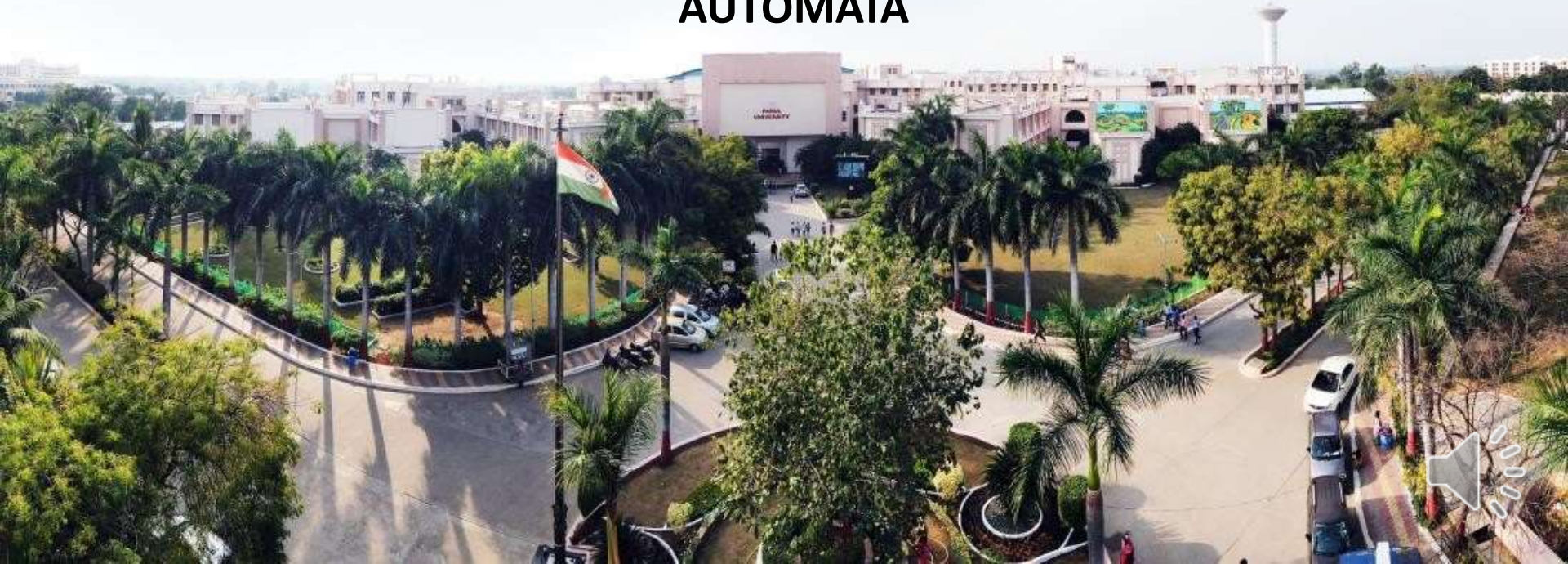
Parul® University

THEORY OF COMPUTATION
CODE:303105306

UNIT 2 : REGULAR LANGUAGE AND FINITE AUTOMATA



Parul®
University

DIGITAL LEARNING CONTENT

CHAPTER-2

Regular Language and Finite Automata

Types of Regular Expression

- **The Regular expression is classified in to three types**

1. Restricted Regular Expression(* , . , +) or standard R.E.
2. Semi Restricted Regular Expression (* , . , + , ∩)
3. Unrestricted (* , . , + , ∩ , ~)

Some identities of Regular Expression

- **If r is any regular expression then following identities follows**
  1. r^* is also an regular expression
  2. r^+ is also an regular expression
  3. If r = ϕ then r^* = € and r^+ = ϕ
  4. If r = € then r^* = € and r^+ = €
  5. $r^* + r^+ = r^*$
  6. $r^* \cdot r = r^+$
  7. $(r^*)^* = r^*$
  8. **Let r1 and r2 are two R.E. then $(r1+r2)^*$ is also a R.E.**

     1. $(r1+r2)^* = (r1^* + r2^*)^*$
     2. $(r1+r2)^* = (r1 + r2^*)^*$
     3. $(r1+r2)^* = (r1^* + r2)^* = (r1^* \cdot r2^*)^* = (r2^* \cdot r1^*)^*$

Regular Expression for some regular languages

- **If r is any regular expression then L(r) is regular language generated byregular expression r.**

Regular Expression	Regular Languages
r = ϕ	L = {}
r = €	L = {€}
r = a	L = {a}
r = ab	L = {ab}
r = (a+b)	L = {a,b}
r = (a+b+c)	L = {a , b , c}
r = w1+w2+w3…+wn	L = {w1,w2,w3,…wn}

Simplification of Regular Expression

- **Simplification of Regular expression means simplify the given regular expression using identities if possible.**
- **Example**
  1. $(0+€)^* = (0^* + €)^*$
     $= (0^*)^*$
     $= 0^*$
  2. $(0^* + 0^+)^* = (0^*)^*$
     $= 0^*$
  3. $(1^* 0)^* + 0^*1 + 1^*$(No simplification required)

PU

Regular Expression for some regular languages

1. Construct the regular expression that generates all the string a's and b's

I.	Including €	$r=(a+b)^*$
II.	Excluding €	$r=(a+b)^+$
III.	Where each string start with ab	$r=ab(a+b)^*$
IV.	Where each string end with ab	$r=(a+b)^*$ ab
V.	Contain substring aba	$r=(a+b)^*$ aba $(a+b)^*$
VI.	Start and end with a	r=a $(a+b)^*$ a +a
VII.	Third symbol from LHS is a	r= (a+b) (a+b)a$(a+b)^*$
VIII.	fourth symbol from RHS is b	$r=(a+b)^*b(a+b)^3$

Regular Expression for some regular languages

2. Construct the regular expression that generates all the string a's and b's where the length of the string is

I.	Exactly 3	$r=(a+b)^3$
II.	At least 3	$r=(a+b)^3\ (a+b)^*$
III.	At most 3	$r=(a+b+ €\ )^*$
IV.	Even	$r=[(a+b)^2]^*$
V.	Odd	$r=(a+b)[(a+b)^2]^*$
VI.	$\|w\|\equiv 2(\text{mod } 3)$	$r =(a+b)^2\ [(a+b)^3]^*$

Regular Expression for some regular languages

3. Construct the regular expression that generates all the string a's and b's where the number of a's in the string is

I.	Exactly 3	r=b*ab*ab*ab*
II.	At most 3	r=b*(a+€) b*(a+€) b*(a+€)b*
III.	At least 3	r=(a+b)*a(a+b)* a(a+b)* a(a+b)*
IV.	Even	r=(b*ab*ab*)*b*
V.	Odd	r=(b*ab*ab*)*ab*
VI.	|w|≡1(mod 3)	r =b*ab*(b*a b*a b*ab*)*

Parul®
University

DIGITAL LEARNING CONTENT

Finite Automata

Definition (DFA)

- **It is finite state automata which is used to accept or reject a sequence of string**
- In DFA, for each input symbol, there must be only one state. Hence, it is called **Deterministic Automaton**. As it has a finite number of states, the machine is called **Deterministic Finite Machine** or **Deterministic Finite Automaton.**

Image source : Google

Formal definition of DFA

- **A DFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where –**

  Q is a finite set of states.

  ∑ is a finite set of symbols called the alphabet.

  δ is the transition function where δ: Q × ∑ → Q {only one state}

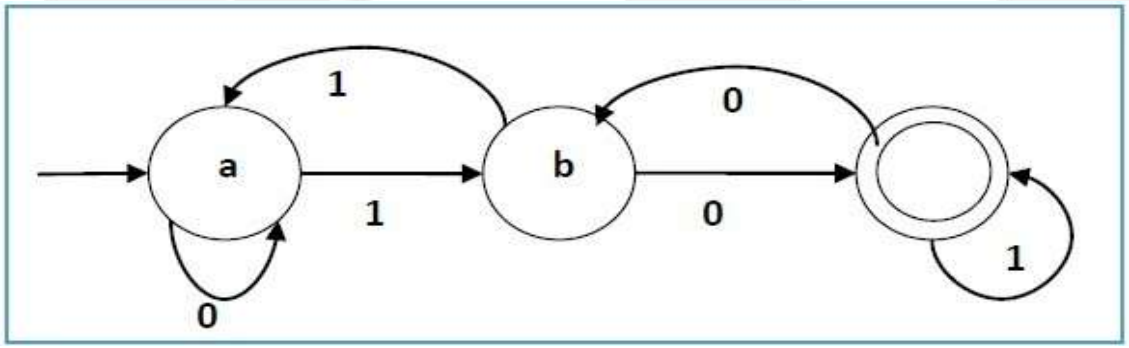
  q_0 is the initial state from where any input is processed ($q_0 \in Q$).

  F is a set of final state/states of Q ($F \subseteq Q$).

Image source : Google

Graphical Representation of a DFA

- A DFA is represented by digraphs called **state diagram**.
- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.



Graphical Representation of DFA

Equivalence of DFA and Regular Expression

- **If a language is accepted by DFA then L is denoted by a Regular expression.**
- Regular expression and Finite Automata , both have the same computational power.
- For each and every regular expression we can design DFA such that *L(R)=L(F).*

L(R)= Language accepted by regular expression

L(F)= Language accepted by finite automata

Acceptability by DFA and NDFA

- A string is accepted by a DFA/NDFA iff the DFA/NDFA starting at the initial state ends in an accepting state (any of the final states) after reading the string wholly.
- A string S is accepted by a DFA/NDFA $(Q, \sum, \delta, q_0, F)$, iff

  $$\delta^*(q_0, S) \in F$$

- The language **L** accepted by DFA/NDFA is

  $$\{S \mid S \in \sum^* \text{ and } \delta^*(q_0, S) \in F\}$$

- A string S' is not accepted by a DFA/NDFA $(Q, \sum, \delta, q_0, F)$, iff

  $$\delta^*(q_0, S') \notin F$$

- The language L' not accepted by DFA/NDFA (Complement of accepted language L) is

  $$\{S \mid S \in \sum^* \text{ and } \delta^*(q_0, S) \notin F\}$$

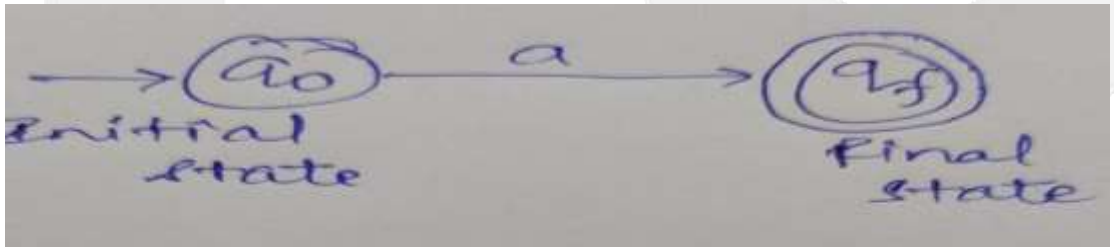
Construction of DFA

- **Rules for construction of DFA**

- Step 1:- Recognize language.
- Step 2:- Find Minimum string using language.
- Step 3:- Create a DFA which accepts or satisfy minimum string.
- Step 4:- Satisfy each input symbol at every state.
- ***NOTE:- Designed DFA will be minimal DFA***

Construction of DFA (Example)

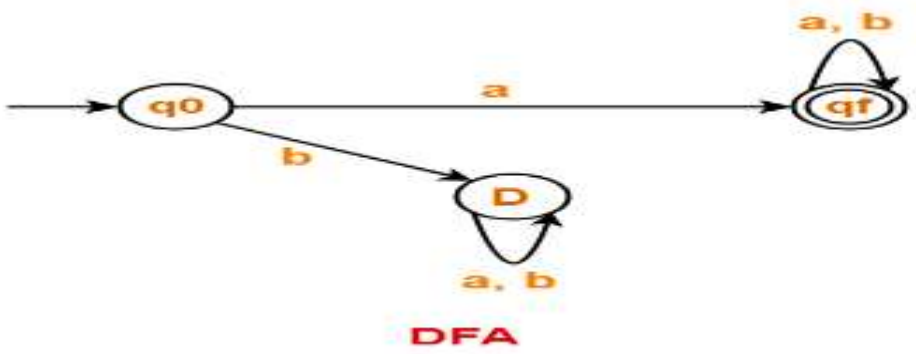
- **Construct DFA which starts with “a”, given input symbol {a , b}**
- DFA which must Start with “a” but necessarily not with “b” i.e. If string will start from “b” then input symbol “b” will move to dead state or trap state.
- So, L={a,ab,abb,aab….}
- Minimal string = a



Construction of DFA (Example)

- **Construct DFA which starts with “a”, given input symbol {a , b} (Continued)**

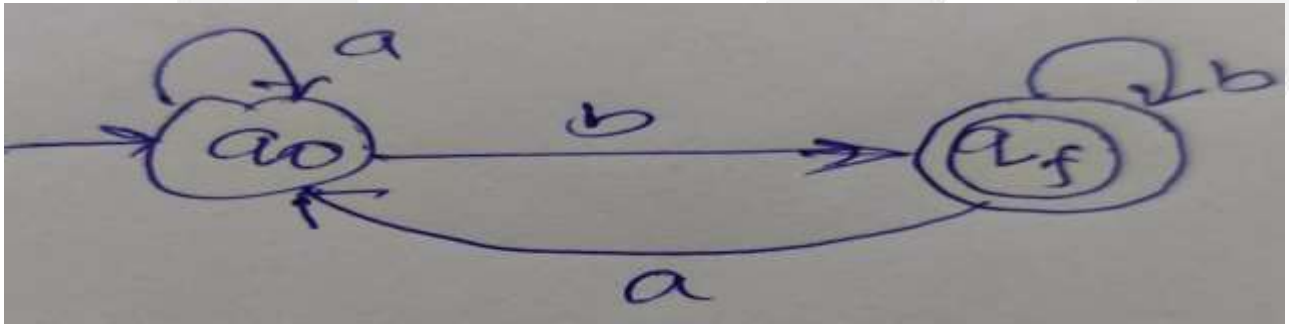
- Now we will satisfy all state with every input and then we will get



DFA

Construction of DFA (Example)

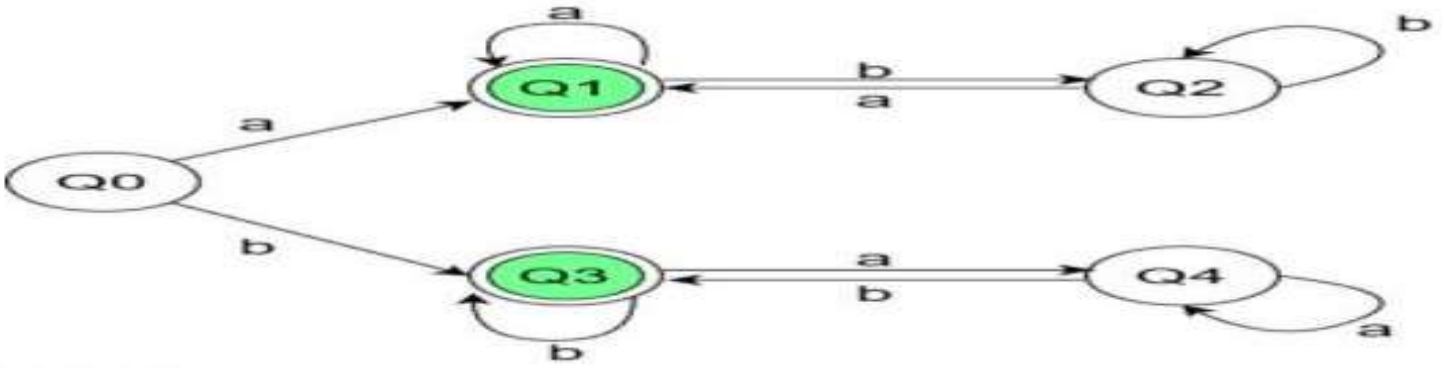
- **Construct DFA which ends with “b”, given input symbol {a , b}**
- DFA will start with “a” and “b” both but necessarily end with “b” only.
- L={b,ab,bb,aab,abab….}
- Minimal string = b
- After following above all rules we will get below designed DFA



Construction of DFA (Example)

- **Construct DFA which starts and ends with same character, given input symbol {a , b}**

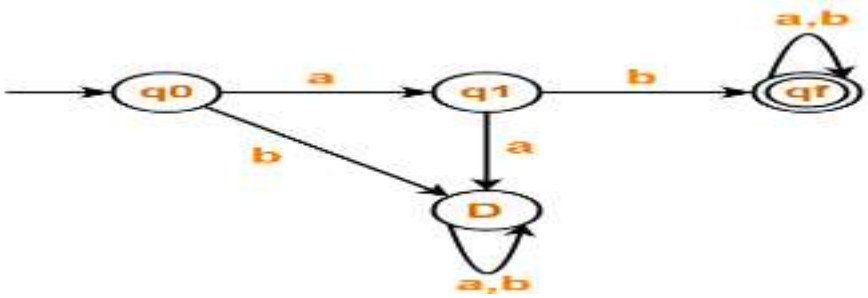
- L={a,b,aa,bb,aba,bab,abba,baab…..}
- Minimal string = {a,b}
- After following above all rules we will get below designed DFA



Construction of DFA (Example)

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

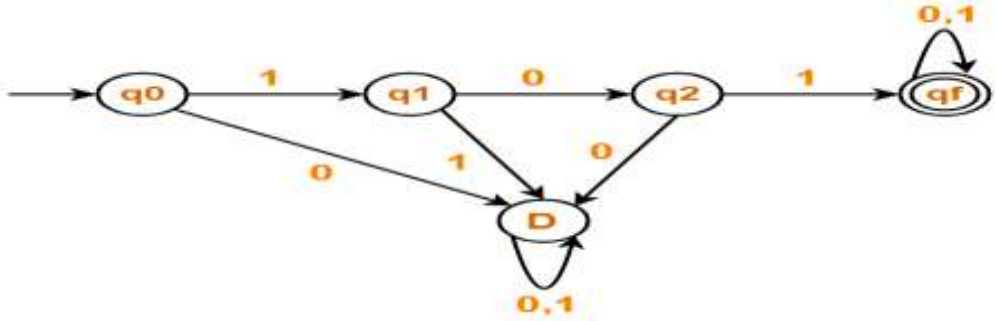
- L={**ab**, **ab**aab, **ab**abab, **ab**bbbbb…..}
- Minimal string = {ab}
- After following above all rules we will get below designed DFA



Construction of DFA (Example)

- **Draw a DFA for the language accepting strings starting with '101' over input alphabets ∑ = {0, 1}**

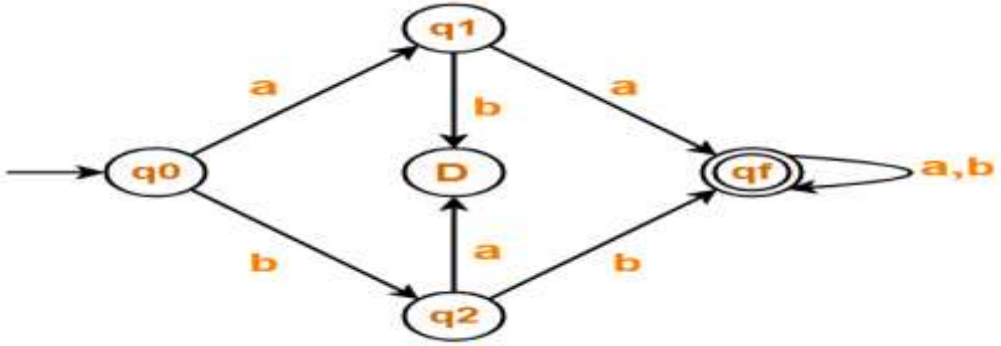
- L={101,1010101,1010000,.....}
- Minimal string = {101}
- After following above all rules we will get below designed DFA



Construction of DFA (Example)

- **Construct a DFA that accepts a language L over input alphabets ∑ = {a, b} such that L is the set of all strings starting with 'aa' or 'bb'.**

- L={aa, bb, aab , bba , bbb , aaa…..}
- Minimal string = (aa + bb)
- After following above all rules we will get below designed DFA



Parul®
University

DIGITAL LEARNING CONTENT

Non-Deterministic Finite Automata (NFA or NDFA)

Definition (NFA)

- **It is non-deterministic finite state automata which is used to accept or reject a sequence of string.**
- In NFA, for each input symbol, there will be more than one state i.e. combination of states. In other words the exact state to which the machine moves cannot be determined. Hence, it is called **Non Deterministic Automaton**. As it has a finite number of states, the machine is called **Non-Deterministic Finite Machine** or **Non-Deterministic Finite Automaton.**

Image source : Google

Formal definition of NDFA or NFA

- **A DFA can be represented by a 5-tuple (Q, ∑, δ, q_0, F) where –**

  Q is a finite set of states.

  ∑ is a finite set of symbols called the alphabet.

  δ is the transition function where δ: Q × ∑ → 2^Q

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

  q_0 is the initial state from where any input is processed ($q_0 \in Q$).

  F is a set of final state/states of Q ($F \subseteq Q$).

Image source : Google

Graphical Representation of a NDFA or NFA

- A NDFA is represented by digraphs called **state diagram**.
- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

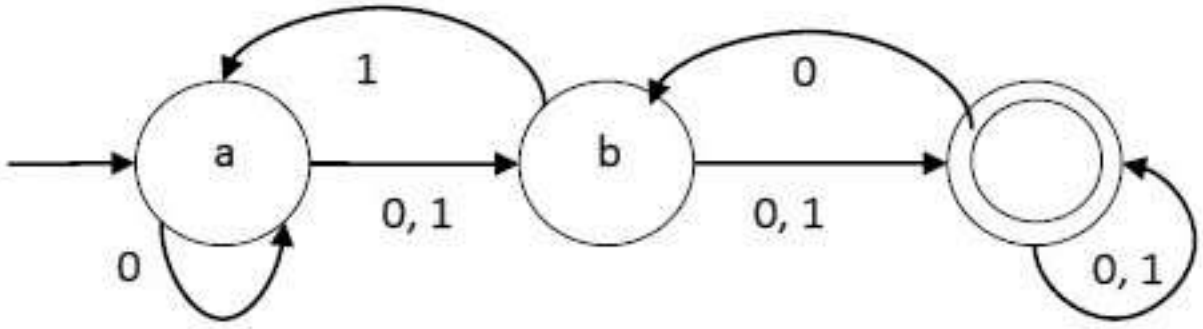


Image source : Google

Graphical Representation of a NDFA or NFA

- A NDFA is represented by digraphs called **state diagram**.
- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

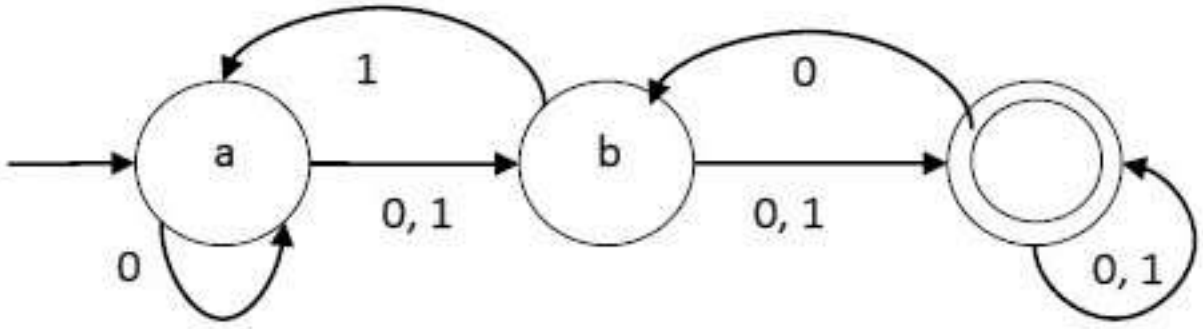


Image source : Google

Difference between DFA and NFA

DFA	NFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called *deterministic*.	The transition from a state can be to multiple next states for each input symbol. Hence it is called *non-deterministic*.
Empty string transitions are not seen in DFA.	NDFA permits empty string transitions.
Backtracking is allowed in DFA	In NDFA, backtracking is not always possible.
Requires more space	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NDFA, if at least one of all possible transitions ends in a final state.

Equivalence of NDFA and Regular Expression

- **If a language is accepted by NDFA then L is denoted by a Regular expression.**
- Regular expression and Non-deterministic FA, both have the same computational power.
- For each and every regular expression we can design NDFA such that *L(R)=L(NF).*

L(R)= Language accepted by regular expression

L(NF)= Language accepted by non-deterministic finite automata

Parul®
University

DIGITAL LEARNING CONTENT

Regular Grammar

Grammars

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

- N or V 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 ∈ N
- P is Production rules for Terminals and Non-terminals. A production rule has the form α → β, where α and β are strings on V U ∑ and least one symbol of α belongs to V

Grammars: example

(({S, A}, {a, b}, S,{S → aAb, aA → aaAb, A → ε })

- Here,
  - **S** and **A** are Non-terminal symbols.
  - **a** and **b** are Terminal symbols.
  - **ε** is an empty string.
  - **S** is the Start symbol, S ∈ N
  - Production **P : S → aAb, aA → aaAb, A → ε**

Derivations from a Grammar

- Strings may be derived using the productions in a grammar.
- For Example

Let us consider the grammar −

$G2 = (\{S, A\}, \{a, b\}, S, \{S \rightarrow aAb, aA \rightarrow aaAb, A \rightarrow \varepsilon\})$

- Derivation of the string: 'aaabbb'

$S \Rightarrow \underline{aA}b$ using production $S \rightarrow aAb$

$\Rightarrow a\underline{aA}bb$ using production $aA \rightarrow aAb$

$\Rightarrow aaa\underline{A}bbb$ using production $aA \rightarrow aAb$

$\Rightarrow aaabbb$ using production $A \rightarrow \varepsilon$

Grammar: Regular grammars

- The set of all strings that can be derived from a grammar is said to be the language generated from that grammar.
- Regular Grammar generates Regular language which is accepted by Finite automata.
- Regular Grammar also known as Type 3 grammar.

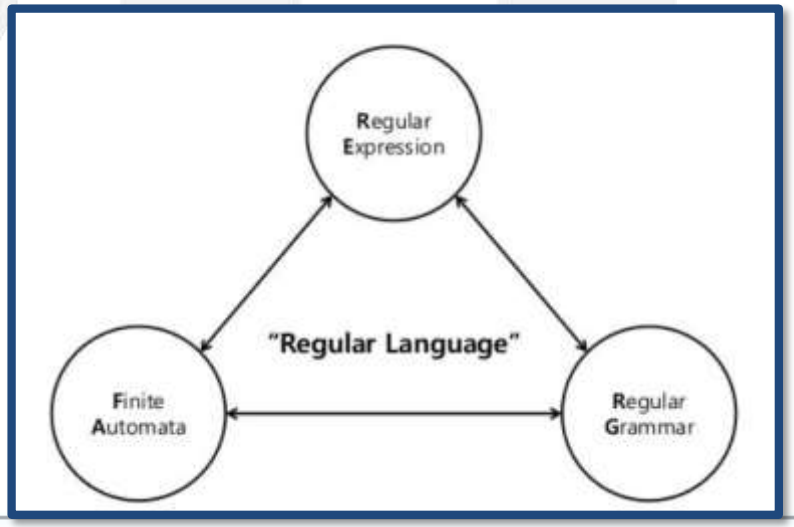


Image source : Google

Grammar: Regular grammars

RLG(**R**ight-**L**inear **G**rammar) : A → tB, A → t

P : S → aA

A → bA | b

- A *right linear grammar* is a linear grammar in which the non-terminal symbol always occurs on the right side.

LLG(**L**eft-**L**inear **G**rammar) : A → Bt, A → t

P : S → Aa

A → Ab | b

- A *left linear grammar* is a linear grammar in which the non-terminal symbol always occurs on the left side.

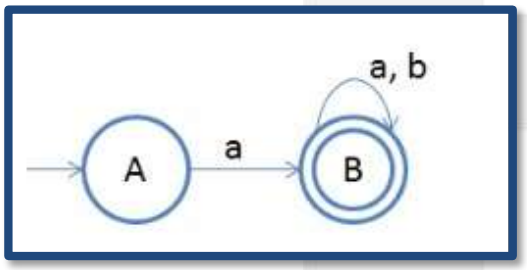
Equivalence with finite automata: FA → RLG

- Steps for Converting Finite Automata to Right linear 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: FA → RLG

Steps:



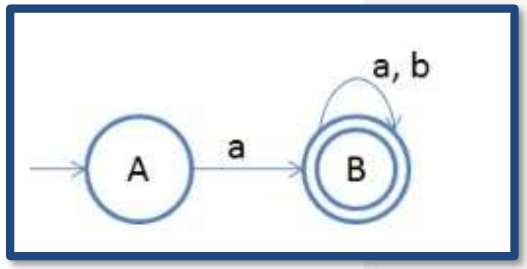
Pick start state and output is on symbol 'a' we are going on state B

So we will write as :

A -> aB

Example: FA → RLG

Steps:



And then we will pick state B and then we will go for each output.

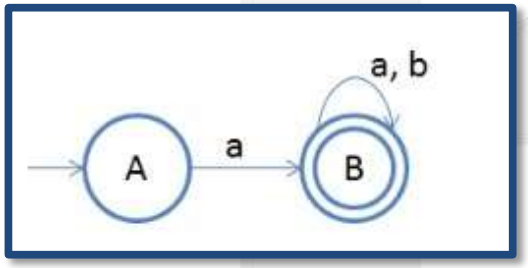
so we will get the below production.

B -> aB/bB/ε

- **B is final state so we have to add ε**

Example: FA → RLG

Final answer:



So final we got right linear grammar as:

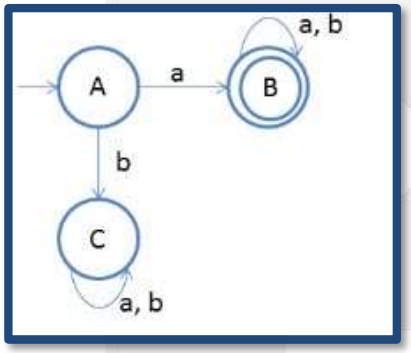
A -> aB
B -> aB/bB/ε

Equivalence with finite automata: FA → LLG

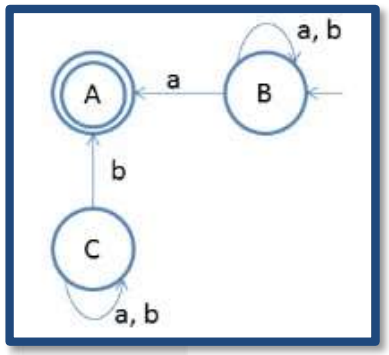
- Steps for Converting Left linear grammar to Finite Automata:
  - Take reverse of the finite automata
  - Then write right linear grammar following the previous lesson
  - Then take reverse of the right linear grammar
  - And you will get the final left linear grammar

Example: FA → LLG

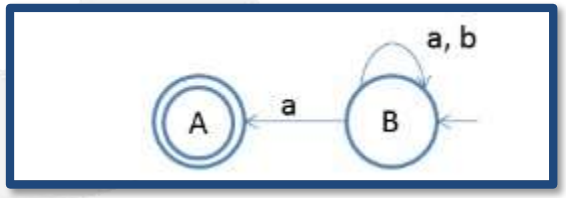
Step 1:Take reverse of the finite automata



Make final state as initial state and vice versa



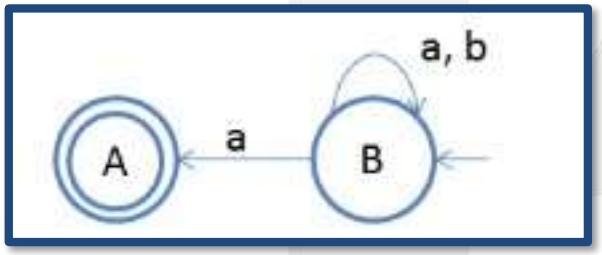
Remove unreachable states



But this is not DFA because **A** has no output
And on **B** we are having two outputs on 'a'.
So the result is a **NFA**

Example: FA → LLG

Step 2:Then write right linear grammar following the previous lesson



we will write down "right linear grammar" for it.

B -> aB/bB/aA
A -> ε

Example: FA → LLG

Step 3: Then take reverse of the right linear grammar

B -> aB/bB/aA
A -> ε

reverse the right linear grammar to get the ouput **= left linear Grammar**

B -> Ba/Bb/Aa
A -> ε

Formula

FA -> Reverse of FA -> Right Linear Grammar -> Reverse of Right Linear Grammar = Left Linear Grammar

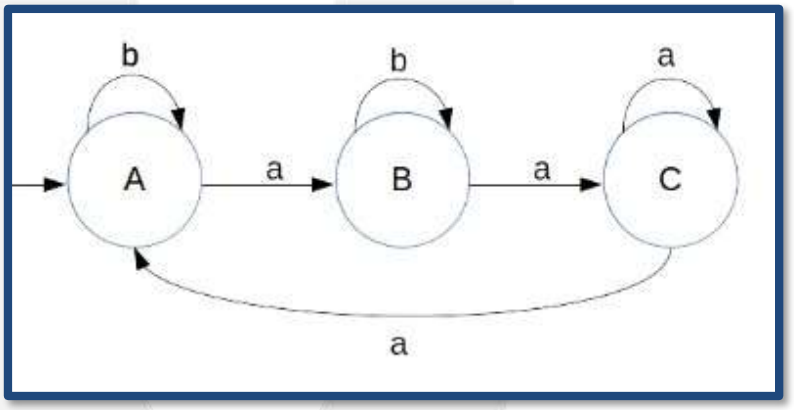
Equivalence with finite automata: RLG → FA

- Conversion from RLG and LLG to FA is very simple.

- Steps for Converting right linear grammar to Finite Automata:
  - Start from the first production
  - Start State: It will be the first production's state
  - And then for every left alphabet go to SYMBOL followed by it
  - Final State: Take those states which end up with input alphabets.

Example: RLG → FA

- A -> aB/bA/b
- B -> aC/bB
- C -> aA/bC/a



Equivalence with finite automata: LLG → FA

- Steps for Converting left linear grammar to Finite Automata:
  - Take reverse of LLG
  - Create Finite automata using previous example
  - Then again take reverse of the FA and that will be our final output
  - Start State: It will be the first production's state
  - Final State: Take those states which end up with input alphabets

Formula

Left Linear Grammar -> Reverse of left Linear Grammar-> FA -> Reverse of FA

Example: LLG → FA

Step 1:

left linear grammar

A -> Ba/Ab/b

B -> Ca/Bb

C -> Aa/Cb

Take the reverse of the LLG

Right linear grammar

A -> aB/bA/b

B -> aC/bB

C -> aA/bC/a

Example: LLG → FA

Step 2:

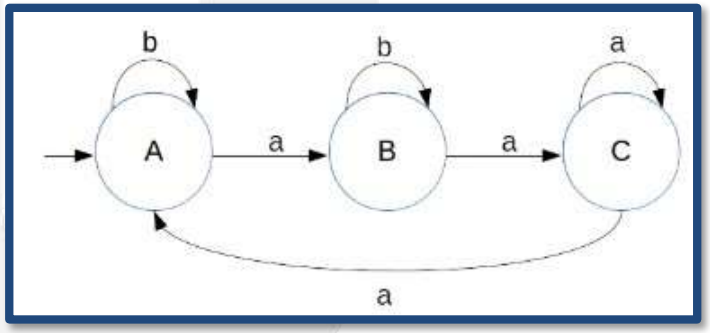
Right linear grammar

A -> aB/bA/b

B -> aC/bB

C -> aA/bC/a

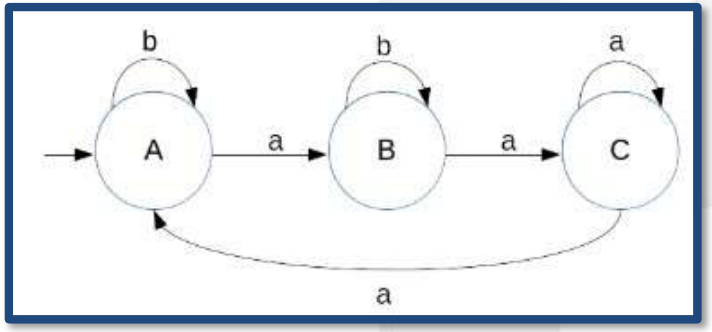
Now create the FA



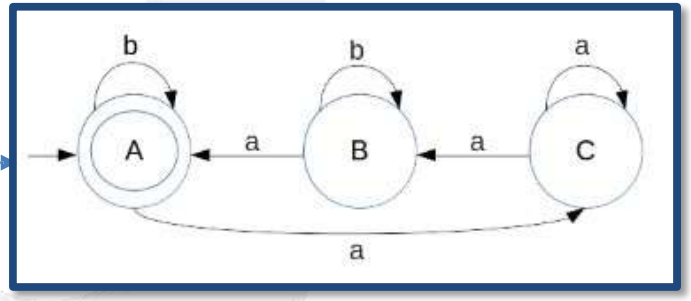
Only state A is final state in this example

Example: LLG → FA

Step 3:



Now take reverse of the above FA (this is final output)



Here state A was itself a final state that is why it again become start state as well as final state.

Parul®
University

DIGITAL LEARNING CONTENT

Properties of regular languages

Properties of regular languages

- Regular languages have two important kinds of properties:

1. Closure properties.
2. Decision properties.

Parul® University

Properties of regular languages: Closure properties

- **Closure properties** on regular languages are defined as certain operations on regular language which are guaranteed to produce regular language.

- Closure refers to some operation on a language, resulting in a new language that is of same “type” as originally operated on i.e., regular.

Properties of regular languages: Closure properties

- Regular languages are closed under following operations.
- Let L and M be regular languages. Then the following languages are all regular:
  - Union: L U M
  - Intersection: L ∩ M
  - Complement: L
  - Difference: L - M = L ∩ M
  - Concatenation: L . M
  - Reversal: L^R
  - Kleen Closure: L^*
  - Positive Closure: L^+

Closure properties: Closure under Complementation

- If $L \subseteq \Sigma^*$, then the complement of L, denoted L, is $\Sigma^* - L$.

•Theorem: *If L is a regular language over Σ, then L is also a regular language.*

- Proof:
  - Construct a DFA for L
  - This can be transformed into a DFA for L by making all accepting states non-accepting and vice versa.

Closure properties: Closure under Reversal

- The reversal of L, written L^R is $\{x \mid x^R \in L\}$
- The reversal of a language L, is the language consisting of the reversals of all its strings
- Example: L = {001, 10, 111}

  L^R = {100, 01, 111}
- Theorem: If L is regular, then so is L^R.
- Proof:
  - Start with a DFA for L.
  - Reverse all of the transitions in the DFA
  - Make the DFA's start state the only accepting state.
  - Create a new start state with ε-transitions to all of the original accepting states.

Properties of regular languages : Decision properties

- A property is a yes/no question about languages.
- Some examples:
  - ❖ Is L empty?
  - ❖ Is L finite?
  - ❖ Are $L1$ and $L2$ equivalent?

•A property is a decision property for regular languages if an algorithm exists that can answer the question (for regular languages).

Properties of regular languages : Decision properties

- Following are the decision properties for FA:

1. Emptiness
2. Finiteness
3. Equivalence
4. Membership

Decision properties: Emptiness and Non-emptiness

- Does the language contain any string at all?

- Algorithm:
  - Select the state that cannot be reached from the initial states & delete them (remove unreachable states).
  - If the resulting machine contains at least one final states, so then the finite automata accepts the non-empty language.
  - if the resulting machine is free from final state, then finite automata accepts empty language.

Decision properties: Finiteness and Infiniteness

- Is the language accepted by FA is finite or infinite?

- Algorithm:
  - Select the state that cannot be reached from the initial state & delete them (remove unreachable states).
  - Select the state from which we cannot reach the final state & delete them (remove dead states).
  - If the resulting machine *contains loops or cycles* then the finite automata accepts *infinite language*.
  - If the resulting machine do not contain loops or cycles then the finite automata accepts finite language.

Decision properties: Equivalence

- We want an algorithm that takes two languages, $L1$ and $L2$, and determines if they are the same?

- Algorithm:
  - Convert $L1$ and $L2$ to DFAs.
  - Convert $L1$ and $L2$ to minimal DFAs. (Refer slide no 40)
  - Determine if the minimal DFAs are the same.

Decision properties: Membership

- Membership is a property to verify an arbitrary string is accepted by a finite automaton or not? i.e. it is a member of the language or not?

- Let M is a finite automata that accepts some strings over an alphabet, and let ‘w’ be any string defined over the alphabet,
  - if there exist a transition path in M, which starts at initial state & ends in anyone of the final state, then string ‘w’ is a member of M, otherwise ‘w’ is not a member of M.

Parul®
University

DIGITAL LEARNING CONTENT

Pumping lemma for regular languages

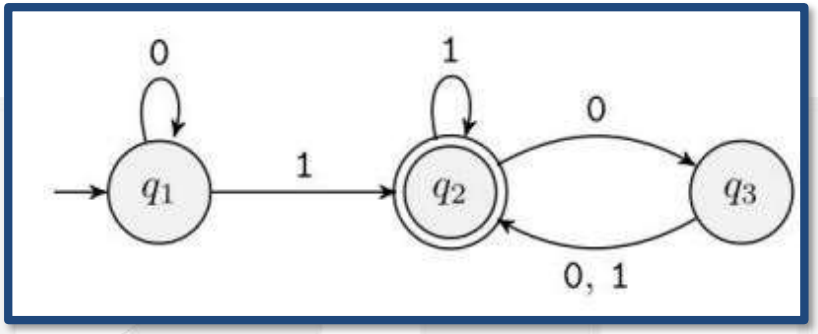
Pumping lemma for regular languages

We need a tool to prove that a language is NOT Regular

Important Ideas:

- **Pumping:**
  - Repeating a section of the string an arbitrary number of times (≥0), with the resulting string remaining in the language.
- **Pumping Length :**
  - "All string in the language can be pumped if they are at least as long as a certain special value, called the pumping length."

Pumping lemma for regular languages



Examples:

101 ⇒ 1(01)*

000001 ⇒ (0)*1

110100 ⇒ 1(1)*0100

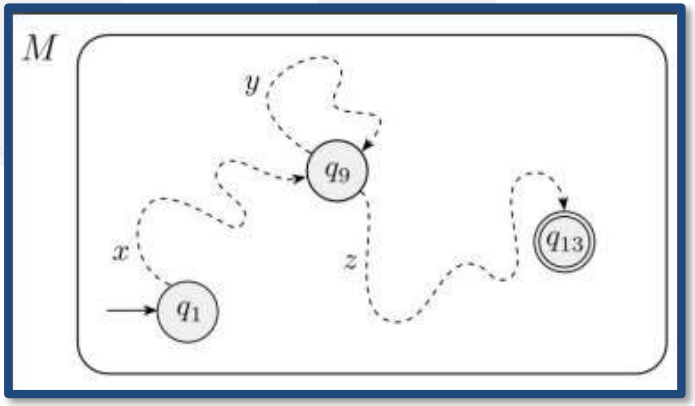
Pumping lemma for regular languages

If A is a Regular Language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into 3 pieces, s = xyz, satisfying the following conditions:

a. For each $i \geq 0$, $xy^iz \in A$,

b. $|y| > 0$, and

c. $|xy| \leq p$.



Pumping lemma for regular languages

Application:

•The pumping lemma is extremely useful in proving that certain sets are non-regular. The general methodology followed during its applications is :

- Select a string s in the language L.
- Break the string s into x, y and z in accordance with the above conditions imposed by the pumping lemma.
- Now check if there is any contradiction to the pumping lemma for any value of i.

Example: Use the pumping lemma to show that following language is not regular: L = {0n1n|n>=1}

Assume L is Regular, then Pumping Lemma must hold.

p is the pumping length given by the PL.

Choose s to be 0^p1^p.

Because s ∈ L and |s| ≥ p, PL guarantees s can be split into 3 pieces, s = xyz, where for any i ≥ 0, xy^iz ∈ L.

Example: Use the pumping lemma to show that following language is not regular: L = {0n1n|n>=1}

- Consider 3 cases:

1. y is only 0s. xyyz has more 0s than 1s, thus a contradiction via condition 1 of PL.
2. y is only 1s. Also a contradiction.
3. y is both 0s and 1s. xyyz may have same number of 1s and 0s, but will be out of order, with some 1s before 0s, also a contradiction.

- Contradiction is unavoidable, thus L is not Regular.

Exercise

Q1: Use Pumping Lemma to show that following language is not regular. $L = \{ ww^R / w \varepsilon \{0,1\}^* \}$

Q2: Prove that the language $L = \{0^n: n \text{ is a prime number}\}$ is not regular.

Parul®
University

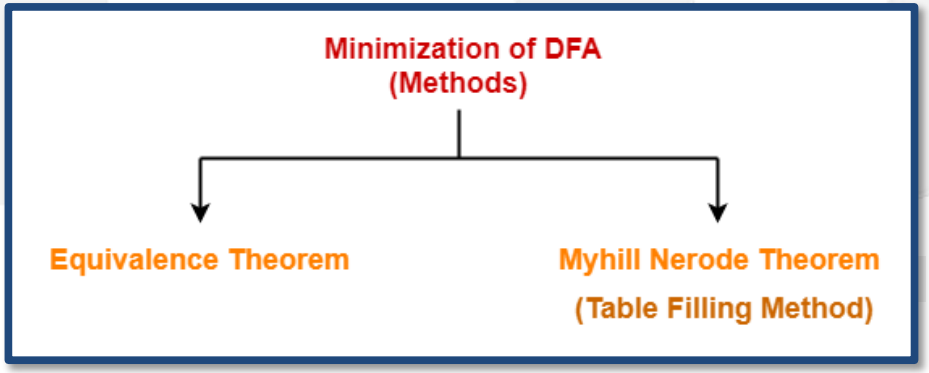
DIGITAL LEARNING CONTENT

Minimization of finite automata

Minimization of finite automata

- DFA minimization stands for converting a given DFA to its equivalent DFA with minimum number of states.

- For ant regular language, more than one DFA can possible but there is only one minimal DFA for that language.



Minimization of finite automata: Equivalence Theorem

- **Step 1:**
  - Eliminate all the inaccessible(unreachable) states from the given DFA (if any).
- **Step 2:**
  - Draw the transition table for all pair of states.
- **Step 3:**
  - We will divide Q (set of states) into two sets. One set will contain all final states and other set will contain non-final states. This partition is called P_0 (0 equivalence)
- **Step 4:**
  - Initialize k = 1

Minimization of finite automata: Equivalence Theorem

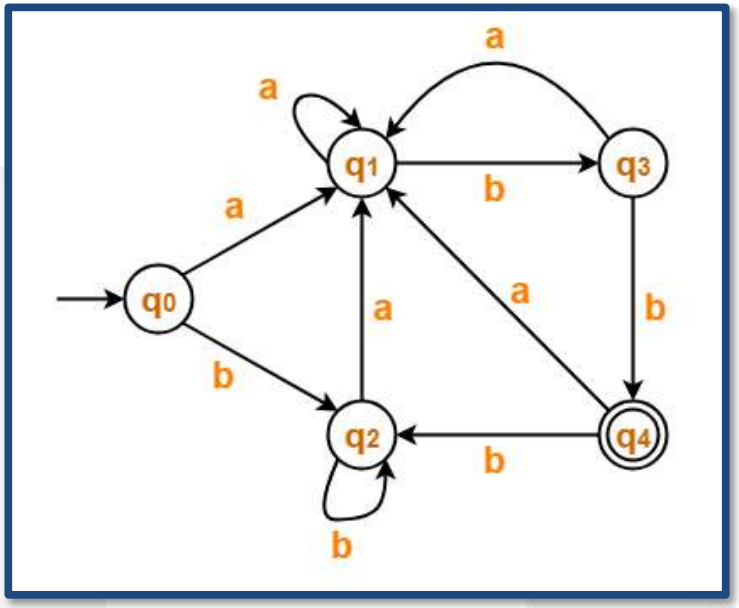
- **Step 5:**
  - Find P_k by partitioning the different sets of P_{k-1}.
  - In each set of P_{k-1}, we will take all possible pair of states.
  - If two states of a set are distinguishable, we will split the sets into different sets in P_k.
- **Step 6:**
  - Stop when $P_k = P_{k-1}$ (No change in partition)
- **Step 7:**
  - All states of one set are merged into one. No. of states in minimized DFA will be equal to no. of sets in P_k.

Minimization of finite automata: Equivalence Theorem

- **How to find whether two states in partition P_k are distinguishable ?**

Two states (qi, qj) are distinguishable in partition P_k if for any input symbol a, δ (qi, a) and δ (qj, a) are in different sets in partition P_{k-1}.

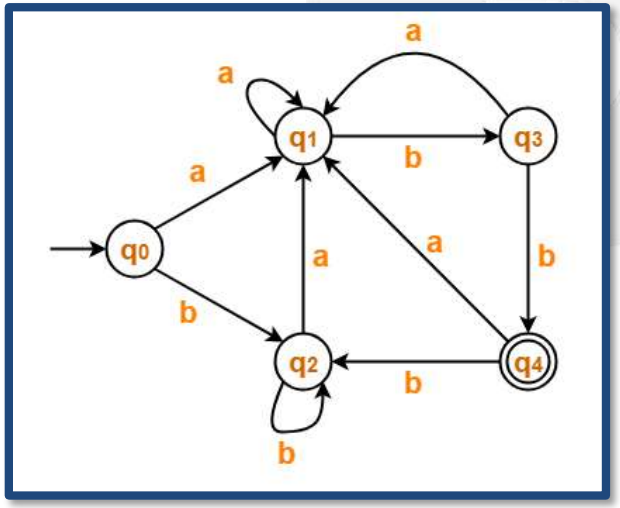
Equivalence Theorem: Example1



- The given DFA contains no dead states and inaccessible states

Equivalence Theorem: Example1

Draw a state transition table.



	a	b
→q0	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	*q4
*q4	q1	q2

Equivalence Theorem: Example1

Now using Equivalence Theorem, we have-

$P_0 = \{ q_0 , q_1 , q_2 , q_3 \} \{ q_4 \}$

$P_1 = \{ q_0 , q_1 , q_2 \} \{ q_3 \} \{ q_4 \}$

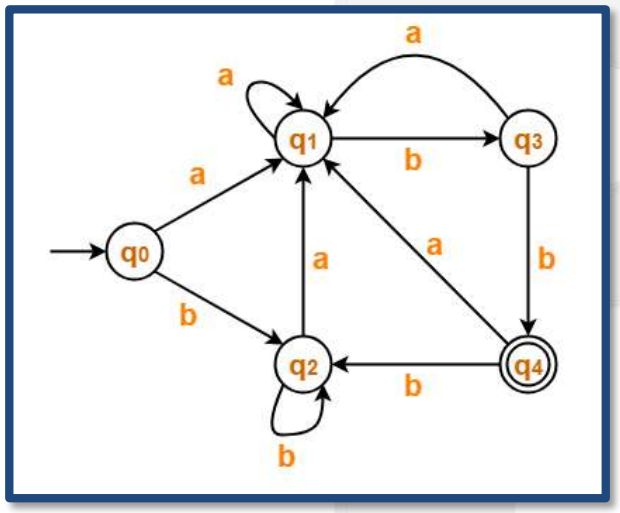
$P_2 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$

$P_3 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$

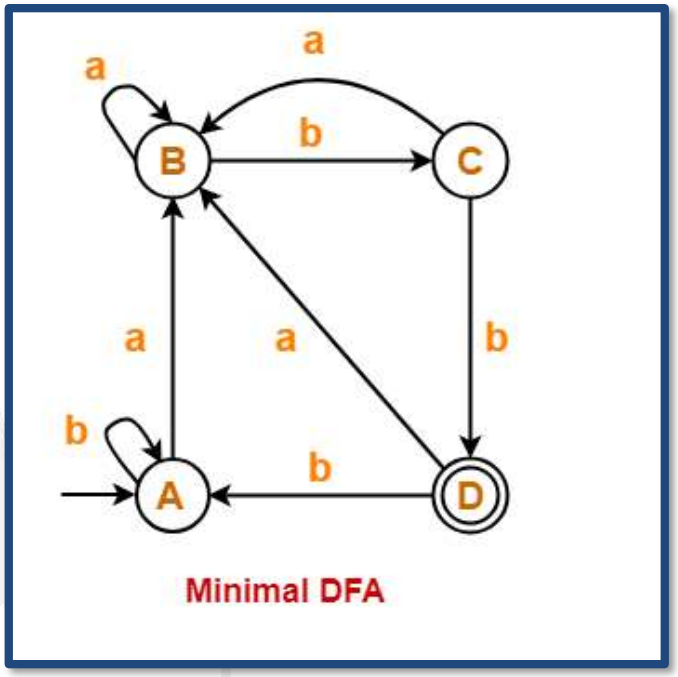
Since $P_3 = P_2$, so we stop.
From P_3, we infer that states q_0 and q_2 are equivalent and can be merged together.

	a	b
→q0	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	*q4
*q4	q1	q2

Equivalence Theorem: Example1

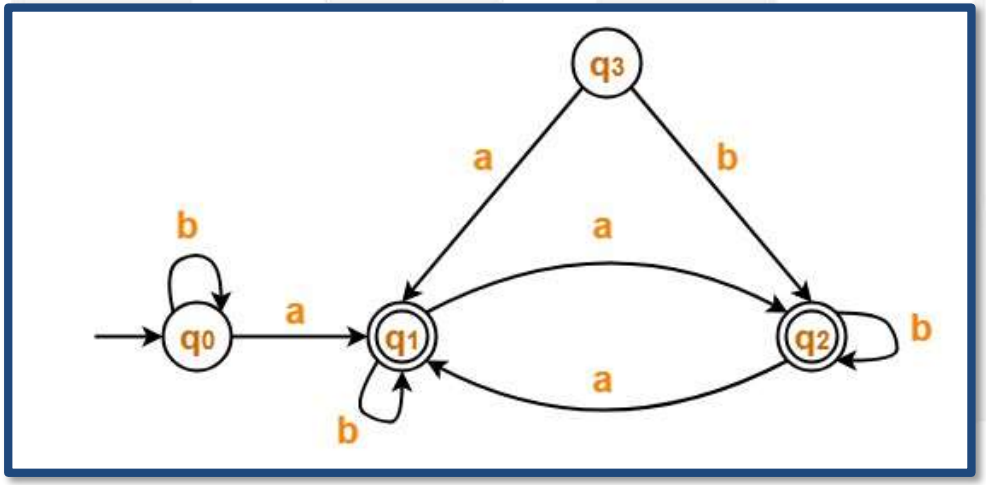


So, Our minimal DFA is-

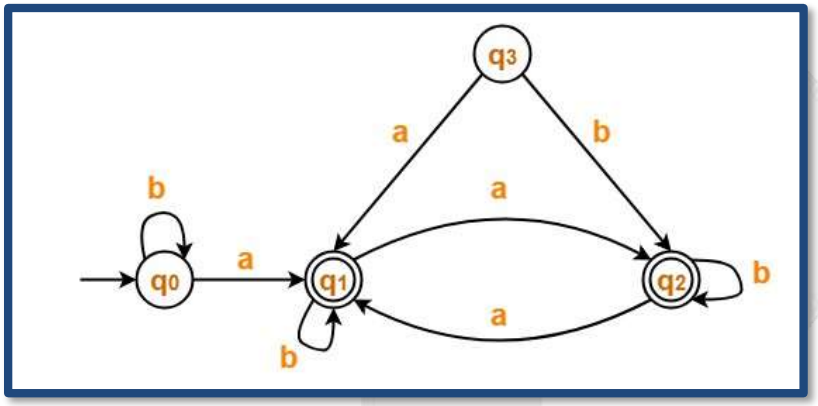


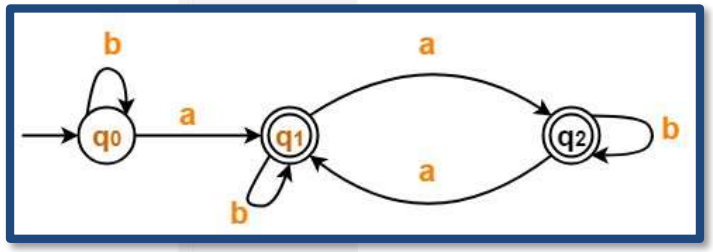
Equivalence Theorem: Example2

Minimize following DFA



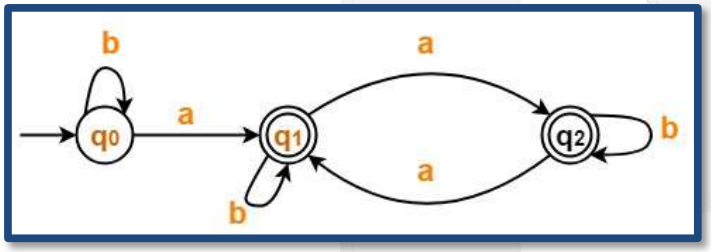
Equivalence Theorem: Example2





State q_3 is inaccessible from the initial state. So, we eliminate it and its associated edges from the DFA.

Equivalence Theorem: Example2



Draw a state transition table

	a	b
→q0	*q1	q0
*q1	*q2	*q1
*q2	*q1	*q2

Equivalence Theorem: Example2

Now using Equivalence Theorem, we have-

$P_0 = \{ q_0 \} \{ q_1 , q_2 \}$
$P_1 = \{ q_0 \} \{ q_1 , q_2 \}$

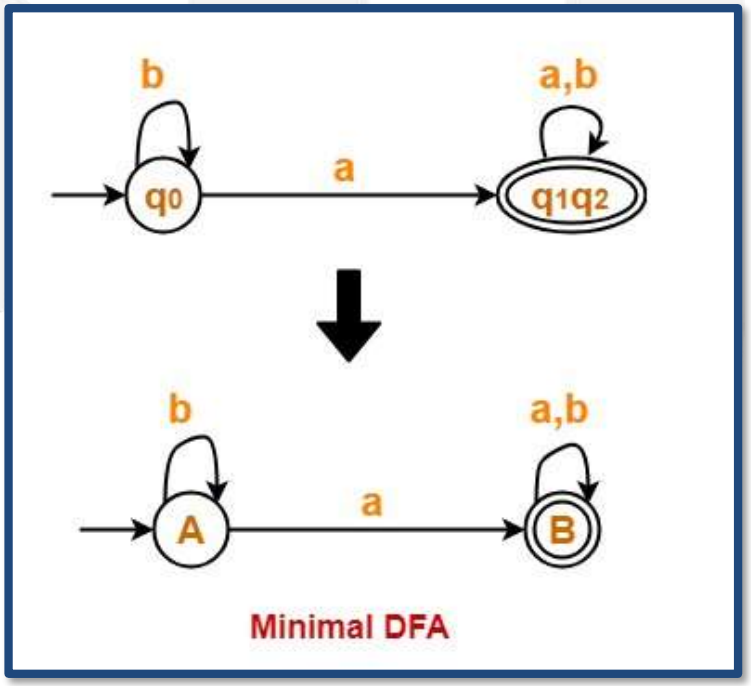
Since $P_1 = P_0$, so we stop.

	a	b
→q0	*q1	q0
*q1	*q2	*q1
*q2	*q1	*q2

Equivalence Theorem: Example2

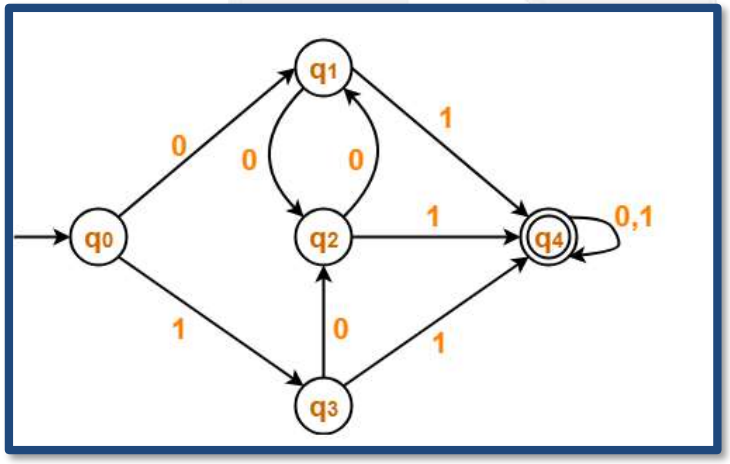
From P_1, we infer that states q_1 and q_2 are equivalent and can be merged together.

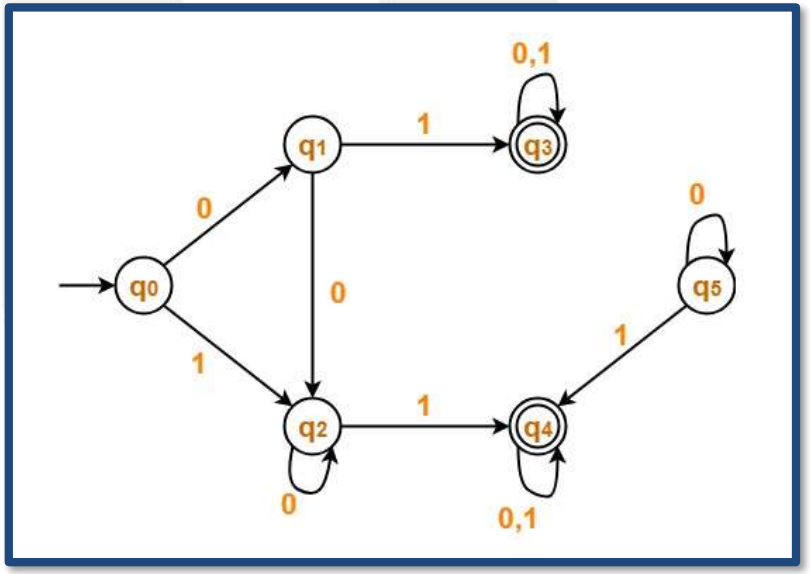
So, Our minimal DFA is-



Exercise

Minimize the following DFAs





DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in