UNIT - II

Regular Expressions

Finite Automata and Regular Expressions, Applications of Regular Expressions, Algebraic Laws for Regular Expressions, Conversion of Finite Automata to Regular Expressions.

Pumping Lemma for Regular Languages, Statement of the pumping lemma, Applications of the Pumping Lemma.

Closure Properties of Regular Languages

Closure properties of Regular languages, Decision Properties of Regular Languages, Equivalence and Minimization of Automata.

UNIT - II

REGULAR EXPRESSIONS

⇒ The language accepted by FA [Finite Automata] can easily described by simple expression called Regular Expression.

* It is most effective way to represent any language.

⇒ The language accepted by some regular expressions are referred to as regular language. (ie) L accepted by RE

⇒ A regular expressions can be described as a sequence of pattern that defines a string.

For instance,

In R.E x^* means zero or more occurence of x

It can generate $\{\varepsilon, x, xx, xxx, \ldots\ldots\}$

In R.E x^+ means one or more occurence of x.

It can generate $(x, xx, xxx, \ldots\}$

The Operators of RE:-

Regular Expressions denote languages.

1) Union: If L and M are two regular languages their union LUM is the set of strings. either L or M $S \begin{smallmatrix} \nearrow L \\ \searrow M \end{smallmatrix}$

(ie) $L \cup M = \{S \mid S \text{ is in } L \text{ or } S \text{ is in } M\}$

Ex: if $L = \{001, 10, 111\}$ and $M = \{\varepsilon, 001\}$

$L \cup M = \{\varepsilon, 10, 001, 111\}$

2). Concatenation:- If L and M are two regular languages.

L and M is the set of strings

It can be formed by taking any string in L and concatenating it with any string in M.

The concatenation operator is frequently called 'dot.'

Ex: if $L = \{001, 10, 111\}$ and $M = \{\varepsilon, 001\}$

$L.M$ or $LM = \{001, 10, 111, 001001, 10001, 111001\}$

Intersection: If L and M are two regular languages and there intersection is also an intersection.

there intersection

$L \cap M = \{st \mid s$ is in L and t is in $M\}$ $L \to s$ $M \to t$

3) Star or Kleene Closure:

~~The~~ If L is regular language then it is denoted Kleen Closure L^*

w is also be regular language.

L^* = Zero or more occurence by language L.

$= \epsilon, (\_\_), (\_\_\_), (\_\_\_\_)$

Building Regular Expressions:

BASIS: The Constants ϵ and ϕ are RE, denoting the language $\{\epsilon\}$ and ϕ, respectively.

(ie) $L(\epsilon) = \{\epsilon\}$ and $L(\phi) = \phi$.

If a is any Symbol, then a is a RE. This expression denotes the language $\{a\}$.

(ie). $L(a) = \{a\}$.

A Variable, usually Capitalized and Italic such as L, is a Variable, representing is any language.

INDUCTION: There are four parts to the inductive step, one for each of the three operators and one for the introduction of parantheses.

1) If E and F are two RE, $E+F$

The union of $L(E)$ and $L(F)$. (ie) $L(E+F) = L(E) \cup L(F)$

2) If E and F are RE, EF

The Concatenation of $L(E)$ and $L(F)$. (ie) $L(EF) = L(E)L(F)$

3) If E is a RE, E^*

The closure of $L(E)$ (ie) $L(E^*) = (L(E))^*$.

4) If E is a RE, $(E) \Rightarrow$ parenthesized E.

denoting same language as E. (ie) $L((E)) = L(E)$

50

Precedence of RE Operators :-

The RE Operators have an assumed order of "precedence" which means that Operators are associated with their Operands in a particular order.

The order of Precedence for the operators

The Star Operator is of highest Precedence.

The concatenation or dot operator.

Finally all unions (+ operators) are grouped with their Operands.

Examples of RE :-

1) Write RE for the language accepting all string containing any number of a's and b's.

Soln: $R.E = (a+b^*)$

This will given the set as

$L = \{\epsilon, a, aa, b, bb, ab, ba, aba, \ldots\}$ any combination of a&b

2). Write RE for the language accepting all the string which are starting with 1 and ending with 0 over $\Sigma = \{0,1\}$

Soln: The first symbol is 1
The last symbol is 0.

$R.E = 1(0+1)^*0$

3) Write RE for the language starting with a but not having connective b's

(ie). $L = [a, aba, aab, ada, bbb \ldots]$

$R.E = \{a^{\vee} + ab\}^*$.

FINITE AUTOMATA AND REGULAR EXPRESSIONS

The RE approach to describing languages is fundamentally different from the finite-automaton approach, these two notations turn out to represent exactly the same set of languages, have termed the "regular languages".

The FA, that DFA and the two kinds of NFA - with and without ε-transitions - accept the same class of languages.

To show that the regular expressions define the same class.

1) Every language defined by one of these automata is also defined by a regular expression.

Proof: We can assume the language is accepted by some DFA.

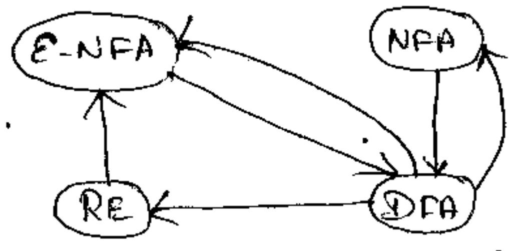
2) Every language defined by a regular expressions is defined by one of these automata.

Proof: The easiest is to show that there is an NFA with ε-transitions accepting the same language.

<u>From DFA's to Regular Expressions:-</u>

The Construction of a regular expression to define the language of any DFA.

The paths are allowed to pass through only a limited subset of the states.



Plan for showing the equivalence of four different notations for regular languages.

<u>Theorem:</u> If $L=L(A)$ for some DFA, A, then there is a regular expression R such that $L=L(R)$.

Proof: Let us suppose that A's states are $\{1, 2, \dots n\}$ for some integer n.

The first n positive integers.

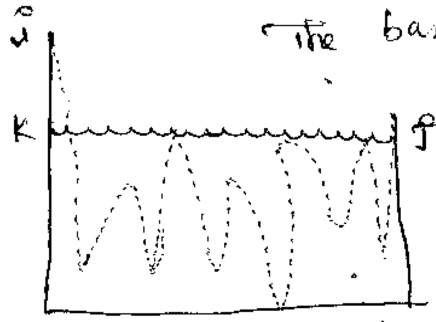
Our first, and most difficult, task is to Construct a Collection of regular expressions that describes progressively broader sets of Paths in the transition diagram of A.

Let us use $R_{ij}^{(k)}$ as the name of a regular expression whose language is the set of strings w such that w is the label of a path from state i to state j in A.

To Construct the expressions $R_{ij}^{(k)}$, use the following inductive definition, starting at $k=0$ and finally reaching $k=n$.

Notice that when $k=n$, there is no restriction at all on the paths represented, since there are no states greater than n.

BASIS: The basis is $k=0$.



The path whose label is in the language of regular expression $R_{ij}^{(k)}$

Since all states are numbered 1 or above, the restriction on paths is that the path must have no intermediate states at all.

There are only two kinds of Paths that meet such a Condition:

1). An arc from node (state) i to node j.
2). A path of length 0 that Consists of only some node i.

If $i \neq j$, then only case (1) is possible

We must examine the DFA A and find those input symbols a such that there is a transition from state i to state j on symbol a.

a). If there is no such symbol a, then $R_{ij}^{(0)} = \phi$

b). If there is exactly, one such symbol a, then $R_{ij}^{(0)} = a$

c) If there are symbols $a_1, a_2, \ldots, a_k$ that label arcs from state i to state j, then $R_{ij}^{(0)} = a_1 + a_2 + \ldots + a_k$.

If $i = j$, then the legal paths are the path of length 0 and all loops from i to itself.

The path of length 0 is represented by the regular expression ε, since that path has no symbols along it.

Thus, we add ε to the various expressions devised in (a) through (c).

⇒ (ie). In case (a) [no symbol a] the expression becomes ε,

⇒ In case (b) [one symbol a] the expression becomes ε + a,

⇒ and in case (c) [multiple symbols] the expression becomes $ε + a_1 + a_2 + \ldots + a_k$

INDUCTION:

Suppose there is a path from state i to state j that goes through no state higher than k.

There are two possible cases.

1). The path does not go through state k at all. In this case, the label of the path is in the language of $R_{ij}^{(k-1)}$

2). The path goes through state k at least once.

We can break the path into several pieces,

The first goes from state i to state k without passing through k, the last piece goes from k to j without passing through k and all the pieces in the middle go from k to itself, without passing through k.

Note that if the path goes through state k only once, then there are no "middle" pieces, just a path from i to k and a path from k to j.

The set of labels for all paths of this type is represented by the regular expression

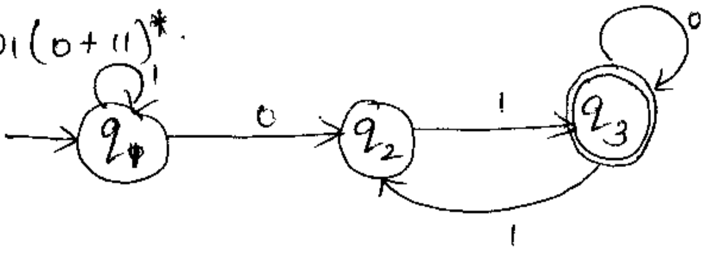
$$R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^{*} R_{kj}^{(k-1)}$$

When we combine the expressions for the paths of the two types above, we have the expression

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^{*} R_{ij}^{(k-1)}$$

Example:

Convert the DFA from in figure to a regular expression. Convert the DFA from in figure to the regular clearly this DFA accepts all strings corresponding to the regular expression $1^{*}01(0+11)^{*}$.



Solution:

$$\text{W.K.T} \Rightarrow R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ij}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^{*} R_{ij}^{(k-1)}$$

i - start state, j - final state, k - intermediate state

$$\left[\text{ie} \left(R_{ij}^{k}\right)\right]$$

$R_{11}^{(0)} = 1+\varepsilon$	$R_{11}^{(1)} = 1^{*}$	$R_{11}^{(2)} = 1^{*}$
$R_{12}^{(0)} = 0$	$R_{12}^{(1)} = 1^{*}0$	$R_{12}^{(2)} = 1^{*}0$
$R_{13}^{(0)} = \phi$	$R_{13}^{(1)} = \phi$	$R_{13}^{(2)} = \phi\ 1^{*}01$
$R_{21}^{(0)} = \phi$	$R_{21}^{(1)} = \phi$	$R_{23}^{(2)} = \phi$
$R_{22}^{(0)} = \varepsilon$	$R_{22}^{(1)} = \varepsilon$	$R_{22}^{(2)} = \varepsilon$
$R_{23}^{(0)} = 1$	$R_{23}^{(1)} = 1$	$R_{23}^{(2)} = 1$
$R_{31}^{(0)} = \phi$	$R_{31}^{(1)} = \phi$	$R_{31}^{(2)} = \phi$
$R_{32}^{(0)} = 1$	$R_{32}^{(1)} = 1$	$R_{32}^{(2)} = 1$
$R_{33}^{(0)} = 0+\varepsilon$	$R_{33}^{(1)} = 0+\varepsilon$	$R_{33}^{(2)} = 0+\varepsilon+11$
(a)	(b)	(c)

Regular expressions for path that can go through
a) no state
b). state 1 only
and
c). state 1 and 2 only.

Now we apply the expression.

$\boxed{K=1}$

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)} (R_{11}^{(0)})^* R_{1j}^{(0)}$$

$$R_{12}^{(1)} = R_{12}^{(0)} + R_{11}^{(0)} (R_{11}^{(0)})^* R_{12}^{(0)}$$

$$= 0 + (1+\varepsilon)(1+\varepsilon)^* 0$$

$$= 0 + 1^* 0$$

$\because (\varepsilon+R)^* = R^*$

$(\varepsilon+1)^* = 1^*$

$(\varepsilon+1)(\varepsilon+1)^* = (\varepsilon+1)1^* = 1^*$.

$\boxed{K=2}$

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)} (R_{22}^{(1)})^* R_{2j}^{(1)}$$

$i=1, j=3$ & $K=3$

$$R_{13}^{(3)} = R_{13}^{(2)} + R_{13}^{(2)} (R_{33}^{(2)})^* R_{33}^{(2)}$$

$$= 1^*01 + 1^*01(0+\varepsilon+11)^*(0+\varepsilon+11)$$

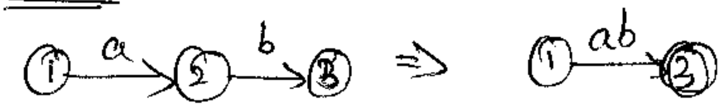
$\because (0+\varepsilon+11)^* = (0+11)^*$

$$\boxed{R_{13}^{(3)} = 1^*01(0+11)^*.}$$

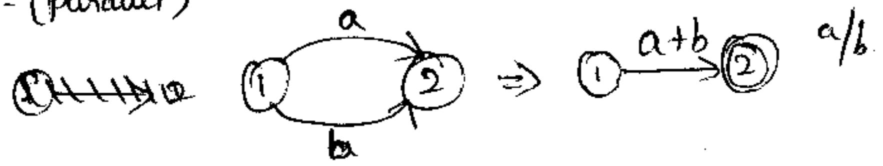
Conversion of Finite Automata to Regular Expressions:-

State Elimination Method:-

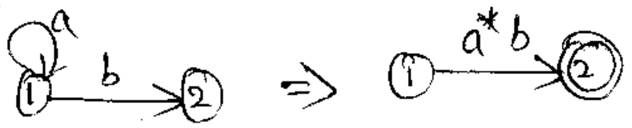
Concatenation :-



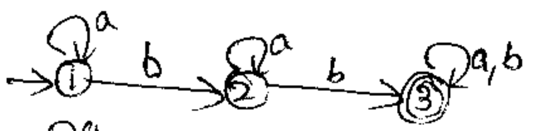
Union:- (parallel)

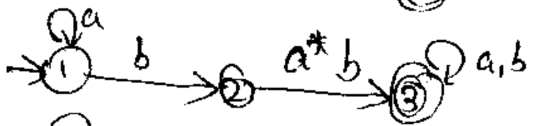


Closure:

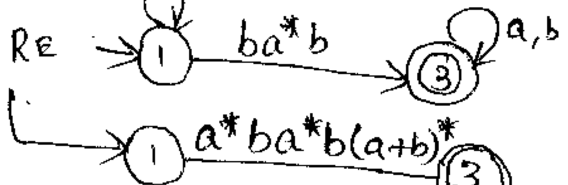


Example:





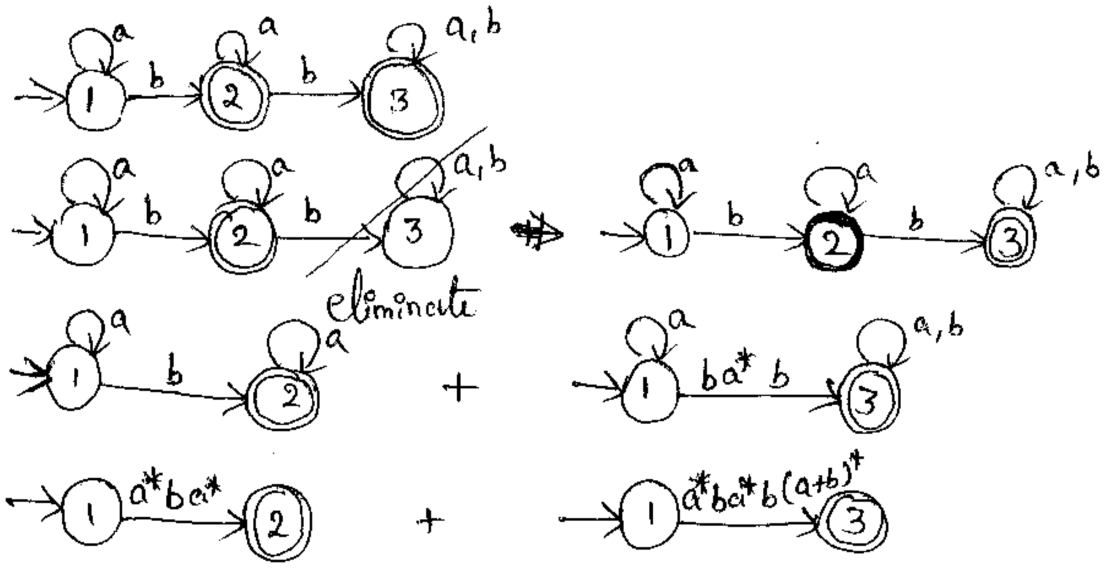
Single RE



$\Rightarrow a,b = (a+b)^*$

56

DFA



$\Rightarrow a^* b a^* + a^* b a^* b(a+b)^*$

CONVERTING DFA's TO REGULAR EXPRESSIONS

The two popular methods for converting a DFA to its regular expression are

Converting DFA to Regular Expression
(Methods)

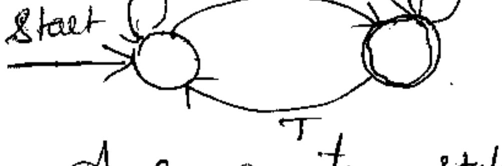
- Arden's Method
- State Elimination Method

By using State Elimination Method

This method involves the following steps in finding the regular expression for any given DFA

Step-01:

For each accepting state q, apply the above reduction process to produce an equivalent automaton with regular-expression labels.



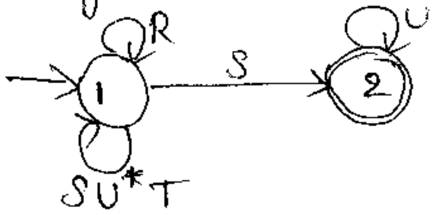
A Generic two-state automaton

57

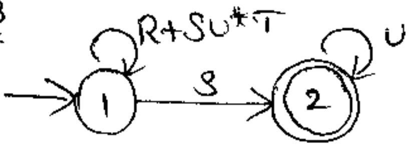
Step 2:

If $q_* \neq q_0$,

The Regular expression for the accepted strings

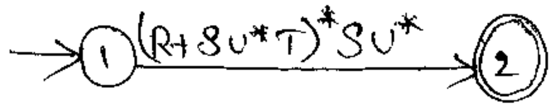


Step 3



→ If the start state is also an accepting state,
Must also perform a state-elimination from the original automaton

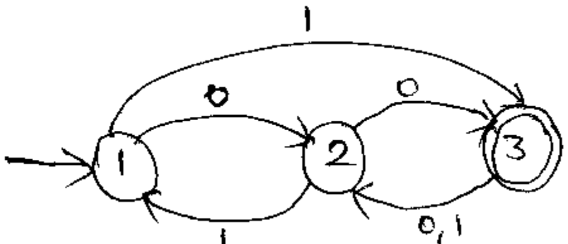
Step 4:



The desired regular expression is the sum (union) of all the expressions derived from the reduced automata for each accepting state, by rules (2) and (3)

Example:

Two looping conditions



Solution:

Here we want to eliminate intermediate ~~state~~ 2. state.

Direct path by using intermediate

$11 - \phi + 01 = 01$

$13 - 1 + 00 = 1+00$

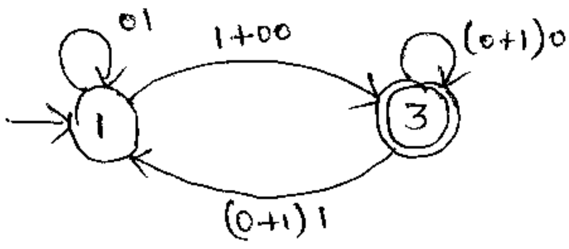
$31 - \phi + (0+1)1 = (0+1)1$

$33 - \phi + (0+1)0 = (0+1)0$

Path 1 to 2

1 to 2 to 1
(ie) 12
21

1 to 2 to 3
3 - 2 - 1
3 - 2 - 3



Based on above.

R → 01
S → 1+00
u → (0+1)0
T → (0+1)1

RE: ⇒

$$\left[\underbrace{01}_{R} + \underbrace{[(1+00)}_{S}\,\underbrace{[((0+1)0)^*}_{u^*}\,\underbrace{(0+1)1]}_{T}\right]^* \underbrace{(1+00)}_{S}\,\underbrace{((0+1)0)^*}_{u^*}$$

Converting Regular Expression to Finite Automata:

To Convert RE to FA, we use a method called Subset method. $RE \rightarrow NFA\text{-}\epsilon \rightarrow NFA \cong DFA$.

Step 1:

Design a transition diagram for given R.E, using NFA with ϵ-moves

Step 2: Convert NFA with ϵ to NFA without ϵ

Step 3: Convert the obtained NFA to equivalent DFA.

THEOREM: Every language defined by a regular expression is also defined by a finite automaton.

PROOF: Suppose $L = L(R)$ for a regular expression R. We Show that $L = L(E)$ for some ϵ-NFA E with:

1) Exactly one accepting state
2). No arcs into the initial state
3). No arcs out of the accepting state

BASIS:- There are three Parts

a). How to handle the expression ϵ.

The language of the automaton (ie) $\{\epsilon\}$,

$\because$ The only Path from the state State to an accepting State is labeled ϵ.

b). It shows the construction for ϕ.

There are no paths from start state to accepting state, So ϕ is the language of this automaton.

c). Gives the automaton for a regular expression a.

59

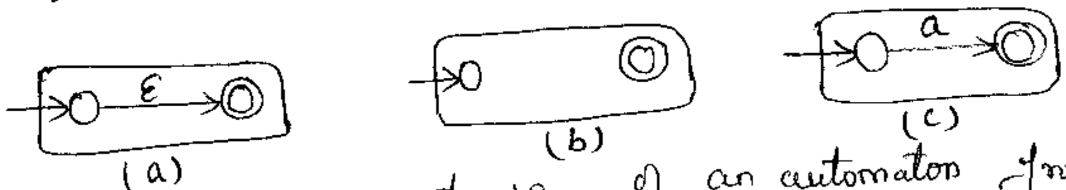
INDUCTION:

The three parts of the induction

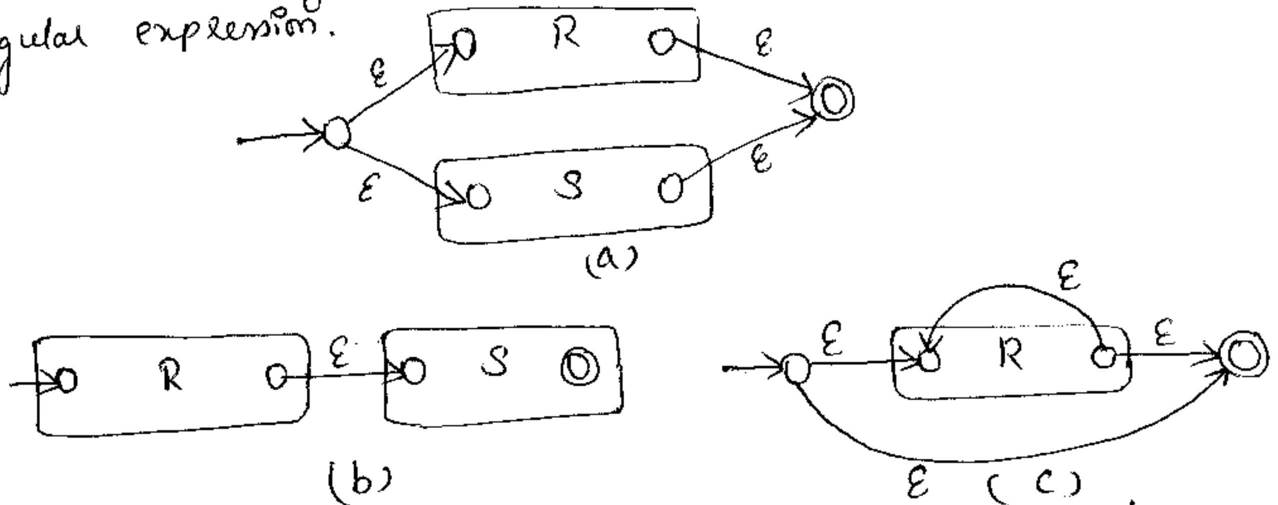
The statement of the theorem is true

We assume that the statement of the theorem is true for the immediate subexpressions of a given regular expressions

(ie), the language of these subexpressions are also the languages of ε-NFA's with a single accepting state.



The basis of the construction of an automaton from a regular expression.



The inductive step in the regular-expression-to-ε-NFA construction.

(Accepting state - following a path labeled by so string L(R) or L(S)

1). The expression is R+S for some smaller expressions R&S

The language of automaton is L(R) ∪ L(S).

2) The expression is RS for some smaller expressions R and S.

The only paths from start to accepting state go first through the automaton R, where it must follow a path labeled by a string in L(R) and then through the automaton S, where it follows a path labeled by a string in L(S).

Those labeled by strings in L(R)L(S)

3) The expression is R^* for some smaller expression R.

a) Directly from the start state to the accepting state along a path labeled ε.

Let us accept ε, which is in $L(R^*)$

b) To the start state of the automaton for R, through that automaton one or more times and then to the accepting state.

This set of paths allows us to accept strings in $L(R), L(R)L(R), L(R)L(R)L(R)$, and so on.

Thus covering all strings in $L(R^*)$ except perhaps ε,

4) The expression is (R) for some smaller expression R.

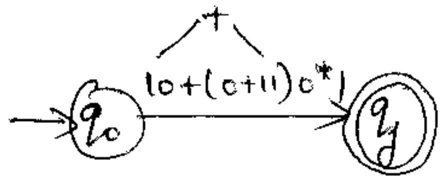
<u>Example:</u>

Design a FA from given Regular Expression

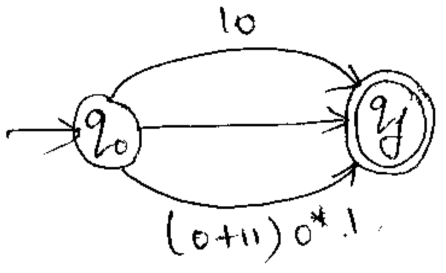
$10+(0+11)0^*1$

<u>Solution:</u>

<u>Step 1:</u>

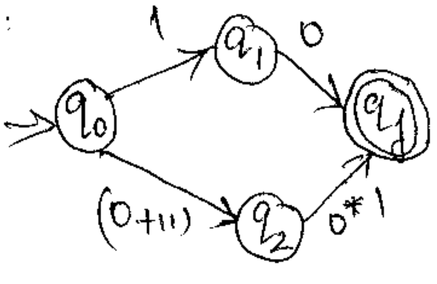


<u>Step 2:</u>

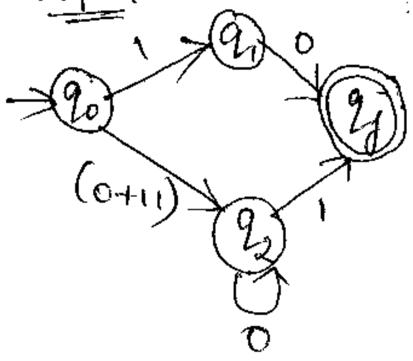


RE
↓
NFA-ε
↓
NFA
↓ ≃
DFA

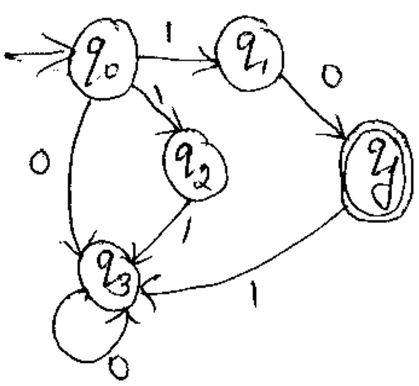
<u>Step 3:</u>



<u>Step 4:</u>



<u>Step 5:</u>



Now we have got NFA without ε. Now Convert NFA ≃ DFA
First we write transition table for NFA

State	0	1
→ q_0	q_3	$\{q_1, q_2\}$
q_1	q_f	ϕ
q_2	ϕ	q_3
q_3	q_3	q_f
q_f	ϕ	ϕ

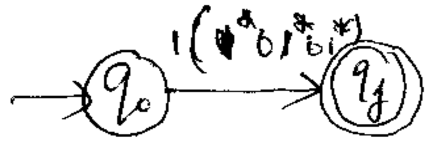
≃

State	0	1
$[q_0]$	$[q_3]$	$[q_1, q_2]$
$[q_1]$	$[q_f]$	ϕ
$[q_2]$	ϕ	$[q_3]$
$[q_3]$	$[q_3]$	$[q_f]$
$[q_1, q_2]$	$[q_f]$	$[q_f]$
$[q_f]$	ϕ	ϕ

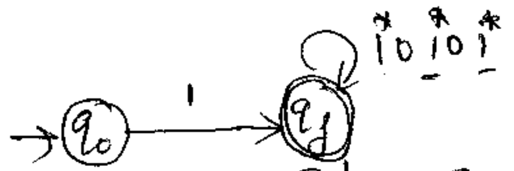
2). Design NFA from given R.E ⇒ $1(1^*01^*01^*)^*$

Solution:

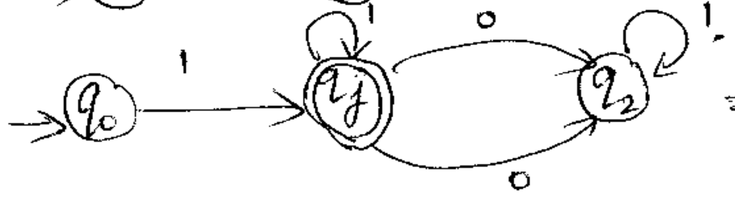
Step 1:



Step 2:



Step 3:

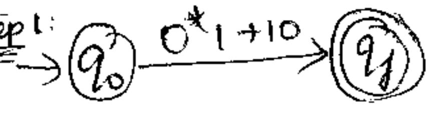


≈ NFA without Epsilon

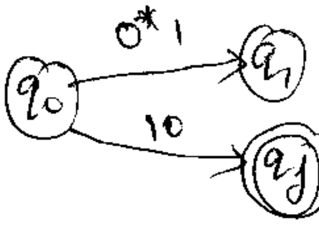
This is NFA with ε

3) Construct FA for RE 0^*1+10

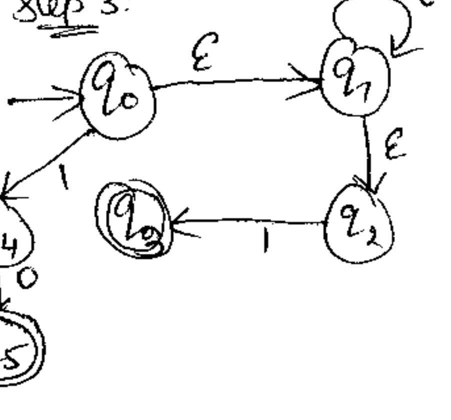
Solution: Step 1:



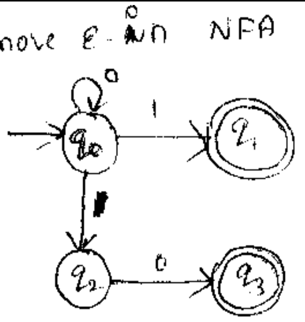
Step 2:



Step 3:

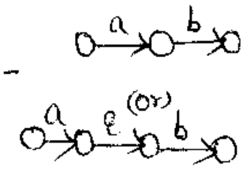


Remove ε in NFA

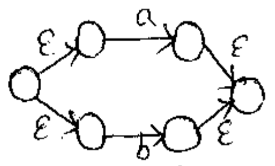


Another Example RE to FA:

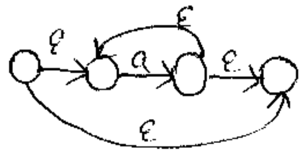
ab -



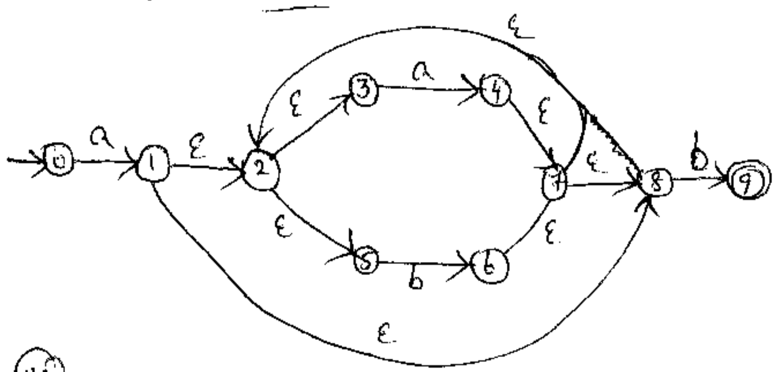
a+b -



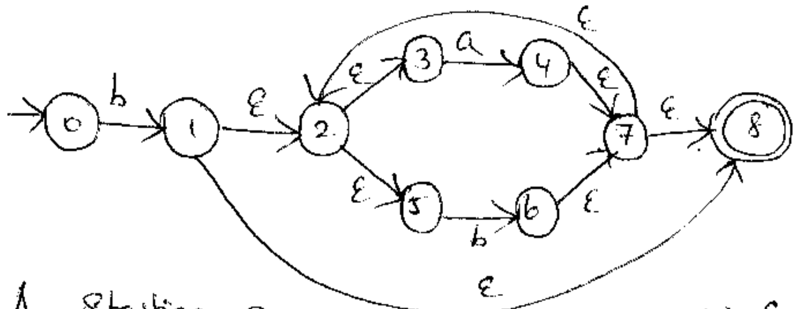
a^* -



$a(a+b)^* b \Rightarrow$ ε-NFA



✗ **Regular Expression: $b(a+b)^* \Rightarrow$ ε-NFA-DFA.**



A - Starting DFA

$\Sigma = \{a, b\}$ $\because$ ε-closure(1) = {1, 2, 8, 3, 5}
$\Downarrow$
final ε transition.

A = ε-closure (0) = {0}

$\delta(A, a)$ = ε-closure$(\delta(0, a)) = \phi = D$ $\because$ D is a dummy state.

$\delta(D, a) = \delta(D, b) = \phi = D$.

63

$\delta(A,b) = \varepsilon\text{-closure}(\delta(0,b)) = \varepsilon\text{ closure}(1) = \{1,2,8,3,5\} = \underline{B}$

$\delta(B,a) = \varepsilon\text{-closure}(\delta(1,a) \cup \delta(2,a) \cup \delta(8,a) \cup \delta(3,a) \cup \delta(5,a))$

$= \varepsilon\text{-closure}(\phi \cup \phi \cup \phi \cup \{4\} \cup \phi)$

$= \varepsilon\text{-closure}(4) = \{4,7,8,2,3,5\} = C.$

$\therefore \delta(A,a) = D,$
$\delta(A,b) = B,$
$\delta(D,a) = \delta(D,b) = \phi = D$

$\delta(B,b) = \varepsilon\text{-closure}(\delta(1,2,8,3,5),b)$

$= \varepsilon\text{-closure}(6) = \{6,7,8,2,3,5\} = E$

$\delta(C,a) = \varepsilon\text{-closure}(\delta(4,7,8,2,3,5),a)$

$= \varepsilon\text{-closure}(4) = C$

$\delta(C,b) = \varepsilon\text{-closure}(\delta(4,7,8,2,3,5),b)$

$= \varepsilon\text{-closure}(6) = E$

$\delta(E,a) = \varepsilon\text{-closure}(\delta(6,7,8,2,3,5),a)$

$= \varepsilon\text{-closure}(4) = C$

$\delta(E,b) = \varepsilon\text{-closure}(\delta(6,7,8,2,3,5),b)$

$= \varepsilon\text{-closure}(6) = E$

Table:

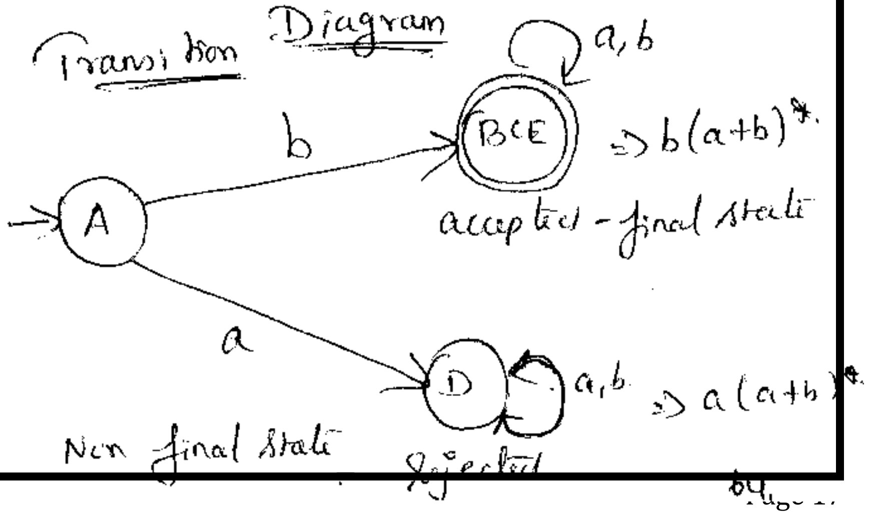
	a	b
→ A	D	B
B^*	C	E
C^*	C	E
D	D	D
E^*	C	E

$\therefore B^* C^* E^*$ = final State

DFA.

	a	b
→ A	D	BCE
D	D	D
BCE	BCE	BCE

Transition Diagram



APPLICATIONS OF REGULAR LANGUAGES:-

A regular expression that gives a "picture" of the pattern we want to recognize is the medium of choice for applications that search for patterns in text.

Consider two important classes of regular-expression based applications: Lexical analyzers
Text Search

Regular Expressions in UNIX:-

The UNIX notation for extended. regular expressions.

Regular Expressions are used in UNIX are extended version of RE's,

Et allowing non-regular languages to be recognized

The UNIX regular expressions allow us to write character classes to represent large sets of characters. .

The rules for character classes are

The symbol . (dot) stands for "any character".

The sequence $[a_1 a_2 \ldots a_k]$ stands for the regular expressions $a_1 + a_2 + \ldots + a_k$.

$[a_1 - a_k] \rightarrow [a_1\, a_2 \ldots a_k]$

eg: $[0-9] \rightarrow [0\,1 \ldots 9] \rightarrow 0+1+2+\ldots+9$

$[A-Z] \rightarrow A+B+\ldots+Z$

$[A-Za-z0-9] \rightarrow$ set of all letters and digits

$[+-.0-9] \rightarrow$ characters for forming signed digits.

Notations:

[:digit:] is the set of ten digits, the same as $[0-9]^3$

[:alpha:] stands for any alphabetic character, as does [A-Za-z]

[:alnum:] stands for the digits and letters (alphabetic and numeric characters) as does [A-Za-z0-9]

There are several operators that are used in UNIX Regular expressions.

①. The Operator | is used in place of + to denote union

②. The Operator ? means "zero or one of".

Thus, R? in UNIX is the same as ε+R

③. The Operator + means "one or ~~to~~ more of.

Thus R+ in UNIX is shorthand for RR^* in our notation.

④. The Operator {n} means "n copies of"

Thus R{5} in UNIX is shorthand for RRRRR.

⑤. Still used in UNIX.

<u>Lexical Analysis:</u>

One of the oldest applications of regular expressions was in specifying the component of a compiler called a "lexical analyzer"

This component scans the source program and recognizes all tokens, those substrings of consecutive characters that belong together logically.

The UNIX command lex and its GNU version flex, accept as input a list of regular expressions, in the UNIX style.

The first line handles the keyword else and the action is to return a symbolic constant (ELSE in this example) to the parser for further processing.

else {return (ELSE); }

The second line contains a regular expression describing identifiers: a letter followed by zero or more letters and/or digits.

[A-Za-z][A-Za-z0-9]* { Code to enter the found identifier in the symbol table;
return (ID);
}

```
>=          {return(GE);}
=           {return(EQ);}
....
```

A Sample of lex input

The action is first to enter that identifier in the symbol table, if not already there;

lex isolates the token found in a buffer, so this piece of code knows exactly what identifier was found.

Finally the lexical analyzer returns the symbolic constant ID,

Finding Patterns in Text :-

The notion that automata could be used to search efficiently for a set of words in a large repository such as the Web.

The regular-expression notation ~~to~~ is valuable for describing searches for interesting patterns.

The general problem for which regular-expression technology has been found useful is the description of a vaguely defined class of patterns in text.

The ending for our regular expression something like

Street | St\. | Avenue | Ave\. | Road | Rd\.

We have used UNIX-style notations, with the vertical bar, rather than +, as the union operator.

Note that, the dots are escaped with a preceding backslash, since dot has the special meaning of "any character" in UNIX expressions.

After discovering that we were missing addresses of this form, we could revise our description of street names to be

[A-Z][a-z]*([A-Z][a-z]*)*

The expression above starts with a group consisting of a capital and zero or more lower-case letters.

Zero or more groups consisting of a blank, another capital letter and zero or more lower-case letters.

The blank is an ordinary character in UNIX expressions

Thus, the expression we use for numbers has an optional capital letter following : [0-9]+[A-Z]?

Note that, we use the UNIX + operator for "one or more" digits and the ? operator for "zero or ~~one~~ one" capital letter.

'[0-9]+[A-Z]? [A-Z][a-~~z~~z]*([A-Z][a-z]*)*
(Street|St\.|Avenue|Ave\.|Road|Rd\.)'

ALGEBRAIC LAWS FOR REGULAR EXPRESSIONS:-

In all cases, the basic issue was that the two expressions were equivalent, in the sense that they defined the same languages

Two expressions with variables are equivalent if whatever languages we substitute for the variables, the results of the two expressions are the same language

<u>Associativity</u> <u>and</u> <u>commutativity</u>:-

Commutativity is the property of an operator that says we can switch the order of its operands and get the same result. eg: $x+y = y+x \Rightarrow$ Commutative law.

Associativity is the property of an operator that allows us to regroup the operands when the operator is applied twice. The associative law of multiplication is $(x \times y) \times z = x \times (y \times z)$.

$\Rightarrow L+M = M+L \Rightarrow$ The commutative law for union, we may take the union of two languages

$\Rightarrow (L+M)+N = L+(M+N) \Rightarrow$ The associative law for union, we may take the union of three languages.

$\Rightarrow (LM)N = L(MN) \Rightarrow$ The associative law for concatenation, we can concatinate three languages.

<u>Identities and Annihilators</u>:-

An identity for an operator is a value such that when the operator is applied to the identity and some other value, the result is the other value.

Instance

0 is the identity for addition, $0+x = x+0 = x$.

1 is the identity for multiplication, $1\times x = x\times 1 = x$.

An Annihilator for an operator is a value such that when the operator is applied to the annihilator ~~for multiplication~~ and some other value, the result is the annihilator.

Instance

0 is an annihilator for multiplication, $0\times x = x\times 0 = 0$.

There is no annihilator for addition.

There are three laws for regular expressions involving these concepts.

$\Rightarrow \phi + L = L + \phi = L$. Asserts that ϕ is the identity for union

$\Rightarrow \epsilon L = L\epsilon = L$. Asserts that ϵ is the identity for concatenation.

$\Rightarrow \phi L = L\phi = \phi$. Asserts that ϕ is the annihilator for concatenation.

<u>Distributive Laws</u>:-

A distributive law involves two operators, and asserts that one operator can be pushed down to be applied to each argument of the other operator individually.

Arithmetic is the distributive law of multiplication over addition, that is $x \times (y+z) = x \times y + x \times z$.

⇒ $L(M+N) = LM + LN$. is the left distributive law of concatenation over union.

⇒ $(M+N)L = ML + NL$. is the right distributive law of concatenation over union.

THEOREM: If L, M and N are any languages, then

$$L(M \cup N) = LM \cup LN.$$

Proof: Similar to another proof about a distributive law

⇒ To show that a string w is in $L(M \cup N)$ if and only if it is in $LM \cup LN$.

⇒ If w is in $L(M \cup N)$, then $w = xy$. (only if)
where x is in L and y is in either M or N.
If y is in M, then xy is in LM, and therefore in $LM \cup LN$.
If y is in N, then xy is in LN, and therefore in $LM \cup LN$

⇒ Suppose w is in $LM \cup LN$.
Then w is in either LM or in LN.

Suppose that w is in LM.
Then $w = xy$, where x is in L and y is in M.
As y is in M, it is also in $M \cup N$.
Thus xy is in $L(M \cup N)$. If w is not in LM, then it is surely in LN.

The Idempotent Law:-

An Operator is said to be idempotent if the result of applying it to two of the same value arguments is that value.

The common arithmetic operators are not idempotent: $x+x \neq x$ in general and $x * x \neq x$ in general.

$L+L=L$. The idempotence law for union, states that if we take the union of two identical expressions, we can replace them by one copy of the expression.

Laws Involving closures:-

There are a number of laws involving the closure operators and its UNIX-style variants + and ?.

$\Rightarrow (L^*)^* = L^*$. that closing an expression that is already closed does not change the language.

The language of $(L^*)^*$ is all strings created by concatenating strings in the language of L^*.

Those strings are themselves composed of strings from L. Thus the string in $(L^*)^*$ is also a concatenation of strings from L and is therefore in the language of L^*.

$\Rightarrow \phi^* = \epsilon$. The closure of ϕ contains only the string ϵ,

$\Rightarrow \epsilon^* = \epsilon$. Easy to check that only string that can be formed by concatenating any number of copies of the empty string is the empty string itself.

$\Rightarrow L^+ = LL^* = L^*L$.

Recall that L^+ is defined to the $L+LL+LLL+\ldots$. Thus Also $L^* = \epsilon+L+LL+LLL+\ldots$

$LL^* = L\epsilon+LL+LLL+LLLL+\ldots$

$L\epsilon = L$, the infinite expansions for LL^* and for L^+ are the same. That proves $L^+ = LL^*$. Proof that $L^+ = L^*L$.

$\Rightarrow L^* = L^+ + \epsilon$. The expansion of L^+ includes every term in the expansion of L^* except ϵ.

Note that if the language L contains the string ϵ, then the additional "$+\epsilon$ term is not needed;

(ie) $L^+ = L^*$

$\Rightarrow L? = \epsilon + L$, This rule is really the definition of the ? operator.

<u>Discovering Laws for Regular Expressions:-</u>

Each of the laws was proved, formally or informally. There is an infinite variety of laws about regular expressions that might be proposed.

Let us consider a proposed law, such as

$$(L+M)^* = (L^* M^*)^*.$$

<u>The Test for a Regular-Expression Algebraic Law:-</u>

We can state and prove the test for whether or not a law of regular expressions is true.

The test for whether $E = F$ is true, where E and F are two regular expressions with the same set of variables

1). Convert E and F to concrete regular expressions C and D respectively, by replacing each variable by a concrete symbol.

2). Test whether $L(C) = L(D)$. If so, then $E=F$ is a true law. and if not. then the "law" is false.

The test for whether two regular expressions denote the same language.

72

Pumping Lemma for Regular Languages:

The properties of regular languages, our first tool for this exploration is a way to prove that certain languages are not regular. This theorem called the "pumping lemma".

The class of languages known as the regular languages has at least four different descriptions.

They are the languages accepted by DFA's, by NFA and by ε-NFA; they are also the languages defined by regular expressions.

Not every language is a regular language, we shall introduce a powerful technique, known as the "pumping lemma".

Statement of the Pumping lemma:

Let us consider the language $L_{01} = \{0^n1^n \mid n \geq 1\}$. This language contains all strings 01, 0011, 000111. and so on, that consist of one or more 0's followed by an equal number of 1's.

THEOREM: Let L be a regular language. Then there exists a constant n (which depends on L) such that for every string in w in L such that $|w| \geq n$, we can break w into three strings, $w = xyz$, such that

1). $y \neq \varepsilon$

2). $|xy| \leq n$

3). For all $k \geq 0$, the string xy^kz is also in L

Proof: Suppose L is regular. Then $L = L(A)$ for some DFA A. Suppose A has n states.

Now, consider any string w of length n or more, say $w = a_1a_2 \ldots a_m$, where $m \geq n$ and each a_i is an input symbol.

For $i = 0, 1, \ldots, n$ define state p_i to be $\hat{\delta}(q_0, a_1 a_2 \ldots a_n)$, where δ is the transition function of A, and q_0 is the start state of A.

73

(ie) p_i is the state A is in after reading the first i symbol of w.

Note that $p_0 = q_0$.

⇒ Find two different integers i and j, with $0 \le i < j \le n$, such that $p_i = p_j$.

We can break $w = xyz$.

1). $x = a_1 a_2 \ldots a_i$

2) $y = a_i + a_{i+1} + a_{i+2} + \ldots a_j$

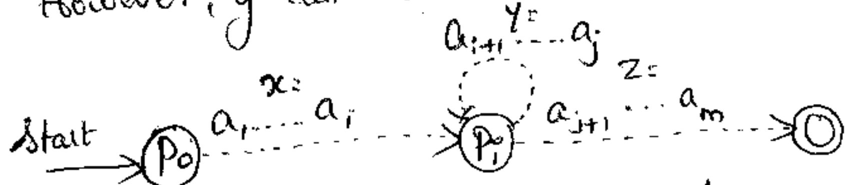
3). $z = a_j +_1 a_j + 2 \ldots a_m$

(ie) x takes ~~upto~~ us to p_i once, y takes us from p_i back to p_i ($\because p_i$ also p_j) and z is the balance of w.

Note that x may be empty, in the case that $i=0$.

Also, z may be empty, if $j = n = m$.

However, y can not be empty, since i is strictly less than j.



If $k > 0$, then A goes from q_0 to p_i on input x, circles from p_i to p_i k times on input y^k, and then goes to the accepting state on input z.

Thus, for any $k \ge 0$, xy^kz is also accepted by A,

(ie) xy^kz is in L.

Applications of the Pumping Lemma:-

Pumping Lemma is to be applied to show that certain languages are not regular.

It should never be used to show a language is regular.

⇒ If L is register, it satisfies Pumping lemma

⇒ If L does not satisfy Pumping lemma, it is non-Regular.

To prove that a languages is not regular using Pumping lemma follow the steps.

- ⇒ At first, we have to assume that L is Regular
- ⇒ So the pumping lemma should hold for L.
- ⇒ It has to have to a pumping length (say p)
- ⇒ All strings longer than P can be pumped $|w| \geq p$.
- → Now find a string 'w' in L such that $|w| \geq p$.
- ⇒ Divide w into xyz.
- ⇒ Show that $xy^iz \notin L$ for some i

$$xy^nz \notin L.$$

- ⇒ Then consider all ways that w can be divided into xyz.
- ⇒ Show that none of these can satisfy all the pumping conditions at the same time.
- ⇒ W cannot be pumped : CONTRADICTION.

Example:

Prove that $L = \{a^ib^i \mid i \geq 0\}$ is not Regular

Solution:

→ At first, we assume that L is regular and n is the number of states.

→ Let $w = a^nb^n$. Thus $|w| = 2n \geq n$.

→ By Pumping lemma, let $w = xyz$ where $|xy| \leq n$

Let $x = a^p$
$y = a^q$
$z = a^rb^n$

where
$p+q+r = n$
$p \neq 0, q \neq 0, r \neq 0$
Thus $|y| \neq 0$

→ Let $k = 2$

Then $xy^2z = a^pa^{2q}a^rb^n$

→ Number of a's $= (p+2q+r)$
$= (p+q+r) + q = n+q$

$= n+q \qquad xy^2z = a$

$\Rightarrow$ Hence $xy^2z = a^{n+q} b^n$ Since $q \neq 0$
xy^2z is not of the form $a^n b^n$.

$\Rightarrow$ Thus xy^2z is not in L.
Hence L is not regular.

Closure Properties of Regular Languages:-

If certain languages are regular, and a language L is formed from them by certain operations (eg., L is the union of two regular languages), then L is also regular.

These theorem are often called closure properties of the regular languages.

The principal closure properties for regular languages.

* The union of two regular languages is regular
* The intersection of two regular languages is regular.
* The complement of a regular language is regular
* The difference of two regular language is regular
* The reversal of a regular language is regular
* The closure (star) of a regular language is regular
* The concatenation of regular language is regular.
* A homomorphism (substitution of strings for symbols) of a regular language is regular.
* The inverse homomorphism of a regular language is regular.

Closure of Regular Languages Under Boolean Operations:-

Closure properties are the three boolean operations:

76

Union, Intersection and Complementation

1) Let L and M be languages over alphabet Σ. Then $L \cup M$ is the language that contains all strings that are in either or both of L and M.

2) Let L and M be languages over alphabet Σ. Then $L \cap M$ is the language that contains all strings that are both L and M.

3). Let L be a language over alphabet Σ. Then $\bar{L}$, the complement of L, is the set of strings in Σ^* that are not in L.

$\therefore$ The Regular languages are closed under all three of the boolean operations.

Closure Under Union:-

Theorem:- If L and M are regular languages, then so is $L \cup M$.

Proof: It is simple. Since L and M are regular, they have regular expressions; $L = L(R)$ and $M = L(S)$. Then

$$L \cup M = L(R) \cup L(S)$$
$$= L(R) + L(S)$$
$$= L(R+S)$$

$$\boxed{L \cup M = L(R+S)}$$

Closure Under Complementation:-

The theorem for union was made very easy by the use of the regular expression representation for the languages. However let us consider complementation.

Find a regular expression for its complement

1). Convert the regular expression to an ϵ-NFA.

2) Convert that ϵ-NFA to a DFA by the subset construction

3) Complement the accepting states of that DFA.

4). Turn the complement DFA back into regular expression using the ~~[illegible]~~ construction.

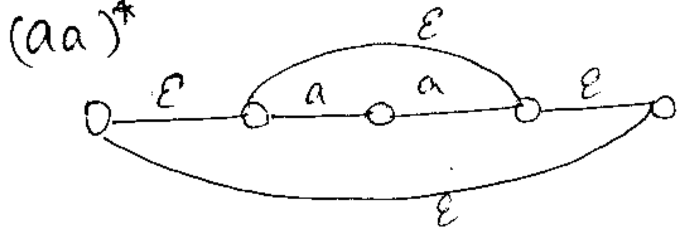
If L is a Regular language over a set of alphabet, Σ, then $\bar{L}$ is Σ also a Regular language.

Eg:

$L = \{a, a^3, a^5, \ldots\}$

$\bar{L} = \{\epsilon, a^2, a^4, a^6, \ldots\}$

RE = $(aa)^*$



Closure under difference:

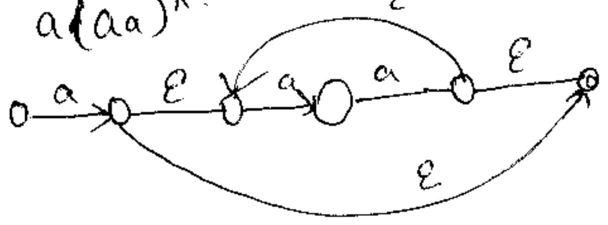
If L and M are Regular language, then $L_1 - L_2$ is also a Regular language.

Eg:

$L = \{a, a^2, a^3, a^4, a^5 \ldots\}$ $M = \{a^2, a^4, a^6 \ldots\}$

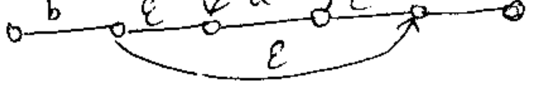
$L - M = \{a, a^3, a^5, a^7, \ldots\}$

RE = $a(aa)^*$.



Reversal:

If L is Regular language, then L^R is also a Regular language.



Eg: $L = \{001, 10, 111, 01\}$ or $L = [ab, aab, aaab, \ldots]$

$L^R = \{100, 01, 111, 10\}$ RE = $a(a)^* b$

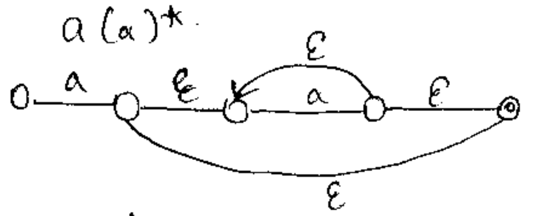
$L^R = [ba, baa, baaa, \ldots]$

RE = $ba(a)^*$ or $b(a)^* a$.

Eg: union $L_1 = \{a, a^3, a^5, \ldots\}$ $M_2 = \{a^2, a^4, a^6, \ldots\}$

$L \cup M = \{a, a^2, a^3, a^4, a^5, a^6, \ldots\}$

RE : $a(a)^*$



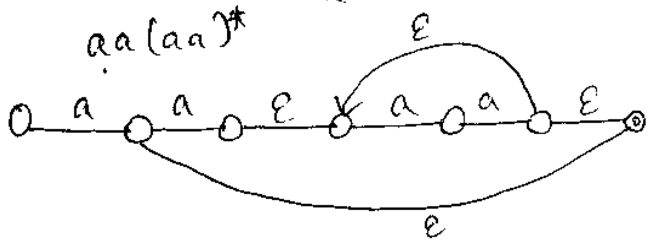
Closure under intersection:-

If L and M are regular languages, then $L \cap M$ is also regular.

Eg: $L = \{a, a^2, a^3\}$ $M = \{a^2, a^4, a^6, \ldots\}$

$L \cap M = \{a^2, a^4, a^6, \ldots\}$

RE : $aa(aa)^*$



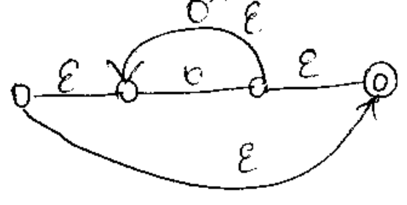
Closure

If L is regular language, then L^* is also a regular language.

eg: $L = \{0\}$

$L^* = \{\varepsilon, 0, 00, 000 \ldots\}$

RE : 0^*.



79

Under closure Concatenation :-

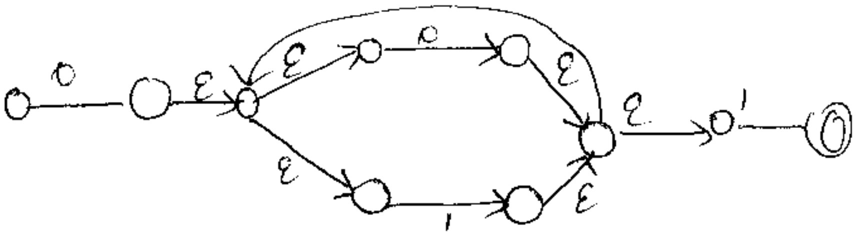
If L and M are the regular languages, then L.M is also a regular language.

Eg: $L_1 = \{0, 00, 01 \ldots\}$ $M = \{1, 01, 111, \ldots\}$

RE = $0(0+1)^*$ RE = $(0+1)^* 1$.

$L.M = \{0, 001, 011, 0011, 0011 \ldots\}$

RE = $0(0+1)^* 1$



The homomorphism :

If L is a regular language and h is a ~~complementation~~ homomorphism, the h(L) is also regular.

$L = \{\varepsilon, 0, 00, 000, \ldots\}$ RE = 0^*. $[h(0) = a]$

$0 \to a$.

~~[illegible] homomorphism~~ RE: a^*.

$h(L) = \{\varepsilon, a, aa, aaa \ldots\}$

A string homomorphism is a function on strings that works by substituting a particular string for each symbol.

Eg: $h(0) = abb$ } homomorphism which replaces each 0 by abb and each 1 by ba.
$h(1) = ba$.

$\therefore$ Thus $h(1011) = baabbbaba$.

Where $E = F + G$ $L(E) = L(F) \cup L(G)$

$h(E) = h(F+G)$.
$= h(F) + h(G)$

$L(h(E)) = L(h(F) + h(G)) = L(h(F)) \cup L(h(G))$

by the definition of what '+' means in regular expressions.

Finally, $h(L(E)) = h(L(F)) \cup L(G)) = h(L(F)) \cup L(h(G))$

Inverse Homomorphism:-

Substitutions of the symbols for the strings of regular language is regular.

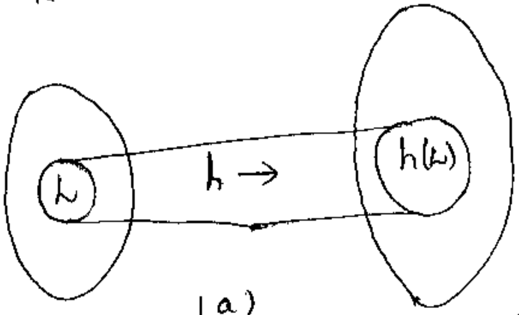
If L is a regular language and L^{-1} is a inverse homomorphism, then $h(L)$ is also RL.

$L = \{\varepsilon, a, aa, \ldots\}$ ∵ $a \to 0$

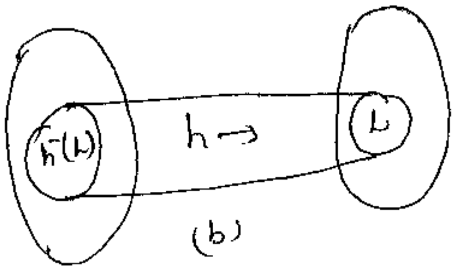
RE: a^*

$\{\varepsilon, 0, 00, \ldots\}$

$RE = 0^*$



(a)



(b)

A homomorphism applied in the forward and inverse direction

(If) Suppose w is n repetitions of ba for some $n \geq 0$. Note that $h(ba) = 1001$, so $h(w)$ is n repetitions of 1001. and show that

(only-if), we must assume that $h(w)$ is in L

w is of the form baba...ba.

The contrapositive of the statement

~~i). If w begins with a, then h(w) ends in 10 and again there is an isolated 0 in h(w)~~

1). If w begin with a, then $h(w)$ begins with 01. It therefore has an isolated 0, and is not in L.

2). If w ends in b, then $h(w)$ ends in 10 and again there is an isolated 0 in $h(w)$.

3). If w has two consecutive a's, then h(w) has a substring 0101. Here too, there is an isolated 0 in w

DECISION PROPERTIES OF REGULAR LANGUAGES:-

The answer different types of questions like whether the given language is empty, finite or infinite various algorithms are required.

Based on representations we can answer such questions one of the ways is, assume that regular sets are represented by finite automata.

The regular sets are represented by regular expression which in turn represented by FA.

~~1) Emptiness~~

Converting Among Representations:- ⇒ - Assume · # symbols = constant
· # states = n.

W.K.T, we can convert any of the four representations for regular languages to any of the other three representations.

The equivalence of four different notations for regular languages gave paths from any representation to any of the others.

Converting NFA's to DFA's:-

When we start with either an NFA or and ε-NFA and convert it to a DFA, the time can be exponential in the number of states of the NFA.

The running time of NFA-to-DFA conversion, including the case where the NFA has ε-transitions, is $O(n^3 2^n)$.

∴ The number of states created is much less than 2^n, often only n states.

The bound on the running time as $O(n^3 s)$, where s is the number of states the DFA.

DFA-to-NFA Conversion:-

It is simple and takes O(n) time on an n-state DFA.

Need to modify the transition table for the DFA by putting set brackets around states and if it output is an ε-NFA, adding a column for ε.

The number of input symbols (i.e. the width of the transition table) as a constant, copying and processing the table takes O(n) time.

Automaton-to-Regular-Expression Conversion:-

At each of n rounds (where n is the number of states of the DFA), we can quadruple the size of the regular expressions constructed, since each is built from four expressions of the previous round.

The n^3 expressions can take time $O(n^3 4^n)$.

Convert an NFA to a DFA and then convert the DFA to a Regular expression, it could take time $O(n^3 4^{n^3 2^n})$, which is doubly exponential.

Regular-Expression-to-Automaton Conversion:-

Conversion of a Regular expression to an ε-NFA takes linear time in the size of the RE.

We need to parse the expression efficiently, using a technique that takes only O(n) time on a Regular expression of length n^3.

An expression tree with one node for each symbol of the Regular expressions.

Testing Emptiness of Regular Languages:-

⇒ Testing if a RL generated by an automaton is empty.

⇒ Equivalent to testing if there exists no path from the ~~states~~ start states to accept an state.

Note that the reachability calculation takes no more time that $O(n^2)$ if the automaton has n states.

ϕ denotes the empty language ; ϵ and a for any input symbol a do not.

There are four cases to consider, corresponding to the ways that R could be constructed.

$R = R_1 + R_2$. Then $L(R)$ is empty if and only if both $L(R_1)$ and $L(R_2)$ are empty.

$R = R_1 R_2$. Then $L(R)$ is empty if and only if either $L(R_1)$ or $L(R_2)$ is empty.

$R = R_1^*$. Then $L(R)$ is not empty ; it always includes at least ϵ.

$R = (R_1)$. Then $L(R)$ is empty if and only if $L(R_1)$ is empty. Since they are same language.

<u>Testing Membership in a Regular Language</u>:-

Given a string w and a regular language L, is w in L. While w is represented explicitly, L is represented by an automaton or regular expression.

If $|w| = n$, and the DFA is represented by a suitable data structure, such as a two-dimensional array that is the transition table, then each transition requires constant time and entire test takes $O(n)$ times.

If the representation is an NFA ~~[illegible]~~ or ϵ-NFA, it is simpler and more efficient to simulate the NFA directly.

If w is of length n and the NFA has s states, then the running time of this algorithm is $O(ns^2)$.

The union at most s set of at most s states each, which requires $O(s^2)$ time.

If the representation of L is a regular expression of size s, we can convert to an ϵ-NFA with at most 2s states, in $O(s)$ time, then perform the simulation taking $O(ns^2)$ time on an input w of length n.

EQUIVALENCE AND MINIMIZATION OF AUTOMATA

To test whether two descriptions for regular languages are equivalent, in the sense that they define the same language.

An important consequence of this test is that there is a way to minimize a DFA.

Testing Equivalence of States:-

When two distinct states q_i and q_j can be replaced by a single state that behaves like both q_i and q_j.

We say that states q_i and q_j are equivalent.

For all input strings w, $\hat{\delta}(q_i, w)$ is an accepting state if and only if $\hat{\delta}(q_j, w)$ is an accepting state.

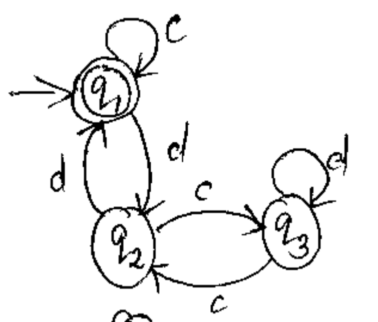
Steps to Identify Equivalence:-

① For any pair of states (q_i, q_j) the transition for input $a \in \Sigma$ is defined by (q_a, q_b) where $\delta\{q_i, a\} = q_a$ and $\delta(q_j, b) = q_b$.

② The two automata are not equivalent if for a pair $\{q_a, q_j\}$ one is intermediate state and other is final state.

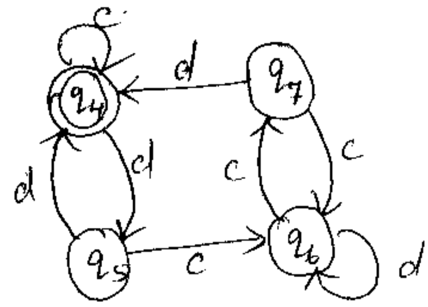
③ If initial state is final state of one automata then, the second automata is also an initial state must be final state for them to be equivalent.

Ex:



(A)

Equivalent



(B)

Pair of states both are - final state or Intermediate State.

∴ fs - Final State
IS - Intermediate State

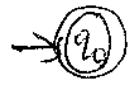
States	c	d
$\{q_1, q_4\}$	$\{q_1, q_4\}$ fs fs	$\{q_2, q_5\}$ IS IS
$\{q_2, q_5\}$	(q_3, q_6) IS IS	(q_1, q_4) fs fs
(q_3, q_6)	$\boxed{(q_2, q_7)}$ IS IS	(q_3, q_6) IS IS
(q_2, q_7)	(q_3, q_6) IS IS	(q_1, q_4) fs fs

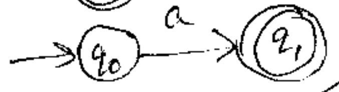
A & B are equivalence.

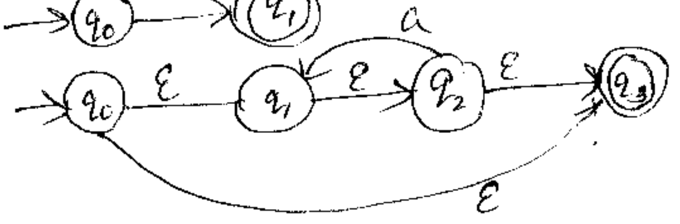
Testing Equivalence of Regular Languages:-

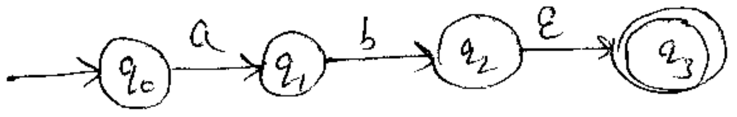
To list if two regular langes are the same. Suppose languages L and M are each represented in some way. One by a regular expression and one by an NFA. Convert each representation to a DFA.

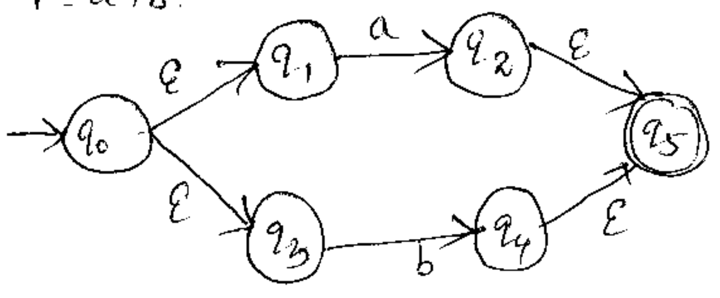
Ex: $(a+b)^*$ (or) $(a/b)^*$.

$\Rightarrow$ $r = \varepsilon$


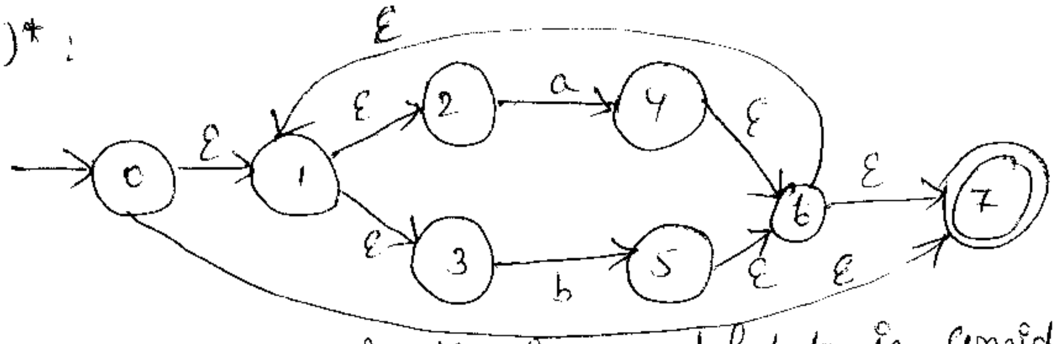
$r = a$


$r = a^*$


$r = a \cdot b$


$r = a/b$ (or) $r = a+b$.


$(a+b)^*$:



∴ The size of the input alphabet is considered a Constant, each pair of states is put on O(1) lists.

There are $O(n^2)$ Pairs, the total work is $O(n^2)$

<u>Minimization of DFA's:</u>

The test for equivalence of states is that we can "minimize" DFA's.

(ie) For each DFA we can find an equivalent DFA that has as few states as any DFA accepting the Same language.

1). First, eliminate any State that Cannot be reached from the Start State.

2) Then, Partition the remaining states into blocks, So that alls States in the Same block are equivalent and no pair of States from different blocks, are equivalent.

∴ The minimization of DFA means Reducing the number of states from a given FA.

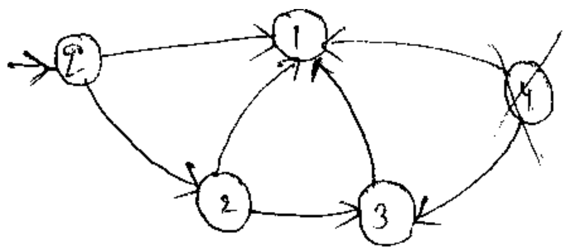
<u>Step 1</u>: Remove all the states that are unreachable from initial State via any <u>set of transition of DFA</u>

<u>Step 2</u>: Draw the transition table for all pairs of states.

<u>Step 3</u>: Now split the transition table into two tables T_1 & T_2.

T_1 Contains <u>all final State</u>

T_2 Contains <u>non-final State</u>



Step 4: Find similar rows from T, such that

$\delta(q, a) = p$

$\delta(r, a) = p$

δ	a	b
q	p	p
r	p	p

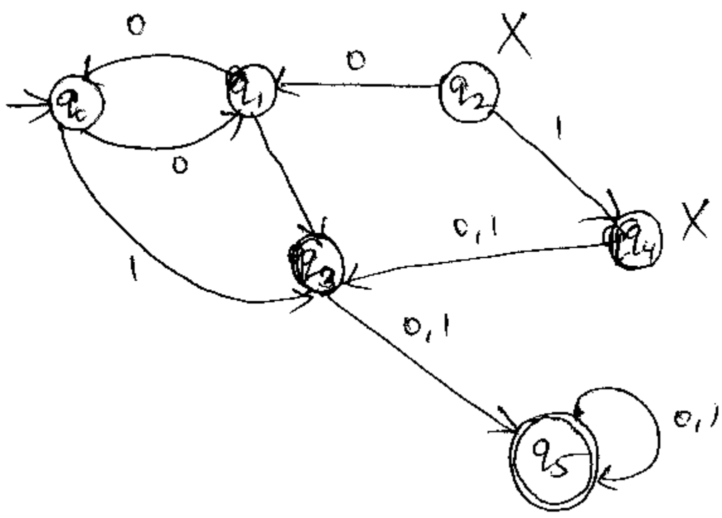
Find the two state which have same value of a & b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in T_1.

Step 6: Repeat step 3 and step 4 for the table T_2 also

Step 7: Now combine the reduced T_1 & T_2 tables is the final transition table of minimized DFA.

Example:



Step 1: Remove q_2 & q_4 in finite automata

unreachable state

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

States	0	1	
$\rightarrow q_0$	q_1	q_3	T_1
q_1	q_0	q_3	
$* q_3$	q_5	q_5	T_2
$* q_5$	q_5	q_5	

Step 3: (i) Tables which starts from non-final states

States	0	1
$\rightarrow q_0$	q_1	q_3
q_1	q_0	q_3

(ii) Tables which starts from final state

$*\ q_3$	q_5	q_5
$*\ q_5$	q_5	q_5

skip

Step 4: (i) Set 1 has no similar rows, they will be same

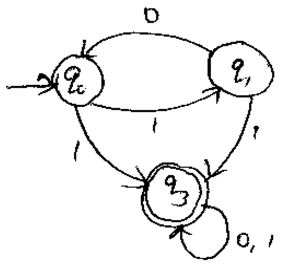
(ii) Set 2 has similar rows, so skip q_5 & then replace q_5 by q_3.

Step 5: New table

	0	1
q_3	q_3	q_3

Step 6: Combine set 1 and set 2.

State	0	1
$\rightarrow q_0$	q_0	q_3
q_1	q_0	q_3
$*\ q_3$	q_3	q_3



* —— * —— * —— * —— *