Automata and Computability

ATC

Dr. Girijamma.
Professor, CSE
RNSIT

130+20=150

Module-1

Theory of computation is useful in two ways:

* It provides a set of abstract structures that are useful for solving certain classes of problems.

These abstract structures can be implemented on whatever hardware/software platform is available.

* It defines provable limits to what can be computed, regardless of processor speed & memory size.

Applications of Theory of Computation:

1) Languages enable both machine/machine and person/machine communication.

eg: Network communication protocols, HTML etc.

2) Both the design and the implementation of modern programming languages rely heavily on the theory of CFL. CFG's are used to document the languages syntax and they form the basis for the parsing techniques that all compilers use.

3) People use natural languages to communicate with each other and we can build programs to manage our words, check our grammar, search www and translate

4) Systems such as parity checkers, vending machines, communication protocols, and building security devices can be described as finite state machines.

5) Many interactive video games are finite state machines.

6) DNA is the language of life. DNA molecules as well as the proteins that they describe are strings made up of symbols. So computational biologists use many of the tools based on FSM and CFG.

7) Security - The undecidability of the correctness of a security model.

8) AI programs solve problems in task domains ranging from medical diagnosis to factory scheduling. The role of undecidability and complexity results in AI.

9) The design of a description language for the semantic web.

10) Graph algorithms in network analysis.

11) Heuristic search algorithms find paths in computer games.

Dr. GIRIJAMMA .H.A
B.E, M.Tech, Ph.D
Professor,
Computer Science & Engg. Department
RNS Institute of Technology
Channasandra, R.R. Nagar,
BANGALORE - 560 098.

Languages and Strings :

Alphabet : An alphabet is a finite nonempty set of symbols represented by Σ.

$\Sigma=\{0,1\}$ - binary alphabet - ε, 0, 00100 are str

$\Sigma=\{a,b,c,..z\}$ - English alphabets, ε, aabcd, aaa are strings.

Strings :

A string is a finite sequence of symbols drawn from some alphabet Σ.

Empty string:

Empty string is a string with no symbols and is denoted by ε.

The set of all possible strings over te alphabet Σ is written as Σ^*.

functions on strings :

* The length of a string s, is the no of symbols in s.

For eg: $|\epsilon|=0$

$|10011|=5$

* For any ~~strings~~ symbol c and string s, we define $\#_c(s)$ to be the no of times that c occurs in s.

eg: $\#_a(abbaaa)=4$

Concatenation :

Concatenation of two strings s and t written s||t or st is the string formed by appending t to s.

eg: $x = good$ and $y = bye$.

$xy = goodbye$.

So $|xy| = |x| + |y|$

empty string ϵ, is the identity for concatenation of strings. So $\forall x \, (x\epsilon = \epsilon x = x)$.

String Replication :

For each string w and natural number i, the string w^i as :

$$w^0 = \epsilon$$

$$w^{i+1} = w^i w$$

eg:

$a^3 = aaa$

$(bye)^2 = byebye$

$a^0 b^3 = bbb$

String Reversed :

w^R

If $|w| = 0$ then $w^R = w = \epsilon$

If $|w| > 1$ then $\exists a \in \Sigma \, (\exists u \in \Sigma^* \, (w = ua))$

$$w^R = a u^R$$

Theorem : Concatenation and Reverse of strings.

If w and x are strings then $(wx)^R = x^R w^R$.

Proof : The proof is by induction on $|x|$:

Base case : $|x| = 0$ then $x = \epsilon$,

$(wx)^R = (w\epsilon)^R = (w)^R = \epsilon w^R = \epsilon^R w^R = x^R w^R$

Prove $\forall\, n \geq 0$

$|x| = n \rightarrow (wx)^R = x^R w^R$

$|x| = n+1 \rightarrow (wx)^R = x^R w^R$.

Consider any string x, where $|x| = n+1$, then $x = ua$ for some character a and $|u| = n$. so

$(wx)^R = (w(\overset{ua}{\cancel{n+1}}))^R$ .. rewrite x as ~~n+1~~ ua

$= ((wu)a)^R$ - associativity of concatenation.

$= a(wu)^R$ defn of reversal.

$= a(u^R w^R)$ induction hypothesis.

$= (au^R)w^R$ associativity of concatenation.

$= (ua)^R . w^R$ defn of reversal.

$(wx)^R = x^R w^R$ - rewrite ua as x.

Relations on strings :

A string s is a substring of a string t iff s occurs contiguously as part of t.

For eg: aaa is substring of aaabbbaaa

aaaaa is not a substring of aaabbbaaa.

A string s is a proper substring of a string t iff s is a substring of t and $s \neq t$.

everystring is a substring of itself.

The empty string ϵ, is a substring of every string.

A string s is a prefix of t iff

$\exists x \in \Sigma^* (t = sx)$

A string s is a proper prefix of a string t iff s is a prefix of t and $s \neq t$.

every string is a prefix of itself.

the empty string ϵ, is a prefix of every string.

eg: The prefixes of abba are ϵ, a, ab, abb, abba

A string s is a Suffix of t iff $\exists x \in \Sigma^* (t = xs)$

A string s is a proper suffix of a string t iff s is a suffix of t and $s \neq t$.

Dr. GIRIJAMMA .H.A
B.E, M.Tech, Ph.D
Professor,
Computer Science & Engg. Department
RNS Institute of Technology
Channasandra, R.R. Nagar,
BANGALORE - 560 098.

Languages:

A language is a (finite or infinite) set of string over a finite alphabet Σ.

eg: Σ = {a, b}

Σ* = {ε, a, b, aa, ab, ba, bb, aab, ... }

Some languages over Σ are

∅, {ε}, {a, b}, {ε, a, aa, aaa, aaaa}, ...

Techniques for defining Languages:

eg: ① All a's preceded all b's.

let L = {w ∈ {a, b}* : all a's precede all b's in w}

The strings ε, a, aa, aabbb, and bb are in L.

The strings aba, ba and abc are not in L.

② Strings that end in a.

$L = \{x : \exists y \in \{a,b\}^* . (x = ya)\}$

The strings a, aa, aaa, bbaa, ba are in L.

The strings ε, bab and bca are not in L.

③ L = {x#y : x, y ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}* and square(x) = y}

12#144 is in L   3#8 not in L

3#9 is in L   12#12#12 not in L.

The empty language and empty string :

Empty language

let $L = \{ \} = \phi$

L is the language containing no strings.

Empty string

let $L = \{\epsilon\}$, the language that contains a single string ϵ

Note that L is different from ϕ.

A Halting problem language :

let L = {w : w is a C program that halts on all inputs}.

prefix Relation on strings :

$L_1 = \{w \in \{a,b\}^*$: no prefix of w contains b}

$= \{\epsilon, a, aa, aaa, aaaa, \ldots\ldots\}$

$L_2 = \{w \in \{a,b\}^*$: no prefix of w starts with b}

$= \{w \in \{a,b\}^*$: the first character of w is a}∪{ε

$L_3 = \{w \in \{a,b\}^*$: every prefix of w starts with b}

$= \phi$

L_3 :

Dr. GIRIJAMMA .H.A
B.E, M.Tech, Ph.D
Professor,
Computer Science & Engg. Department
RNS Institute of Technology
Channasandra, R.R. Nagar,
BANGALORE - 560 098.

Using Replication to define a language.

$L = \{a^n : n \geq 0\}$

$L = \{\epsilon, a, aa, aaa, \ldots\}$

A language Generator, enumerates ('lists') the elements of the language.

A language Recogniser, decides whether or not a string is in the language and returns true if it is and false if it isn't.

lexicographic Enumeration:

$L = \{x \in \{a,b\}^* : \text{all } a\text{'s precede all } b\text{'s}\}$

$L = \{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb,$
$aaab, \ldots\}$

functions on Languages:

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

$L_1 = \{\text{strings with an even number of } a\text{'s}\}$

$L_2 = \{\text{strings with no } b\text{'s}\} = \{\epsilon, a, aa, aaa, \ldots\}$

$L_1 \cup L_2 = \{\text{strings with an even or odd no of } a\text{'s}\}$
$= \{\epsilon, a, aa, aaa, aaaa, \ldots\}$

$L_1 \cap L_2 = \{ \epsilon, aa, aaaa, aaaaaa, aaaaaaaa, \ldots \}$

$L_2 - L_1 = \{ a, aaa, aaaaa, aaaaaaa, \ldots \}$

$\neg (L_2 - L_1) = \{$ strings with at least one b $\} \cup$
$\{$ string with an even no of a's $\}$

Concatenation of languages :

let $L_1 = \{$ Cat, dog, mouse, bird $\}$

$L_2 = \{$ bone, food $\}$

$L_1 L_2 = \{$ Catbone, catfood, dogbone, dogfood, mousebone, mousefood, birdbone, birdfood $\}$

Kleene star of L :

let L be a language defined over Σ.

$$L^* = \{\epsilon\} \cup \{ w \in \Sigma^* : (\exists k \geq 1 \; (\exists w_1, w_2, \ldots w_k \in L \; w = w_1 w_2 \ldots w_k)) \}.$$

L^* is the set of strings that can be formed by concatenating together zero or more strings from L.

$\Sigma = \{0, 1\}$ $\qquad \Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \ldots$

$\Sigma^0 = \{\epsilon\}$ $\qquad \Sigma^* = \Sigma^+ \cup \{\epsilon\}$

$\Sigma^1 = \{0, 1\}$

$\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 111, 110\}$

$\Sigma^* = \{ \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \ldots$

let $L = \{ w \in \{a, b\}^* : \#_a(w)$ is odd ad $\#_b(w)$ is even $\}$

$L = \{ abb, bab, abbaa, \ldots \}$

If $L = \emptyset$ then $L^* = \{\epsilon\}$

If $L = \{\epsilon\}$ then L^* is $\{\epsilon\}$

$L^+ = L^* - \{\epsilon\}$ & $L^+ = L L^*$

Concatenation and Reverse of languages :

Theorem : If L_1 and L_2 are language then

$$(L_1 L_2)^R = L_2^R \, L_1^R .$$

$(L_1 L_2)^R = \{ (xy)^R : x \in L_1$ and $y \in L_2 \}$

$= y^R x^R : x \in L_1$ and $y \in L_2 \}$

$= L_2^R \cdot L_1^R .$

Module - 1

Finite state machines (FSM)

Deterministic Finite state machines : (DFSM)

A finite state machine (FSM) is a Computational

device whose input is a string and whose output is one of two values that we can call Accept and Reject.

DFSM :

A telephone switching circuit can easily be modeled as a DFSM.

Formally, a DFSM is a quintuple $(K, \Sigma, \delta, s, A)$

where
- $*$ K is a finite set of states
- $*$ Σ is the input alphabet
- $*$ $s \in K$ is the start state
- $*$ $A \subseteq K$ is the set of accepting states
- $*$ δ is the transition function. which maps from $K \times \Sigma$ to K.

A Configuration of a DFSM M is an element of $K \times \Sigma^*$.

Dr. GIRIJAMMA .H.A
B.E, M.Tech, Ph.D
Professor,
Computer Science & Engg. Department
RNS Institute of Technology
Channasandra, R.R. Nagar,
BANGALORE - 560 098.

figuration of a DFSM M on input w is (S_M, w) where S_M is the start state of M.

$$(q_1, cw) \vdash_M (q_2, w) \text{ iff } ((q_1, c), q_2) \in \delta$$

$\vdash_M$ - yields in one step

$$C_1 \vdash_M^* C_2$$

A computation by M is a finite sequence of configurations $C_0, C_1, C_2, \ldots C_n$ for some $n \geq 0$ such that

* C_0 is an initial configuration, C_n is a form of (q, ϵ)

* $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M C_3 \ldots \vdash_M C_n$.

let w be an element of Σ^*. then

* M accepts w iff $(s, w) \vdash_M^* (q, \epsilon)$ for some $q \in A_M$.

Any configuration (q, ϵ) for some $q \in A_M$ is called an accepting configuration of M.

* M rejects w iff $(s, w) \vdash_M^* (q, \epsilon)$ for some $q \notin A_M$.

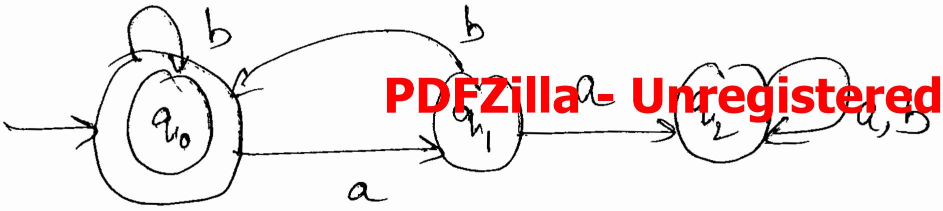
Any configuration (q, ϵ) for some $q \notin A_M$ is called rejecting configuration of M.

The language accepted by M, denoted $L(M)$ is set of all strings accepted by M.

DFSM problems

(1) let $L=\{w \in \{a,b\}^* :$ every a is immediately followed by b$\}$

$L = \{b, ab, abb, ababb, ----\}$



DFSM $M=(\{q_0,q_1,q_2\}, \{a,b\}, \delta, q_0, \{q_0\})$

where $\delta = \{ ((q_0,a,q_1), ((q_0,b), q_0), ((q_1,a), q_2), ((q_1,b), q_0), ((q_2,a), q_2), ((q_2,b), q_2))\}$

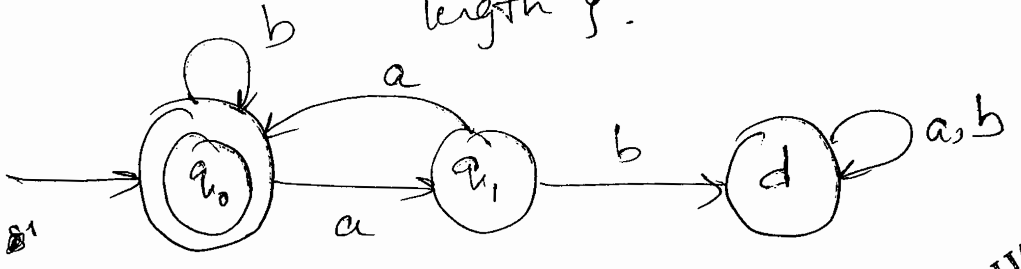
let $w = abbabab$.

M's computation is a sequence of configurations

$(q_0, abbabab), (q_1, bbabab), (q_0, babab), (q_0, abab), (q_1, bab), (q_0, ab), (q_1, b), (q_0, \epsilon)$. Since q_0 is an accepting state M accepts.

q_2 - dead state.

(2) let $L = \{w \in \{a,b\}^* :$ every a region in w is of even length$\}$.

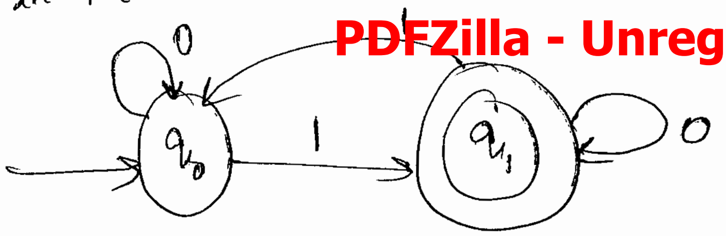


Dr. GIRIJAMMA H.A
B.E, M.Tech, Ph.D
Professor,
Computer Science & Engg. Department
RNS Institute of Technology
Channasandra, R.R. Nagar,
BANGALORE - 560 098.

③ checking for odd parity.

let $L = \{ w \in \{0,1\}^* : w$ has odd parity $\}$.

A binary string has odd parity iff the number of 1's in it is odd.



DFSMs Halt:

Theorem: every DFSM M, on input w, halts after |w| steps.

proof: on input w, M executes some computation $C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_n$, where C_0 is the initial configuration and C_n is of the form (q, ϵ) for some state $q \in K_M$.

C_n is either an accepting & a rejecting configuration

so M will halt when it reaches C_n.

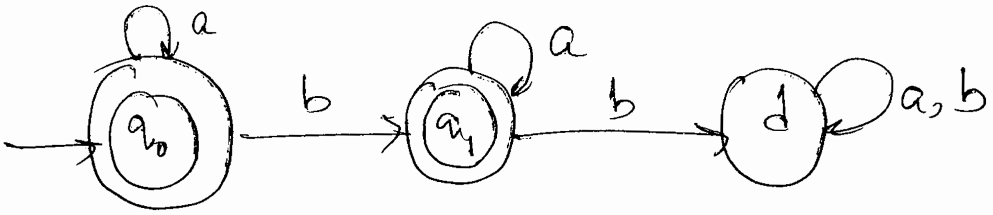
$n = |w|$ thus M will halt after $|w|$ steps.

The Regular languages :

we define the set of regular languages to be exactly those that can be accepted by DFSM.

eg : let $L = \{w \in \{a,b\}^* :$ w contains no more than one b$\}$

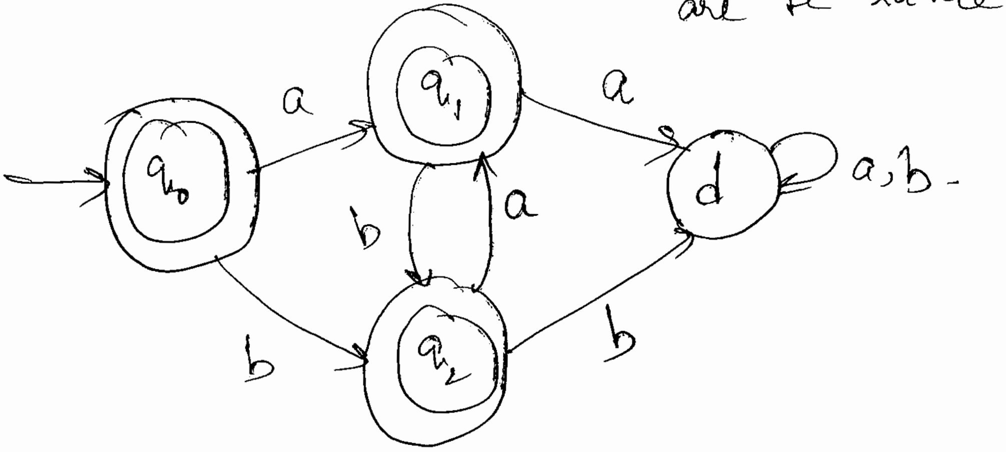
$L = \{a, b, ab, aab, aba, abaa, aaabaa, \ldots\}$



L is regular becase it can be accepted by the DFSM M.

No two consecutive characters are the same:

let $L = \{w \in \{a,b\}^* :$ no two consecutive characters are the same$\}$.



$L = \{ab, ba, aba, bab, abab, \ldots\}$

$\bar{L} = \{aa, bb, abb, abaa, baa, \ldots\}$

Floating point Numbers:

let FLOAT = { w : w is the string representation of a floating point number }.

Syntax (Rules) for floating point numbers:

* A floating point number has an optional ± sign, followed by a decimal number followed by an optional exponent.

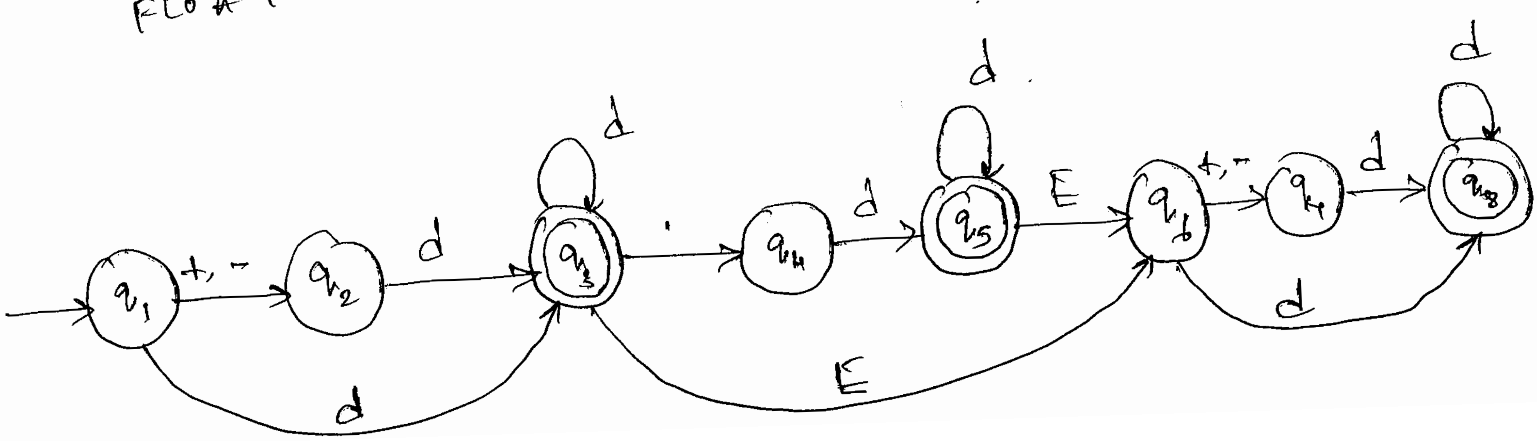
* A decimal number may be of the form x.y, where x and y are nonempty strings of decimal digits.

* An exponent begins with E and is followed by an optional sign and then an integer.

* An integer is a nonempty string of decimal digits.

eg: +3.0, 3.0, 0.3E1, 0.3E+1, -0.3E+1, -3E8 ....

FLOAT is regular because it can be accepted by DFSM



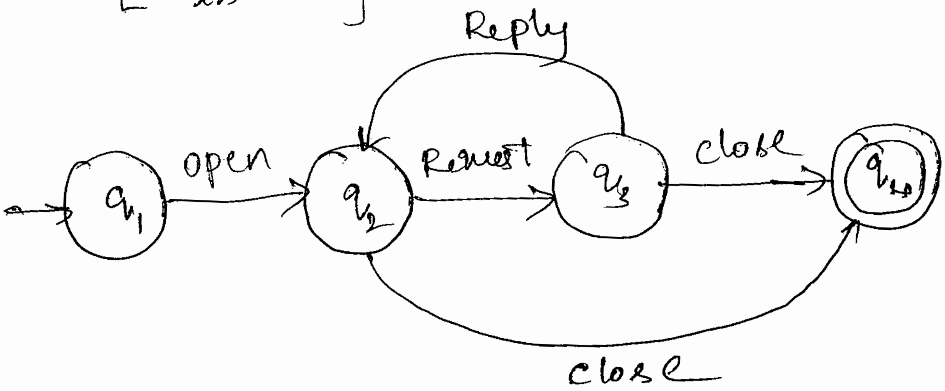
A Simple Communication protocol

let L be a language that contains all the legal sequences of messages that can be exchanged between a client and a server using a simple communication protocol.

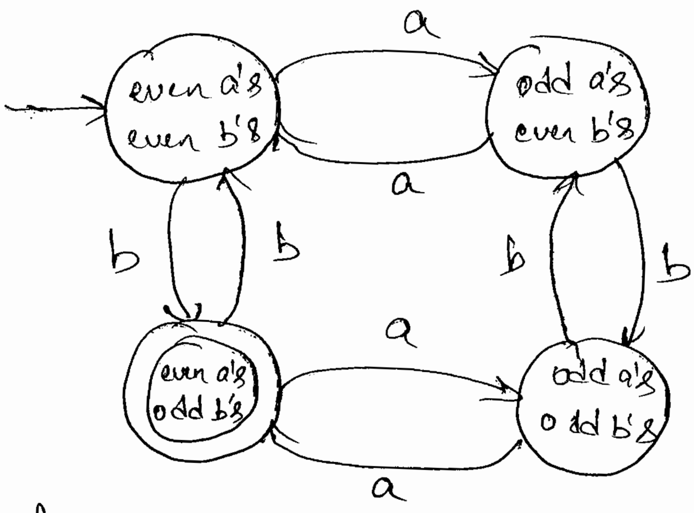
let Σ_L = {open, Request, Reply, Close}

every string in L begins with open and ends with close every request should be followed by reply except last and no unsolicited Reply's can occur.

L is regular because it is accepted by DFSM.



Designing DFSM's:

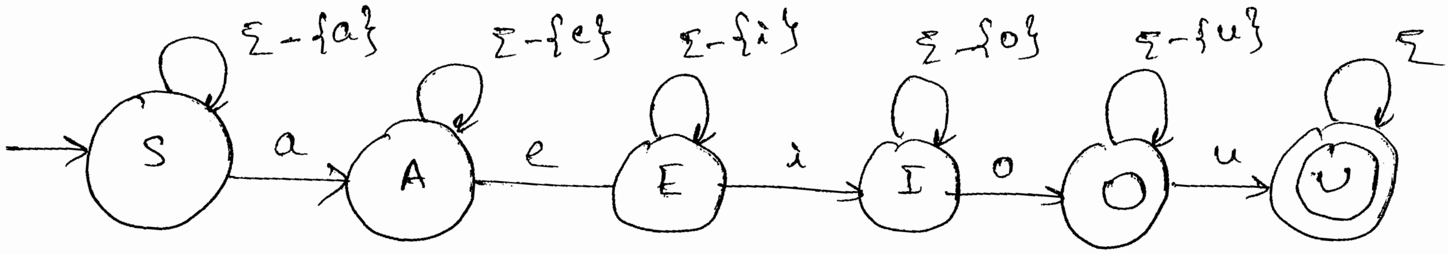


let L = {w ∈ {a, b}* : w contains an even no of a's and an odd no of b's} vowels in alphabetical order!

let $L = \{ w \in \{a-z\}^*$: all five vowels a, e, i, o and u occur in w in alphabetical order $\}$.

eg: $L = \{$ abstemious, facetious, sacrilegious, ... $\}$

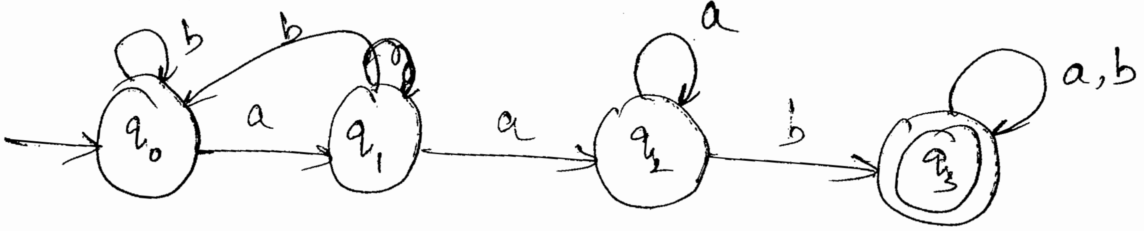
$\bar{L} = \{$ tenacious, ... $\}$



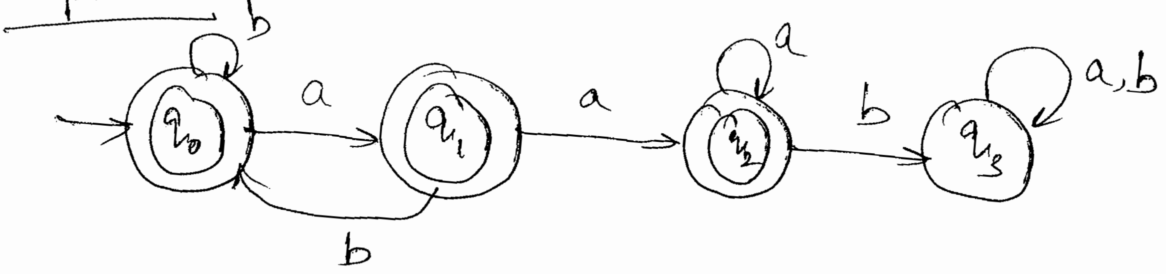
A substring that doesn't occur :

let $L = \{ w \in \{a,b\}^*$: w does not contain the substring aab $\}$

First construct DFSM with substring accepting ad then complement it. by making final states as non-final



Complement :



Dr. GIRIJAMMA .H.A
B.E, M.Tech, Ph.D
Professor,
Computer Science & Engg. Department
RNS Institute of Technology
Channasandra, R.R. Nagar,
BANGALORE - 560 098.

The missing letter language :

let $\Sigma = \{a, b, c, d\}$

let $L_{\text{Missing}} = \{ w :$ there is a symbol $a_i \in \Sigma$ not appearing in $w \}$.

L_{missing} is regular

DFSM : * The start state : all letters are still missing

After one character has been read, M could be in any one of,

* a read, so b, c and d still missing.
* b read, a, c, d still missing.
* c read, a, b, d still missing
* d read so a, b, c still missing.

After second character read,

* a and b read so c, d still missing
* a ad c read so b, d . . .
* a and d . . - -

Next after third character.

ab, c read d missing

⋮

After fourth character -

* All characters read, so nothing is missing.

very state except he last is an accepting state

n is Complicated. but it would be possible to write

Nondeterministic FSMs (NDFSMs)

A nondeterministic FSM (NDFSM) M is a quintuple (K, Σ, Δ, s, A), where

K is a finite set of states.

Σ is an alphabet.

$s \in K$ is the start state.

$A \subseteq K$ is the set of final states.

Δ is the transition relation. It is a finite subset of

$$(K \times (\Sigma \cup \{\varepsilon\})) \times K.$$

In other words, each element of Δ contains a (state, input symbol & ε) pair, and a new state.

Let w be an element of Σ^*. Then we will say that

* M accepts w iff at least one of its computations accepts.

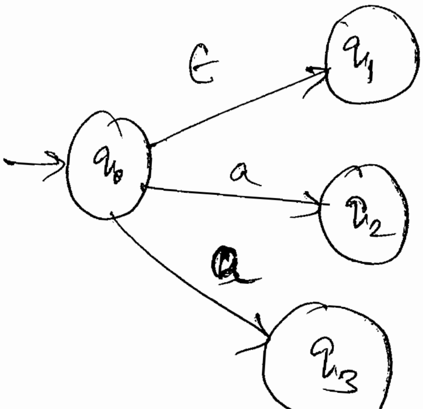
* M rejects w iff none of its computations accept.

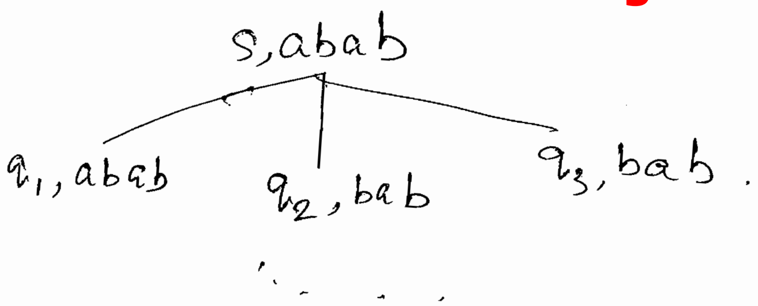
L(M) - is the set of all strings accepted by M.

Differences between DFSM and NDFSM.

DFSM	NDFSM
1) In every configuration, a DFSM can make exactly one move.	1) Δ be an arbitrary relation, this is not necessarily true for an NDFSM.

eg:

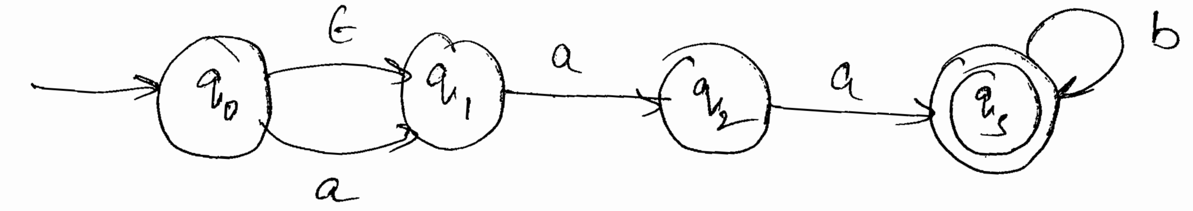




An optional initial a:

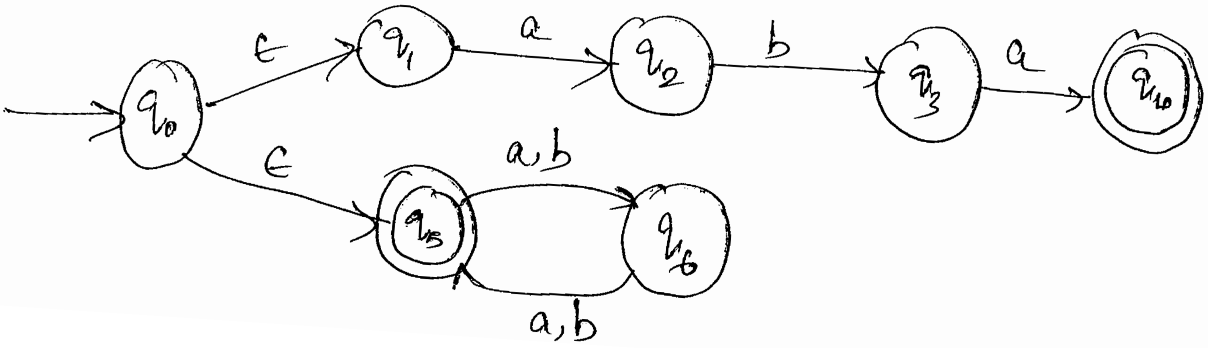
let $L = \{w \in \{a,b\}^* :$ w is made up of an optional a followed by aa followed by zero & more b's$\}$.

The following NDFSM accepts L:



Two different sublanguages:

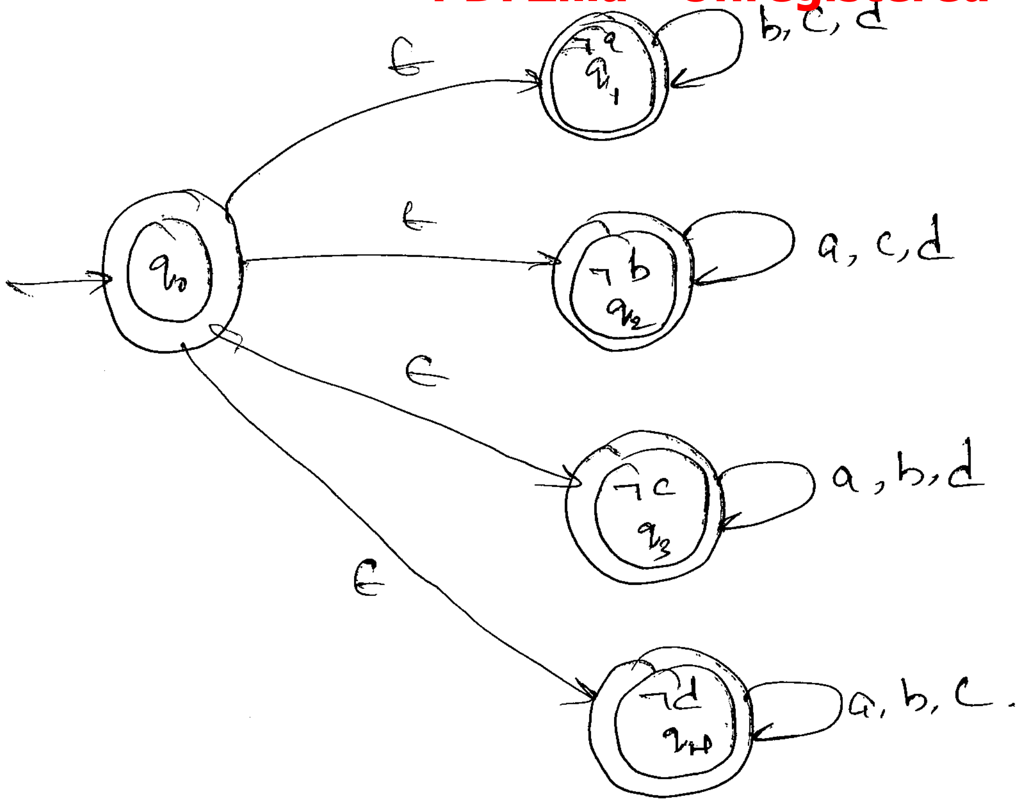
let $L = \{w \in \{a,b\}^* ; w = aba$ & $|w|$ is even$\}$.



The upper machine accepts $\{w \in \{a,b\}^* : w = aba\}$.

The lower one accepts $\{w \in \{a,b\}^* : |w| \text{ is even}\}$.

The missing letter language :



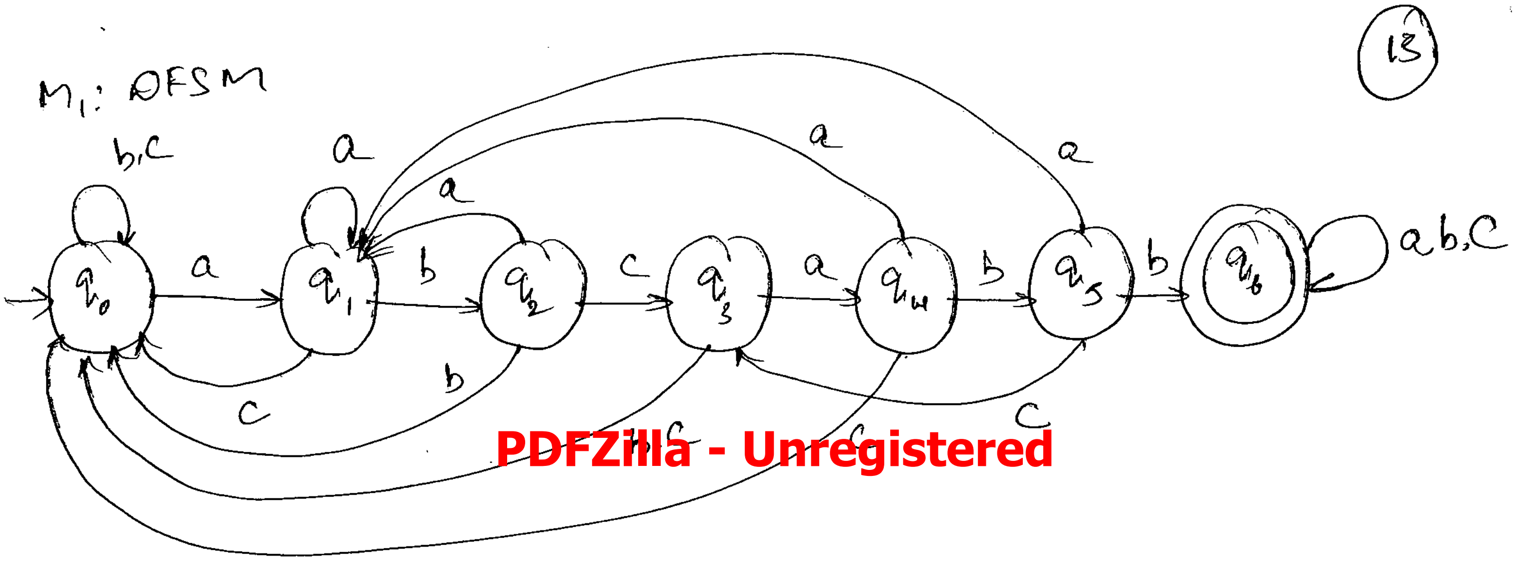
NDFSMs for pattern and substring matching

Exploiting Nondeterminism for keyword matching

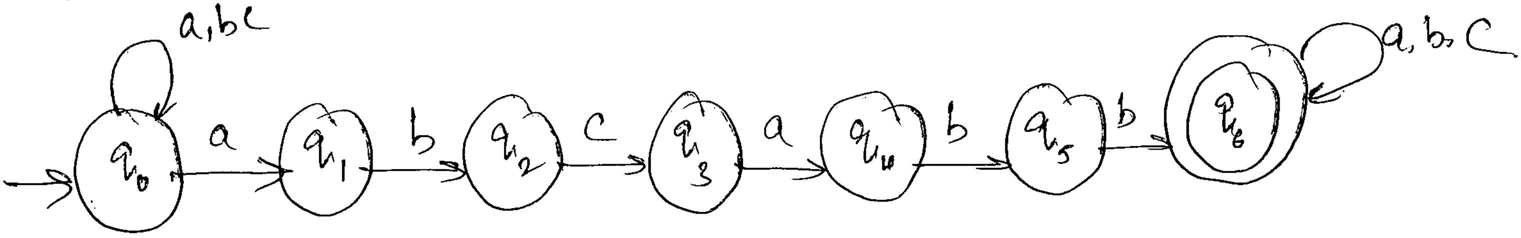
let $L = \{w \in \{a,b,c\}^* : \exists x, y \in \{a,b,c\}^* \ (w = x\,abcabb\,y)\}$.

In other words, w must contain at least one occurence of the substring abcabb.

M_1: DFSM



M_2: NDFSM

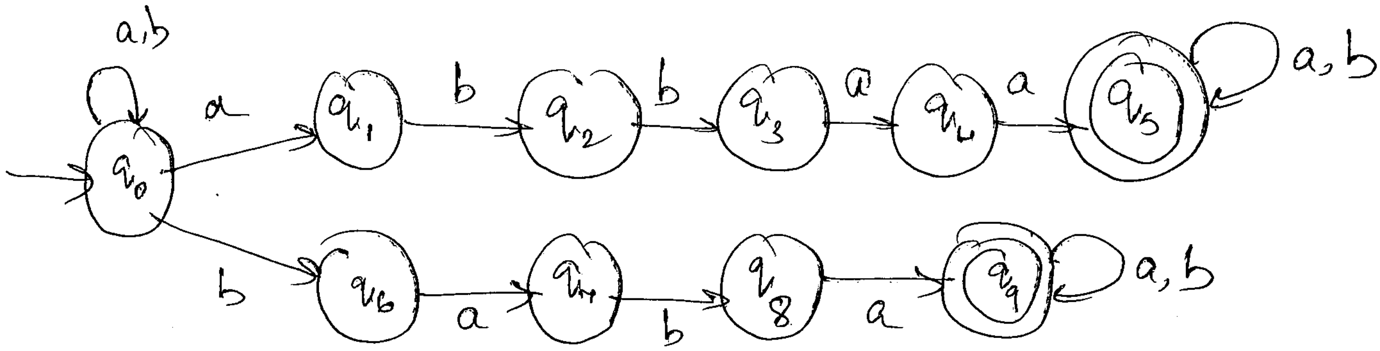


String searching is a fundamental operation in every word processing & text editing system.

multiple keywords :

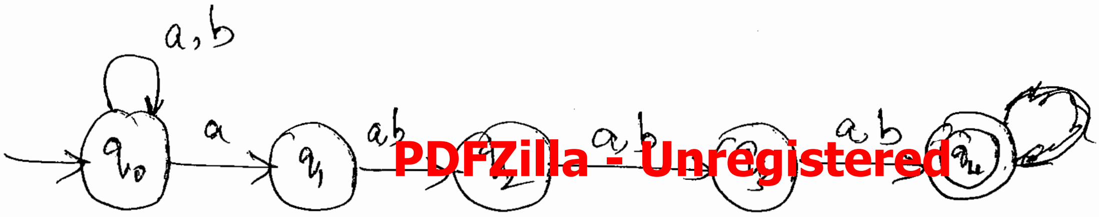
let $L = \{ w \in \{a,b\}^* : \exists x, y \in \{a,b\}^* ((w = xabbaay) \vee (w = xbabay)) \}$.

In words, w contains at least one occurence of the substring abbaa & the substring baba.



Other kinds of patterns:

let $L = \{ w \in \{a,b\}^* :$ the fourth from the last character is $a\}$



$L = \{ aaaa, aaba, abbb, abaabb, \ldots \}$

Analyzing Nondeterministic FSMs:

Handling ε-transitions

$eps : K_M \rightarrow \mathcal{P}(K_M)$

$eps(q)$, where q is some state in M, to be the set of states of M that are reachable from q by following zero or more ε-transitions.

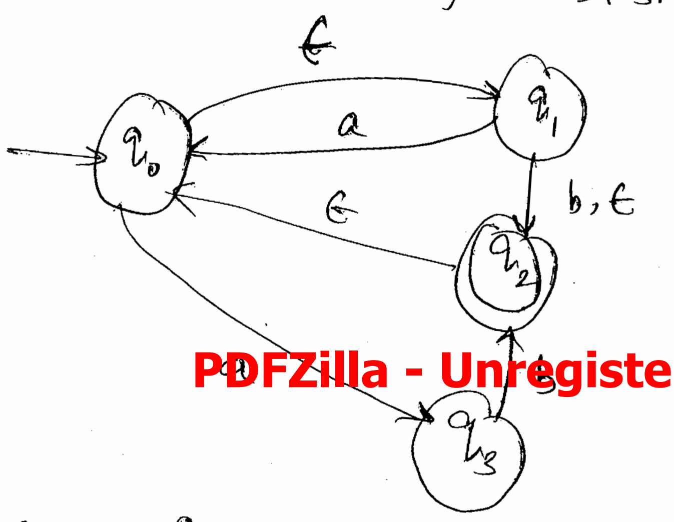
$eps(q) = \{ p \in K : (q, w) \vdash_M^* (p, w) \}$.

or $eps(q)$ is the closure of $\{q\}$ under the relation $\{(p, r) :$ there is a transition $(p, \epsilon, r) \in \Delta \}$.

The following algorithm computes eps:

```
eps(q: state) =
    1. result = {q}
    2. while there exists some $p \in$ result and some
       $r \notin$ result and some transition $(p, \epsilon, r) \in \Delta$ do:
           insert r into result.
    3. Return result
```

eg: Consider the following NDFSM M:



$eps(q_0) = \{q_0, q_1, q_2\}$

$eps(q_1) = \{q_0, q_1, q_2\}$

$eps(q_2) = \{q_0, q_1, q_2\}$

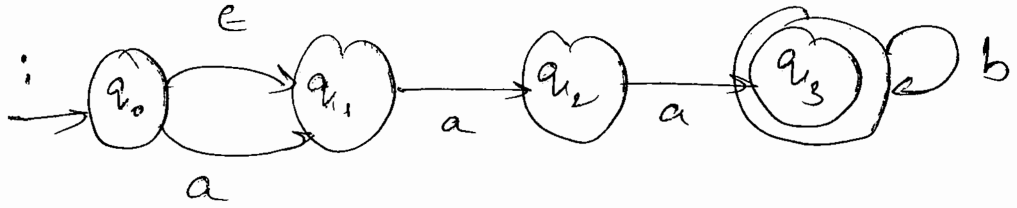
$eps(q_3) = \{q_3\}$.

ε-loop: a loop that can be traversed by following only ε-transitions.

A Simulation Algorithm: For tracing all paths in parallel through an NDFSM M:

```
ndfsmsimulate (M: NDFSM, w: string) =
1. Current-state = eps(s)
2. while any input symbols in w remain to be read do:
    2.1. c = get-next-symbol(w)
    2.2. next-state = ∅
    2.3. For each state q in Current-state do:
            For each state p such that (q, c, p) ∈ Δ do
                next-state = next-state ∪ eps(p).
    2.4. Current-state = next-state
3. If current-state contains any states in A, accept.
                                   Else reject.
```

eg:



w = aab.

eps(q0) = {q0, q1}   eps(q2) = {q2}

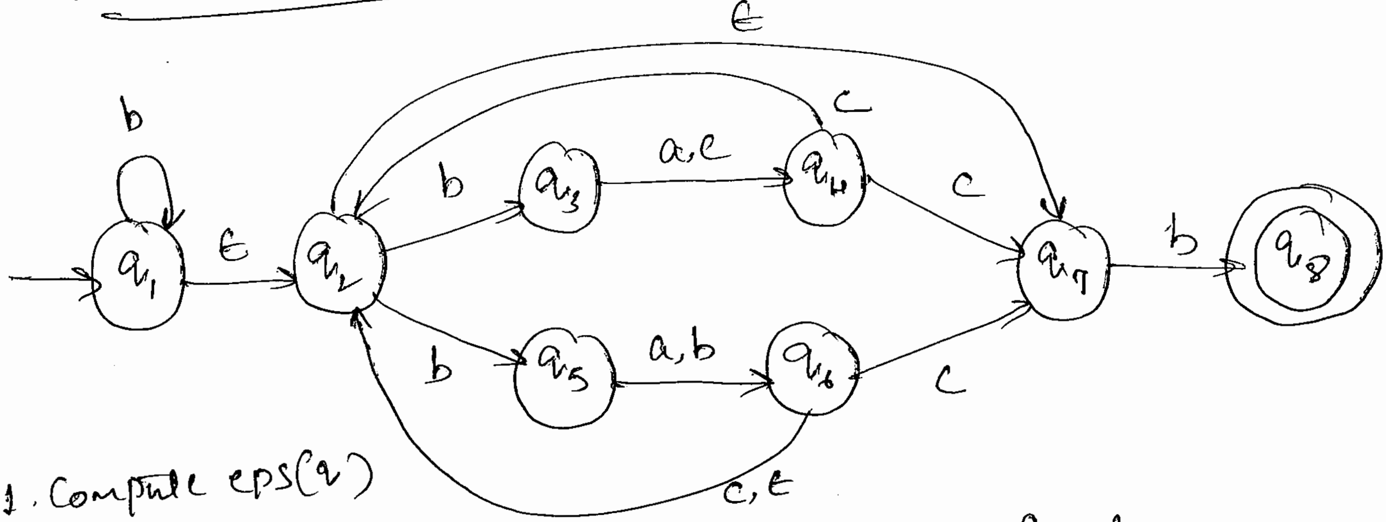
eps(q1) = {q1}   eps(q3) = {q3}

$(q_0, a, q_1) \in \Delta$,. $(q_1, a, q_2) \in \Delta$. $(q_2, b, q_3) \notin \Delta$ ✗

$(q_1\ a, q_2) \in \Delta$ .. $(q_2, a, q_3) \in \Delta$, $(q_3, b, q_3) \in \Delta$ ⇐

The equivalence of Nondeterministic and Deterministic FSM

NDFSM to DFSM: first problem and then algorithm



1. Compute eps(q)

eps(q1) = {q1, q2, q7}   eps(q7) = {q7}

eps(q2) = {q2, q7}   eps(q8) = {q8}

eps(q3) = {q3}

eps(q4) = {q4}

eps(q5) = {q5}

eps(q6) = {q2, q6, q7}

$= eps(S) = \{q_1, q_2, q_7\}$

3. compute δ';

Active-state = $\{ (q_1, q_2, q_7\} \}$.

$\delta'(\{q_1, q_2, q_7\}, a) = \phi$

$\delta'(q_1, q_2, q_7\}, b) = eps(\{q_1, q_3, q_5, q_8\})$

$= \{q_1, q_2, q_3, q_5, q_7, q_8\}$

$\delta'(\{q_1, q_2, q_7\}, c) = eps(\phi)$

$= \phi$

Active-States = $\{ \{q_1, q_2, q_7\}, \phi, \{q_1, q_2, q_3, q_5, q_7, q_8\} \}$

Consider ϕ: $((\phi, a), \phi), ((\phi, b), \phi), ((\phi, c), c) \}$

So. ϕ is a dead state.

Consider

$((\{q_1, q_2, q_3, q_5, q_7, q_8\}, a), \{q_2, q_4, q_6, q_7\})$.

$((q_1, q_2, q_3, q_5, q_7, q_8\}, b), \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\})$

$((q_1, q_2, q_3, q_5, q_7, q_8\}, c), \{q_4\})$.

Active-states = $\{ \{q_1, q_2, q_7\}, \phi, \{q_1, q_2, q_3, q_5, q_7, q_8\},$

$\{q_2, q_4, q_6, q_7\}, \{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}, \{q_4\} \}$.

Consider $\{q_2, q_4, q_6, q_7\}$

$\delta'(\{q_2, q_4, q_6, q_7)\}, a\} = \emptyset$

$\delta'(\{q_2, q_4, q_6, q_7\}, b\} = \{q_3, q_5, q_8\}$

$\delta' \{q_2, q_4, q_6, q_7\}, c\} = \{q_2, q_7\}$

ℛ DFSM Transition table

	a	b	c
→ $\{q_1, q_2, q_7\}$	$\emptyset$	$\{q_1, q_2, q_3, q_5, q_7, q_8\}$	$\emptyset$
$\{q_1, q_2, q_3, q_5, q_7, q_8\}$	$\{q_2, q_4, q_6, q_7\}$	$\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}$	$\{q_4\}$
$\{q_2, q_4, q_6, q_7\}$	$\emptyset$	$\{q_3, q_5, q_8\}$	$\{q_2, q_7\}$
$\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}$	$\{q_2, q_4, q_6, q_7\}$	$\{q_1, q_2, q_3, q_5, q_6, q_7, q_8\}$	$\{q_2, q_4, q_7\}$
$\{q_4\}$	$\emptyset$	$\emptyset$	$\{q_2, q_7\}$
$q_3, q_5, q_8\}$	$\{q_2, q_4, q_6, q_7\}$	$\{q_2, q_6, q_7\}$	$\{q_4\}$
$\{q_2, q_7\}$	$\emptyset$	$\{q_3, q_5, q_8\}$	$\emptyset$
$\{q_2, q_4, q_7\}$	$\emptyset$	$\{q_3, q_5, q_8\}$	$\{q_2, q_7\}$
$\{q_2, q_6, q_7\}$	$\emptyset$	$\{q_3, q_5, q_8\}$	$\{q_2, q_7\}$

The Equivalence of Nondeterministic and Deterministic FSMs

Theorem: If there is a DFSM for L, there is an NDFSM fr L.

Proof: let M be a DFSM that accepts some language L. M is also an NDFSM that happens to contain no ε-transitions. So we can claim NDFSM is simply M

Theorem: If there is an NDFSM fr L, there is a DFSM fr L.

Statement: Given an NDFSM $M = (K, \Sigma, \Delta, S, A)$ that accepts some language L. there exists an equivalent DFSM that accepts L.

Proof: The proof is by construction of an equivalent DFSM M'. The construction is based on te function eps and on te simulation algorithm.

so $M' = \{K', \Sigma, \delta', s', A'\}$

* K' contains one state fr each element of $P(K)$

* $s' = eps(s)$

* $A' = \{Q \subseteq K : Q \cap A \neq \emptyset\}$

* $\delta'(Q, c) = \cup \{eps(p) : \exists q \in Q((q, c, p) \in \Delta)\}$

The following algorithm computes M' given M;

```
ndfsmtodfsm(M: NDFSM) =
1. For each state q in K do:
       Compute eps(q)
2. s' = eps(s)
3. Compute δ':
     a. active-states = {s'}
     b. δ' = ∅
     c. while there exists some element Q of active-states
        for which δ' has not yet been computed do:
        For each character c in Σ do:
            new-state = ∅
            For each state q in Q do:
            For each state p such that (q, c, p) ∈ Δ do:
            new-state = new-state ∪ eps(p).
            Add the transition (Q, c, new-state) to δ'
            If new-state ∉ active-states then insert it
            into active-states.
4. K' = active-states
5. A' = { Q ∈ K' : Q ∩ A ≠ ∅ }
```

From FSMs to operational systems;

An FSM is an abstraction.

FSMs for real problems can be turned into operational systems in any no of ways!

* An FSM can be translated into a circuit design and implemented directly in hardware.

eg! parity checking FSM

* An FSM can be simulated by a general purpose interpreter.

* An FSM can be used as a specification for some critical aspect of the behavior of a complex system. The specification can then be implemented in software.

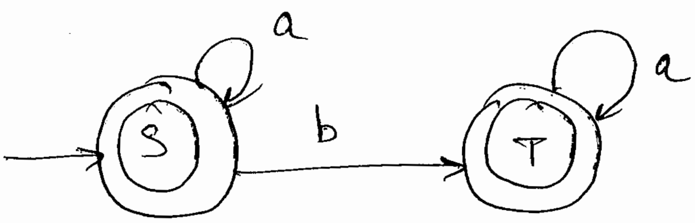
Simulators for FSMs

once FSM has been created, next step is to simulate its executed.

* Simulating DFSMs.

eg: Consider the following deterministic FSM M that accepts the language

L = { w ∈ {a,b}* : w contains no more than one b}.



al a specification for the following program

until accept or reject do;

```
S :    s = get-next-symbol
       If s=end-of-file then accept
       else if s=a then goto S.
       else if S=b then goto T.
T :    s = get-next-symbol.
       If s=end-of-file then accept.
       else if s=a then goto T.
       else if s=b then reject.
```

simple interpreter for a DFSM M=(K, Σ, δ, s, A):

```
dfsmsimulate (M : DFSM, w: string) =
1. st = s
2. Repeat :
     2.1  c=get-next-symbol(w).
     2.2  If c ≠ end-of-file then:
            2.2.1  st = δ (st, c).
          until  c=end-of-file
3. If st ∈ A then accept else reject.
```

Simulating Nondeterministic FSMs

one solution is:

ndfsmconvertandsimulate(M : NDFSM) =
    dfsmsimulate(ndfsmtodfsm(M)).

If M is K states, it could take time and space equal to $O(2^K)$ just to do the conversion and simulation would take equal to $O(|w|)$. So it takes $O(2^K) + O(|w|$

Alternate method: ndfsmsimulate,

NDFSM $M = (K, \Sigma, \Delta, s, A)$ running on an input string w:

```
ndfsmsimulate(M : NDFSM, w : string) =
1. Declare the set st.   /* st will hold the current state
                            (a set of states from K).
2. Declare the set st1.  /* st1 will be built to contain
                            the next state.
3. st = eps(s)
4. Repeat:
      c = get-next-symbol(w).
      If c ≠ end-of-file then do:
         st1 = ∅                                  O(|K|^2) steps.
         For all q ∈ st do:                       Total cost = O(w.|K|^2)
            For all r : (q, c, r) ∈ Δ do:
               st1 = st1 ∪ eps(r).
            st = st1.
            If st = ∅ then exit.
   until c = end-of-file
5. If st ∩ A ≠ ∅ then accept else reject.
```

Minimizing FSMs

DFSM M is minimal iff there is no other DFSM M' such that L(M) = L(M') and M' has fewer states than M does.

1. Given a language L, is there a minimal DFSM that accepts L?
2. If there is a minimal machine, is it unique?
3. Given a DFSM M that accepts some language L, can we tell whether M is minimal?
4. Given a DFSM M, can we construct a minimal equivalent DFSM M'?

Building a minimal DFSM for a language:

x and y are indistinguishable with respect to L, which we will write as $x \approx_L y$ iff:

$$\forall z \in \Sigma^* \text{ (either both } xz \text{ and } yz \in L \text{ or neither is)}.$$

x, y are prefixes of some longer string.

$\approx_L$ is a relation $x \approx_L y$ when either xz and yz are in L or both are not.

How $\approx_L$ Depends on L:

If $L=\{a\}^*$, then $a \approx_L aa \approx_L aaa$.

If $L=\{w\in\{a,b\}^* : |w| \text{ is even}\}$,

then $a \approx_L aaa$, but it not the case that $a \approx_L aa$ because if $z=a$, we have $aa\in L$ but $aaa\notin L$.

x and y are distinguishable with respect to L, iff they are not indistinguishable

$\approx_L$ is an equivalence relation because it is:

* Reflexive: $\forall x\in\Sigma^* (x \approx_L x)$, because $\forall x,z \in \Sigma^* (xz\in L \Leftrightarrow xz\in L)$

* Symmetric: $\forall x,y\in\Sigma^* (x\approx_L y \rightarrow y\approx_L x,)$. because $\forall x,y,z\in\Sigma^* ((xz\in L\Leftrightarrow yz\in L) \Leftrightarrow (yz\in L \Leftrightarrow xz\in L))$.

* Transitive: $\forall x,y,z\in\Sigma^*$ $(((x\approx_L y)\wedge(y\approx_L w))\rightarrow(x\approx_L w))$, because: $\forall x,y,z\in\Sigma^* (((xz\in L\Leftrightarrow yz\in L)\wedge(yz\in L\Leftrightarrow wz\in L)) \rightarrow (xz\in L \Leftrightarrow wz\in L))$.

Notations for equivalence classes of $\approx_L$:

[1], [2], ... explicitly numbered classes

[x] describes the equivalence class that contains the string x

[some logical expression P] strings that satisfy P.

① Determining $\approx_L$:

let $\Sigma = \{a, b\}$ and let $L = \{w \in \Sigma^*$: every a is immedi
followed by a

[1] $\{\epsilon, b, abb, ---\}$ [all strings in L]

[2] $\{a, abbbba, ----\}$ [all strings that end in a and have no prior a that is not followed by a b]

[3] $\{aa, abaa, ..\}$ [all strings that contain at least one instance of aa]

② when more than one class contains strings in L

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

let $L = \{w \in \{a, b\}^*$: no two adjacent characters are the same $\}$.

The equivalence classes of $\approx_L$ are:

[1] [ε} [ε]

[2] [a, aba, ababa, ---] [all nonempty strings that end in a and have no identical adjacent characters]

[3] [b, ab, bab, abab, ---] [all nonempty strings that end in b and have no identical adjacent characters]

[4] [aa, abaa, ababb, ---] [all strings that contain at least one pair of identical adjacent characters]

$\approx_L$ for A^nB^n

let $\Sigma = \{a, b\}$. let $L = A^nB^n = \{a^nb^n : n \geq 0\}$

$[1] \quad [\epsilon]$

$[2] \quad [a]$

$[3] \quad [aa]$

$[4] \quad [aaa]$

$\{[n] : n$ is a positive integer and $[n]$ contains the single string $a^{n-1}\}$

$\approx_L$ has an infinite no of equivalence classes.

A^nB^n. This is not regular.

equivalence classes of $\approx_L$ are going to correspond to the states of the machine to accept L, Then there will be finite no of equivalence classes. precisely in case L is regular.

Theorem: $\approx_L$ imposes a lower bound on the minimum no of states of a DFSM for L

Theorem 1 let L be a regular language and let $M = (K, \Sigma, \delta, s, A)$ be a DFSM that accepts L. The no of states in M is greater than or equal to the no of equivalence classes of $\approx_L$.

proof: Suppose that the no of states in M were less then the no of equivalence classes $\approx_L$. Then by pigeon hole principle, there must be at least one state q that contains strings from atleast two equivalence classes of $\approx_L$.

$|K|$ is equal to the no of equivalence class of $\approx_L$.

Theorem: There exists a unique minimal DFSM for every regular language.

Building a minimal DFSM from $\approx_L$

let $L = \{w \in \{a,b\}^* :$ no two adjacent characters are the same $\}$.

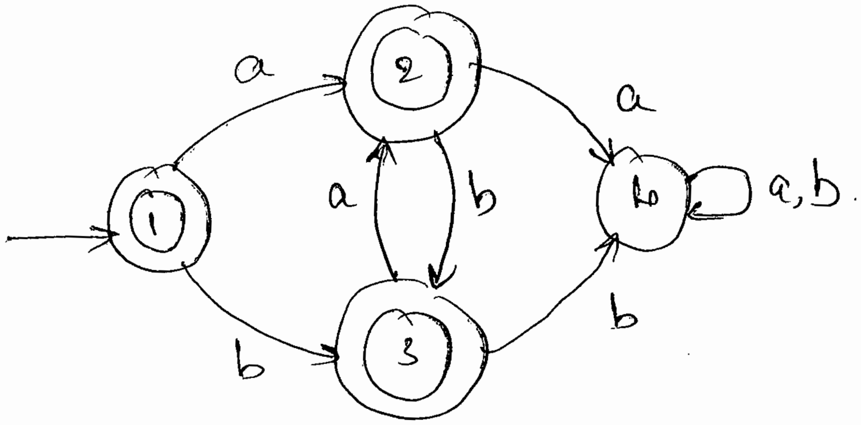
The equivalence classes of $\approx_L$ are:

[1] $[\epsilon]$ $\{\epsilon\}$

[2] [a, aba, ababa, ..] {strings that end with a}

[3] [b, ab, bab, abab ..] {strings that end with b}

[4] [aa, abaa, ababb ..] {strings that contain one pair of identical adjacent characters}



We build minimal DFSM M to acept L as follows.

- The equivalence classes of $\approx_L$ become the states of M
- The start state is $[\epsilon] = [1]$
- The Accepting states are all equivalence classes that contain [illegible] namely [1], [2] and [3]

Myhill-Nerode Theorem:

Theorem: A language L is regular iff the number of equivalence classes of $\approx_L$ is finite

Proof: L regular $\longrightarrow$ the no of equivalence classes of $\approx_L$ is finite

by Theorem 5.6.

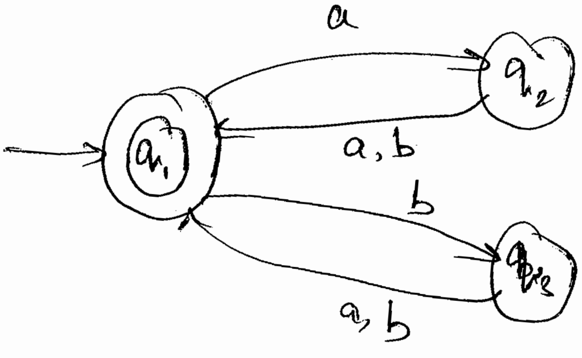
The number of equivalence classes of $\approx_L$ is finite $\longrightarrow$ L regular

by. 5.5 theorem.

Minimizing an existing DFSM.

let $L = \{w \in \Sigma^* : |w| \text{ is even}\}$

Consider the following FSM that accepts L:

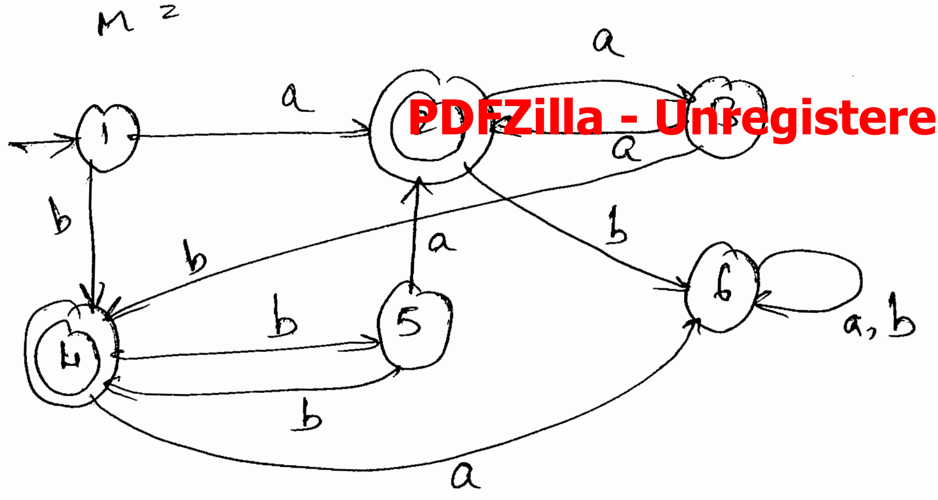


In this machine, state $q_2 \equiv$ state q_3

using minDFSM to find a minimal machine

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

let. M =



	a	b
→ 1	{2}	{4}
* 2	{3}	{6}
3	{2}	{4}
* 4	6	5
5	2	4
* 6	6	6

min DFSM :

Initialy classes , = { [2, 4] , [1, 3, 5, 6] }

Step 1 :

((2, a) , [1, 3, 5, 6]) ((4, a), [1, 3, 5, 6])

((2, b) , [1, 3, 5, 6]) ((4, b), [1, 3, 5, 6])

((1, a) , [2, 4]) ((3, a), [2, 4])

((1, b), [2, 4]) ((3, b), [2, 4])

((5, a), [2, 4]) ((6, a), [1, 3, 5, 6])

((5, b), [2, 4]) ((6, b), [1, 3, 5, 6])

There are two different patterns, so we must split into two classes [1, 3, 5] and [6].

classes = { [2, 4], [1, 3, 5], [6] }

Step 2:

((2,a), [1, 3, 5]) ((4, a), [6]) These two must be split

((2, b), [6]) ((4, b), [1, 3, 5])

((1,a), [2,4]) ((3, a), [2, 4]) ((5,a), [2, 4])

((1,b), [2,4]) ((3, b), [2,4]) ((5,b), [2, 4])

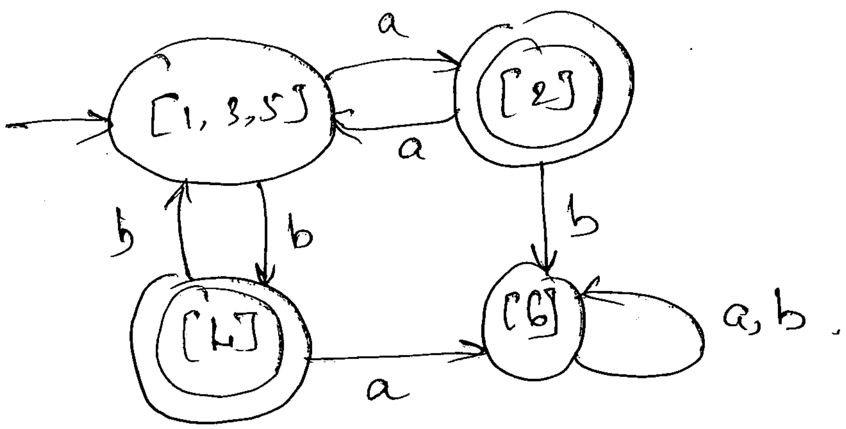
classes = { [2], [4], [1, 3, 5], [6] }

At step 3:

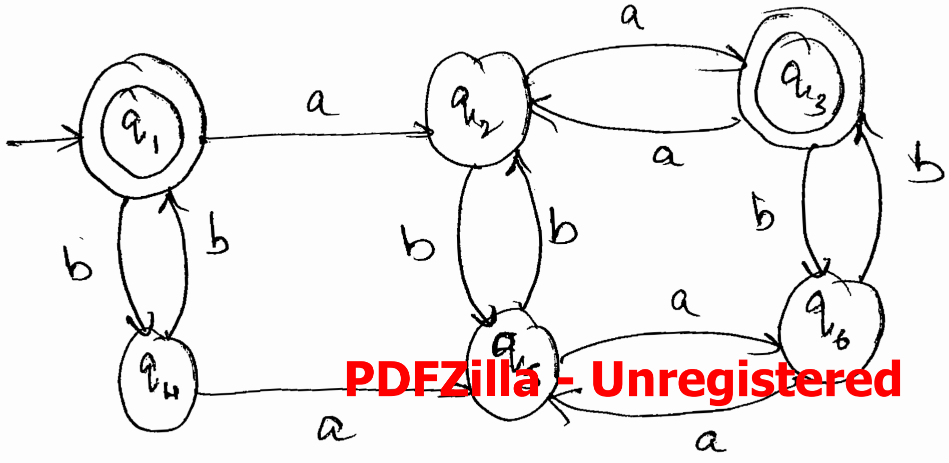
((1,a), [2]) ((3,a), [2]) ((5,a), [2])

((1,b), [4]) ((3,b), [4]) ((5,b), [4])

classes = { [2], [4], [1, 3, 5], [6] }



2



	a	b
→*1	2	4
2	3	5
* 3	2	6
4	5	1
5	6	2
6	5	3

Initially,

equivalence classes = { [1, 3], [2, 4, 5, 6] }

Step 1 :

	a	b
[1]	[2, 4, 5, 6]	[2, 4, 5, 6]
[3]	[2, 4, 5, 6]	[2, 4, 5, 6]

	a	b
[2]	[1, 3]	[2, 4, 5, 6]
[4]	[2, 4, 5, 6]	[1, 3]
[5]	[2, 4, 5, 6]	[2, 4, 5, 6]
[6]	[2, 4, 5, 6]	[1, 3]

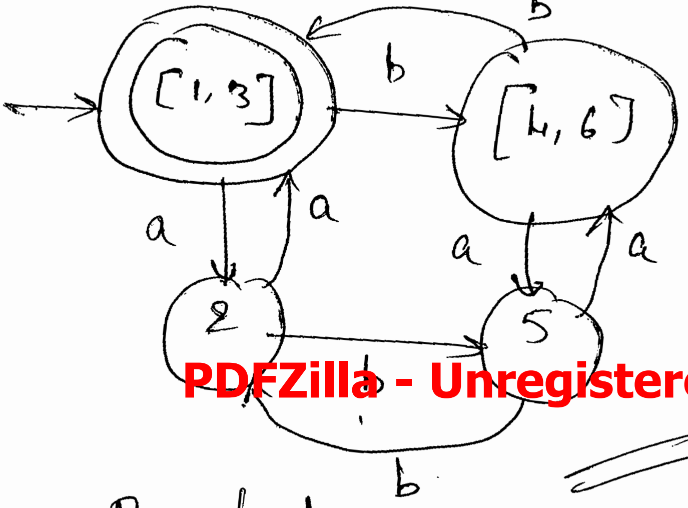
Step 2 :

equivalence classes = { [1, 3], [4, 6], [2], [5] }

	a	b
[1]	[2]	[4, 6]
[3]	[2]	[4, 6]

	a	b
[4]	[5]	[1, 3]
[6]	[5]	[1, 3]

Equivalence classes = { [1, 3], [4, 6], [2], [5] } (2



Minimized DFA

(3)

	0	1
→ A	B	F
B	G	C
* C	A	C
D	C	G
E	H	F
F	C	G
G	G	E
H	G	C

write transition graph of diagram

Initially, { [C] [A, B, D, E, F, G, H] }

	0	1
A	[A, B, D, E, F, G, H]	[A, B, D, E, F, G, H]
B	[A, B, D, E, F, G, H]	[C]
D	[C]	[A, B, D, E, F, G, H]
E	[A, B, D, E, F, G, H]	[A, B, D, E, F, G, H]
F	[C]	[A, B, D, E, F, G, H]
G	[A, B, D, E, F, G, H]	[A, B, D, E, F, G, H]
H	[A, B, D, E, F, G, H]	[C]

Equivalence class = { [C], [A, E, G], [B, H], [D, F] }

Step 2:

	0	1
[A]	[B, H]	[D, F]
[E]	[B, H]	[D, F]
[G]	[A, E, G]	[A, E, G]

	0	1
[B]	[A,E,G]	[C]
[H]	[A,E,G]	[C]

	0	1
[D]	[C]	[A, E, G
[F]	[C]	[A, E, G

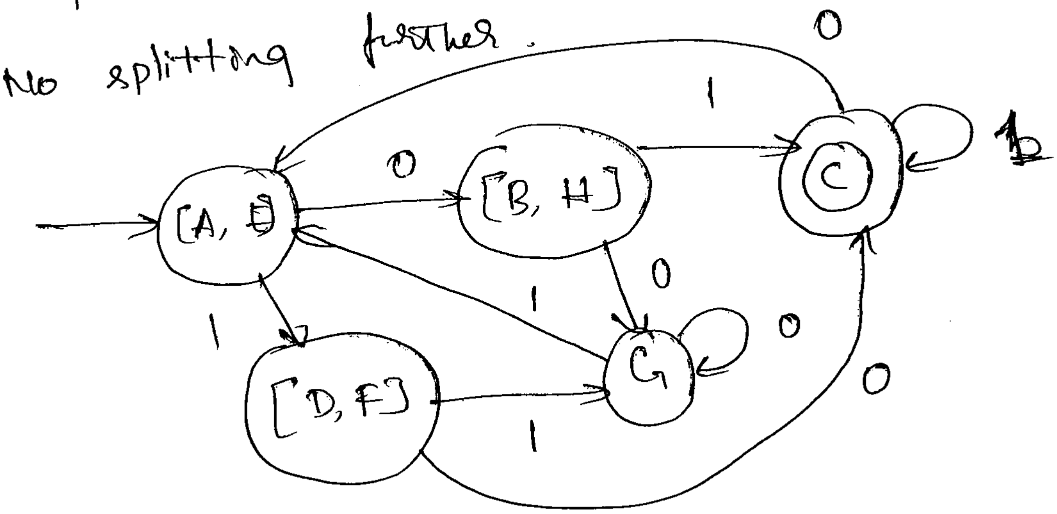
Equivalence classes = { [C], [A, E], [G], [B, H], [D, F] }

	0	1
[A]	[B, H]	[D, F]
[E]	[B, H]	[D, F]

	0	1
[B]	[G]	[C]
[H]	[G]	[C]

	0	1
[D]	[C]	[G]
[F]	[C]	[G]

No splitting further.



5.5.

Theorem: There exits a unique minimal DFSM for every ~~languag~~. Regular language.

Theorem: let L be a regular language over some alphabet Σ. Then there is a DFSM M that accepts L and has precisely n states where n is the number of equivalence classes of $\approx_L$.

Proof: The proof is by construction of

$M = (K, \Sigma, \delta, s, A)$ where:

* K contains n states, one for each equivalence class of $\approx_L$.
* $s = [\epsilon]$, the equivalence class of ϵ under $\approx_L$.
* $A = \{ [x] : x \in L \}$
* $\delta([x], a) = [xa]$.

For this construction to prove the theorem, we must show:

* K is finite. Since L is regular, it is accepted by some DFSM M'. M' has some finite no of states m. By theorem 5.4 $n \leq m$. So K is finite.
* δ is a function. It produces a unique value and defined for all (state, input) pairs.
* $L = L(M)$.

$\forall s, t\ (([\epsilon], st) \vdash_M^* ([s], t))$

M starts in its start state and has a strong s and t.

Induction on $|s|$.

If $|s| = 0$ then we have $([\epsilon], \epsilon t) \vdash_M^* ([\epsilon], t)$.

$|s| = k$.

$|s| = k+1$.

$|s| \geq 1$, So $s = yc$ where $y \in \Sigma^*$ and $c \in \Sigma$, we have

/* M reads the first k characters:

$([\epsilon], yct) \vdash_M^* ([y], ct)$

/* M reads one more character:

$([y], ct) \vdash_M^* ([yc], t)$

/* Combining those two, after M has read k+1 characters:

$([\epsilon], yct) \vdash_M^* ([yc], t)$

$([\epsilon], st) \vdash_M^* ([s], t)$

let t be ϵ. and let s be any string in Σ^*.

$([\epsilon], s) \vdash_M^* ([s], \epsilon)$.

M will accept s iff $[s] \in A$.

So M accepts precisely those strings that are in M

* There exists no smaller machine M# that also accepts L. & There is no different machine M # that has n states and accepts L.