete and NP hard problems, Un-decidability, examples of these problems like vertex cover problem, Hamiltonian path problem, traveling sales man problem.	8
	Total	40

Office of Dean Academic Affairs
Rajasthan Technical University, Kota

THEORY OF COMPUTATION

IV C.S.

1 FINITE AUTOMATA AND REGULAR EXPRESSION

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 What is Moore Machine? *[R.T.U. 2019]*

Ans. Moore Machine : Moore Machine is an FSM whose outputs depend on only present state. It can be described by a 6 tuple $(Q, \Sigma, O, \delta, X, g_0)$ where :

- Q is a finite set of states.
- Σ is a finite set of symbols called input alphabets.
- O is a finite set of symbols called output alphabets.
- δ is input transition function where $\delta : Q \times \Sigma \rightarrow Q$
- X is the output transition function where $X : Q \times \Sigma \rightarrow 0$
- g_0 is initial state from where any input is processed ($g_0 \in Q$). State diagram is as shown below :

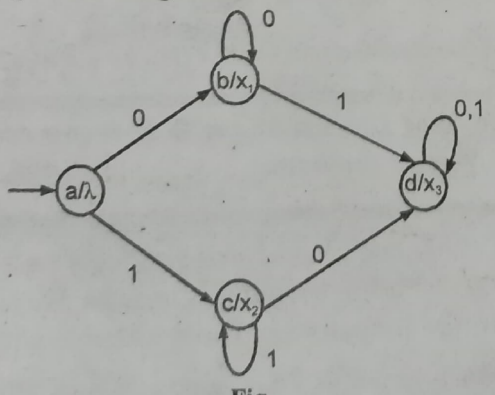


Fig.

Q.2 What is NDFA? *[R.T.U. 2019]*

Ans. Non-deterministic Finite Automaton (NDFA) : In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called **Non-deterministic Automaton**. As it has finite number of states, the machine is called **Non-deterministic Finite Machine** or **Non-deterministic Finite Automaton**.

Formal Definition of an NDFA

An NDFA can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where :

- **Q** is a finite set of states.
- Σ is a finite set of symbols called the alphabets.
- δ is the transition function where $\delta : Q \times \Sigma \rightarrow 2^Q$

(Here the power set of Q (2^Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states.)

- q_0 is the initial state from where any input is processed ($q_0 \in Q$).
- **F** is a set of final state/states of Q ($F \subseteq Q$).

Q.3 What is finite automata? *[R.T.U. 2019]*

Ans. Finite Automaton : A finite automaton can be represented by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

(i) Q is a finite non-empty set of states.

(ii) Σ is a finite non-empty set of inputs called input alphabets.

(iii) δ is a function which maps $Q \times \Sigma$ into Q and is usually called direct transition function. This is the function which describes the changes of states during the transition. This mapping is usually represented by a transition table or a transition diagram.

(iv) $q_0 \in Q$ is the initial state.

(v) $F \subseteq Q$ is the set of final states. It is assumed here that there may be more than one final state.

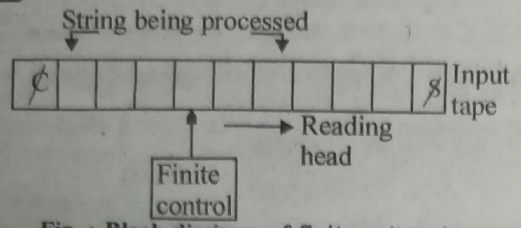


Fig. : Block diagram of finite automaton

(vi) Input Tape : The input tape is divided into squares, each square containing a single symbol from the input alphabet Σ. The end square of the tape contain end-markers ¢ at the left end and $ at the right end. Absence of end-markers indicates that the tape is of infinite length. The left-to-right sequence of symbol between the end markers is the input string to be processed.

(vii) Reading Head : The head examines only one square at a time and can move one square either to the left or to the right. For further analysis, we restrict the movement of R-head only to the right side.

(viii) Finite Control : The input to the finite control will be usually : symbol under the R-head, say a, or the present state of the machine, say q, to give the following outputs : (a) A motion of R-head along the tape to the next square (In some a null move *i.e.* R-head remaining to the same square is permitted); (b) The next state of a finite state machine given by δ (q, a).

Initial states are q_0 and q_1.

Final state is q_3.

For checking the string acceptability, we start the string from initial state. If we reach the final state after completing the string, then we say that this string is accepted by transition system or not.

Q.4 What are graphs? ***[R.T.U. 2019]***

Ans. Graph : A graph is a picture designed to express words, particularly the connection between two or more quantities.

A simple graph usually shows the relationship between two numbers or measurements in the form of a grid. If this is a *rectangular graph* using Cartesian coordinate system, the two measurements will be arranged into two different lines at right angle to one another. One of these lines will be going up (the *vertical* axis). The other one will be going right (the *horizontal* axis). These lines (or *axes*, the plural of axis) meet at their ends in the lower left corner of the graph.

Both of these axes have *tick marks* along their lengths. So each measurement is indicated by the length of the associated tick mark along the particular axis.

A graph is a kind of chart or diagram. However, a chart or a diagram may not relate one quantity to other quantities. *Flowcharts* and *tree diagrams* are charts or diagrams that are not graphs.

Q.5 ***$M = (\{q_1, q_2, q_3\}, \{0, 1\}, \delta, q_1, \{q_3\})$ is nondeterministic finite automaton, where δ is given by***

$\delta(q_1, 0) = \{q_2, q_3\}$,
$\delta(q_1, 1) = \{q_1\}$
$\delta(q_2, 0) = \{q_1, q_2\}$,
$\delta(q_2, 1) = \phi$
$\delta(q_3, 0) = \{q_2\}$,
$\delta(q_3, 1) = \{q_1, q_2\}$

Construct an equivalent DFA. ***[R.T.U. 2016]***

Ans. $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \delta, q_1, \{q_3\})$

Where :

$\delta(q_1, 0) = \{q_2, q_3\}$, $\delta(q_1, 1) = \{q_1\}$
$\delta(q_2, 0) = \{q_1, q_2\}$, $\delta(q_2, 1) = \phi$
$\delta(q_3, 0) = \{q_2\}$, $\delta(q_3, 1) = \{q_1, q_2\}$

The DFA equivalent of this NFA can be obtained as follows :

State	0	1
$\{q_1\}$	$\{q_2, q_3\}$	$\{q_1\}$
$\{q_2\}$	$\{q_1, q_2\}$	ϕ
$\{q_3\}$	$\{q_2\}$	$\{q_1, q_2\}$
ϕ	ϕ	ϕ

The transition diagram associated with this DFA is shown in following figure.

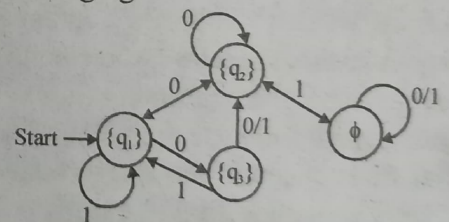


Fig. : Transition diagram for $M = (\{q_1, q_2, q_3\}, \{0, 1\}\ \delta, q_1, \{q_3\})$

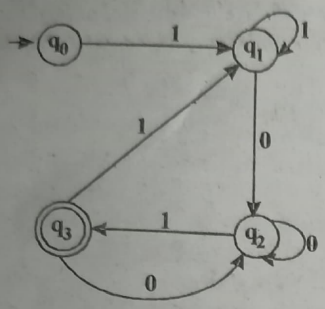
Q.6 ***Show that $L = \{a^n b^n : n \geq 1\}$ is not regular using Mayhill-Nerode theorem.*** ***[R.T.U. 2013, 2012]***

Ans. Let us assume a set of strings $S = \{a^n \mid n > 0\}$. The set S is over alphabet {a, b} and it is infinite. We have to show that its strings are pairwise distinguishable with respect to language L.

Assume that a^i and a^j be arbitrary two different members of the set S, where i and j are positive integers and $i \neq j$.

Let b^j as a string to be appended to a^i and a^j. It then becomes a^ib^j and a^jb^j. From these two strings a^ib^j is not L while a^jb^j is in L. hence we can say that a^i and a^j are arbitrary strings of S and are distinguishable with respect to L. Thus S satisfies conditions of Myhil-Nerode theorem. This proves that L is non-regular.

Q.7 ***Find the regular expression corresponding to the given automaton.***



[Raj.Univ. 2007]

Ans. We get the following equations :

$$q_0 = \wedge$$

$$q_1 = q_0 1 + q_1 1 + q_3 1$$

$$q_2 = q_1 0 + q_2 0 + q_3 0$$

$$q_3 = q_2 1$$

Therefore,

$$q_2 = q_1 0 + q_2 0 + q_2 10 = q_1 0 + q_2 (0+10)$$

Applying Arden's theorem, we get

$$q_2 = q_1 0(0+10)^*$$

Now $q_1 = 1 + q_1 1 + q_2 11 = 1 + q_1 1 + q_1 0(0+10)^* 11$

$$= 1 + q_1 (1 + 0(0+10)^* 11)^*$$

By Arden's theorem we get,

$$q_1 = 1(1+0(0+10)^*11)^*$$

$$q_3 = q_2 1 = 1(1+0(0+10)11)^* 0(0+10)^* 1$$

As q_3 is only the final state, the regular expression corresponding to the given diagram is

$$1(1+0(0+10)^*11)^*0(0+10)^*1$$

Q.8 ***Define finite state machine.***

Ans. Finite State Machine (FSM) : Finite state machine model is a simple, widely known and important model for describing control aspects. It is a model of the system to describe an entity which is characterized by its operations and behavior. Such machines are basically appropriate tools to express the requirements and design specifications of software product.

FSM models are represented by a set of states and specifies transitions between the states. The transition from one state to another state is based on the input. So, such models show system states and events which cause transition from one state to another.

A Finite State Machine (FSM) consists of a finite set of symbols that are given below.

J = a finite set of states

ε = a finite set of input

S = the starting state

δ = a transition, from (input × state) to 2 state ξ,

which can be a partial function, *i.e.* can be undefined for some values of its domains.

δ : Input × state → state

$\delta : \varepsilon \times J \rightarrow J$

These notations are used to make the finite state machine diagram or transition tables. To draw a diagram, first, we draw a circle for each state in the diagram. Then for each state and possible input, we draw a directed arrow to indicate the transition from one state to another.

Q.9 ***Define in brief Discrete Automaton.***

Ans. An automaton is an abstract model of a digital computer. It has a mechanism to read input, which is a string over a given alphabet. In computer science, the term automaton means **discrete automaton** and is defined in some abstract way as shown in Figure.

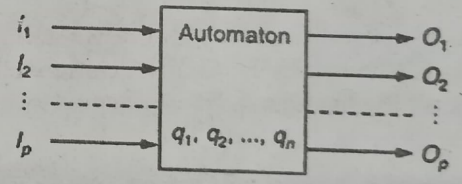


Fig. : Model of a discrete automaton

Q.10 ***Difference between melay & moore machine.***

Ans. Mealy Machine v/s Moore Machine

S. No.	Mealy Machine	Moore Machine
(i)	Output depends both upon present state and present input.	Output depends only upon present state.
(ii)	Generally, it has fewer states than Moore machine.	Generally, it has more states than Mealy machine
(iii)	Output changes at clock edges.	Input change can cause change in output change as soon as logic is done.
(iv)	Mealy machines react faster to inputs.	In moore machines, more logic is needed to decode the output since it has more circuit delays.

Q.11 ***Give applications of pumping lemma.***

Ans.(i) Assume L is regular : Let n be the number of state in the FA which accepts L.

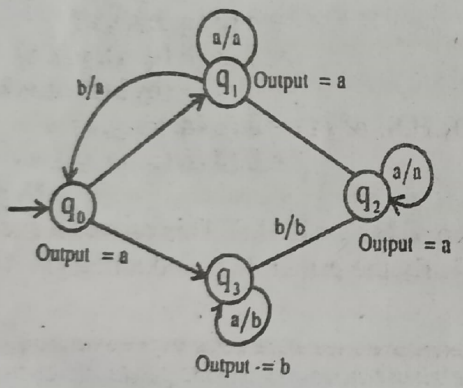
(ii) Choose a string w such that $|w| \geq n$. Using pumping lemma write $w = xyz$ with $|xy| \leq n$ and $|y| > 0$.

(iii) Find suitable integer i such that $xy^i z \notin L$. So it contradicts the assumption we have made in step (i) above. Hence L is not regular.

It is then clear that, the most important aspect of the method is to find i such that $xy^i z \notin L$. Some times we have to prove $xy^i z \notin L$ by considering $|xy^i z|$ or some times we heed to use the structure of strings in L.

PART-B

Q.12 Consider the Moore Machine shown in figure. What is the output for the input ababa?



[R.T.U. 2019]

Ans. For the input = ababa

$$\underset{q_0}{a} \xrightarrow{a} \underset{q_1}{a} \xrightarrow{b} \underset{q_0}{a} \xrightarrow{a} \underset{q_1}{a} \xrightarrow{b} \underset{q_0}{a} \xrightarrow{a} \underset{q_1}{a}$$

The ouput for the input string = ababa is aaaaaa

Q.13 Convert the following Moore Machine into Mealy Machine :

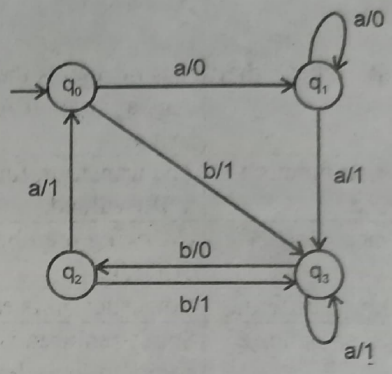
State	Input		Output
	a	b	
$\rightarrow q_0$	q_1	q_3	1
q_1	q_3	q_1	0
q_2	q_0	q_3	0
q_3	q_3	q_2	1

[R.T.U. 2019]

Ans. Mealy Machine :

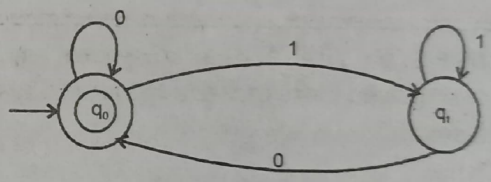
Next State

P.S.	i = a	o/p	i/p = b	o/p
$\rightarrow q_0$	q_1	0	q_3	1
q_1	q_3	1	q_1	0
q_2	q_0	1	q_3	1
q_3	q_3	1	q_2	0



Q.14 Design a FA which checks whether the given binary number is even. *[R.T.U. 2019]*

Ans.



Q.15 ***Explain the difference between deterministic and non-deterministic finite automaton.*** *[R.T.U. 2019]*

OR

State the difference between deterministic and non deterministic finite automata.

[R.T.U. 2015, 2014]

Ans. Deterministic and Non-deterministic Finite Acceptors

S. No.	Deterministic Finite Acceptors	Non- deterministic Finite Acceptors
1.	For every symbol of the alphabet, there is only one state transition.	We do not need to specify how does the NFA react according to some symbol.
2.	Cannot use empty string transition.	Can use empty string transition.
3.	Can be understood as one machine.	Can be understood as multiple little machines computing at the same time.
4.	It will reject the string if it end at other than accepting state.	If all of the branches of NFA dies or rejects the string, we can say that NFA reject the string.

S. No.	Deterministic Finite Acceptors	Non- deterministic Finite Acceptors
1.	For every symbol of the alphabet, there is only one state transition.	We do not need to specify how does the NFA react according to some symbol.
2.	Cannot use empty string transition.	Can use empty string transition.
3.	Can be understood as one machine.	Can be understood as multiple little machines computing at the same time.
4.	It will reject the string if it end at other than accepting state.	If all of the branches of NFA dies or rejects the string, we can say that NFA reject the string.
5.	The transition function is single valued.	The transition function is multi-valued.
6.	Checking membership is easy.	Checking membership is difficult.
7.	Construction is difficult.	Construction is easy.
8.	Space required is more.	Space required is comparatively less.
9.	Backtracking is allowed.	Not possible in every case.
10.	Can be constructed for every input and output.	Cannot be constructed for every input and output.
11.	There can be more than one final state.	There can only be one final state.

Q.16 ***(a) Describe the block diagram of a finite automaton. Consider the transition system given below :***

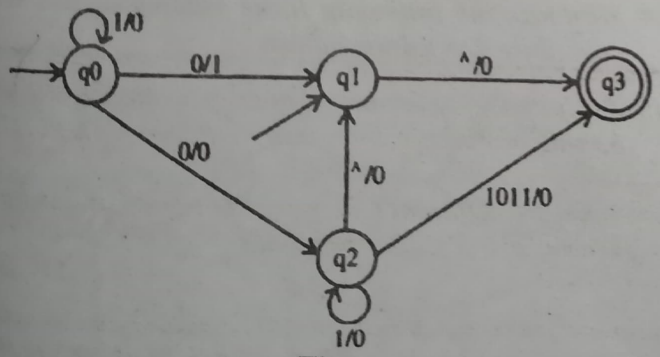


Fig.

Determine the initial states, the final state and the acceptability of 101011 and 111010.

(b) Prove that for any transition function δ and for any two input string x and y.

$\delta(q, xy) = \delta(\delta(q, x), y)$ *[R.T.U. 2016]*

Ans.(a) Finite Automaton : *Refer to Q.3.*

For string 101011, the path value is $q_0\ q_0\ q_2\ q_3$. Since q_3 is the final state so this string is accepted by above transition system.

For string 111010, there is no path value. So this string is not accepted by the above transition system.

Ans.(b) Properties of Transition Functions

(1) $\delta(q, \wedge) = q$ is a finite automaton. This means the state of the system can be changed only by an input symbol.

(2) For all strings w and input symbols a,

$$\delta(q, aw) = \delta(\delta(q, a), w)$$
$$\delta(q, wa) = \delta(\delta(q, w), a)$$

This property gives the state after the automaton consumers or reads the first symbol of a string aw and the state after the automation consumers a prefix in of the string wa.

Example : Prove that for any transition function δ and for any two input string x and y,

$$\delta(q, xy) = \delta(\delta(q, x), y) \quad \ldots(1)$$

Proof : By the method of induction on $|y|$, i.e. length of y.

Basis : When $|y| = 1$, $y = a \in \Sigma$,

L.H.S. of equation (1)

$= \delta(q, xa) = \delta(\delta(q, x), a)$ (by property 2)

$= $ R.H.S. of (1)

Assume the result, *i.e.* (1) for all strings x and string y with $|y| = n$. Let y be a string of length $n + 1$. Write $y_1 = y, a$ where $|y| = n$.

L.H.S of (1)

$= \delta(q, xy_1 a) = \delta(q, xy_1 a), x_1 = xy_1$

$= \delta(\delta(q, x_1), a)$ (by property 2)

$= \delta(\delta(q, xy_1), a)$

$= \delta(\delta(\delta(q, x), y_1), a)$

(by induction hypothesis)

R.H.S. of (1) $= \delta(\delta(q, x), y, a)$

$= \delta(\delta(\delta(q, x), y_1), a)$

(by property 2)

Hence, L.H.S. = R.H.S. This proves for any string of length $n + 1$. By the principle of induction (1) is true for all strings.

Q.17 ***Write short note on Finite State Machine (FSM).***

[RTU 2017, 2016, 2013, 12]

OR

Explain Finite State Machine (FSM) Models.

[RTU 2014]

Ans. Finite State Machine (FSM) : *Refer to Q.8.*

Example 1 : Draw a FSM model for the following description :

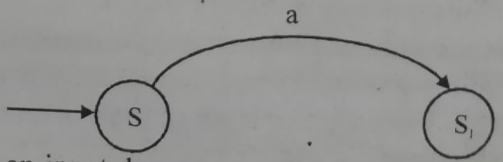
$T = \{S, S_1\}$

$\varepsilon = \{a\}$

$S =$ starting state

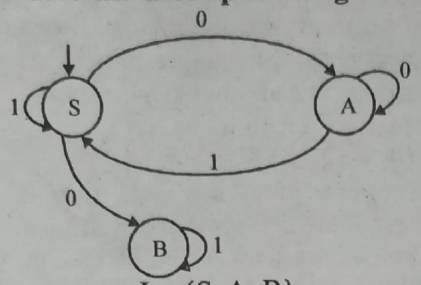
$\delta(S, a) = S_1$

Ans.



When an input does not result in a state change, we can indicate this in the diagram by showing an arrow originating and ending in the same state.

Example 2 : Give the description of given FSM :



Ans. J = {S, A. B}

ε = {0, 1}, S = Starting state

δ (S, 0) = A, δ(S, 1)= S

δ (A, 0) = A, δ(A, 1) = S

δ (B, 0) = B , δ (B, 1) = B

Example 3 : The FSA in given fig. recognizes the following set of words : {bet, ball, beg} :

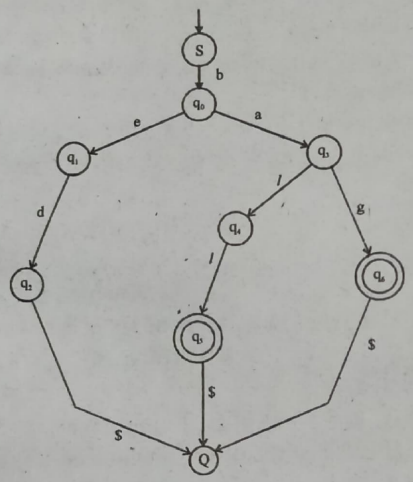


Fig.

Ans. ε = {b, e, d, a, l, g}

J = {s, q_0, q_1, q_2, q_3, q_4, q_5, q_6, Q }

S = Starting state

A = Q (Accepting state)

$\delta(s, b) = q_0$, $\delta(q_3, 1) = q_4$

$\delta(q_0, e) = q_1$, $\delta(q_3, g) = q_6$

$\delta(q_0, a) = q_3$, $\delta(q_4, 1) = q_5$

$\delta(q_1, d) = q_2$, $\delta(q_5, \$) = Q$

$\delta(q_2, \$) = Q$ $\delta(q_6, \$) = Q$

and $f = \{q_2, q_5, q_6\}$

Q.18 ***Discuss Mealy & Moore machines.*** *[R.T.U. 2015]*

Ans. Finite automata may have outputs corresponding to each transition. There are two types of finite state machine that generate output

(i) Mealy Machine

(ii) Moore Machine

Mealy Machine:

Mealy Machine is an FSM whose output depends on present state as well as present input

It can be described by a six tuple $(Q, \Sigma, O, \delta, X, g_0)$ where :

- Q is finite set of states
- Σ is finite set of symbols called input alphabets
- O is finite set of symbols called output alphabets
- δ is input transition function where $\delta : Q \times \Sigma \rightarrow Q$
- X is output transition function where $X : Q - O$
- g_0 is initial state from where any input is processed ($g_0, \in Q$). State diagram of Mealy machine is shown below :

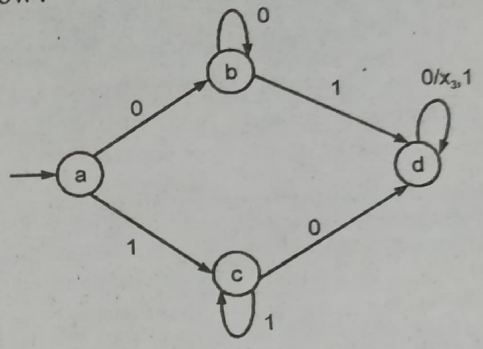


Fig.

Moore Machine: *Refer to Q.1.*

Difference between Mealy and Moore Machine : *Refer to Q. 10.*

Q.19 ***Minimize the following finite automata. Also write procedure for minimization.***

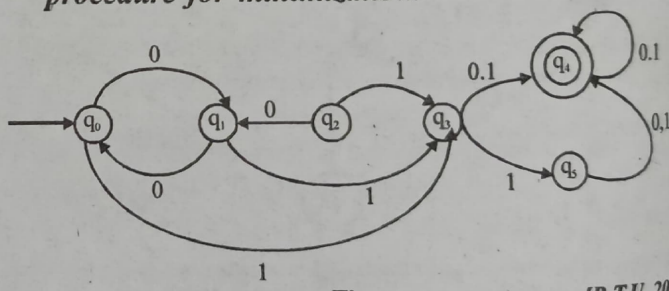


Fig. ***[R.T.U. 2014]***

Ans. Step 1 : Firstly we make state transition table for above given state transition diagram is given by as follows.

State / Σ	Input	
	0	1
q_0	q_1	q_3
q_1	q_0	q_3
q_2	q_1	q_3
q_3	q_4	$[q_4, q_5]$
(q_4)	q_4	q_4
q_5	q_4	q_4

Step 2 : Now we obtain π_o by use of grouping

$$\pi_o = \{\{q_4\}, \{q_0, q_1, q_2, q_3, q_5\}\}$$

Step 3 : Partition set of non final states in such a way that equivalent states are grouped together in separate sets. This is π_1 equivalence criteria used here is:

Two states are said to be equivalent if their state transitions corresponding to same input belong to same set.

Step 4 : The step 3 is repeated for π_2, π_3 and so on until we get

$$\pi_k = \pi_{k+1}$$

where k is any integer and π_{k+1} is required solution.

Step 5 : In this step state transition table is generated in such a way that the set obtained in π_{k+1} are considered to be as one state, the rest of state transitions in the state transition table goes according to original state transition table.

Step 6 : State transition diagram is generated from state transition table.

after applying step 3 we get π_1

$$\pi_1 = \{\{q_4\}, \{q_0, q_1, q_2\}, \{q_3\}, \{q_5\}\}$$

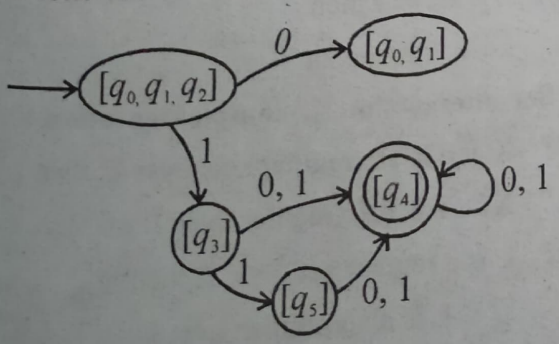
$$\pi_2 = \{\{q_4\}, \{q_0, q_1, q_2\}, \{q_3\}, \{q_5\}\}$$

we get $\pi_2 = \pi_1$ so we stop here.

Now we generate state transition table for minimizing finite automata. The transition table is as follows:

State / Σ	Input	
	0	1
$[q_0, q_1, q_2]$	$[q_0, q_1]$	$[q_3]$
$[q_3]$	$[q_4]$	$[q_4, q_5]$
$[q_4]$	$[q_4]$	$[q_4]$
$[q_5]$	$[q_4]$	$[q_4]$

State transition diagram for above state transition table is given as follows :



Q.20 ***Construct a deterministic finite automata equivalent to following NDFA.***

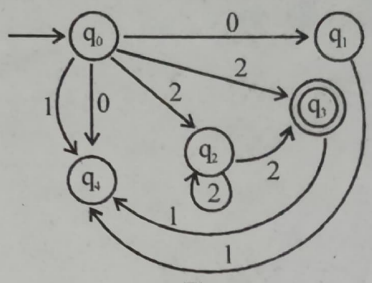
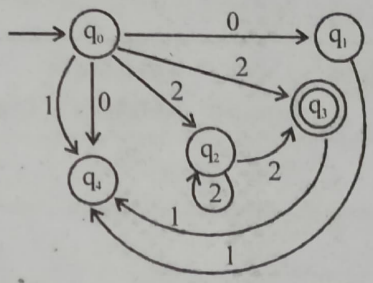


Fig. *[R.T.U. 2013]*

Ans.

Given, NDFA



Step (1) : Find transition table of given NDFA

Table : State table of given NDFA

State (Σ)	0	1	2
→ q_0	q_1, q_4	q_4	q_3, q_2
q_1		q_4	
q_2			q_3, q_2
q_3		q_4	
q_4			

Step (2) : Now,

(i) Start with starting state that is q_0 we get $[q_1, q_4]$, $[q_4]$ and $[q_3, q_2]$ as new states.

(ii) Then we construct δ (transition function) for $[q_4]$ and we do not get any new state.

(iii) Then we construct δ for $[q_1, q_4]$ and we do not get any new state.

(iv) Then we construct δ for $[q_3, q_2]$ and we do not get any new state.

Step (3) : Now, we construct the table for DFA of given NDFA using states generated in step (2).

Table : State table for equivalent DFA of given NDFA

State (Σ)	0	1	2
$[q_0]$	$[q_1, q_4]$	$[q_4]$	$[q_3, q_2]$
$[q_4]$	ϕ	ϕ	ϕ
$[q_1, q_4]$	ϕ	$[q_4]$	ϕ
$[q_3, q_2]$	ϕ	$[q_4]$	$[q_3, q_2]$

Q.21 ***Consider a Mealy machine given by transition diagram. Construct a moore machine equivalent to this mealy machine.***

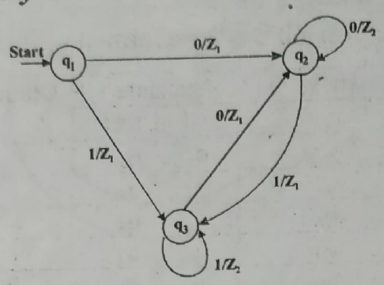


Fig. *[R.T.U. 2013, 2009]*

Ans. First of all we have to convert the transition diagram into transition table as follows :

Present state	Next state			
	Input 0	Output	Input 1	Output
$\rightarrow q_1$	q_2	Z_1	q_3	Z_1
q_2	q_2	Z_2	q_3	Z_1
q_3	q_2	Z_1	q_3	Z_2

Step 1 : We look in the next state column for states q_1, q_2 and q_3.

q_1 is associated with no output, q_2 and q_3 are associated with two different outputs *i.e.* Z_1 and Z_2.

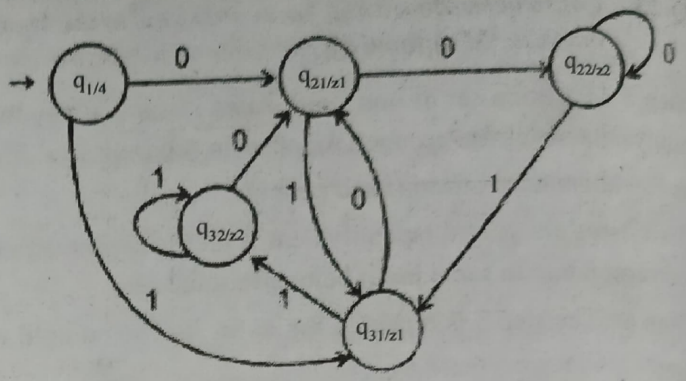
Step 2 : We don't split q_1 now we split q_2 and q_3 into q_{21}, q_{22} and q_{31}, q_{32} respectively. After splitting these state, the transition table we get is as follows :

Present state	Next state			
	Input 0	Output	Input 1	Output
$\rightarrow q_1$	q_{21}	Z_1	q_{31}	Z_1
q_{21}	q_{22}	Z_2	q_{31}	Z_1
q_{22}	q_{22}	Z_2	q_{31}	Z_1
q_{31}	q_{21}	Z_1	q_{32}	Z_2
q_{32}	q_{21}	Z_1	q_{32}	Z_2

Step 3 : Now we rearrange the transition table in such a way that each state in present state column is associated with a single output.

Present state	Next state		Output
	Input 0	Input 1	
$\rightarrow q_1$	q_{21}	q_{31}	0
q_{21}	q_{22}	q_{31}	Z_1
q_{22}	q_{22}	q_{31}	Z_2
q_{31}	q_{21}	q_{32}	Z_1
q_{32}	q_{21}	q_{32}	Z_2

The transition diagram for Moore Machine.



This is the required solution.

Q.22 ***Write closure property of regular set.*** *[R.T.U. 2013, 2012]*

Ans. The closure properties of regular sets are as follows:

(i) Set union
(ii) Concatenation
(iii) Closure (iteration)
(iv) Transpose
(v) Set intersection
(vi) Complementation

We can understand these all closure properties as follows:

(i) Set Union: The union of two regular expressions R_1 and R_2, written as $R_1 + R_2$ is also a regular expression.

(ii) Concatenation: The concatenation of two regular expressions R_1 and R_2, written as R_1R_2 is also a regular expression.

(iii) Closure (iteration): The closure (or iteration) of a regular expression R, written as R^* is also a regular expression.

(iv) Transpose :

If L is regular then

L^T is also regular

(v, vi) Set intersection & complementation :

If X and Y are regular sets over Σ, then

$X \cap Y$ is also regular over Σ

If L is a regular set over Σ, then

$\Sigma^* - L$ is also regular over Σ.

Q.23 ***Construct a Moore machine equivalent to the Mealy machine M defined by the table given below :***

Present state	Next state			
	a = 0		a = 1	
	State	o/p	State	o/p
q_1	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

[R.T.U. 2012]

Ans. Convert Mealy machine equivalent to the Moore machine :

Present State	Next State			
	a=0	Output	a=1	Output
$\rightarrow q_1$	q_1	1	q_2	0
q_2	q_4	1	q_4	1
q_3	q_2	1	q_3	1
q_4	q_3	0	q_1	1

Step 1 : We look in the next state column for q_1, q_2, q_3 and q_4

q_1 is associated with one output : 1

q_2 is associated with two outputs : 0 & 1

q_3 is associated with two outputs : 0 & 1

q_4 is associated with one output : 1

Step 2 : Now we split q_1 and q_3 into q_{20}, q_{21} & q_{30}, q_{31} respectively.

Present State	Next State			
	a=0	Output	a=1	Output
$\rightarrow q_1$	q_1	1	q_{20}	0
q_{20}	q_4	1	q_4	1
q_{21}	q_4	1	q_4	1
q_{30}	q_{21}	1	q_{31}	1
q_{31}	q_{21}	1	q_{31}	1
q_4	q_3	0	q_1	1

Step 3 : Now rearranging the transition table in such a way the each state in present state column is associate with a single output.

Present state	Next state		Output
	a=0	a=1	
$\rightarrow q_1$	q_1	q_{20}	1
q_{20}	q_4	q_4	0
q_{21}	q_4	q_4	1
q_{30}	q_{21}	q_{31}	0
q_{31}	q_{21}	q_{31}	1
q_4	q_3	q_1	1

This is Moore machine.

Note : Here, we see that the initial state q_1 is associated with output 1. We add a new starting state q_0 whose transitions are similar to q_1 but output is 0.

So state transition table is transformed as follows :

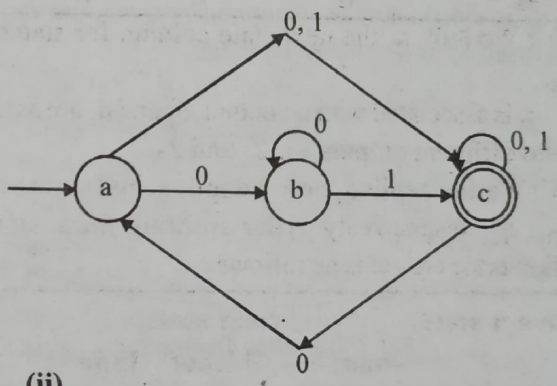
Present state	Next state		Output
	a=0	a=1	
$\rightarrow q_0$	q_1	q_{20}	0
q_1	q_1	q_{20}	1
q_{20}	q_4	q_4	0
q_{21}	q_4	q_4	1
q_{30}	q_{21}	q_{31}	0
q_{31}	q_{21}	q_{31}	1
q_4	q_3	q_1	1

Q.24 ***Let $\Sigma = \{a, b, c\}$***

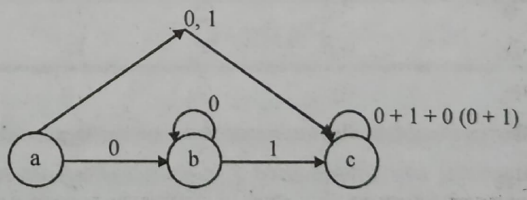
(i) Draw a DFA that rejects all words for which the last two letters match.

(ii) Draw a DFA that rejects all words for which the first two letters match. *[R.T.U. 2010]*

Ans.(i)



(ii)



Q.25 ***A public telephone (PCO) accepts one or rupees two coins. The call can be made only when the total amount inserted is rupees two. Suppose the telephone has two LED : GREEN and RED. The GREEN LED is set when the call is being made. The RED LED is set when the total amount inserted is rupees three or more. This call is possible only when, RED is off. Construct a DFA corresponding to this machine.*** *[R.T.U. 2009]*

Ans. PCO Telephone Light

Input	1	2
q_0	q_1	q_f
q_1	q_f	q_3
q_3	q_3	q_3
q_f	–	–

The States

q_0 = Initial state q_1 = When amount inserted is 1.

q_3 = Red light glowing *i.e.* amount more than 2.

q_f = Green light, call can be made.

In the initial state q_0, when 1 rupee coin is inserted, it goes to state q_1 where it waits for another 1 rupee.

In the q_1, 1 rupee is inserted and it waits for another '1'.

If '1' then 'q_f' *i.e.* call can be made.

If '2' then 'q_3' *i.e.* red light glows as total amount is $1+2=3$.

In q_3, any rupee entered will continue glowing red light as amount ≥ 3.

q_f is the final state, where correct amount *i.e.* rupees two is inserted and call can be made.

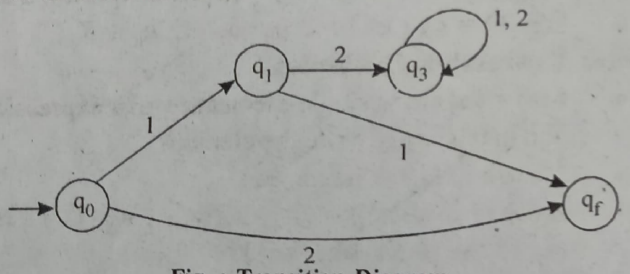


Fig. : Transition Diagram

PART-C

Q.26 Explain the procedure for minimization of finite automata with example. ***[R.T.U. 2019, 13]***

OR

Describe the algorithm to minimize number of states in a finite automaton. ***[Raj. Univ. 2003]***

Ans. Procedure : We construct an automaton with minimum number of states equivalent to given automaton M.

ALGO

Step 1: When we are asked to minimize a finite automaton, we are given a state transition table or a state transition diagram. The state transition diagram must be converted into a state transition table.

Step 2: Now we apply the procedure for partitioning Q, where partitions are denoted by π_0, π_1, π_2... and so on. π_0 is obtained by grouping final states in one set and non-final states in another state.

Step 3: Now we leave the set containing the final state as such and partition the other set of non-final states in separate status. This is π_1.

Step 4: The step 3 is repeated for π_2, π_3 and so on until we get $\pi_k = \pi_k + 1$. Where k is any integer. π_k+1 is the required solution.

Step 5: In this step the state transition table is generated in such a way that the states obtained in π_k+1 are considered to be as one state. The rest of the state transition in the state transition table goes according to the original state transition table.

Step 6: State transition diagram is generated from the state transition table.

Example: Suppose we have a finite automata as described in following fig:

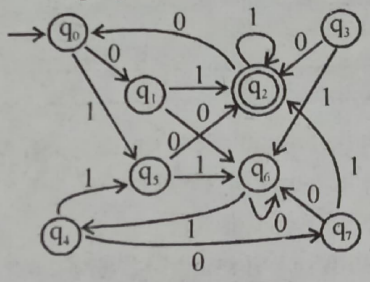


Fig. : Finite automation

Now, we can construct the transition table of above automaton as follows:

Table : Transition table of above automaton

	State (Σ)	0	1
(Start in state) →	q_0	q_1	q_5
	q_1	q_6	q_2
(Final state)	q_2	q_0	q_2
	q_3	q_2	q_6
	q_4	q_7	q_5
	q_5	q_2	q_6
	q_6	q_6	q_4
	q_7	q_6	q_2

Now,

Step (1) : Divide all states in two sets, one of final state(s) and another non final states as follows:

$$\pi_0 = \left\{ \underbrace{\{q_2\}}_{\text{final state(s)}}, \underbrace{\{q_0, q_1, q_3, q_4, q_5, q_6, q_7\}}_{\text{Non final states}} \right\}$$

Step (2) : Now reconstruct the group by considering q_1 and any one of the state q_3, q_5, q_7. With the help of transition table we find that q_1 is not 1-equivalent to q_3 and q_5 but 1-equivalent to q_7. Hence we form a new group as $\{q_1, q_7\}$.

Similarly, we form other groups

$\{q_3, q_5\}$

$\{q_0, q_4, q_6\}$

So, now we have

$\pi_1 = \{\{q_2\}, \{q_0, q_4, q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

Note: To form groups, we should follow these steps:

Suppose we are matching q_1 and q_4

Now at 0 q_1 goes to q_6 and at 0 q_4 goes to q_7 } Same group

but at 1 q_1 goes to q_2 and at 1 q_4 goes to q_5 } Different group

So, q_1, q_4 cannot form a group

Now, Suppose we are matching q_1 and q_7

at 0 q_1 goes to q_6 and at 0 q_7 goes to q_6 } Same group

and at 1 q_7 goes to q_2 and at 1 q_1 goes to q_2 } Same group

Step (3) : Now following step (2) to form further group

$\pi_2 = \{\{q_2\}, \{q_0, q_4\}, \{q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

Step (4) : Similarly,

$\pi_3 = \{\{q_2\}, \{q_0, q_4\}, \{q_6\}, \{q_1, q_7\}, \{q_3, q_5\}\}$

Step (5) : As $\pi_2 = \pi_3$

So we have no need to form further groups.

Step (6) : Reconstruct the transmission table according to π_2 or π_3.

Table : Transition table after minimization

State (Σ)	0	1
$\{q_0, q_4\}$	$\{q_1, q_7\}$	$\{q_3, q_5\}$
$\{q_1, q_7\}$	$\{q_6\}$	$\{q_2\}$
$\{q_2\}$	$\{q_0, q_4\}$	$\{q_2\}$
$\{q_3, q_5\}$	$\{q_2\}$	$\{q_6\}$
$\{q_6\}$	$\{q_6\}$	$\{q_0, q_4\}$

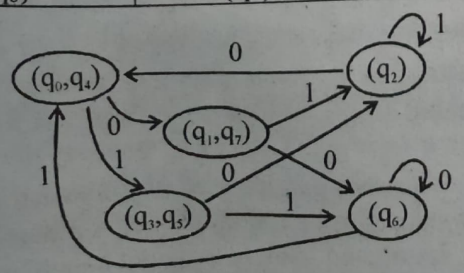


Fig. : Minimized automaton

Q.27 ***What do you understand by finite automata and regular expression.*** *[R.T.U. 2015]*

Ans. Finite Automata : *Refer to Q.3.*

Regular Expression : An expression R is a **regular expression** if R is

1. a for some a in some alphabet Σ,
2. $\in$,
3. ϕ,
4. $(R_1 \cup R_2)$ for some regular expressions R_1 and R_2,
5. $(R_1 . R_2)$ for some regular expressions R_1 and R_2, or
6. $(R_1)^*$ for some regular expression R_1.

When the meaning is clear from the context, () and . can be removed from the expression.

The Language Represented by a Regular Expression

For a regular expression R, L(R) denotes the language R expresses.

1. For each $a \in \Sigma$, $L(a) = \{a\}$.
2. $L(\in) = \{\in\}$
3. $L(\phi) = \phi$
4. $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$
5. $L(R_1 . R_2) = L(R_1) . L(R_2) = \{uv \mid u \in R_1 \text{ and } v \in R_2\}$
6. $L(R_1^*) = \{\in\} \cup \{u_1 \ldots u_k \mid u_1, \ldots, u_k \in R_1\}$

Regular Expression Examples

- $L(a) = \{a\}$ (for a single-element regular expression, you may simply write the element)
- $L(abab \cup bc) = \{abab, bc\}$
- $L((abab \cup abc)^*) = \{\in\} \cup \{w_1 \ldots w_k \mid k \geq 1 \text{ and } w_1, \ldots, w_k \in \{abab, abc\}\}$.
- $L((abab \cup abc)^* \cup c^* \cup abc(abca)^*)$
  $= \{\in\} \cup \{w_1 \ldots . w_k \mid k \geq 1 \text{ and } w_1 \ldots . , w_k \in \{abab, abc\}\}$
  $\cup \{w \mid w \text{ is a repetition of c' s }\}$
  $\cup \{abc, abcabca, abcabcaabca, \ldots\}$

Finite Automata and Regular Expressions :

(1) Transition System and Regular Expression : The following theorem describes the relation between transition system and regular expression.

Theorem : Every regular expression R can be recognized by a transition system i.e. for every string w in the set R, there exists a path from the initial state to a final state with path value w.

Proof : The proof is by the principle of induction on the total number of characters in R. By "Character" we mean elements of Σ, ∧, ϕ, * and +, for example if R = ∧ + 10 * 11 * 0, the characters are ∧ + 1, 0, *, 1, 1, *, 0 and the number of character is 9.

Basis : Let the number of characters in R be 1. Then $R = \wedge$, or $R = \wedge$, or $R = \phi$, or $R = a_i$, $a_i \in \Sigma$. The transition system given will recognize these regular expressions.

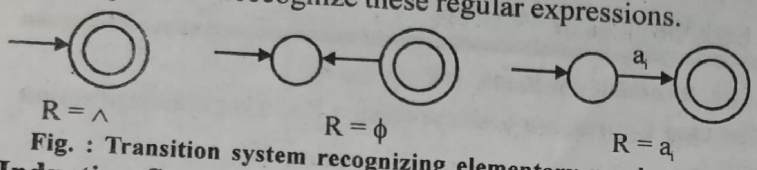


Fig. : Transition system recognizing elementary regular sets

Induction Step : Assume the theorem is true for regular expressions with n character or less. We must prove that it is also true for n + 1 characters. Let R have n + 1 characters. Then

$$R = P + Q \text{ or } R = PQ \text{ or } R = P^*$$

where P and Q are regular expressions each having n characters or less. By induction hypothesis, P and Q can be recognized by transition system G and H, respectively.

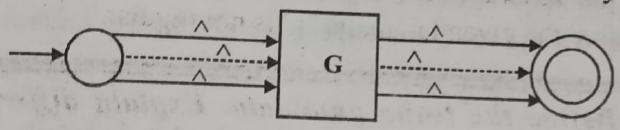


Fig. : Transition system recognizing P

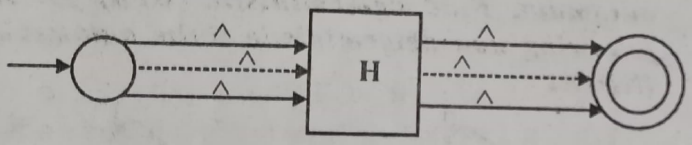


Fig. : Transition system recognizing Q

(2) Transition System Containing ∧-Moves : The transition systems can be generalized by permitting ∧-transitions or ∧-moves which are associated with a null symbol ∧. These transition can occur when no input is applied. But it is possible to convert a transition system with ∧-moves into an equivalent transition system with ∧-moves into an equivalent transition system without ∧-moves we shall give a simple method of doing it with the help of an example.

Suppose we want to replace a ∧-move from vertex v_1 to vertex v_2. Then we proceed as follows :

Step - 1 : Find all the edges starting from v_2.

Step - 2 : Duplicate all these edges starting from v_1 without changing the edge labels.

Step - 3 : If v_1 is an initial state, make v_2 also an initial state.

Example : Consider a finite automaton with ∧-moves given in Fig. below. Obtain an equivalent automaton without ∧-moves.

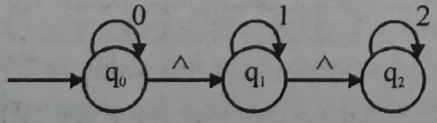
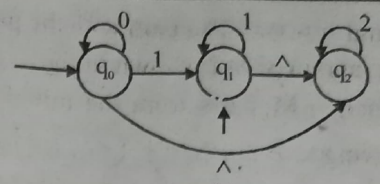
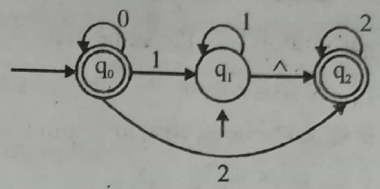


Fig. : FA of Example

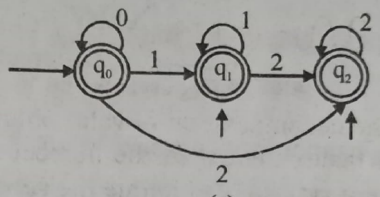
We first eliminate the ∧-moves from q_0 to q_1 to get Fig. (a). q_1 is made an initial state. Then we eliminate the ∧-moves from q_0 to q_2 in Fig. (a) to get Fig. (b). As q_2 is a final state, q_0 is also made a final state. Finally, the ∧-moves from q_1 to q_2 is eliminated in Fig. (c).



(a)



(b)



(c)

Fig. : Transition system without ∧-moves

Q.28 ***State and explain pumping lemma. Prove that the following language L = {aⁿ : n is a perfect square} is not regular.*** *[R.T.U. 2012]*

Ans. Pumping Lemma for Regular Sets : If we give a necessary condition for an input string to belong to a regular set, the result is called *pumping lemma* as it gives a method of pumping (generating) many input strings from a given string. As pumping lemma gives a necessary condition, it can be used to show that certain sets are not regular.

Theorem : (Pumping Lemma) : Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton with n states. Let L be the regular set accepted by M. Let $w \in L$ and $|w| \geq m$. If $m \geq n$, then there exists x, y, z such that $w = xyz, y \neq \Lambda$ and $xyz \in L$ for each $i \geq 0$.

Proof : Let

$$w = a_1 a_2 \ldots\ldots a_m, \quad m \geq n$$

$$\{\delta(q_0, a_1 a_2 \ldots a_i)\} = q_i$$

for $i = 1, 2, \ldots\ldots m$; $Q_1 = \{q_0, q_1, \ldots\ldots, q_m\}$

That is, Q_1 is the sequence of states in the path with path value $w = a_1 a_2 \ldots a_m$. As there are only n distinct states, at least two states in Q_1 must coincide. Among the various pairs of repeated states, we take the first pair. Let us take them as q_j and $q_k (q_j = q_k)$. Then j and k satisfy the condition $0 \leq j < k \leq n$. The string w can be decomposed into three substrings $a_1 a_2 \ldots a_j, a_{j+1} \ldots a_k$ and $a_{k+1} \ldots a_m$. Let x, y, z denote these strings $a_1 a_2 \ldots a_j, a_{j+1} \ldots a_k, a_{k+1} \ldots a_m$ respectively. As

$k \le n, |xy| \le n$ and $w = xyz$. The path with the path value w in the transition diagram of M is shown in fig.

The automaton M starts from the initial state q_0. On applying the string x, it reaches $q_j (= q_k)$. On applying the string y, it comes back to $q_j (= q_k)$. So after application y^i for each $i \ge 0$, the automaton is in the same state q_j. On applying z, it reaches q_m, a final state. Hence $xy^i z \in L$. As every state in Q_1 is obtained by applying an input symbol, $y \ne \Lambda$.

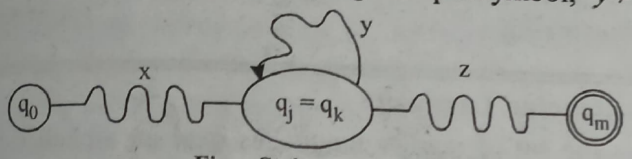


Fig. : String accepted by M

Note : The decomposition is valid only for strings of length greater than or equal to the number of states. For such a string $w = xyz$, we can iterate the substring y in xyz as many times as we like and get strings of the form $xy^i z$ which are longer than xyz and are in L. By considering the path from q_0 to q_k and then the path from q_k to q_m (without going through the loop), we get a path ending in a final state with path value xz. (This corresponds to the case when i = 0)

Proof :

L = {a^n : n is a perfect square}

Let $n = i^2$

(a) Given that $i \ge 1$

For $i = 1, a^{i^2} = a^{1^2} = 0$, length $= 1^2$

$i = 2, a^{i^2} = a^{2^2} =$ aaaa, length $= 2^2$

$i = 3, a^{i^3} = a^3 =$ length $= 3^2$

As we can see, the length of each string is a perfect square.

(b) Assume that L is a regular language

(c) Let 'm' be the integer constant of pumping lemma

(d) Let $z = a^{m^2}$, where length of z is

$|z| = n^2$

(e) By pumping lemma, 'z' may be written as,

$z = uvw$.

Where $1 \le (v) \le m$

and $uv^i w$ is in L for $i \ge 0$

(f) As per assumption that L is regular and $z = uvw$ is in L, it is assummed for i = 1 that $uv^i w \subset L$

Let i = 2

$1 \le |(v)| \le n$

$m^2 + 1 \le |uvw| \le m + m^2$

as, "uv^2w" is concatenation of "uvw" and 'v' where |uvw| = |z| = m^2 by our assumption. Therefore, we have "m^2" on both the sides of $1 \le |v| \le m$

(g) $n^2 + 1 \le |uv^2w| \le m^2 + m$

but $m^2 + m < (m + 1)^2$

Therefore the equation becomes

$m^2 + 1 \le |uv^2w| < (m+1)^2$

i.e. $m^2 < |uv^2w| < (m + 1)^2$

i.e. for i = 2, the length of the string "uv^iw" resides between m^2 and $(m+1)^2$, two consecutives square values. Thus, the $|uv^2 w|$ is not a perfect square. Therefore "uv^2 w" is not in L. But this is a contradiction to the lemma statement. Then our assumption that "L" is regular must be wrong. Therefore the given language 'L' is not regular.

Q.29 ***Define the finite automata. Explain difference between deterministic and non-deterministic finite automata. Find deterministic (DFA) for the following non-deterministic finite automation (NDFA)***

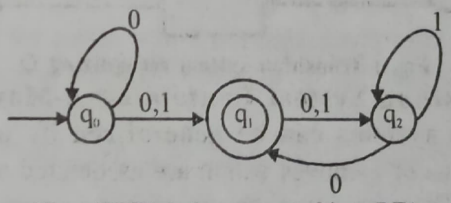


Fig. : NDFA to be converted into DFA

[R.T.U. 2011]

Ans. Finite Automata : *Refer to Q.3.*

Difference between Deterministic and Non-deterministic Finite Automata : *Refer to Q.15.*

Since a state transition diagram is given it must be converted into a state transition table.

States	Input	
	0	1
→ q_0	q_0, q_1	q_1
(q_1)	q_2, q_1	q_2, q_1
q_2	q_1	q_2

Step 1 : Input symbol for the resultant DFA

= Input symbols for the given NDFA

= 0 and 1

Step 2 : Start state for the resultant DFA

= Set of start states of given NDFA

= q_0

Step 3 : Since there are three states in the given NDFA, so the number of states in the resultant DFA = 2^3

All the states of resultant DFA = Set of subsets of the set of states of the given NDFA.

$= \phi, q_0, q_1, q_2, [q_0, q_1], [q_0, q_2], [q_1, q_2], [q_0, q_1, q_2]$

Step 4 : Now we will find out the accessible states from the states derived in step 3 *i.e.* which are accessible from the start state of the DFA which is q_0 in this case.

$$\delta(q_0, 0) = [q_0, q_1]$$
$$\delta(q_0, 1) = [q_1]$$

Taking each of the two input symbols once with the start state, we get two new states *i.e.* $[q_0, q_1]$ and q_1. Now we apply the same procedure for the two new states as under

$$\begin{aligned}\delta([q_0, q_1], 0) &= \delta(q_0, 0) \cup \delta(q_1, 0)\\ &= \delta[q_0, q_1] \cup [q_2, q_1]\\ &= [q_0, q_1, q_2]\end{aligned}$$

$$\begin{aligned}\delta([q_0, q_1], 1) &= \delta(q_0, 1) \cup \delta(q_1, 1)\\ &= [q_1] \cup [q_2, q_1]\\ &= [q_2, q_1]\end{aligned}$$

$$\delta(q_1, 0) = [q_1, q_2]$$
$$\delta(q_1, 1) = [q_1, q_2]$$

Thus, after processing the 2 new states we get 2 new states $[q_1, q_2]$ and $[q_0, q_1, q_2]$. Now we again process these 2 new states.

$$\begin{aligned}\delta([q_1, q_2], 0) &= \delta(q_1, 0) \cup \delta(q_2, 0)\\ &= [q_1, q_2] \cup [q_1]\\ &= [q_2, q_1]\end{aligned}$$

$$\begin{aligned}\delta([q_1, q_2], 1) &= \delta(q_1, 1) \cup \delta(q_2, 1)\\ &= [q_1, q_2] \cup [q_2]\\ &= [q_1, q_2]\end{aligned}$$

$$\begin{aligned}\delta([q_0, q_1, q_2], 0) &= \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0)\\ &= [q_0, q_1] \cup [q_1, q_2] \cup [q_1]\\ &= [q_0, q_1, q_2]\end{aligned}$$

$$\begin{aligned}\delta([q_0, q_1, q_2], 1) &= \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1)\\ &= [q_1] \cup [q_1, q_2] \cup [q_2]\\ &= [q_1, q_2]\end{aligned}$$

Thus on processing we do not get any new state and so we simply stop processing now. The accessible states are :

$[q_0], [q_1], [q_0, q_1], [q_1, q_2], [q_0, q_1, q_2]$

Step 5 : Final state for the resultant DFA = Those accessible state that consists of at least one of the final state of the given NDFA. Final state of NDFA = $[q_2]$. From accessible state q_2 containing states are (q_1, q_2) and (q_0, q_1, q_2).

So, final state of the resultant DFA

$= (q_1, q_2)$ and (q_0, q_1, q_2)

Q.30 ***Define regular expressions and languages associated with regular expressions. Write the regular expression and finite automata (transition diagram) for following language over alphabets*** $\sum = \{a, b\}$

(i) The set of strings that start with "ab" and end with "bb".

(ii) The set of strings that starts with 'a' and ends with 'b' and contain at least one sequence of "a a a" in that string. ***[RTU 2011]***

Ans. Regular Language

The set of regular languages over an alphabet Σ is defined recursively as below. Any language belonging to this set is a regular language over Σ.

Definition of set of Regular Languages

Basic Clause : $\{\Lambda\}$ and $\{a\}$ for any symbol $a \in \Sigma$ are regular languages.

Inductive Clause: If L_r and L_s are regular languages, then $L_r \cup L_s$, $L_r L_s$ and L_r^* are regular languages.

Extremal Clause: Nothing is a regular language unless it is obtained from the above two clauses. For example, let $\Sigma = \{a, b\}$.

Then since $\{a\}$ and $\{b\}$ are regular languages, $\{a,b\}$ $(a = \{a\} \cup \{b\})$ and $\{ab\} (= \{a\}\{b\})$ are regular languages. Also since $\{a\}$ is regular, $\{a\}^*$ is a regular language which is the set of strings consisting of a's such a Λ, a, aa, aaa, aaaa etc. Note also that Σ^*, which is the set of strings consisting of a's and b's, is a regular language because $\{a,b\}$ is regular.

Regular Expression

Regular expressions are used to denote regular languages. They can represent regular languages and operations on them succinctly.

The set of regular expressions over an alphabet Σ is defined recursively as below. Any element of that set is a regular expression.

Basic Clause : ϕ, Λ and a are regular expressions corresponding to languages ϕ, $\{\Lambda\}$ and $\{a\}$, respectively, where a is an element of Σ.

Inductive Clauses : If r and s are regular expressions corresponding to language L_r and L_s, then $(r + s)$, (rs) and (r^*) are regular expressions corresponding to languages $L_r \cup L_s$, $L_r L_s$ and $L_r{}^*$, respectively.

Extremal Clause : Nothing is a regular expression unless it is obtained from the above two clauses.

Conventions on regular expressions

(1) When there is no danger of confusion, bold face may not be used for regular expression. So for example, (r + s) is used instead of (r + s).

(2) The operation * has precedence over concatenation, which has precedence over union (+). Thus the regular expression (a+(b(c*))) is written as a + bc*,

(3) The concatenation of k r's, where r is a regular expression, is written as r^k. Thus for example $rr = r^2$. The language corresponding to r^k is L_r^k, where L_r is the language corresponding to the regular expression r. For a recursive definition of L_r^k

(4) We use (r^+) as a regular expression to represent L_r^+.

Examples of regular expression and regular languages corresponding to them

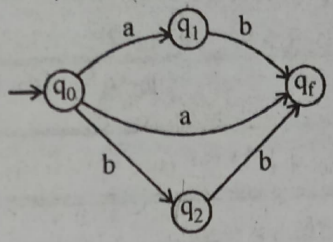
- $(a+b)^2$ corresponds to the language {aa, ab, ba, bb}. That is the set of strings of length 2 over the alphabet {a,b}. In general $(a+b)^k$ corresponds to the set of strings of length k over the alphabet {a,b}. (a+b)* corresponds to the set of all strings over the alphabet {a,b}.
- a^*b^* corresponds to the set of strings consisting of zero or more a's followed by zero or more b's.
- $a^*b^+a^*$ corresponds to the set of strings consisting of zero or more a's followed by one or more b's followed by zero or more a's.
- $(ab)^+$ corresponds to the language {ab, abab, ababab, ...}, that is, the set of strings of repeated ab's.

Note: A regular expression is not unique for a language. That is, a regular language, in general, corresponds to more than one regular expressions. For example (a + b)* and (a*b*)* correspond to the set of all strings over the alphabet {a,b}.

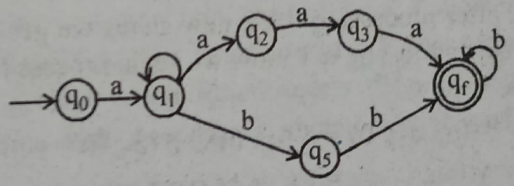
Definition of Equality of Regular Expressions

Regular expressions are **equal** if and only if they correspond to the same language. Thus for example $(a + b)^* = (a^*b^*)^*$, because they both represent the language of all strings over the alphabet {a,b}. **In general, it is not easy to see by inspection whether or not two regular expressions are equal.**

(i) Regular expression is ab + (a + bb), corresponding transition diagram for this expression is



(ii) Regular expression is a (a+b)* (aaa+bb) (b)*, transition diagram is a, b



❑❑❑

CONTEXT FREE GRAMMARS 2

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***What is Binary tree?*** *[R.T.U. 2019]*

Ans. Binary Tree : A binary tree is a tree data structure where each node has up to two child nodes, creating the branches of the tree. The two children are usually called the left and right nodes. Parent nodes are nodes with children, while child nodes may include references to their parents.

A binary tree is made up of at most two nodes, often called the left and right nodes and a data element. The topmost node of the tree is called the root node and the left and right pointers direct to smaller subtrees on either side.

Binary trees are used to implement binary search trees and binary heaps. They are also often used for sorting data as in a heap sort.

Q.2 ***What is sets and subsets?*** *[R.T.U. 2019]*

Ans. Sets and Subsets

- A **set** is a well-defined collection of objects.
- Each object in a set is called an **element** of the set.
- Two sets are **equal** if they have exactly the same elements in them.
- A set that contains no elements is called a **null set** or an **empty** set.
- If every element in Set *A* is also in Set *B*, then Set *A* is a **subset** of Set *B*.

Q.3 ***What is context free grammar?*** *[R.T.U. 2019]*
OR
Explain context free grammar in brief.

Ans. Context free grammar is also called type 2 Grammar.

$G=\{V_N,\Sigma,P,S\}$ is called context free grammar it every production has form $A\to\alpha$ where $A\in V_N$ and $\alpha\in(V_N\cup\Sigma)^*$

Let $G=\{\{S,A,B\},\{a,b\},$

$\{S\to a,S\to aS,S\to aAS,S\to aB,A\to ab\},S\}$

then G is called context free grammar.

Q.4 ***Define degree of ambiguity of W.***

Ans. Ambiguity in CFG

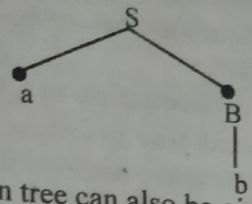
A CFG, G = (V, T, P, S) is ambiguous if there is at least one string W in L(G) for which there are at least two different parse trees, each with its root labeled S and yielding W. Each parse tree corresponds to a left-most or a right-most derivation. The number of different parse trees of a string W is called the degree of ambiguity of W .

Q.5 ***Show the grammar***
$S\to aB/ab$
$A\to aAB/a$
$B\to ABb/b$
is ambiguous.

Ans. The left most derivation tree for grammar is

$S\Rightarrow aB$ (using $S\to aB$)
$S\Rightarrow ab$ (using $B\to b$)

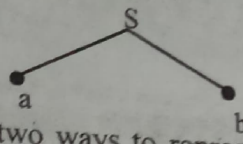
The parse tree can be given as



Left most derivation tree can also be given as :

$S \Rightarrow ab$ (using $S \rightarrow ab$)

The corresponding parse tree is-



Hence we have two ways to represent this grammar. Thus the grammar is ambiguous.

Q.6 ***Consider the following production :***

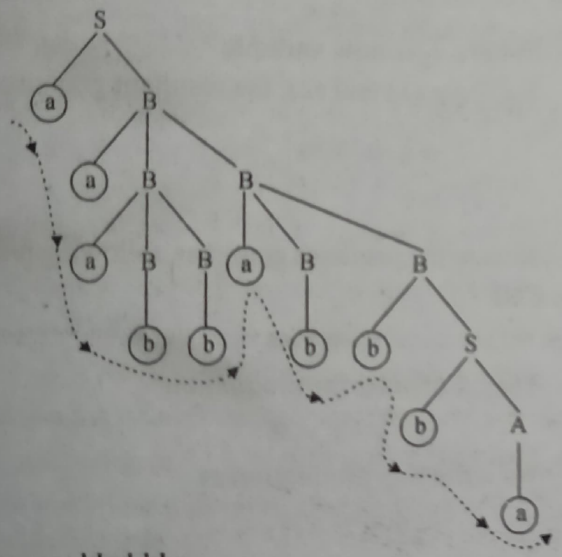
$S \rightarrow aB/bA$

$A \rightarrow as/bAA/a$

$B \rightarrow bs/aBB/b$

find the parse tree.

Ans. Parse Tree



String : aaabbabbba

Q.7 ***Reduce the grammar to CNF***

$S \rightarrow aB/ab$

$A \rightarrow aAB/a$

$B \rightarrow ABb/b$

Ans. Production $A \rightarrow a$ and $B \rightarrow b$ are already in CNF. If introduce two more productions, say

$R_1 \rightarrow a$

$R_2 \rightarrow b$

above grammer can be written as;

$S \rightarrow R_1B \mid R_1 R_2$

$A \rightarrow R_1A B \mid a$

$B \rightarrow A B R_2 \mid b$

Still there are two production which are not in CNF, namely $A \rightarrow R_1AB$ and $B \rightarrow ABR_2$. After further breakup and introducing new symbol R_3 we can write the given grammer in CNF as,

$S \rightarrow R_1 B \mid R_1 R_2$

$A \rightarrow R_1 R_3 \mid a$

$B \rightarrow R_3 R_2 \mid b$

$R_1 \rightarrow a$

$R_2 \rightarrow b$

$R_3 \rightarrow AB$

PART-B

Q.8 ***Consider the context free grammar -***

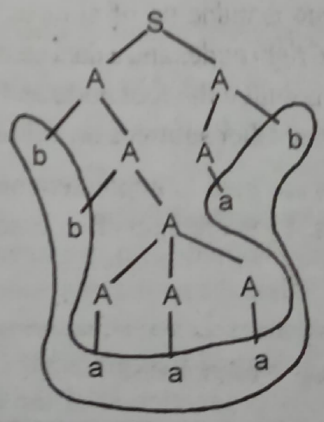
$S \rightarrow AA$

$A \rightarrow AAA \backslash bA \backslash Ab \backslash a$

Find the parse tree for the string bbaaaab

[R.T.U. 2019]

Ans. $S \rightarrow \underline{A}A$



$\Rightarrow b\underline{A}A$

$\Rightarrow bb\underline{A}A$

$\Rightarrow bb\underline{A}AAA$

$\Rightarrow bba\underline{A}AA$

$\Rightarrow bbaa\underline{A}A$

$\Rightarrow bbaaa\underline{A}$

$\Rightarrow bbaaa\underline{A}b$

$\Rightarrow bbaaaab$

Q.9 ***Reduce the following grammars in Chomsky normal form :***

(i) $S \to |A \mid 0B, A \to |AA \mid 0S \mid 0, B \to 0BB \mid IS \|$

(ii) $G = (\{S\}, \{a, b, c\}, \{S \to a \mid b \mid cSS\}, S)$

(iii) $S \to a\,b\,S\,b \mid a \mid a\,A\,b, A \to b\,S \mid a\,A\,A\,b$

[R.T.U. 2016]

Ans. (i) As there are no null productions or unit productions, we can proceed to step 1.

Step 1 : Let $G_1 = (V_N', \{0,1\}, P_1, S)$, where P_1 and V_N' are constructed as follows :

(i) $A \to 0$, $B \to 1$ are included in P_1.

(ii) $S \to 1A$, $B = 1S$ give rise to $S \to C_1A$, $B \to C_1S$ and $C_1 \to 1$

(iii) $S \to 0B$, $A \to 0S$ give rise to $S \to C_0B$, $A \to C_0S$ and $C_0 \to 0$

(iv) $A \to 1AA$, $B \to OBB$ give rise to $A \to C_1AA$ and $B \to C_0BB$.

$$V'_N = \{S, A, B, C_0, C\}$$

Step 2 : $G_2 = (V_N'', \{0,1\}, P_2, S)$, where P_2 and V_N'' are constructed as follows :

(i) $A \to 0$, $B \to 1$, $S \to C_1A$, $B \to C_1S$

$C_1 \to 1$, $S \to C_0B$, $A \to C_0S$, $C_0 \to 0$ are included in P_2

(ii) $A \to C_1AA$ and $B \to C_0BB$ are replaced by $A \to C_1D_1$, $D_1 \to AA$, $B \to C_0D_2$, $D_2 \to BB$.

Thus $G_2 = \{S, A, B, C_0, C_1, D_1, D_2\}, \{0,1\}, \{P_2, S\}$ is in CNF and equivalent to the given grammar where P_2 consists of

$$S \to C_1A \mid C_0B, A \to 0 \mid C_0S \mid C_1D_1,$$

$$B \to 1 \mid C_1S \mid C_0D_2, C_1 \to 1, C_0 \to 0, D_1 \to AA$$

and $D_2 \to BB$.

(ii) Chomsky Normal Form

In the chomsky normal form, we have restrictions on the length of R.H.S. and the nature of symbols in the R.H.S. of productions.

Definition : A context-free grammar G is in chomsky normal form if every production is of the form $A \to a$, or $A \to BC$, and $S \to \wedge$ is in G if $\wedge \in L(G)$. When $\wedge$ is in L (G), we assume that S does not appear on the R.H.S. of any production.

For example, consider G whose productions are $S \to AB \mid \wedge$, $A \to a$, $B \to b$. Then G is in Chomsky normal form.

Remark : For a grammar in CNF, the derivation tree has the following property :

Every node has atmost two descendants - either two internal vertices or a single leaf.

When a grammar is in CNF, some of the proofs and constructions are simpler.

The techniques applied in this example are used in the following theorem.

Theorem : (Reduction to Chomsky Normal Form) For every context-free grammar, there is an equivalent grammar G_2 in chomsky normal form.

$$G = (\{s\}, \{a, b, c\}, \{s \to a/b/css\}, s)$$

Given production of G is

$$s \to a/b/css \quad \ldots (A)$$

In a given grammar (A) following productions are in CNF

$$s \to a$$

$$s \to b$$

Also, in the given grammar (A), following productions is not in CNF

$$s \to css$$

So, Consider the production

$$s \to css$$

We write this production as

$$s \to v_1ss \quad \ldots (1)$$

$$v_1 \to c \quad \ldots (2)$$

Where v_1 is now variable.

So, from (1) and (2), the resultant grammar become

$$\left.\begin{aligned} s &\to a/b/v_1ss \\ v_1 &\to c \end{aligned}\right\} \quad \ldots (B)$$

Now in the resultant grammar, following production is not in CNF

$$s \to v_1ss$$

Thus, Consider the production

$$s \to v_1ss$$

We write this production as

$$s \to v_1v_2 \quad \ldots (3)$$

$$v_2 \to ss \quad \ldots (4)$$

v_2 also a new variable.

So, from (B), (4), the resultant grammar become

$$\left.\begin{aligned} s &\to a/b/v_1v_2 \\ v_1 &\to c \\ v_2 &\to ss \end{aligned}\right\} \quad \ldots (c)$$

Thus, the resultant grammar (c) is in CNF.

Ans. (iii) Step (1) : Since S appears in R.H.S., we add a new state. S_0 and $S_0 \to S$ is added to the production set and it becomes :

$$S_0 \to S$$

$$S \to abSb \mid a \mid aAb$$

$A \rightarrow bS \mid aAAb$

Step 2 : There is no null production to remove in the given production. So move towards next step.

Step 3 : Now we will remove the unit productions. But, here in this question there is only one unit production which is

$S_0 \rightarrow S$

After removing $S_0 \rightarrow S$, production becomes :

$S_0 \rightarrow abSb \mid a \mid aAb$

$S \rightarrow abSb \mid a \mid aAb$

$A \rightarrow bS \mid aAAb$

Step 4 : Now make right hand side not contain more than 2 non-terminals or 1 terminal

$S_0 \rightarrow XYSY \mid a \mid XAY$ $\quad X \rightarrow a$

$S \rightarrow XYSY \mid a \mid XAY$ $\quad Y \rightarrow b$

$A \rightarrow YS \mid XAAY$

Step 5 : Now transform the production into Chomsky normal form.

$S_0 \rightarrow XP \mid a \mid XQ$

$S \rightarrow XP \mid a \mid XQ$

$A \rightarrow YS \mid XR$

$X \rightarrow a$

$Y \rightarrow b$

$P \rightarrow YSY$

$Q \rightarrow AY$

$R \rightarrow AAY$

Now production of S_0, S, A, X, Y and Q are in Chomsky Normal form but P and R is not so make them.

$P \rightarrow YM$

$R \rightarrow AN$

$M \rightarrow SY$

$N \rightarrow AY$

Now the final production in Chomsky normal form is

$S_0 \rightarrow XP \mid a \mid XQ$

$S \rightarrow XP \mid a \mid XQ$

$A \rightarrow YS \mid XR$

$X \rightarrow a$

$Y \rightarrow b$

$P \rightarrow YM$

$R \rightarrow AN$

$Q \rightarrow AY$

$M \rightarrow SY$

$N \rightarrow AY$

Q.10 ***Consider the following productions :***

$S \rightarrow aB \mid bA$

$A \rightarrow aS \mid bAA \mid a$

$B \rightarrow bS \mid aBB \mid b$

for the string a a a b b a b b b a, find

(i) the leftmost derivation,

(ii) the rightmost derivation, and

(iii) the parse tree ***[R.T.U. 2016, 2009]***

Ans. String : aaabbabbba

$S \rightarrow aB \mid bA$

$A \rightarrow aS \mid bAA \mid a$

$B \rightarrow bS \mid aBB \mid b$

(i) Left Most Derivation

$S \rightarrow aB$

$S \rightarrow a\underline{aBB}$ $\quad [B \rightarrow aBB]$

$S \rightarrow aa\underline{aBB}B$

$S \rightarrow aaa\underline{b}BB$ $\quad [B \rightarrow b]$

$S \rightarrow aaab\underline{b}B$

$S \rightarrow a\,aabbaBB$ $\quad [B \rightarrow aBB]$

$S \rightarrow aaabbabB$ $\quad [B \rightarrow b]$

$S \rightarrow aaabbabbS$ $\quad [B \rightarrow bS]$

$S \rightarrow aaabbabbbA$ $\quad [S \rightarrow bA]$

$S \rightarrow aaabbabbba$ $\quad [A \rightarrow a]$

(ii) Right Most Derivation

$S \rightarrow aB$

$S \rightarrow aaBB$ $\quad [B \rightarrow aBB]$

$S \rightarrow aaBaBB$ $\quad [B \rightarrow aBB]$

$S \rightarrow aaBaBbS$ $\quad [B \rightarrow bS]$

$S \rightarrow aaBaBbbA$ $\quad [S \rightarrow bA]$

$S \rightarrow aaBaBbba$ $\quad [A \rightarrow a]$

↓

S → aaBabbba [B → b]

↓

S → aaaBBabbba [B → aBB]

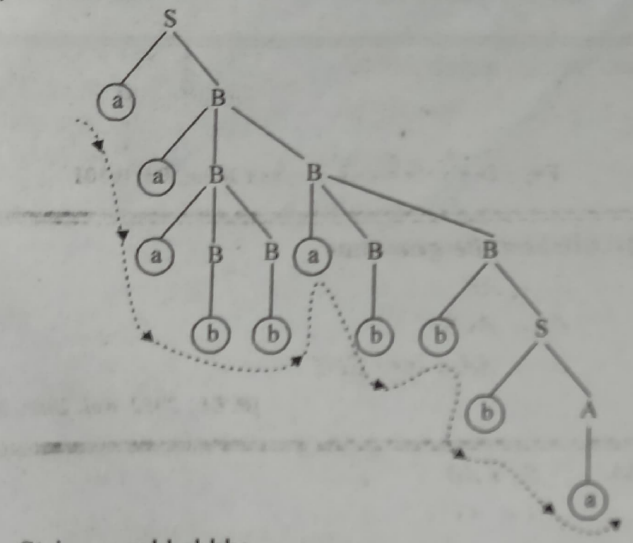
↓

S → aaaBbabbba [B → b]

S → aaabbabbba [B → b]

Hence, string is accepted

(iii) Parse Tree



String : aaabbabbba

Q.11 ***Let G be the grammar***

bAaBS\|→

bAAaSaA\|\|→

aBBbSbB\|\|→

For the string "baaabbabba" find left most derivation, right most derivation and parse tree.

[R.T.U. 2015]

Ans. Given grammar G is :

bA aB S | →

bAA aS a A | | →

aBB bS b B | | →

In this question production system is to be like this :

S → aB|bA

A → a|aS|bAA

B → b|bS|aBB

Let us take the string baaabbabba

Left Most Derivation :

S → bA

→baS [A → aS]

→baaB [S → aB]

→baaaBB [B → aBB]

→baaabB [B → b]

→baaabbS [B → bS]

→baaabbaB [S → aB]

→baaabbabS [B → bS]

→baaabbabbA [S → bA]

→baaabbabba [A → a]

Yield =baaabbabba

Right Most Derivation :

S → bA

→baS [A → aS]

→baaB [S → aB]

→baaaBB [B → aBB]

→baaaBbs [B → bS]

→baaaBbaB [S → aB]

→baaaBbabs [B → bS]

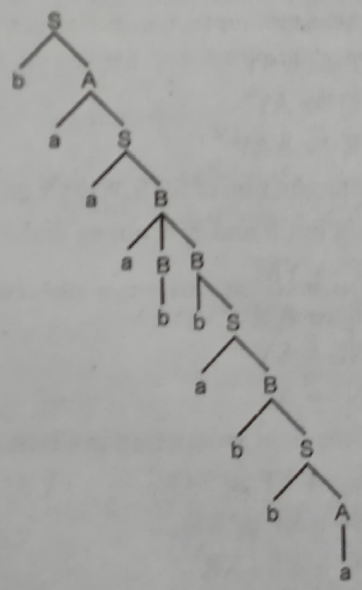
→baaaBbabbA [S → bA]

→baaaBbabba [A → a]

→baaabbabba [B → b]

Yield =baaabbabba

Parse Tree



Q.12 ***Explain Greibach normal form in detail.***

[R.T.U. 2014]

Ans. In computer science and formal language theory, a context-free grammar is in Greibach normal form (GNF) if the right-hand sides of all production rules start with a terminal

symbol, optionally followed by some variables. A non-strict form allows one exception to this format restriction for allowing the empty word (epsilon, ε) to be a member of the described language. The normal form bears the name of Sheila Greibach.

More precisely, a context-free grammar is in Greibach normal form, if all production rules are of the form:

$$A \to aA_1A_2 \cdots A_n$$

or $S \to \varepsilon$

where A is a nonterminal symbol, a is a terminal symbol, $A_1A_2 \ldots A_n$ is a (possibly empty) sequence of nonterminal symbols not including the start symbol, S is the start symbol, and ε is the empty word.

Observe that the grammar does not have left recursions.

Every context-free grammar can be transformed into an equivalent grammar in Greibach normal form.

Some definitions do not consider the second form of rule to be permitted, in which case a context free grammar that can generate the empty word cannot be so transformed. In particular, there is a construction ensuring that the resulting normal form grammar is size at most $O(n^4)$, where n is the size of the original grammar. This conversion can be used to prove that every context free language can be accepted by a non-deterministic pushdown automaton.

Given a grammar in GNF and a derivable string in the grammar with length n, any top-down parser will halt at depth n.

Q.13 ***The production of any grammar $\in$ is given by***

$S \to 0B/1A$ $\quad A \to 0/0S/1AA$

$B \to 1/1S/0BB$

For the string 00110101, find leftmost derivation, rightmost derivation and derivation tree.

[R.T.U. 2013]

Ans. **(i) Left most derivation**

$S \to 0B$
$00\overline{BB}$
$001\overline{B}$
$0011\overline{S}$
$00110\overline{B}$
$001101\overline{S}$
$0011010\overline{B}$
00110101

or

$S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 001B \Rightarrow 0011S \Rightarrow 0110B \Rightarrow 001101S \Rightarrow 00110101$

(ii) Right most derivation

$S \to 0B$
$00\overline{BB}$
$00B\overline{1S}$
$00B1\overline{0B}$
$00B10\overline{1S}$
$00B1010\overline{B}$
$00B10101$
00110101

or

$S \Rightarrow 0B \Rightarrow 00BB \Rightarrow 00B1S \Rightarrow 00B10B \Rightarrow 00B101S \Rightarrow 00B1010B \Rightarrow 00B10101 \Rightarrow 00110101$

(iii) Derivation Tree

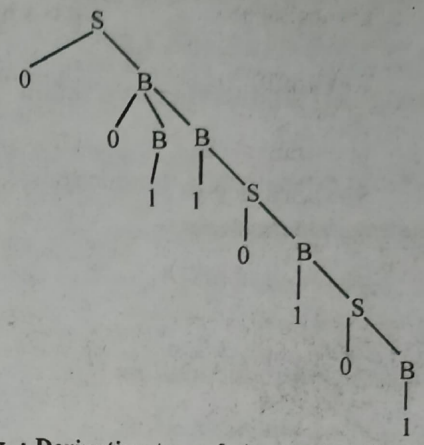


Fig. : Derivation tree of given string 00110101

Q.14 ***Convert the grammar***

$S \to AB$

$A \to Bs/b$

$B \to SA/a$ into GNF

[R.T.U. 2012, Raj. Univ. 2006]

Ans. $S \to AB$

$A \to BS \mid b$

$B \to SA \mid a$

Step 1 : Rename S, A, B as A_1, A_2, A_3 respectively. The productions are

$A_1 \to A_2\ A_3$

$A_2 \to A_3A_1 \mid b$

$A_3 \to A_1A_2 \mid a$

Step 2 : (a) The A_1 production

$A_1 \to A_2A_3$ is in required form.

(b) A_2 production $A_2 \to A_3A_1 \mid b$ are in required form.

(c) $A_3 \to a$ is in required form.

(d) $A_3 \to A_1A_2$ is not in required form so $A_3 \to A_2A_3A_2$ (replacing $A_1 \to A_2A_3$)

$A_3 \to A_3A_1A_3A_2 \mid bA_3A_2$

Step 3 : A_3 productions are :

$A_3 \to a \mid b\,A_3\,A_2$

$A_3 \to A_3A_2A_3A_2$

Introducing new variable z_3

The resulting productions are :

$A_3 \to a \mid bA_3A_2$

$A_3 \to aZ_3 \mid bA_3A_2Z_3$

$Z_3 \to A_1A_3A_2,\ Z_3 \to A_1A_3A_2Z_3$

Step 4 : (a) The A_3 production are :

$$A_3 \to a \mid bA_3A_2 \mid aZ_3 \mid bA_3A_2Z_3 \quad ...(1)$$

(b) Among A_2 productions, we retain $A_2 \to b$ and illuminate

$$A_2 \to A_3A_1 \text{ (replace } A_3 \text{ productions)}$$

The resulting inductions are :

$$A_2 \to aA_1 \mid bA_3A_2A_1 \mid aZ_3A$$

$$A_2 \to bA_3A_2Z_3A_1$$

The modified A_2 productions are

$$A_2 \to b \mid aA_1 \mid bA_3A_2A_1 \mid aZ_3A_1$$

$$A_2 \to bA_3A_2Z_3A_1 \quad ...(2)$$

(c) $A_1 \to A_2A_3$ applying Lemma to get (we replace A_2 production in above production)

$$A_1 \to bA_3 \mid aA_1A_3 \mid bA_3A_2A_1A_3$$

$$A_1 \to aZ_3A_1A_3 \mid bA_3A_2Z_3A_1A_3 \quad ...(3)$$

Step 5 : The Z_3 production to be modified are

$$Z_3 \to A_1A_3A_2 \mid A_1A_3A_2Z_3$$

We apply Lemma, to replace A_1 from Z_3 productions

$$Z_3 \to bA_3A_3A_2 \mid bA_3A_3A_2Z_3$$

$$Z_3 \to aA_1A_3A_3A_2 \mid aA_1A_3A_3A_2Z_3$$

$$Z_3 \to bA_3A_2A_1A_3A_3A_2 \mid bA_3A_2A_1A_3A_3A_2Z_3$$

$$Z_3 \to aZ_3A_1A_3A_3A_2 \mid aZ_3A_1A_3A_3A_2Z_3$$

$$Z_3 \to bA_3A_2Z_3A_1A_3A_3A_2 \mid bA_3A_2Z_3A_1A_2A_3A_3A_2Z_3 \quad ...(4)$$

The required grammar in GNF is given by (1) – (4)

Q.15 ***Define the context – free grammar and find the context free grammar for the following languages :***

(i) $I=\{a^n b^m : n \le m+3\}$

(ii) $L=\{a^n b^m : n \ne 2m\}$ *[R.T.U. 2011]*

Ans. Context – Free Grammar : A context free grammar (CFG) is a 4–tuple G — (V, Σ, S, P), where V and Σ are disjoint finite sets, S is an element of V and P is a finite set of formulas of the form $A \to \alpha$, where $A \in V$ and $a \in (V \cup \Sigma)^*$.

The elements of V are called variables, or non terminal symbols and those of the alphabet Σ are called terminal symbols, or terminals. S is called the start symbol, and the elements of P are called grammar rules, or productions.

(i) We use A to generate $L_1 = \{a^n b^m : n \le m + 3\}$ and B to generate $L_2 = \{b^m \mid 0 \le m\}$. One can verify that $L = L_1L_2$ because attaching L_2 to L_1 means we can add any number of b's to the end of the string in L_1. Thus, m, the number of b's is no longer bounded by n, the number of a's.

$S \to AB$

$A \to \lambda|a|aa|aaa|aAb$

$B \to \lambda|bB$

(ii) We use E to generate $L_1 = \{a^n b^m \mid n = 2m\}$. A to generate non-empty string of a's and B non-empty strings of b's. Thus, AE and EB can generate $\{a^n b^m \mid n > 2m\}$ and $\{a^n b^m \mid n < 2m\}$, respectively

$S \to AE|EB$

$A \to a|aA$

$B \to b|bB$

$E \to \lambda|aaEb$

Q.16 ***If L is context free language and R being regular then prove.***

(i) $L \cap R$ ***is context free.***

(ii) $L - R = L \cap \bar{R}$ ***is also a context free.***

[R.T.U. 2010]

Ans.(i) Let P be the PDA that accepts L by final state and let M be the DFA that accepts R. The PDA recognize $L \cap R$ simulates P and R simultaneously and accepts if both P and M accept the input.

(1) Like in "Cross Product" construction states of the new machine are pairs of states, where one element of the pair corresponds to the state of P and the other corresponds of the state of M.

(2) Whenever, an input symbol is read, both P and M are simulated, if P needs to make an ∈ move then M remains stationary.

i.e. $L \cap R$ is context free.

(ii) Let L be accepted by a PDA $A = \{Q_A, \Sigma, T, \delta_A, q_0, Z_0, F_A)$ by final state and R by DFA $M = \{Q_M, \Sigma, \delta_M, P_0, F_M\}$.

We define a PDA M′ accepting $L \cap R$ by final state in such a way that M′ simulates moves of A on input a in Σ and changes the state of M using δ_M. On input ∧, M′ simulates A without changing the state of M. Let

$M' = (Q_M * Q_A, \Sigma, \Gamma, \delta[p_0, q_0], Z_0, F_M * F_A)$

Where δ is defined as follows :

$\delta([p, q], a, x)$ contains $([p', q'], y)$.

When

$$\delta_M(p, a) = p'$$

and $\delta_A(q, a, x)$ contains (q', y). $\delta([p, q], \wedge, x)$ contains $([p, q'], r)$. When $\delta_A(q, \wedge, x)$ contains (q', r).

To prove that $L - R = L \cap \bar{R}$ is also a context free.

Q.17 ***Find a grammar in Chomsky Normal Form equivalent to*** $S \to aAbB, A \to aA/a, B \to bB/b.$

[R.T.U. 2010, Raj. Univ. 2005, 2004, 2002]

Ans. As there are no unit productions or null productions we need not carry out of elimination of unit productions or null production. We proceed to next steps.

Let $G_1 = (V'_N \{a, b\}, P_1, S)$, where P_1 and V'_N are constructed as follows :

(i) $A \to a$, $B \to b$ are added to P_1

(ii) $S \to aAbB$, $A \to aA$, $B \to bB$ yield $S \to C_aAC_bB$, $A \to C_aA$, $B \to CB$, $C_a \to a$, $C_b \to b$.

$V'_N = \{S, A, B, C_a, C_b\}$.

P_1 consists of

$S \to C_aAC_bB$, $A \to C_aA$, $B \to C_bB$, $C_a \to a$, $C_b \to b$, $A \to a$, $B \to b$.

$S \to C_aAC_bB$ is replaced by $S \to (C_aC_1, C_1 \to AC_2, C_2 \to C_bB$. The remaining productions in P_1 are added to P_2.

Let

$G_2 = (\{S, A, B, C_a, C_b, C_1, C_2\}, \{a, b\}, P_2, S)$,

where P_2 consists of $S \to C_aC_1$, $C_1 \to AC_2$, $C_2 \to C_bB$ $A \to C_aA$, $B \to C_bB$, $C_a \to a$, $Cb \to b$, $A \to a$, and $B \to b$.

G_2 is in CNF and equivalent to the given grammar.

Q.18 ***Find a grammar in GNF equivalent to the grammar*** $E \to E+T \mid T, T \to T*F \mid F, F \to (E) \mid a$

[R.T.U. 2009]

Ans. $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid a$

GNF (Greibach Normal Form)

GNF is defined as the normal form of a grammar whose production are in the format

$A \to a\alpha$ where $a \in \Sigma$

$\alpha \in (V_N \cup \Sigma)$

α may be $\wedge$

In the given question,

Terminal symbols $\Sigma = \{a\}$

Non-terminals $V_N = \{E, T, F, +, *, (,)\}$
$A_1\ A_2\ A_3\ A_4\ A_5\ A_6\ A_7$

We convert all V_N into A_1, A_2, A_3

Now, the production rules are

$A_1 \to A_1A_4A_2 \mid A_2$ ....(1)

$A_2 \to A_2A_5A_3 \mid A_3$ ....(2)

$A_3 \to A_6A_1A_7 \mid a$ ....(3)

Since $A_3 \to a$ (Required form) ....(i)

$A_2 \to A_3 \Rightarrow A_2 \to a$ (Required form)....(ii)

and $A_1 \to A_2 \Rightarrow A_1 \to a$ (Required form)....(iii)

Now, from (1)

$A_1 \to A_1A_4A_2$

From (iii) $A_1 \to a$

$A_1 \to aA_4A_2$ (Required form)

Similarly (2)

$A_2 \to A_2A_5A_3$

$A_2 \to aA_5A_3$ (Required form)

Now

$A_3 \to A_6A_1A_7$

$A_3 \to A_6aA_7$

Hence, the converted grammer in GNF is

$A_1 \to aA_4A_2 \mid a$

$A_2 \to aA_5A_3 \mid a$

$A_3 \to a$

In the original symbols

$E \to a+T \mid a$

$T \to a*F \mid a$

$F \to (a) \mid a$

Q.19 ***Find a grammar in CNF equivalent to the grammar*** $S \to \sim S \mid [S \supset S] \mid p \mid q$ ***(S being the only variable).***

[R.T.U. 2009]

Ans. $S \to \sim S \mid [S \supset S] \mid p \mid q$

$V_N = \{S\}$

$\Sigma = \{\sim, [, \supset,], p, q\}$

$S \to$ Start symbol production

$S \to \sim S$

$S \to [S \supset S]$

$S \to p$

$S \to q$

Given context free grammar is free from null variable, unit production and useless symbol.

Step 1 : Eliminate null production.

Result : In the grammar no null variable so ignore this step.

Step 2 : Eliminate unit production.

Result : There is no unit production so ignore this step also.

Step 3 : Eliminate useless symbol.

Result : No useless symbol, ignore this step.

Step 4 : Convert it into CNF.

Terminal a and b are already in CNF means production

$\left.\begin{matrix} S \to p \\ S \to q \end{matrix}\right\}$ are already in CNF so these will not be converted into CNF.

Following production are not in CNF

$$S \to \sim S$$

$$S \to [S \supset S]$$

Consider $S \to \sim S$

$$S \to V_1 S$$

$$V_1 \to \sim$$

Where V_1 is new non-terminal.

Consider

$$S \to [S \supset S]$$

$$S \to V_2 S \supset S]$$

Where V_2 is new non-terminal.

Now grammar is

$$\left.\begin{matrix} S \to p \\ S \to q \\ S \to V_1 S \\ V_1 \to \sim \\ S \to V_2 S \supset S] \\ V_2 \to] \end{matrix}\right\} \quad ....(A)$$

In grammar (A) following are not in CNF

$$S \to V_2 S \supset S]$$

$$S \to V_2 S V_3 S]$$

$$V_3 \to \supset$$

Now grammar becomes

$$\left.\begin{matrix} S \to V_1 S \\ V_1 \to \sim \\ S \to p \mid q \\ S \to V_2 S V_3 S] \\ V_3 \to \supset \\ V_2 \to [\end{matrix}\right\} \quad ....(B)$$

In grammar (B) production $S \to V_2 S V_3 S]$ is not in CNF. So consider it

$$S \to V_2 S V_3 S]$$

$$S \to V_2 S V_3 S V_4$$

$$V_4 \to]$$

V_4 is new non-terminal.

Now grammar has become

$$\left.\begin{matrix} S \to p \mid q \mid V_1 S \\ V_1 \to N \\ S \to V_2 S V_3 S V_4 \\ V_2 \to [\\ V_3 \to \supset \\ V_4 \to] \end{matrix}\right\} \quad ....(C)$$

In grammar (C) production $S \to V_2 S V_3 S V_4$ is not in CNF. So consider it

$$S \to V_2 S V_3 S V_4$$

$$S \to V_5 V_3 S V_4$$

$$V_5 \to V_2 S$$

Now grammar

$$\left.\begin{matrix} S \to V_1 S \mid p \mid q \\ V_1 \to \sim \\ S \to V_5 V_3 S V_4 \\ V_4 \to] \\ V_5 \to V_2 S \\ V_2 \to [\end{matrix}\right\} \quad ...(D)$$

Now in grammar (D) production $S \to V_5 V_3 S V_4$ is not in CNF, so consider it

$$S \to V_S V_3 S V_4$$

$$S \to V_5 V_6 V_4$$

$$V_6 \to V_3 S$$

Now grammar (D) becomes

$$\left.\begin{matrix} S \to V_1 S \mid p \mid q \\ V_1 \to \sim \\ S \to V_5 V_6 V_4 \\ V_5 \to V_2 S \\ V_2 \to [\\ V_6 \to V_3 S \\ V_2 \to [\\ V_3 \to \supset \\ V_4 \to] \end{matrix}\right\} \quad ....(E)$$

In grammar (E) production $S \to V_5 V_6 V_4$ is not in CNF, consider it

$$S \to V_5 V_6 V_4$$

$$S \to V_7 V_4$$

$$V_7 \to V_5 V_6$$

Now grammar (E) has become

$S \to V_7V_4 \mid V_1S \mid p \mid q$
$V_1 \to \sim$
$V_4 \to]$
$V_7 \to V_5V_6$
$V_5 \to V_2S$
$V_2 \to [$
$V_6 \to V_3S$
$V_3 \to \supset$

This is in required CNF form.

PART-C

Q.20 ***Give detailed description of ambiguity in context free grammar.*** ***[R.T.U. 2015]***

Ans. Ambiguity in CFG

A CFG, G = (V, T, P, S) is ambiguous if there is at least one string W in L(G) for which there are at least two different parse trees, each with its root labeled S and yielding W. Each parse tree corresponds to a left-most or a right-most derivation. The number of different parse trees of a string W is called the degree of ambiguity of W . If no string produced by a grammar G has a degree of ambiguity more than x, the degree of ambiguity of G is x. It is possible to classify ambiguous grammars based on their degree of ambiguity. If the number of distinct parse trees for each string increases with the length of strings generated by a grammar, it is possible that the degree of ambiguity of that grammar is infinite.

Context-free grammar G = (V, P, T, S) is ambiguous if (and only if) there exists a string W in the language of G such that there are two (or more) different parse trees with root S and yield W in G. We can therefore show that a given context-free grammar G = (V, T, P, S) is ambiguous by a direct application of this definition by giving a string W ∈L(G) along with two parse trees with root S and yield W in G.

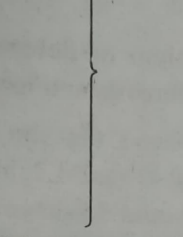
For example, consider the context-free grammar G = (V, T, P, W), where V = {W, A, B), T = {0, 1), W is the start variable, and the grammar includes the following rules.

$W \to A \mid B$

$A \to 0W \mid \in$

$B \to 1W \mid \in$

In order to see that this grammar is ambiguous, it is sufficient to give two different parse trees with root W and with same string, W = ∈, as yield, such as the following.



It is also true that a context-free grammar G = (V, T, P, S) is ambiguous if and only if there exists a string W in the language of G such that there are two or more left-most derivations of W from the start variable.

Thus we can also show that the above grammar ambiguous by noting that the string w = 2 has two left-most derivations, namely,

$W \to GA$ (using the rule $W \to A$)
$\to G\in$ (using the rule $A \to \in$)

and

$W \to GB$ (using the rule $W \to B$)
$\to G\in$ (using the rule $B \to \in$)

Finally, it is true that a context-free grammar G = (V, T, P, S) is ambiguous if and only if there exists a string W in the language of G such that there are two or more right-most derivations of W from the start variable. Thus we can prove that the above grammar G is ambiguous by giving two right-most derivations of the empty string from the start variable W, as well.

Example 2 : Given the following ambiguous grammar,

$E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to a$

There are two left-most derivations for a*a+a

$E \to E+E \to E*E+E \to a*E+E \to a*a+E \to a*a+a$

$E \to E*E \to a*E \to a*E+E \to a*a+E \to a*a+a$

Hence, the degree of ambiguity of a*a+a is two.

Ambiguity Detection and Removal

The problem of deciding whether a given (context-free) grammar for a language is ambiguous is unsolvable . In other

words, there is no general algorithm that can tell us whether a CFG is ambiguous or not. The problem of finding a solution to Post's Correspondence Problem (PCP), which is known to be undecidable, is reducible to the problem of detecting ambiguity in a context-free grammar. Hence, the problem of context-free grammar ambiguity detection is also undecidable.

To handle the ambiguity arising from reduce-shift or reduce-reduce conflicts, disambiguating rules can be written. Disambiguating rules attempt to remove specific known ambiguities. Disambiguating rules can assign priorities to rules (i.e., which rule to choose when a reduce-reduce conflict occurs) and to operations (i.e., whether to perform a shift or a reduce when a shift-reduce conflict occurs). However, there is no algorithm which, given an ambiguous CFG as input, can always produce an unambiguous context-free grammar as output that generates the same language.

Inherent Ambiguity

If all grammars that generate a language are ambiguous, that language is said to be inherently ambiguous. An ambiguous grammar does not necessarily generate an ambiguous language. In other words, for a language L to be unambiguous, at least one of the grammars that can generate it should be unambiguous.

The problem of determining whether an arbitrary language is inherently ambiguous is recursively unsolvable.

In a language L that can be defined as the union of two other languages L1 and L2, all sentences in the intersection of the sets L1 and L2 have two different interpretations because they belong to both L1 and L2. This means that ambiguity is inherent in L, and it is not possible to disambiguate languages such as L.

For example, the language

$L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$, with the language $\{a^n b^n c^n d^n \mid n \geq 1\}$ as $L1 \cap L2$, is inherently ambiguous.

❑❑❑

PUSHDOWN AUTOMATON 3

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***Write three properties of CFL.***

Ans. **Properties of Context Free Languages**

(i) The reverse of a context-free language is context-free, but the complement need not be.

(ii) Every regular language is context-free because it can be described by a regular grammar.

(iii) The intersection of a context-free language and a regular language is always context-free.

Q.2 ***Give definition of PDA.***

Ans. In the theory of computation, a branch of theoretical computer science, a pushdown automation (PDA) is a type of automation that employs a stack. Pushdown automata are used in theories about what can be computed by machines. They are more capable than finite-state machines but less capable than Turing machines.

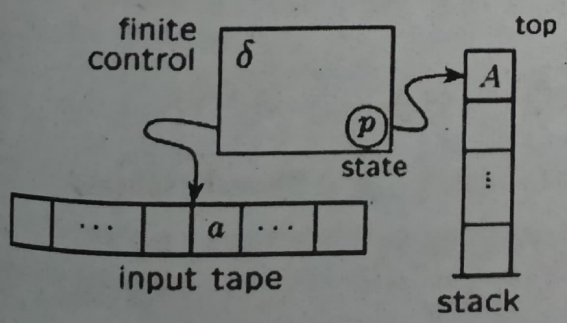
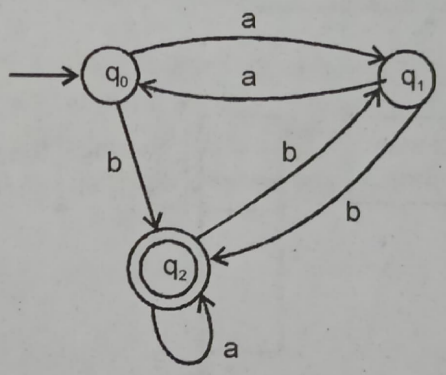


Fig.

PART-B

Q.3 Check whether the strings abb, aba and abbabb are accepted by transition graph or not.[R.T.U. 2019]

Ans. Let us assume the transition graph as following :



$Q = \{q_0, q_1, q_2\}$

I.S. $= q_0$

Final state $= q_2$

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

Take our first string abb;

$\delta(q_0, a) = q_1$

$\delta(q_1, b) = q_2$

$\delta(q_2, b) = q_1$

and q_1 is not final state so this string is not accepted.

Now take the second string for the given transition system aba;

$\delta(q_0, a) = q_1$

$\delta(q_1, b) = q_2$

$\delta(q_2, a) = q_2$

So q_2 is final state, this string is accepted.

Now take the third string for the given system abbabb;

$\delta(q_0, a) = q_1$

$\delta(q_1, b) = q_2$

$\delta(q_2, b) = q_1$

$\delta(q_1, a) = q_0$

$\delta(q_0, b) = q_2$

$\delta(q_2, b) = q_1$

Since q_1 is not final state, so this string is not accepted by this transition graph.

Q.4 ***Define pushdown automaton model and its role, also illustrate the move relation in details.***

[R.T.U. 2016]

Ans. Pushdown Automaton :

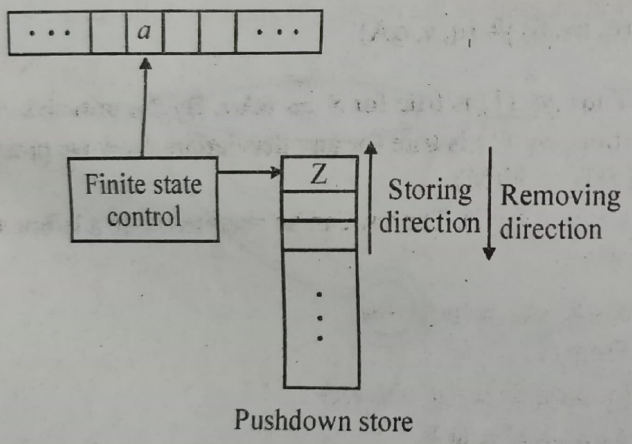


Fig. : Model of pushdown automaton

Definition : A pushdown automaton consists of

(i) A finite nonempty set of states denoted by Q,

(ii) A finite nonempty set of input symbols denoted by Σ,

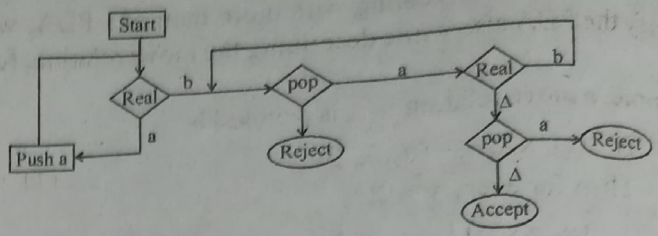
(iii) A finite nonempty set of pushdown symbols denoted by Γ.

(iv) A special pushdown symbol called the initial symbol on the pushdown store denoted by Z_0.

(vi) The set of final states, a subset of Q denoted by F, and

(vii) The transition function δ from $Q \times (\Sigma \cup (\wedge)) \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.

Symbolically, a PDA is a 7-tuple, viz. $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.



This is a language in which equal number of a's are followed by equal number of b's, the logic for this PDA can be applied as: first we will push all a's into the stack then on reading every single b each a is popped from the stack.

$$(q_0, aaabbbab, z_0) \rightarrow (q_0, aabbbab, az_0)$$
$$\rightarrow (q_0, abbbab, aaz_0)$$
$$\rightarrow (q_0, bbbab, aaaz_0)$$
$$\rightarrow (q_1, bbab, aaz_0)$$
$$\rightarrow (q_1, bab, az_0)$$
$$\rightarrow (q_1, ab, z_0)$$
$$\rightarrow (q_1, b, az_0)$$
$$\rightarrow (q_1, \varepsilon, z_0)$$
$$\rightarrow (q_2, \varepsilon)$$

Accept State

Let A be a pushdown automata (PDA) A move relation (denoted by $\vdash$) between ID's is defined as $(q, a_1, a_2 \ldots a_n, z_m \ldots z_2 z_1) \vdash (q', a_2 \ldots a_n, \beta z_m \ldots z_2)$ if $\delta(q, a_1, z_1)$ contains (q', β).

As $\vdash$ defines a relation in the set of all IDs of a PDA, we can define the reflexive-transitive closure $\overset{*}{\vdash}$ which represents a definite sequence a definite sequence of n moves where n is any non-negative integer.

If $(q, x, \alpha) \overset{*}{\vdash} (q', y, \beta)$ represents n moves, we write

$$(q, x, \alpha) \overset{n}{\vdash} (q', y, \beta).$$

In particular $(q, x, \alpha) \overset{o}{\vdash} (q, x, \alpha)$.

Also $(q, x, \alpha) \overset{*}{\vdash} (q, y, \beta)$ can be split as

$(q, x, \alpha) \vdash (q_1, x_1, \alpha_1) \vdash (q_2, x_2, \alpha_2) \vdash \ldots$
$\vdash (q', y, \beta)$ for some $x_1, x_2, \ldots \in \Sigma^*, \alpha_1 \alpha_2 \ldots \in \Gamma$.

Note : When we are dealing with more than one PDA, we specify the PDA also while describing the move relation, for example, a move relation in A is denoted by $\vdash^A$

If $(q_1, x, \alpha) \vdash (q_2, \lambda, \beta)$ ... (1)

Then for every $y \in \Sigma^*$,

$(q_1, xy, \alpha) \vdash (q_2, y, \beta)$ ... (2)

Conversely if $(q_1, xy, \alpha) \overset{*}{\vdash} (q_2, y, \beta)$ for some $y \in \Sigma^*$, then $(q1, x, \mu) \overset{*}{\vdash} (q_2, \lambda, \beta)$.

The property of move relation of PDA is :

$(q, x, \alpha) \overset{*}{\vdash} (q', \lambda, \nu)$

then for every $\beta \in \Gamma^*$.

$(q, x, \beta\alpha) \vdash (q', \lambda, \beta\nu)$

Q.5 ***If L is context free language then prove that there exists PDA M such that L=N(M)*** ***[R.T.U. 2015]***

Ans. Let G = (N, T, S, P) be a CFG generating L(G).

Let $M = (\{q_0\}, \Sigma, \Gamma, \delta, q_0, z_0, \phi)$

where $\Sigma = T$

$\Gamma = N \cup T$

$z_0 = S$

and δ is defined as follows. For each $a \in \Sigma$ and $A \in N$.

(i) $\delta(q_0, \lambda, A) = \{(q_0, \alpha^R)/A \to \alpha \text{ is in p}\}$

(ii) $\delta(q_0, a, a) = \{(q_0, \lambda)\}$

This pushdown symbols in M are terminals and non-terminals, if the PDA reads a variable A on the top of pushdown store, it match a λ–move by replacing the reverse of R.H.S. of any A-production (after erasing A). If the PDA read, a terminal 'a' on PDS and if it matches with the current input symbol, then the PDA erases a. In other cases PDA halts.

If $w \in L(G)$ is obtained by a leftmost derivation $S \Rightarrow u_1A_1\alpha_1 \Rightarrow u_1u_2A_2\alpha_2\alpha_1 \Rightarrow \ldots \Rightarrow w$, then A can empty the PDS on application of input string w. The first move of M is by a λ move corresponding to $S \to u_1A_1\alpha_1$. The PDA erases the symbols in u_1 by processing prefix of w. Now the topmost symbol in PDS in A_1. Once again by applying the λ–move corresponding to $A_1 \to u_2A_2\alpha_2$, the PDA erases A_1 and stores $\alpha_2A_2u_2$ above α_1 proceeding in this way, the PDA empties the PDA by processing the entire string w.

Now we prove that L(G) = N (M). Let $w \in L(G)$. Then it can be derived by a leftmost derivation. Any sentential form in a leftmost derivation is of the form $u A \alpha$ where $u \in T^*$, $A \in N$ and $\alpha \in (N \cup T)^*$. We prove the following auxillary result.

If $S \Rightarrow uA\alpha$ by a leftmost derivation, then $(q, uv, S) \vdash (q, v, \alpha A)$ for every $v \in \Sigma^*$ ....(1)

We prove eq. (1) by induction on the number of steps in the derivation of $uA\alpha$.

If $S \overset{.}{\Rightarrow} uA$, then $u = \lambda$, $\alpha = \lambda$ and $S = A$.

As $(q, v, S) \vdash (q, v, S)$, there is a basis for induction.

Suppose $S \overset{n+1}{\Rightarrow} uA\alpha$ by a leftmost derivation. This derivation can be split as $\overset{n}{\vdash} u_1A_1\alpha_1 \Rightarrow uA\alpha$. If in the A_1 – production we apply in the last step $A_1 \to u_2A\alpha_2$, then $u = u_1u_2$, $\alpha = \alpha_1\alpha_2$.

As $S \overset{n}{\Rightarrow} u_1A_1\alpha_1$, by induction hypothesis,

$(q_1, u_1u_2, v, S) \overset{n}{\vdash} (q_1, u_2v, \alpha_1A_1)$ ....(2)

As $A_1 \to u_2A\alpha_2$ is a production in P, by rule (i), we get $(q, \lambda, A_1) \overset{n}{\vdash} (q, \lambda, \alpha_2, Au_2)$. We get

$(q, u_2v, \alpha_1A) \vdash (q, u_2v, \alpha_1\alpha_2, Au_2) \vdash (q, u, A\alpha_1\alpha_2)$ by rule (ii)

Hence $(q, u_2v, \alpha_1A_1) \overset{n}{\vdash} (q, v, \alpha_1\alpha_2, A)$

But $u_1u_2 = u$ and $\alpha_1\alpha_2 = \alpha$. So from eq. (2) and (3), we have

$(q, uv, S) \overset{n}{\vdash} (q, v, \alpha A)$

Thus eq. (1) is true for $S \overset{n+1}{\Rightarrow} uA\alpha$. By the principle of induction, eq. (1) is true for any derviation. Now we prove that $L(G) \subseteq N(M)$.

Let $w \in L(G)$. Then w can be obtained from a leftmost derivation.

$S \overset{.}{\Rightarrow} uAv \Rightarrow uu'v = w$

From (1)

$(q, uu'v, S) \overset{n}{\vdash} (q, u'v, vA)$

As $A \to u'$ is in P,

$(q, a\,u'v, vA) \overset{n}{\vdash} (q, u'v, vu')$

By rule (2)

$(q, u'v, vu') \overset{n}{\vdash} (q, \lambda, \lambda)$

Therefore,

$w = uu'v \in N(A)$ proving $L(G) \subseteq N(m)$. Next we prove $N(m) \subseteq L(G)$.

Before proving the inclusion, let us prove the following auxillary result.

$S \overset{.}{\Rightarrow} u\alpha$, if $(q, uv, S) \overset{n}{\vdash} (q, v, \alpha)$ ...(4)

We prove eq. (4) by the number of moves in $(q, uv, S) \overset{n}{\vdash} (q, v, \alpha)$. If $(q, uv, S) \overset{n}{\vdash} (q, v, \alpha)$, then $u = \lambda$, $S = \alpha$; obviously $S \overset{.}{\Rightarrow} \lambda\alpha$.

Thus there is a basis for induction.

Let us assume eq. (4) when the number of moves is n. Assume

$(q, uv, S) \vdash^n (q, v, \alpha)$ ...(5)

The last move in eq. (5) is obtained either from $(q, \lambda, A) \vdash (q, \lambda, \alpha')$ or $(q, \alpha, \alpha,) \vdash (q, \lambda, \lambda)$. In the first case (5) can be split as

$(q, uv, S) \vdash^n (q, v, \alpha^2 A) \vdash (q, v, \alpha_2\alpha_1) = (q, v, \alpha)$

By induction hypothesis, $S \Rightarrow uA\alpha_2$, and the last move is induced by $A \to \alpha_1$.

Thus

$S \Rightarrow uA\alpha_2$ implies $\alpha_1\alpha_2 = \alpha$. So $S \Rightarrow uA\alpha_2 \Rightarrow u\alpha_1\alpha_2 = u\alpha$.

In the second case (5) can be split as

$(q, uv, S) \vdash^n (q, av, \alpha a) \vdash (q, v, \alpha)$

Also $u = u'a$ for some $u' \in \Sigma$.

So $(q, u'av, S) \vdash^n (q, av, \alpha a)$ implies (by induction hypothesis) $S \Rightarrow u'a\alpha = u\alpha$. Thus in both cases we have shown that $S \Rightarrow u\alpha$. By the principle of induction eq. (4) is true.

Now we can prove that if $w \in N(M)$ then $w \in L(G)$. As $w \in N(A)$,

$(q, w, S) \vdash^n (q, \lambda, \lambda)$. By taking $u = w$, $v = \lambda$, $\alpha = \lambda$ and applying (4), we get $S \Rightarrow w\lambda = w$

i.e. $w \in L(G)$.

Thus $L(G) = N(M)$

Q.6 ***Explain the steps involving in conversion from context free grammar to pushdown automata with example.*** *[R.T.U. 2013]*

Ans. Suppose we have to construct a PDA A equivalent to the following CFG.

$\delta \to 0BB$

$B \to 0S/1S/0$

Now, we can define PDA A as follows:

$A = (\{q\}, \{0, 1\}, \{S, B, 0, 1\}, \delta, q, s, \phi)$

Where δ is defined by the following rules:

$R_1 : \delta(q, \wedge, S) = \{(q, 0BB)\}$

$R_2 : \delta(q, \wedge, B) = \{(q, 0S), (q, 1S), (q, 0)\}$

$R_3 : \delta(q, 0, 0) = \{(q, \wedge)\}$

$R_4 : \delta(q, 1, 1) = \{(q, \wedge)\}$

Note : Rules R_1 to R_4 generated according to given CFG, for example

We generated rule 2 (R_2) for

$B \to 0S/1S/0$

Now if we want to test whether a string is N (A) or not, we an test as follows :

For example,

We want to test whether 010000 is in N (A).

(q, 010000, S)
(q, 010000, 0BB) by R_1
(q, 10000, BB) by R_3
(q, 10000, 1SB) by R_2
(q, 0000, SB) by R_4
(q, 0000, 0BBB) by R_1
(q, 000, BBB) by R_3
(q, 000, 000) by R_2
(q, ∧, ∧) by R_3

Thus

$010000 \subseteq N(A)$

Q.7 ***Convert the given PDA to CFG***
$A = (\{q_0, q_1\}, \{a, b\}, \{z_0, z\}, s, q_0, z_0\phi)$
S Ps given by
$S(q_0, b, z_0) = (q_0, zz_0)$
$S(q_0, n, z_0) = (q_0, n)$
$S(q_0, b, z) = (q_0, zz)$
$S(q_0, a, z) = (q_1, z)$
$S(q_0, b, z) = (q_1, n)$
$S(q_1, a, z_0) = (q_0, z_0)$ *[R.T.U. 2012]*

Ans. $G = (V_N, \{a, b\}, P, S)$

Where V_N consists of S, $[q_0, z_0, q_0]$

$[q_0, z_0, q_1]$, $[q_0, z, q_0]$, $[q_0, z_1, q_1]$, $[q_1, z_0, q_0]$,

$[q_1, z_0, q_1]$, $[q_1, z, q_0]$, $[q_1, z, q_1]$

The productions are

$P_1 : S \to [q_0, z_0, q_0]$

$P_2 : S \to [q_0, z_0, q_1]$

$\delta(q_0, b, z_0) = \{(q_0, z\ z_0)\}$ yields

$P_3 : [q_0, z_0, q_0] \to b[q_0, z, q_0][q_0, z_0, q_0]$

$P_4 : [q_0, z_0, q_0] \to b[q_0, z, q_1][q_1, z_0, q_0]$

$P_5 : [q_0, z_0, q_1] \to b[q_0, z, q_0][q_0, z_0, q_1]$

$P_6 : [q_0, z_0, q_1] \to b[q_0, z, q_1][q_1, z_0, q_1]$

$\delta(q_0, \wedge, z_0) = \{(q_0, \wedge)\}$ gives

$P_7 : [q_0, z_0, q_0] \to n$

$\delta(q_0, b, z) = \{(q_0, zz)\}$ gives

$P_8 : [q_0, z, q_0] \to b[q_0, z, q_0][q_0, z, q_0]$

$P_9 : [q_0, z, q_0] \to b[q_0, z, q_1][q_1, z, q_0]$

$P_{10} : [q_0, z, q_1] \to b[q_0, z, q_0][q_0, z, q_1]$

$P_{11} : [q_0, z, q_1] \to b[q_0, z, q_0][q_0, z, q_1]$

$\delta(q_0, a, z) = \{(q_1, z)\}$ yields

$P_{12} : [q_0, z, q_0] \to a[q_1, z, q_0]$

$P_{13} : [q_0, z, q_1] \to a[q_1, z, q_1]$

$\delta(q_1, b, z) = \{(q_1, \wedge)\}$ gives

$P_{14} : [q_1, z, q_2] \to b$

$\delta(q_1, a, z_0) = \{(q_0, z_0)\}$ gives

$P_{15} : [q_1, z_0, q_0] \to a[q_1, z_0, q_0]$

$P_{16} : [q_1, z_0, q_1] \to a[q_0, z_0, q_1]$

$P_1 - P_{16}$ gives the productions in P.

Q.8 ***What is PDA, Explain. Construct PDA equivalent to $L = \{a^n b^{n+m} a^m / n, m \geq 0\}$.*** *[R.T.U. 2012]*

Ans. PDA : Refer to *Q.4*.

The PDA A accepting $\{a^n b^{n+m} a^{m/n}, m \geq 0\}$ is defined as follows :

$A = (\{q_0, q_1\}, \{a, b\}, \{a, z_0\}, \delta, q_0, z_0, \phi)$

Where δ is defined by

$R_1 : \delta(q_0, a, z_0) = \{(q_0, az_0)\}$

$R_2 : \delta(q_0, a, a) = \{(q_0, aa)\}$

$R_3 : \delta(q_0, b, a) = \{(q_1, a)\}$

$R_4 : \delta(q_1, b, a) = \{(q_1, a)\}$

$R_5 : \delta(q_1, a, a) = \{(q_1, \wedge)\}$

$R_6 : \delta(q_1, \wedge, z_0) = \{(q_1, \wedge)\}$

We start storing a's until a b occurs. When the current input symbol is b, the state changes, but no change in PDS occurs. Once all the b's in the input string are exhausted, the remaining a's are erased. Using R_6, z_0 is erased. So

$(q_0, a^n b^{m+n} a^{m/n}, z_0) \vdash^* \ldots\ldots (q_1, \wedge, z_0) \vdash \ldots\ldots (q_1, \wedge, \wedge)$

This means that $a^n b^{m+n} a^{m/n} \in N(A)$. We can show that

$$N(A) = \{a^n b^{m+n} a^{m/n} \mid m \geq 0\}$$

By using Rule

Define $G = (V_N, \{a, b\}, P, S)$ where V_N consisting of

$[q_0, z_0, q_0], [q_1, z_0, q_0], [q_0, a, q_0], [q_1, a, q_0]$

$[q_0, z_0, q_1], [q_1, z_0, q_1], [q_0, a, q_1], [q_1, a, q_1]$

The production in P are constructed as follows :

The S – production are

$P_1 : S \to [q_0, z_0, q_0]$

$P_2 : S \to [q_0, z_0, q_1]$

$\delta(q_0, a, z_0) = \{(q_0, az_0)\}$ induces

$P_3 : [q_0, z_0, q_0] \to a[q_0, a, q_0][q_0, z_0, q_0]$

$P_4 : [q_0, z_0, q_0] \to a[q_0, a, q_1][q_1, z_0, q_0]$

$P_5 : [q_0, z_0, q_1] \to a[q_0, a, q_0][q_0, z_0, q_1]$

$P_6 : [q_0, z_0, q_1] \to a[q_0, a, q_1][q_1, z_0, q_1]$

$\delta(q_0, a, a) = \{(q_0, aa)\}$ yields

$P_7 : [q_0, a, q_0] \to a[q_0, a, q_0][q_0, a, q_0]$

$P_8 : [q_0, a, q_0] \to a[q_0, a, q_1][q_1, a, q_0]$

$P_9 : [q_0, a, q_1] \to a[q_0, a, q_0][q_0, a, q_1]$

$P_{10} : [q_0, a, q_1] \to a[q_0, a, q_1][q_1, a, q_1]$

$\delta(q_0, b, a) = \{(q_1, a)\}$ gives

$P_{11} : [q_0, a, q_0] \to b[q_1, a, q_0]$

$P_{12} : [q_0, a, q_1] \to b[q_1, a, q_1]$

$\delta(q, b, a) = \{(q_1, a)\}$ yields

$P_{13} : [q_1, a, q_0] \to b[q_1, a, q_0]$

$P_{14} : [q_1, a, q_1] \to b[q_1, a, q_1]$

$\delta(q, a, a) = \{(q_1, \wedge)\}$ gives

$P_{15} : [q_1, a, q_1] \to a$

$\delta(q_1 \wedge, Z_0) = \{(q_1 \wedge)\}$ yields

$P_{16} : [q_1, q] \to \wedge$

Q.9 ***Convert the given PDA to CFG***

$A = (\{q_0, q_1\}, \{a, b\}, \{z_0, z\}, s, q_0, z_0\phi)$

S Ps given by

$S(q_0, b, z_0) = (q_0, zz_0)$

$S(q_0, n, z_0) = (q_0, n)$

$S(q_0, b, z) = (q_0, zz)$

$S(q_0, a, z) = (q_1, z)$

$S(q_0, b, z) = (q_1, n)$

$S(q_1, a, z_0) = (q_0, z_0)$ *[R.T.U. 2012]*

Ans. $G = (V_N, \{a, b\}, P, S)$

Where V_N consists of S, $[q_0, z_0, q_0]$

$[q_0, z_0, q_1]$, $[q_0, z, q_0]$, $[q_0, z_1, q_1]$, $[q_1, z_0, q_0]$,

$[q_1, z_0, q_1]$, $[q_1, z, q_0]$, $[q_1, z, q_1]$

The productions are

$P_1 : S \to [q_0, z_0, q_0]$

$P_2 : S \to [q_0, z_0, q_1]$

$\delta(q_0, b, z_0) = \{(q_0, z\ z_0)\}$ yields

$P_3 : [q_0, z_0, q_0] \to b[q_0, z, q_0][q_0, z_0, q_0]$

$P_4 : [q_0, z_0, q_0] \to b[q_0, z, q_1][q_1, z_0, q_0]$

$P_5 : [q_0, z_0, q_1] \to b[q_0, z, q_0][q_0, z_0, q_1]$

$P_6 : [q_0, z_0, q_1] \to b[q_0, z, q_1][q_1, z_0, q_1]$

$\delta(q_0, \wedge, z_0) = \{(q_0, \wedge)\}$ gives

$P_7 : [q_0, z_0, q_0] \to n$

$\delta(q_0, b, z) = \{(q_0, zz)\}$ gives

$P_8 : [q_0, z, q_0] \to b[q_0, z, q_0][q_0, z, q_0]$

$P_9 : [q_0, z, q_0] \to b[q_0, z, q_1][q_1, z, q_0]$

$P_{10} : [q_0, z, q_1] \to b[q_0, z, q_0][q_0, z, q_1]$

$P_{11} : [q_0, z, q_1] \to b[q_0, z, q_0][q_0, z, q_1]$

$\delta(q_0, a, z) = \{(q_1, z)\}$ yields

$P_{12} : [q_0, z, q_0] \to a[q_1, z, q_0]$

$P_{13} : [q_0, z, q_1] \to a[q_1, z, q_1]$

$\delta(q_1, b, z) = \{(q_1, \wedge)\}$ gives

$P_{14} : [q_1, z, q_2] \to b$

$\delta(q_1, a, z_0) = \{(q_0, z_0)\}$ gives

$P_{15} : [q_1, z_0, q_0] \to a[q_1, z_0, q_0]$

$P_{16} : [q_1, z_0, q_1] \to a[q_0, z_0, q_1]$

$P_1 - P_{16}$ gives the productions in P.

Q.10 ***Define the context – free grammar and find the context free grammar for the following languages :***

(i) $L=\{a^n b^m : n \le m+3\}$

(ii) $L=\{a^n b^m : n \ne 2m\}$ ***[R.T.U. 2011]***

Ans. Context – Free Grammar : A context free grammar (CFG) is a 4–tuple G — (V, Σ, S, P), where V and Σ are disjoint finite sets, S is an element of V and P is a finite set of formulas of the form $A \to \alpha$, where $A \in V$ and $a \in (V \cup \Sigma)^*$.

The elements of V are called variables, or non terminal symbols and those of the alphabet Σ are called terminal symbols, or terminals. S is called the start symbol, and the elements of P are called grammar rules, or productions.

(i) We use A to generate $L_1 = \{a^n b^m : n \le m + 3\}$ and B to generate $L_2 = \{b^m \mid 0 \le m\}$. One can verify that $L = L_1 L_2$ because attaching L_2 to L_1 means we can add any number of b's to the end of the string in L_1. Thus, m, the number of b's is no longer bounded by n, the number of a's.

$S \to AB$

$A \to \lambda|a|aa|aaa|aAb$

$B \to \lambda|bB$

(ii) We use E to generate $L_1 = \{a^n b^m \mid n = 2m\}$. A to generate non-empty string of a's and B non-empty strings of b's. Thus, AE and EB can generate $\{a^n b^m \mid n > 2m\}$ and $\{a^n b^m \mid n < 2m\}$, respectively

$S \to AE|EB$

$A \to a|aA$

$B \to b|bB$

$E \to \lambda|aaEb$

Q.11 ***Construct a PDA that accepts the language generated by the grammar***

$S \to aSbb$

$S \to aSbb$

OR

Construct a PDA that accepts the language generated by grammar with predictions $S \to aSbb\ |a$. ***[R.T.U. 2011]***

Ans. The PDA $A = (\{q\}, \{a,b\}, \{S,a,b\}), \delta, q, S\}$ where δ :

(i) $\delta(q, z_0, S) = \{(q, aSbb), (q, aab)\}$

(ii) $\delta(q, a, a) = \{(q, \varepsilon)\}$

(iii) $\delta(q, b, b) = \{(q, \varepsilon)\}$

Q.12 ***Let $L = \{a^i b^j c^k / i, j, k \ge 1$ and $i + j = k\}$. Find a PDA (which accepts via final state) that recognizes L.*** ***[R.T.U. 2010]***

Ans. An informal description of a pushdown automaton that recognizes the language

$$A = \{a^i b^j c^k / i, j, k \ge 1 \text{ and } i + j = k\}$$

The language A is the union of the languages

$\{a^i b^j c^k \mid i, k \ge 0\}$

and

$\{a^i b^k c^k \mid i, k \ge 0\}$

A PDA for the first of these language is

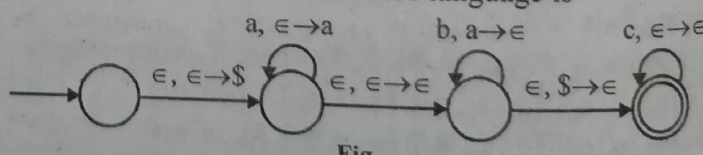


Fig.

A PDA for the second language is

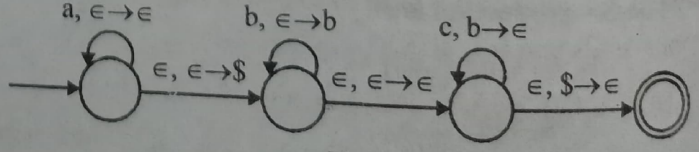


Fig.

Therefore, a PDA for the language A is

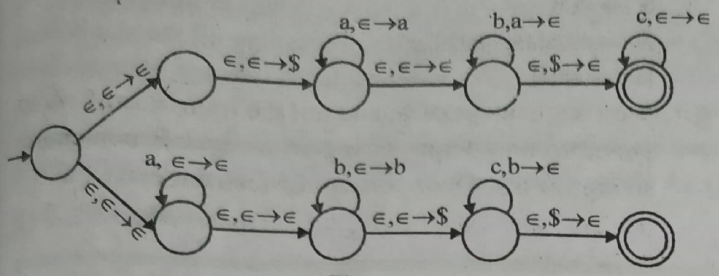


Fig.

Design a PDA that accepts the language {w | w contains an equal number of a's and b's}

(1) Strategy will be to keep the excess symbols, either a's and b's on the stack.
(2) One state will represent an excess of a's.
(3) Another state will represent an excess of b's.
(4) We can tell when the excess switches from one symbol to the other because at the point the stack will be empty.
(5) In fact, when the stack is empty, we may return to the start state.

Q.13 ***(a) Construct a PDA accepting the set of all strings over {a, b} with equal number of a's and b's.***

(b) What is difference between finite automaton and push down automaton?

[R.T.U. 2009, Raj.Univ. 2007]

Ans.(a) Let $A=(\{q\},[a,b],[z_0,a,b],\delta,q,z_0,\phi)$

Where δ is defined by the following rules:

$$\delta(q,a,z_0)=\{(q,a\,z_0)\} \qquad \delta(q,b,z_0)=\{(q,bz_0)\}$$

$$\delta(q,a,a)=\{(q,aa)\} \qquad \delta(q,b,b)=\{(q,bb)\}$$

$$\delta(q,a,b)=\{(q,\Lambda)\} \qquad \delta(q,b,a)=\{(q,\Lambda)\}$$

$$\delta(q,\Lambda,z_0)=\{(q,\Lambda)\}$$

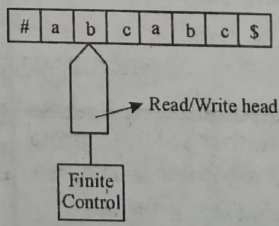
Construction of δ : If A is in initial ID, then using Rule, A pushes the first symbol of the input string on PDS. If it is a or b. Here we want to match the number of occurrences of a and b, so the construction is simpler. We start by storing a symbol of the input string and continue storing until the other symbol occurs. If the topmost symbol in PDS is a and the current input symbol is b, a in PDS is erased. If w has equal number of a's and b's , then

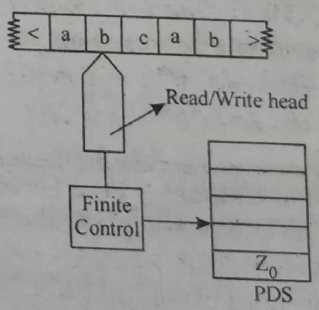
$(q,w,z_0)\vdash^{*}(q,\Lambda,z_0)\vdash(q,\Lambda,\Lambda)$. So $w\in N(A)$.

We can show that N(A) is the given set of strings over {a, b} using the construction of S.

Ans. (b)

Finite Automaton	Push Down Automaton
1. It consists of 5 tuples	1. It consists of 7 tuples
2. $M=(Q,\Sigma,\delta,q_0,F)$	2. $M=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$
Q = Finite non-empty set of states	Q = Finite non-empty set of states
Σ = Finite non-empty set of input symbols	Σ = Finite non-empty set of input symbols
δ = Transition mapping function which maps $Q\times\Sigma$ into Q	δ = Transition function which maps $Q\times\{\Sigma\cup\{\Lambda\}\}\times\Gamma$ into $Q\times\Gamma^*$ *i.e.* $Q\times\{\Sigma\cup\{\Lambda\}\}\times\Gamma$ into $Q\times\Gamma^*$
q_0 = Initial State	q_0 = Initial state
F = Final state	F = Final state
Γ = External symbol	
Z_0 = Initial symbol of PDS (Push down stack)	
3. There are 2 types of finite automata (i) NDFA (Non deterministic FA) (ii) DFA (Deterministic FA)	3. There are 2 types of push down automata. (i) NPFA (Non deterministic FA) (ii) DPDA (Deterministic PDA)
4. FA is less powerful than PDA	4. PDA is more powerful than FA.
5. FA does not store any information about previous or current input symbol. Since does not have storage area.	5. PDA stores the information about previous and current input symbol. Since, it have push down stack (PDS) which stores related information.
6. NDFA and DFA have equal power. That means NDFA = DFA.	6. 2 stack/n-stack PDA has more power than single stack NPDA. So, 2 stack/ n-stack PDA are more powerful.
7. Structure of FA is:	7. Structure of PDA is :





Q.14 ***Design a PDA to accept the language :***

$$L=\{wcw^T \mid w\in(a,b)^*\}$$

[Raj.Univ. 2007]

Ans. Consider the given PDA :

$$A=(\{q_0,q_1,q_f\},\{a,b,c\},\{a,b,z_0\},\delta,q_0,z_0,\{q_f\})$$

Where δ is defined as :

$$\delta(q_0,a,z_0)=\{(q_0,a\,z_0)\}, \quad \delta(q_0,b,z_0)=\{(q_0,b\,z_0)\}$$

$\delta(q_0,a,a)=\{(q_0,aa)\}$, $\delta(q_0,b,a)=\{(q_0,ba)\}$

$\delta(q_0,a,b)=\{(q_0,ab)\}$, $\delta(q_0,b,b)=\{(q_0,bb)\}$

$\delta(q_0,c,a)=\{(q_1,a)\}$, $\delta(q_0,c,b)=\{(q_1,b)\}$

$\delta(q_0,c,z_0)=\{(q_1,z_0)\}$

$\delta(q_1,a,a)=\delta(q_1,b,b)=\{(q_1,\wedge)\}$

$\delta(q_1,\wedge,z_0)=\{(q_f,z_o)\}$

Let $wcw^T \in L$, write $w=a_1a_2....a_n$, where each a_i is either a or b. Then, we have :

$(q_0,\ a_1a_2...a_n\ cw^T, z_0)$

$\vdash^* (q_0, cw^T, a_n\ a_{n-1}...a_1\ z_0)$

$\vdash^* (q_1, a_n\ a_{n-1}...a_1, a_n\ a_{n-1}...a_1\ z_0)$

$\vdash^* (q_1,\wedge,z_0)$

$\vdash (q_f,\wedge,z_0)$

Therefore, $wcw^T \in T(A)$, i.e. $L \subseteq T(A)$

To prove the reverse inclusion, it is enough to show that $L^c \subseteq T(A)^c$

Let $x \in L^c$.

Case 1 : x does not have the symbol C. In this case the PDA never makes a transition to q_1. So the PDA cannot make a transition to q_f as we cannot apply rule. Thus, $x \in T(A)^c$.

Case 2 : $x = w_1 cw_2$

$w_2 \neq w_1^T$

$(q_0, w_1\ cw_2, z_0)$

$\vdash^* (q_0, cw_2, w_1^T\ z_0)$

$\vdash (q_1, w_2, w_1^T\ z_0)$

As $w_2 \neq w_1^T$, the PDA cannot reach an ID of the form $(q_1, \wedge, z_0)$. So we cannot apply rule.

Therefore, $x \in T(A)^c$.

Thus we have proved $L^c \subseteq T(A)^c$

Second type of acceptance

Let $A=(Q,\Sigma,\tau,\delta,q_0,z_0,F)$ be a PDA. The set N(A) accepted by null store (or empty store) is defined by $N(A)=\{w \in \Sigma^* \mid q_0,w,z_0 \vdash^* (q,\wedge,\wedge)$ for some $q \in Q\}$

In other words, w is in N(A). If A is in initial ID (q_0, w, z_0) and empties the PDS after processing all the symbols of w. So in defining N(A), we consider the change brought about an PDS by application of w, and not the transition of states.

Q.15 *Write short note on Pumping Lemma for CFG*

[Raj.Univ. 2007]

Ans. Pumping Lemma for Context-Free Languages : The pumping lemma for context-free languages gives a method of generating an infinite number of strings from a given sufficiently long string in a context-free language L. It is used to prove that certain languages are not context-free. The construction we make use of in proving pumping lemma yields some decision algorithms regarding context-free languages.

Lemma : Let G be a context-free grammar in CNF and T be a derivation tree in G. If the length of the longest path in T is less than or equal to k, then the yield of T is of length less than or equal to 2^{k-1}.

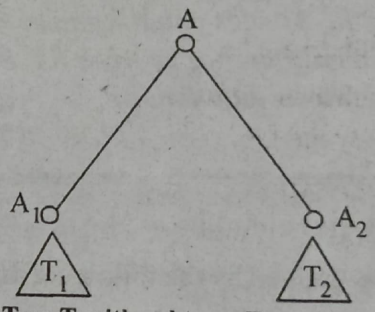


Fig. Tree T with subtrees T_1 and T_2.

Proof : We prove the result by induction on k, the length of the longest path for all A-trees (Recall an A-tree is a derivation tree whose root has lable A). When the longest path is an A-tree is of length 1, the root has only one son whose label is a terminal (when the root has two sons, the labels are variables). So the yield is of length 1. Thus, there is basis for induction.

Assume the result for $k-1(k>1)$. Let T be an A-tree with a longest path of length less than or equal to k. As $k>1$, the root of T has exactly two sons with labels A_1 and A_2. The two subtrees with the two sons as roots have the longest paths of length less than or equal to k – 1

If w_1 and w_2 are their yields, then by induction hypothesis, $|w_1| \leq 2^{k-2}, |w_2| \leq 2^{k-2}$.

So the yield of $T = w_1w_2, |w_1w_2| \leq 2^{k-2}+2^{k-2}=2^{k-1}$. By the principle of induction, the result is true for all A-trees, and hence for all derivation trees.

Theorem : (Pumping lemma for context-free languages): Let L be a context-free language. Then we can find a natural number n such that :

(i) Every $z \in L$ with $|z| \geq n$ can be written as uvwxy for some strings u, v, w, x, y.

(ii) $|vx| \geq 1$

(iii) $|vwx| \leq n$

(iv) $uv^k wx^k y \in L$ for all $k \geq 0$

Proof : When $\Lambda \in L$; we consider $L - \{\Lambda\}$ and construct a grammar $G = (V_N, \Sigma, P, S)$ CNF generating $L - \{\Lambda\}$ (when $\Lambda \notin L$, we construct G in CNF generating L).

Let $|V_N| = m$ and $n = 2^m$. To prove that n is the required number, we start with $z \in L, |z| \geq 2^m$, and construct a derivation tree T (parse tree) of z. If the length of a longest path in T is at most m, by lemma, $|z| \leq 2^{m-1}$ (since z is the yield of T). But $|z| \geq 2^m > 2^{n-1}$. So T has a path, say Γ of length greater than or equal to $m+1$.Γ has at least m + 2 vertices and only the last vertex is a leaf. Thus in Γ all the labels except the last one are variables. As $|V_N| = m$ some label is repeated.

We choose a repeated label as follows : We start with the leaf of Γ and travel along Γ upwards. We stop when some label, say B, is repeated (Among several repeated labels, B is the first). Let v_1 and v_2 be the vertices with label B, v_1 being nearer the root. In Γ, the portion of the path from v_1 to the leaf has only one label, namely B, which is repeated, and so its length is at most m + 1.

Let T_1 and T_2 be the subtrees with v_1, v_2 as roots and z_1, w as yields, respectively. As Γ is a longest path in T, the portion of Γ from v_1 to the leaf is a longest path in T_1 and of length at most m + 1. By lemma $|z_1| \leq 2^m |z_1| \leq 2^m$ (since z_1 is the yield of T_1).

For better understanding, we illustrate the construction for the grammar whose productions are

$S \to AB, A \to aB \mid a, B \to bA \mid b,$ as in Fig.

In the figure,

$\Gamma = S \to A \to B \to A \to B \to b$

$z = ababb, z_1 = bab, w = b$

$v = ba,\ x = \Lambda, u = a, y = b$

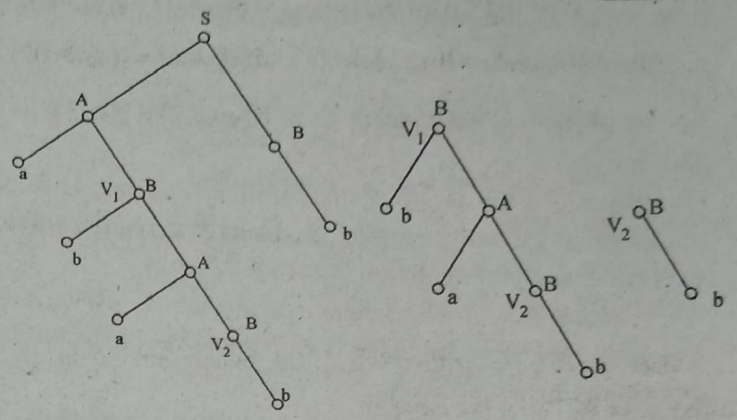


Fig. : Tree T and its subtrees T_1 and T_2

As z and z_1 are the yields of T and a proper subtree T_1 of T, we can write $z = uz_1y$. As z_1 and w are the yields of T_1 and a proper subtree T_2 of T_1, we can write $z_1 = vwx$. Also $|vwx| > |w|$. So $|vx| \geq 1$. Thus, we have $z = uvwxy$ with $|vwx| \leq n$ and $|vx| \geq 1$. This proves the points (i)-(iii) of the theorem.

As T is an S-tree and T_1, T_2 are B-trees, we get

$S \overset{*}{\Rightarrow} uBy, B \overset{*}{\Rightarrow} vBx$ and

$B \overset{*}{\Rightarrow} w, As\ S \overset{*}{\Rightarrow} uBy \Rightarrow uwy, uv^0wx^0y \in L$.

For $k \geq 1, S \overset{*}{\Rightarrow} uBy \overset{*}{\Rightarrow} uv^kBx^ky \overset{*}{\Rightarrow} uv^kwx^ky \in L$. This proves the point (iv) of the theorem.

Corollary : Let L be a context-free language and n be the natural number obtained by using the pumping lemma. Then (i) $L \neq \phi$ if and only if there exists $w \in L$ with $|w| < n$ and (ii) L is infinite if and only if there exists $z \in L$ such that $n \leq |z| < 2n$

Proof : (i) We have to prove the 'only if' part. If $z \in L$ with $|z| \geq n$, we apply the pumping lemma to write $z = uvwxy$, where $1 \leq |vx| \leq n$. Also $uwy \in L$ and $|uwy| < |z|$. Applying the pumping lemma repeatedly, we can get $z' \in L$ such that $|z'| < n$. Thus (i) is proved.

(ii) If $z \in L$ such that $n \leq |z| < 2n$, by pumping lemma we can write $z = uvwxy$. Also, $uv^kwx^ky \in L$ for all $k \geq 0$. Thus we get an infinite number of elements in L. Conversely, if L is infinite, we can find $z \in L$ with $|z| \geq n$. If $|z| < 2n$, there is nothing to prove. Otherwise, we can apply the pumping lemma to write $z = uvwxy$ and get $uwy \in L$. Every

time we apply the pumping lemma we get a smaller string and the decrease in length is at most n (being equal to $|vx|$). So, we ultimately get a string z' in L such that $n \leq |z'| < 2n$. This proves (ii).

Note : As the proof of the corollary depends only on the length of vx, we can apply the corollary to regular sets as well (refer to pumping lemma for regular sets).

The corollary given provides us algorithms to test whether a given context-free language is empty or infinite. But these algorithms are not efficient.

We use the pumping lemma to show that a language L is not a context-free language. We assume that L is context-free. By applying the pumping lemma we get a contradiction.

The procedure can be carried out by using the following steps :

Step 1 : Assume L is context-free. Let n be the natural number obtained by using the pumping lemma.

Step 2 : Choose $z \in L$ so that $|z| \geq n$. Write $z = uvwxy$ using the pumping lemma.

Step 3 : Find a suitable k so that $uv^k wx^k y \notin L$. This is a contradiction, and so L is not context-free.

PART-C

Q.16 How can a pushdown automata be constructed for a given language? Explain with example.

[R.T.U. 2019, 13]

Ans. As we know that the sets accepted by PDA (by null store or final state) are precisely the context free languages.

So, if L is a context-free language.

Then we can construct a PDA A accepting L by empty store, *i.e.* L = N (A).

We construct A by making use of productions in G.

Let $L = L(G)$

Where $G = (V_N, \Sigma, P, S)$ is a CFG (Context Free Grammar) we construct a PDA A as

$$A((q), \Sigma, V_N \cup \Sigma, \delta, q, s, \phi)$$

Where, δ is defined by the following rules:

R1 : $\delta(q_1, \wedge, A) = \{(q, \alpha)\} | A \to \alpha$ is in P}

R2 : $\delta(q, a, a) = \{(q, \wedge)\}$ for every a in Σ

Explanation

- The pushdown symbols in A are variables and terminals.
- If the PDA reads a variable A on the top of PDS, it makes a $\wedge$ - move by placing the R.H.S. of any A - production (after erasing A).
- If the PDA reads a terminal a on PDS and if it matches with the current input symbol then the PDA erases a.
- In other cases PDA halts.

Example

If $W \in L(G)$ is obtained by a leftmost derivation

$$\delta = u_1A_1\alpha_1 = u_1u_2A_2\alpha_2\alpha_1 = \ldots = w$$

Then

- A can empty the PDS on application of input string was follows:
- The first move of is by a $\wedge$- move corresponding to $\delta \to \mu_1A_1\alpha_1$
- The PDA erases δ and stores $\mu_1A_1\alpha_1$ then using R_2
- The PDA erases the symbols in u_1 by processing a perfix of w

Now, The top most symbols in PDS is A_1 By applying the null move corresponding to $A_1 \to \mu_2A_2\alpha_2$,

- The PDA erases A_2 and stores $\mu_2A_2\alpha_2$ above α_1. *i.e.*

Proceeding in this way, the PDA emties the PDS by processing the entire string w.

Q.17 What is push down automata? Design a push down automata for language.

$$L = \{a^n b^n : n \geq 1\},$$

also check the acceptability of string "a aa b bb a b".

[R.T.U. 2014]

Ans. A finite automaton cannot accept L, i.e. strings of the form a^nb^n as it has to remember the number of a's in a string and so it will require infinite number of states. This difficulty can be avoided by adding an auxiliary memory in the form of a 'stack' (In a stack we add the elements in a linear way. While removing the elements we follow the last in first out (LIFO) basis, i.e. the most recently added element is removed first). The a's in the given string are added to the stack. When the symbol b is encountered in the input string an a is removed from the stack. Thus the matching of number of a's and number of b's is accomplished. This type of arrangement where a finite automaton has a stack leads to the generation of a pushdown automaton.

Before giving the rigorous definition, let us consider the components of a pushdown automaton and the way it operates. It has a read only input tape, input alphabet, finite state control, a set of final states, and an initial state as in the case of an FA. In addition to these, it has a stack callec the pushdown store (abreviated as PDA). It is a read write pushdown store as we add elements to PDA or remove elements from PDA. A finite automaton is in some state and on reading, an input symbol moves to a new state. The pushdown automaton is also in some state and on reading an input symbol and the topmost symbol in PDA, moves to a new state and writes (adds) a string of symbols in PDA. Fig. illustrates the pushdown automaton.

Pushdown Automation : *Refer to Q.4.*

Q.18 ***Construct a PDA accepting*** $\{a^n b^m a^n \mid m,n \geq 1\}$ ***by null store. Construct the corresponding context free grammar accepting the same set?*** *[R.T.U. 2009]*

Ans. Construction of a PDA accepting $\{a^n b^m a^n \mid m,n \geq 1\}$

The accepting string is $a^n b^m a^n$ where $m,n \geq 1$ which means it consists of a string which initiate with 'n' number of a's then followed by m number of b's and then end with again n number of a's.

Examples of the string accepted by the required PDA.

$\underset{n=3\ m=1}{aaa\ b\ aaa}$ $\underset{n=4,\ m=2}{aaaa\ bb\ aaaa}$, $\underset{n=1,\ m=3}{a\ bbb\ a}$ etc.

Examples of the string not accepted by the required PDA

- aaabaa, abbaa, aaaaabbaa etc.

We need to define a model for context free grammar which is to be represented or accepted by the model called Push Down Automaton (PDA).

PDA is the model that accepts/reject strings defined in context free grammar.

Push Down Automata (PDA) is defined as the 7-tuple system.

Let A be the PDA,

$$A = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$$

Where

Q = A non-empty set of all the states

Σ = Non-empty set of terminal symbols

Γ = Tap symbol, elements of pushdown store

δ = Transition rules

q_0 = Initial state

Z_0 = Initial value of pushdown store

F = Final states (set)

Some Other Definitions

Input Tape : A tape consisting of the string to be checked whether it is acceptable by the model or not.

Finite State Control : Unit holding the current state

Push Down Store : A stack like structure with LIFO structure. Used to hold the tab symbols.

Now, we define the PDA for the string $\{a^n b^m a^n\}$ where $m,n \geq 1$

To define a PDA, we have to define all the 7 tuples :

$Q = \{q_0, q_1, q_2, q_f\}$ //Set of all states

$\Sigma = \{a, b\}$ //Set of terminal symbols

$\Gamma = \{a, z_0\}$

δ = To be defined.

$q_0 = \{q_0\}$ //Initial state

$z_0 = \{z_0\}$ //Initial value of PDS

$F = \{q_f\}$ //Final state.

Only transition rules are yet to be defined which are defined using logical sequences :

(1) Let q_0 be initial state.

Our approach would be to store all the initial *a*'s to PDS and then retrieve them when traversing the ending *a*'s.

If any sequence occurs when, we do not find any a's in PDS, we come to halt condition.

String is accepted only when at the end of the string, PDS is empty.

(2) Starting with q_0, if string element contains *a*, then we push it onto PDS.

$$\delta(q_0, a, z_0) = (q_0, az_0)$$

$$\delta(q_0, a, a) = (q_0, aa)$$

(3) When 'b' is encountered, it switches state to q, and continue without PUSH or POP.

$$\delta(q_0, b, a) = (q_1, a)$$

(4) The state will continue unless 'a' is encountered.

$$\delta(q_1, b, a) = (q, a)$$

(5) When a is reached, it goes to state 'q_2' and pops an element

$$\delta(q_1, a, a) = (q_2, \wedge)$$

(6) Now, it continuously 'pop' at state q_2, for every occurrence of 'a'

$$\delta(q_2,a,a)=(q_2,\wedge)$$

(7)Accepting condition would be when at the end of the string, PDS becomes empty

$$\delta(q_2,\wedge,z_0)=(q_2,\wedge)$$

Hence PDA is constructed

$$A=(\{q_1,q_2\},\{a,b\},\{a,z_0\},\delta,\{q_0\},\{z_0\},\{\})$$

δ is defined by

$$\delta(q_0,a,z_0)=(q_0,az_0)$$

$$\delta(q_0,a,a)=(q_0,aa)$$

$$\delta(q_0,b,a)=(q_1,a)$$

$$\delta(q_1,b,a)=(q_1,a)$$

$$\delta(q_1,a,a)=(q_2,\wedge)$$

$$\delta(q_2,a,a)=(q_2,\wedge)$$

$$\delta(q_2,\wedge,z_0)=(q_2,\wedge)$$

Conversion of PDA to CFG

Context free grammer (CFG) is a 4-tuple system.

$$G=(V_N,\Sigma,P,S)$$

V_N = Set of terminal or non-terminal symbols

Σ = Set of terminals

P = Production rules

S = Initial state

Rules of construction of CFG from PDA

V_N = {Set of all combinations of $[q,z,q']$ where $q,q'\in Q$ and $\{z\in\Gamma\}\cup\{S\}$

$$=S[q_0,a,q_0],[q_0,a,q_1],[q_0,a,q_2],[q_0,z_0,q_0],[q_0,z_0,q_1]$$

$$[q_0,z_0,q_2],[q_1,a,q_0],[q_1,a,q_1],[q_1,a,q_2],[q_1,z_0,q_0]$$

$$[q_1,z_0,q_1],[q_1,z_0,q_2],[q_2,a,q_0],[q_2,a,q_1],[q_2,a,q_2]$$

$$[q_2,z_0,q_0],[q_2,z_0,q_1],[q_2,z_0,q_2]\}$$

Σ = Same as the PDA's

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

Rules for P

Rule No. 1 : $S\to[q_0,z_0,q]$ where $q\in Q$

Rule No. 2 : For $\delta(q,a,z)=(q',\wedge)$

$$[q,z,q']\to a$$

Rule No. 3 : For $\delta(q_0,a,z)=(q_1,z_1,z_2,z_3....z_m)$

$$[q_0,z,q]\to a[q_1,z_1,q_2][q_2,z_2,q_3]....[q_{m-1},z_{m-1},q_m]$$

We derive, now from all δ in PDA

$\Rightarrow$ $\delta(q_0,a,z_0)=(q_0,az_0)$

We apply Rule No. 3

P_1 $[q_0,z_0,q_0]\to a[q_0,a,q_0][q_0,z_0,q_0]$

P_2 $[q_0,z_0,q_0]\to a[q_0,a,q_1][q_1,z_0,q_0]$

P_3 $[q_0,z_0,q_0]\to a[q_0,a,q_2][q_2,z_0,q_0]$

P_4 $[q_0,z_0,q_1]\to a[q_0,a,q_0][q_0,z_0,q_1]$

P_5 $[q_0,z_0,q_1]\to a[q_0,a,q_1][q_1,z_0,q_1]$

P_6 $[q_0,z_0,q_1]\to a[q_0,a,q_2][q_2,z_0,q_2]$

P_7 $[q_0,z_0,q_2]\to a[q_0,a,q_0][q_0,z_0,q_2]$

P_8 $[q_0,z_0,q_2]\to a[q_0,a,q_1][q_1,z_0,q_2]$

P_9 $[q_0,z_0,q_3]\to a[q_0,a,q_2][q_2,z_0,q_2]$

$\Rightarrow$ $\delta(q_0,a,a)=(q_0,aa)$ (Rule No. 3)

P_{10} $[q_0,a,q_0]\to a[q_0,a,q_0][q_0,a,q_0]$

P_{11} $[q_0,a,q_0]\to a[q_0,a,q_0][q_0,a,q_0]$

P_{12} $[q_0,a,q_0]\to a[q_0,a,q_2][q_2,a,q_0]$

P_{13} $[q_0,a,q_1]\to a[q_0,a,q_0][q_0,a,q_1]$

P_{14} $[q_0,a,q_1]\to a[q_0,a,q_1][q_1,a,q_1]$

P_{15} $[q_0,a,q_1]\to a[q_0,a,q_2][q_2,a,q_1]$

P_{16} $[q_0,a,q_2]\to a[q_0,a,q_0][q_0,a,q_2]$

P_{17} $[q_0,a,q_2]\to a[q_0,a,q_1][q_1,a,q_2]$

P_{18} $[q_0,a,q_2]\to a[q_0,a,q_2][q_2,a,q_2]$

$\Rightarrow$ $\delta(q_0,b,a)=(q_1,a)$ (Rule No. 3)

P_{19} $[q_0,a,q_0]\to b[q_1,a,q_0]$

P_{20} $[q_0,a,q_1]\to b[q_1,a,q_1]$

P_{21} $[q_0,a,q_2]\to b[q_1,a,q_2]$

$\Rightarrow\delta(q_1b,a)=(q_1,a)$ (Rule No. 3)

P_{22} $[q_1,a,q_0]\to b[q_1,a,q_0]$

P_{23} $[q_1,a,q_1]\to b[q_1,a,q_1]$

P_{24} $[q_1,a,q_2]\to b[q_1,a,q_2]$

$\Rightarrow$ $\delta(q_1,a,a)=(q_2,\Lambda)$ (Rule No. 2)

P_{25} $[q_1,a,q_2]\to a$

$\Rightarrow$ $\delta(q_2, a, a) = (q_2, \Lambda)$

P_{26} $[q_2, a, q_2] \rightarrow a$

$\Rightarrow$ $\delta(q_2, \Lambda, z_0) = (q_2, \Lambda)$

P_{27} $[q_2, z_0, q_2] \rightarrow \Lambda$

From Rule 1,

P_{28} $S \rightarrow [q_0, z_0, q_0]$

P_{29} $S \rightarrow [q_0, z_0, q_1]$

P_{30} $S \rightarrow [q_0, z_0, q_2]$

Hence, the CFG is defined with all tuples defined.

Q.19 *Design a PDA to accept the language :*

$$L = \{ w c w^T \mid w \in (a,b)^* \}$$

[Raj.Univ. 2007]

Ans. Consider the given PDA :

$$A = (\{q_0, q_1, q_f\}, \{a,b,c\}, \{a,b,z_0\}, \delta, q_0, z_0, \{q_f\})$$

Where δ is defined as :

$\delta(q_0, a, z_0) = \{(q_0, a z_0)\}$, $\delta(q_0, b, z_0) = \{(q_0, b z_0)\}$

$\delta(q_0, a, a) = \{(q_0, aa)\}$, $\delta(q_0, b, a) = \{(q_0, ba)\}$

$\delta(q_0, a, b) = \{(q_0, ab)\}$, $\delta(q_0, b, b) = \{(q_0, bb)\}$

$\delta(q_0, c, a) = \{(q_1, a)\}$, $\delta(q_0, c, b) = \{(q_1, b)\}$

$\delta(q_0, c, z_0) = \{(q_1, z_0)\}$

$\delta(q_1, a, a) = \delta(q_1, b, b) = \{(q_1, \wedge)\}$

$\delta(q_1, \wedge, z_0) = \{(q_f, z_o)\}$

Let $wcw^T \in L$, write $w = a_1 a_2 .... a_n$, where each a_i is either a or b. Then, we have :

$$(q_0,\ a_1 a_2 ... a_n\, cw^T, z_0)$$

$$\vdash^* (q_0, cw^T, a_n\, a_{n-1} ... a_1\, z_0)$$

$$\vdash^* (q_1, a_n\, a_{n-1} ... a_1, a_n\, a_{n-1} ... a_1\, z_0)$$

$$\vdash^* (q_1, \wedge, z_0)$$

$$\vdash (q_f, \wedge, z_0)$$

Therefore, $wcw^T \in T(A)$, i.e. $L \subseteq T(A)$

To prove the reverse inclusion, it is enough to show that $L^c \subseteq T(A)^c$

Let $x \in L^c$.

Case 1 : x does not have the symbol C. In this case the PDA never makes a transition to q_1. So the PDA cannot make a transition to q_f as we cannot apply rule. Thus, $x \in T(A)^c$.

Case 2 : $x = w_1 c w_2$

$w_2 \neq w_1^T$

$(q_0, w_1 c w_2, z_0)$

$\vdash^* (q_0, cw_2, w_1^T z_0)$

$\vdash (q_1, w_2, w_1^T z_0)$

As $w_2 \neq w_1^T$, the PDA cannot reach an ID of the form $(q_1, \wedge, z_0)$. So we cannot apply rule.

Therefore, $x \in T(A)^c$.

Thus we have proved $L^c \subseteq T(A)^c$

Second type of acceptance

Let $A = (Q, \Sigma, \tau, \delta, q_0, z_0, F)$ be a PDA. The set N(A) accepted by null store (or empty store) is defined by $N(A) = \{w \in \Sigma^* \mid q_0, w, z_0 \vdash^* (q, \wedge, \wedge)$ for some $q \in Q\}$

In other words, w is in N(A). If A is in initial ID (q_0, w, z_0) and empties the PDS after processing all the symbols of w. So in defining N(A), we consider the change brought about an PDS by application of w, and not the transition of states.

❑❑❑

TURING MACHINES 4

PREVIOUS YEARS QUESTIONS

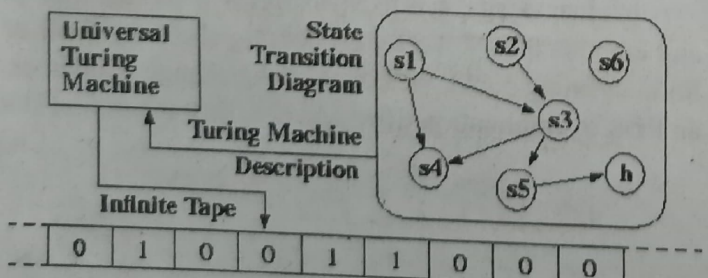
PART-A

Q.1 ***What is Universal Turing Machine?*** ***[R.T.U. 2019]***

Ans. Universal Turing Machine

A Turing Machine is the mathematical tool equivalent to a digital computer. It was suggested by the mathematician Turing in the 30s, and has been since then the most widely used model of computation in computability and complexity theory. The model consists of an input output relation that the machine computes. The input is given in binary form on the machine's tape, and the output consists of the contents of the tape when the machine halts.

What determines how the contents of the tape change is a finite state machine (or FSM, also called a finite automaton) inside the turing machine. The FSM is determined by the number of states it has, and the transitions between them.



At every step, the current state and the character read on the tape determine the next state the FSM will be in, the character that the machine will output on the tape (possibly the one read, leaving the contents unchanged), and which direction the head moves in, left or right.

The problem with turing machines is that a different one must be constructed for every new computation to be performed, for every input output relation.

This is why we introduce the notion of a universal turing machine (UTM), which along with the input on the tape, takes in the description of a machine M. The UTM can go on then to simulate M on the rest of the contents of the input tape. A universal turing machine can thus simulate any other machine.

Q.2 ***What is Turing Machine?*** ***[R.T.U. 2019]***

OR

Explain turing machine in brief.

Ans. Turing Machine : The Turing Machine (TM) is a simple mathematical model of a general purpose computer. In other words, Turing machine model is the computing power of a computer *i.e.* the Turing machine is capable of performing any calculation which can be performed by any computing machine.

The Turing machine can be thought of as a finite state automaton connected to R/W (read / write) head. It has one tape which is divided into a number of cells.

Q.3 ***Explain the properties of context – sensitive languages*** ***[R.T.U. 2014]***

Ans. Context Sensitive Language

• Context sensitive languages are also called type 1 grammar.

• Context sensitive languages have both left and right context.

• The production is of the form :

$$\phi\alpha\psi \rightarrow \phi\beta\psi$$

Here $\phi \rightarrow$ left context

$\psi \rightarrow$ right context

and $\alpha \rightarrow \beta$ is the producton

E.g. : $aAbcd \rightarrow abcdbcd$

Here, $A \rightarrow bcd$ is the production

and $a \rightarrow$ left context

$bcd \rightarrow$ right context

In this $S \rightarrow \Lambda$ is not allowed and S does not appear on the right hand side (RHS) of the any production.

The Turing machine is used to recognize context sensitive language : Turing machine is a 7 tuple machine.

$M = (Q, \Sigma, \Gamma, \delta, q_0, \Delta \text{ or } b, F)$

Here

Q = Finite non empty set of states

Σ = Finite non empty set of input symbols

Γ = External symbol

$\delta \rightarrow$ Transition function which maps $\Sigma \times Q$ into Q

q_0 = Initial state

Δ or b or B = Blank symbol

F = Final state.

$\Gamma \rightarrow$ Maps $\Sigma \times Q$ into Δ.

Q.4 ***Prove the transpose closure property of context sensitive language.*** *[R.T.U. 2013]*

Ans. Transpose : Transpose is obtained by reversing the sequence of symbols in a string. The transpose of s is s^T.

Let $s = pqrs$

$s^T = srqp$

It should be noted that when we reversing the sequence of symbols then new string may or may not be a valid string.

Suppose $L = \{\text{set of five members divisible by 4}\}$

$= \{4, 8, 12, 16, 20\}$

Here $W = 12$ is a word in the language (L) then,

Transpose $(W^T) = 21$, which is not in language.

Q.5 ***Define indexed language in brief.***

Ans. Indexed languages are a class of formal languages. They are described by indexed grammars and can be recognized by nested stack automata.

Indexed languages are a proper subset of context-sensitive languages. They qualify as an abstract family of languages and hence satisfy many closure properties. However, they are not closed under intersection or complement.

Q.6 ***Define union in brief.***

Ans. Union : In the theory of computation, the union (denoted as $\cup$) of a collection of sets is the set of all distinct elements in the collection. The union of a collection of sets gives a set. The union of two sets A and B is the collection of points which are in A or in B (or in both). A simple example :

$A = \{1, 2, 3, 4\}$

$B = \{5, 6, 7, 8\}$

PART-B

Q.7 ***Explain the difference between Deterministic and Non Deterministic Pushdown Automata.*** *[R.T.U. 2019]*

Ans. In general pushdown automata may have several computations on a given input string, some of which may be halting in accepting configurations while others are not. Thus we have a model which is technically known as a "nondeterministic pushdown automaton" (NPDA). No determinism means that there may be more that just one transition available to follow, given an input signal, state and stack symbol. If in every situation only one transition is available as continuation of the computation, then the result is a **deterministic Pushdown Automaton** (DPDA), a strictly weaker device.

Q.8 ***Design a Turing machine over {1, b} which can compute a concatenation function over Σ = {1}.***

If a pair of words (w1, w2) is the input, the output has to be w1 w2. *[R.T.U. 2016]*

Ans. Suppose w_1 is given as a string $w_1 = 11$ and $w_2 = 111$ and the two words are written in the input tape separated by a blank as

B	1	1	B	1	1	1	B

Input tape

The output tape should be

B	1	1	1	1	1	B

Output tape

When the blank symbol is found, it must be replaced by 1 and when the rightmost 1 is found, it is replaced by B. The tape head return to the starting position.

The machine M can be defined as :

$M = (\{q_0, q_1, q_2, q_3, q_f\}, \{1\}, \{1, B\}, \delta, q_0, B, \{q_f\})$

The moves of the Turing machine are shown below in the form of transition table :

Table : Transition Table

Present State	1	B
$\rightarrow q_0$	$(q_0, 1, R)$	$(q_1, 1, R)$
q_1	$(q_1, 1, R)$	(q_2, B, L)
q_2	(q_3, B, L)	–
q_3	$(q_3, 1, L)$	(q_f, B, R)
$\boxed{q_f}$	–	–

The transition diagram corresponding to the above transition table is shown in following figure.

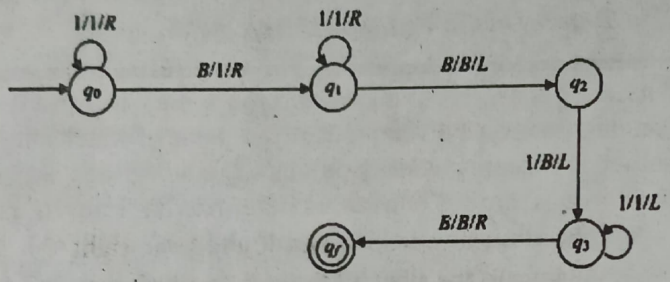


Fig. : Representation of Turing Machine of Transition Table

Q.9 ***Consider the TM description below. Draw the computation sequence of the input string 00.***

Present State	*Tape Symbol*		
	b	*0*	*1*
$\rightarrow q_1$	$1Lq_2$	$0Rq_1$	
q_2	bRq_3	$0Lq_2$	$1Lq_2$
q_3		bRq_4	bRq_5
q_4	$0Rq_5$	$0Rq_4$	$1Rq_4$
$\boxed{q_5}$	$0Lq_2$		

[R.T.U. 2016]

Ans.

Present State	Tape Symbol :: b	Tape Symbol :: 0	Tape Symbol :: 1
q_1	$1Lq_2$	$0Rq_1$	–
q_2	bRq_3	$0Lq_2$	$1Lq_3$
q_3	–	bRq_4	bRq_5
q_4	$0Rq_5$	$0Rq_4$	$1Rq_4$
q_5	$0Lq_2$	–	–

Such a computation sequence can be shown in terms of the contents of the tape and the current state. So, we get following sequence.

q_1 00b $\vdash$ 0q_1 0b $\vdash$ 00 q_1 b $\vdash$ 0 q_2 01 $\vdash$ q_2 001

$\vdash$ q_2 b 001 $\vdash$ b q_3 001 $\vdash$ bb q_4 01 $\vdash$ bb 0 q_4 1 $\vdash$ bb 01 q_4 b

$\vdash$ bb010 q_5 $\vdash$ bb 01 q_2 00 $\vdash$ bb 0 q_2 100 $\vdash$ bb q_2 0100

$\vdash$ b q_2 b 0100 $\vdash$ bb q_3 0100 $\vdash$ bbb q_4 100 $\vdash$ bbb 1 q_4 00

$\vdash$ bbb 10 q_4 0 $\vdash$ bbb 100 q_4 b $\vdash$ bbb 1000 q_5 b

$\vdash$ bbb 100 q_2 00 $\vdash$ bbb 10 q_2 000 $\vdash$ bbb 1 q_2 0000

$\vdash$ bbb q_2 10000 $\vdash$ bb q_2 b 10000 $\vdash$ bbb q_3 10000 $\vdash$ bbbb q_5 0000

Q.10 ***Explain how a Turing machine with multiple tracks of the tape can be used to determine the given number is prime or not.*** *[R.T.U. 2015]*

Ans. Multi Tape Turing machine to determine whether the given number is prime or not : We want to construct a total TM that accepts its input string if the length of the string is prime. We will give a TM implementation of the sieve of Eratosthenes, which can be described informally as follows. We write down all the numbers from 2 to n in order, then repeat the following:

(i) Find the smallest number in the list.

(ii) Declare it prime.

(iii) Then cross off all multiples of that number.

Repeat until each number in the list has been either declared prime or crossed off as a multiple of a smaller prime.

For example, to check whether 23 is prime, we would start with all the numbers from 2 to 23:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23

In the first pass, we cross off multiples of 2:

~~2~~ 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23

The smallest number remaining is 3, and this is prime. In the second pass, we cross off multiples of 3:

~~2~~ ~~3~~ ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23

Then 5 is the next prime, so we cross off multiples of 5; and so forth. Since, 23 is prime, it will never be crossed off as a multiple of anything smaller, and eventually we will discover that fact when everything smaller has been crossed off.

Now, we show how to implement this on a TM. Suppose we have a written on the tape. We illustrate the algorithm with p = 23.

$\vdash$ a a a a a a a a a a a a a a a a a a a a a ␣ ␣ ␣ ...

If p = 0 or p = 1, reject. We can determine this by looking at the first three cells of the tape. Otherwise, there are at least two a's. Erase the first a, scan right to the end of the input, and replace the last a in the input string with the symbol $. We now have an a in positions 2, 3, 4, ... p – 1 and $ at position p.

⊢ ␣ a a a a a a a a a a a a a a a a a a $ ␣ ␣ ␣ ...

Now we repeat the following loop. Starting from the left endmarker ⊢ scan right and find the first non blank symbol, say occurring at position m. Then m is prime (this is an invariant of the loop). If this symbol is the $, we are done: p = m is prime, so we halt and accept. Otherwise, the symbol is an a. Mark it with a ^ and everything between there and the left endmarker with.

⊢ ␣ â a a a a a a a a a a a a a a a a a a $ ␣ ␣ ␣ ...

We will now enter an inner loop to erase all the symbols occurring at positions that are multiples of m.

First, erase the a under the ^ (Formally, just write the symbol ␣̂ .)

⊢ ␣ ␣̂ a a a a a a a a a a a a a a a a a a $ ␣ ␣ ␣ ...

Shift the marks to the right one at a time a distance equal to the number of marks. This can be done by shuttling back and forth, erasing marks on the left and writing them on the right. We know when we are done because the ^ is the last mark moved.

⊢ ␣ ␣ ā â a a a a a a a a a a a a a a a a a a $ ␣ ␣ ␣ ...

When this is done, erase the symbol under the ^.This is the symbol occurring at position 2m.

⊢ ␣ ␣ ā ␣̂ a a a a a a a a a a a a a a a a a a $ ␣ ␣ ␣ ...

Keep shifting the marks and erasing the symbol under the ^ in this fashion until we reach the end.

⊢ ␣ ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ $ ␣̂ ␣ ␣ ...

If we find ourselves at the end of the string wanting to erase the $, reject-p is a multiple of m but not equal to m. Otherwise, go back to the left and repeat. Find the first nonblank symbol and mark it and everything to its left.

⊢ ␣ ␣ â ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ a ␣ $ ␣ ␣ ␣ ...

Alternately erase the symbol under the ^ and shift the marks until we reach the end of the string.

⊢ ␣ ␣ ␣ a ␣ a ␣ ␣ ␣ a ␣ a ␣ ␣ ␣ a ␣ a ␣ ␣ ␣ a ␣ a ␣ ␣ ␣ $̂ ␣ ␣ ␣ ...

Go back to the left and repeat.

⊢ ␣ ␣ ␣ ␣ â ␣ a ␣ ␣ ␣ a ␣ a ␣ ␣ ␣ ␣ a ␣ a ␣ ␣ ␣ ␣ $ ␣ ␣ ␣ ...

If we ever try to erase the $, reject-p is not prime. If we manage to erase all the a's, accept.

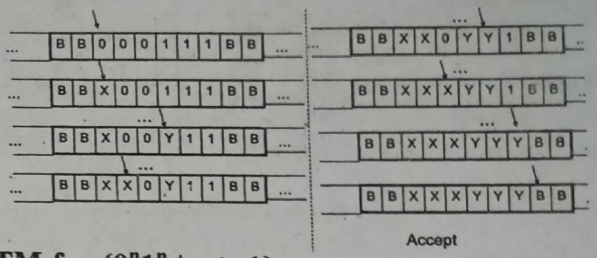
Q.11 ***Construct a Turing machine for the language { } 1\|01 ≥ =nLnn***

[Note : The expression { } 1 \| 01 ≥= nLnn Consider as $0^n1^n \mid n \geq 1$*]*

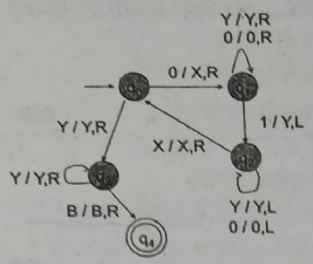
[R.T.U. 2015, 2012, 2010 Raj. Univ. 2007]

Ans. $L = \{0^n1^n \mid n \geq 1\}$

- Strategy: w = 000111



TM for $\{0^n1^n \mid n \geq 1\}$



1. Mark next unread 0 with X and move right.
2. Move to the right all the way to the first unread 1, and mark it with Y.
3. Move back (to the left) all the way to the last marked X, and then move one position to the right.
4. If the next position is 0, then go to step 1.

Else move all the way to the right to ensure there are no excess 1s. If not move right to the next blank symbol and stop & accept.

TM for $\{0^n1^n \mid n \geq 1\}$

	Next Tape Symbol				
Curr. State	0	1	X	Y	B
→ q_0	(q_1,X,R)	-	-	(q_3,Y,R)	-
q_1	$(q_1,0,R)$	(q_2,Y,L)	-	(q_1,Y,R)	-
q_2	$(q_2,0,L)$	-	(q_0,X,R)	(q_2,Y,L)	-
q_3	-	-	-	(q_3,Y,R)	(q_4,B,R)
*q_4	-	–	-	-	-

Table representation of the state diagram

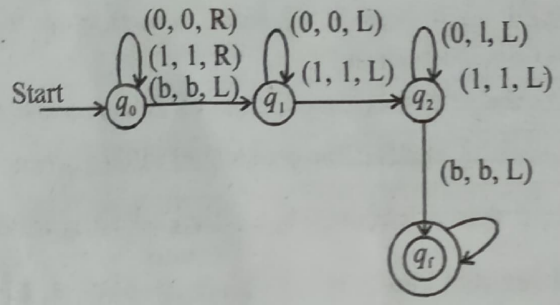
Q.12 ***(a) Design a Turing Machine that computes 2's complement of the given string over the*** $\sum = \{0,1\}$ ***. Also show the output of the machine for string "00000".***

[R.T.U. 2014, 2011]

(b) Write a short note on Universal Turing Machine in detail. *[R.T.U. 2014, 2013, 2012, 2011, Raj. Univ. 2007, 2003]*

Ans. (a) The two's complement of any binary number can be obtained by reading the binary number from right to left, keeping all zero's as it is. As we get first 1 from right, this 1

is also kept unchanged and there after, we can use it all 0's by 1's and by 0's from right to left.



For example: 2's complement of binary string '00000' can be obtained as follows:

00000: Number itself
11111: 1's complement
+1 : Add 1
00000: 2's Complement

By our idea we get the same result.

b00000b Move the right most end upto b.
↑

b00000b Move left
↑

b00000b Move left
↑ It is 0, so keep it as it is and move left

b00000b It is 0, so keep it as it is and move left
↑

b00000b It is 0, so keep it as it is and move left
↑

b00000b It is 0, so keep it as it is and move left
↑

b00000b It is 0, so keep it as it is and move left
↑

b00000b since 'b' is readed, stop!
↑

Thus two way infinite tape beings significant power to turing machine.

Ans. (b) *Refer to Q.1.*

Q.13 ***Design a Turing Machine M to recognize the language $\{1^n 2^n 3^n / n > 1\}$.[R.T.U. 2013, Raj. Univ. 2002]***

Ans. Before designing the required Turing machine M, let us evolve a procedure for processing the input string 111222333. After processing we require the output tape containing ␣␣␣␣␣␣␣␣␣ q_1. The processing is done by using five steps.

Step1 : q_1 is the initial state. The R/W head scans the leftmost 1, replaces 1 by and move to the right M enter q_2.

Step 2 : On scanning the leftmost 2, the R/W head replaces 2 by LJ and moves to the right. M enters q_3.

Step 3 : On scanning the left most 3, the R/W head replaces 3 by and moves to the right. M enters q_4.

Step 4 : After scanning the rightmost 3, the R/W heads moves to the left until it finds the left most 1. As a result, the left most 1, 2, 3 are replaced by b.

Step 5 : Steps 1 - 4 are repeated until all 1's, 2's and 3's are replace by blanks.

Thus we can construct as-

Present state	Input tape symbol 1	2	3	␣
$\rightarrow q_1$	$\sqcup q_2 R$	-	-	$\sqcup q_1 R$
q_2	$1 q_2 R$	$\sqcup q_3 R$	-	$\sqcup q_2 R$
q_3	-	$2 q_3 R$	$\sqcup q_4 R$	$\sqcup q_3 R$
q_4	-	0	$3 q_5 L$	$\sqcup q_7 L$
q_5	$1 q_6 L$	$2 q_5 L$	-	$\sqcup q_5 L$
q_6	$1 q_6 L$	-	-	$\sqcup q_1 R$
q_7	-	-	-	-

This is the required solution.

Q.14 ***Write short note on recursive and recursively enumerable language.***
[R.T.U. 2013, 2012, 2011, Raj. Univ. 2007, 06, 04, 03]

Ans. Recursive and Recursively Enumerable Language : In mathematics, logic and computer science, a recursively enumerable language is a type of formal language which is also called partially decidable or Turing-acceptable. It is known as a type-0 language in the Chomsky hierarchy of formal languages. The class of all recursively enumerable language is called RE.

Definitions

There exist three equivalent major definitions for the concept of a recursively enumerable language.

1. A recursively enumerable formal language is a recursively enumerable subset in the set of all possible words over the alphabet of the language.
2. A recursively enumerable language is a formal language for which there exists a Turing machine (or other computable function) which will enumerate all valid strings of the language. Note that, if the language is infinite, the enumerating algorithm provided can be chosen so that it avoids repetitions, since we can test whether the string produced for number n is "already" produced for a number which is less than n. If it is already produced, use the output for input n+1 instead (recursively), but again, test whether it is "new".

A recursively enumerable language is a formal language for which there exists a Turing machine (or other computable function) that will halt and accept when presented with any string in the language as input but may either halt and reject or loop forever when presented with a string not in the

language. Contrast this to recursive language, which require that the Turing machine halts in all cases.

Closure Properties

Recursively enumerable languages are closed under the following operations. That is, if L and P are two recursively enumerable language, then the following languages are recursively enumerable as well :

- The Kleene star L* of L.
- The concatenation of L and P.
- The union.
- The intersection.

Note that recursively enumerable languages are not closed under set difference or complementation. The set difference L-P may or may not be recursively enumerable. If L is recursively enumerable, then the complement of L is recursively enumerable if and only if L is also recursive. There are three possible outcomes of executing a Turing machine over a given input. The Turing machine may

- Halt and accept the input
- Halt and reject the input or
- Never halt.

A language is recursive if there exists a Turing machine that accepts every string of the language and rejects every string (over the same alphabet) that is not in the language.

Q.15 *(a) Write short note on :*

(i) Halting problem

(ii) Multitape and Multi dimensional turing machine *[R.T.U. 2012, 2011, Raj. Univ. 2007]*

(b) Design turing machine M that recognize the language $\{a^n b^l c^n / n^3 \geq 1\}$ *[R.T.U. 2012, 2011]*

Ans.(a) (i) Halting Problem : For a given configuration of a TM, two cases arise :

(a) The machine starting at this configuration will halt after a finite number of steps.

(b) The machine starting at this configuration never halts no matter how long it runs. Given any TM, problem of determining whether it halts ever or not is called halting problem.

To solve the halting problem, we should have some mechanism to which given any functional matrix, input data type and initial configuration of the TM for which we want to detect, determine whether the process will ever halt or not. In reality one cannot solve the halting problem. The halting problem is unsolvable. That means, there exist no TM which can determine whether given TM 'T' will ever halt or not.

Reduction Technique is used to prove the undecidability of halting problem of Turing machine. We say that problem A is reducible to problem B if a solution to problem B can be used to solve problem A.

Example : If A is the problem of finding some root of $x^4 - 3x^2 + 2 = 0$ and B is the problem of finding some root of $x^2 - 2 = 0$, then A is reducible to B. As $x^2 - 2$ is a factor of $x^4 - 3x^2 + 2$, a root of $x^2 - 2 = 0$ is also a root of $x^4 - 3x^2 + 2 = 0$.

Note : If A is reducible to B and B is decidable then A is decidable. If A is reducible to B and A is undecidable, then B is undecidable.

Theorem : $HALT_{TM} = |(M, w)|$ The Turing machine M halts on input {w} is undecidable.

Proof : We assume that $HALT_{TM}$ is decidable and get a contradiction. Let M_1 be the TM such that $T(M_1) = HALT_{TM}$ and let M_1 halt eventually on all (M, w). We construct a TM M_2 as follows:

1. For M_2,(M,w) is an input.
2. The TM M_1 acts on (M,w).
3. If M_1 rejects (M,w) then M_2 rejects (M,w).
4. If M_1 accepts (M,w), simulate the TM, M on the input string w until M halts.
5. If M has accepted w, M_2 accepts (M,w); otherwise M_2 rejects (M,w).

When M_1 accepts (M,w) (in step 4), the Turing machine M halts on w. In this case either an accepting state q or a state q' such that $\delta(q',a)$ is undefined till some symbol a in w is reached. In the first case (the first alternative of step 5) m_2 accepts (M,w). In the second case (the second alternative of step 5) M_2 rejects (M,w).

It follows from the definition of M_2 that M_2 halts eventually.

Also $T(M_2) = |(M,w)|$ The Turing machine accepts w} = A_{TM}

This is a contradiction since A_{TM} is undecidable.

(ii) Multitape and Multi Dimensional Turing Machine :

Multi-tape Turing Machines

In practical analysis, various types of multi-tape Turing machines are often used. Multi-tape machines are similar to single-tape machines, but there is some constant k number of independent tapes.

The table has full independent control over all the heads, any of all of which move and print/erase their own tapes. Most models have tapes with left ends, right ends unbounded.

This model intuitively seems much more powerful than the single-tape model, but any multi-tape machine, no matter how large the k, can be simulated by a single-tape machine using only quadratically more computation time. Thus, multi-tape machines cannot calculate any more function than single-tape machines and none of the robust complexity classes (such as polynomial time) are affected by a change between single-tape and multi-tape machines.

A k-tape Turning machine can be described as 6-tuple where

- Q is a finite set of states.
- Γ is a finite set of the tape alphabet
- v is the initial state.
- B is the blank symbol.
- F is the set of final or accepting states.
- δ is a partial function called the transition function, where L is left shift, R is right shift, S is no shift.

A Multidimensional Turing machine has a multidimensional "tape:" For example, a two-dimensional Turing machine would read and write on an infinite plane divided into squares, like a checkerboard. Possible directions that the tape head could move might be labelled {N, E, S, W}. A three-dimensional Turing machine might has possible direction {N, E, S, W, U, D} and so on.

Theorem : Multidimensional Turing machine are equivalent to Standard Turing machines.

Ans. (b) Before desiging the required Turing machine M, let us evolve a procedure for processing the input string aabbcc. The processing is done by using 5 steps :

Step 1 : q_1 is the initial state. The R/W head scans the leftmost 1, replaces 1 by b, and moves to the right M enters q_2.

Step 2 : On scanning the leftmost 2, the R/W head replaces 2 by b and moves to the right M enters q_3.

Step 3: On scanning the leftmost 3, the R/W head replaces 3 by b, and moves to the right, M enters q_4.

Step 4 : After scanning the rightmost 3, the R/W heads moves to the left until it finds the leftmost 1. As a result, the leftmost 1, 2, 3 and replaced by b.

Step 5 : Step 1–4 are repeated until all 1's, 2's and 3's are replaced by blanks. The changes of IDs due to processing of 112233 is given as

$q_1\ 112233 \vdash bq_2\ 12233 \vdash b1q_2\ 2233 \vdash b1b\ q_3\ 22 \vdash b1b2\ q_3\ 33 \vdash b1b2bq_3 \vdash bIb2\ q_5\ b3 \vdash b1bq_5\ 2b3 \vdash b1bq_5\ 2b3 \vdash b1q_5\ b2\ b3 \vdash bq_5 1b2b \vdash q_6 b1b2b3 \vdash bq_1 1b2b3 \vdash bbq_2 b2b3 \vdash bbbq_2\ 2b3 \vdash bbbbq_3 b3 \vdash bbbbbq_3\ 3 \vdash bbbbbb\ q_4\ b \vdash bbbbb\ q_7\ bb$

Thus $q_1\ 112233 \vdash q_7\ bbbbbb$

As q_7 is an accepting state, the input string 112233 is accepted. Now construct the transition table for M.

Present State	Input tape Symbol			
	1	2	3	b
$\longrightarrow q_1$	bRq_2			bRq_1
q_2	$1Rq_2$	bRq_3		bRq_2
q_3		$2Rq_3$	bRq_4	bRq_3
q_4			$3Lq_5$	bLq_7
q_5	$1Lq_6$	$2Lq_5$		bLq_5
q_6	$1Lq_6$			bRq_1
(q_7)				

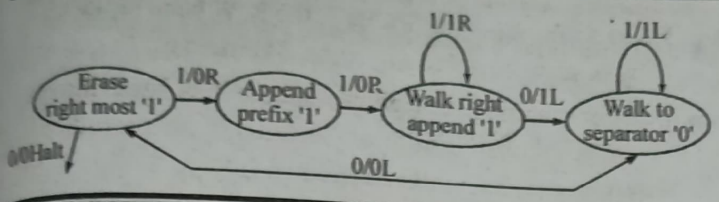
It can be seen from the table that strings other than those of the form $1^n\ 2^n\ 3^n$ are not accepted.

Q.16 ***Design a Turing Machine to compute the following function for 'X' represented in unary. $\Sigma=\{1\}$, $F[x]=\left[\frac{x}{2}\right]$. Explain the output for x = 5.***

[R.T.U. 2011]

Ans.

	0	1	
0	0R7	0R1	Erase a group of 3
1	0R7	0R2	Halt if
2	0R7	0R3	All gone
3	0R4	1R3	Go to end of 2nd block
4	1L5	1R4	
5	0L6	1L5	Return to
6	0R0	1L6	Start 1st block and continue



Q.17 ***Construct a Turing machine over an alphabet {0, 1, #}, where 0 indicates blank, which takes a non null string of 1's and #'s and transfers the right most symbol to the left hand end. Thus .... 000 # 1 # 1 # 1000 becomes ....0001 # 1 # 1 # 000. The Head is initially at the leftmost non blank symbol.***

[R.T.U 2010]

Ans. The function $F : x \times y \rightarrow x$ defined by $F(x, y) = x - y$ is turing computable. The input is encoded as a unary string $x-y$. For example $11111 - 11$ should produce the output 111.

To demonstrate that the above subtraction function is turing computable we construct a Turing machine that computes f(x, y).

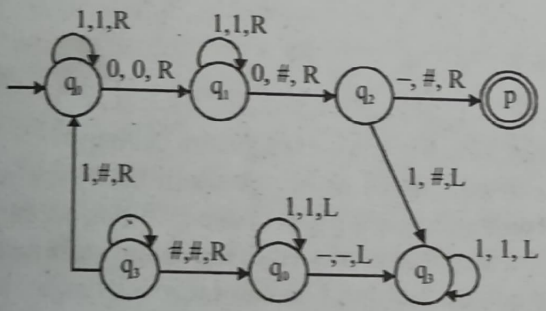


Fig.

Q.18 ***Design a Turing machine to perform the subtraction m – n, where m and n are +ve integers.***

[Raj.Univ. 2007]

Ans. Assume that the input tape has 0^m10^n where $m-n$ is required. We have the following steps:

(a) The leftmost 0 is replaced by b and the R/W head moves to the right.

(b) The R/W head replaces the first 0 after 1 by 1 and moves to the left. On reaching the blank at the left end the cycle is repeated.

(c) Once the 0's to the left of 1's are exhausted, M replaces all 0's and 1's by b's, a - b is the number of 0's left over in the input tape and equal to 0.

(d) Once the 0's to the right of 1's are exhausted, n 0's have been changed to 1's and n + 1 of m 0's have been changed to b. M replaces 1's(there are n + 1 1's) by one 0 and n b's. The number of 0's remaining gives the values of a - b. The transition table is defined below:

Present State	*Input Symbol*		
	0	1	*b*
q_0	bRq_1	bRq_5	
q_1	$0Rq_1$	$1Rq_2$	
q_2	$1Lq_3$	$1Rq_2$	bLq_4
q_3	$0Lq_3$	$1Lq_3$	bRq_0
q_4	$0Lq_4$	bLq_4	$0Rq_6$
q_5	bRq_5	bRq_5	bRq_6
q_6			

Q.19 ***Show that the following functions are primitive recursive :***

(i) Constant function a(X) = a

(ii) Transpose, trans (X) = X^T

(iii) Head function; head $(a_1, a_2, ...a_n) = a_1$

[Raj.Univ. 2007]

Ans. **(i)** Constant function $a(X) = a$

As a(x)= cons a (nil (x)), the function a(x) is the composition of the initial function constant with the initial function nil and is hence primitive recursive.

(ii) Transpose, trans(X) = X^T

trans $(\Lambda) = \Lambda$

trans (ax) = concat (trans(x), a(x))

trans (bx) = concat(trans(x), b(x))

Therefore, trans(x) is primitive recursive.

(iii) Head function; head $(a_1, a_2, a_3 .....a_n) = a_1$

head $(\Lambda) = \Lambda$

head $(ax) = a(x)$

head $(bx) = b(x)$

Therefore, head (x) is primitive recursive.

Q.20 ***What is T(M)? Show that $T(M)^T$ is regular using FAM shown in figure.***

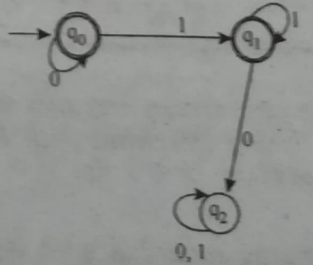


Fig. : Finite Automaton

[Raj.Univ. 2007]

Ans. As the elements of T(M) are given by path values of paths from q_0 to itself or from q_0 to q_1 (note that we have two final states q_0 and q_1), we can construct T(M) by inspection.

As shown in figure, arrows do not come into q_0, the paths from q_0 to itself are self-loops repeated any number of times. The corresponding path values are $0^i, i \geq 1$. As no arrow comes from q_2 to q_0 or q_i, the paths from q_0 to q_1 are of the form $q_0 \ldots\ldots \rightarrow q_0 \ldots q_1 \rightarrow q_1$. The corresponding path values are $0^i 1^j$, where $i \geq 0$ *and* $j \geq 1$. As the initial state q_0 is also a final state, $\Lambda \in T(M)$. Thus,

$$T(M) = \{0^i 1^j \mid i, j \geq 0\}$$

Hence, $T(M)^T = \{1^j 0^i \mid i, j \geq 0\}$

The transition system M' is concerned as follows :

(i) The initial states of M' are q_0 and q_1.

(ii) The (only) final state of M' is q_0.

(iii) The direction of the directed edges is reversed. M' is given in figure.

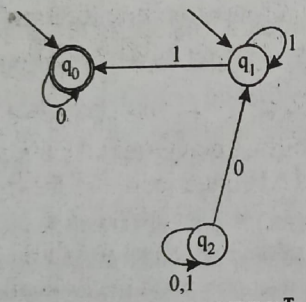


Fig. : Finite Automaton of $T(M)^T$

From (i) - (iii) it follows that

$$T(M') = T(M)^T$$

Hence, $T(M)^T$ is regular.

In above case, we can see by inspection that $T(M') = \{1^j 0^i \mid i, j \geq 0\}$. The strings of $T(M')$ are obtained as path values of paths from q_0 to itself or from q_1 to q_0.

PART-C

Q.21 ***Explain Turing Machine with its various way of representation.*** ***[R.T.U. 2019]***

OR

Explain turing machine with its various ways of representation. Draw diagram wherever required. ***[R.T.U. 2012]***

Ans. (i)Standard Turing Machine : A turing machine M is defined by

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

Where

Q = Set of internal states

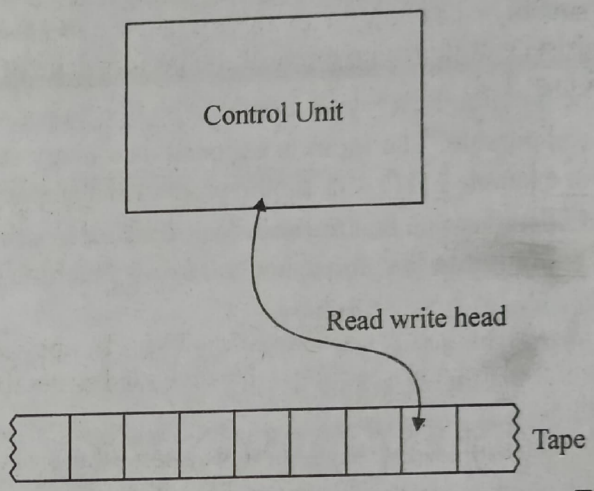
Σ = Input alphabet

Γ = Finite set of symbols

$\square \in F$ a special symbol called the blank

$q_0 \in Q$ is the initial state

$F \subseteq Q$ is the set of final state



(ii) Turing Machine as Language Acceptor : Turing machine can be viewed as acceptors in following sense. A string w is written on the tape, with blanks filling out the unused portion. The machine is started in the initial state q_0 with the read write head positioned on the leftmost symbol of w. If, after a sequence of moves, the Turing Machine enters a final state and halts, then w is considerd to be accepted.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a Turing machine. Then the language accepted by M is

$$L(M) = \{w \in \Sigma^+ : q_0 wt^* x_1 q_f x_2 \text{ for some } q_F \in F, x_1, x_2 \in \Gamma^*\}$$

(iii) Turing Machine as Transducer : Turing Machine are not only interesting as language acceptors, they provide us with a simple abstract model for digital computers in general. Since the primary purpose of a computer is to transform input into output, it acts as a transducer. If we want to model computer using Turing machine, we have to look at this aspect more closely.

We can view a Turing Machine transducer m as an implementation of a function f defined by

$$w = f(w)$$

provided that

$$q_0 wt^*_m q_F w$$

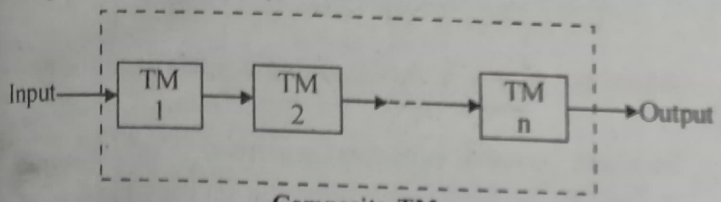
for some final state q_f

Defination : A function f with domain Dis said to be Turing-computable or just computable if there exist some Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, \sqcup, F)$ such that

$$q_0 w \Gamma^*_m q_f f(w) \quad q_f \in F$$

for also $w \in D$

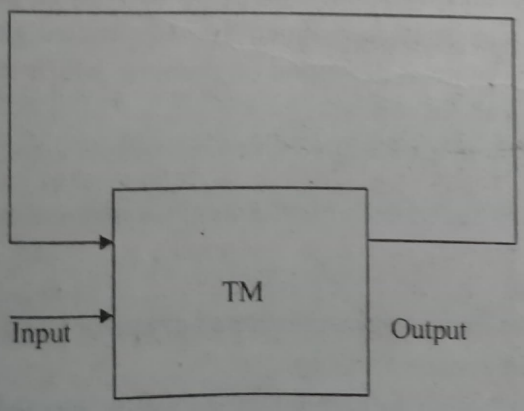
(iv) Composite Turing Machine (TM) and Iterated Turing Machine : Two or more Turing Machine can be combined to solve a collection of simpler problems, so that the output of one Turing Machine forms the input to the next Turing Machine and so on. This is called as **Composition.** For realizing a compsite TM, the functional matrics of the component TMs are combined by increasing and relabelling I and suitably branching to an appropriate state rather than the halt state at the completion of the performance of each component TM.



Composite TM

Another way of having a combination TM is by applying its own output as input respectively. This is called as iteration or recursion.

The idea of composite TM gives rise to the concept of breaking the complicated job into number of jobs implementing each separately and then combining them together to get answer for the job required to be done. Therefore, we can divide a problem into simple jobs and design different TM, for each. Then we can take the composition of all TMs to get work done what we want initially. Modular programming is definatcly influened by CTM (Composite TM).



Iterative TM

(v) Universal TM (or UTM) : *Refer to Q.1.*

(vi) Multistack TM: This symbols to the left of the head of the TM can be stored on one stack, while the symbols on the right of the head can be placed on the other stack. On each stack, symbols closer to the turing machine's head are placed closer to the top of the stack. This type is called as Multistack Turing Machine.

Q.22 ***Explain Chomsky classification of language with the help of suitable example.*** *[R.T.U. 2019, 13, 12]*

OR

Write short note on Chomsky Hierarchy of languages in detail.

[R.T.U. 2016, 2014, 2011, Raj. Univ. 2007, 2006, 2004,2003]

OR

Discuss chomsky hierarchy in detail. *[R.T.U. 2015]*

Ans. According to Chomsky (Name of Scientist) there are four types of grammar:

(1) Type-3 grammar or regular grammar
(2) Type-2 grammar or context free grammar
(3) Type-1 grammar or context sensitive grammar
(4) Type-0 grammar or unrestricted grammar

(1) Type-3 Grammar or Regular Grammar :

These types of grammar follows the following rule of production:

$m \to n$ is a production rule for regular grammar where, $m \in V_N$ and $n \in \{\lambda, a, b, aA, bB, bA, aB\}$

for $V_N = \{A, B\}$

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

and A is starting non terminal (start symbol).

Example 1 : Production rules for regular grammar.

(i) $A \to \lambda$ (ii) $A \to a$ (iii) $A \to b$
(iv) $A \to aA$ (v) $A \to bB$ (vi) $B \to b$
(vii) $B \to aA$ (viii) $B \to aB$

Solution : All of the (i) to (viii) are regular grammar.

In regular grammar left side of production will always be only one variable and in right side there will be single terminal or one non terminal (variable) followed by terminal or λ only.

Example 2 :

Given $V_N = \{A, B, C\}$

$\Sigma = \{0, 1\}$, A is start symbol. Productions rules are following :

(a) $C \to \lambda$ (b) $A \to BC$ (c) $A \to 0C$
(d) $AB \to 0B$ (e) $AC \to 0$ (f) $A \to 01$

Solution : (a), (c) are production for regular grammar.

(b) $A \to BC$ are not in RG (Regular Grammar) because right side

(d) $AB \to 0B$ are not in RG because left side must contain only one non terminal (variable)

(e) $AC \to 0$ (same as (d)) (not in RG)

(f) $A \to 01$ are not in RG (Same logic for b).

(2) Type-2 Grammar or Context Free Grammar:

These types of grammar follows the following rule of production :

$m \to n$ is a production rule for context free grammar (CFG) where

$m \in V_N$

and $n \in (V_N \cup \Sigma)^*$

Example 3 : Following production are under the context free grammar (CFG) for $V_N = \{S, A, B\}$ and $\Sigma = \{0, 1\}$. Production rules are following-

(a) $S \to 0$ (b) $S \to 0A$ (c) $A \to \wedge$

(d) $S \to 0SA$ (e) $S \to 0AB$ (f) $S \to B$

Solution : Here from (a) to (f) left side of production are single variable and right side of production are any combination of terminal and variable means $(V_N \cup \Sigma)^*$ or $(S, A, B, 0, 1)^*$.

In the above example

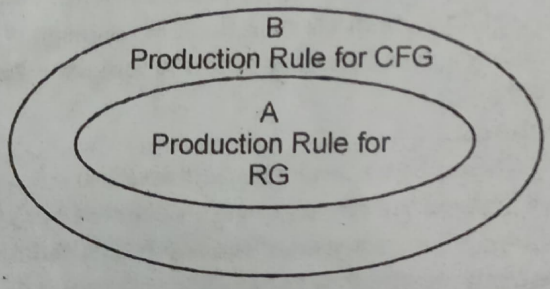
(a) $S \to 0$ (b) $S \to 0A$ (c) $A \to \wedge$

are also follow the production rules for regular grammar or type-3 grammar. So we can say that production rules for regular grammar is subset of production rule for context free grammar

= Production rule for RG $\subseteq$ Production rule for CFG

Diagrammatically we can understand this.

So, any grammar that will be regular will also be context free grammar but the converse is not always true *i.e.*, some production which follows the CFG property ($A \to aBB$) do not follow the regular grammar property.



Example 4 :

Given $V_N = \{A, B, C\}$

$\Sigma = \{0, 1\}$, A is start symbol

(a) $A \to 0AB$ (b) $A \to AB0$ (c) $A \to A0C$

(d) $A \to 0CB$ (e) $A \to 0$ (f) $AB \to 0C$

(g) $BC \to 1A$ (h) $A1 \to 1B$

Solution : From (a) to (e) are production rule for context free grammar because left hand side are only one variable and right side have element of $(V_N \cup \Sigma)^*$.

(f), (g), (h) are not production rule for context free grammar because left hand side are not only one variable or non terminal.

(3) Type-1 Grammar or Context Sensitive Grammar: This types of grammar follows the following rule of production:

$\phi A \psi \to \phi \alpha \psi$ if $\alpha \neq \lambda$ and erasing of A is not permitted.

Example 5 : $\dfrac{a\,A\,bcD}{\phi\,A\,\psi} \to \dfrac{a\,bc\,bcD}{\phi\,\alpha\,\psi}$

In the above grammar if A = A and ψ = bcD, ϕ =a, α = bc then it follows:

$\phi A \psi \to \phi \alpha \psi$ where $\alpha \neq \lambda$

So it is context sensitive grammar

Example 6 : $\dfrac{A\ B}{\phi\ A} \to \dfrac{A\ bBc}{\phi\ \alpha}$

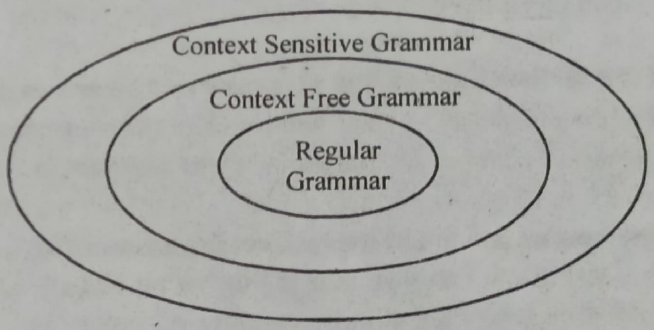
When we add λ in the above grammar on both sides then there will be no difference between added ϕ grammar and original grammar so we can write above grammar as:

$$\frac{A\ B\ \lambda}{\phi\ A\ \psi} \to \frac{A\ bBc\ \lambda}{\phi\ \alpha\ \psi}$$

where ϕ = A, $\psi = \lambda$, A = B and α = bBC which is not null.

So above grammar is context sensitive.

Production set for RG $\subseteq$ Production set for CFG $\subseteq$ Production set for CFG $\subseteq$ Production set for CSG



(4) Type - 0 Grammar or Unrestricted Grammar: Every production which follow the production rule for grammar are called unrestricted grammar. So, unrestricted grammar definition is same as definition of grammar which we have discussed in above section.

$\Rightarrow m \to n$

where $m \in (V_N \cup \Sigma)^*$ which contain at least one variable

and $n \in (V_N \cup \Sigma)^*$ So, $ab \to cd$

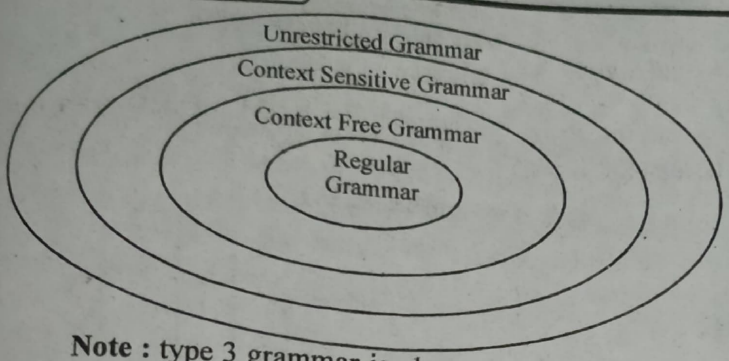
are not Grammar on unrestricted grammar because left side does not contain variables.

** It is called unrestricted grammar because this grammar does not follow any particular types.*

So we can say

Production set for RG $\subseteq$ Production set for CFG $\subseteq$ Production set for CSG $\subseteq$ Production set for unrestricted grammar.

Diagrammatically:



Note : type 3 grammar is also regular grammar.

The following fig describes the relation b/w the four types of languages and automata:

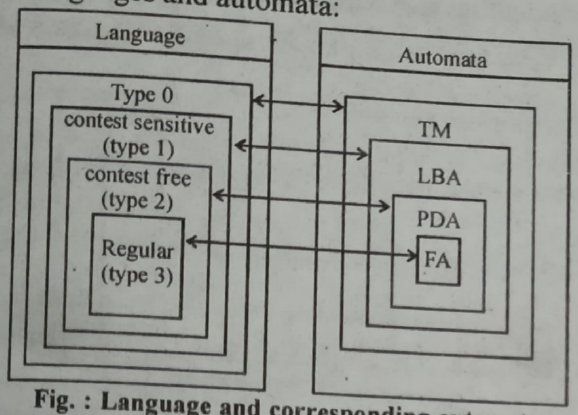


Fig. : Language and corresponding automata

Q.23 ***Make the comment on the following statement finite state machine with two stack is as powerful as Turing Machine.*** ***[R.T.U. 2010, Raj.Univ. 2007]***

Ans. Pushdown automata is differ from finite state machines in two ways

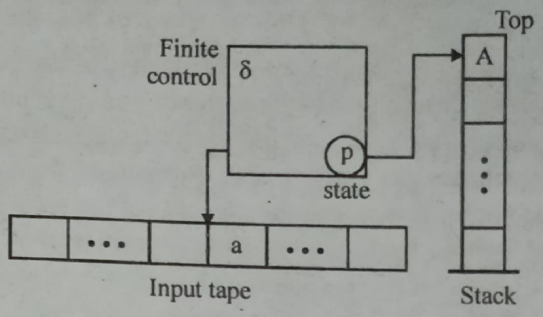
1. They can use the top of the stack to decide which transition to take.

2. They can manipulate the stack as part of performing a transition.

Pushdown automata choose a transition by indexing a table by input signal, current state and the symbol at the top of the stack. This means that those three parameters completely determine the transition path that is chosen. Finite state machines just look at the input signal and the current state : they have no stack to work with. Pushdown automata add the stack as a parameter for choice.

Pushdown automata can also manipulate the stack, as part of performing a transition. Finite state machines choose a new state, the result of following the transition. The manipulation can be to push a particular symbol to the top of the stack, or to pop off the top of the stack. The automaton can alternatively ignore the stack and leave it as it is. The choice of manipulation (or no manipulation) is determined by the transition table.

Put together : Given an input signal, current state, and stack symbol, the automaton can follow a transition to another state and optionally manipulate (push or pop) the stack.



Refer to Q.7.

If we allow a finite automaton access to two stacks instead of just one, we obtain a more powerful device, equivalent in power to a **Turing machine**. A **linear bounded automaton** is a device which is more powerful than a pushdown automaton but less so than a Turing machine.

Pushdown automata are equivalent to context-free grammars : for every context-free grammar, there exists a pushdown automaton such that the language generated by the grammar is identical with the language generated by the automaton, which is easy to prove. The reverse is true, though harder to prove : for every pushdown automaton there exists a context-free grammar such that the language generated by the automaton is identical with the language generated by the grammar.

Definition

A pushdown automaton consists of

- A finite nonempty set of states denoted by Q,
- A finite nonempty set of input symbols denoted by Σ,
- A finite nonempty set of pushdown symbols denoted by Γ,
- A special state called the initial state denoted by q_0,
- A special pushdown symbol called the initial symbol on the pushdown store denoted by Z_0,
- The set of final states, a subset of Q denoted by F, and
- The transition function δ from $Q \times (\Sigma \cup \{\Lambda\} \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$.)

Symbolically, a PDA is a 7-tuple, viz. $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.

An element is a transition of finite automata M. It has the intended meaning that M, in state, with in the input and with as topmost stack symbol, may read a, change the state to q, pop A, replacing it by pushing. The letter $\in$ (epsilon) denotes the empty string and the component of the transition relation is used to formalize that the PDA can either read a letter from the input, or proceed leaving the input untouched. In many texts the transition relation is replaced by an (equivalent) formalization, where is the transition function,

mapping into finite subsets of. Here δ(p, a, A) contains all possible actions in state p with A on the stack, while reading a on the input. One writes for the function precisely when for the relation. Note that finite in this definition is essential.

Computations

In order to formalize the semantics of the pushdown automaton a description of the current situation is introduced. Any 3-tuple is called an instantaneous description (ID) of M, which includes the current state, the part of the input tape that has not been read, and the contents of the stack (topmost symbol written first). The transition relation d defines the step-relation of M on instantaneous descriptions. For instruction there exists a step, for every and every.

In general pushdown automata are nondeterministic meaning that in a given instantaneous description (p, w, β) there may be several possible steps. Any of these steps can be chosen in a computation. With the definition in each step always a single symbol (top of the stack) is popped, replacing it with as many symbols as necessary. As a consequence no step is defined when the stack is empty.

Computations of the pushdown automaton are sequences of steps. The computation starts in the initial state q_0 with the initial stack symbol Z on the stack and a string w on the input tape, thus with initial description (q_0, w, Z). There are two modes of accepting. The pushdown automaton either accepts by final state, which means after reading its input the automaton reaches an accepting state (in F), or it accepts by empty stack (), which means after reading its input the automaton empties its stack. The first acceptance ı .ode uses the internal memory (state), the second the external memory (stack).

Formally one defines

1. with and (final state)
2. with (empty stack)

Here represents the reflexive and transitive closure of the step relation meaning any number of consecutive steps (zero, one or more). For each single pushdown automaton these two languages need to have no relation : they may be equal but usually this is not the case. A specification of the automaton should also include the intended mode of acceptance. Taken over all pushdown automata both acceptance conditions define the same family of languages.

Theorem : For each pushdown automaton M one may construct a pushdown automaton M such that L(M) = N(M′), and vice versa, for each pushdown automaton M′ such that N(M) = L(M′)

Example : The following is the formal description of the PDA which recongnizes the language by fixed state : PDA for (by final state),

where Q = {p, q, r}

Σ = {0, 1}

F = {A, Z}

F = {r}

d consists of following six instructions :

(p, 0, Z, p, AZ), (p, 0 A, p, AA), (p, ε, Z, q, A), (p, ε, A, q, A), (q, 1, A, q, ε) and (q, ε, Z, r, Z).

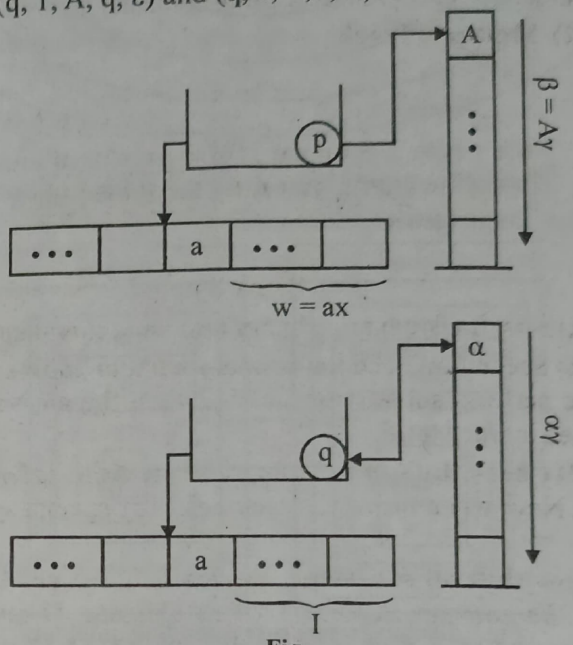


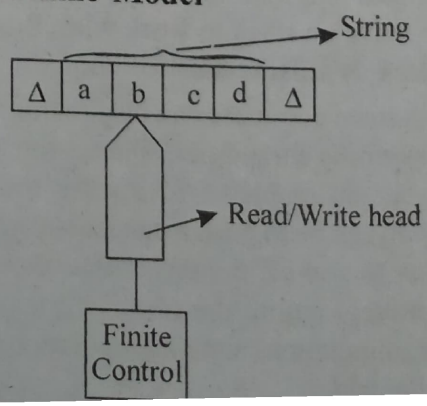
Fig.

In words in state p for each symbol 0 read, one A is onto the stack. Pushing symbol A on top of another A is formalized as replacing top A by AA. In state q for each symbol 1 read one A is popped. At any moment the automaton may move from state p to state q, while it may move from state q to accepting state r only when the stack consists of a single Z. There seems to be no generally used representation for PDA. Here we have depicted the instruction (p, a, A, q, α) by an edge from state p to state q labeled by a;A / α (read a; replace A by a).

Q.24 ***Explain the properties of context sensitive language. Which type of model we can use to recognize these languages.*** *[RTU 2009]*

Ans. Context Sensitive Language : *Refer to Q.3.*

Turing Machine Model

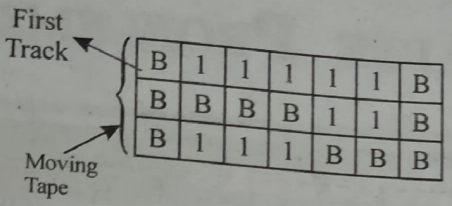


There are four programming techniques of turing machine

(1) Storage in finite control (2) Multiple track
(3) Check the symbol (4) Subroutine

(1) Storage in Finite Control : Turing machine stores some amount of information in finite control.

(2) Multiple Track



B → Blank Symbol

(a) First track contains unary number equivalent to 5.

(b) Second track contains unary number equivalent to 2. If we perform subtraction i.e. 5–2 then the answer 3 is contained in track third.

(3) Check the Symbol : Suppose the string is AABBC AABB. Now, when we read A we mark it by special symbol *.

Now shift all A's. When we reach C we get A right after C.We compare A with marked element. If matched then convert A to *. Now, backtrack until we get * before C. This procedure continues.

So, AA BBCAA BB
↑

*A BBCAA BB
↑

*A BBCAA BB
↑

*A BBCAA BB
↑

*A BBCAA BB
↑

*A BBCAA BB
↑

Compare A with * matches so.

*A BBC *A BB
↑

*A BBC *A BB
↑

*A BBC *A BB
↑

*A BBC *A BB
↑

We get * read A and move it

**BB C *ABB
↑

** BB C *ABB
↑

**BB C *ABB
↑

Continues

**BB C *A BB
↑
matches

** BBC **BB
↑

** BBC **BB
↑

Moving B to * and moves upto * after C

*** BC **BB
↑
matches

*** BC ***B
↑

**** C ***B
↑
matches

**** C ****
↑

String completely recognized.

(4) Subroutine : Subroutines are used to build modularity in program development process.

❑❑❑

TRACTABLE AND UNTRACTABLE PROBLEMS 5

PREVIOUS YEARS QUESTIONS

PART-A

Q.1 ***What is vertex cover problem?*** ***[R.T.U. 2019]***

Ans. Vertex Cover Problem : The vertex cover takes a graph G and integer K as input and asks whether there exists a vertex cover for G which contains at most K vertex or not. It is already noticed that vertex cover is in NP. Now we have to show that it is NP-hard. For this we have to reduce the 3-SAT problem in polynomial time. This reduction is accomplished in two steps :

- First, it represents an example in which a logic problem is reduced to a graph problem.
- Second, it describes an application of the component design proof technique.

Q.2 ***Define P in brief.***

Ans. P : P is the set of decision problems with a yes–no answer that is polynomial bound.

A problem is said to be polynomial–bound if there exists a polynomial bound algorithm for it. It is also to be noted that not for all the problem, the class P has "acceptably efficient" algorithm. Also, if a problem does not belong to class P then it is intractable.

An algorithm is said to be polynomial bounded if its worst–case complexity is bound by a polynomial function P of input size n. That means, for each input of size n, the algorithm terminates after atmost P(n) steps;

For instance, $n^7 + 24n^2 + 65$

Q.3 ***Define NP hard in brief.***

Ans. NP–Hard : If a language L_1, defines some decision problem then it reduces to the language L_2 in polynomial time only.

If there exists a function F which can be computed in polynomial time, the function F takes an input x to L_1 and converts it to an input F(x) of L_2, in such a manner that $x \in L_1$ if and only if $F(x) \in L_2$.

The notation $L_1 \xrightarrow{poly} L_2$ refers to the language L_1 which is polynomial time reducible to language L_2.

We can also say that language L_2, defining some decision problem, is NP–hard if for every $L_1 \in$ NP, where language $L_1 \in$ NP is polynomial–time reducible to language L_2.

Q.4 ***Define NP complete in brief.***

Ans. NP Complete : If a language L_2 is NP–hard and it also belongs to the class NP, then language L_2 is said to be NP–complete. If any problem that belongs to NP–complete is solved in polynomial time then all the problems belonging to the class NP can be solved in polynomial time. It can be observed that if anyone shows a deterministic polynomial–time algorithm for even one NP–complete problem, then P = NP.

Q.5 ***Show 3 SAT is NP complete.***

Ans. A special case of SAT that is particularly useful in proving NP-hardness results is called 3-SAT.

$T_{CSAT}(n) \le O(n) + T_{3SAT}(O(n)) \Rightarrow T_{3SAT}(n) \ge T_{CSAT}(\Omega(n) - O(n))$

As 3SAT is NP-hard and because 3SAT is a special case of SAT, it is also in NP. Therefore, 3SAT is NP-complete.

PART-B

Q.6 ***Explain the Cook's theorem with suitable example.*** *[R.T.U. 2016]*

OR

What is the use of Cook's theorem? Prove it with an example.

[R.T.U. 2012, 2011, 2010, Raj. Univ. 2003]

OR

What is Cook's theorem? Explain. *[R.T.U. 2013, 2009]*

Ans. Cook's theorem states that satisfiability is in P if and only if P = NP. If P = NP, then satisfiability is in P. It remains to be shown that if satisfiability is in P, then P = NP.

Proof : To do this, we show how to obtain from any polynomial time nondeterministic decision algorithm A and input I is a formula Q (A, I) such that Q is satisfiable if A has a successful termination with input I.

If the length of I is n and the time complexity of A is p(n) for some polynomial p(), then the length of Q is $O(p^3(n) \log n) = O(p^4(n))$. The time needed to construct Q is also $O(p^3(n) \log n)$. A deterministic algorithm Z to determine the outcome of A on any input I can be easily obtained.

Algorithm Z simply-computes Q and then uses a deterministic algorithm for the satisfiability problem to determine whether Q is satisfiable.

If O (p(m)) is the time needed to determine whether a formula of length m is satisfiable, then the complexity of Z is $O(p^3(n)\log n + q(p^3(n)\log n))$. If satisfiability is in P, then q(m) is a polynomial function of m and the complexity of Z becomes O(r(n)) for some polynomial r(). Hence, if satisfiability is in P, then for every nondeterministic algorithm A in NP we can obtain a deterministic Z in P. So, the above construction shows that if satisfiability is in P, then P = NP.

Before going into the construction of Q from A and I, we make some simplifying assumptions on our nondeterministic machine model and on the form of A. These assumptions do not in any way alter the class of decision problems in NP or P. The simplifying assumptions are as follows :

1. The machine on which A is to be executed is word oriented. Each word is w bits long. Multiplication, addition, subtraction, and so on, between numbers one word long take one unit of time. If numbers are longer than a word, then the corresponding operations take at least as many units as the number of words making up the longest number.

2. A simple expression is an expression that contains at most one operator and all operands are simple variables (*i.e.*, no array variables are used). Some simple expression are -B, B + C, D or E, and F. We assume that all assignments in A are in one of the following forms:

(a) (simple variable) : = (simple expression)

(b) (array variable) : = (simple variable)

(c) (simple variable) : = (array variable)

(d) (simple variable) : = Choice(S)

where S is a finite set $\{S_1, S_2, \ldots, S_k\}$ or l, u. In the latter case the function chooses an integer in the range [l : u].

Indexing within an array is done using a simple integer variable and all index values are positive. Only one-dimensional arrays are allowed. Clearly, all assignment statements not falling into one of the above categories can be replaced by a set of statements of these types. Hence, this restriction does not alter the class NP.

3. All variables in A are of type integer or boolean.

4. Algorithm A contains no read or write statements. The only input to A is via its parameters. At the time A is invoked, all variables (other than the parameters) have value zero (or false if boolean).

5. Algorithm, A contains no constants. Clearly, all constants in any algorithm can be replaced by new variables. These new variables can be added to the parameter list of A and the constants associated with them can be a part of the input.

6. In addition to simple assignment statements, A is allowed to contain only the following types of statements :

(a) The statement goto k, where k is an instruction number.

(b) Success(), Failure().

(c) Algorithm A may contain type declaration and dimension statements. These are not used during execution of A and so need not be translated into Prob. The dimension information is used to allocate array space. It is assumed that successive elements in an array are assigned to consecutive words in memory. It is assumed that the instructions in A are numbered sequentially from 1 to l (if A has l instruction).

Every statement in A has a number. The goto instructions in (a) and (b) use this numbering scheme to effect a branch. It should be easy to see how to rewrite repeat-until, for and so on statements in terms of goto and if b then goto a statement. Also, note that the goto k statement can be replaced by the statement if true then goto k. So, this may also be eliminated.

7. Let p(n) be a polynomial such that A takes no more than p(n) time units on any input of length n. Because of the complexity assumption of 1, A cannot change or use more than p(n) words of memory. We assume that A uses some subset of the words indexed 1, 2, 3, ...p(n). This assumption does not restrict the class of decision problems in NP. To see this, let f (1), f(2), ...,f(k), $1 \le k \le p(n)$, be the distinct words

used by A while working on input I. We can construct another polynomial time nondeterministic algorithm A' that uses 2p(n) words index 1, 2, .., 2p(n) and solves the same decision problem as A does. A' simulates the behaviour of A. However, A' maps the addresses f (1), f(2), ..., f(k) onto the set {1, 2, ..., k}. The mapping function used is determined dynamically and is stored as a table in words p(n) + 1 through 2p(n).

If the entry at word p(n) + i is j, then A' uses word i to hold the same value that A stored in word j. The simulation of A proceeds as follows: Let k be the number of distinct words referenced by A upto this time. Let j be a word referenced by A in the current step. A' searches its table to find word p(n) + i, $1 \le i \le k$, such that the contents of this word is j. If no such i exists, then A' sets k: = k + 1; i : = k; and word p(n) + k gives the value j. A' makes use of the word i to do whatever A would have done with word j. Clearly, A' and A solve the same decision problem.

The compelxity of A; is $O(p^2(n))$ as it takes A'p(n) time to search its table and simulate a step of A. Since $p^2(n)$ is also a polynomial in n, restricting our algorithms to use only consecutive words that does not alter the classes P and NP.

Q.7 ***Explain NP and Hard NP Complete with example.***

[R.T.U. 2016]

OR

Explain the terms P, NP, NP-Hard, NP-complete with suitable example. Also give relationship between them. ***[R.T.U. 2014]***

Ans.(i) P : P is the set of decision problems with a yes–no answer that is polynomial bound.

A problem is said to be polynomial–bound if there exists a polynomial bound algorithm for it. It is also to be noted that not for all the problems the class Phas "acceptably efficient" algorithm. Also, if a problem does not belong to class P then it is **intractable.**

Note : An algorithm is said to be polynomial bounded if its worst–case complexity is bound by a polynomial function P of input size n. That means, for each input of size n the algorithm terminates after atmost P(n) steps; For instance, $n^7 + 24n^2 + 65$

Decision Problems : The problems under this class have the single bit output which shows 0 or 1 *i.e.*, the answer for the problem is either zero or one.

For instance, some decision problems are :

- Given two sets of strings S_1 and S_2, does S_2 a substring of S_1 ?
- Given two sets of elements S_1 and S_2, does both the sets contain same number of elements ?

Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as optimization problem. For solving optimization problem, an optimization algorithm is used. For instance, the optimization problem is as follows.

- Given a weighted graph G, and an integer i, does G have a minimum spanning tree of weight at most i ?
- Given S, does there exist a subset of elements that fits in the knapsack, and has total profit of at least S ?

We can say that a given algorithm A accepts the string 'S' only when A produces the output 'yes' on input 'S'. As a set of string is referred to a language, a decision problem can also be viewed as a set L of strings where L is a language. Thus, an algorithm A accepts languages L if A produces the output 'yes' on input 'S' which belong to L otherwise it produces output 'no'.

It is to be noted that the class P problems include all the decision problems (or languages) L that can be accepted in the worst–case running time. Thus, for algorithm A, it accepts $S \in L$, in polynomial time p(w), where 'n' is the input size of S and produces output 'yes'. But it is noticeable that the class P definition does not say anything about output 'no'. We refer to this situation as a complement of the algorithm A for output 'yes' for a given set of binary strings that are not present in L.

We can also create an algorithm C that accepts the complement of L if given an algorithm A that accepts a language L in polynomial time, p(n). Therefore, if a language L, showing such decision problem, is in class P.

(ii) NP : The complexity class NP includes the complexity class P but allows for the languages that are not present in P. But, in the case of NP problems we perform an additional operation:

Select : This problem selects a bit (0 or 1) in a non-deterministic way and assigns it to b. When an algorithm A takes the advantage of Select primitive operation then we say that A is non–deterministic. In this approach out of several calls to Select operation, those calls are chosen which lead to acceptance if there exists a set of outcomes. It is noticeable that this operation's working is not same as of using random choices.

The complexity class NP includes all decision problems (or languages L) that can be accepted non-deterministically in the polynomial time. Thus for a given algorithm A, if $S \in L$, an input S, there exists a set of outcomes to the Select calls in A so that it produces output 'yes' in polynomial time,p(n), where n is the input size of S.

The definition of complexity class NP does not say anything about rejection of the string. Algorithm A running in polynomial time p(n) can take more than p(n) steps when A produces outputs 'no'. Also, a polynomial number of calls to select is involved in the non–deterministic acceptance, the complement of L is not necessarily in NP, where L is the language in NP.

There exists a special class, called co–NP which includes all the languages whose complement is in NP, and many researchers and scientists believe that co–NP ≠ NP.

Is P = NP ? : Most of the researchers and scientists believe that class P problems are different from NP and co – NP or their intersection.

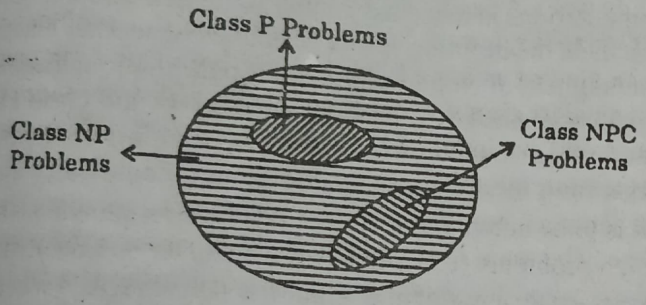


Fig.

Problem : Given a Graph G, does this graph G contain a Hamiltonian cycle. That means whether there exists a simple cycle such that each vertex is visited only once other than the starting vertex or not.

Lemma : Hamiltonian cycle is in NP.

Proof : Here we will define a non–deterministic algorithm A that takes an input graph G encoded as an adjacency list in binary notation, with the vertices labelled from 1 to n.

Then we make A to call the Select method iteratively for obtaining the sequence of vertices, and perform the check so that all the vertices appear only once except the start vertex (which comes twice). This can be done by sorting the sequence. Also, we verify that the sequence defines a cycle of vertices and edges in G.

If there exists a cycle in G where all vertices are visited only once except the first and the last which are the same then there also exists the sequence for which A produces output 'yes'– Similarly, we can say that if A outputs 'yes' then a graph G has cycle in such a manner that all vertices are visited once except first and last which are the same. That means A non–deterministically accepts the language Hamiltonian – Cycle. Thus, we can say that Hamiltonain–Cycle is in NP.

(iii) NP–Complete : A decision problem L is NP–complete if it is in class NP for every other problem L' in NP, $L \propto L'$. That means L' is polynomially reducible to L, (that means NP–complete problem are the hardest in NP).

Theorem : If any NP–complete problem belongs to class P, then P = NP.

Proof : Let any decision problem $L \in NP$ and also $L \in P$. Then by the rules of NP–complete problems

$$\forall L' \in NP,\ L \propto L\ L \in P.$$

$$\forall L' \in NP, L' \propto L \wedge L \in P$$

Then according to the polynomial reducibility {if $L_1 \propto L_2$ and $L_2 \in P$ then $L_1 \in P$}

$$\forall L' \in NP, L' \in P$$

That means, P = NP.

All NP problems are NP- hard, but some NP-hard problems are not known to be NP complete.

Since deterministic algorithm are just a special case of non-deterministic ones, we conclude that $P \subseteq NP$.

It is easy to see that there are NP- hard problems that are not NP complete. Only a decision problem can be NP complete. However, an optimization problem may be NP- hard. Furthermore if L_1 is a decision problem and L_2 is an optimization problem it is quite possible that $L_1 \propto L_2$. One can trivially show that knapsack decision problem reduces to knapsack optimization problem.

For clique problem one can easily show that the clique decision problem reduces to clique optimization problem. In fact one can also show that these optimization problem reduces to their corresponding decision problem. Yet optimization problem cannot be NP complete whereas decision problem can. There also exist NP- hard decision problem that are not NP complete.

Q.8 ***Write short notes on the following :***

(a) Complexity classes of decision problems.

(b) Approximation algorithms.

(c) Cook's theorem and its applications. ***[R.T.U. 2015]***

Ans.(a) The purposes of complexity theory are to ascertain the amount of computational resources required to solve important computational problems, and to classify problems according to their difficulty.The resource most often discussed is computational time, although memory (space) and circuitry (orhardware) have also been studied. The main challenge of the theory is to prove lower bounds, i.e.,that certain problems cannot be solved without expending large amounts of resources. Although it is easy to prove that inherently difficult problems exist, it has turned out to be much more difficult to prove that any interesting problems are hard to solve. There has been much more success in providing strong evidence of intractability, based on plausible, widely-held conjectures.

In both cases, the mathematical arguments of intractability rely on the notions of reducibility and completeness. Before one can understand reducibility and completeness, one must grasp the notion of a complexity class.

First, however, we want to demonstrate that complexity theory really can prove to even the most skeptical practitioner that it is hopeless to try to build programs or circuits that solve certain problems. As our example, we consider the manufacture and testing of logic circuits and communication protocols. Many problems in these domains are solved by building a logical formula over a certain vocabulary, and then determining whether the formula is logically valid, or whether counter examples (that is, bugs) exist. The choice of vocabulary for the logic is important here.

Typically, a complexity class is defined by

(1) A model of computation

(2) A resource (or collectionof resources)

(3) A function known as the complexity bound for each resource.

The models used to define complexity classes fall into two main categories:

(a) Machine based models

(b) Circuit-based models.

Turing Machines (TMs) and Random-Access Machines (RAMs) are the two principal families of machine models. Other kinds of (Turing) machines were also introduced including deterministic, nondeterministic, alternating, and oracle machines. When we wish to model real computations, deterministic machines and circuits are our closest links to reality. Then why consider the other kinds of machines? There are two main reasons. The most potent reason comes from the computational problems whose complexity we are trying to understand. The most notorious examples are the hundreds of natural NP-complete problems. To the extent that we understand anything about thecomplexity of these problems, it is because of the model of non-deterministic Turing machines. Non-deterministic machines do not model physical computation devices, but they do model real computational problems. There are many other examples where a particular model of computation has been introduced in order to capture some well-known computational problem in a complexity.

The second reason is related to the first. Our desire to understand real computational problems has forced upon us a repertoire of models of computation and resource bounds. In order to understand the relationships between these models and bounds, we combine and mix them and attempt to discover their relative power. Consider, for example, nondeterminism. By consideringthe complements of languages accepted by non-deterministic machines, researchers were naturally led to the notion of alternating machines. When alternating machines and deterministic machines were compared, a surprising virtual identity of deterministic space and alternating time emerged.

Subsequently, alternation was found to be a useful way to model efficient parallel computation.

This phenomenon, whereby models of computation are generalized and modified in order to clarify their relative complexity, has occurred often through the brief history of complexity theory, and has generated some of the most important new insights. Other underlying principles in complexity theory emerge from the major theorems showing relationships between complexity classes. These theorems fall into two broad categories. Simulation theorems show that computations in one class can be simulated by computations that meet the defining resource bounds of another class. The containment of Nondeterministic Logarithmic space(NL) in polynomial time (P), and the equality of the class P with alternating logarithmic space,are simulation theorems. Separation theorems show that certain complexity classes are distinct. Complexity theory currently has precious few of these. The main tool used in those separation theorems we have is called diagonalization. This ties in to the general feeling in computer science that lower bounds are hard to prove. Our current inability to separate many complexity classes from each other is perhaps the greatest challenge to our intellect posed by complexity theory.

Time and Space Complexity Classes:

We begin by emphasizing the fundamental resources of time and space for deterministic and nondeterministicTuring machines. We concentrate on resource bounds between logarithmic and exponential,because those bounds have proved to be the most useful for understanding problems that arise in practice. Time complexity and space complexity were defined. Fundamental time classes and fundamental space classes, given functions t(n) and s(n):

- DTIME [t(n)] is the class of languages decided by deterministic Turing machines of time complexity t(n).
- NTIME [t(n)] is the class of languages decided by nondeterministic Turing machines of time complexity t(n).
- DSPACE [s(n)] is the class of languages decided by deterministic Turing machines of space complexity t(n).
- NSPACE [s(n)] is the class of languages decided by nondeterministic Turing machines of space complexity t(n).

We some times abbreviate DTIME [t(n)] to DTIME[t] (and so on) when t is understood to be a function, and when no reference is made to the input length n.

Canonical Complexity Classes

The following are the canonical complexity classes:

- L = DSPACE [log n] (determine log space)
- NL = NSPACE [log n] (nondeterministic log space)
- P = DTIME $[n^{0(1)}] = U_{k\geq 1}$ DTIME $[n^k]$ (nondeterministic polynomial time)
- PSPACE = DSPACE $[n^{0(1)}] = U_{k\geq 1}$ DSPACE $[n^k]$ (polynomial time)
- E = DTIME $[2^{0(n)}] = U_{k\geq 1}$ DTIME $[k^n]$
- NE = NTIME $[2^{0(n)}] = U_{k\geq 1}$ NTIME $[k^n]$
- EXP = DTIME $[2^{n0(n)}] = U_{k\geq 1}$ DTIME $[2n^k]$ (deterministic exponential time)
- NEXP = NTIME $[2^{n0(1)}] = U_{k\geq 1}$ NTIME $[2n^k]$ (nondeterministic exponential time)
- EXPSPACE = DSPACE $[2^{n0(1)}] = U_{k\geq 1}$ DSPACE $[2n^k]$ (exponential time)

The space classes PSPACE and EXPSPACE are de_ned in terms of the DSPACE complexity measure.By Savitch's Theorem.

Ans.(b) An approximation algorithm returns a solution to a combinatorial optimization problem that is probably close to optimal (as opposed to a heuristic that may or may not find a good solution). Approximation algorithms are typically used when finding an optimal solution is intractable, but can also be used in some situations where a near-optimal solution can be found quickly and an exact solution is not needed.

Many problems that are NP-hard are also non-approximable assuming P?NP. There is an elaborate theory that analyzes hardness of approximation based on reductions from core non-approximable problems that is similar to the theory of NP-completeness based on reductions from NP-complete problems; we will not discuss this theory in class but a sketch of some of the main results can be found , which is also a good general reference for approximation. Instead, we will concentrate on some simple examples of algorithms for which good approximations are known, to give a feel for what approximation algorithms look like.

- 2-approximation for vertex cover via greedy matchings.
- 2-approximation for vertex cover via LP rounding.
- Greedy O(log n) approximation for set-cover.
- Approximation algorithms for MAX-SAT.

Suppose we are given an NP-complete problem to solve. Even though (assuming P 6= NP) we can't hope for a polynomial-time algorithm that always gets the best solution, can we develop polynomial-time algorithms that always produce a "pretty good" solution? We consider such approximation algorithms, for several important problems.

Suppose we are given a problem for which (perhaps because it is NP-complete) we can't hope for a fast algorithm that always gets the best solution. Can we hope for a fast algorithm that guarantees to get at least a "pretty good" solution? E.g., can we guarantee to find a solution that's within 10% of optimal? If not that, then how about within a factor of 2 of optimal? Or, anything non-trivial? The class of NP-complete problems are all equivalent in the sense that a polynomial-time algorithm to solve any one of them would imply a polynomial-time algorithm to solve all of them (and, moreover, to solve any problem in NP). However, the difficulty of getting a good approximation to these problems varies quite a bit. In this lecture we will examine several important NP-complete problems and look at to what extent we can guarantee to get approximately optimal solutions, and by what algorithms.

Approximates Strategies :

We will define optimization problems in a traditional way. Each optimization problem has three defining features: the structure of the input instance, the criterion of a feasible solution to the problem, and the measure function used to determine which feasible solutions are considered to be optimal. It will be evident from the problem name whether we desire a feasible solution with a minimum or maximum measure. To illustrate, the minimum vertex cover problem may be defined in the following way.

Instance : An undirected graph G = (V, E).

Solution: A subset $S \subseteq V$ such that for every $\{u, v\} \in E$, either $u \in S$ or $v \in S$.

Measure : $|S|$

We use the following notation for items related to an instance I.

- Sol(I) is the set of feasible solutions to I.
- m_I : Sol(I) $\rightarrow$ R is the measure function associated with I. and

Opt(I) $\subseteq$ Sol(I) is the feasible solutions with optimal measure (be it minimum or maximum).

Hence, we may completely specify an optimization problem Π by giving a set of tuples $\{(I, Sol(I), m_I, Opt(I))\}$ over all possible instances I. It is important to keep in mind that Sol(I) and I may be over completely different domains. In the above example the set of I is all undirected graphs, while Sol(I) is all possible subsets of vertices in a graph.

Approximation and Performance : Roughly speaking, an algorithm approximately solves an optimization problem if it always returns a feasible solution whose measure is close to optimal. This intuition is made precise below.

Let Π be an optimization problem. We say that an algorithm A feasibly solves Π if given an instance $I \in \Pi$. A(I) $\in$ Sol(I); that is, A returns a feasible solution to I.

Let A feasibly solve Π. Then we define the approximation ratio α(A) of A to be the minimum possible ratio between the measure of A(I) and the measure of an optimal solution. Formally,

$$\alpha(A) = \min_{I \in \Pi} \frac{m_1(A(I))}{m_1(Opt(I))}$$

For minimization problems, this ratio is always at least 1. Respectively, for maximization problems, it is always at most 1.

Complexity Background : We define a decision problem as an optimization problem in which the measure is 0 – 1 valued. That is, solving an instance I of a decision problem corresponds to answering a yes/no question about I (where yes corresponds to a measure of 1, and no corresponds to a measure of 0). We may therefore represent a decision problem as a subset S of the set of all possible instances: members of S represent instances with measure 1.

Informally, P (polynomial time) is defined as the class of decision problems Π for which there exists a corresponding algorithm A_Π such that every instance $I \in \Pi$ is solved by A_Π within a polynomial ($|I|^k$ for some constant k) number of steps on any "reasonable" model of computation. Reasonable models include single-tape and multi-tape Turing machines, random access machines, pointer machines, etc.

Ans.(c) Cook's Theorem : Cooks modeled a NP-problem (an infinite set) to an abstract turing machine. Then he developed a polytransformation from the machine (i.e., all NP-Class problems) to a particular decision problem, namely, the boolean satisfiability (SAT) problem.

Satisfiability is in P if and only if NP = P.

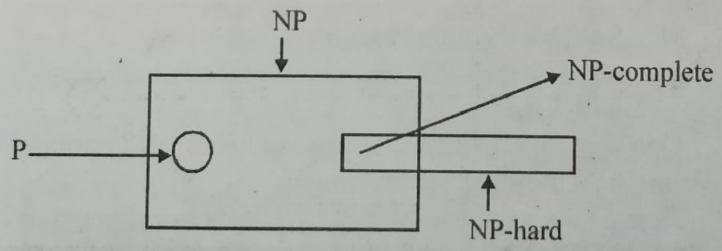


Fig.

Definition : "A problem L is NP-hard if and only if satisfiability reduces to L(satisfiability $\propto$ L). A problem L is NP-complete if and only if L is NP-hard and $L \in NP$.

Significance of Cook's Theorem : If one can find a poly-algorithm for SAJ, then by using cook's poly-transformation one can solve all NP-Class problems in poly-time (Consequently, P-class = NP – class would be proved).

SAT is the historically first identified NP-hard problem.

Further Significance of Cook's Theorem : If you find 2 poly-transformation from SAT to another problem Z, then Z becomes another NP-hard problem. That is, if anyone finds a poly algorithm for Z, then by using your polytransformation from SAT to Z, anyone will be able to solve any SAT problem-instance in poly-time, and hence would be able to solve all NP-class problems in poly-time (by cook's theorem).

Q.9 ***Prove that the circuit satisfiability problem is NP complete.*** *[R.T.U. 2014]*

Ans. We construct a non deterministic algorithm for accepting circuit-sat in polynomial time. We first use the choose method to "guess" the values of the input nodes as well as the output value of each logic gate. Then, we simply visit each logic gate g in C, that is, each vertex with at least one incoming edge. We then check that the "guessed" value for the output of g is in fact the correct value for g's boolean function, be it an AND, OR, or NOT, based on the given values for the inputs for g. This evaluation process can easily be performed in polynomial time. If any check for a gate fails, or if the "guessed" value for the output is 0, then we output "no." If, on the other hand, the check for every gate succeeds and the output is "1," the algorithm outputs "yes." Thus, if there is indeed a satisfying assignment of input values for C, then there is a possible collection of outcomes to the choose statements so that the algorithm will output "yes" in polynomial time. Likewise, if there is a collection of outcomes to the choose statements so that the algorithm outputs "yes" in polynomial time algorithm, there must be a satisfying assignment of input values for C. Therefore, circuit-sat is in NP.

Q.10 ***Write algorithm for approximation for vertex cover problem with suitable example.*** *[R.T.U. 2014]*

Ans. Vertex Cover Problem : *Refer to Q.1.*

Let us consider that 'B_{fg}' be a given instance of the 3-SAT problem, that is, a CNF Boolean formula, where each clause has exactly three literals. Now, we create a graph G and an integer K such that G has a vertex cover of size at most K if and only if 'B_{fg}' is satisfiable. For this we add the following:

- For each input operand I_i in the Boolean formula 'B_{fg}', we add two vertices in G, one of which is labelled as I_i and other as $\bar{I}_i$. After this we add the edge $(I_i, \bar{I}_i)$.
- For each clause $C_i = (m + n + z)$ in B_{fg}, we form a triangle consisting of three vertices and three edges.
- At least two vertices per triangle must be in the cover for the edges in the triangle, for a total of at least 2C vertices.
- Lastly, we create a flat structure where each literal is connected to the corresponding vertices in the triangle which shares the same literal.

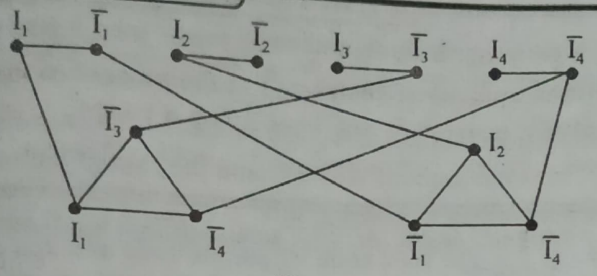


Fig.

The above graph will have a vertex cover of size n + 2C if and only if the expression is satisfiable. Every cover must have at least n + 2C vertices. For showing that our reduction is correct, we have to show the following :

For every satisfying truth assignment there exists a cover : For this select the n vertices that correspond to the true literals to be in the cover. As it is a satisfying truth assignment, atleast one of the three cross edges associated with each clause must already be covered. Now, select the other two vertices to complete the cover.

There exists a satisfying truth assignment for every vertex cover : For this, every vertex cover must contain n first level vertices and 2C second level vertices. Let the truth assignment be defined by the first level vertices. To get the cover at last, one cross–edge must be covered, so that the truth assignment satisfies.

It can be noticed that for a cover to have n + 2C vertices, all the cross edges must be incident on a selected vertex. Let us consider that the n selected vertices from the first level corresponds to true literals. If there exists a satisfying truth assignment, then that means atleast one of the three cross edges from each triangle is incident on a true literal vertex. It is to be noted that by adding the other two vertices to the cover, we cover all the edges associated with the clause.

Vertex-cover problem is to find a vertex cover of minimum size. Using approximation algorithm we have to find a sub-optimal solution to the problem. As a result of this algorithm, we will get a vertex-cover with size no more than twice the size of an optimal vertex cover.

Algorithm Approx_ vertex_ cover

Input to the algorithm is the graph G.

Step 1. Initialize the vertex-cover D to be null.

$C \leftarrow \phi$

Step 2. The set of edges in G is E.

Step 3. Repeat steps 4 to 6 till the set of edges E is empty.

Step 4. Choose an arbitrary edge (u, v) of E.

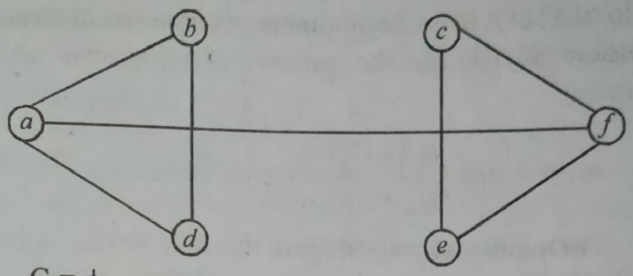
Step 5. Add the endpoints u, v to vertex cover C.

Step 6. Remover every edge incident on either u or v from the set of edges E.

Step 7. return C and Exit.

The running time of this algorithm is O(V + E).

Example



$C = \phi$

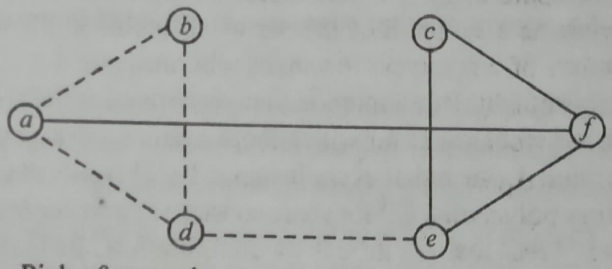
E = {ab, ad, bd, de, af, cf, ef, ce}

pick bd aribitrarily

C = {b, d}

Remove the edges associated with b or d, that is ab, bd, ad and de.

Now E = {af, cf, ef, ce}



Pick cf at random

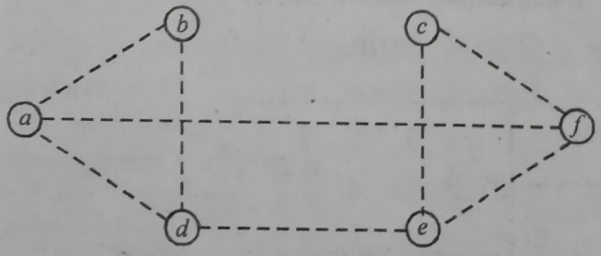
C = {b, d, c, f}

Remove edges associated with c or f, that is af, cf, ef and ce

Now $E = \phi$

So, stop

The cover is c



Q.11 ***Solve the Travelling Salesman Problem (TSP) for the following graph by using the branch and bound algorithm, the tour must be start from vertex 1 and generate only tour in which 2 is visited before 3.***

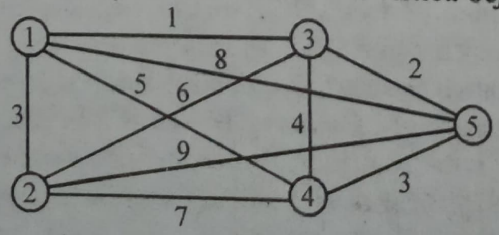


Fig.

[R.T.U. 2013]

Ans. Step 1 : First we have to make the cost matrix from the given graph. Rows and columns denotes the number of cities in the graph.

From city →	1	2	3	4	5	To city →
1	∞	3	1	5	8	
2	∞	∞	6	7	9	
3	∞	∞	∞	4	2	
4	∞	∞	∞	∞	3	
5	∞	∞	∞	∞	∞	

Cost of going from city i to city i is ∞

∴ edge (i, i) is ∞ .

∴ The edges (1, 1), (2, 2), (3, 3) (4, 4) and (5, 5) all is ∞.

So, weight of other edges are as provided in the cost matrix.

Step 2 : Now, we have to reduced cost matrix, R. This is done by selecting the smallest element from each row and column and subtracting it from all other elements.

The aim is to have at least one zero in each row and column. The other entries of the matrix must be non-negative. Reduced cost matrix is constructed to get better minimum cost value.

	1	2	3	4	5	
1	∞	3	1	5	8	−1
2	∞	∞	6	7	9	−6
3	∞	∞	∞	4	2	−2
4	∞	∞	∞	∞	3	−3
5	∞	∞	∞	∞	∞	

	1	2	3	4	5
1	∞	2	0	4	7
2	∞	∞	0	1	3
3	∞	∞	∞	2	0
⇒ 4	∞	∞	∞	∞	0
5	∞	∞	∞	∞	∞

−2 −1

	1	2	3	4	5
1	∞	0	0	3	7
2	∞	∞	0	0	3
3	∞	∞	∞	1	0
⇒ 4	∞	∞	∞	∞	0
5	∞	∞	∞	∞	∞

Now every column and rows are at least 1 zero.

Total amount of subtracted in the process of finding reduced cost matrix = 1 + 6 + 2 + 3 + 2 + 1 = 15

This total amount will become the root of the search tree corresponding to given graph.

Step 3 : If 1 is the root node of parent node and 2 is child node there is an edge between the two nodes as (1, 2). Change all entries in row 1 and column 2 of the reduced cost matrix to ∞. This prevents the use of any more edges leaving vertex 1 or entering vertex 2.

Q.12 ***Prove that TSP problem is NP-complete.***

[R.T.U. 2012]

Ans. In the TRAVELLING-SALES PERSON problem, we have given a graph G and integer parameter I, such that each edge of graph is associated with certain integer cost w, and we are asked if there is a cycle such that it visits all the vertices in G and has total cost almost I.

Given an instance of problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contain each vertex exactly once, sum up the edge costs, and checks whether the sum is at most K. The process can certainly be done in polynomial time.

To prove that Travelling- Salesman Problem (TSP) is NP hard, we show that Hamilton cycle $\leq_p$ TSP. Let G = (V, E) be an instance of Hamiltonian. We construct an instance of TSP as follow :

We form the complete graph G'=(V,E), where E' = {(i, i) :i, j ∈ V} and we define the cost function c by

$$c(i, j) = \begin{cases} 0 \text{ if } (i, j) \in E \\ 1 \text{ if } (i, j) \notin E \end{cases}$$

The instance of TSP is then (G', c, 0), which is easily formed in polynomial time.

We now show that graph G has a Hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a Hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0, since the cost of edges in E' are 0 and 1, concluded that h is Hamiltonian cycle in graph G .

PART-C

Q.13 *Explain Hamiltonian path problem.* *[R.T.U. 2019]*

OR

Prove that Hamilton cycle problem in NP complete. *[R.T.U. 2016]*

OR

Show that the Hamilton Cycle problem is NP-complete. *[R.T.U. 2014]*

Ans. Let us define a nondeterministic algorithm A that takes, as input, a graph G encoded as an adjacency list in binary notation, with the vertices numbered 1 to N. We define A to first iteratively call the choose method to determine a sequence S of N + 1 numbers from 1 to N. Then, we have A check that each number from 1 to N appears exactly once in S (for example, by sorting 5), except for the first and last numbers in S, which should be the same. Then, we verify that the sequence S defines a cycle of vertices and edges in G. A binary encoding of the sequence S is clearly of size at most n, where n is the size of the input. Moreover, both of the checks made on the sequence S can be done in polynomial time in n.

Observe that if there is a cycle in G that visits each vertex of G exactly once, returning to its starting vertex, then there is a sequence S for which A will output "yes." Likewise, if A outputs "yes," then it has found a cycle in G that visits each vertex of G exactly once, returning to its starting point. That is, A non-deterministically accepts the language HAMILTONIAN-CYCLE. In other words, Hamiltonian-Cycle is in NP.

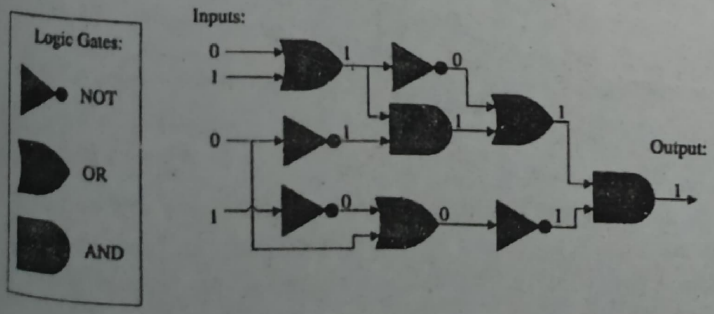


Fig. : An example of Boolean circuit.

Our next example is a problem related to circuit design testing. A Boolean circuit is a directed graph where each node, called a logic gate, corresponds to a simple boolean function, AND, OR, or NOT. The incoming edges for a logic gate correspond to inputs for its boolean function and the outgoing edges correspond to outputs, which will all be the same value, of course, for that gate. (see fig.) Vertices with no incoming edges are input nodes and a vertex with no outgoing edges is an output node.

Circuit-Sat is the problem that takes as input a boolean circuit with a single output node, and asks whether there is an assignment of values to the circuit's inputs so that its output value is "1." Such an assignment of values is called a satisfying assignment.

Q.14 *Suggest an approximation algorithm for traveling sales person problems using minimum spanning tree algorithm. Assume that the cost function satisfies the triangle inequality.* *[R.T.U. 2016]*

Ans. If for the set of vertices a,b,c, $\in$ V, it is true that t (a,c) $\leq$ t(a, b) + t(b, c) where t is the cost function, we say that t satisfies the triangle inequality.

First create a minimum spanning tree the weight of which is a lower bound on the cost of an optimal traveling salesman tour. Using this minimum spanning tree let us create a tour the cost of which is at most 2 times the weight of the spanning tree.

Approximation – Traveling Sales Person Problem

Input: A complete graph G (V. E)

Output: A Hamiltonian cycle

1. Select a "root" vertex r $\in$ V [G].
2. Use MST-Prim (G, c, r) to compute a minimum spanning tree from r.
3. Assume L to be the sequence of vertices visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H that visits the vertices in the order L.

The next set of figures show the working of the proposed algorithm.

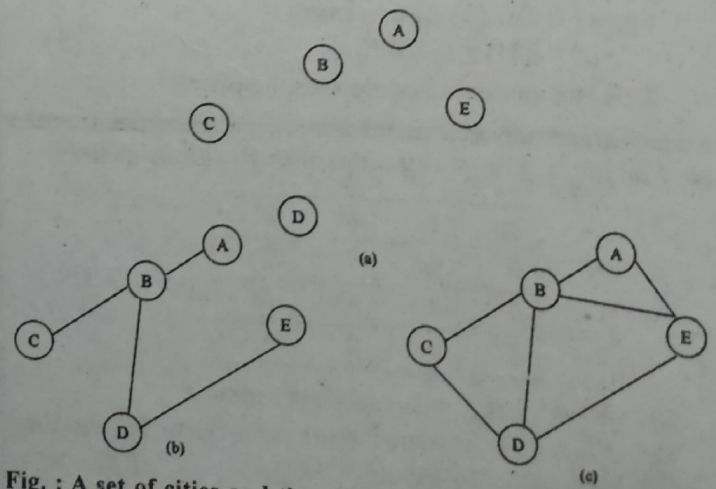


Fig. : A set of cities and the resulting connection after the MST-Prim algorithm has been applied

In figure (a) a set of vertices is shown. Part (b) illustrates the result of the MST-Prim thus the minimum spanning tree MST-Prim constructs. The vertices are visited like {A, B, C, D, E, A) by a preorder walk, part (c) shows the tour, which is returned by the complete algorithm.

Now, assume H* to be an optimal tour for a set of vertices. A spanning tree is constructed by deleting edges from a tour. Thus, an optimal tour has more weight than the minimum-spanning tree, which means that the weight of the minimum spanning tree forms a lower bound on the weight of an optimal tour.

$$c(t) \le c(H^*) \quad ...(1)$$

Let a full walk of T be the complete list of vertices when they are visited regardless if they are visited for the first time or not. The full walk is W.

W = A, B, C, B, D, B, E, B, A

The full walk crosses each edge exactly twice. Thus, we can write:

$$c(W) = 2c(T) \quad ...(2)$$

From equations (1) and (2) we can write that

$$c(W) \le 2c(H^*) \quad ...(3)$$

Which means that the cost of the full path is at most 2 time worse than the cost of an optimal tour. The full path visits some of the vertices twice which means it is not a tour. We can now use the triangle inequality to erase some visits without increasing the cost. The fact we are going to use is that if a vertex a is deleted from the full path if it lies between two visits to b and c the result suggests going from b to c directly.

We are left with the tour: A, B, C, D, E, A. This tour is the same as the one we get by a preorder walk. Considering this preorder walk let H be a cycle deriving from this walk. Each vertex is visited once so it is a Hamiltonian cycle. We have derived H deleting edges from the full walk so we can write:

$$c(H) \le c(W) \quad ...(4)$$

From (3) and (4) we can imply:

$$c(H) \le 2\, c(H^*) \quad ...(5)$$

This last inequality completes the proof.

15 ***For given Travelling salesman problem matrix:***

∞	6	0	2
5	∞	7	0
0	4	∞	14
8	0	0	∞

(i) What is the reduced cost matrix?

(ii) Find the optimal tour of given Travelling Salesman Problem. ***[R.T.U. 2014]***

Ans.(i) Suppose a salesman wants to visits a certain number of cities alloted to him. He knows the distance (or cost or time) of journey between every pair of cities usually denoted by C_{ij} i.e., from city i to city j. His problem is to select such a route that starts from his home city, passes through each city once and only once, and returns to his home city in the shortest possible distance (or at the least cost or in least time).

Formulation of Travelling-Salesman Problem

Suppose C_{ij} is the distance (or cost or time) from city i to city j and $x_{ij} = 1$ if the salesman goes directly from city i to city j, and $x_{ij} = 0$ otherwise.

The minimizes $\sum_i \sum_j x_{ij} C_{ij}$ with the additional restriction that the x_{ij} must be so chosen that no city is visited twice before the tour of all cities is completed.

In a particular he cannot go directly from city i to j itself. This possibility may be avoided in the minimization process by adopting the convention $C_{ij} = \infty$ which ensures that x_{ij} can never be unity.

Alternatively, omit the variable x_{ij} from the problem specification. It is also important to note that only single $x_{ij} = 1$ for each of i and j. The distance (or cost or time) matrix for this problem is given in table.

From City \	A_1	A_2	....	A_n
A_1	∞	C_{12}	....	C_{in}
A_2	C_{21}	∞	...	C_{2n}
⋮	⋮	⋮	⋮	⋮
A_n	C_{n1}	C_{n2}	....	∞

The given travelling salesman problem is

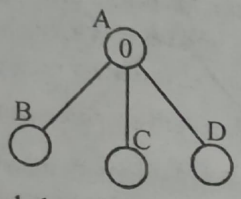
$$C = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 6 & 0 & 2 \\ 5 & \infty & 7 & 0 \\ 0 & 4 & \infty & 14 \\ 8 & 0 & 0 & \infty \end{bmatrix} \end{array}$$

As seen in the above given matrix every row and every column have zero and entire row or column have value ∞. Hence this matrix is already reduced so no further action is needed.

$$RC = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & 6 & 0 & 2 \\ 5 & \infty & 7 & 0 \\ 0 & 4 & \infty & 14 \\ 8 & 0 & 0 & \infty \end{bmatrix} \end{array}$$

Lower Bound (LB) = 0

(ii) The Construction tree for this problem will start at a root vertex of cost 0 (zero). The root node corresponds to vertex A.



Expand this node by computing path value from A to B, C and D.

Considering path $A \rightarrow B$, we formulate a matrix M by setting row A and column B to infinity (∞). Also entry [B, A] = ∞, so that the path does not track back to A.

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 7 & 0 \\ 0 & \infty & \infty & 14 \\ 8 & \infty & 0 & \infty \end{bmatrix} \end{array}$$

As seen in the above matrix every row & every column have zero and entire row & column have value ∞. Hence this matrix is already reduced so r = 0

Node value, l_b = LB + r + RC(A, B)
= 0 + 0 + 6
= 6

Now consering the path $A \rightarrow C$, for which the corresponding matrix will have row A and column C set to ∞. And the entry [C, A] = ∞, so that the path does not track back to A.

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 5 & \infty & \infty & 0 \\ \infty & 4 & \infty & 14 \\ 8 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

As seen in the above matrix the row C and column A does not have zero. So we require to subtract the minimum value from each row.

Row C = Row C – 4

Column A = Column A – 5

So reduced matrix is

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 0 \\ \infty & 0 & \infty & 10 \\ 3 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

So r = 4 + 5 = 9, them

Node value, l_C = LB + r + RC (A, C)
= 0 + 9 + 0
l_C = 9

Now considering the path $A \rightarrow D$, we have a matrix with row A and column D set to ∞, and the entry [D, A] = ∞. It is shown below :

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 5 & \infty & 7 & \infty \\ 0 & 4 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix} \end{array}$$

As seen in above matrix, the row B does not have zero. So we require to subtract the minimum value by that row, so

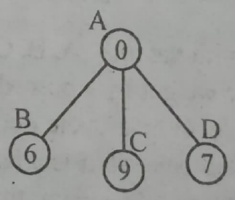
Row B = Row B – 5

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 2 & \infty \\ 0 & 4 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix} \end{array}$$

So r = 5, then

Node value, l_d = LB + r + RC(A, D)
= 0 + 5 + 2
= 7

Hence the constructed tree



At this stage we have

l_b = 6 (minimum)

l_c = 9

l_d = 7

Thus, at this stage, we find that the path (A, B) is the most promising with the minimum cost 6. Therefore, we expand the node B.

Now Taking path $A \rightarrow B \rightarrow C$, we formulate the matrix M by setting row A, row B and column C to ∞. Also the entries [B, A] and [C, A] are set to ∞. Thus

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ 0 & 4 & \infty & 14 \\ 8 & 0 & \infty & 0 \end{bmatrix} \end{array}$$

As seen the above matrix, the row C, column A and column D does not have zero, so we first subtract 4 from row C then subtract 10 from column D and then subtract 8 from column A.

So the reduced matrix is

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & 0 \\ 0 & 0 & \infty & \infty \end{bmatrix} \end{array}$$

So $r = 8 + 4 + 10 = 22$, then

Node value, $l_c = l_b + r + RC(B, C)$

$= 6 + 22 + 7$

$= 35$

Now we take the path $A \rightarrow B \rightarrow D$, we formulate the matrix M by setting row A, row B and column D to ∞. Also the entries [B, A] and [D, A] are set to ∞. Thus

$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & 4 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix} \end{array}$$

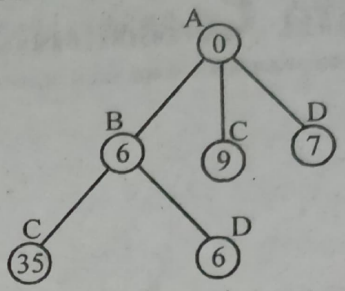
As seen in the above matrix, the matrix is already reduced. So $r = 0$, then

Node value, $l_d = l_b + r + RC(B, D)$

$= 6 + 0 + 0$

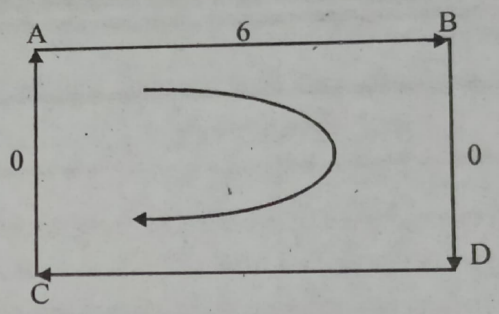
$= 6$

So the constructed tree is



Since there is no different path now to explore, we found the solution. The minimum distance node is D with value 6, therefore the solution path is

$A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$



Total cost = C[A, B] + C[B, D] + C[D, C] + C[C, A]

$= 6 + 0 + 0 + 0$

$= 6$

Note that this is same as value of last node in tree, where we found the solution.

❑❑❑