RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL

New Scheme Based On AICTE Flexible Curricula

CSE-Data Science/Data Science, V semester

CD 501- Theory of Computation

COURSE OBJECTIVES:

This course will help students to learn several formal mathematical models of computation along with their relationships with formal languages and grammars. Students will also learn about solvable and unsolvable problems.

COURSE OUTCOMES:

After completing the course student should be able to:

1. Compare and analyze different theoretical computational models, languages and grammars.
2. Design and construct finite automata, pushdown automata and Turing machine for various problems.
3. Identify limitations of some computational models and possible methods of proving them.
4. Describe the concept of computable and non-computable problems.

Unit-I

Introduction of Automata Theory: Review of Sets, Mathematical formal proofs including proof by induction and by contradiction,Introduction to languages, grammars and automata: Alphabet, Representation of language and grammar, Types of Automata, Finite Automata as a language acceptor and translator, Moore machines and mealy machines, composite machine, Conversion from Mealy to Moore and vice versa.

Unit-II

Types of Finite Automata: Non Deterministic Finite Automata (NDFA), Deterministic finite automata machines, conversion of NDFA to DFA, minimization of automata machines, regular expression, applications of regular expressions, Arden's theorem. Meaning of union, intersection, concatenation and closure, 2 way DFA.

Unit-III

Grammars: Types of grammar, context sensitive grammar, and context free grammar, regular grammar. Derivation trees, ambiguity in grammar, simplification of context free grammar,conversion of grammar to automata machine and vice versa, Chomsky hierarchy of grammar, Chomsky normal form and Greibach normal form.

Unit-IV

Push down Automata: example of PDA, deterministic and non-deterministic PDAs, Context Free Grammar, Parsing, Ambiguity, Normal form of CFGs, CFG to NPDA, NPDA to CFGs CFG equivalent to PDA, Petri nets model.

Unit-V

Turing Machine: Turing Machine as acceptor, Recognizing a Language, Universal TMs, Linear Bounded Automata, Context Sensitive Languages, Recursive and Recursively Enumerable Languages, Unrestricted Grammars. Halting problem of Turing machine & the post correspondence problem, Concept of Solvability and Unsolvability, Church's Thesis, Complexity Theory – P and NP problems.

RECOMMENDED BOOKS

1. Hopcroft, Ullman, Motwani, "Introduction to Languages, Automata and Computation", 3rd Edition, Pearson Education, 2008.
2. John C. Martin, "Introduction to Languages and the Theory of Computation", Fourth Edition, Mc Graw Hill, 2010.
3. Peter Linz, "An Introduction to Formal Languages and Automata", Sixth Edition, Jones and Bartlett, 2016.
4. Lewis and Papadimitiriou, "Elements of Theory of Computation", Second Edition, Pearson Education, 2015.
5. K.L.P. Mishra and N. Chandrasekaran, "Theory of Computer Science: Automata, Languages and Computation", Third Edition, Prentice Hall, 2006.
6. Cohen John, "Introduction to Computer Theory", Second Edition, Wiley and Sons, 2007.
7. Theory of Computation, Wood, Harper & Row.

LIST OF EXPERIMENTS

1. Design a Program for creating machine that accepts three consecutive one.

2. Design a Program for creating machine that accepts the string always ending with 101.

3. Design a Program for Mode 3 Machine

4. Design a program for accepting decimal number divisible by 2.

5. Design a program for creating a machine which accepts string having equal no. of 1's and 0's.

6. Design a program for creating a machine which count number of 1's and 0's in a given string.

7. Design a Program to find 2's complement of a given binary number.

8. Design a Program which will increment the given binary number by 1.

9. Design a Program to convert NDFA to DFA.

10. Design a Program to create PDA machine that accept the well-formed parenthesis.

11. Design a PDA to accept WCWR where w is any string and WR is reverse of that string and C is a Special symbol.

12. Design a Turing machine that's accepts the following language $a^nb^nc^n$ where n>0.

Subject Notes
CS501- Theory of Computation
B.Tech, CSE-5th Semester

Syllabus: Introduction of Automata Theory: Examples of automata machines, Finite Automata as a language acceptor and translator, Moore machines and mealy machines, composite machine, Conversion from Mealy to Moore and vice versa.

Unit-I: Introduction of Automata Theory

Automata

The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

Automata are computational devices to solve language recognition problems. Language recognition problem is to determine whether a word belongs to a language.

Automaton = abstract computing device, or "machine".

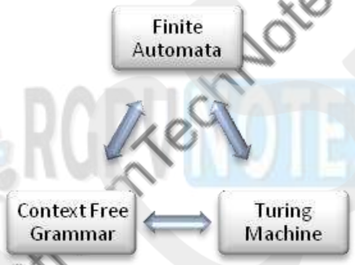


Figure 1.1: Computational Model of Automata Theory

Examples of automata machines:

- **Sequential machine:** A sequential machine is a mathematical model of a certain type of simple computational structure. Its behavior represents the working process of finite Automata.
- **Vending Machines:** A vending machine is an automated machine that dispenses numerous items such as cold drinks, snacks, beverages, alcohol etc. Vending machine is works on finite state automate to control the functions process.
- **Traffic Lights:** The optimization of traffic light controllers in a city is a systematic representation of handling the instructions of traffic rules. Its process depends on a set of instruction works in a loop with switching among instruction to control traffic.
- **Video Games:** Video games levels represent the states of automata. In which a sequence of instructions is followed by the players to accomplish the task.
- **Text Parsing:** Text parsing is a technique which is used to derive a text string using the production rules of a grammar to check the acceptability of a string.
- **Regular Expression Matching:** It is a technique to checking the two or more regular expression are similar to each other or not. The finite state machine is useful to checking out that the expressions are acceptable or not by a machine or not.
- **Speech Recognition:** Speech recognition via machine is the technology enhancement that is capable to identify words and phrases in spoken language and convert them to a machine-readable format.

Receiving words and phrases from real world and then converted it into machine readable language automatically is effectively solved by using finite state machine.

-

Characteristics:

- FA is having only a finite number of states.
- Finite Automata can only "count" (that is, maintain a counter, where different states correspond to different values of the counter) a finite number of input scenarios.
- Able to solve limited set of problem
- Outcome in the form of “yes “or “No” (Accept / Reject)

Limitation:

There is no finite automaton that recognizes these strings:

- The set of binary strings consisting of an equal number of 1's and 0's
- The set of strings over '(' and ')' that have "balanced" parentheses.

Applications of TOC:

- Finite State Programming
- Event Driven Finite State Machine (FSM)
- Virtual FSM
- DFA based text filter in Java
- Acceptors and Recognizers
- Transducers
- UML state diagrams
- Hardware Application

Finite Automata:

An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

An automaton has a mechanism to read input, which is string over a given alphabet. This input is actually written on an input tape /file, which can be read by automaton but cannot change it. Input file is divided into cells each of which can hold one symbol. Automaton has a control unit which is said to be in one of finite number of internal states.

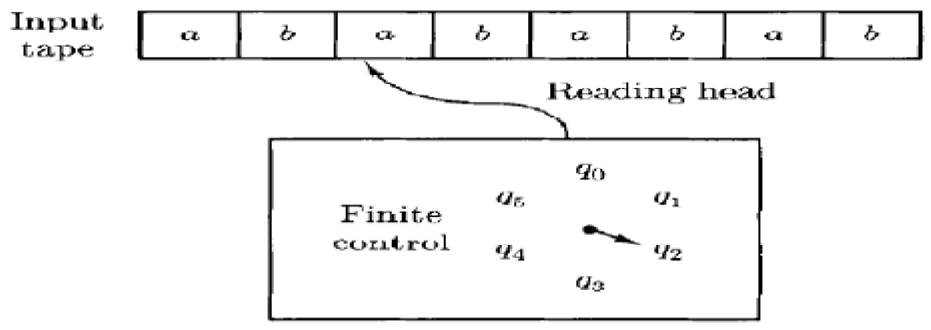


Figure 1.2: Finite Automata

Formal definition of Finite Automata:

A finite automaton is a 5-tuple M = (Q, Σ, δ, q, F), where

1. Q is a finite set, whose elements are called states,
2. Σ is a finite set, called the alphabet; the elements of Σ are called symbols,
3. $\delta: Q \times \Sigma \rightarrow Q$ is a function, called the transition function,
4. q is an element of Q; it is called the start state or initial state,
5. F is a subset of Q; the elements of F are called accept states or final state.

Related Terminologies:

Alphabet: An alphabet is any finite set of symbols.

Example: Σ = {a, b, c, d} is an alphabet set where 'a', 'b', 'c', and d' are symbols.

String: A string is a finite sequence of symbols taken from Σ.

Example: 'cabcad' is a valid string on the alphabet set Σ = {a, b, c, d}

Length of a String: It is the number of symbols present in a string. (Denoted by |S|).

Examples: If S='cabcad', |S|= 6

If |S|= 0, it is called an empty string (Denoted by λ or ε)

Language: A language is a subset of Σ* for some alphabet Σ. It can be finite or infinite.

Example: If the language takes all possible strings of length 2 over Σ = {a, b}, then L = {ab, bb, ba, bb}

Classification of Finite Automata:

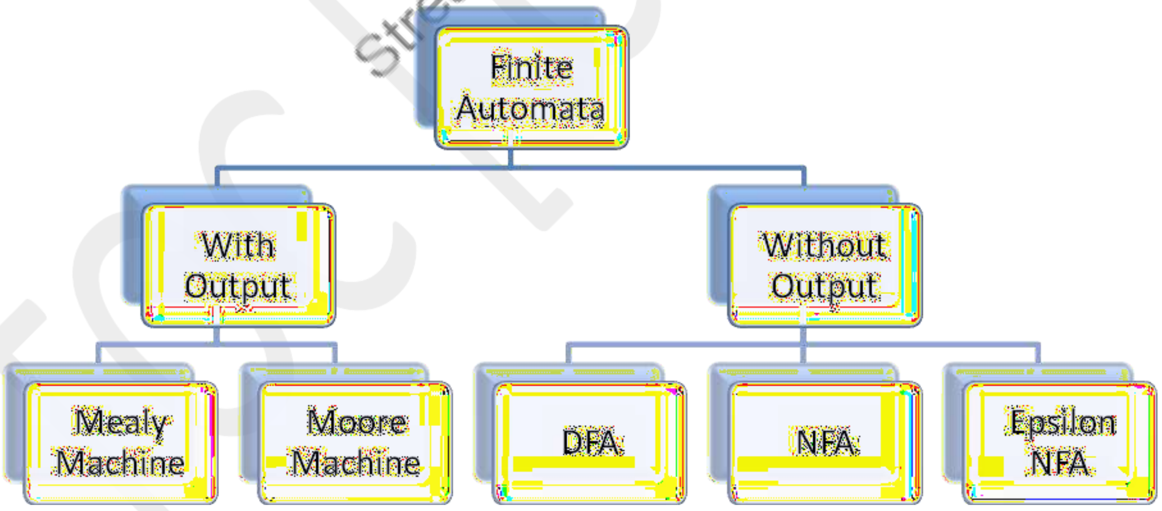


Figure 1.3: Classification of Finite Automata

An example of finite automata

Let A = {w: w is a binary string containing an odd number of 1s}.

We claim that this language A is regular. In order to prove this, we have to construct a finite automaton M such that A =L(M).

Steps to construct finite automata:

- The FA reads the input string w from left to right and keep scanning on the number of 1's
- After scanning the entire string, it counts the number of 1's to check whether it is odd or even
- If the number of 1's is odd then the string is acceptable otherwise it is rejected

Using this approach, the finite automaton needs a state for every integer i ≥ 0; indicating that the number of 1s read so far is equal to i. Hence, to design a finite automaton that follows this approach, we need an infinite number of states. But, the definition of finite automaton requires the number of states to be finite. A better, and correct approach, is to keep track of whether the number of 1s read so far is even or odd. This leads to the following finite automaton:

• The set of states is Q = {q_0, q_1}. If the finite automaton is in state q1, then it has an even number of 1's; if it is in state q0, then it has an odd number of 1's.

• The alphabet is Σ = {0, 1}.

• The start state is q0, because at the start, the number of 1's read by the automaton is equal to 0, and 0 is even.

• The set F of accept states is F = {q_1}.

• The **transition function δ** is given by the following table:

State	Input 0	Input 1
q_0	**q_0**	**q_1**
q_1	**q_1**	**q_0**

Table 1.1: Transition Table/Matrix

This finite automaton M = (Q, Σ, δ, q_0, F) can also be represented by its state diagram.

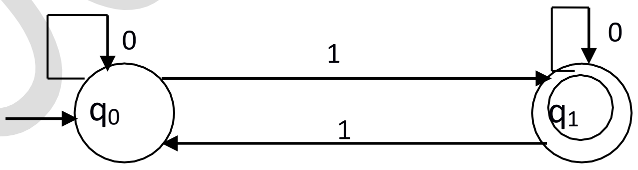


Figure 1.4: Transition Graph

Transition Graph: it is a finite directed labeled graph in which each vertex (or node) represent a state and the directed edges indicate the transition of state. Edges are labeled with input symbol.

Transition Matrix: It is two-dimension matrixes between states of automata and Input symbol. Elements of matrix are state form mapping (Σ X Q) into Q.

Finite state acceptor is a finite state machine with no outputs. The user of a finite state acceptor cares only about the final state: if the machine ends in an **accepting** state after processing a series of inputs, the machine is said to have **accepted** the input; otherwise, it is said to have **rejected** the input.

Description of Finite-State Machines using Graphs

Any finite-state machine can be shown as a graph with a finite set of nodes. The nodes correspond to the states. There is no other memory implied other than the state shown. The start state is designated with an arrow directed into the corresponding node, but otherwise unconnected.

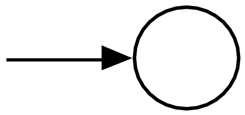


Figure 1.5: An unconnected in-going arc indicates that the node is the start state

The arcs and nodes are labeled differently, depending on whether we are representing a transducer, a classifier, or an acceptor. In the case of a **transducer**, the arcs are labeled as shown below, where q1 is the input symbol and q2 is the output symbol. The state- transition is designated by virtue of the arrow going from one node to another.

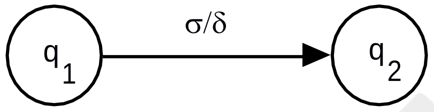


Figure 1.6: Transducer transition from q to q , based on input ⍰, giving output ⍰

In the case of an **acceptor**, instead of labeling the states with categories 0 and 1, we sometimes use a double-lined node for an accepting state and a single-lined node for a rejecting state.

Figure 1.7: Acceptor, an accepting state

Acceptor Example

Let us give an acceptor that accepts those strings with exactly one edge. We can use the state transitions from the previous classifier. We need only designate those states that categorize there being one edge as accepting states and the others as rejecting states.

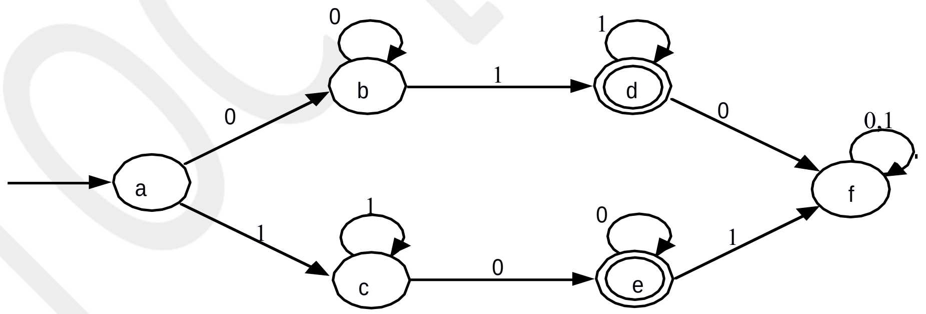


Figure 1.8: Acceptor for strings with exactly one edge. Accepting states are d and e.

Examples for Practice

Problem-01: Draw a DFA for the language accepting strings starting with 'ab' over input alphabets ∑ = {a, b}

Solution-Regular expression for the given language = ab (a + b)*

Step-01: All strings of the language start with substring "ab".

So, length of substring = 2.

Thus, Minimum number of states required in the DFA = 2 + 2 = 4.

It suggests that minimized DFA will have 4 states
Step-02: We will construct DFA for the following strings-
Ab, aba, abab
Step-03: The required DFA is-

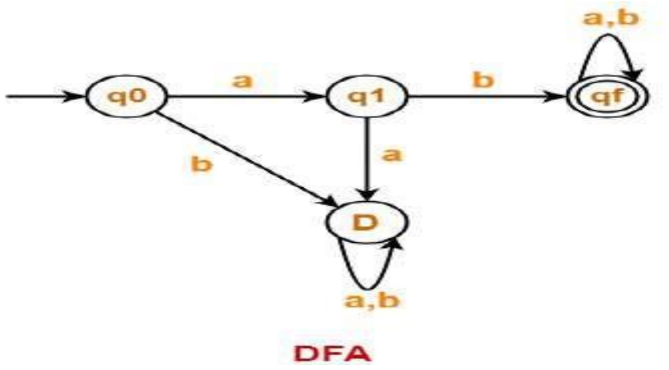


Figure 1.9: Transition Graph

Problem-02: Draw a DFA for the language accepting strings starting with 'a' over input alphabets ∑ = {a, b}
Solution-Regular expression for the given language = a(a + b)*
Step-01: All strings of the language starts with substring "a".
So, length of substring = 1.
Thus, Minimum number of states required in the DFA = 1 + 2 = 3.
It suggests that minimized DFA will have 3 states.
Step-02: We will construct DFA for the following strings-
A,aa
Step-03: The required DFA is-

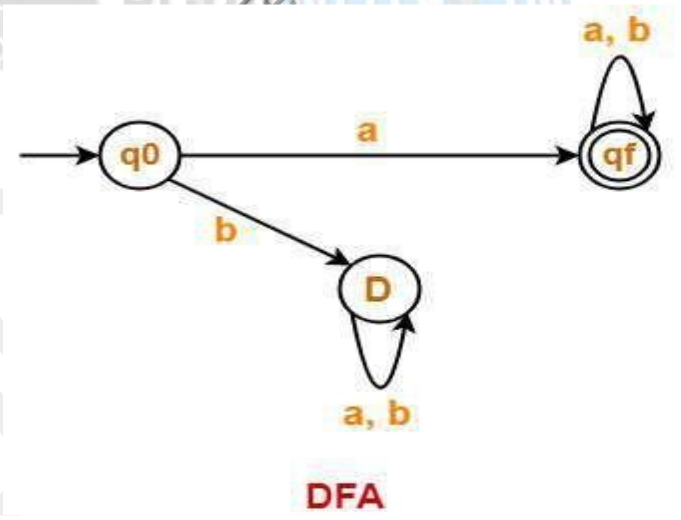


Figure 1.10: Transition Graph

Problem 3: Construct a minimal DFA, which accepts set of all strings over {0, 1}, which when interpreted as binary number is divisible by '3'. Means 110 in binary is equivalent to 6 in decimal and 6 is divisible by 3. **Answer** So if you think in the way of considering remainders if you divide by 3 that is {0, 1, 2*}*

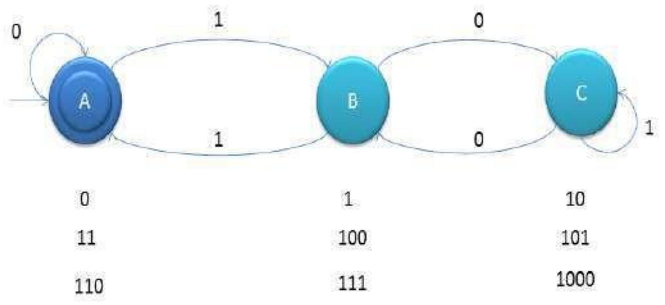


Figure 1.11: Transition Graph

As you can see that binary number which is divisible by 2 is appearing on left state. Short Trick to create such DFAs

Create Transition Table as below

Table creation rules

State	0	1
q0	q0	q1
q1	q2	q0
q2	q1	q2

- First write the input alphabets, example 0, 1
- There will n states for given number which is divisor.
- Start writing states, as for n=3: q0 under 0, q1 under 1
- Q2 under 0 and q0 under 1
- Q1under 0 and q2 under 1

If the input alphabets are 0, 1, 2 then table will expand accordingly with same rules above. Example For ternary pattern and n=4, table will be as follows;

State	*0*	*1*	*2*
q0	*q0*	*q1*	*q2*
q1	*q3*	*q0*	*q1*
q2	*q2*	*q3*	*q0*
q3	*q1*	*q2*	*q3*

Mealy and Moore Machine:

These machines are modeled to show transition and output symbol. These machines do not define a language by accepting or rejecting input string, so there is no existence of final state. Finite automata generate outputs related to each act. While two types of finite state machines create output −

- **Mealy Machine**
- **Moore machine**

Mealy Machine:

A Mealy Machine is considered as an FSM, the output will be based on the present state and the present input.
Mealy machine is explained as a 6 tuple (Q, ∑, O, δ, X, q0) where –

- Q is a finite set of states.
- ∑ is a finite set of symbols called the input alphabet.
- is a finite set of symbols called the output alphabet.
- δ is the input transition function where δ: Q × ∑ → Q
- X is the output transition function where X: Q × ∑ → O
- q0 is the initial state from where any input is processed (q0 ∈ Q).

The state table of a Mealy Machine is mentioned below –

Present state	**Next state**			
	input = 0		**input = 1**	
	State	**Output**	**State**	**Output**
→ a	b	x_1	c	x_1
b	b	x_2	d	x_3
c	d	x_3	c	x_1
d	d	x_3	d	x_2

The state diagram of the above Mealy Machine is –

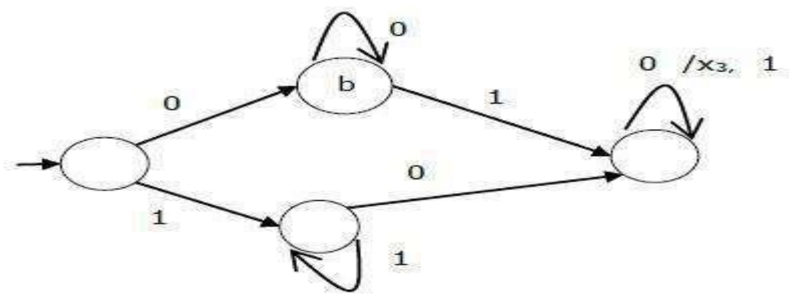


Figure 1.12: Transition Graph

Moore Machine

Moore machine is also considered as an FSM and the outputs depend on the present state.

A Moore machine is also explained by a 6 tuple (Q, ∑, O, δ, X, q0) where –

- Q is a finite set of states.
- ∑ is a finite set of symbols called the input alphabet.
- is a finite set of symbols called the output alphabet.
- δ is the input transition function where δ: Q × ∑ → Q
- X is the output transition function where X: Q → O
- q0 is the initial state from where any input is processed (q0 ∈ Q).

The state table of a Moore Machine is shown below –

Present state	**Next State**	**Output**

	Input = 0	Input = 1	
→ a	b	c	x_2
b	b	d	x_1
c	c	d	x_2
d	d	d	x_3

The state diagram of the above Moore Machine is −

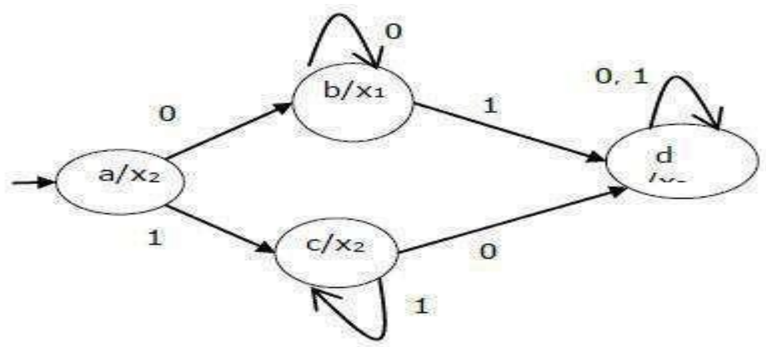


Figure 1.13: Transition Graph

Composite finite-state machines:

A finite state machine, FSM, is a box with C input/output channels, and S states, and a fixed map f:S×C→S×CU0f:S×C→S×CU0. If a state (ci,sj) (ci,sj) is mapped to the 0 element it means it enters a loop and can't exit. if we join the channels of several FSM's pair wise, we obtain a new FSM, i.e. we get a system with some number of un-joined channels, C, and internal states given by the product of the number of internal states of each component FSM, S.A composite FSM is allowed to have internal loops, i.e. we may never exit through an un-joined channel.

Conversion from Moore Machine to Mealy Machine

Algorithm
Input − Moore Machine
Output − Mealy Machine
Step 1 − Take a blank Mealy Machine transition table format.
Step 2 − Copy all the Moore Machine transition states into this table format.
Step 3 − Now check the present states and their corresponding outputs in the Moore Machine state table; if for a state Qi output is m, copy it into the output columns of the Mealy Machine state table wherever Qi appears in the next state.

Example

Let's see the following Moore machine −

Present State	Next State		Output
	a = 0	a = 1	
→ a	d	b	1

b	a	d	0
c	c	c	0
d	b	a	1

Now we apply Algorithm to convert it to Mealy Machine.

Step 1 & 2 –

Present State	**Next State**			
	a = 0		**a = 1**	
	State	**Output**	**State**	**Output**
→ a	d		b	
b	a		d	
c	c		c	
d	b		a	

Step 3 –

Present State	**Next State**			
	a = 0		**a = 1**	
	State	**Output**	**State**	**Output**
=> a	d	1	b	0
b	a	1	d	1
c	c	0	c	0
d	b	0	a	1

Conversion from Mealy Machine to Moore Machine

Algorithm

Input – Mealy Machine

Output – Moore Machine

Step 1 – Here measure the number of different outputs for each state (Qi) that are available in the state table of the Mealy machine.

Step 2 – Incase if all the outputs of Qi are same, copy state Qi. If it has n distinct outputs, break Qi into n states as Qin where n = 0, 1, 2.......

Step 3 – If the output of the initial state is 1, insert a new initial state at the beginning which gives 0 output.

Example

Let's see the following Mealy Machine –

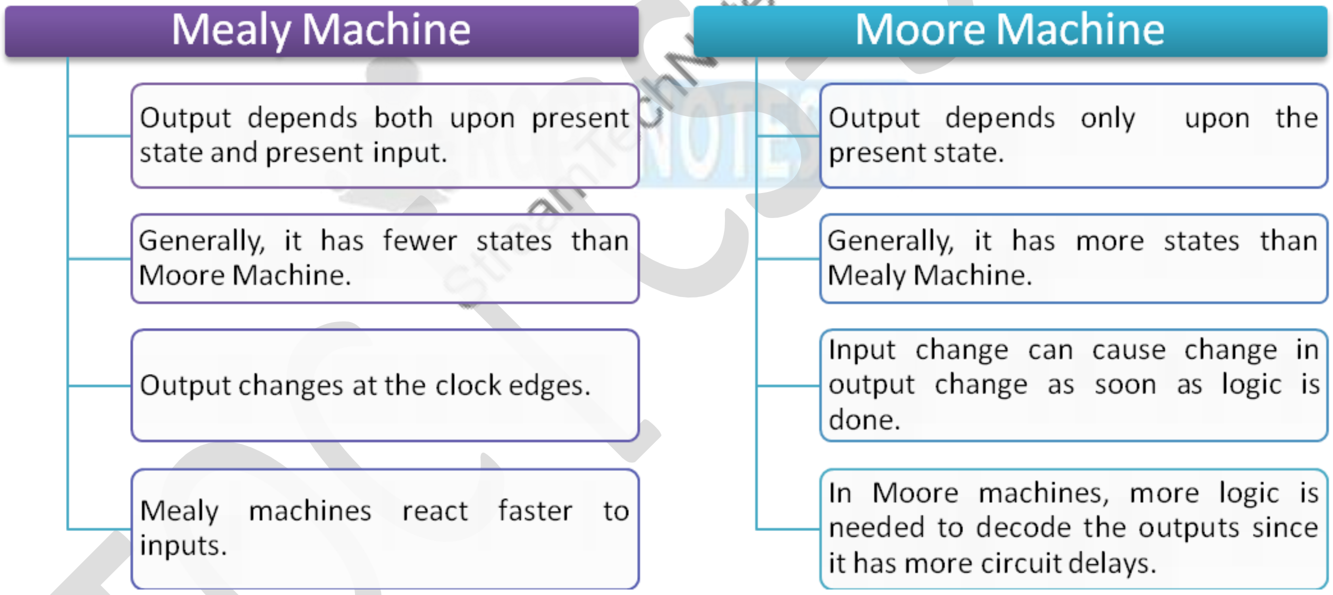
Present State	**Next State**

	a = 0		a = 1	
	Next State	**Output**	**Next State**	**Output**
→ a	d	0	b	1
b	a	1	d	0
c	c	1	c	0
d	b	0	a	1

While the states ‘a’ and ‘d’ provide only 1 and 0 outputs respectively, so we create states ‘a’ and ‘d’. But states ‘b’ and ‘c’ delivers different outputs (1 and 0). So, we divide b into b0, b1 and c into c0, c1.

Present State	**Next State**		**Output**
	a = 0	**a = 1**	
→ a	d	b_1	1
b_0	a	d	0
b_1	a	d	1
c_0	c_1	C_0	0
c_1	c_1	C_0	1
d	b_0	a	0

Difference between Mealy and Moore Machine:



Mealy Machine

Output depends both upon present state and present input.

Generally, it has fewer states than Moore Machine.

Output changes at the clock edges.

Mealy machines react faster to inputs.

Moore Machine

Output depends only upon the present state.

Generally, it has more states than Mealy Machine.

Input change can cause change in output change as soon as logic is done.

In Moore machines, more logic is needed to decode the outputs since it has more circuit delays.

Figure 1.12: Difference between Mealy & Moore Machine

Subject Notes
CS501- Theory of Computation

B. Tech, CSE-5th Semester

Unit -2

Syllabus: Types of Finite Automata: Non-Deterministic Finite Automata (NDFA), Deterministic finite automata machines, conversion of NDFA to DFA, minimization of automata machines, regular expression, Arden's theorem. Meaning of union, intersection, concatenation and closure, 2-way DFA.

Unit Objective: Relate practical problems to languages, automata, computability and complexity. Constructs abstract models of computing and check their power to recognize the language.

……….

Finite Automata: An automaton with a finite number of states is called a Finite Automaton (FA) or FiniteState Machine (FSM).

An automaton has a mechanism to read input, which is string over a given alphabet. This input isactually written on an input tape /file, which can be read by automaton but cannot change it. Input file is divided into cells each of which can hold one symbol. Automaton has a control unitwhich is said to be in one of finite number of internal states.

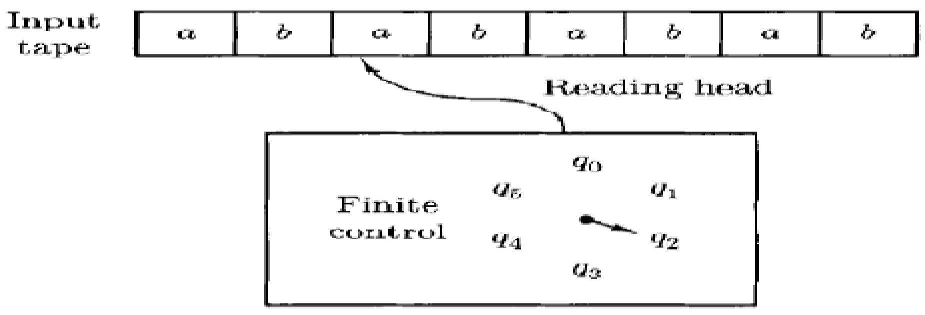


Figure 2.1: Finite Automata

Formal definition of Finite Automata:

A finite automaton is a 5-tuple M = (Q, Σ, δ, q, F), where

1. Q is a finite set, whose elements are called states,
2. Σ is a finite set, called the alphabet; the elements of Σ are called symbols,
3. δ: Q × Σ → Q is a function, called the transition function,
4. q is an element of Q; it is called the start state or initial state,
5. F is a subset of Q; the elements of F are called accept states or final state.

Related Terminologies:

Alphabet: An alphabet is any finite set of symbols.

Example: Σ = {a, b, c, d} is an alphabet set where 'a', 'b', 'c', and'd' are symbols.

String: A string is a finite sequence of symbols taken from Σ.

Example: 'cabcad' is a valid string on the alphabet set Σ = {a, b, c, d}

Length of a String: It is the number of symbols present in a string. (Denoted by |S|).

Examples: If S='cabcad', |S|= 6

If |S|= 0, it is called an empty string (Denoted by λ or ε)

Language: A language is a subset of Σ* for some alphabet Σ. It can be finite orinfinite.

Example: If the language takes all possible strings of length 2 over Σ = {a, b},then L = {ab, bb, ba, bb

1.1 Types of Finite Automata:

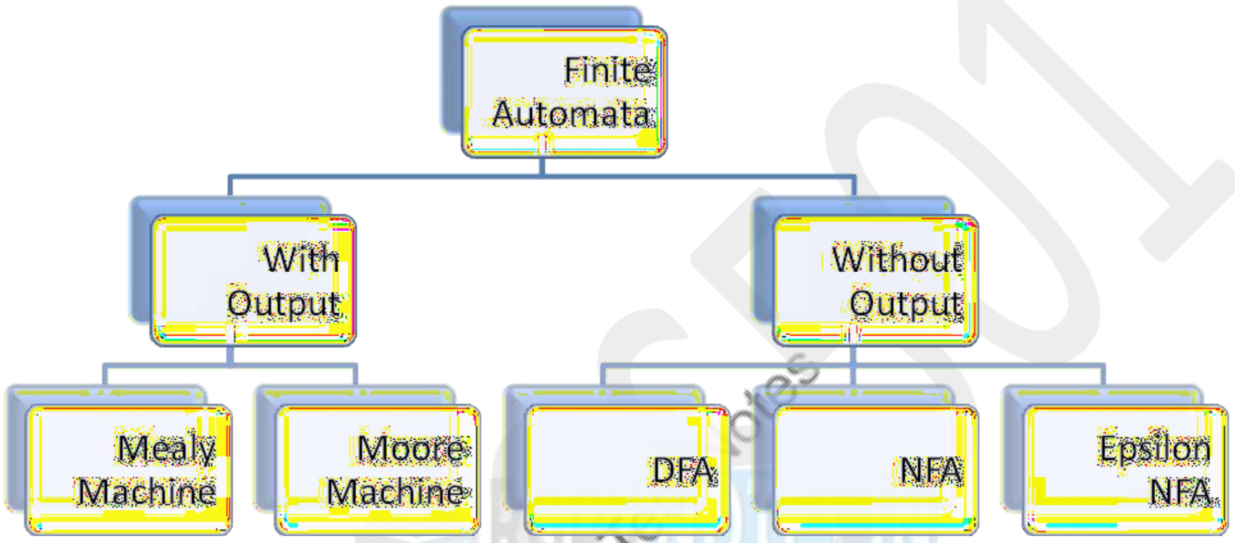


Figure 2.2: Classification of Finite Automata

An example of finite automata

Let A = {w: w is a binary string containing an odd number of 1s}.

We claim that this language A is regular. In order to prove this, we have to construct a finite automaton M such that A =L(M).

Steps to construct finite automata:

- The FA reads the input string w from left to right and keep scanning on the number of 1's
- After scanning the entire string, it counts the number of 1's to check whether it is odd or even
- If the number of 1's is odd then the string is acceptable otherwise it is rejected

Using this approach, the finite automaton needs a state for every integer i ≥ 0; indicating that the number of 1s read so far is equal to i. Hence, to design a finite automaton that follows this approach, we need an infinite number of states. But, the definition of finite automaton requires the number of states to be finite. A better, and correct approach, is to keep track of whether the number of 1s read so far is even or odd. This leads to the following finite automaton:

• The set of states is Q = {q_0, q_1}. If the finite automaton is in state q1, then it has an even number of 1's; if it is in state q0, then it has an odd number of 1's.

• The alphabet is Σ = {0, 1}.

• The start state is q0, because at the start, the number of 1's read by the automaton is equal to 0, and 0 is even.

• The set F of accept states is F = {q_1}.

• The **transition function δ** is given by the following table:

State	Input 0	Input 1
q_0	**q_0**	**q_1**
q_1	**q_1**	**q_0**

Table 2.1: Transition Table/Matrix

This finite automaton M = (Q, Σ, δ, q_0, F) can also be represented by its state diagram.

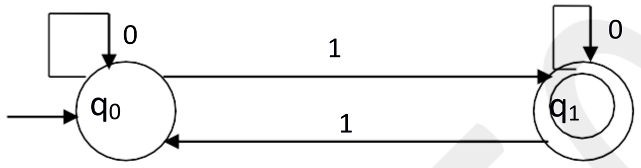


Figure2.3: Transition Graph

Transition Graph: it is a finite directed labeled graph in which each vertex (or node) represent a state and the directed edges indicate the transition of state. Edges are labeled with input symbol.

Transition Matrix: It is two-dimension matrixes between states of automata and Input symbol. Elements of matrix are state form mapping (Σ X Q) into Q.

Deterministic Finite Automata (DFA):

Deterministic automaton is one in which each move (transition from one state to another) is uniquelydetermined by the current configuration. If the internal states input and contents of the storage are known it is possible to predict the next (future) behavior of the automaton.

Formal Definition of a DFA

A DFA can be represented by **a 5-tuple (Q, Σ, δ, q0, F)** where:

1. Q is a finite set of states.
2. Σ is a finite set of symbols called the alphabet.
3. δ is the transition function where δ: Q × Σ → Q
4. q0 is the initial state from where any input is processed (q0 ∈ Q).
5. F is a set of final state/states of Q (F ⊆ Q).

DFA can be represented by Transition Graph and Transition Diagram as shown in Table 1.1 and Figure 1.4

Non-Deterministic Finite Automata (NDFA):

In NDFA, for a particular input symbol, the machine can move to any combination of thestates. In other words, the exact state to which the machine moves cannotbe determined. Hence, it is called Non-deterministic Automaton.

Formal Definition of NDFA

An NDFA can be represented by a **5-tuple (Q, Σ, δ, q0, F)** where:
1. Q is a finite set of states.
2. Σ is a finite set of symbols called the alphabets.
3. δ is the transition function where δ: Q × Σ → 2Q

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

4. q0 is the initial state from where any input is processed (q0 ∈ Q).
5. F is a set of final state/states of Q (F ⊆ Q).

Difference between DFA & NDFA:

S. No.	DFA	NFA
1.	DFA stands for deterministic finite automata.	NDFA stands for non-deterministic finite automata.
2.	when processing a string in DFA , there is always a unique state to go next when each character is read. It is because for each state in DFA , there is exactly one state that corresponds to each character being read.	In NDFA several choices may exist for the next state. Can move to more than one states.
3.	DFA can not use empty string transition.	NDFA can use empty string transition.
4.	In DFA we cannot move from one state to another without consuming a symbol.	NDFA allows € (null) as the second argument of the transition function. This means that the NDFA can make a transition without consuming an input symbol.
5	For every symbol of the alphabet, there is only one state transition in DFA.	We do not need to specify how does the NDFA react according to some symbol.
6	DFA can understood as one machine.	NDFA can be understood as multiple title machines computing at the same time.
7	DFA will reject the string if it end at other than accepting state	If all the branches of NDFA dies or rejects the string, we can say that NDFA reject the string.
8	It is more difficult to construct DFA.	NDFA is easier to construct.
9	DFA requires more space.	NDFA requires less space.
10	For every input and output we can construct DFA machine.	It is not possible to construct an NDFA machine for every input and output.

Figure 2.4: Difference between DFA & NDFA

Comparison between Deterministic Finite Automata (DFA) and the Nondeterministic Finite Automata (NFA):

S. No.	Title	NFA	DFA

1.	Power	Same	Same
2.	Supremacy	Not all NFA are DFA.	All DFA are NFA
3.	Transition Function	Maps Q → (∑U{λ}→2^Q), the number of next states is zero or one or more.	Q × ∑→Q, the number of next states is exactly one
4.	Time complexity	The time needed for executing an input string is more as compare to DFA.	The time needed for executing an input string is less as compare to NFA.
5.	Space	Less space requires.	More space requires.

Table No. 2.2 Comparison between NFA and DFA

Equivalence of DFA and NDFA:

Although the DFA and NFA have distinct definitions, an NFA can be translated to equivalent DFA using the subset construction algorithm. i.e., the constructed DFA and the NFA recognize the same formal language. Both types of automata recognize only regular languages.

Therefore, every language that can be described by some NDFA can also be described by some DFA.

Theorem

Let M = (Q, Σ, δ, q_0, F) be a non-deterministic finite automaton. There exists a deterministic finite automaton M', such that L(M') =L(M).

i.e. for every NDFA there exists a DFA which simulates the behavior of NDFA. Hence if language L is accepted by N9DFA, then there exist a DFA M' which also accept L. Where M' = (Q', Σ, δ', q'_0, F')

Conversion from NFA to DFA:

In NFA, when a specific input is given to the current state, the machine goes to multiple states. It can have zero, one or more than one move on a given input symbol. On the other hand, in DFA, when a specific input is given to the current state, the machine goes to only one state. DFA has only one move on a given input symbol.

Let, M = (Q, ∑, δ, q0, F) is an NFA which accepts the language L(M). There should be equivalent DFA denoted by M' = (Q', ∑', q0', δ', F') such that L(M) = L(M').

Steps for converting NFA to DFA:

Step 1: Initially Q' = φ

Step 2: Add q0 of NFA to Q'. Then find the transitions from this start state.

Step 3: In Q', find the possible set of states for each input symbol. If this set of states is not in Q', then add it to Q'.

Step 4: In DFA, the final state will be all the states which contain F (final states of NFA)

Example 1:

Convert the given NFA to DFA.

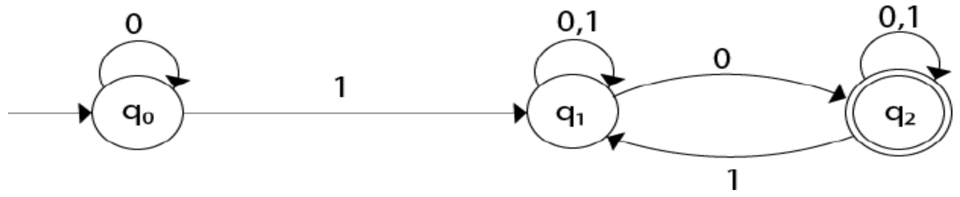


Figure No. 2.5 Transition Graph

Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

Table No. 2.3 Transition Table

Now we will obtain δ' transition for state q0.

1. δ'([q0], 0) = [q0]
2. δ'([q0], 1) = [q1]

The δ' transition for state q1 is obtained as:

1. δ'([q1], 0) = [q1, q2] (new state generated)
2. δ'([q1], 1) = [q1]

The δ' transition for state q2 is obtained as:

1. δ'([q2], 0) = [q2]
2. δ'([q2], 1) = [q1, q2]

Now we will obtain δ' transition on [q1, q2].

1. δ'([q1, q2], 0) = δ (q1, 0) U δ (q2, 0)
2. = {q1, q2} U {q2}
3. = [q1, q2]
4. δ'([q1, q2], 1) = δ (q1, 1) U δ (q2, 1)
5. = {q1} U {q1, q2}
6. = {q1, q2}
7. = [q1, q2]

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]

*[q1, q2]	[q1, q2]	[q1, q2]

Table No. 2.4 Transition Table

The Transition diagram will be:

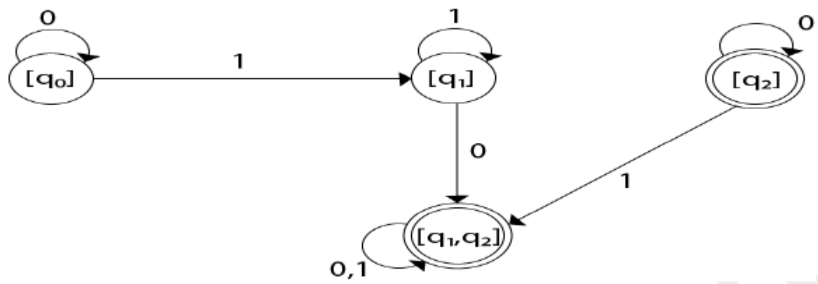


Figure No. 2.6 Transition Graph

The state q2 can be eliminated because q2 is an unreachable state.

Minimization of DFA

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM(finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. 1. δ (q, a) = p
2. 2. δ (r, a) = p

That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

Example:

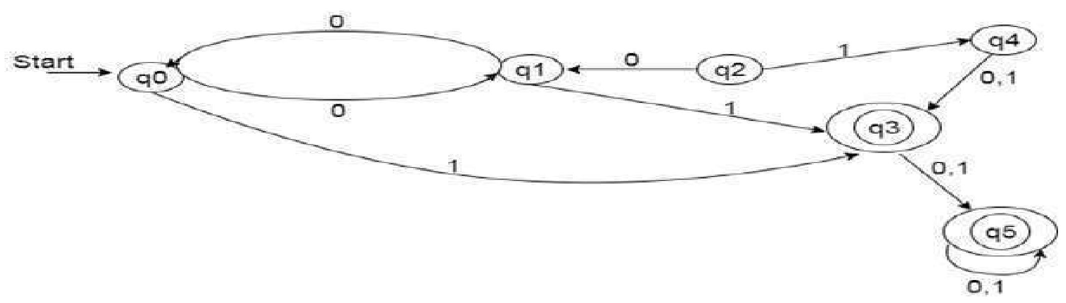


Figure 2.7: Transition Graph

Solution: Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

State	**0**	**1**
→q0	q1	q3
q1	q0	q3
***q3**	q5	q5
***q5**	q5	q5

Table 2.5: Transition Table/Matrix

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

State	**0**	**1**
q0	q1	q3
q1	q0	q3

Table 2.6: Transition Table/Matrix

2. Another set contains those rows, which starts from final states.

State	**0**	**1**
q3	q5	q5
q5	q5	q5

Table 2.7: Transition Table/Matrix

Step 4: Set 1 has no similar rows so set 1 will be the same.

Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

State	0	1
q3	q3	q3

Table 2.8: Transition Table/Matrix

Step 6: Now combine set 1 and set 2 as:

State	0	1
→q0	q1	q3
q1	q0	q3
***q3**	q3	q3

Table 2.9: Transition Table/Matrix

Now it is the transition table of minimized DFA.

Regular Expression

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expression are referred to as Regular languages.
- A regular expression can also be described as a sequence of pattern that defines a string.
- Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

For instance:

In a regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx, ...}

In a regular expression, x^+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx,..... }

Operations on Regular Language

Properties of Regular Sets

Property 1. The union of two regular set is regular.

Proof −

Let us take two regular expressions

$RE_1 = a(aa)^*$ and $RE_2 = (aa)^*$

So, L_1 = {a, aaa, aaaaa, ... } (Strings of odd length excluding Null)

and L_2 ={ ε, aa, aaaa, aaaaaa, ... } (Strings of even length including Null)

L_1 U L_2 = { ε, a, aa, aaa, aaaa, aaaaa, aaaaaa, ... }

(Strings of all possible lengths including Null)

RE (L_1 U L_2) = a* (which is a regular expression itself)

Property 2. The intersection of two regular set is regular.

Proof –

Let us take two regular expressions

RE_1 = a(a*) and RE_2 = (aa)*

So, L_1 = { a,aa, aaa, aaaa, ... } (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

$L_1 \cap L_2$ = { aa, aaaa, aaaaaa,.......} (Strings of even length excluding Null)

RE ($L_1 \cap L_2$) = aa(aa)* which is a regular expression itself.

Property 3. The complement of a regular set is regular.

Proof –

Let us take a regular expression –

RE = (aa)*

So, L = {ε, aa, aaaa, aaaaaa, ... } (Strings of even length including Null)

Complement of **L** is all the strings that is not in **L**.

So, L' = {a, aaa, aaaaa, ...} (Strings of odd length excluding Null)

RE (L') = a(aa)* which is a regular expression itself.

Property 4. The difference of two regular set is regular.

Proof –

Let us take two regular expressions –

RE_1 = a (a*) and RE_2 = (aa)*

So, L_1 = {a, aa, aaa, aaaa, ... } (Strings of all possible lengths excluding Null)

L_2 = { ε, aa, aaaa, aaaaaa,.......} (Strings of even length including Null)

$L_1 – L_2$ = {a, aaa, aaaaa, aaaaaaa, ... }

(Strings of all odd lengths excluding Null)

RE ($L_1 – L_2$) = a (aa)* which is a regular expression.

Property 5. The reversal of a regular set is regular.

Proof −

We have to prove L^R is also regular if **L** is a regular set.

Let, L = {01, 10, 11, 10}

RE (L) = 01 + 10 + 11 + 10

L^R = {10, 01, 11, 01}

RE (L^R) = 01 + 10 + 11 + 10 which is regular

Property 6. The closure of a regular set is regular.

Proof −

If L = {a, aaa, aaaaa, ... } (Strings of odd length excluding Null)

i.e., RE (L) = a (aa)*

L* = {a, aa, aaa, aaaa , aaaaa, ... } (Strings of all lengths excluding Null)

RE (L*) = a (a)*

Property 7. The concatenation of two regular sets is regular.

Proof −

Let RE_1 = (0+1)*0 and RE_2 = 01(0+1)*

Here, L_1 = {0, 00, 10, 000, 010, ... } (Set of strings ending in 0)

and L_2 = {01, 010,011, ...} (Set of strings beginning with 01)

Then, L_1 L_2 = {001,0010,0011,0001,00010,00011,1001,10010, ... }

Set of strings containing 001 as a substring which can be represented by an RE − (0 + 1)*001(0 + 1)*

Identities Related to Regular Expressions

Given R, P, L, Q as regular expressions, the following identities hold −

- ∅* = ε
- ε* = ε
- RR* = R*R
- R*R* = R*
- (R*)* = R*
- RR* = R*R
- (PQ)*P =P(QP)*
- (a+b)* = (a*b*)* = (a*+b*)* = (a+b*)* = a*(ba*)*

- R + ∅ = ∅ + R = R (The identity for union)
- R ε = ε R = R (The identity for concatenation)
- ∅ L = L ∅ = ∅ (The annihilator for concatenation)
- R + R = R (Idempotent law)
- L (M + N) = LM + LN (Left distributive law)
- (M + N) L = ML + NL (Right distributive law)
- ε + RR* = ε + R*R = R*

Note: Two regular expressions are equivalent if languages generated by them are same. For example, (a+b*) * and (a+b) * generate same language. Every string which is generated by (a+b*) * is also generated by (a+b)* and vice versa.

Example 1:

Write the regular expression for the language accepting all combinations of a's, over the set ∑ = {a}

Solution:

All combinations of a's mean a may be zero, single, double and so on. If a is appearing zero times, that means a null string. That is, we expect the set of {ε, a, aa, aaa, ... }. So, we give a regular expression for this as:

1. R = a*

That is Kleene closure of a.

Example 2:

Write the regular expression for the language accepting all combinations of as except the null string, over the set ∑ = {a}

Solution:

The regular expression has to be built for the language

1. L = {a, aa, aaa, ... }

This set indicates that there is no null string. So, we can denote regular expression as:

R = a^+

Example 3:

Write the regular expression for the language accepting all the string containing any number of a's and b's.

Solution:

The regular expression will be:

1. r.e. = (a + b) *

This will give the set as L = {ε, a, aa, b, bb, ab, ba, aba, bab ... }, any combination of a and b.

The (a + b) * shows any combination with a and b even a null string.

Conversion of RE to FA

To convert the RE to FA, we are going to use a method called the subset method. This method is used to obtain FA from the given regular expression. This method is given below:

Step 1: Design a transition diagram for given regular expression, using NFA with ε moves.

Step 2: Convert this NFA with ε to NFA without ε.

Step 3: Convert the obtained NFA to equivalent DFA.

Example 1:

Design a FA from given regular expression 10 + (0 + 11)0* 1.

Solution: First we will construct the transition diagram for a given regular expression.

Step 1:

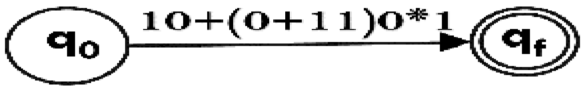


Figure 2.8: Transition Graph

Step 2:

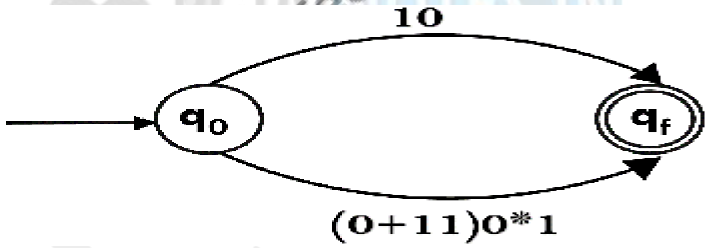


Figure 2.9: Transition Graph

Step 3:

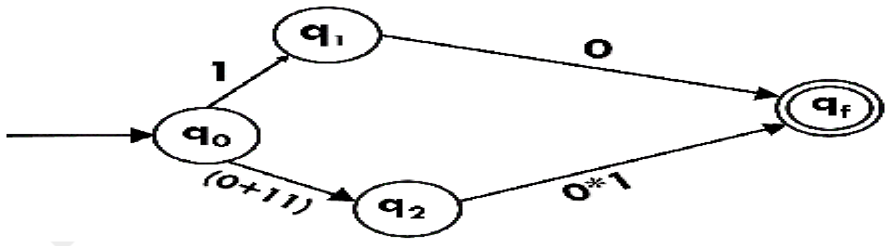


Figure 2.10: Transition Graph

Step 4:

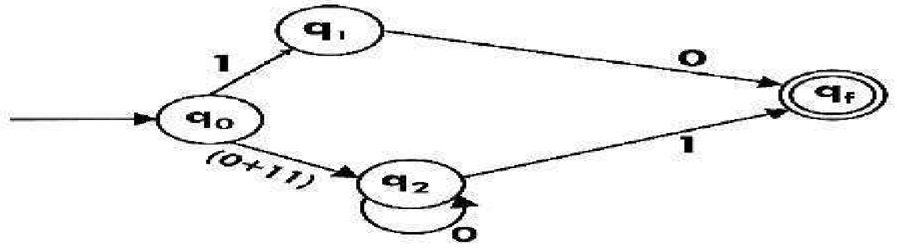


Figure 2.11: Transition Graph

Step 5:

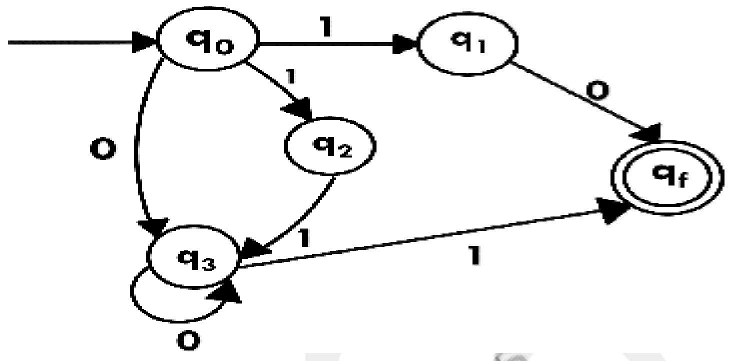


Figure 2.12: Transition Graph

Now we have got NFA without ε. Now we will convert it into required DFA for that, we will first write a transition table for this NFA.

State	0	1
→q0	q3	{q1, q2}
q1	qf	φ
q2	φ	q3
q3	q3	qf
***qf**	φ	φ

Table 2.9: Transition Table/Matrix

The equivalent DFA will be:

State	0	1
→[q0]	[q3]	[q1, q2]
[q1]	[qf]	φ
[q2]	φ	[q3]
[q3]	[q3]	[qf]
[q1, q2]	[qf]	[qf]
***[qf]**	φ	φ

Table 2.10: Transition Table/Matrix

Construction of an FA from a RE

Method

Step 1 Construct an NFA with Null moves from the given regular expression.

Step 2 Remove Null transition from the NFA and convert it into its equivalent DFA.

Problem

Convert the following RA into its equivalent DFA − 1 (0 + 1) * 0

Solution

We will concatenate three expressions "1", "(0 + 1) *" and "0"

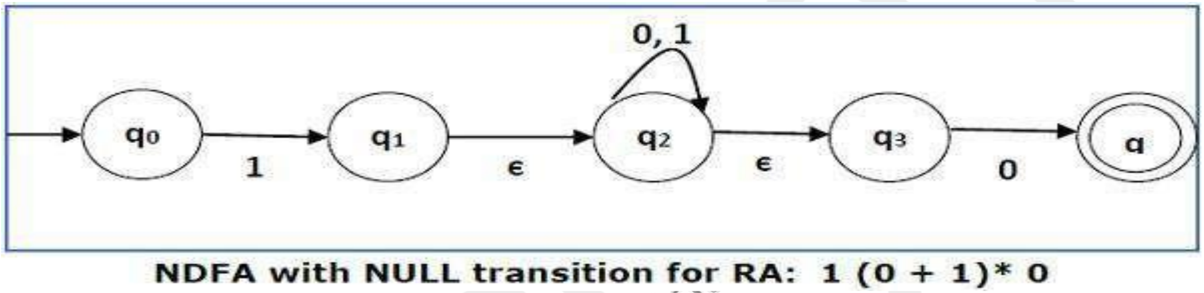


Figure 2.13: Transition Graph

Now we will remove the **ε** transitions. After we remove the **ε** transitions from the NDFA, we get the following −

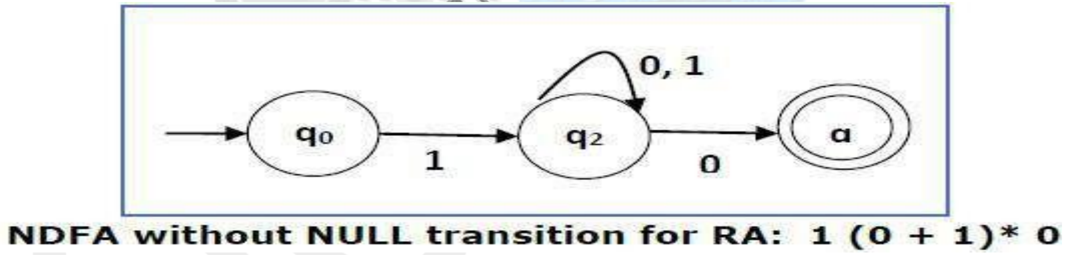


Figure 2.14: Transition Graph

It is an NDFA corresponding to the RE − 1 (0 + 1) * 0. If you want to convert it into a DFA, simply apply the method of converting NDFA to DFA

Applications:

- Regular expressions are useful in a wide variety of text processing tasks, and more generally string processing, where the data need not be textual. Common applications include data validation, data scraping (especially web scraping), data wrangling, simple parsing, the production of syntax highlighting systems, and many other tasks.
- While regexps would be useful on Internet search engines, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex.

Arden's Theorem:
In order to find out a regular expression of a Finite Automaton, we use Arden's Theorem along with the properties of regular expressions.

Statement:

Let P and Q be two regular expressions.
If P does not contain null string, then R = Q + RP has a unique solution that is R = QP*

Proof:
R = Q + (Q + RP) P [After putting the value R = Q + RP]
R= Q + QP + RPP
When we put the value of R recursively again and again, we get the following equation:
R = Q + QP + QP2 + QP3….
R = Q (ϵ + P + P2 + P3 + ….)
R = QP* [As P* represents (ϵ + P + P2 + P3 + ….)]
Hence, proved.

Assumptions for Applying Arden's Theorem:
1. The transition diagram must not have NULL transitions
2. It must have only one initial state

Following algorithm is used to build the regular expression form given DFA.
1. Let q_1 be the initial state.
2. There are q_2, q_3, q_4 …….. qn number of states. The final state may be some q_j where j<= n.
3. Let α_{ji} represents the transition from q_j to q_i.
4. Calculate q_i such that
$q_i = \alpha_{ji} * q_j$
If q_j is a start state then we have:
$q_i = \alpha_{ji} * q_j + \varepsilon$
5. Similarly, compute the final state which ultimately gives the regular expression 'r'.

Example:
Construct the regular expression for the given DFA

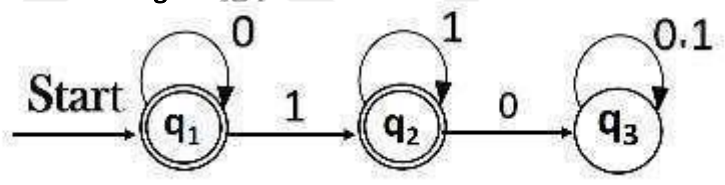


Figure 2.15: Transition Graph

Solution:
Let us write down the equations
q1 = q1 0 + ε
Since q1 is the start state, so ε will be added, and the input 0 is coming to q1 from q1 hence we write State = source state of input × input coming to it
Similarly,
q2 = q1 1 + q2 1
q3 = q2 0 + q3 (0+1)
Since the final states are q1 and q2, we are interested in solving q1 and q2 only. Let us see q1 first
q1 = q1 0 + ε
We can re-write it as
q1 = ε + q1 0
Which is similar to R = Q + RP, and gets reduced to R = OP*.
Assuming R = q1, Q = ε, P = 0
We get

$q1 = \varepsilon. (0) *$
$q1 = 0* (\varepsilon.R*= R*)$
Substituting the value into q2, we will get
$q2 = 0* 1 + q2\ 1$
$q2 = 0* 1 (1) * (R = Q + RP \rightarrow Q P*)$

The regular expression is given by
$r = q1 + q2$
$= 0^* + 0^* 1.1^*$
$r = 0^* + 0^* 1^+ (1.1^* = 1^+)$

Two-way finite automata

The finite automata discussed so far has a δ transition which indicate where to next from current state on receiving particular input. But 2-way FA is a model in which linear direction is mentioned on receiving particular and being in some current state. There are two direction that are allowed in 2-FA and these are left and right direction.

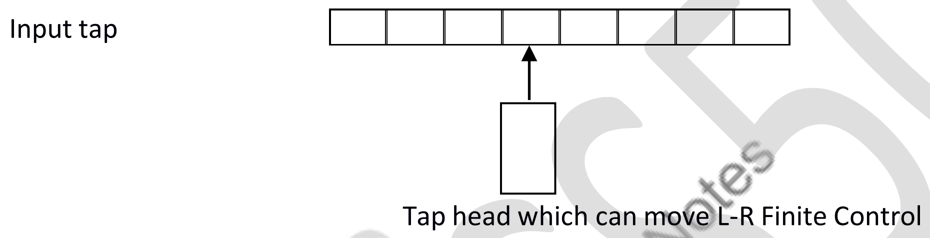


Figure 2.16: Two-way finite automata

Formal Definition of Two-way finite automata
It is a collection of 5-tuple M = (Q, Σ, δ, q_0, F), where
- Q: a finite set of states
- Σ: a finite set called the input alphabet
- δ: a transition function which maps Q x {L,R}
- q_0: a start state (also called initial state) which is an element of Q ($q_0 \in Q$)
- F: Set of final states.

Example :

Consider the transition table .check the string "101001" whether it is accepted by Two - Way DFA or not?

States	0	1
-> q0	(q0, R)	(q1, R)
q1 (Final State)	(q1, R)	(q2, L)
q2	(q0, R)	(q2, L)

Table 2.11: Transition Table/Matrix

Acceptability of the string using two - way DFA is –

$$\begin{aligned}
q_0 101001 &\vdash 1q_1 01001 \\
&\vdash 10q_1 1001 \\
&\vdash 1q_2 01001 \\
&\vdash 10q_0 1001 \\
&\vdash 101q_1 001 \\
&\vdash 1010q_1 01 \\
&\vdash 10100q_1 1 \\
&\vdash 1010q_2 01 \\
&\vdash 10100q_0 1 \\
&\vdash 101001q_1
\end{aligned}$$

Unit III: Grammars:

3.1 Grammars:

Grammars denote syntactical rules for conversation in natural languages. Linguistics has attempted to define grammars since the inception of natural languages like English, Sanskrit, Mandarin, etc.
In formal language also grammar defines the rule or syntax or structure of the language.

Example: "Dog runs"

<Sentence> -> <Noun> <Verb>
< Noun > -> < Dog >
<Verb> -> <Run>

Formal Definition of Grammar:

A grammar G can be formally written as a 4-tuple (V, T, S, P) where −

- VN is a set of variables or non-terminal symbols.
- T or ∑ is a set of Terminal symbols.
- S is a special variable called the Start symbol, S ∈ V
- P is Production rules for Terminals and Non-terminals.

A production rule has the form $\alpha \rightarrow \beta$, where α and β are strings on VN U ∑ and least one symbol of α belongs to VN.
Example:

Grammar G1 −
({S, A, B}, {a, b}, S, {S → AB, A → a, B → b})
Here:

S, A, and B are Non-terminal symbols;
a and b are Terminal symbols
S is the Start symbol, S ∈ N
Productions, P : S → AB, A → a, B → b

3.2 Types of Grammar (Chomsky Classification):

Type 0 Unrestricted/ Phase structured/ Recursively Enumerable
Type 1 Context Sensitive Grammar (CSG)
Type 2 Context Free Grammar (CFG)
Type 3 Regular Grammar

Type 0: Unrestricted/ Phase structured/ Recursively Enumerable:

- The productions have no restrictions.
- The productions can be in the form of α → β where α is a string of terminals and non-terminals with at least one non-terminal and α cannot be null.
- β is a string of terminals and non-terminals.

Type 1: Context Sensitive Grammar (CSG):

- First of all Type 1 grammar should be Type 0.
- The productions can be in the form of α → β & with restriction that |α| <= |β | that is count of symbol in Left side of the production:|α| is less than or equal to Right side: |β|

Type 2: Context Free Grammar (CFG):

- First of all it should be Type 1.
- The productions can be in the form of α → β & with restriction that Left hand side of production can have only one variable (Non-Terminal).| α | = 1.
- There is no restriction on β it can be (V U T)*.

Type 3: Regular Grammar:

- First of all it should be Type 2.
- It is most restricted form of grammar. It should be in the given form only :
  V –> VT* / T* or V –> T*V /T*
- It can of two types either left linear or right linear

3.2.1 Chomsky hierarchy and classification:

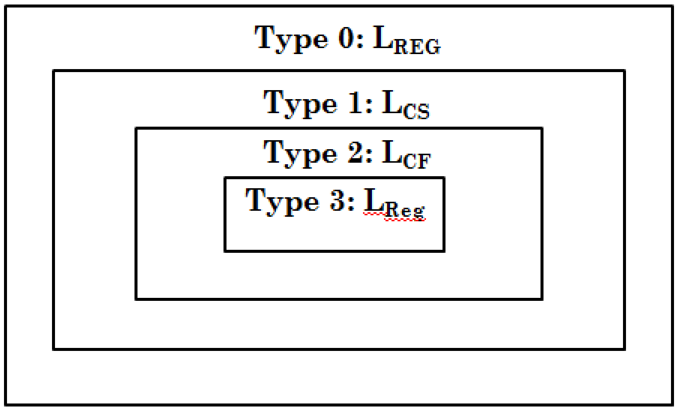


Fig: 3.1 Chomsky hierarchy.

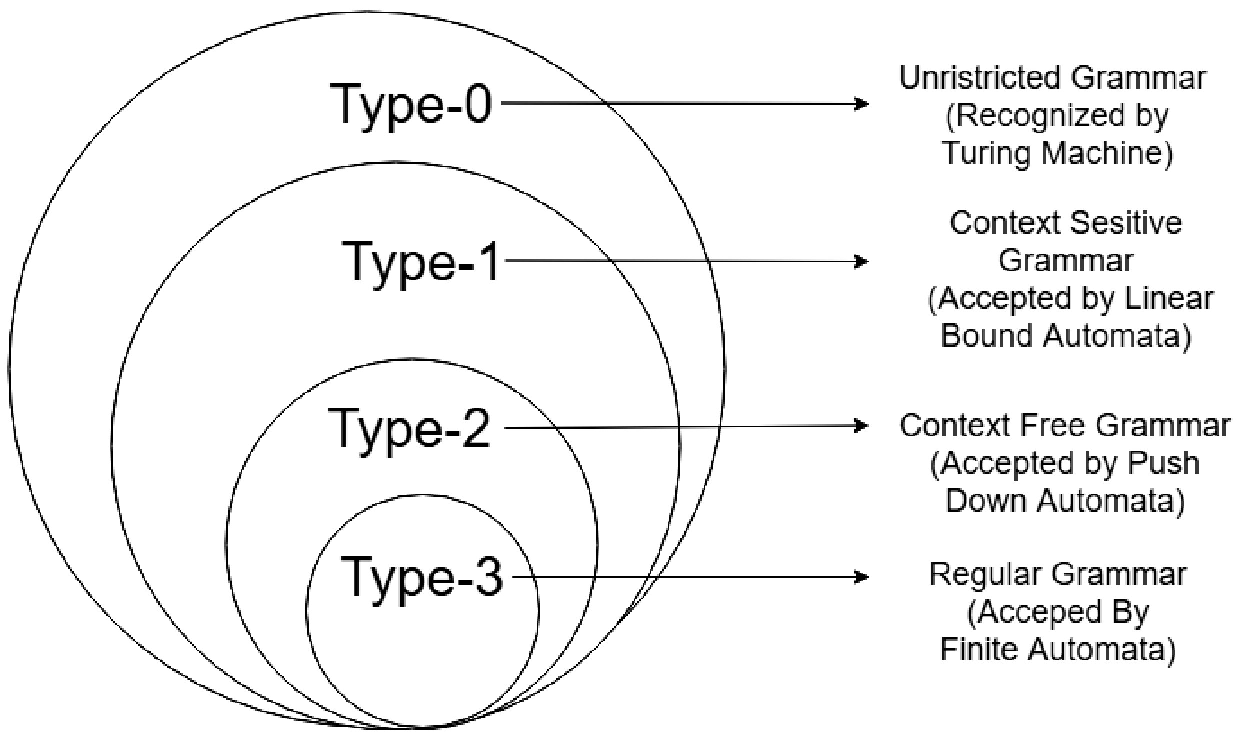

Type-0
Type-1
Type-2
Type-3
Unristricted Grammar (Recognized by Turing Machine)
Context Sesitive Grammar (Accepted by Linear Bound Automata)
Context Free Grammar (Accepted by Push Down Automata)
Regular Grammar (Acceped By Finite Automata)

Fig: 3.2 Chomsky Classifications

3.3 Derivation

To infer any string by replacing variable using productions of given CFG is known as derivation. Consider CFG G = (V, T, P , S) and αAβ be a string of terminals and variables with A is variable from V and α, β are string of terminals and variables. Lets there is production A → γ in grammar G then there will be derivation αAβ => α γ β.

When derivation begins from start symbol S and end in string of terminals T then these inferred string of terminal is language of Grammars G.

Example: Grammar: Arith. Exp

E----> E + E
E----> E * E
E----> (E)
E ----> a | b | c

Here is a sequence of replacements that leads to a sequence of terminal symbols.

LMD (Left Most Derivative):

E ===> E * E ===> (E) * E ===> (E + E) * E ===> (a + E) * E ===> (a + b) * E ===> (a + b) * c

In this case, we obtained the final sentence (a + b) * c starting from E. One says that "E derives (a + b) * c" or (to use passive voice) "(a + b) * c is derived from E". The sequence is called a derivation sequence. In this case it is a leftmost derivation sequence, because at each stage the leftmost non-terminal was replaced. Each non-terminal that is replaced is colored red above. Sometimes there was only one non-terminal, so it is leftmost on an honorary basis.)

RMD (Right Most Derivative):

E ===> E * E ===> E * c ===> (E) * c ===> (E + E) * c ===> (E + b) * c ===> (a + b) * c
This was a rightmost derivation sequence, because at each stage the rightmost non-terminal was replaced.

Parse tree:

The sentence (a + b) * c has a unique leftmost derivation, a unique (different) rightmost derivation and the unique parse tree shown below:

```
Parse Tree: ( a + b ) * c
------------------------------------
    E      |          E
   /|\     |         /|\
  E * E    |        / | \
  /|\ \    |       /  |  \
 ( E ) c   |      /   |   \
   /|\     |     /    |    \
  E + E    |    E     |     E
  |   |    |   /|\    |     |
  a   b    |  / | \   |     |
           | /  E  \  |     |
           |/  /|\  \ |     |
           |  / | \  ||     |
           |  E | E  ||     |
           |  | | |  ||     |
           |( a + b )*      c
```

Fig 3.3 Parse Tree

3.4 Ambiguity:

There are other sentences derived from E above that have more than one parse tree, and corresponding left- and rightmost derivations. Example: the very simple sentence a + b * c. The table looks at leftmost derivations and parse trees:

```
1st Leftmost Der.    2nd Leftmost Der.

E ===> E + E         E ===> E * E
  ===> a + E           ===> E + E * E
  ===> a + E * E       ===> a + E * E
  ===> a + b * E       ===> a + b * E
  ===> a + b * c       ===> a + b * c

1st Parse Tree       2nd Parse Tree

     E                    E
    /|\                  /|\
   / | \                / | \
  E  +  E              E  *  E
  |    /|\            /|\    |
  |   / | \          / | \   |
  a  E  *  E         E  +  E  c
     |     |         |     |
     b     c         a     b
```

Fig 3.4 Parse Tree showing Ambiguity

- Even if some parse trees are unique, if there are multiple parse trees for any sentence, then the grammar is called ambiguous.
- In a programming language it is not acceptable to have more than one possible reading of a construct. We can't flip a coin to decide which parse tree to use. There are several ways around this problem:
    1. Rewrite the grammar so that it is no longer ambiguous yet still accepts exactly the same language. This is not always possible.
    2. Introduce extra rules that allow the program to decide which of multiple parse trees to use. These are called disambiguating rules.
- An ambiguous grammar may signal problems with language design, and the programming language itself might be changed.

3.5 Simplification of CFG:

- As we have seen, various languages can efficiently be represented by a context-free grammar. All the grammars are not always optimized that means the grammar may consist of some extra symbols (non-terminal).
- Grammars having extra symbols, unnecessary increase the length of grammar.
- Simplification of grammar means reduction of grammar by removing useless symbols.

The properties of reduced grammar are given below:

- Each variable (i.e. non-terminal) and each terminal of G appears in the derivation of some word in L.
- There should not be any production as X → Y where X and Y are non-terminal.
- If ε is not in the language L then there need not to be the production X → ε.

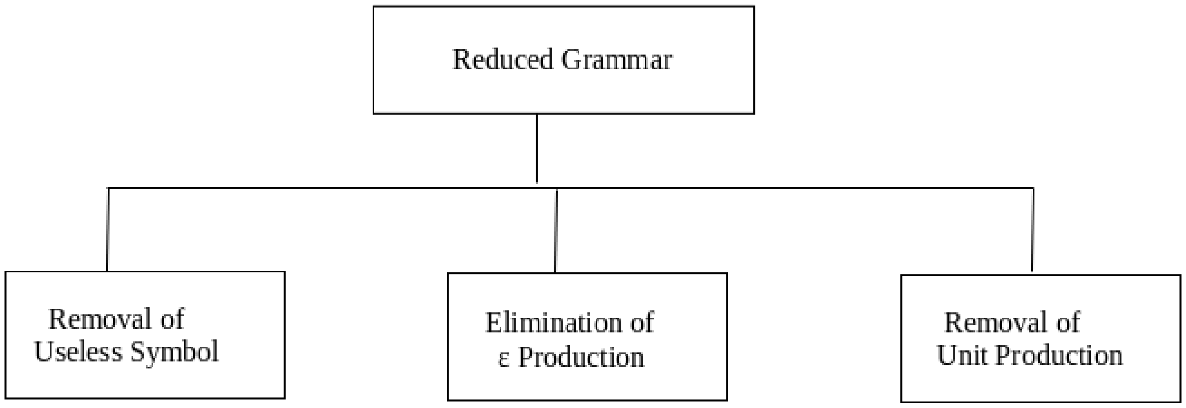

Reduced Grammar
Removal of Useless Symbol
Elimination of ε Production
Removal of Unit Production

Fig 3.5 Simplification (Reduction) of Grammar

3.5.1 Elimination of Useless production/symbols from context free grammar:
Conditions of Useless Symbol:
1) We will entitle any variable useful only when it is deriving any terminal.
2) Also if a symbol is deriving a terminal but not reachable from Start state.
3) All terminals will be useful symbols

Example: Remove Useless Productions/Symbols :

S -> AB/a
A -> BC/b
B -> aB/C
C -> aC/B

Solution:

- Useful Symbols: {a, b, S, A}

And any combination of useful symbols will also make LHS a useful symbol.
So we could see that Symbol B and C are useless symbol, remove them (whole production in which it contains):

S -> a
A -> b

Since A is not reachable so we will remove A -> b as well:

S -> a

One more example to remove useless:

S -> AB/AC
A -> aAb/bAa/a
B -> bbA/aaB/AB
C -> abCA/aDb
D -> bD/aC

Solution:

First find out useful Symbols: {a, b, A, B, S}
And useless symbols are: {C, D}
So remove them and write the whole grammar again:
S -> AB
A -> aAb/bAa/a
B -> bbA/aaB/AB

3.5.2 Elimination of null production from context free grammar

If ε belongs to the language then we are supposed to generate it and thus we will not remove it. Using below example we will understand the whole concept.

Example 1

S -> aSb/aAb/ab/a
A -> ε

- To know whether ε is generated in the CFG or not, we find all variable which are generating ε.
- So only A is genrating ε Thus ε does not belong to the language.

Now we will proceed with elimination of NULL production:

- Replace NULL producing symbol with and without in R.H.S. of remaining states and drop the productions which have ε directly. eg. A -> ε

S -> aSb/aAb/ab/ab/a   But we no need to write "ab" twice

So,

S -> aSb/aAb/ab/a

Example 2 Remove Null productions

S -> AB
A -> aAA/ε
B -> bBB/ε

Solution: Nullale Variables are {A, B, S}
Because start state also a Nullable symbol so ε belongs to given CFG
We will proceed with the method:

S -> AB/A/B/ε

A -> aAA/aA/a

B -> bAA/bA/b

3.5.3 Elimination of Unit production from context free grammar:

A Unit production is like below:

S -> B means V -> V that is a single Variable (non-terminal) producing another single Variable (non-terminal).

Steps to remove Unit production:

- Write production without Unit production
- Check what we are missing because of Step 1

Example:

Given grammar is :

S -> Aa/B/c

B -> A/bb

A -> a/bc/B

Remove unit Productions

Solution:

Now we will apply step 1:

S -> Aa/c

B -> bb

A -> a/bc

Now check what we are missing after applying Step 1:

First : S -> B -> bb

And : S -> B -> A -> a

And : S -> B -> A -> bc

So add these in the production list of "S"

S -> Aa/c/bb/a/bc

B -> bb

A -> a/bc

Second : B -> A -> a

And : B -> A -> bc

So add this in the prodcution list of "B"

S -> Aa/c/bb/a/bc

B -> bb/a/bc

A -> a/bc

Third: A-> B -> bb

So add this in the prodcution list of "A"

S -> Aa/c/bb/a/bc

B -> bb/a/bc

A -> a/bc/bb

One more example with 1 variation:

S -> AC

A -> a

C -> B/d

B -> D

D -> E

E -> b

Solution:

Now apply step 1:

S -> AC

A -> a

C -> b

Now check what are we missing after applying Step 1:

For C: C -> B -> D -> E -> b

So add this in the production of "C"

S -> AC

A -> a

C -> d/b

For B: B -> D -> E -> b

So add this in the production of "B"

S -> AC

A -> a

C -> d/b

B -> b

Similarily for D and E also add:

S -> AC

A -> a

C -> d/b

B -> b

D -> b

E -> b

But if we see that B -> b, D -> b and E -> b

Productions are useless as they can't be reached, Remove them:

S -> AC

A -> a

C -> d/b

3.6 Conversion of Grammar to Finite Automata:

Steps for converting grammar to Finite Automata are:

- Start from the first production
- And then for every left alphabet go to SYMBOL followed by it
- Start State: It will be the first production's state
- Final State: Take those states which end up with input alphabets. eg. State A and C are below CFG

Example: Here we are giving one right linear grammar

A -> aB/bA/b

B -> aC/bB

C -> aA/bC/a

now see the output and you will understand what just happened.

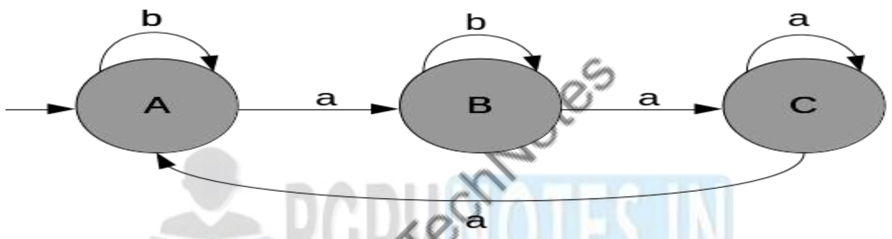


Fig 3.5 Grammar to automata

Explanation:

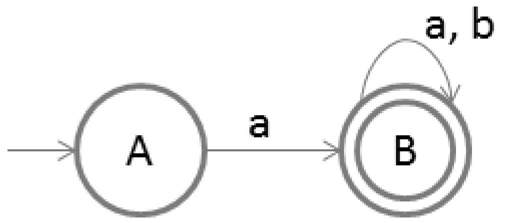
- As you can see for state A, transition on 'a' is going on state B and on 'b' it is going on state A itself. This method will apply to all the states.
- Note: Final states are A and C, cause they are having terminal production

3.7 Conversion of Finite Automata to Grammar:

Steps for converting finite automata to grammar:

- Repeat the process for every state
- Begin the process from start state
- Write the production as the output followed by the state on which the transition is going
- And at the last add ε because that's is required to end the derivation
- Note: We have added ε because either you could continue the derivation or would like to stop it. So to stop the derivation we have written ε

Example:


$$\begin{tikzcd} \arrow[r] & A \arrow[r, "a"] & B \arrow[loop above, "{a, b}"] \end{tikzcd}$$

Solution: So we are applying the above procedure
Pick start state and output is on symbol 'a' we are going on state B
So we will write as :

A -> aB

And then we will pick state B and then we will go for each output.
so we will get the below production.

B -> aB/bB/ε

So final we got right linear grammar as:

A -> aB
B -> aB/bB/ε

3.8 Chomsky's Normal Form (CNF):

CNF stands for Chomsky normal form. A CFG(context free grammar) is in CNF(Chomsky normal form) if all production rules satisfy one of the following conditions:

- Start symbol generating ε. For example, A → ε.
- A non-terminal generating two non-terminals. For example, S → AB.
- A non-terminal generating a terminal. For example, S → a.

Steps for converting CFG into CNF:

Step 1: In the grammar, remove the null, unit and useless productions. You can refer to the Simplification of CFG.
Step 2: Eliminate terminals from the RHS of the production if they exist with other non-terminals or terminals. For example, production S → aA can be decomposed as:
S → RA
R → a
Step 3: Eliminate RHS with more than two non-terminals. For example, S → ASB can be

decomposed as:

S → RS

R → AS

Example: Convert given grammar into CNF

S -> bA/aB

A -> bAA/aS/a

Solution:

Now we have to add new production:

S -> NA/MB

A -> NAA/MS/a

N -> a

M -> b

Now combine NAA with either NA or AA, So

S -> NA/MB

A -> NO/MS/a

N -> a

M -> b

O -> AA

3.9 Greibach Normal Form (GNF):

GNF stands for Greibach normal form. A CFG(context free grammar) is in GNF (Greibach normal form) if all the production rules satisfy one of the following conditions:

- A start symbol generating ε. For example, S → ε.
- A non-terminal generating a terminal. For example, A → a.
- A non-terminal generating a terminal which is followed by any number of non-terminals. For example, S → aASB.

For example:

G1 = {S → aAB | aB, A → aA| a, B → bB | b}

G2 = {S → aAB | aB, A → aA | ε, B → bB | ε}

- The production rules of Grammar G1 satisfy the rules specified for GNF, so the grammar G1 is in GNF.
- However, the production rule of Grammar G2 does not satisfy the rules specified for GNF as A → ε and B → ε contains ε(only start symbol can generate ε). So the grammar G2 is not in GNF.

Steps for converting CFG into GNF:

Step 1: Convert the grammar into CNF.

If the given grammar is not in CNF, convert it into CNF. You can refer the following topic to convert the CFG into CNF: Chomsky normal form

Step 2: If the grammar exists left recursion, eliminate it.

If the context free grammar contains left recursion, eliminate it. You can refer the following topic to eliminate left recursion: Left Recursion

Step 3: In the grammar, convert the given production rule into GNF form.

If any production rule in the grammar is not in GNF form, convert it.

Example: Convert CFG to GNF

S → XB | AA
A → a | SA
B → b
X → a

Solution:

As the given grammar G is already in CNF and there is no left recursion, so we can skip step 1 and step 2 and directly go to step 3.

The production rule A → SA is not in GNF,
so we substitute S → XB | AA in the production rule A → SA as:

S → XB | AA
A → a | XBA | AAA
B → b
X → a

The production rule S → XB and B → XBA is not in GNF, so we substitute X → a in the production rule S → XB and B → XBA as:

S → aB | AA
A → a | aBA | AAA
B → b
X → a

Now we will remove left recursion (A → AAA), we get:

S → aB | AA
A → aC | aBAC
C → AAC | ε
B → b
X → a

Now we will remove null production C → ε, we get:

S → aB | AA
A → aC | aBAC | a | aBA
C → AAC | AA

B → b
X → a
The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule S → AA as:
S → aB | aCA | aBACA | aA | aBAA
A → aC | aBAC | a | aBA
C → AAC
C → aCA | aBACA | aA | aBAA
B → b
X → a
The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule C → AAC as:
S → aB | aCA | aBACA | aA | aBAA
A → aC | aBAC | a | aBA
C → aCAC | aBACAC | aAC | aBAAC
C → aCA | aBACA | aA | aBAA
B → b
X → a
Hence, this is the GNF form for the grammar G.

Subject Notes
CS501- Theory of Computation
B. Tech, CSE-5th Semester

Syllabus: Push down Automata: example of PDA, deterministic and non-deterministic PDA, conversion of PDA into context free grammar and vice versa, CFG equivalent to PDA, Petri net model.

Objective: The goal is to make a pushdown automaton that will accept all of the input strings that the context-free grammar accepts.

Unit-IV: Push down Automata:

A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically, a pushdown automaton is: "Finite state machine" + "a stack".

A pushdown automaton has three components:

- An input tape,
- A control unit, and
- A stack with infinite size.

The stack head scans the top symbol of the stack.

A stack does two operations:

- Push: a new symbol is added at the top.
- Pop: the top symbol is read and removed.

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.

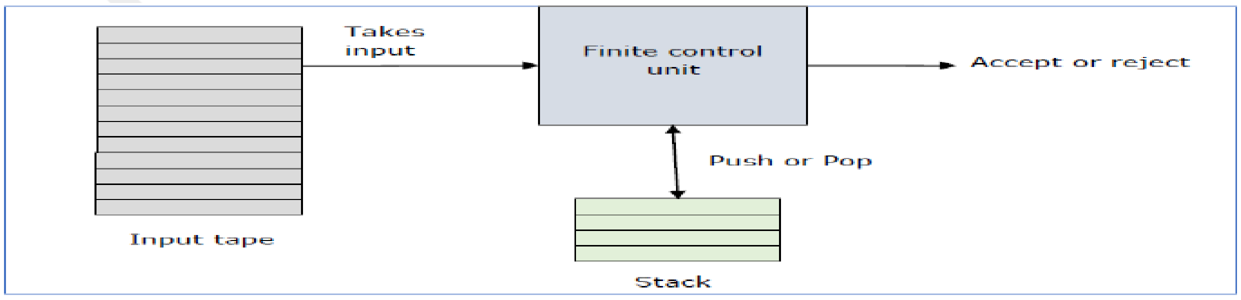


Figure 4.1: Pushdown Automata

Applications of PDA:

1. Online Transaction process system.
2. Used in compiler design (parser design for syntactic analysis)
3. Tower of Hanoi (Recursive Solution)

Formal Definition of PDA:

A deterministic pushdown automaton is a 7 -tuples M = (Q, Σ, Γ, δ, q0, Z0, F), where

- Q is a finite set of states,
- Σ is a finite set of tape alphabet or input symbol
- Γ is a finite set of stack alphabet
- q0 is initial state, q0 is an element of Q
- Z0 is Initial symbol on top of stack
- F set of final state which is sub set of Q.
- δ is transition function which maps (Q x Σ x Γ) into Q x Γ*

Transition of PDA:

Transition function of PDA is denoted as δ (q, a, X) = (p, Y) which specified transition in PDA is function of three components:

1. Present state of PDA, q
2. Symbol of input alphabet, a, being read by PDA.
3. Symbol at top of stack, X.

In every transition

- PDA enters into new state p or remain into same state p = q
- if a = ε than no symbol is consumed
- If Y = zX, symbol z is pushed into the stack at the top.
- If Y = ε, Symbol X at the top of stack is popped.

- If Y = X, No change of symbol at top of stack.

Example: The following diagram shows a transition in a PDA from a state q_1 to state q_2, labeled as "a,b/c"

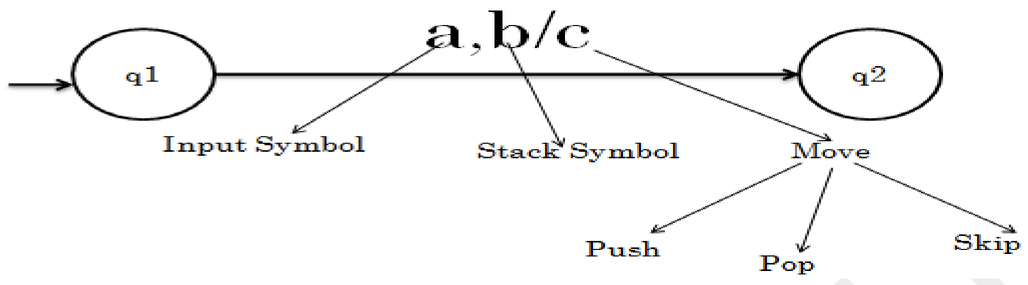


Figure 4.2: Example of PDA

Representation of PDA:

Corresponding to transition function δ (q, a, X) = (p, Y) transition diagram will have

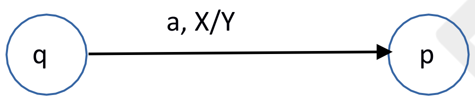


Figure 4.3: Representation of PDA

Terminologies related to PDA:

Instantaneous Description:

The instantaneous description (ID) of a PDA is represented by a triplet (q, w, s) where

- q is the state
- w is unconsumed input
- s is the stack contents

Turnstile Notation:

The "turnstile" notation is used for connecting pairs of ID's that represent one or many moves of a PDA. The process of transition is denoted by the turnstile symbol "⊢".

Consider a PDA (Q, Σ, S, δ, q0, I, F). A transition can be mathematically represented by the following turnstile notation:

$$(p, aw, T\beta) \vdash (q, w, \alpha b)$$

This implies that while taking a transition from state p to state q, the input symbol 'a' is consumed, and the top of the stack 'T' is replaced by a new string 'α'.

Note: If we want zero or more moves of a PDA, we have to use the symbol (⊢*) for it.

Example: Define the pushdown automata for language {anbn | n > 0}

Solution: M = where Q = {q0, q1} and Σ = {a, b} and Γ = {A, Z} and &delta is given by

&delta (q0, a, Z) = {(q0, AZ)}

&delta (q0, a, A) = {(q0, AA)}

&delta (q0, b, A) = {(q1, ∈)}

&delta (q1, b, A) = {(q1, ∈)}

&delta (q1, ∈, Z) = {(q1, ∈)}

Let us see how this automaton works for aabb

Deterministic PDA:

A PDA is said to be deterministic if and only if following conditions are met

1. δ (q, a, X) has at most one transition.
2. δ (q, ε, X) = Φ

Acceptance of PDA:

There are two different ways to define PDA acceptability.

Final State Acceptability:

In final state acceptability, a PDA accepts a string when, after reading the entire string, the PDA is in a final state. From the starting state, we can make moves that end up in a final state with any stack values. The stack values are irrelevant as long as we end up in a final state.

For a PDA (Q, Σ, S, δ, q0, I, F), the language accepted by the set of final states F is:

$$L(PDA) = \{w \mid (q0, w, I) \vdash^* (q, \varepsilon, x), q \in F\}$$

for any input stack string x.

Empty Stack Acceptability:

Here a PDA accepts a string when, after reading the entire string, the PDA has emptied its stack.

For a PDA (Q, Σ, S, δ, q0, I, F), the language accepted by the empty stack is:

$$L(PDA) = \{w \mid (q0, w, I) \vdash^* (q, \varepsilon, \varepsilon), q \in Q\}$$

Example: Construct a PDA that accepts L= $\{0^n1^n \mid n \geq 0\}$

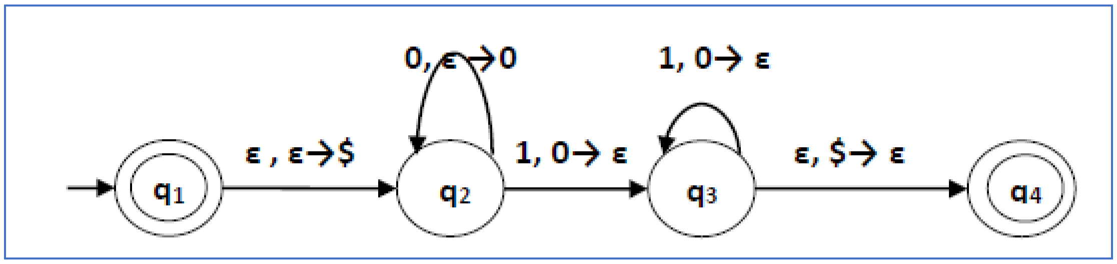


Figure 4.4: Example of PDA

This language accepts L = {ε, 01, 0011, 000111, ... }

Here, in this example, the number of ‘a’ and ‘b’ have to be same.

- Initially we put a special symbol '$' into the empty stack.
- Then at state q2, if we encounter input 0 and top is Null, we push 0 into stack. This may iterate. And if we encounter input 1 and top is 0, we pop this 0.
- Then at state q3, if we encounter input 1 and top is 0, we pop this 0. This may also iterate. And if we encounter input 1 and top is 0, we pop the top element.
- If the special symbol '$' is encountered at top of the stack, it is popped out and it finally goes to the accepting state q4.

Example: Construct a PDA that accepts the language L over {0, 1} by empty stack which accepts all the string of 0's and 1's in which a number of 0's are twice of number of 1's.

Solution: There are two parts for designing this PDA:

- If 1 comes before any 0's
- If 0 comes before any 1's.

We are going to design the first part i.e. 1 comes before 0's. The logic is that read single 1 and push two 1's onto the stack. Thereafter on reading two 0's, POP two 1's from the stack. The δ can be

1. δ (q0, 1, Z) = (q0, 11, Z) Here Z represents that stack is empty
2. δ (q0, 0, 1) = (q0, ε)

Now, consider the second part i.e. if 0 comes before 1's. The logic is that read first 0, push it onto the stack and change state from q0 to q1. [Note that state q1 indicates that first 0 is read and still second 0 has yet to read].

Being in q1, if 1 is encountered then POP 0. Being in q1, if 0 is read then simply read that second 0 and move ahead. The δ will be:

1. δ (q0, 0, Z) = (q1, 0Z)
2. δ (q1, 0, 0) = (q1, 0)
3. δ (q1, 0, Z) = (q0, ε) (indicate that one 0 and one 1 is already read, so simply read the second 0)
4. δ (q1, 1, 0) = (q1, ε)

Now, summarize the complete PDA for given L is:

1. δ (q0, 1, Z) = (q0, 11Z)
2. δ (q0, 0, 1) = (q1, ε)
3. δ (q0, 0, Z) = (q1, 0Z)
4. δ (q1, 0, 0) = (q1, 0)
5. δ (q1, 0, Z) = (q0, ε)
6. δ (q0, ε, Z) = (q0, ε) ACCEPT state

Non-deterministic Pushdown Automata

The non-deterministic pushdown automata are very much similar to NFA. We will discuss some CFGs which accepts NPDA.

The CFG which accepts deterministic PDA accepts non-deterministic PDAs as well. Similarly, there are some CFGs which can be accepted only by NPDA and not by DPDA. Thus, NPDA is more powerful than DPDA.

Example: Design PDA for Palindrome strips.

Solution: Suppose the language consists of string L = {aba, aa, bb, bab, bbabb, aabaa, ...]. The string can be odd palindrome or even palindrome. The logic for constructing PDA is that we will push a symbol onto the stack till half of the string then we will read each symbol and then perform the pop operation. We will compare to see whether the symbol which is popped is similar to the symbol which is read. Whether we reach to end of the input, we expect the stack to be empty.

This PDA is a non-deterministic PDA because finding the mid for the given string and reading the string from left and matching it with from right (reverse) direction leads to non-deterministic moves. Here is the ID.

1. δ(q1, a, Z) = (q1, aZ)
2. δ(q0, b, Z) = (q1, bZ)
3. δ(q0, a, a) = (q1, aa)
4. δ(q1, a, b) = (q1, ab)
5. δ(q1, a, b) = (q1, ba)
6. δ(q1, b, b) = (q1, bb)

Pushing the symbols onto the stack

7. δ(q1, a, a) = (q2, ε)
8. δ(q1, b, b) = (q2, ε)
9. δ(q2, a, a) = (q2, ε)
10. δ(q2, b, b) = (q2, ε)
11. δ(q2, ε, Z) = (q2, ε)

Popping the symbols on reading the same kind of symbol

Simulation of abaaba

1. δ(q1, abaaba, Z) Apply rule 1
2. ⊢ δ(q1, baaba, aZ) Apply rule 5
3. ⊢ δ(q1, aaba, baZ) Apply rule 4
4. ⊢ δ(q1, aba, abaZ) Apply rule 7
5. ⊢ δ(q2, ba, baZ) Apply rule 8
6. ⊢ δ(q2, a, aZ) Apply rule 7

7. ⊢ δ(q2, ε, Z) Apply rule 11

8. ⊢ δ(q2, ε) Accept

PDA & Context Free Grammar:

If a grammar G is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar G. A parser can be built for the grammar G.

Also, if P is a pushdown automaton, an equivalent context-free grammar G can be constructed where

L(G) = L(P)

Algorithm to find PDA corresponding to a given CFG:

Step 1: Convert the given productions of CFG into GNF.

Step 2: The PDA will only have one state {q}.

Step 3: The initial symbol of CFG will be the initial symbol in the PDA.

Step 4: For non-terminal symbol, add the following rule:

1. δ(q, ε, A) = (q, α)

Where the production rule is A → α

Step 5: For each terminal symbols, add the following rule:

1. δ (q, a, a) = (q, ε) for every terminal symbol

Example 1:

Convert the following grammar to a PDA that accepts the same language.

1. S → 0S1 | A
2. A → 1A0 | S | ε

Solution:

The CFG can be first simplified by eliminating unit productions:

1. S → 0S1 | 1S0 | ε

Now we will convert this CFG to GNF:

1. S → 0SX | 1SY | ε
2. X → 1
3. Y → 0

The PDA can be:

R1: δ (q, ε, S) = {(q, 0SX) | (q, 1SY) | (q, ε)}

R2: δ (q, ε, X) = {(q, 1)}

R3: δ (q, ε, Y) = {(q, 0)}

R4: δ (q, 0, 0) = {(q, ε)}

R5: δ (q, 1, 1) = {(q, ε)}

Example 2:

Construct PDA for the given CFG, and test whether 0104 is acceptable by this PDA.

1. S → 0BB
2. B → 0S | 1S | 0

Solution:

The PDA can be given as:

1. A = {(q), (0, 1), (S, B, 0, 1), δ, q, S,?}

The production rule δ can be:

R1: δ (q, ε, S) = {(q, 0BB)}

R2: δ (q, ε, B) = {(q, 0S) | (q, 1S) | (q, 0)}

R3: δ (q, 0, 0) = {(q, ε)}

R4: δ (q, 1, 1) = {(q, ε)}

Testing 010^4 i.e. 010000 against PDA:

1. δ (q, 010000, S) ⊢ δ (q, 010000, 0BB)
2. ⊢ δ (q, 10000, BB) R1
3. ⊢ δ(q, 10000,1SB) R3
4. ⊢ δ(q, 0000, SB) R2
5. ⊢ δ(q, 0000, 0BBB) R1
6. ⊢ δ(q, 000, BBB) R3
7. ⊢ δ(q, 000, 0BB) R2
8. ⊢ δ(q, 00, BB) R3
9. ⊢ δ(q, 00, 0B) R2
10. ⊢ δ(q, 0, B) R3
11. ⊢ δ(q, 0, 0) R2
12. ⊢ δ(q, ε) R3
13. ACCEPT

Thus 010^4 is accepted by the PDA.

Example 3:

Draw a PDA for the CFG given below:

1. S → aSb
2. S → a | b | ε

Solution:The PDA can be given as:

P = {(q), (a, b), (S, a, b, z0), δ, q, z0, q}

The mapping function δ will be:

R1: δ(q, ε, S) = {(q, aSb)}

R2: δ(q, ε, S) = {(q, a) | (q, b) | (q, ε)}

R3: δ(q, a, a) = {(q, ε)}

R4: δ(q, b, b) = {(q, ε)}

R5: δ(q, ε, z0) = {(q, ε)}

Simulation: Consider the string aaabb

1. δ(q, εaaabb, S) ⊢ δ(q, aaabb, aSb) R3
2. ⊢ δ(q, εaabb, Sb) R1
3. ⊢ δ(q, aabb, aSbb) R3
4. ⊢ δ(q, εabb, Sbb) R2
5. ⊢ δ(q, abb, abb) R3
6. ⊢ δ(q, bb, bb) R4
7. ⊢ δ(q, b, b) R4
8. ⊢ δ(q, ε, z0) R5
9. ⊢ δ(q, ε)
10. ACCEPT

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q_0, I, F) such that the non- terminals of the grammar G will be {X_{wx} | w,x ∈ Q} and the start state will be $A_{q0,F}$.

Step 1 − For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m) contains (x, ε), add the production rule X_{wx} → a X_{yz}b in grammar G.

Step 2 − For every w, x, y, z ∈ Q, add the production rule X_{wx} → $X_{wy}X_{yx}$ in grammar G.

Step 3 − For w ∈ Q, add the production rule $X_{ww} \rightarrow \varepsilon$ in grammar G.

Example : **Obtain CFG for the given PDA below:**

$\delta(q_0,a,z_0) \rightarrow (q_1,az_0)$

$\delta(q_1,a,a) \rightarrow (q_1,aa)$

$\delta(q_1,b,a) \rightarrow (q_2,\lambda)$

$\delta(q_2,b,a) \rightarrow (q_2,\lambda)$

$\delta(q_2,\lambda,z_0) \rightarrow (q_f,\lambda)$

Sol. The productions of the grammar are as follows: -

$S \rightarrow [q_0, z_0, q]$

1) $[q_0, z_0, q] \rightarrow a\,[q_1, a, p]\,[p, z_0, q]$

2) $[q_1, a, q] \rightarrow a\,[q_1, a, p]\,[p, z_0, q]$

3) $[q_1, a, q] \rightarrow b$

4) $[q_2, a, q] \rightarrow b$

5) $[q_2, z_0, q] \rightarrow \lambda$

Where p & q are q_0 to q_f all combinations.

Petri nets Models

Petri nets are a basic model of parallel and distributed systems (named after Carl Adam Petri). The basic idea is to describe state changes in a system with transitions.

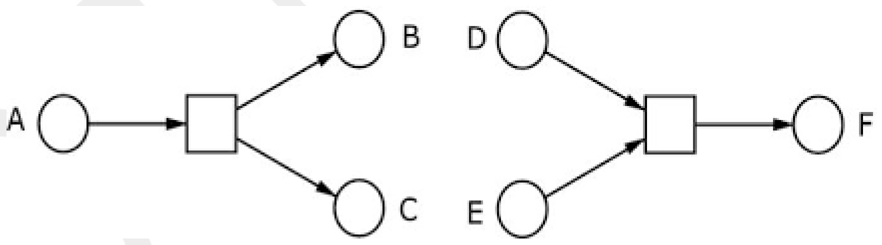


Figure 4.5: Perti-net

Petri nets contain places Circle and rectangle and transitions that may be connected by directed arcs. Places symbolize states, conditions, or resources that need to be met/be available before an action can be carried out. Transitions symbolize action.

Places may contain tokens that may move to other places by executing ("firing") actions. A token on a place means that the corresponding condition is fulfilled or that a resource is available:

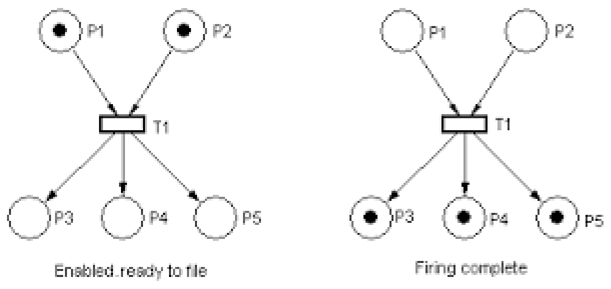


Figure 4.6: Petri net

In the example, transition t may “fire” if there are tokens on places p1 and p2. Firing t will remove those tokens and place new tokens on p3, p4 and p5

A Petri net is a tuple N = h P, T, F, W, m0i, where

• P is a finite set of places,

• T is a finite set of transitions,

• the places P and transitions T are disjoint (P ∩ T = ∅),

• F ⊆ (P × T) U (T × P) is the flow relation,

• W : ((P × T) U (T × P)) → IN is the arc weight mapping

(where W(f) = 0 for all f ∈/ F, and W(f) > 0 for all f ∈ F), and

• m0: P → IN is the initial marking representing the initial distribution of tokens.

Example: Dining philosophers

There are philosophers sitting around a round table. There are forks on the table, one between each pair of philosophers.

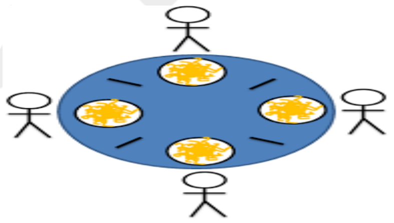


Figure 4.7: Dining philosophers

The philosophers want to eat spaghetti from a large bowl in the center of the table.

Dining philosophers: Petri net

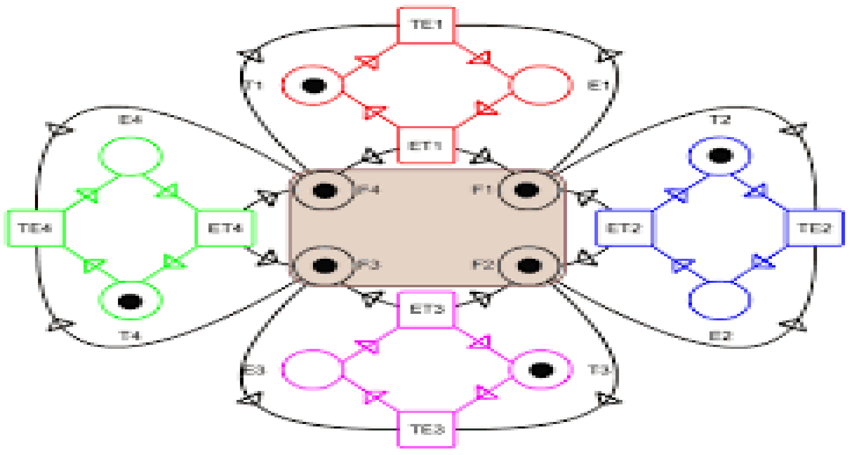


Figure 4.7: Dining philosophers with Petri-net

Subject Notes
CS-501- Theory of Computation
B. Tech V semester

Turing Machine: Techniques for construction. Universal Turing machine Multi-tape, multi-head and multidimensional Turing machine, N-P complete problems. Decidability and Recursively Enumerable Languages, decidability, decidable languages, undecidable languages, Halting problem of Turing machine & the post correspondence problem.

………

Objective: To develop an overview of how automata theory, languages and computation are applicable in engineering application.

………..

Unit-V: Turing Machine

Introduction:

Turing machine is considered as a simple model of a real computer. Turing machines can be used to accept all context-free languages, but also languages such as L = $\{a^m b^n c^{mn} : m \geq 0, n \geq 0\}$ which is not class of language comes under regular and context free. Every problem that can be solved on a real computer can also be solved by a Turing machine.

Description of Turing Machine:

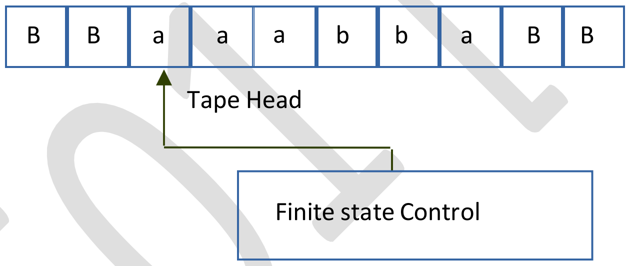


Figure 5.1: Turing Machine

- There are k tapes, for some fixed k ≥ 1 of infinite length. Each tape is divided into cells, each cell stores a symbol belonging to a finite set of tape symbols/ alphabets Γ. B iscalled blank symbol which also belongs to Γ.2. If a cell contains B, then this means that the cell is actually empty. (The given diagram is TM of single tape).
- Each tape has a tape head which can move along the tape, one cell per move. It can also read the cell it currently scans and replace the symbol in this cell by another tape symbol.
- There is a finite state control, which can be in any one of a finite number of states. The finite set of states from Q. The set Q contains three special states: a start (initial) state, an accept state, and a reject state.

Working of Turing Machine:

The Turing machine performs a sequence of computation steps. In one such step, it does the following:

- Immediately before the computation step, the Turing machine is in a state q of Q, and tape heads is on a certain cell.
- Depending on the current state q and the symbol that are read by the tape heads,

❖ The Turing machine switches to a state p of Q (which may be equal to p),

❖ Each tape head writes a symbol of Γ in the cell it is currently scanning (this symbol may be equal to the symbol currently stored in the cell), and

- Each tape head either moves one cell to the left, moves one cell to the right, or stays at the current cell.

Formal Definition of Turing Machine: A Deterministic Turing machine is a 7-tuple
M = (Q,Σ, Γ, δ, q0, B, F) where

- Q is a finite set of states,
- Σ is a finite set of input alphabet; the blank symbol B is not contained in Σ,
- Γ is a finite set of tape alphabet; this alphabet also contains the blank symbol B, and Σ ⊆ Γ,
- δ is called the transition function, which maps: Q × Γ into Q × Γ × D.
- q0 is start state, element of Q
- B is Blank sym δ bol element of Γ,
- F is Set of final states which is subset of Q.

Transitions occurs in Turing Machine:
Transition function of TM is denoted as δ(q, X) = (p, Y, D) which specified transition in TM is function of two components:

- Present state of TM, q
- Tape Symbol X, being scanned by TM.

In every transition

- TM enters into new state p or remain into same state p = q.
- A new symbol Y is written on the scanning cell in place of symbol X.
- If Y = X then there is no change in symbol of scamming cell.
- If D = L or ←, tape head moves one cell left to cell being scanned.
- If D = R or →, tape head moves one cell right to cell being scanned.

Computation by Turing Machine:
Consider TM, T = ({q_1 q_2 q_3 q_4 q_5}, {0,1}, {0,1, b}, δ, q_1, b, {q_5})
Transition Function δ is given by following transition table

Present State	**Tape Symbol**		
	B	**0**	**1**
→ q_1	(q_2 ,1, L)	(q_1 ,0, R)	-
q_2	($q_{3,\ b}$, R)	(q_2 ,0, L)	(q_2 ,1, L)
q_3	-	($q_{0,\ b}$, R)	($q_{5,\ b}$, R)
q_4	(q_5 ,0, R)	(q_4 ,0, R)	(q_4 ,1, R)
$\bigcirc q_5$	(q_2 ,0, L)	-	-

Table 5.1: Computation by Turing Machine

Computation sequence for input string w = 00

Initial ID: $q_1 00$

$q_1 00$ |- $0q_1 0$ |- $00q_1$ |- $0q_2 01$ |- $q_2 001$ |- $q_2 b001$ |- $q_3 001$ |- $q_4 01$ |- $0q_4 1$ |- $01q_4$ |- $010q_5$ |- $01q_2 00$ |-*- $q_5 000$

Transition Diagram of Turing Machine:

For transition function δ (q, β) = (p, Y, D) The transition diagram will have

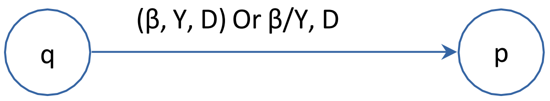


Figure 5.2: Transition Diagram

Acceptance of Language by Turing Machine:

A TM accepts a language if it enters into a final state for any input string w. A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing machine.

A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine. There may be some cases where a TM does not stop. Such TM accepts the language, but it does not decide it.

Example 1:

Design a TM to recognize all strings consisting of an odd number of α's.

Solution:

- The Turing machine M can be constructed by the following moves:
- Let q1 be the initial state.
- If M is in q1; on scanning α, it enters the state q2 and writes B (blank).
- If M is in q2; on scanning α, it enters the state q1 and writes B (blank).

From the above moves, we can see that M enters the state q1 if it scans an even number of α's, and it enters the state q2 if it scans an odd number of α's. Hence q2 is the only accepting state.

Hence,

M = {{q1, q2}, {1}, {1, B}, δ, q1, B, {q2}} where δ is given by:

Tape Alphabet Symbol	Present State "q1"	Present State "q2"
a	BRq1	BRq2

Table 5.2: Example of Turing Machine

Example 2:

Design a Turing Machine that reads a string representing a binary number and erases all leading 0's in the string. However, if the string comprises of only 0's, it keeps one 0.

Solution:

Let us assume that the input string is terminated by a blank symbol, B, at each end of the string.

The Turing Machine, M, can be constructed by the following moves:

- Let q0 be the initial state.
- If M is in q0, on reading 0, it moves right, enters the state q1 and erases 0. On reading 1, it enters the state q2 and moves right.
- If M is in q1, on reading 0, it moves right and erases 0, i.e., it replaces 0's by B's. On reaching the leftmost 1, it enters q2 and moves right. If it reaches B, i.e., the string comprises of only 0's, it moves left and enters the state q3.

- If M is in q2, on reading either 0 or 1, it moves right. On reaching B, it moves left and enters the state q4. This validates that the string comprises only of 0's and 1's.
- If M is in q3, it replaces B by 0, moves left and reaches the final state qi.
- If M is in q4, on reading either 0 or 1, it moves left. On reaching the beginning of the string, i.e., when it reads B, it reaches the final state qf.

Hence, M = {{q0, q1, q2, q3, q4, qf}, {0,1, B}, {1, B}, δ, q0, B, {qf}} where δ is given by:

Tape Alphabet Symbol	Present State "q0"	Present State "q1"	Present State "q2"	Present State "q3"	Present State "q4"
0	BRq1	BRq1	0Rq2	-	0Lq4
1	1Rq2	1Rq2	1Rq2	-	1Lq4
B	BRq1	BLq3	BLq4	0Lqf	BRqf

Table 5.3: Example of Turing Machine

Techniques of Construction:

Multi-tape Turing Machine:

Multi-tape Turing Machines have multiple tapes where each tape is accessed with a separate head. Each head can move independently of the other heads. Initially the input is on tape 1 and others are blank. At first, the first tape is occupied by the input and the other tapes are kept blank. Next, the machine reads consecutive symbols under its heads and the TM prints a symbol on each tape and moves its heads.

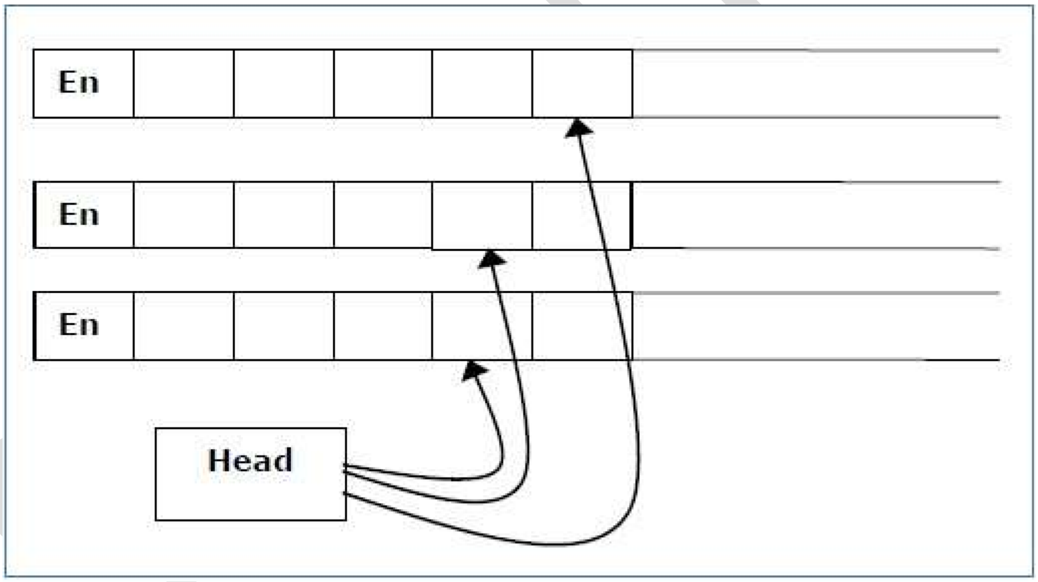


Figure 5.3: Multi-tape Turing Machine

A Multi-tape Turing machine can be formally described as a 7-tuple M = (Q,Σ, Γ, δ, q0, B, F) where −

- Q is a finite set of states
- Σ is set of input alphabet
- Γ is the tape alphabet
- B is the blank symbol
- δ is a relation on states and symbols where δ: Q × X^k → Q × (X × {Left shift, Right shift, OnShift}) k where there is **k** number of tapes
- q0 is the initial state
- F is the set of final states

Note: Every Multi-tape Turing machine has an equivalent single-tape Turing machine.

Non-Deterministic Turing Machine:

In a NDTM, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a **Decider** and if for some input, all branches are rejected, the input is also rejected.

A non-deterministic Turing machine can be formally defined as a 7-tuple (Q, ∑, Γ, δ, q0, B, F) where −

- Q is a finite set of states
- Γ is the tape alphabet
- ∑ is the input alphabet
- δ is a transition function; δ: Q × X → P (Q × X × {Left shift, Right shift}).
- q0 is the initial state
- B is the blank symbol
- F is the set of final states

Universal Turing Machine:

A universal Turing machine (UTM) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input thereof from its own tape.

Linear Bounded Automata:

A linear bounded automaton is a multi-track non-deterministic Turing machine with a tape of some bounded finite length.

Length = function (Length of the initial input string, constant c)

Here,

Memory information ≤ c × Input information

The computation is restricted to the constant bounded area. The input alphabet contains two special symbols which serve as left end markers and right end markers which mean the transitions neither move to the left of the left end marker nor to the right of the right end marker of the tape.

A linear bounded automaton can be defined as an 8-tuple (Q, X, Σ, q0, ML, MR, δ, F) where:

- Q is a finite set of states
- X is the tape alphabet
- Σ is the input alphabet
- q0 is the initial state
- ML is the left end marker

- MR is the right end marker where MR≠ ML
- δ is a transition function which maps each pair (state, tape symbol) to (state, tape symbol, Constant 'c') where c can be 0 or +1 or -1
- F is the set of final states

A deterministic linear bounded automaton is always context-sensitive and the linear bounded automaton with empty language is undecidable.

Offline Turing Machine:

An Offline Turing Machine has two types:

- One tape is read only and contains the input.
- The other is read-write and is initially blank.

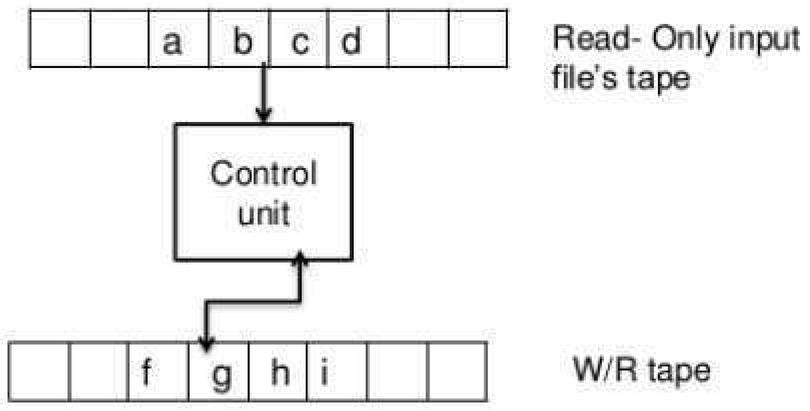


Figure 5.4: Offline Turing Machine

A standard Turing Machine is simulated by Offline Turing Machine and an Offline Turing Machine simulated by standard Turing Machine.

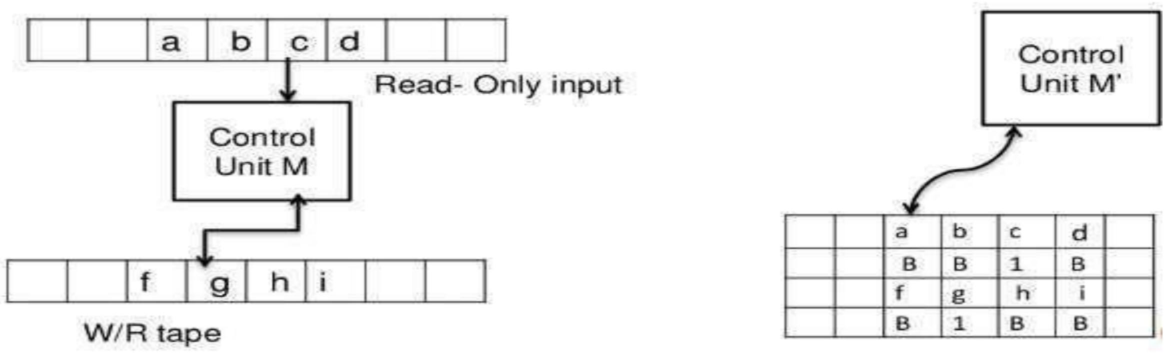


Figure 5.5: Offline Turing Machine

Equivalence of Single Tape & Multi Tape Turing Machine

In the classical framework k-tape Turing machines have the same computational power of Single-tape Turing machines and given a Multi-tape Turing Machine M_K it is always possible to define a Single-tape Turing Machine which is able to fully simulate its behavior and therefore to completely execute its computations. The Gross-one methodology allows us to give a more accurate definition of the equivalence among different machines as it provides the possibility not only to separate different classes of infinite sets with respect to their cardinalities but also to measure the number of elements of some of them. With reference to Multi-tape Turing machines, the Single-tape Turing Machines adopted for their simulation use a particular kind of tape which is divided into tracks (multitrack tape). In this way, if the tape has m tracks, the head is able to access (for reading and/or writing) all the m characters on the tracks during a single operation. This tape organization leads to a straightforward definition of the behavior of a Single-tape Turing machine able to completely execute the computations of a given Multi-tape Turing machine

Recursive & Recursively Enumerable Language:

A Turing Machine may

- Halt and accept the input
- Halt and reject the input, or
- Never halt/loop

Recursive Enumerable or Type-0 Language: RE languages or type-0 languages are generated by type-0 grammars. A RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. RE languages are also called as Turing recognizable languages.

Recursive Language: A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language. e.g.; L= $\{a^nb^nc^n|n>=1\}$ is recursive because we can construct a Turing machine which will move to final state if the string is of the form $a^nb^nc^n$ else move to non-final state. So, the TM will always halt in this case. REC languages are also called as Turing decidable languages.

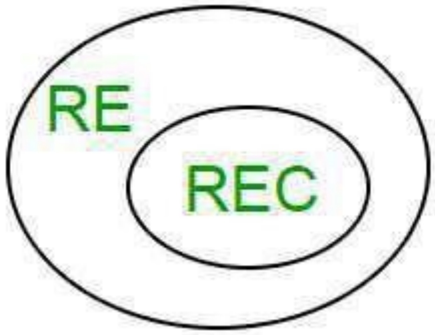


Figure 5.6: Relationship between RE & REC

Language Decidability:

A language is called Decidable or Recursive if there is a Turing machine which accepts and halts on every input string w. Every decidable language is Turing-Acceptable.

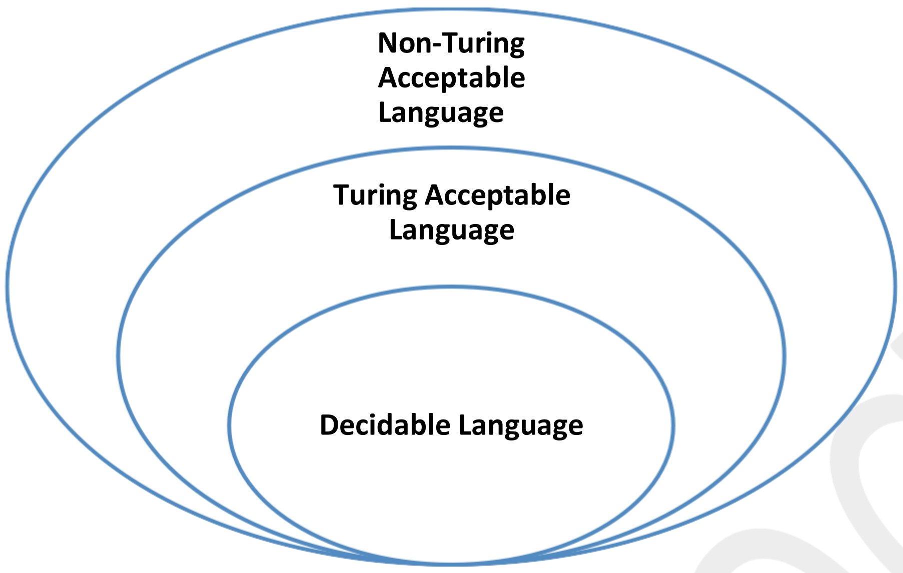


Figure 5.7: Language Decidability

A decision problem P is decidable if the language L of all yes instances to P is decidable.

For a decidable language, for each input string, the TM halts either at the accept or the reject state as depicted in the following diagram:

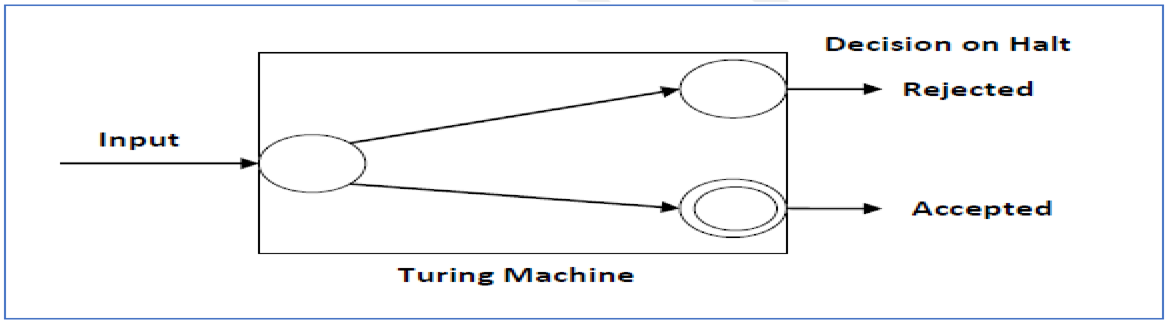


Figure 5.8: Decidable Language

Example 1: Find out whether the following problem is decidable or not: Is a number 'm' prime?

Solution:

Prime numbers = {2, 3, 5, 7, 11, 13, ... }

Divide the number 'm' by all the numbers between '2' and '√m' starting from '2'. If any of these numbers produce a remainder zero, then it goes to the "Rejected state", otherwise it goes to the "Accepted state". So, here the answer could be made by 'Yes' or 'No'.

Hence, it is a decidable problem.

Example 2: Given a regular language L and string w, how can we check if w∈ L?

Solution:

Take the DFA that accepts L and check if w is accepted

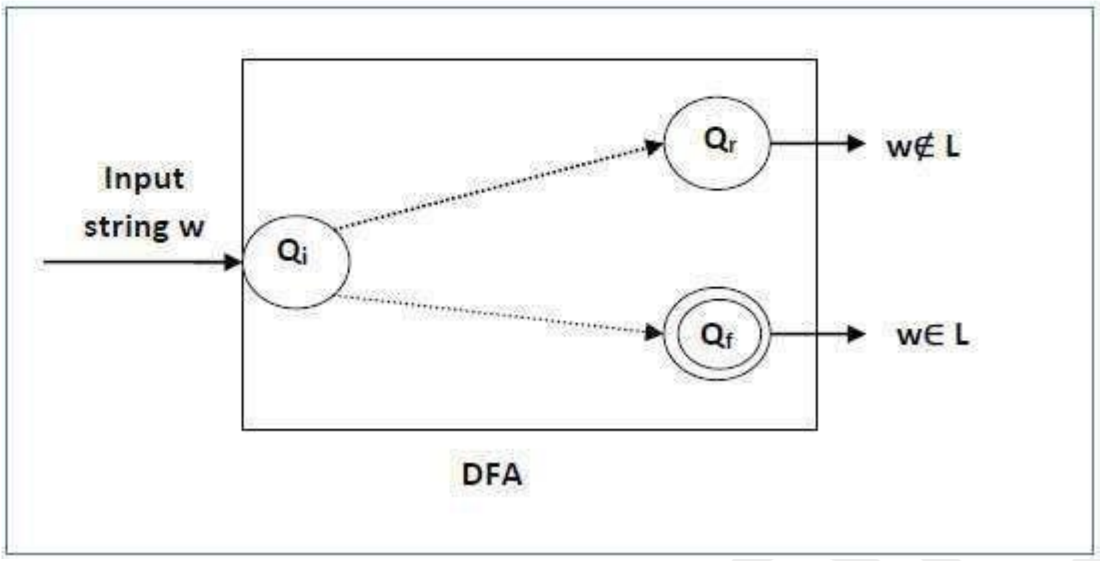


Figure 5.9: Example of Language Decidable

Note:

1. If a language L is decidable, then its complement L' is also decidable.

2. If a language is decidable, then there is an enumerator for it.

Undecidable Language:

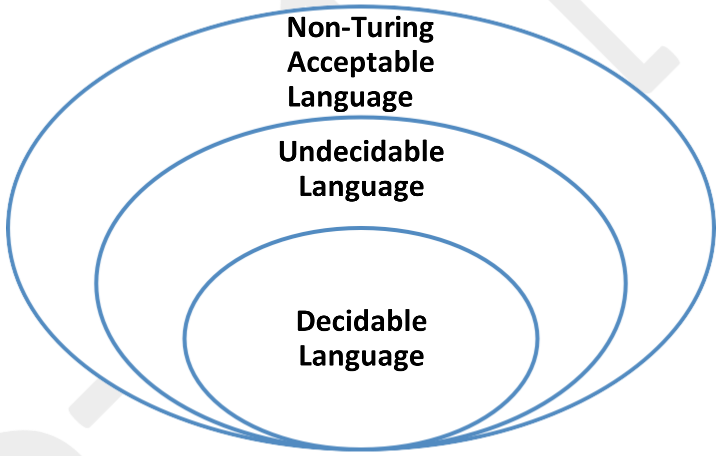


Figure 5.10: Undecidable Language

For an undecidable language, there is no Turing Machine which accepts the language and makes a decision for every input string w (TM can make decision for some input string though). A decision problem P is called "undecidable" if the language L of all yes instances to P is not decidable. Undecidable languages are not recursive languages, but sometimes, they may be recursively enumerable languages.

Example:

- The halting problem of Turing machine
- The mortality problem
- The mortal matrix problem
- The Post correspondence problem, etc.

Turing Machine Halting Problem:
Input: A Turing machine and an input string w.

Problem: Does the Turing machine finish computing of the string w in a finite number of steps? The answer must be either yes or no.

Proof: At first, we will assume that such a Turing machine exists to solve this problem and then we will show it is contradicting itself. We will call this Turing machine as a Halting machine that produces a 'yes' or 'no' in a finite amount of time. If the halting machine finishes in a finite amount of time, the output comes as 'yes', otherwise as 'no'. The following is the block diagram of a Halting machine:

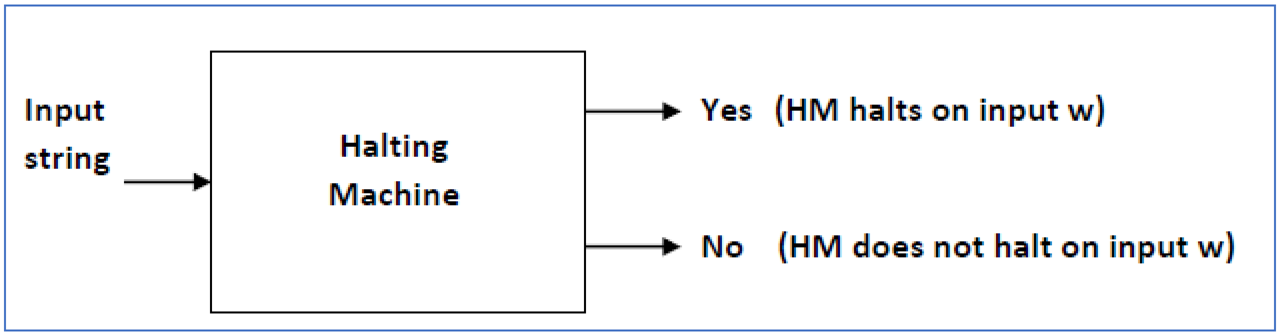


Figure 5.11: Turing Machine Halting Problem

Now we will design an inverted halting machine (HM)' as:

- If H returns YES, then loop forever.
- If H returns NO, then halt.

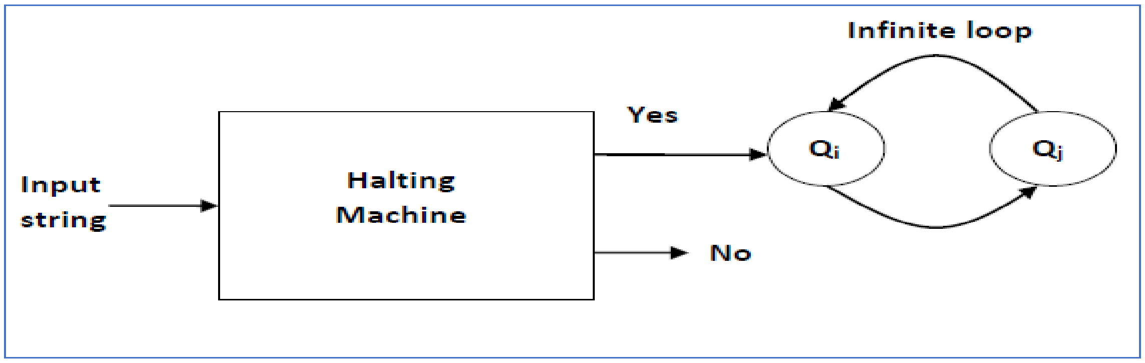


Figure 5.12: Inverted Halting Machine

Further, a machine (HM)2 which input itself is constructed as follows:

- If (HM)2 halts on input, loop forever.
- Else, halt.

Here, we have got a contradiction. Hence, the halting problem is undecidable

Introduction of P, NP, NP Complete & NP Hard Problem:

P & NP Problems:
If a problem can be solved in polynomial time, it is said to belong to the **P** class of problems. P-type problems

are tractable.
For some intractable problems, you can verify that the solution is correct using a P-type algorithm. For example, you can verify that a given solution to the TSP visits every city. These problems are referred to as Non-deterministic Polynomial problems or NP-type problems. The challenge for programmers is to find a P-type solution to NP-type problems.

P Problems:
As the name says these problems can be solved in polynomial time, i.e.; O(n), O(n^2) or O(nk) where k is a constant.

NP Problems:
Some people think NP as Non-Polynomial. But actually, it is Non-deterministic Polynomial time. i.e.; "yes" instances of these problems can be solved in polynomial time by a non-deterministic Turing machine and hence can take up to exponential time (some problems can be solved in sub-exponential but super polynomial time) by a deterministic Turing machine. In other words, these problems can be verified (if a solution is given, say if it is correct or wrong) in polynomial time. Examples include all P problems. One example of a problem not in P but in NP is Integer Factorization.

NP Complete Problems (NPC):
Over the years many problems in NP have been proved to be in P (like Primarily Testing). Still, there are many problems in NP not proved to be in P. i.e.; the question still remains whether P=NP (i.e.; whether all NP problems are actually P problems).
NP Complete Problems helps in solving the above question. They are a subset of NP problems with the property that all other NP problems can be reduced to any of them in polynomial time. So, they are the hardest problems in NP, in terms of running time. If it can be showed that any NPC Problem is in P, then all problems in NP will be in P (because of NPC definition), and hence P=NP=NPC.
All NPC problems are in NP (again, due to NPC definition). Examples of NPC problems

NP Hard Problems (NPH):
These problems need not have any bound on their running time. If any NPC Problem is polynomial time reducible to a problem X, that problem X belongs to NP Hard class. Hence, all NP Complete problems are also NPH. In other words, if a NPH problem is non-deterministic polynomial time solvable, it is a NPC problem. Example of a NP problem that is not NPC is Halting Problem.

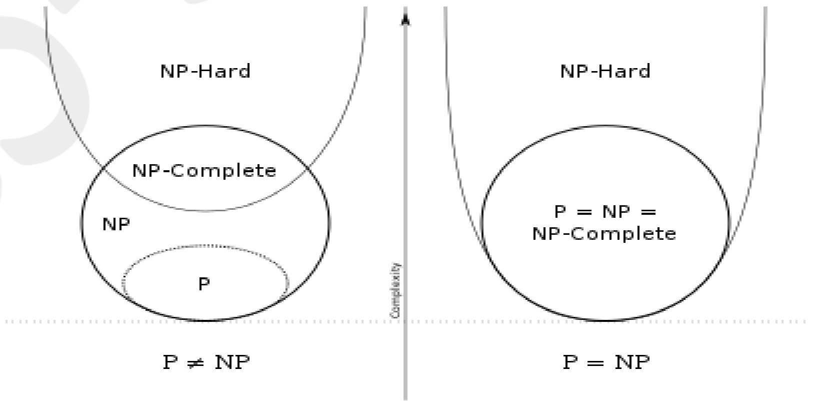


Figure 5.13: NP complete problem

From the figure 5.10, it's clear that NPC problems are the hardest problems in NP while being the simplest ones in NPH. i.e.; NP∩NPH=NPC

Given a general problem, we can say it's in NPC, if and only if we can reduce it to some NP problem (which shows it is in NP) and also some NPC problem can be reduced to it (which shows all NP problems can be reduced to this problem).
Also, if a NPH problem is in NP, then it is NPC

Examples of NP Problems:
Boolean Satisfiability Problem:
Boolean Satisfiability or simply SAT is the problem of determining if a Boolean formula is satisfiable or unsatisfiable.
Satisfiable: If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the formula is satisfiable.
Unsatisfiable: If it is not possible to assign such values, then we say that the formula is unsatisfiable.

Examples (as shown in table 5.4):
F= A^B', is satisfiable, because A = TRUE and B = FALSE makes F = TRUE.
G=A^A', is unsatisfiable, because:

A	A'	G
True	false	False
False	True	False

Table 5.4: Example of SAT

Boolean satisfiability problem is NP-complete (This was proved by Cook's Theorem).

2-SAT Problem:
- 2-SAT is a special case of Boolean Satisfiability Problem and can be solved in polynomial time.
- To understand this better, first let us see what is Conjunctive Normal Form (CNF) or also known as Product of Sums (POS).

CNF: CNF is a conjunction (AND) of clauses, where every clause is a disjunction (OR).
F = $(A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge (A_3 \vee B_3)$ .... $(A_m \vee B_m)$
- Now, 2-SAT limits the problem of SAT to only those Boolean formula which are expressed as a CNF with every clause having only **2 terms** (also called **2-CNF**)
- For 2-SAT problem the CNF value is TRUE, if value of every clause is TRUE. Let one of the clauses be (A V B) so we can say (A V B) = TRUE in following two conditions
  - If A = 0, B must be 1 i.e. $(A' \Rightarrow B)$
  - If B = 0, A must be 1 i.e. $(B' \Rightarrow A)$

  Thus (A V B) is true equivalent to $(A' \Rightarrow B) \wedge (B' \Rightarrow A)$

Vertex Cover Problem:
A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover.
Following are some examples (as shown in fig 5.11): -

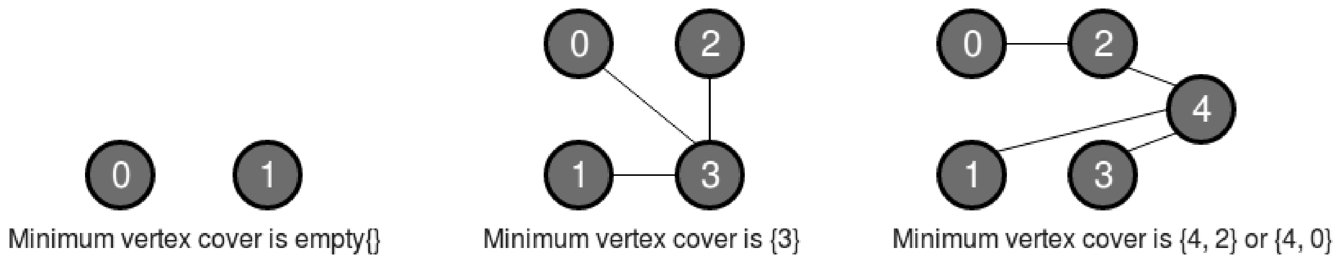

0
1
Minimum vertex cover is empty{}
0
2
1
3
Minimum vertex cover is {3}
0
2
4
1
3
Minimum vertex cover is {4, 2} or {4, 0}

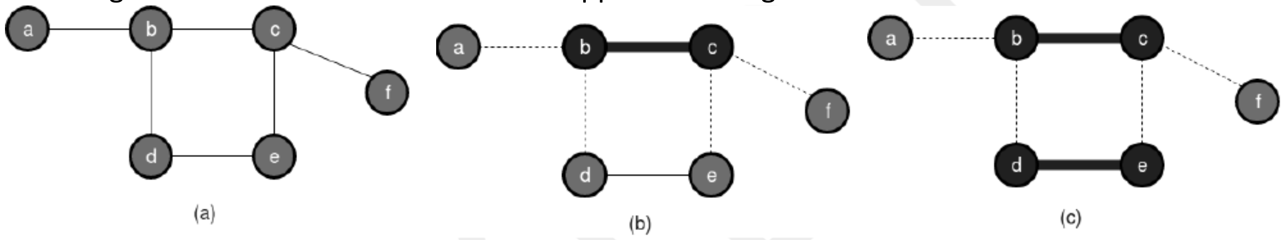
Figure 5.14: Examples for vertex cover problem

Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for this unless P = NP. Although There can be an approximate polynomial time algorithm to solve the problem. Following is a simple approximate algorithm.

Approximate Algorithm for Vertex Cover:

1) Initialize the result as {}
2) consider a set of all edges in given graph. Let the set be E.
3) Do following while E is not empty
   a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
   b) Remove all edges from E which are either incident on u or v.
4) Return result

Below figure 5.12 show execution of above approximate algorithm:



Minimum vertex cover is {b,c,d} or {b,c,e}

Figure 5.15: Execution steps of vertex cover problem

Hamiltonian Cycle Problem:

A Hamiltonian cycle is a cycle in a graph that visits each vertex exactly once. To show Hamiltonian Cycle Problem is NP-complete, we first need to show that it actually belongs to the class NP, and then use a known NP-complete problem to Hamiltonian Cycle.
So does Hamiltonian Cycle Problem ∈ NP?

Given: $Graph\ G = (V, E)$
Certificate: List of vertices on Hamiltonian Cycle
To check if this list is actually a solution to the Hamiltonian cycle problem, one counts the vertices to make sure they are all there, and then checks that each is connected to the next by an edge, and that the last is connected to the first. It takes time proportional to n, because there are n vertices to count and n edges to check. n is a polynomial, so the check runs in polynomial time.
Therefore, Hamiltonian Cycle ∈ NP.
Prove Hamiltonian Cycle Problem ∈ NP-Complete
Reduction: Vertex Cover to Hamiltonian Cycle
Definition: Vertex cover is set of vertices that touch all edges in the graph.
Given a $graph\ G$ and integer k, construct a $graph\ G'$ such that G has a vertex cover of size k if G' has a $Hamiltonian\ cycle$.

Idea: To construct widget for each edge in the graph.
$i.e\ \forall\ uv$ in the $Graph\ G$, create a widget shown below: -

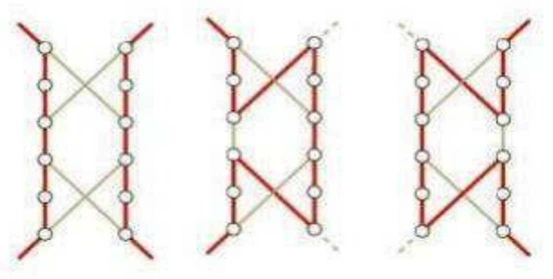


Figure 5.16: An Example for Hamiltonian Cycle Graph

As shown in figure 5.13, there are three ways to traverse a widget; 1. Enter from u, go somewhere else in the graph, and then come back through the other side i.e. v 2. Enter and Exit through u 3. Enter and Exit through v Construct G' for G (Vertex cover) of size k = 2 with the construction, any graph with a vertex cover, can be used to make a graph with a Hamiltonian Cycle graph. Since creating such a graph can be done under polynomial time, simply replace edges with widgets and make proper connections, we have a reduction from Vertex Cover to Hamiltonian Cycle. This means that finding whether a graph has a Hamiltonian Cycle or not is NP Hard. As we have seen earlier it's also in NP, therefore, Hamiltonian Cycle is an NP Complete Problem.

Traveling Salesman Problem:

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

The traveling salesman problem can be described as follows:

TSP = {(G, f, t): G = (V, E) a complete graph that contains a traveling salesman tour with cost that does not exceed t.}

f is a function V×V → Z, t ∈ Z,

Example: Consider the following set of cities as shown in figure 5.14:

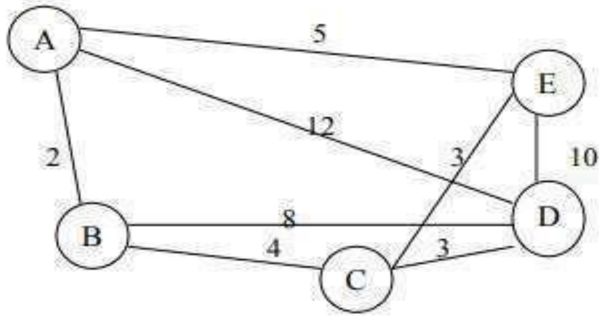


Figure 5.17: An example of TSP

The problem lies in finding a minimal path passing from all vertices once. For example, the path Path1 {A, B, C, D, E, A} and the path Path2 {A, B, C, E, D, A} pass all the vertices but Path1 has a total length of 24 and Path2 has a total length of 31.

Theorem: The traveling salesman problem is NP-complete.

Proof:

First, we have to prove that TSP belongs to NP. If we want to check a tour for credibility, we check that the tour contains each vertex once. Then we sum the total cost of the edges and finally we check if the cost is minimum. This can be completed in polynomial time thus TSP belongs to NP.

Secondly, we prove that TSP is NP-hard. One way to prove this is to show that Hamiltonian cycle <= TSP (given that the Hamiltonian cycle problem is NP-complete). Assume G = (V, E) to be an instance of Hamiltonian cycle. An instance of TSP is then constructed. We create the complete graph = (V, ≤ P G′ E′) where E′ = {(I, j): i, j ∈ V and i ≠ j}. Thus, the cost function is defined as:

$$t(i,j)=\begin{cases}0 \text{ if } (i, j)\in E,\\ 1 \text{ if } (i, j)\notin E.\end{cases}$$

Now suppose that a Hamiltonian cycle h exists in G. It is clear that the cost of each edge in h is 0 in G′ as each edge belongs to E. Therefore, h has a cost of 0 in G′. Thus, if graph G has a Hamiltonian cycle then graph G′ has a tour of 0 costs.
Conversely, we assume that G′ has a tour h′ of cost at most 0. The cost of edges in E′ are 0 and 1 by definition. So, each edge must have a cost of 0 as the cost of h′ is 0. We conclude that h′ contains only edges in E.
So, we have proven that G has a Hamiltonian cycle if and only if G′ has a tour of cost at most 0. Thus, TSP is NP-complete.