18CSC301T Formal Language and Automata

UNIT-1

UNIT 1

- Introduction to Automaton
- Mathematical concepts
- Formal Languages: Strings, Languages, Properties
- Finite Representation : Regular Expressions
- Problems related to regular expressions
- Finite Automata :Deterministic Finite Automata
- Nondeterministic Finite Automata
- Finite Automaton with €- moves
- Problems related to Deterministic and Nondeterministic Finite Automata
- Problems related to Finite Automaton with €-moves
- Minimization of DFA
- Problems related to Minimization of DFA
- Regular Languages : Equivalence of Finite Automata and Regular Languages
- Equivalence of Finite Automata and Regular Grammars
- Problems related to Equivalence of Finite Automata and Regular Languages and Regular Grammars
- Variants of Finite Automata :Two-way Finite Automaton Mealy Machines
- Properties of Regular Languages: Closure Properties
- Set Theoretic Properties & Other Properties
- Pumping Lemma

Introduction of Automaton

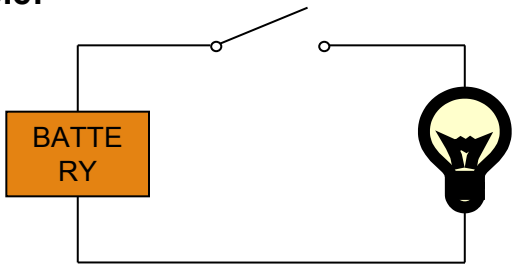
An automata is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

Automata-based programming is a programming paradigm in which the program or its part, is thought as a model of a finite state machine or any other formal automation.

What is Automata Theory?

- Automata theory is the study of abstract computational devices
- Abstract devices are (simplified) models of real computations
- Computations happen everywhere: On your laptop, on your cell phone, in nature, …

Example:



input: switch

output: light bulb

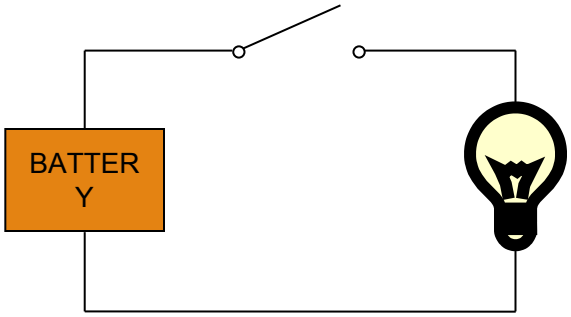
actions: flip switch

states: on, off

1. Electric Switch
2. Fan Regulator
3. Automatic door controller
4. Binary string with divisible by 4

Simple Computer

Example:



input : switch

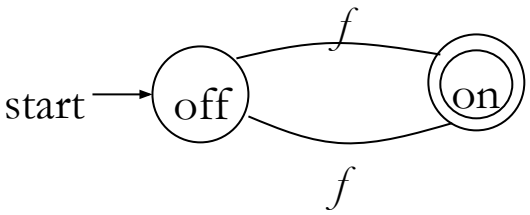
Output : light bulb

Actions : flip switch

States : on, off

OTHER EXAMPLES

1. Electric Switch
2. Fan Regulator
3. Automatic door controller
4. Binary string with divisible by 4



bulb is on if and only if there was an odd number of flips

Types of Automata

finite automata	Devices with a finite amount of memory. Used to model “small” computers.
push-down automata	Devices with finite memory that can be accessed in a restricted way. Used to model parsers, etc.
Turing Machines	Devices with infinite memory. Used to model any computer.

Introduction of Formal Proof - Basic Definitions

1. **Alphabet** - a finite set of symbols.
   - Notation: Σ .

   Examples: Binary alphabet {0,1},
   English alphabet {a,...,z,!,?,...}

2. **String over an alphabet** Σ - a finite sequence of symbols from Σ.
   - Notation: (a) Letters u, v, w, x, y, and z denote strings.
     (b) Convention: concatenate the symbols. No parentheses or commas used.
   - Examples: 0000 is a string over the binary alphabet.
     a!? is a string over the English alphabet.

3. **Empty string**: e or ε denotes the empty sequence of symbols.

4. **Language over alphabet** Σ - a set of strings over Σ.
   - Notation: L.
   - Examples:
     - {0, 00, 000, ...} is an "infinite" language over the binary alphabet.
     - {a, b, c} is a "finite" language over the English alphabet.

5. **Empty language** - empty set of strings. Notation: Φ.

6. **Binary operation on strings**: Concatenation of two strings u.v - concatenate the symbols of u and v.
   - Notation: uv
   - Examples:
     - 00.11 = 0011.
     - ε.u = u.ε = u for every u. (identity for concatenation)

Binary relations on strings

1. **Prefix** - u is a prefix of v if there is a w such that v = uw.
   - Examples:
     - ε is a prefix of 0 since 0 = ε0
     - apple is a prefix of appleton since appleton = apple.ton
2. **Suffix** - u is a suffix of v if there is a w such that v = wu.
   - Examples:
     - 0 is a suffix of 0 since 0 = ?
     - ton is a suffix of appleton since ?

3. **Substring** - u is a substring of v if there are x and y such that v = xuy.
   - Examples:
     - let is a substring of appleton since appleton = app.let.on
     - 0 is a substring of 0 since 0 = epsilon.0.epsilon

Observe that prefix and suffix are special cases

of substring.

Applications of the Concepts AND New Developments

Applications include:

- Text processing (editors, etc.)
- Compiler design
- Verification through model checking

Development's include :

DNA Computing

Quantum Turing Machines

Introduction to Mathematical concepts

Inductive Proof

Deductive Proof

INDUCTIVE PROOF

It is a special form of proof that deals about the objects in recursion.

Induction Principle:

If we prove S(i) and we prove that for all n>=I, S(n)=>S(n+1), then we may conclude that s(n) for all n>=1.

It has two parts.

1.Basis part.

2.Inductive part.

Basis step: We have to show S(i) for a particular integer value “i”, here “i “ may be zero or one.

Inductive step: We assume n>=i, where “i “ is the basis integer and we have to show that if S(n) then S(n+1) i.e., S(n) =>S(n+1).

Problems

Prove by an induction method on “n”

$i=0 \sum n\ i2 =((n(n+1)(2n+1))/6)$.

Solution:

$i=0 \sum n\ i2 =((n(n+1)(2n+1))/6)$. ----> Equ 1

Induction Principle:

If we prove S(i) and we prove that for all n>=I, S(n)=>S(n+1), then we may conclude that s(n) for all n>=1.

Basis:

Put n=0 in the equ 1

RHS= (0 (0+1) (2*0+1))/6 => 0

LHS = i=0 ∑ n (02) => 0

i.e., LHS = RHS.

Inductive step:

Put n =1 in the equ 1

S(n) = (n(n+1)(2n+1))/6.

= (2n3+2n2+n2+n)/6. ---> Equ 2

Put n=n+1 in the equ 1

S(n+1) => ((n+1)(n+2)(2n+3))/6.

=> ((n2+3n+2) (2n+3))/6.

=> (2n3+6n2+4n+3n2+9n+6)/6.->equ 3

From the induction principle:

= S(n+1) =S(n) + (n+1)2.

= ((2n3+3n2+n)/6) + (n+1)2

$= ((2n^3+3n^2+n)/6) + (n^2+2n+1)$

Cross multiplying the ‘6’ on the numerator.

$= ((2n^3+3n^2+n + 6n^2+12n+6))/6.$

$= ((2n^3+9n^2+13n+6))/6.$ --□ equ 4

Equ 4 satisfies equ 3

Hence proved.

Deductive proof:

Consists of sequence of statements whose truth lead us from some initial statement called the hypothesis or the give statement to a conclusion statement.

Deductive Principle:

The theorem is proved when we go from a hypothesis H to a conclusion C is the atatement " if H then C". i.e C is deduced from H

Additional forms of proof:

Proof of sets

Proof by contradiction

Proof by counter example

Direct proof (AKA) Constructive proof: If p is true then q is true

Eg: if a and b are odd numbers then product is also an odd number. Odd number can be represented as 2n+1

a=2x+1, b=2y+1 product of a X b = (2x+1) X (2y+1)= 2(2xy+x+y)+1 = 2z+1 (odd number)

Proof by Contrapositve

The Contrapositive of the statement " ifH then C" is "if not C then not H".A statement and its contrapositive are either both true or both false.

Theorem R U (S ∩ T) =(R U S) ∩ (R U T)

	Statement	Justification
1.	x is in $R \cup (S \cap T)$	Given
2.	x is in R or x is in $S \cap T$	(1) and definition of union
3.	x is in R or x is in both S and T	(2) and definition of intersection
4.	x is in $R \cup S$	(3) and definition of union
5.	x is in $R \cup T$	(3) and definition of union
6.	x is in $(R \cup S) \cap (R \cup T)$	(4), (5), and definition of intersection

	Statement	Justification
1.	x is in $(R \cup S) \cap (R \cup T)$	Given
2.	x is in $R \cup S$	(1) and definition of intersection
3.	x is in $R \cup T$	(1) and definition of intersection
4.	x is in R or x is in both S and T	(2), (3), and reasoning about unions
5.	x is in R or x is in $S \cap T$	(4) and definition of intersection
6.	x is in $R \cup (S \cap T)$	(5) and definition of union

Proof by Contradiction:

H and not C implies falsehood.

Regular Expressions

In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as

$$(5 + 3) \times 4 .$$

Similarly, we can use the regular operations to build up expressions describing languages, which are called *regular expressions*. An example is:

$$(0 \cup 1)0^* .$$

Regular Expressions

- Regular expressions describe regular languages

- Example:

$$(a + b \cdot c)^*$$

describes the language

$$\{a, bc\}^* = \{\in, a, bc, aa, abc, bca, \ldots\}$$

Recursive Definition

Given regular expressions r_1 and r_2

$r_1 + r_2$

$r_1 \cdot r_2$

$r_1 *$

(r_1)

Are regular Expressions

Say that R is a *regular expression* if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

Regular Expressions

A regular expression: $(a + b \cdot c)^* \cdot (c + \varnothing)$

Not a regular expression: $(a + b +)$

Regular Expressions

$L(r)$: language of regular expression r

Example

$$L((a+b\cdot c)^*)=\{\in,a,bc,aa,abc,bca,...\}$$

$$L(\varnothing)=\varnothing$$

$$L(\in)=\{\in\}$$

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

Regular Expressions

- For regular expressions r_1 and r_2
- $$L(r_1 + r_2) = L(r_1) \cup L(r_2)$$

$$L(r_1 \cdot r_2) = L(r_1)\, L(r_2)$$

$$L(r_1 *) = (L(r_1))*$$

$$L((r_1)) = L(r_1)$$

Regular Expressions

- Regular expression: $(a+b)\cdot a*$

$$
\begin{aligned}
L((a+b)\cdot a*) &= L((a+b))\,L(a*) \\
&= L(a+b)\,L(a*) \\
&= (L(a)\cup L(b))(L(a))* \\
&= (\{a\}\cup\{b\})(\{a\})* \\
&= \{a,b\}\{\in,a,aa,aaa,...\} \\
&= \{a,aa,aaa,...,b,ba,baa,...\}
\end{aligned}
$$

Regular Expressions

- Regular expression $r = (a+b)^*(a+bb)$

$$L(r) = \{a, bb, aa, abb, ba, bbb, \ldots\}$$

- Regular expression $r = (0+1)^*00(0+1)^*$

$L(r)$ = { all strings containing substring 00 }

Theorem

$$\left\{\begin{array}{l}\text{Languages}\\ \text{Generated by}\\ \text{Regular Expressions}\end{array}\right\} = \left\{\begin{array}{l}\text{Regular}\\ \text{Languages}\end{array}\right\}$$

Example Problems in Regular Expressions

1. $0^*10^* = \{w \mid w \text{ contains a single } 1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string } 001 \text{ as a substring}\}$.
4. $1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$.[5]
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of } 3\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^*.
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$.
    Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.

Solve Problems in Regular Expressions

1. $(0 \cup 1)^*$
2. $(0 \cup 1)(0 \cup 1)^*$
3. $0(0 \cup 1)^*0$
4. $(0 \cup 1)^*0(0 \cup 1)(0 \cup 1)(0 \cup 1)$
5. $0^*10^*10^*10^*$

Deterministic Finite Automata (DFA)

A DFA is a quintuple

$$A = (Q, \Sigma, \delta, q_0, F)$$

- Q is a finite set of *states*
- Σ is a *finite alphabet* (=input symbols)
- δ is a *transition function* $(q, a) \longmapsto p$
- $q_0 \in Q$ is the *start state*
- $F \subseteq Q$ is a set of *final states*

Formal definition of DFA

DFA $M = (Q, \Sigma, \delta, s, F)$

Where,

- Q is finite set of states
- Σ is input alphabet
- $s \in Q$ is initial state
- $F \subseteq Q$ is set of final states
- $\delta: Q \times \Sigma \rightarrow Q$

DFA

The DFA has:

- a finite set of states
- 1 special state - initial state
- 0 or more special states - final states
- input alphabet
- transition table containing

  (state, symbol) -> next state

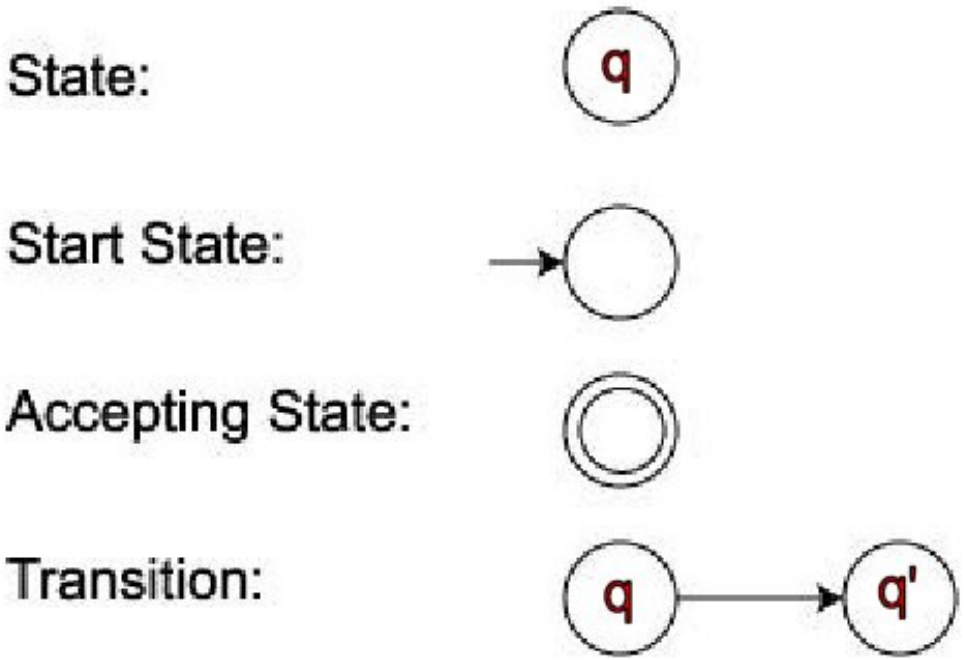
How does a DFA work?

After reading input string,

- if DFA state final, input accepted
- if DFA state not final, input rejected

Language of DFA -- set of all strings accepted by DFA.

Pictorial representation of DFA



$(q,\sigma) \rightarrow q'$

Formal definition of FA

1. Finite set of *states*, typically Q.
2. Alphabet of *input symbols*, typically Σ.
3. One state is the *start/initial* state, typically q_0.
4. Zero or more *final/accepting* states; the set is typically F.

5. A *transition function*, typically δ. This function:

   - Takes a state and input symbol as arguments.
   - Returns a state.
   - One "rule" of δ would be written $\delta(q, a) = p$, where q and p are states, and a is an input symbol.
   - Intuitively: if the FA is in state q, and input a is received, then the FA goes to state p (note: $q = p$ OK).

- A FA is represented as the five-tuple: $A = (Q, \Sigma, \delta, q_0, F)$.

Transition Diagram

Transition Diagram

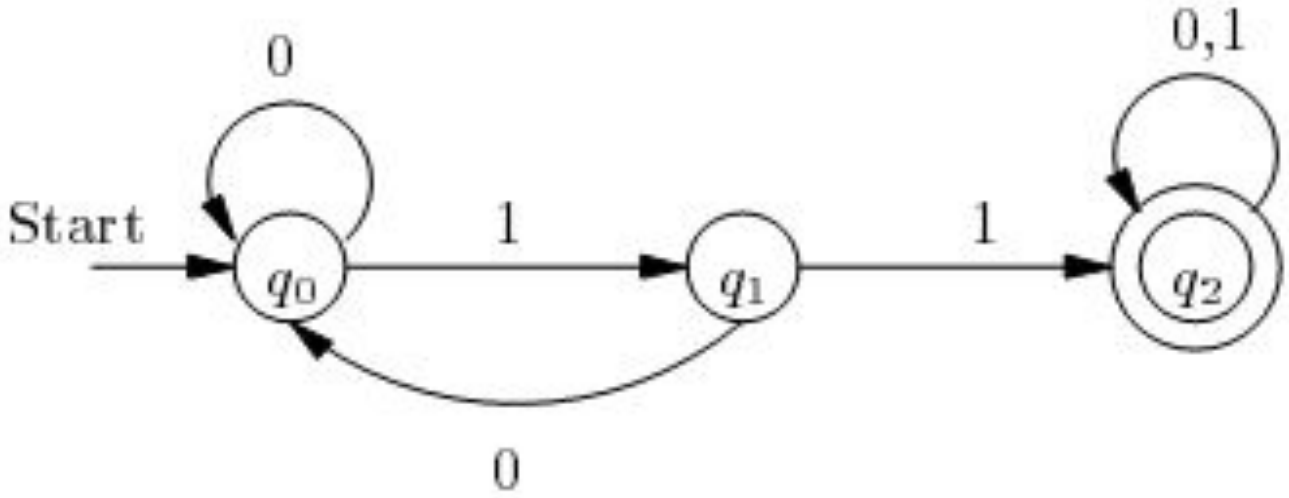
A FA can be represented by a graph; nodes = states; arc from q to p is labeled by the set of input symbols a such that $\delta(q, a) = p$.

- No arc if no such a.
- Start state indicated by word "start" and an arrow.
- Accepting states get double circles.

Example

Example

For the clamping FA:



Transition table

A transition table is a conventional tabular representation of functions like δ that takes the two argument and returns another state.

The row of the table corresponding to the states.

The column of the table corresponding to the inputs.

Example

Example: An automaton A that accepts

$L = \{x01y : x, y \in \{0,1\}^*\}$

The automaton $A = (\{q_0, q_1, q_2\}, \{0,1\}, \delta, q_0, \{q_1\})$ as a *transition table*:

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

Extending the Transition functions to the strings

Extension of δ to Paths

Intuitively, a FA *accepts* a string $w = a_1 a_2 \cdots a_n$ if there is a path in the transition diagram that:

1. Begins at the start state,
2. Ends at an accepting states, and
3. Has sequence of labels $a_1, a_2, \ldots, a_n$.

Formally, we extend transition function δ to $\hat{\delta}(q, w)$, where w can be any string of input symbols:

- Basis: $\hat{\delta}(q, \epsilon) = q$ (i.e., on no input, the FA doesn't go anywhere.
- Induction: $\hat{\delta}(q, wa) = \delta\big(\hat{\delta}(q, w), a\big)$, where w is a string, and a a single symbol (i.e., see where the FA goes on w, then look for the transition on the last symbol from that state).
- Important fact with a straightforward, inductive proof: $\hat{\delta}$ really represents paths. That is, if $w = a_1 a_2 \cdots a_n$, and $\delta(p_i, a_i) = p_{i+1}$ for all $i = 0, 1, \ldots, n-1$, then $\hat{\delta}(p_0, w) = p_n$.

Language of DFA

Acceptance of Strings

A FA $A = (Q, \Sigma, \delta, q_0, F)$ accepts string w if $\hat{\delta}(q_0, w)$ is in F.

Language of a FA

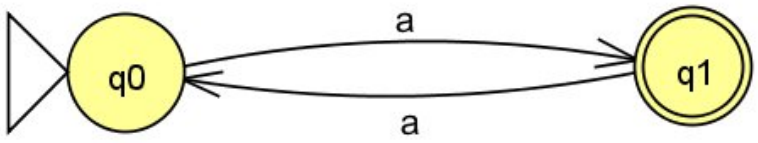
FA A accepts the language $L(A) = \{w \mid \hat{\delta}(q_0, w)$ is in $F\}$.

Example: Diagram of DFA

$L = \{a^{2n+1} \mid n >= 0\}$

Answer:

L = {a, aaa, aaaaa, ...}



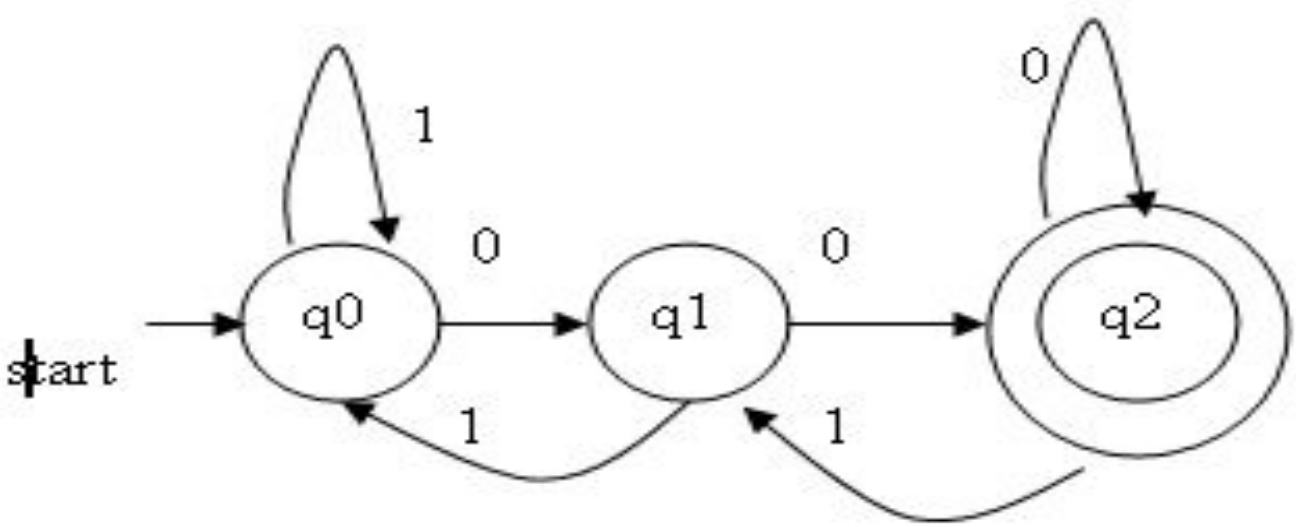
Problem 1

Given a DFA accepting the following language over the alphabet {0,1}

Set of all the strings ending with 00.

In this problem ,

"00" as 2 inputs and it needs 3 states



Formal definition of L(M)

L(M) - Language accepted by M

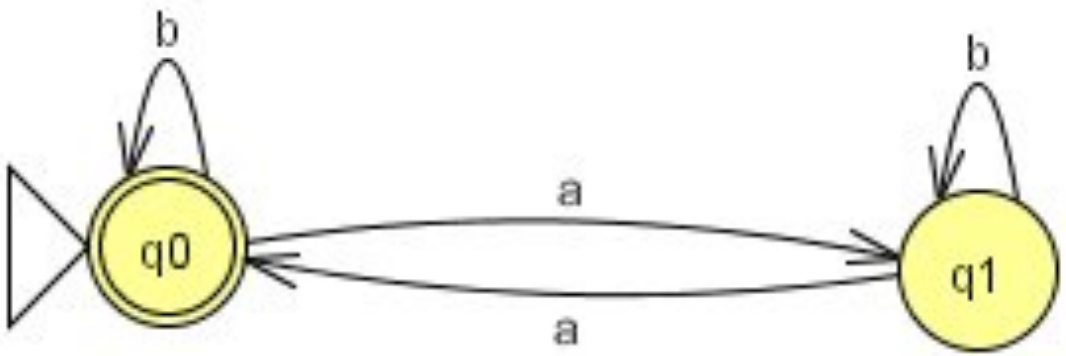
Define δ^*:

- $\delta^*(q, \varepsilon) = q$
- $\delta^*(q, w\sigma) = \delta(\delta^*(q,w),\sigma)$

Definition: L(M) = { w in Σ^* | δ^* (s,w) in F }.

Problem 2

Example: L(M) = {w in {a,b}* | w contains even no. of a's}



Regular Languages

Definition: A Language is regular iff there is a DFA that accepts it.

Examples:

- ϕ
- $\{\varepsilon\}$
- Σ^*
- {w in {0,1}* | second symbol of w is a 1} Exercise
- {w in {0,1}* | second last symbol of w is a 1} Exercise
- {w in {0,1}* | w contains 010 as a substring} - (importance?)

Closure properties of Regular Languages

Regular languages are closed under:

- Union
  - Notation: $\cup$
- Intersection
  - Notation: $\cap$

$L_1 \cup L_2$ is regular if L_1 and L_2 are regular.

$L_1 \cap L_2$ is regular if L_1 and L_2 are regular.

Examples

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

Let L_1 = { w in Σ^* | w has even number of a's}.

- Is L_1 regular?

L_2 = { w in Σ^* | w has odd number of b's}.

- Is L_2 regular?

$L_1 \cup L_2$ = ?

- Ans: {w in Σ^* | w has even a's or odd b's}.

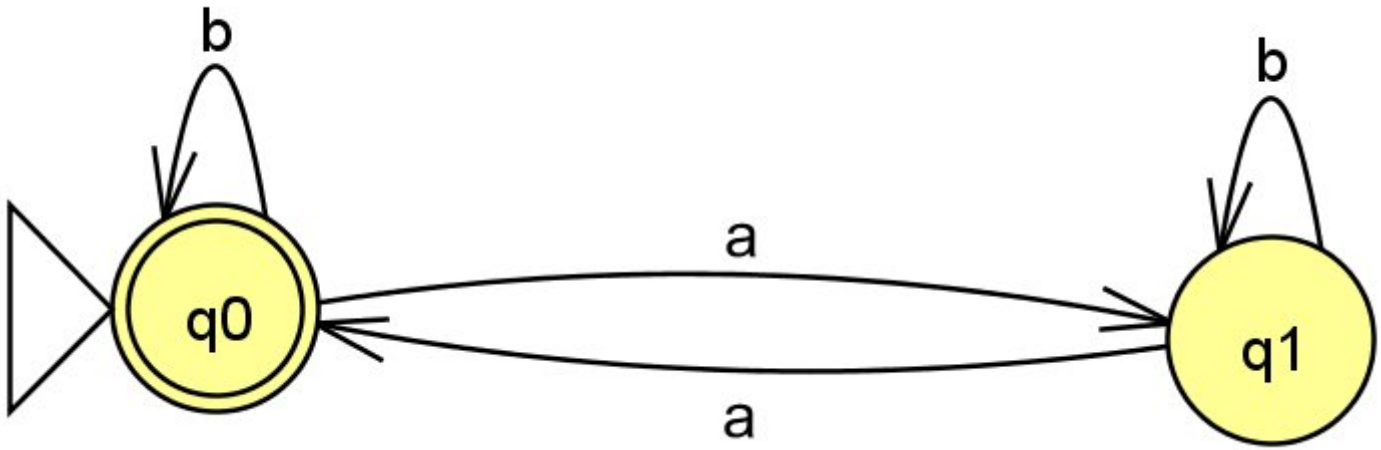
$L_1 \cap L_2$ = ?

- Ans: {w in Σ^* | w has even a's and odd b's}.

By closure properties, both these are regular.

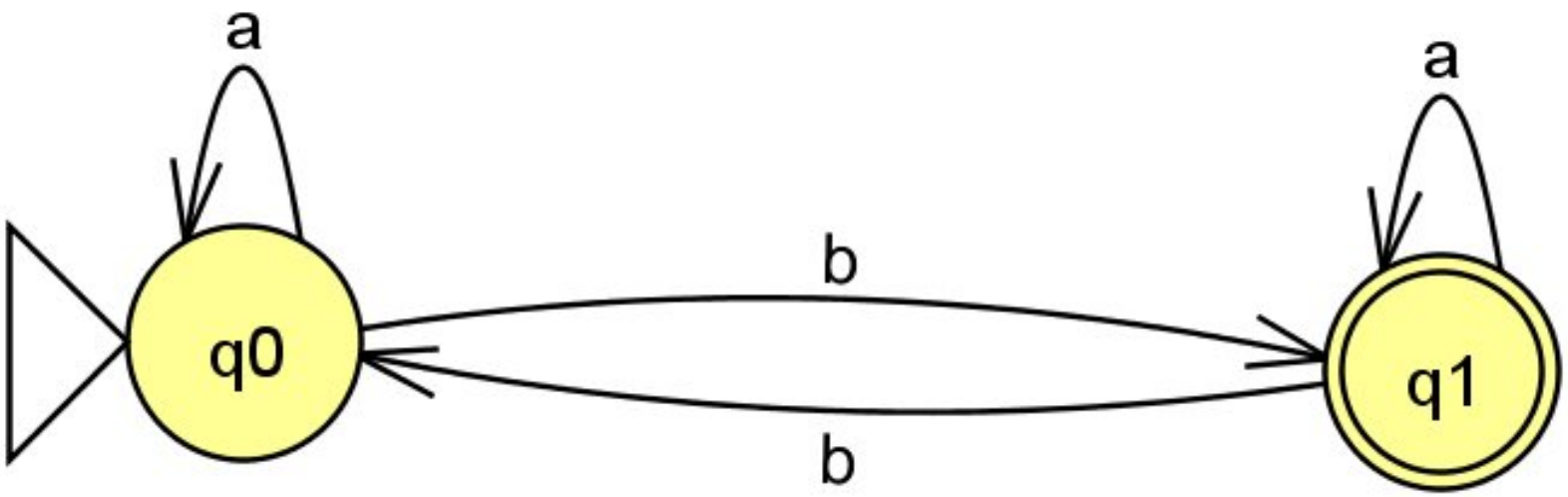
Problem 3

DFA of L_1 = {w in {a,b}* | w has even number of a's}.



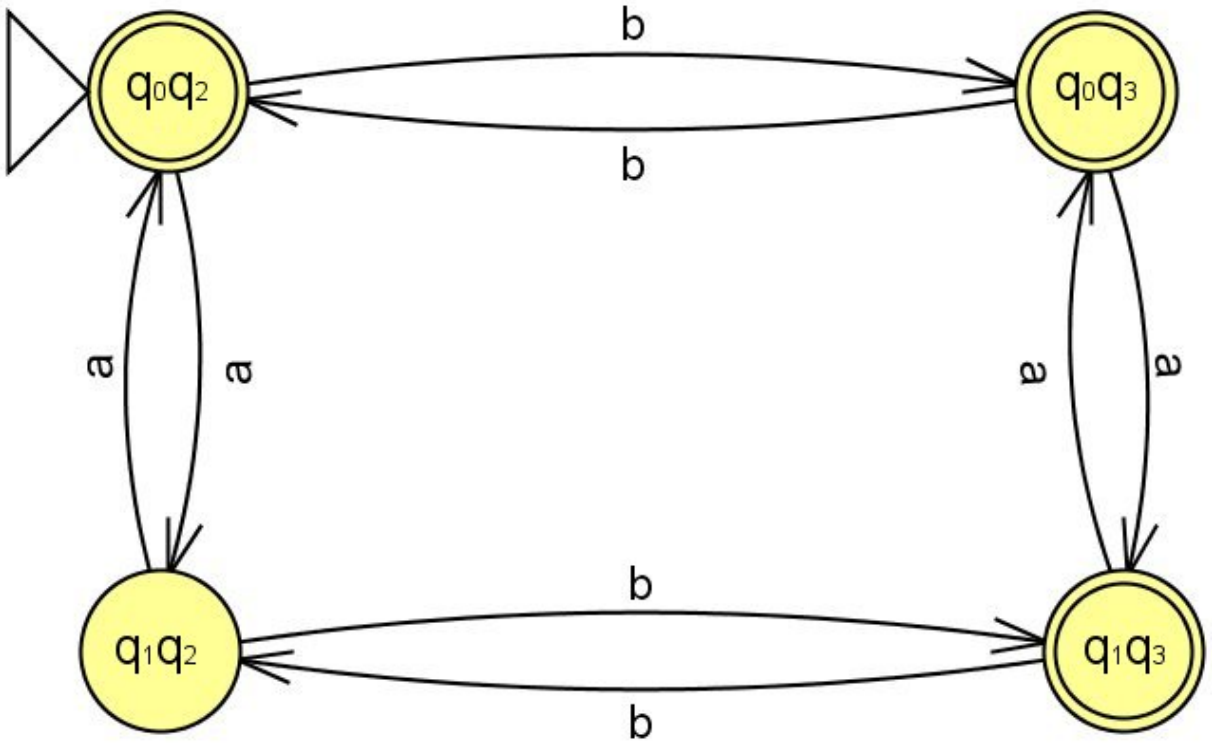
Problem 4

DFA of L_2 = { w in {a,b}* | w has odd number of b's}.



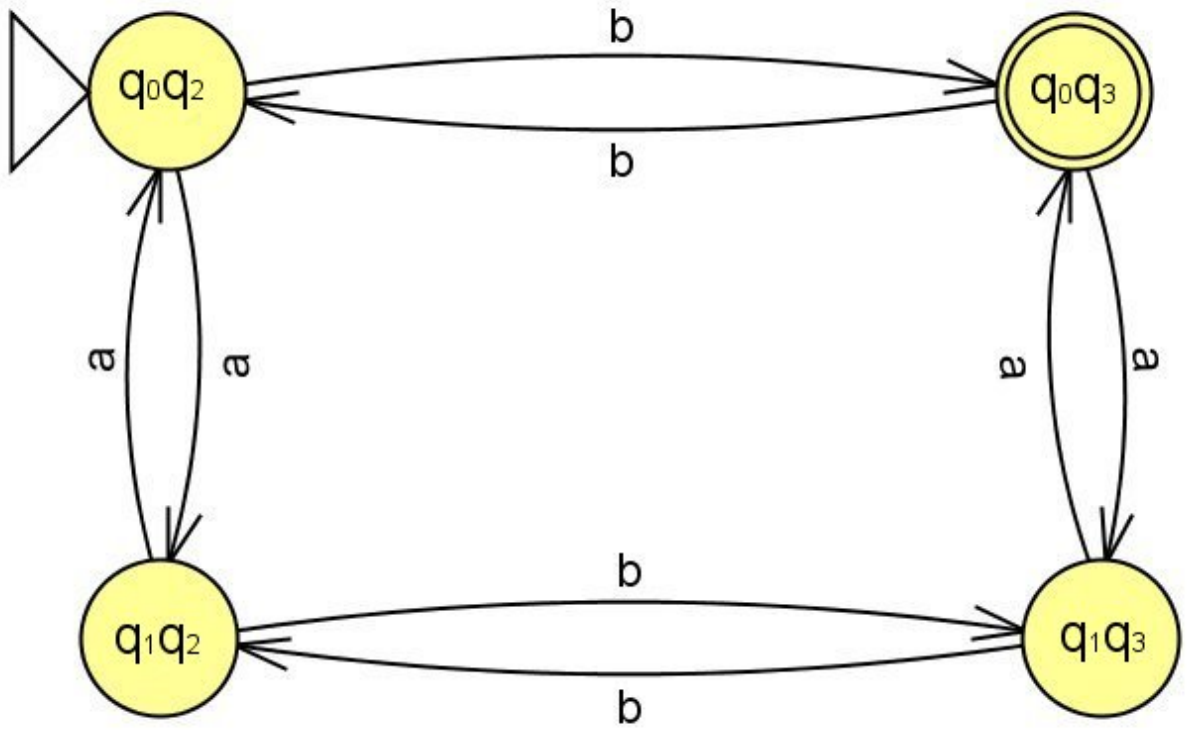
Problem 5

DFA of $L_1 \cup L_2$ = {w in {a,b}* | w has even a's or odd b's}.



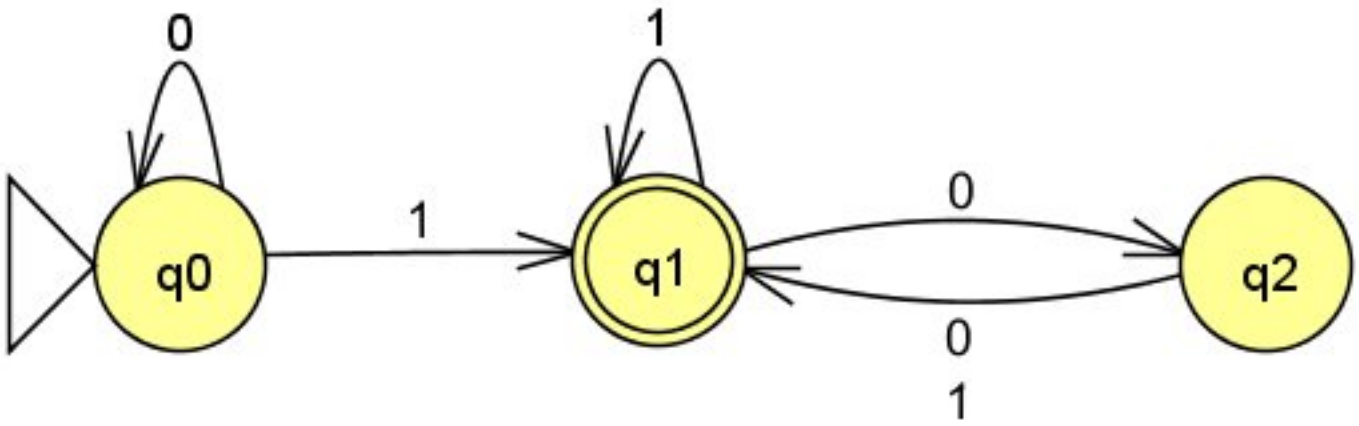
Problem 6

DFA of $L_1 \cap L_2$ = {w in {a,b}* | w has even a's and odd b's}.



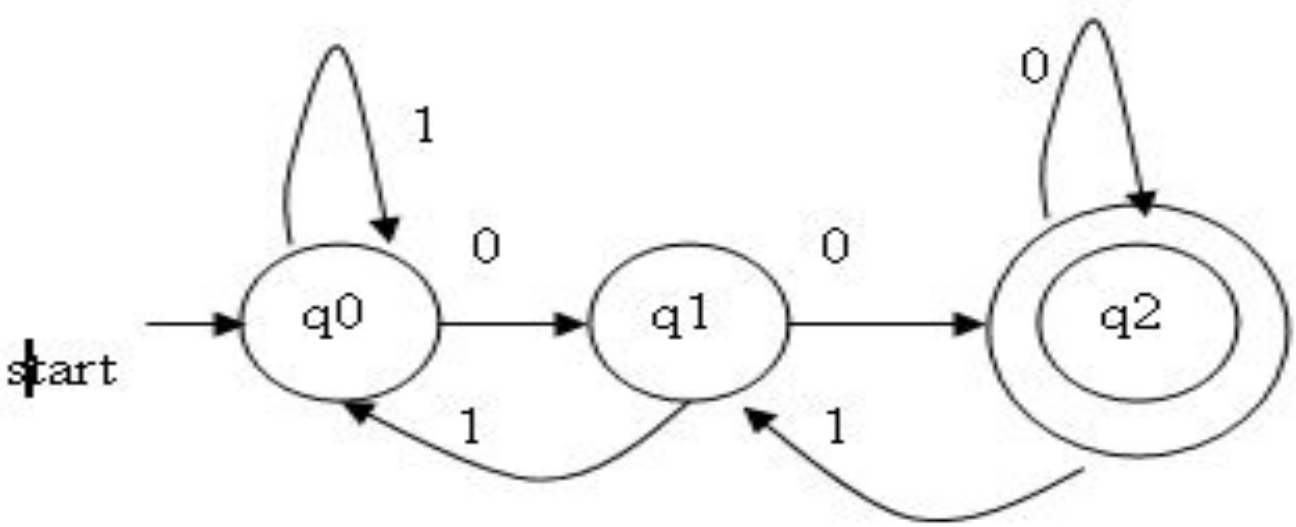
Problem 8

DFA of A = {w | w contains at least one 1 and an even number of 0s follow the last 1}



Problem 9

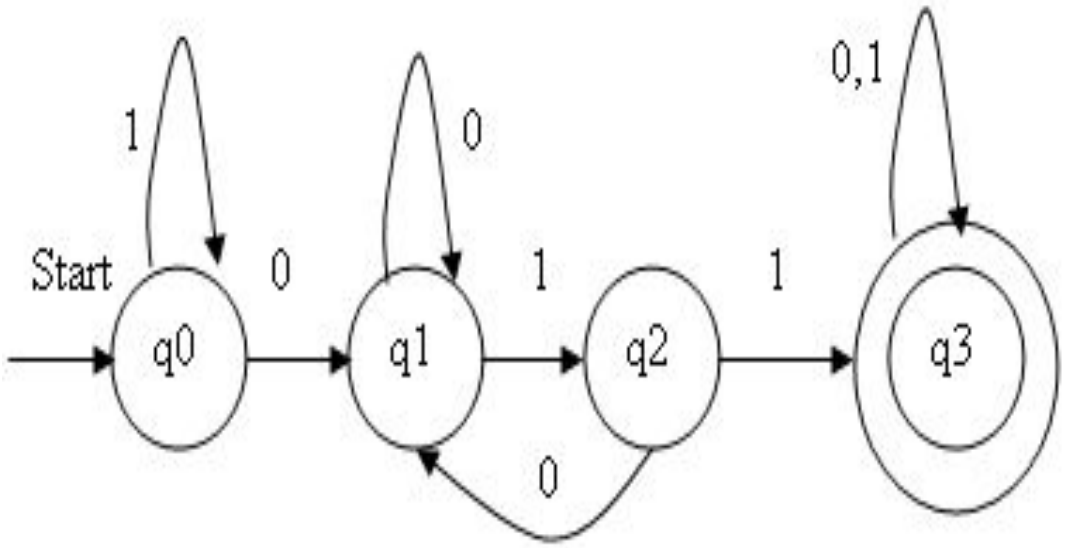
Given a DFA accepting the following language over the alphabet {0,1} Set of all the strings ending with 00.



Problem 10

Given a DFA accepting the following language over the alphabet {0,1}

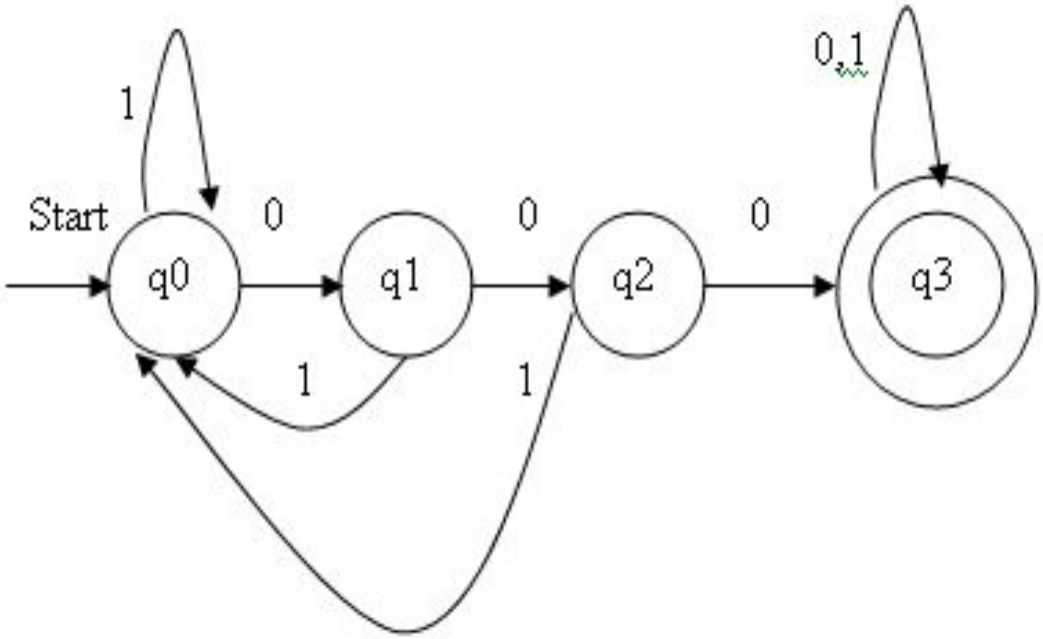
Set of all the strings with 3 consecutive 0's (not necessarily at the end).



Problem 11:

Given a DFA accepting the following language over the alphabet {0,1}

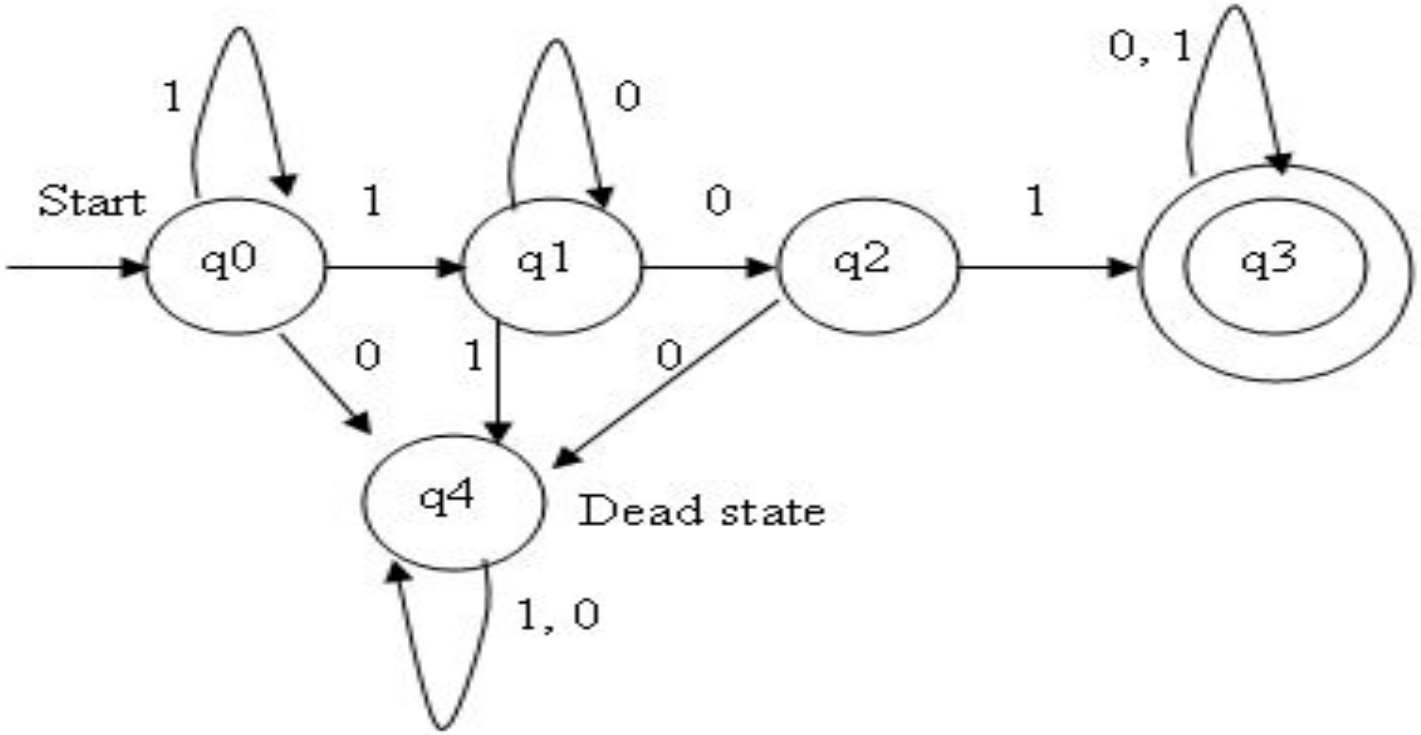
Set of all the strings with a 011 as a substring.



Problem 13

Given a DFA accepting the following language over the alphabet {0,1}

Set of all the strings beginning with a 101.



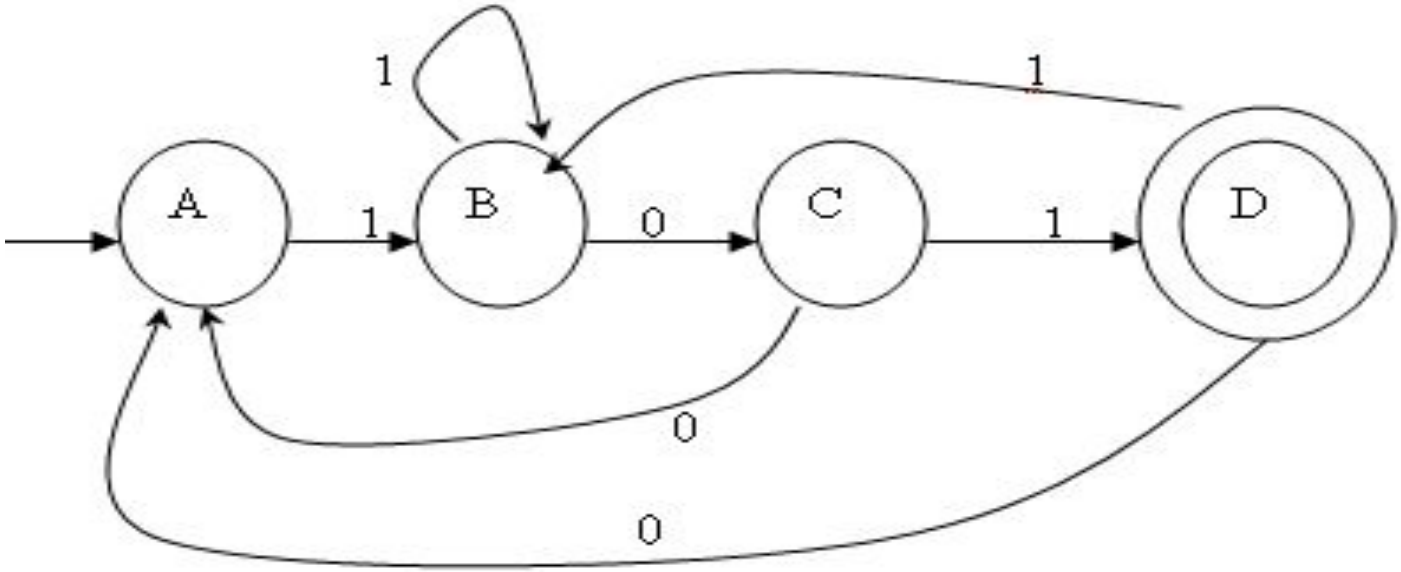
Problem 14

Given a DFA accepting the following language over the alphabet {0,1}

Set of all the strings with three consecutive zeros.

Problem 14

Consider the following transition diagram and to check the input string 110111,011101 is accepted or not.



Transition table:

Let A be a DFA which has A={Q,∑,δ,q0,F)

	0	1
→ A	A	B
B	C	B
C	A	D
* D	A	B

Input: 110111

δ(A,ϵ) =A.

δ(A,1) =B.

δ(A,11) =δ(A,1),1) =δ(B,1) =B.

δ(A,110) =δ(A,11),0) =δ(B,0) = C.

δ(A,1101) =δ(A,110),1) =δ(C,1) = D.

δ(A,11010) =δ(A,1101),0) =δ(D,1) = B.

δ(A,110101) =δ(A,11010),0) =δ(B,1) = B.

It does not reach the final state.

The given string is not accepted by DFA.

Input: 011101

δ(A,ϵ) =A.

δ(A,0) =A.

δ(A,01) =δ(A,0),1) =δ(A,1) =B.

δ(A,011) =δ(A,01),1) =δ(B,1) = B.

δ(A,0111) =δ(A,011),1) =δ(B,1) = B.

δ(A,01110) =δ(A,0111),0) =δ(B,0) = C.

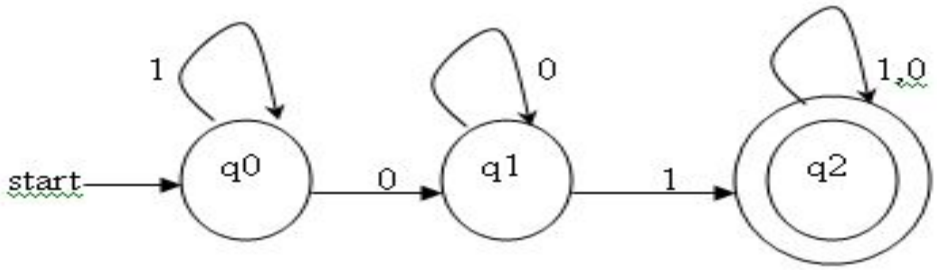
δ(A,011101) =δ(A,011101),0) =δ(C,1) = D.

It reaches the final state.

The given string is accepted by DFA.

Problem 15

Give a DFA accepting the following language over the alphabet {0,1}.
All the strings with a substring 01 and to check the given input string 10010 is accepted or not by a DFA.



	0	1
q0	q1	q0
q1	q1	q2
*q2	q2	q2

Input 10010

δ(q0,ϵ) =q0.

δ(q0,1) =q0.

δ(q0,10) =δ(q0,1),0) =δ(q0,0) =q1.

δ(q0,100) =δ(q0,10),0) =δ(q1,0) =q1.

δ(q0,1001) =δ(q0,100),1) =δ(q1,1) =q2.

δ(q0,10010) =δ(q0,1001),0) =δ(q2,0) =q2.

It reaches the final state

The given string is accepted by DFA.

Problem16 :

Design a DFA to accept the language
L ={w/w has an even number of 0's and even number of 1's}and to check the given input strings are 110101 and 10010 is accepted or not by an DFA.

It has the 4 states like q0,q1,q2,q3.

q0: Number of 1's and number of 0's are even.

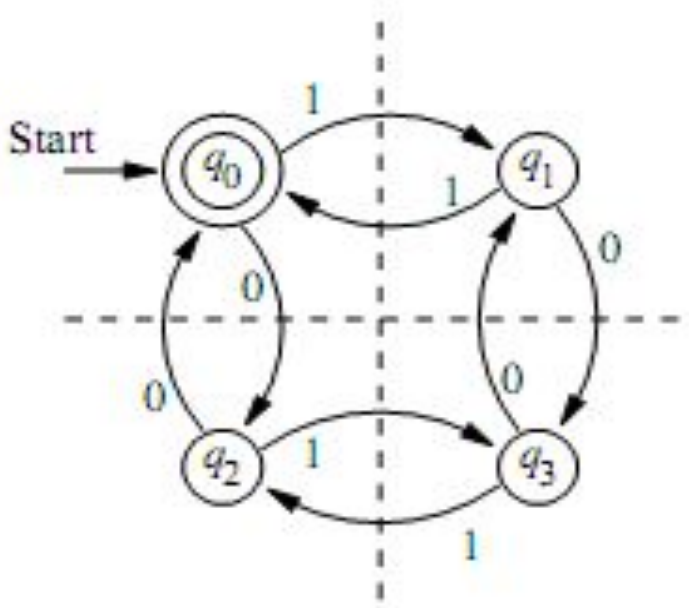
q1: Number of 0's is even and number of 1's is odd.

q2: Number of 0's is odd and number of 1's is even.

q3: Number of 0's and number of 1's are odd.

Let A be a DFA which has A={Q,∑,δ,q0,F)

Transition diagram:



	0	1
⋆ → q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Input: 110101

$\delta(q0,\epsilon) = q0.$

$\delta(q0,1) = q1.$

$\delta(q0,11) = \delta(q0,1),1) = \delta(q1,1) = q0.$

$\delta(q0,110) = \delta(q0,11),0) = \delta(q0,0) = q2.$

$\delta(q0,1101) = \delta(q0,110),1) = \delta(q2,1) = q3.$

$\delta(q0,11010) = \delta(q0,1101),0) = \delta(q3,0) = q1.$

$\delta(q0,110101) = \delta(q0,11010),0) = \delta(q1,1) = q0.$

It reaches the final state. The given string is accepted by DFA.

Input: 10010

δ(q0,ϵ) =q0.

δ(q0,1) =q1.

δ(q0,10) =δ(q0,1),0) =δ(q1,0) =q3.

δ(q0,100) =δ(q0,10),0) =δ(q3,0) =q1.

δ(q0,1001) =δ(q0,100),1) =δ(q1,1) =q0.

δ(q0,10010) =δ(q0,1001),0) =δ(q0,0) =q2.

The state q2 is not a accepting state.

The given string is not accepted by DFA.

Nondeterministic Finite Automaton (NFA)

Generalization of NDFA. Allows:

- 0 or more next states for the same (q, σ).
  - Guessing
- Transitions labeled by the empty string.
  - Changing state without reading input

Motivation: Flexibility.

- Easier to prove many closure properties.

How does an NFA work?

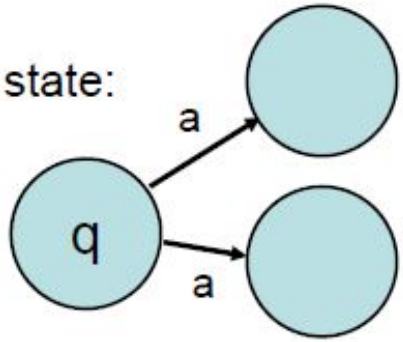
w is accepted by an NFA provided there is a sequence of guesses that leads to a final state.

Language accepted by NFA is the set of all strings accepted by it.

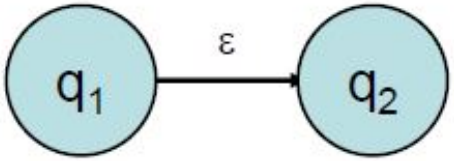
Nondeterministic Finite Automata:

- Generalize FAs by adding nondeterminism,allowing several alternative computations on the same input string.
- Ordinary deterministic FAs follow one path on each input.

- Two changes:
  - Allow $\delta(q, a)$ to specify more than one successor state:



  - Add ε-transitions, transitions made "for free", without "consuming" any input symbols.



- Formally, combine these changes:

Nondeterministic Finite Automata:

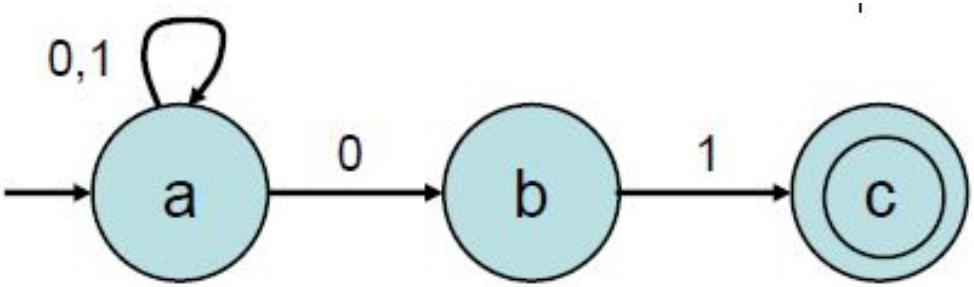
- An NFA is a 5-tuple (Q, Σ, δ, q_0, F), where:
  - Q is a finite set of states,
  - Σ is a finite set (alphabet) of input symbols,
  - δ: $Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the transition function,
  - $q_0 \in Q$, is the start state, and
  - $F \subseteq Q$ is the set of accepting, or final states.
- How many states in P(Q)?

  $2^{|Q|}$
- Example: Q = { a, b, c }

  P(Q) = { $\varnothing$, {a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,c} }

Nondeterministic Finite Automata:

NFA Example



$Q = \{ a, b, c \}$

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

$q_0 = a$

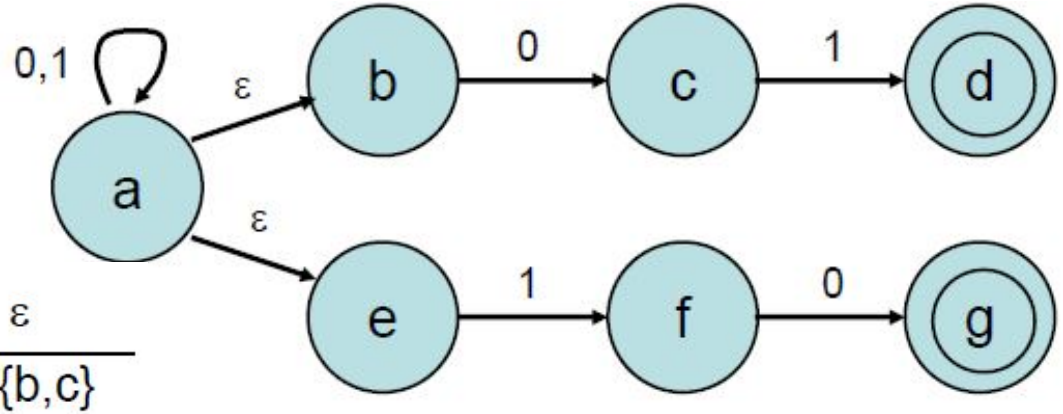
$F = \{ c \}$

δ:

	0	1	ε
a	{a,b}	{a}	∅
b	∅	{c}	∅
c	∅	∅	∅

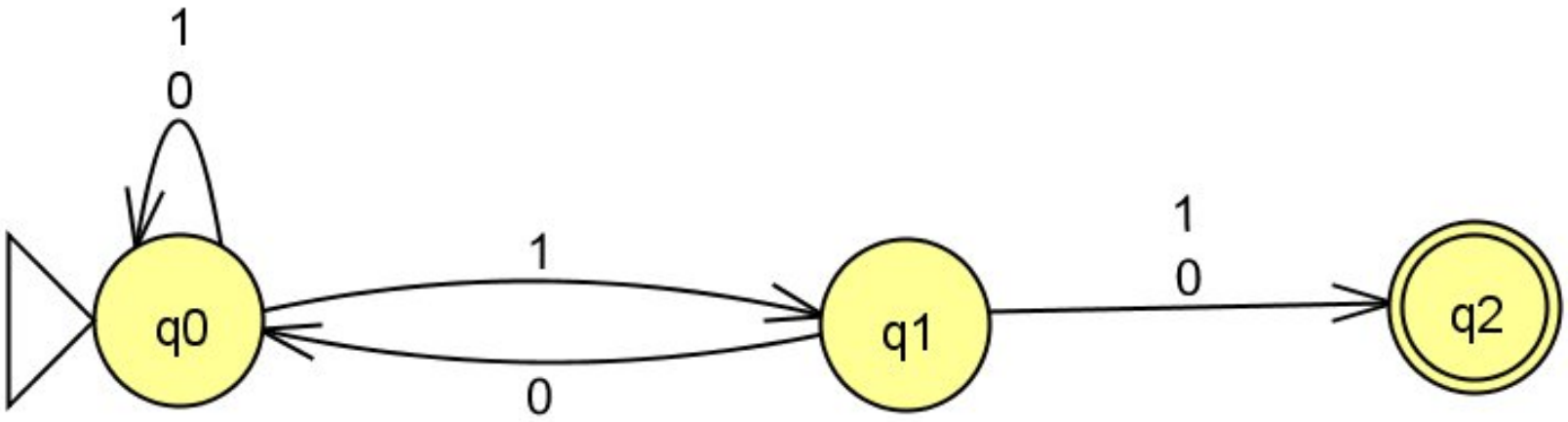
Nondeterministic Finite Automata:

NFA Example



	0	1	ε
a	{a}	{a}	{b,c}
b	{c}	∅	∅
c	∅	{d}	∅
d	∅	∅	∅
e	∅	{f}	∅
f	{g}	∅	∅
g	∅	∅	∅

Example: {w in {0,1}* | the second last symbol of w is a 1}



Formal definition of NFA

Notation: $\Sigma_e = \Sigma \cup \{e\}$.

NFA $M = (Q, \Sigma, \Delta, s, F)$ where:

- Q - finite set of states
- Σ - input alphabet
- s - initial state
- $F \subseteq Q$ - set of final states
- Δ is a subset of $Q \times \Sigma_e \times Q$.

If (p, u, q) in Δ, then NFA in state p can read u and go to q.

Formal definition of NFA acceptance

Define $\Delta^*(q, w)$ as a set of states: p ε $\Delta^*(q, w)$ if there is a directed path from q to p labeled w

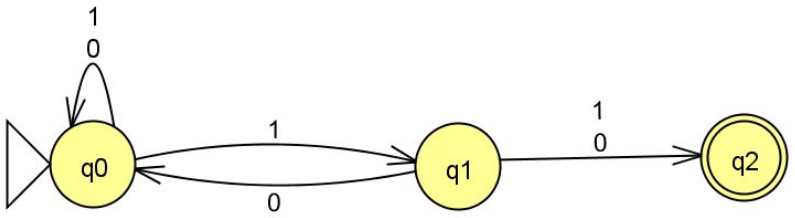
- Example: consider NFA of Lecture 3

$\Delta^*(q_0, 1) = ?$

- Ans: $\{q_0, q_1\}$

$\Delta^*(q_0, 11) = ?$

- Ans: $\{q_0, q_1, q_2\}$



NFA acceptance

---

w is accepted by NFA M iff $\Delta^*(q_0, w) \cap F$ is nonempty.

L(M) = {w in Σ* | w is accepted by M}.

NFA vs. DFA

Is NFA more powerful than DFA?

- Ans: No.

Theorem:

- For every NFA M there is an equivalent DFA M'

Proof Idea:

- NFA is in a set of states at any point during reading a string.
- DFA will use a lot of states to keep track of this.

Important Assumption:

- No transition labeled by epsilon.

(Will get rid of this assumption later.)

Equivalent DFA construction

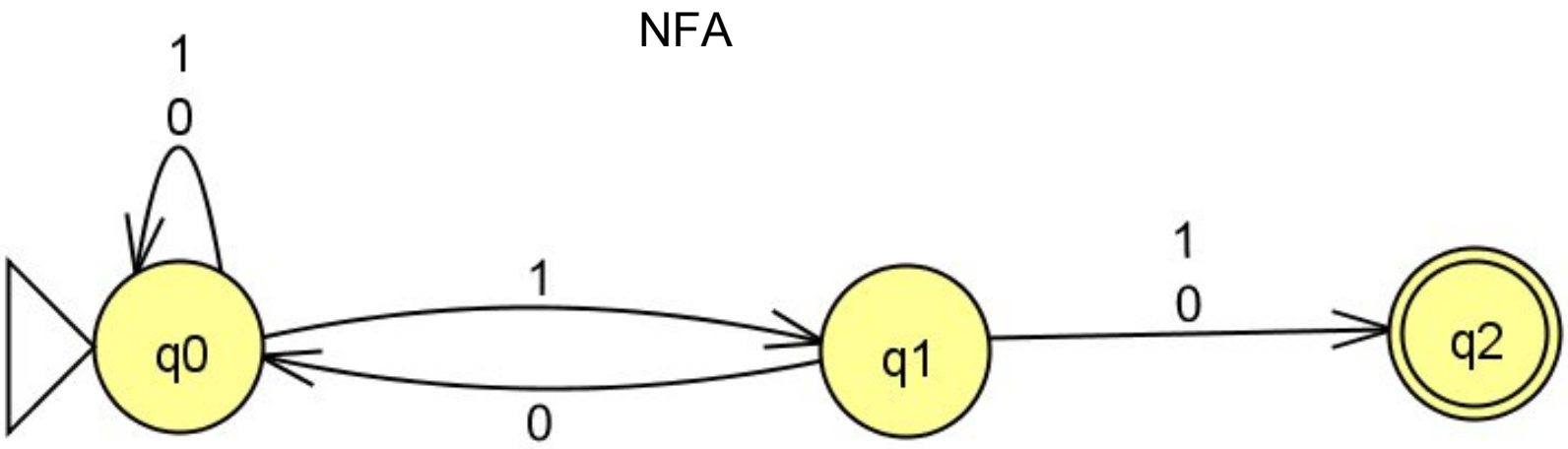
NFA $M = (Q, \Sigma, \Delta, s, F)$

DFA $M' = (Q', \Sigma, \delta, s', F')$ where:

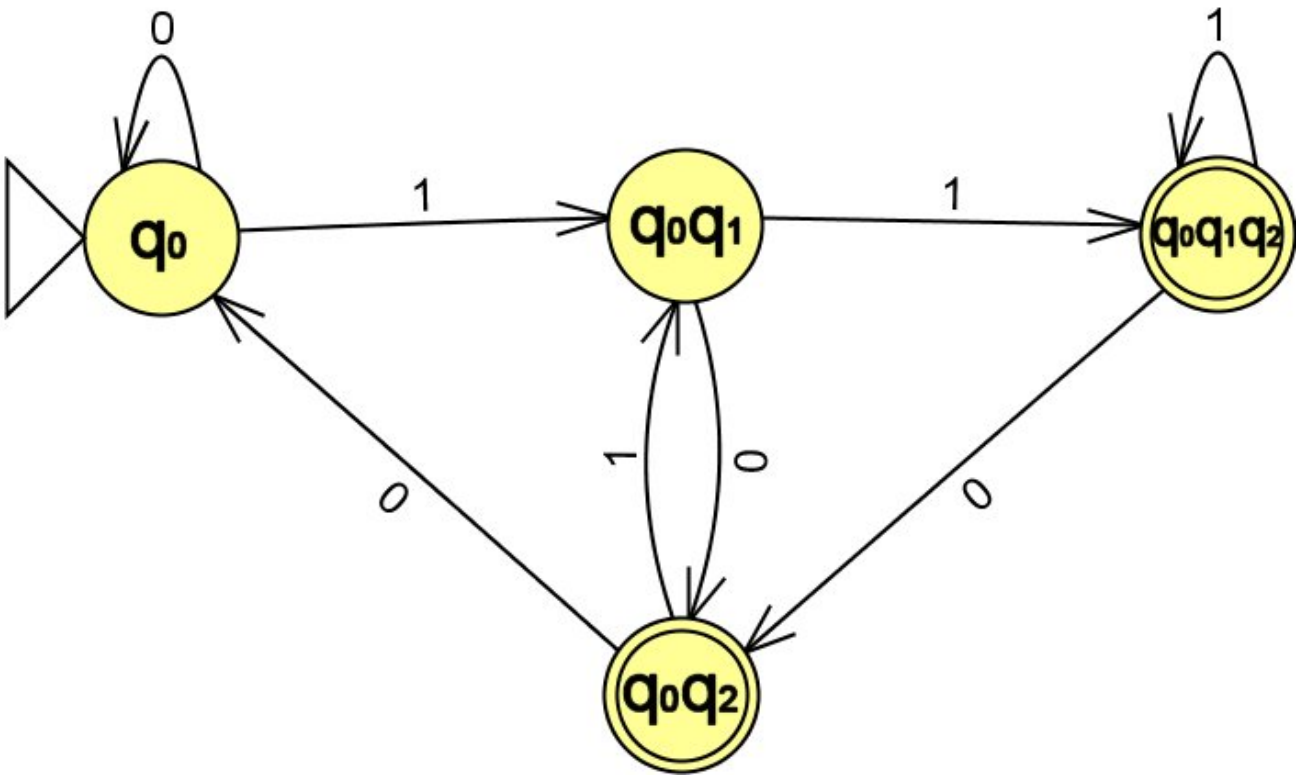
- $Q' = 2^Q$
- $s' = \{s\}$
- $F' = \{P \mid P \cap F \text{ is nonempty}\}$
- $\delta(\{p_1, p_2, p_m\}, \sigma) = \Delta^*(p_1, \sigma) \cup \Delta^*(p_2, \sigma) \cup \ldots \cup \Delta^*(p_m, \sigma)$

i.e. find all the states that can be reached on σ from all the NFA states in a DFA state.

Example: Equivalent DFA construction



Equivalent DFA construction

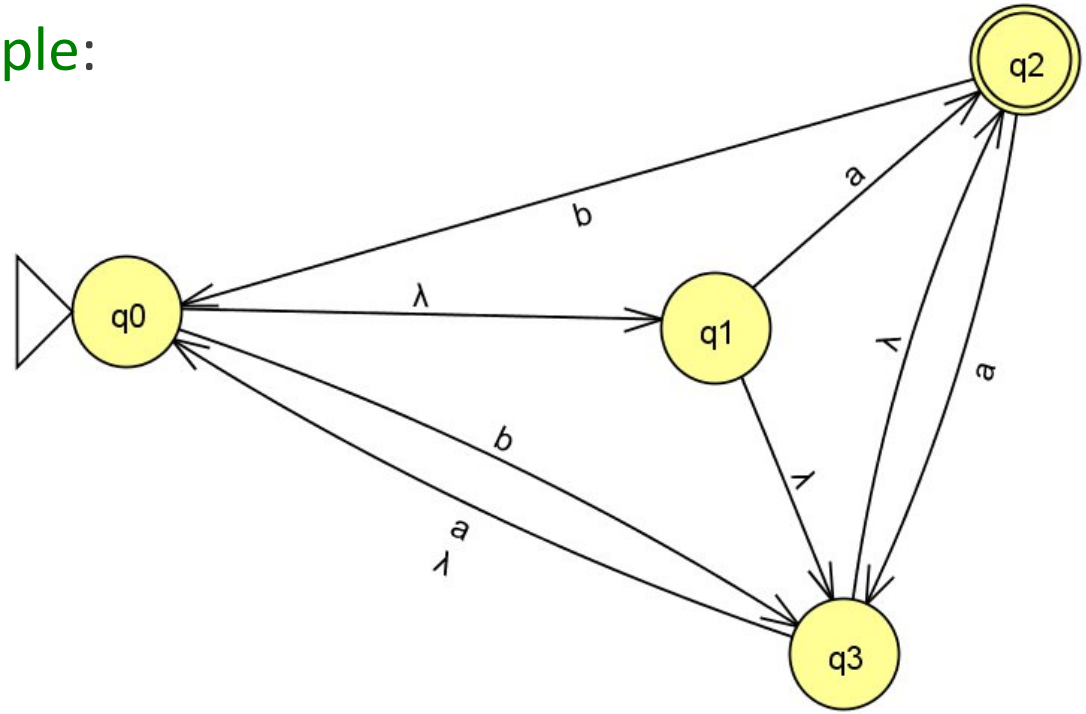


How to handle epsilon transitions?

Define e-closure of state q as Δ*(q, ε).

- notation: e-closure(q).

Example:



Handling epsilon transitions (contd.)

Extend e-closure to sets of states by:

- e-closure($\{s_1, \ldots, s_m\}$) = e-closure(s_1) $\cup$ ... $\cup$ e-closure(s_m)

Now let

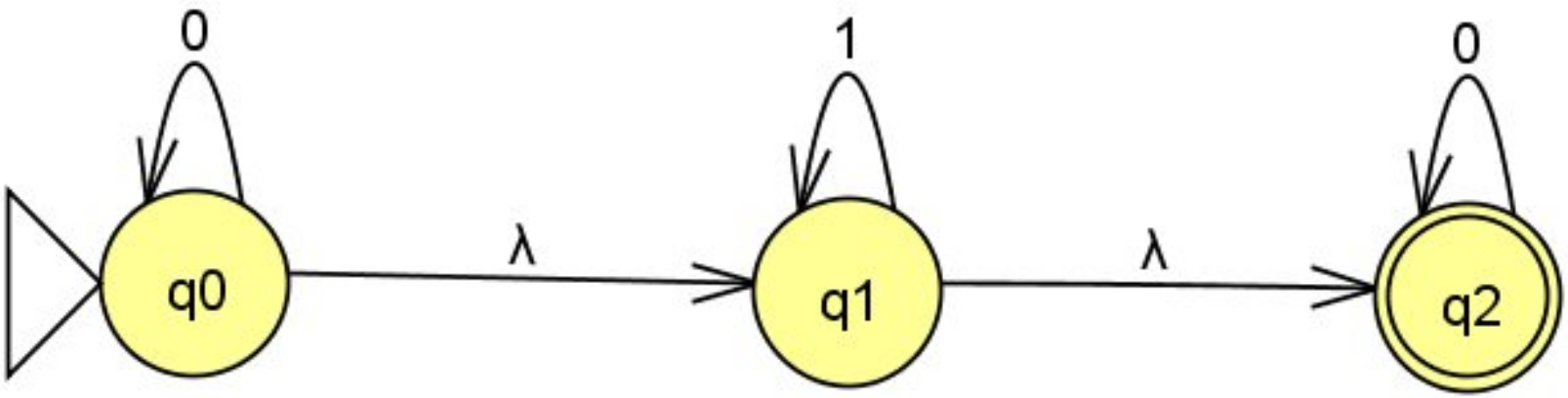
s' = e-closure({s}).

and,

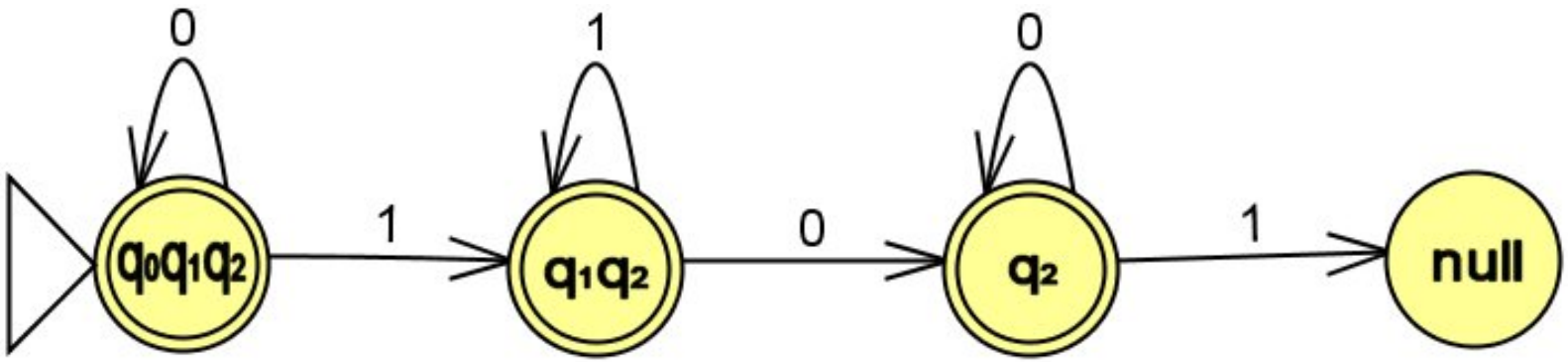
$\delta(\{p_1, \ldots, p_m\}, \sigma)$ = e-closure($\Delta^*(p_1, \sigma)$) $\cup$ ... $\cup$ e-closure($\Delta^*(p_m, \sigma)$)

to complete construction of DFA.

Example: Handling epsilon transitions.



DFA = ?



Language Operations

1. Concatenation. Notation: L·L' or just LL'
   - $L \cdot L' = \{uv \mid u \text{ in } L, v \text{ in } L'\}$.
2. Kleene Star. Notation: L*
   - $L^* = \{ w \text{ in } \Sigma^* \mid w = w_1 \ldots w_k \text{ for some } k >= 0 \text{ and each } w_i \text{ in } L\}$.

Examples: if $L = \{a^{(2n+1)} \mid n >= 0\}$. $L' = \{b^{(2n)} \mid n > = 0\}$.

LL' = ?

- Ans: $LL' = \{a^{(2n+1)} b^{(2m)} \mid n, m > = 0\}$

L* = ?

- Ans: $\{a^n \mid n >= 0\}$

U, ., * are called regular operations.

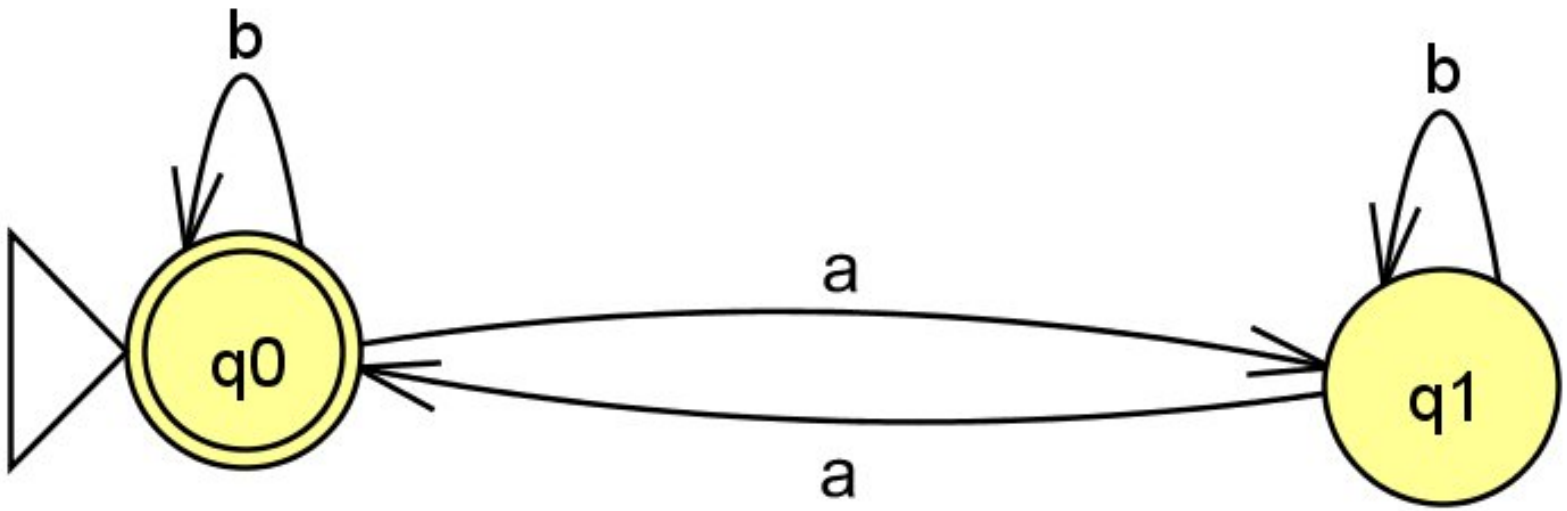
Closure properties of regular languages.

Previously we saw closure under $\cup$ and $\cap$.

New: Regular languages are closed under

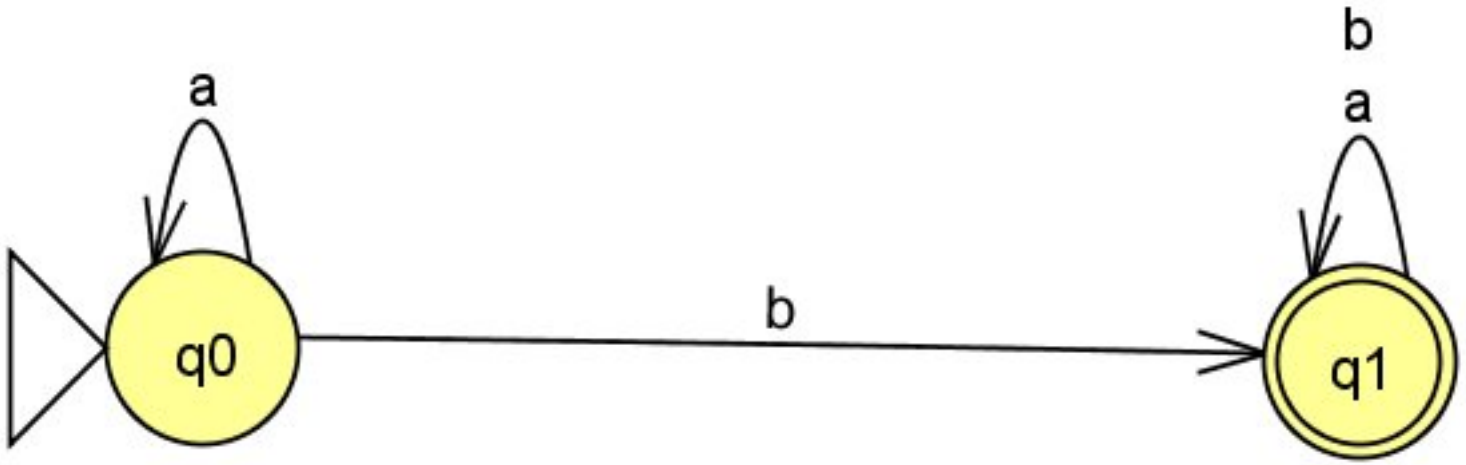
- Concatenation
- Kleene star
- Complement.

Examples



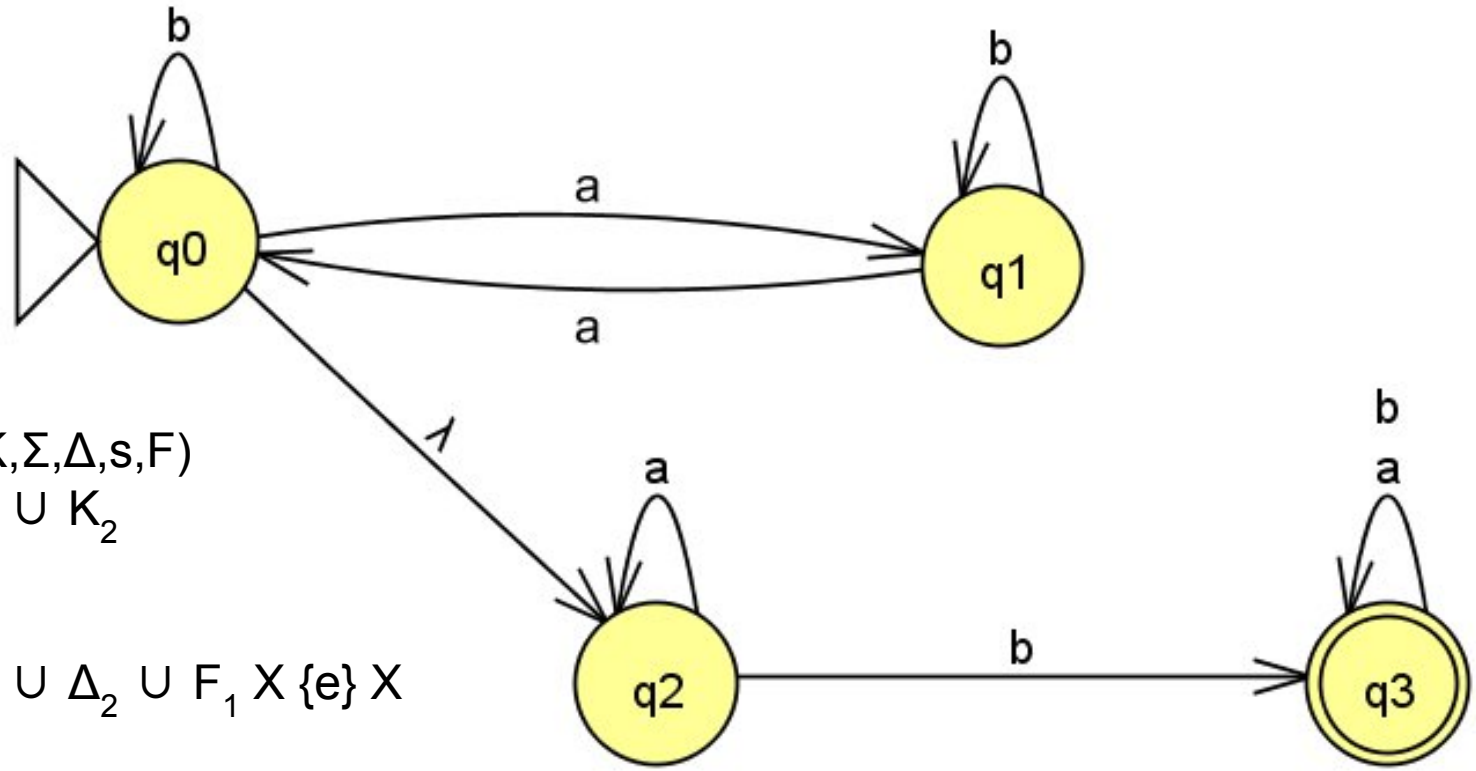
L = {w in {a,b}* | w has even a's }

Examples



L' = {w in {a,b}* | w has at least one b}

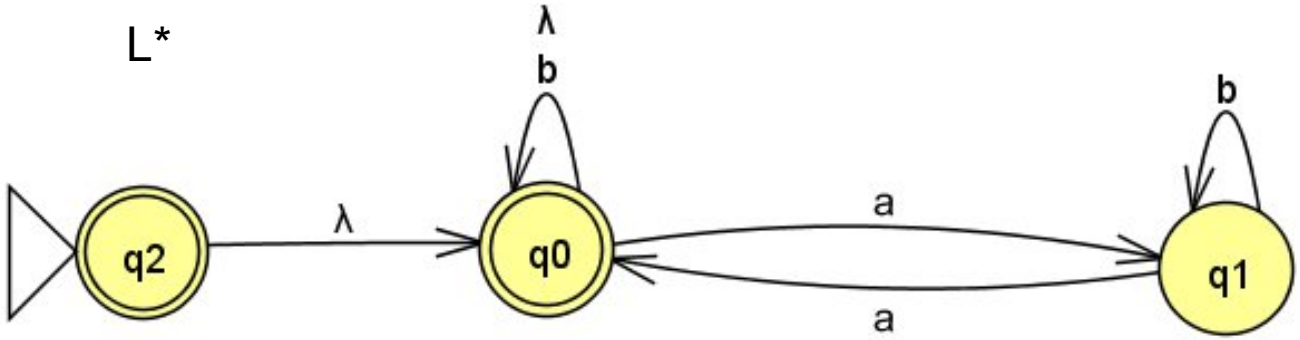
Construction for L·L'



L'' = (K,Σ,Δ,s,F)
$K = K_1 \cup K_2$
$s = s_1$
$F = F_2$
$\Delta = \Delta_1 \cup \Delta_2 \cup F_1 \times \{e\} \times \{s_2\}$

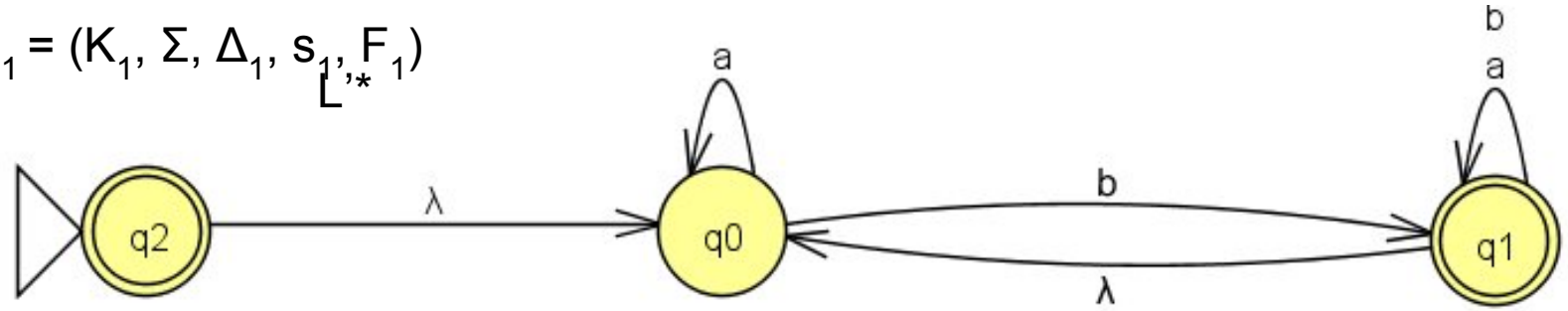
L* and L'*

L*

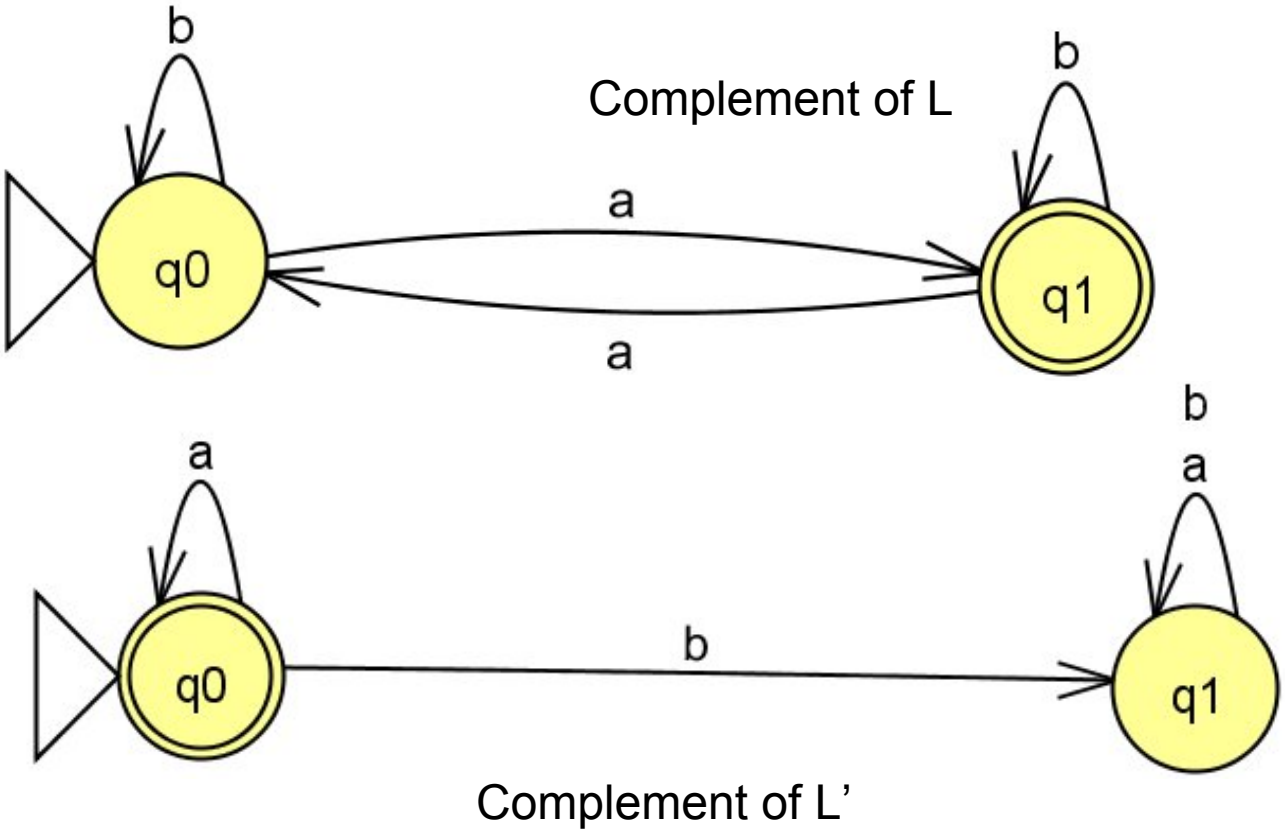


M = (K, Σ, Δ, s, F)
K = {s} ∪ K_1
F = {s} ∪ F_1
Δ = Δ_1 ∪ F_1 X {e} X {s_1} ∪ {(s, e, s_1)}
Given M_1 = (K_1, Σ, Δ_1, s_1, F_1)

L'*



Complement of L and L'



General Construction for Complement

DFA $M = (K, \Sigma, \delta, s, F)$

$K = K_1$

$s = s_1$

$F = K - F_1$

$\delta = \delta_1$

$L(M)$ = Complement of $L(M_1)$

DFA $M_1 = (K_1, \Sigma, \delta_1, s_1, F_1)$

Exercise: Will this construction work for NFAs?

Explain your answer.

Regular expressions versus FA's

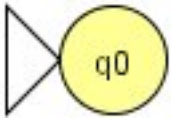
Regular expressions generate exactly the class of regular languages.

Theorem:

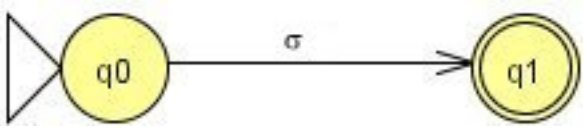
- (a) For every regular expression there is an equivalent NFA
- (b) For every DFA there is an equivalent regular expression.

Proof of (a):

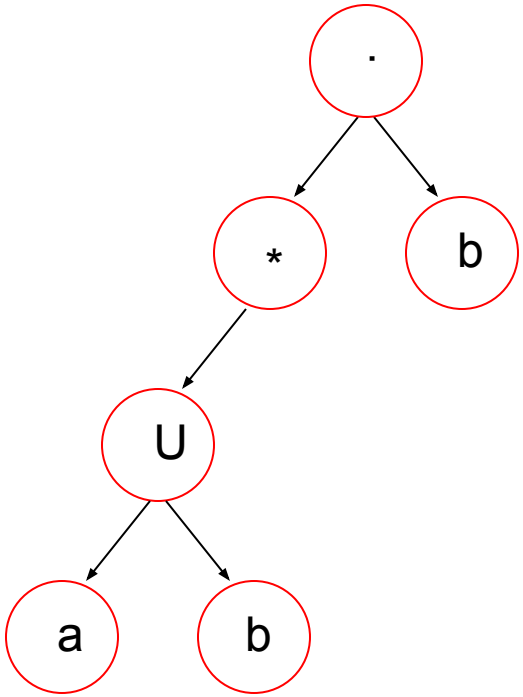
- For ϕ, the NFA is:
- For σ, the NFA is:



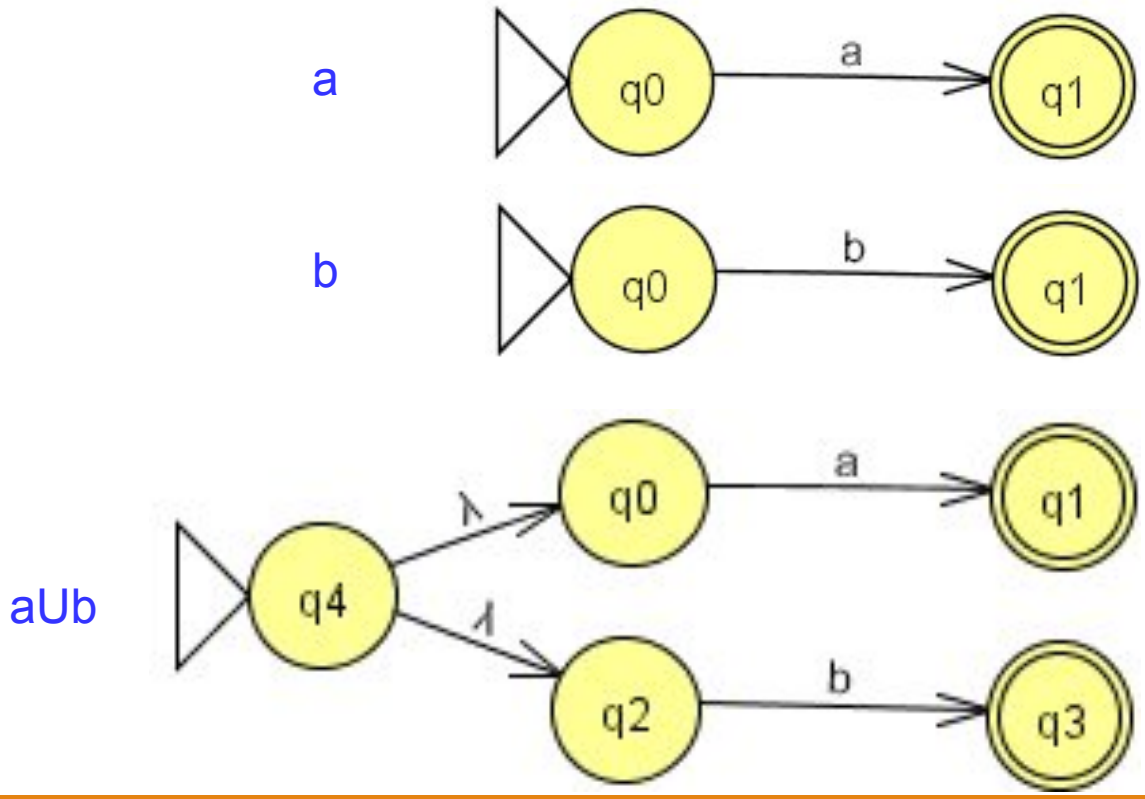
- For composite regular expressions: use closure under $\cup$, $\cdot$ and $*$



Parse Tree for (a U b)* · b

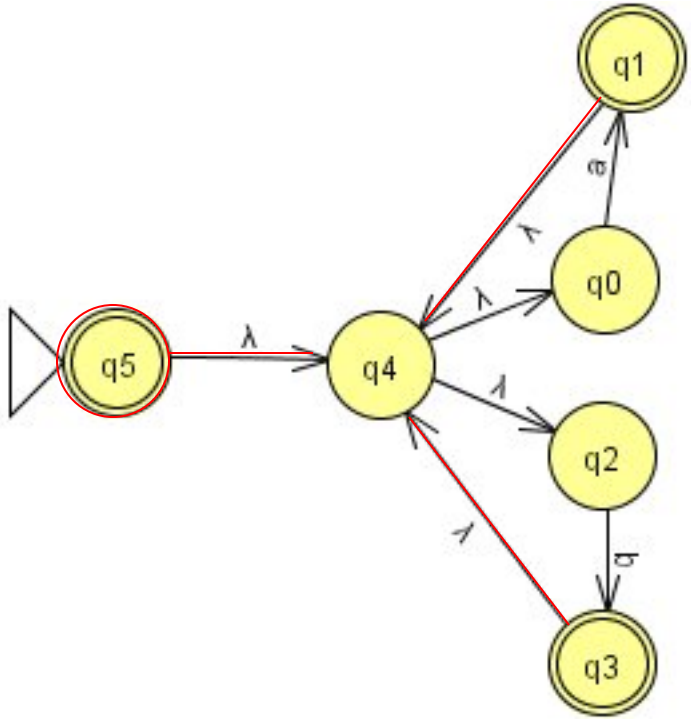


Example: NFA for (a U b)*b



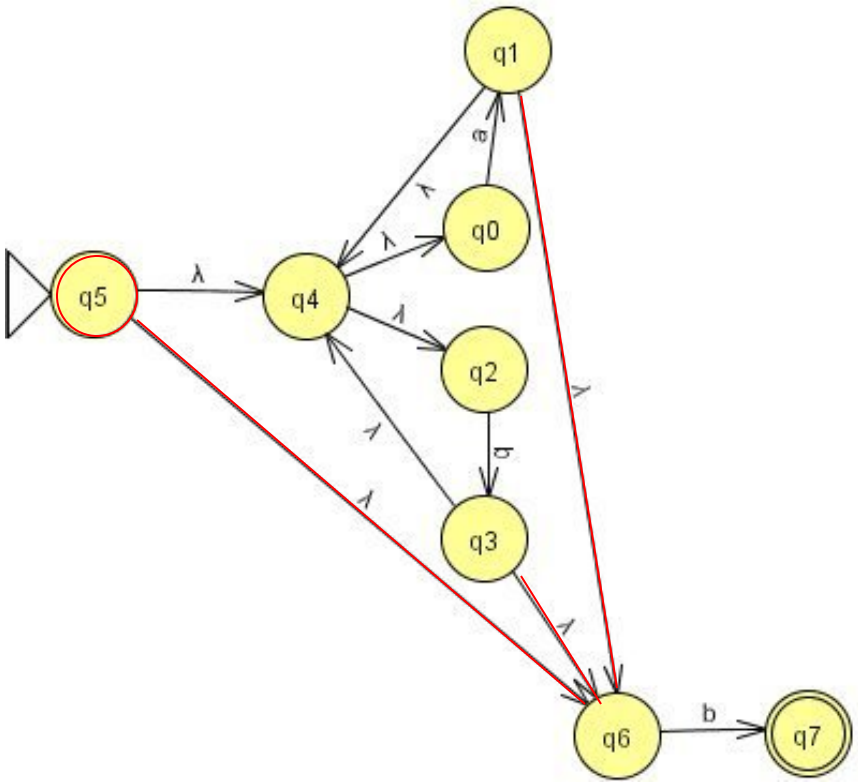
Example (contd.) : NFA for (a U b)*b

(aUb)*



Example (contd.) : NFA for (a U b)*b

(a U b)*b



DFA → regular expression

Easier to do:

- DFA → GNFA → regular expression.

GNFA (Generalized NFA)

- labels of transitions can be regular expressions.

Need special GNFA that satisfies:

(1) start state has no incoming transition

(2) only one final state

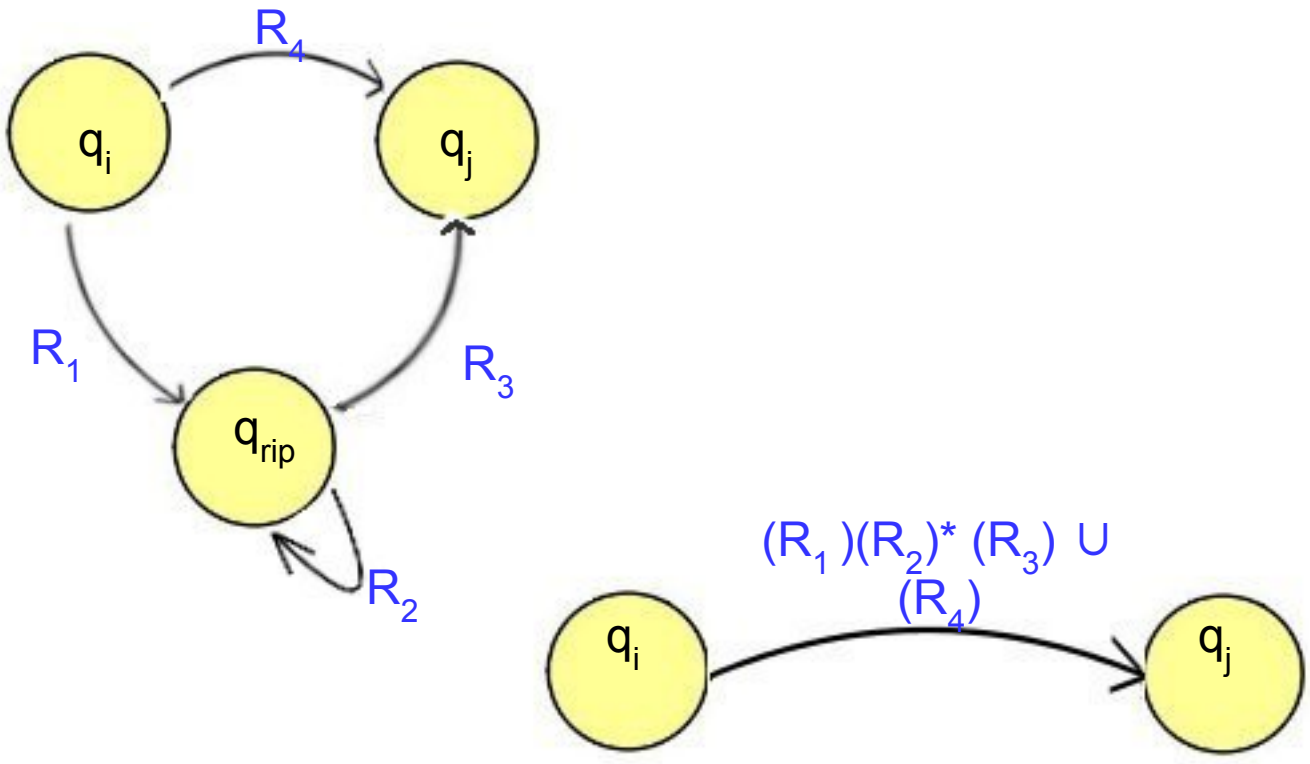
(3) final state has no outgoing transition.

DFA → regular expression (contd.)

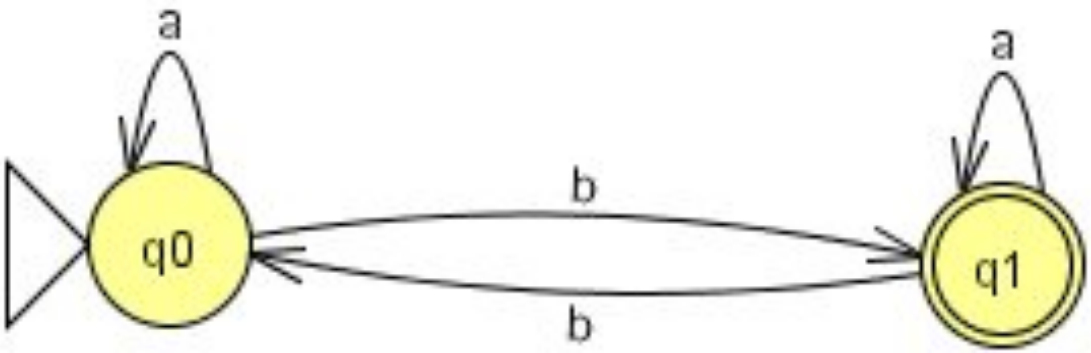
Idea:

- Convert DFA → special GNFA.
- Eliminate all states, except start and final state, one state at a time.
- Output the label on the single transition left at the end.

Eliminating state q{rip}

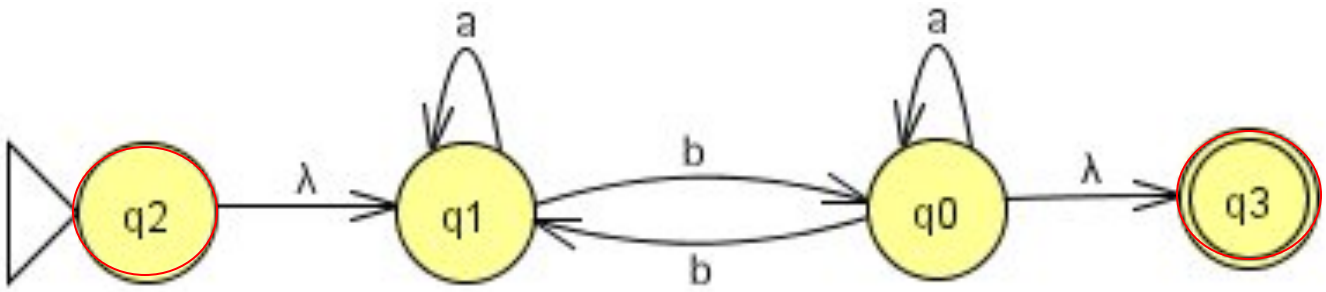


Constructing regular expression.



DFA L = {w in {a, b}* | w has odd number of b's }

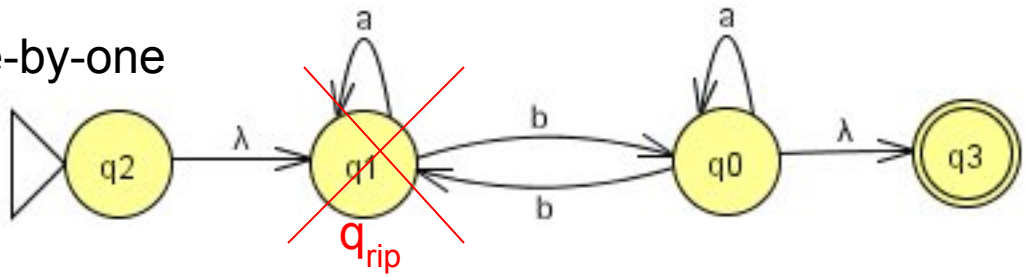
Constructing regular expression (contd.)



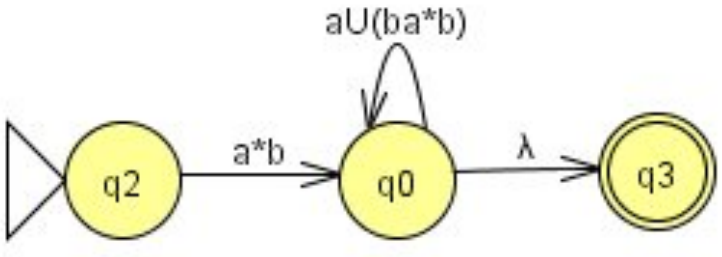
Added: a new start state and a new final state with empty transitions

Constructing regular expression (contd.)

Eliminate states one-by-one

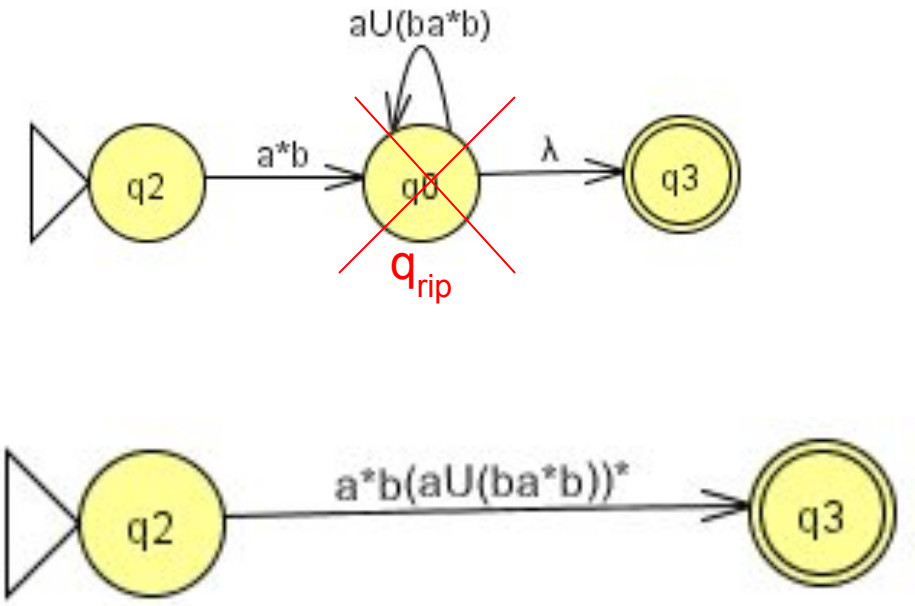


Before we take out q_{rip}, we have to see all the paths to the q_{rip} state and all possible transitions.



We take out the q_{rip} state and it leaves us with a 3 state Finite Automata

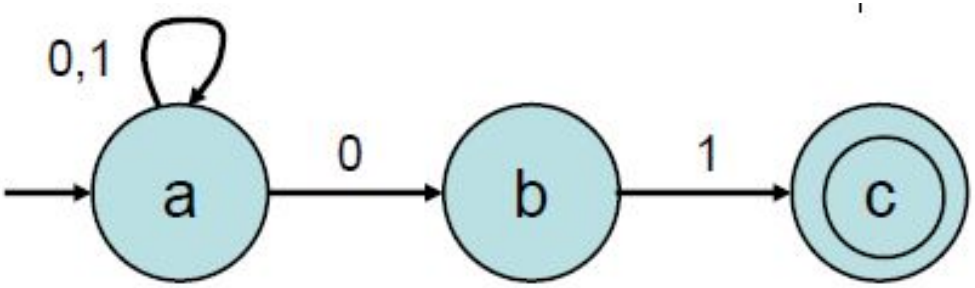
Constructing regular expression (contd.)



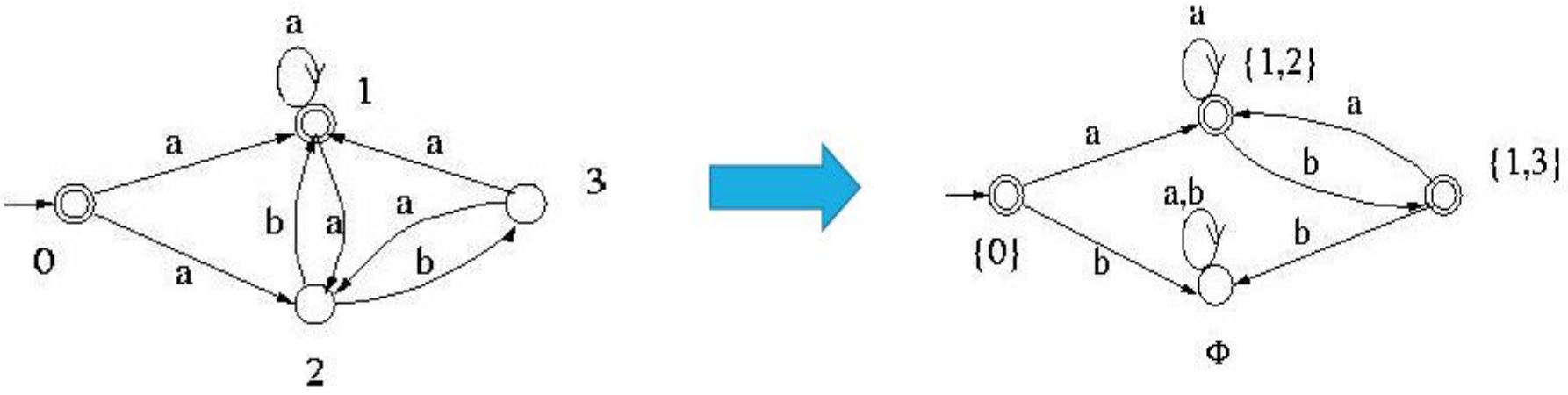
We take out the q_{rip} state and we are left with the regular expression

Converting NFA to DFA

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

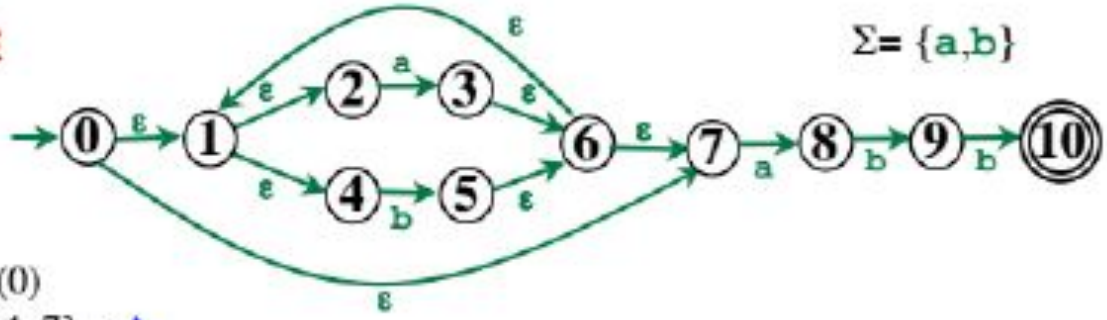


Converting NFA to DFA



Converting NFA to DFA

Example



Start state:

ε-Closure (0)

= {0, 1, 2, 4, 7} = A

$Move_{DFA}$(A,a)

= ε-Closure ($Move_{NFA}$(A,a))

= ε-Closure ({3,8})

= {1,2,3,4,6,7,8} = B

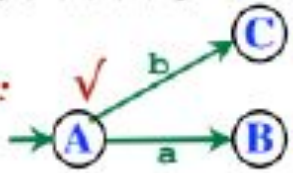
$Move_{DFA}$(A,b)

= ε-Closure ($Move_{NFA}$(A,b))

= ε-Closure ({5})

= {1,2,4,5,6,7} = C

So far: √



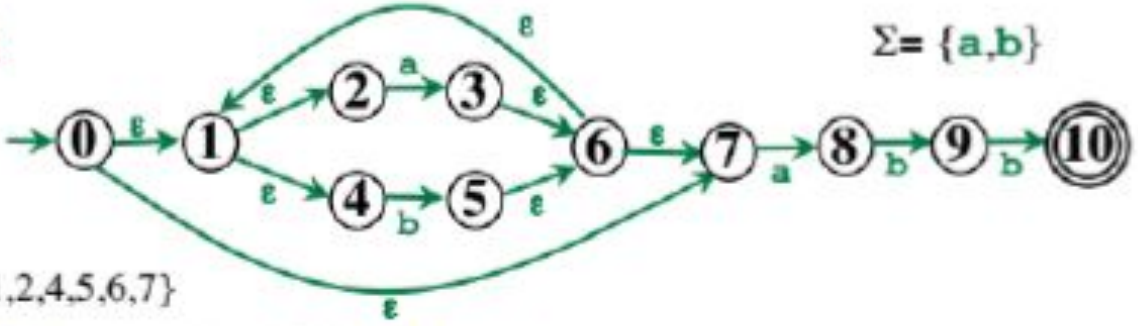
A is now done; mark it!

B and C are unmarked.

Let's do B next...

Converting NFA to DFA

Example



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

Process C = {1,2,4,5,6,7}

$Move_{DFA}(C,a) = \{1,2,3,4,6,7,8\} = B$

$Move_{DFA}(C,b) = \{1,2,4,5,6,7\} = C$

Process D = {1,2,4,5,6,7,9}

$Move_{DFA}(D,a) = \{1,2,3,4,6,7,8\} = B$

$Move_{DFA}(D,b) = \{1,2,4,5,6,7,10\} = E$

Process E = {1,2,4,5,6,7,10}

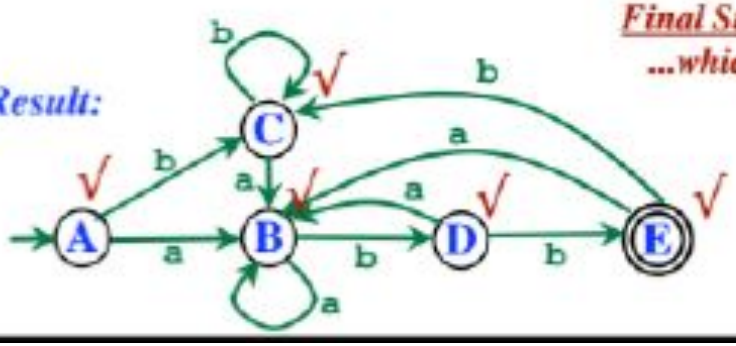
$Move_{DFA}(E,a) = \{1,2,3,4,6,7,8\} = B$

$Move_{DFA}(E,b) = \{1,2,4,5,6,7\} = C$

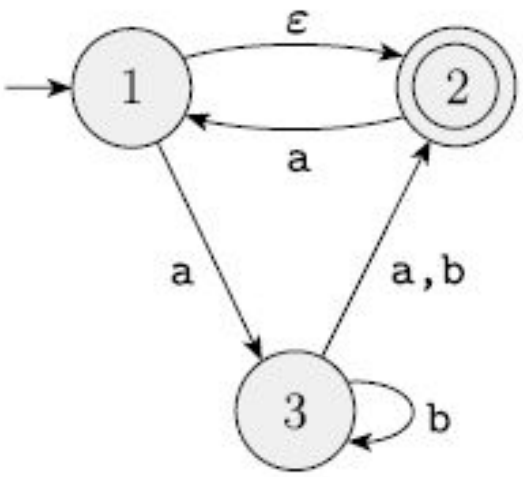
Final States in DFA?

...which state(s) contain 10?

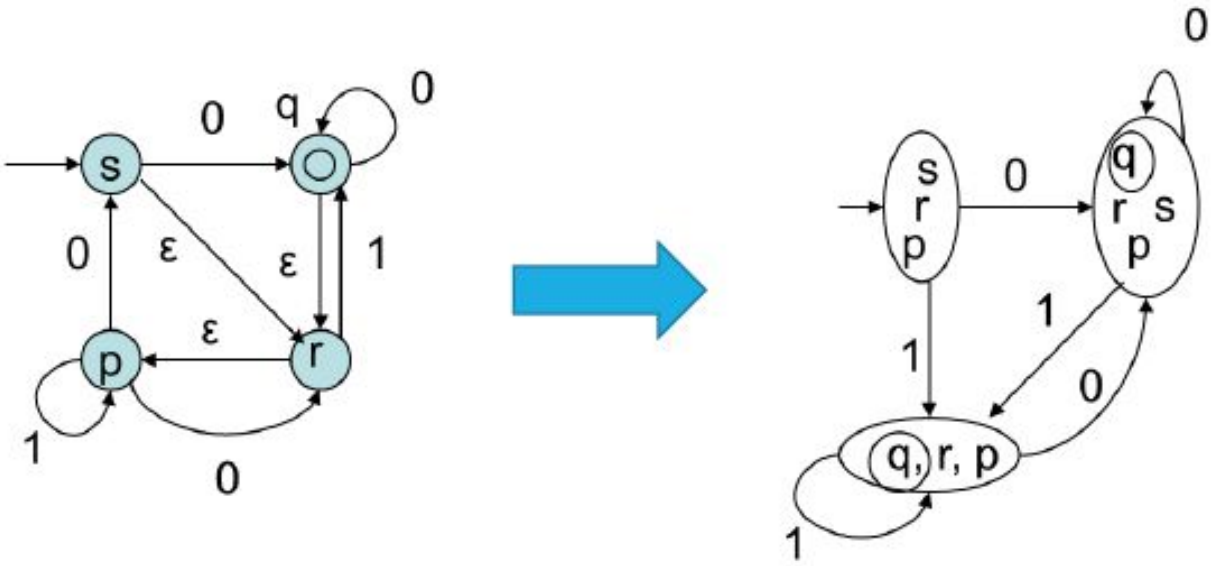
Final Result:



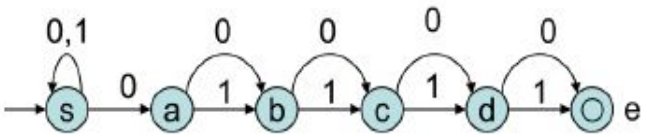
Converting NFA to DFA

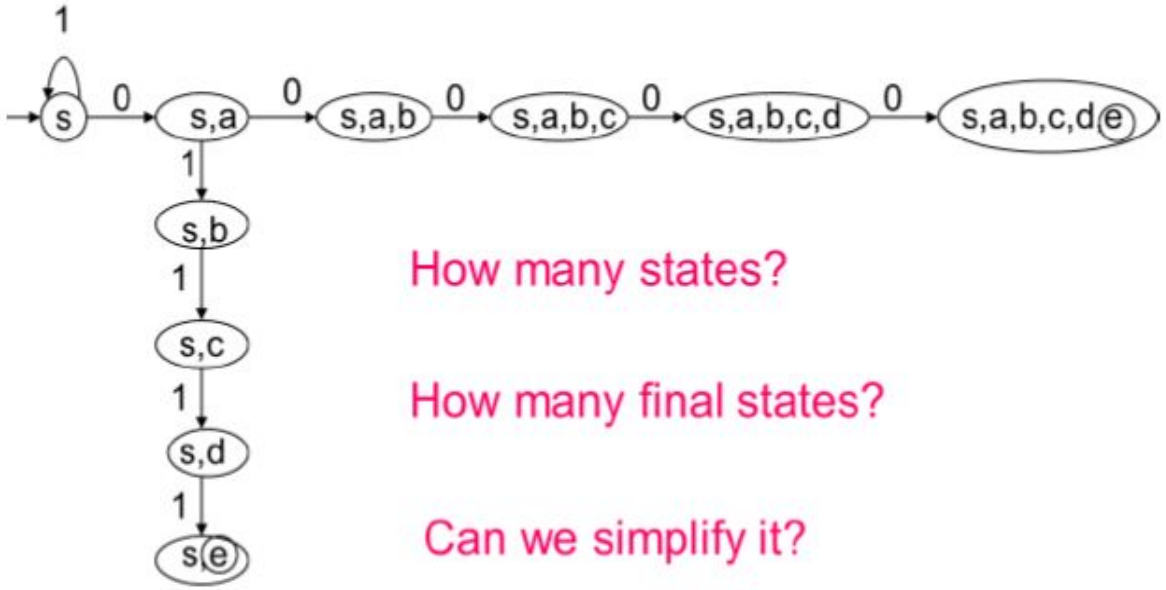


Converting NFA to DFA



Converting NFA to DFA





How many states?

How many final states?

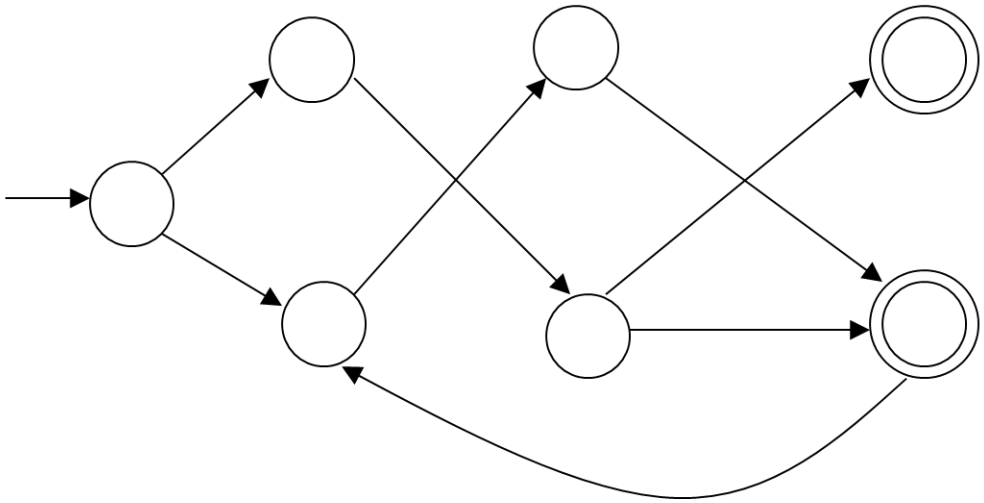
Can we simplify it?

The Pumping Lemma

Take an infinite regular language L

(contains an infinite number of strings)

There exists a DFA that accepts L

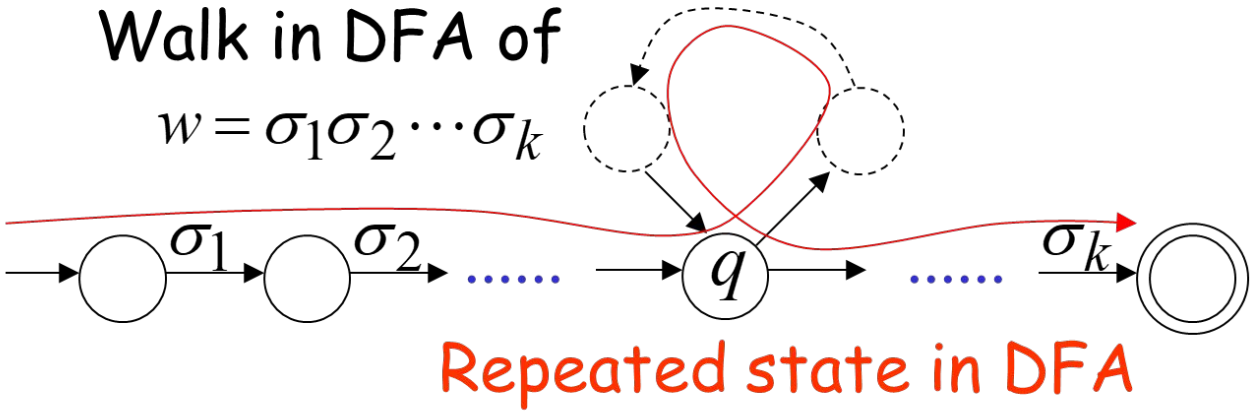


m states

Take string $w \in L$ with $|w| \geq m$

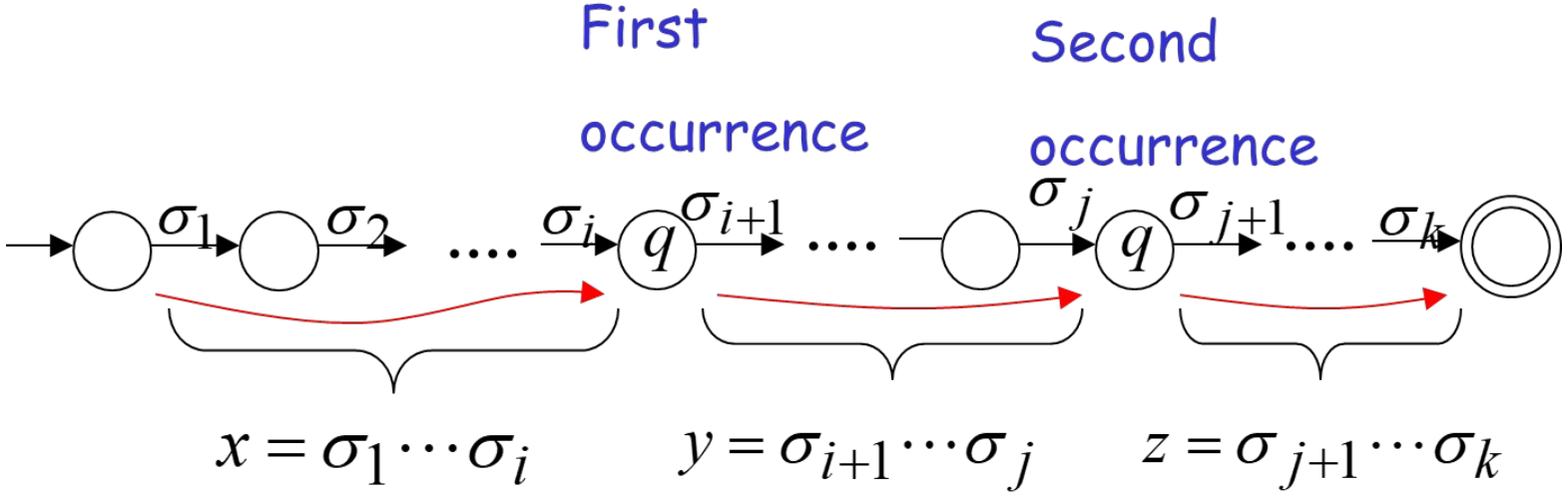
(number of states of DFA)

then, at least one state is repeated in the walk of w



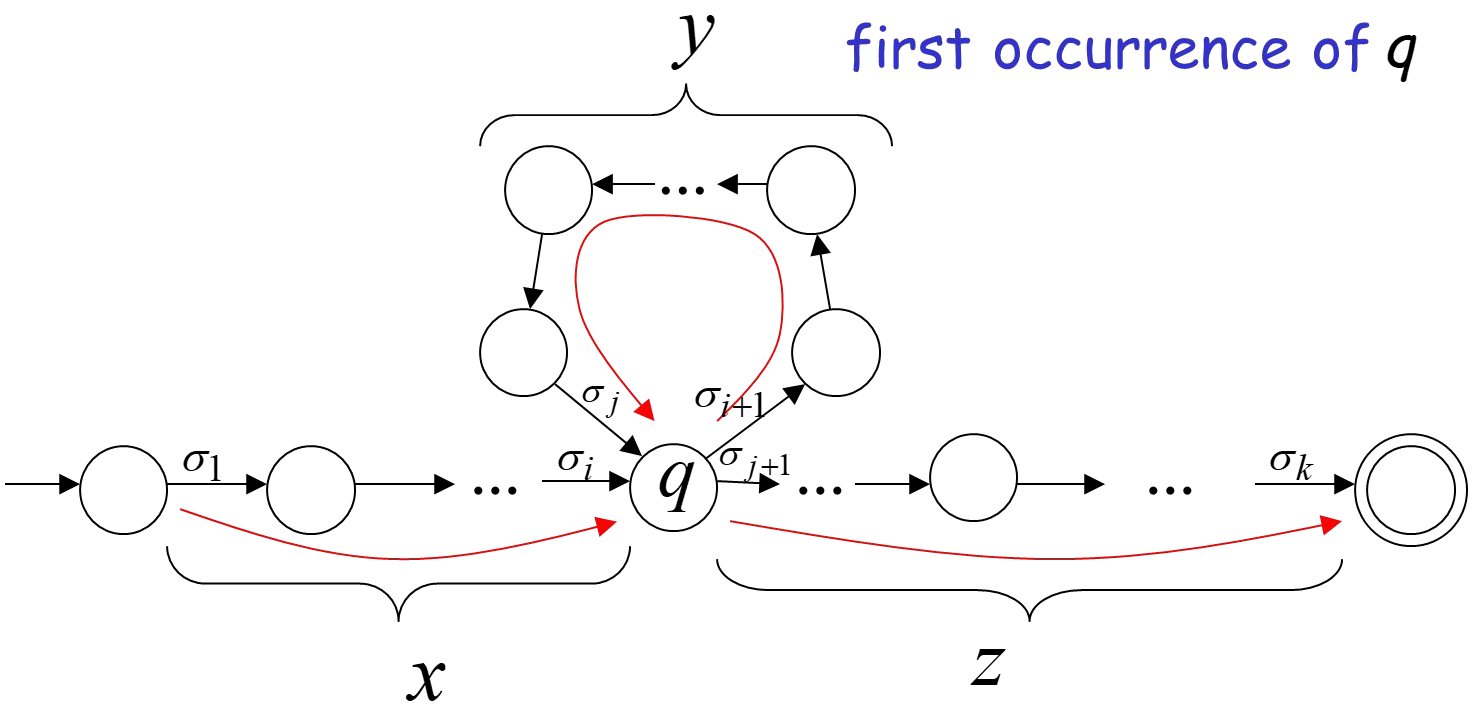
We can write $w = xyz$

One dimensional projection of walk w:



In DFA: $w = x\ y\ z$

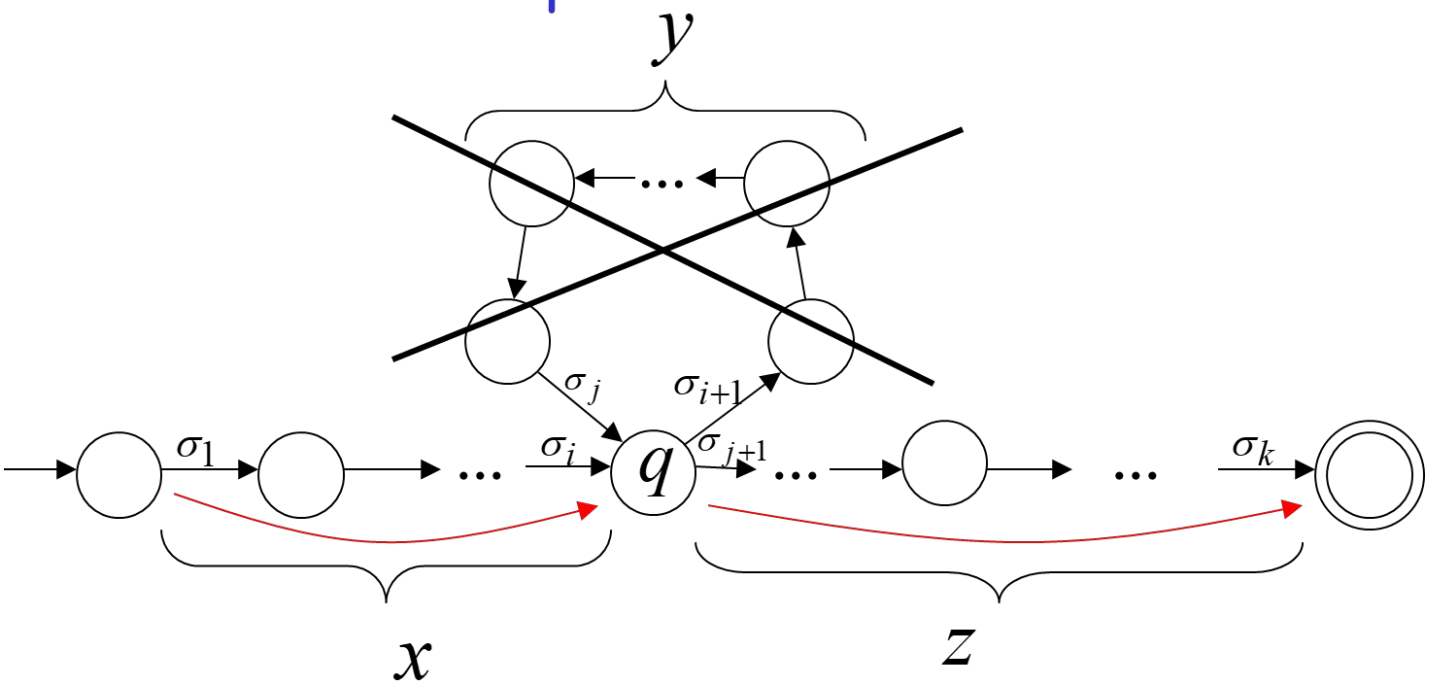
contains only
first occurrence of q



Additional string:

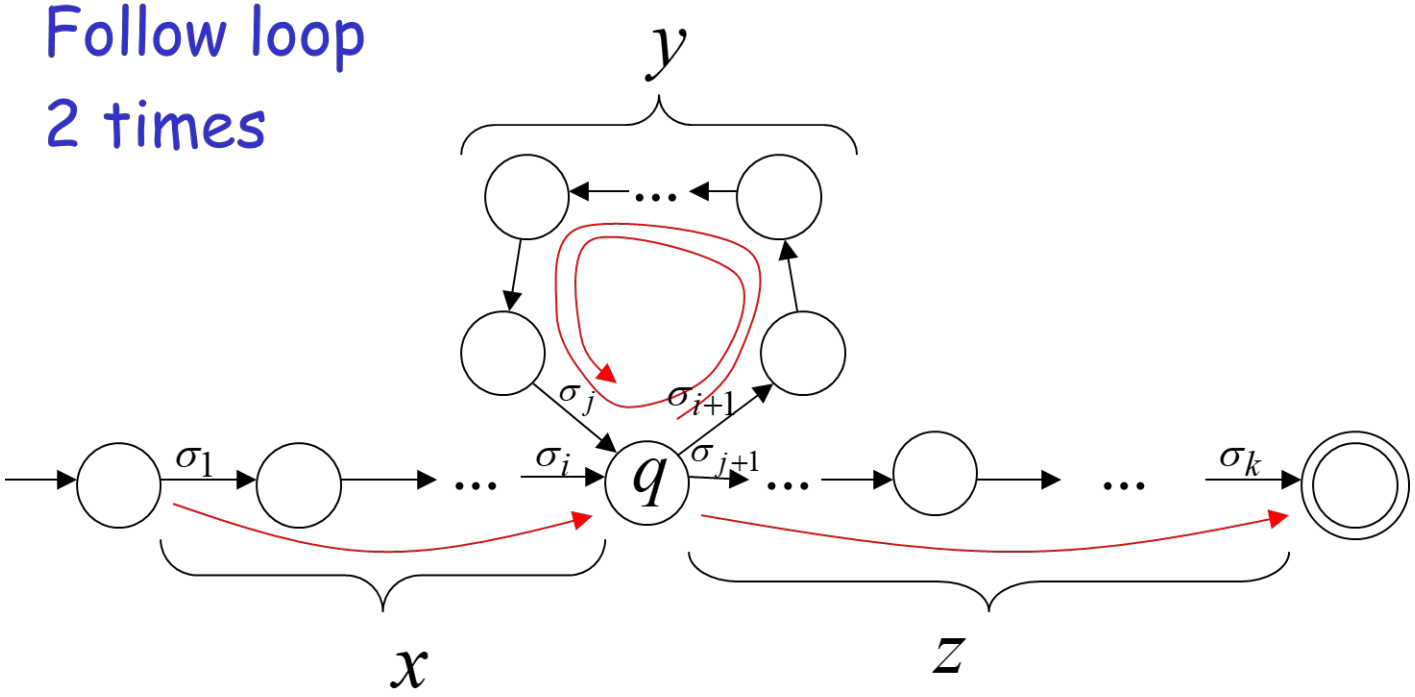
The string $x\,z$ is accepted

Do not follow loop



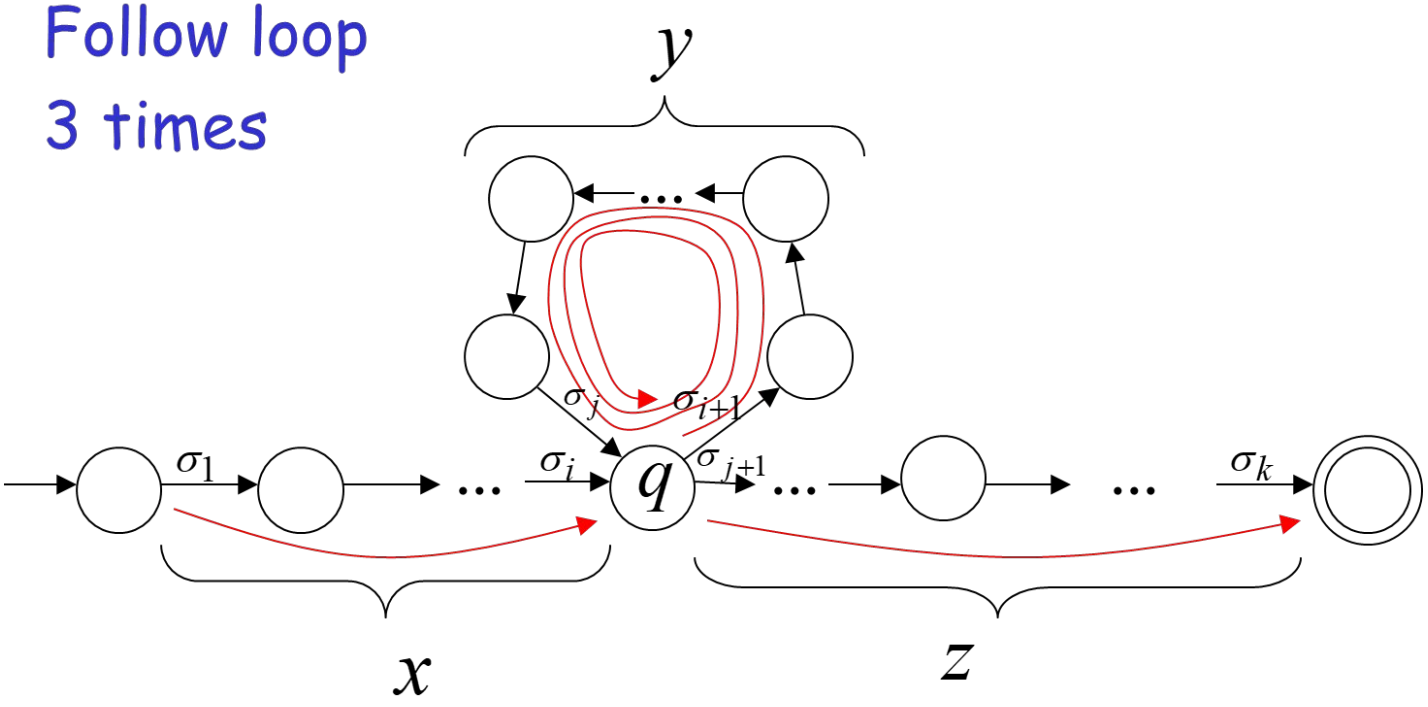
Additional string: The string $x\ y\ y\ z$ is accepted

Follow loop 2 times



Additional string: The string $x\ y\ y\ y\ z$ is accepted

Follow loop 3 times

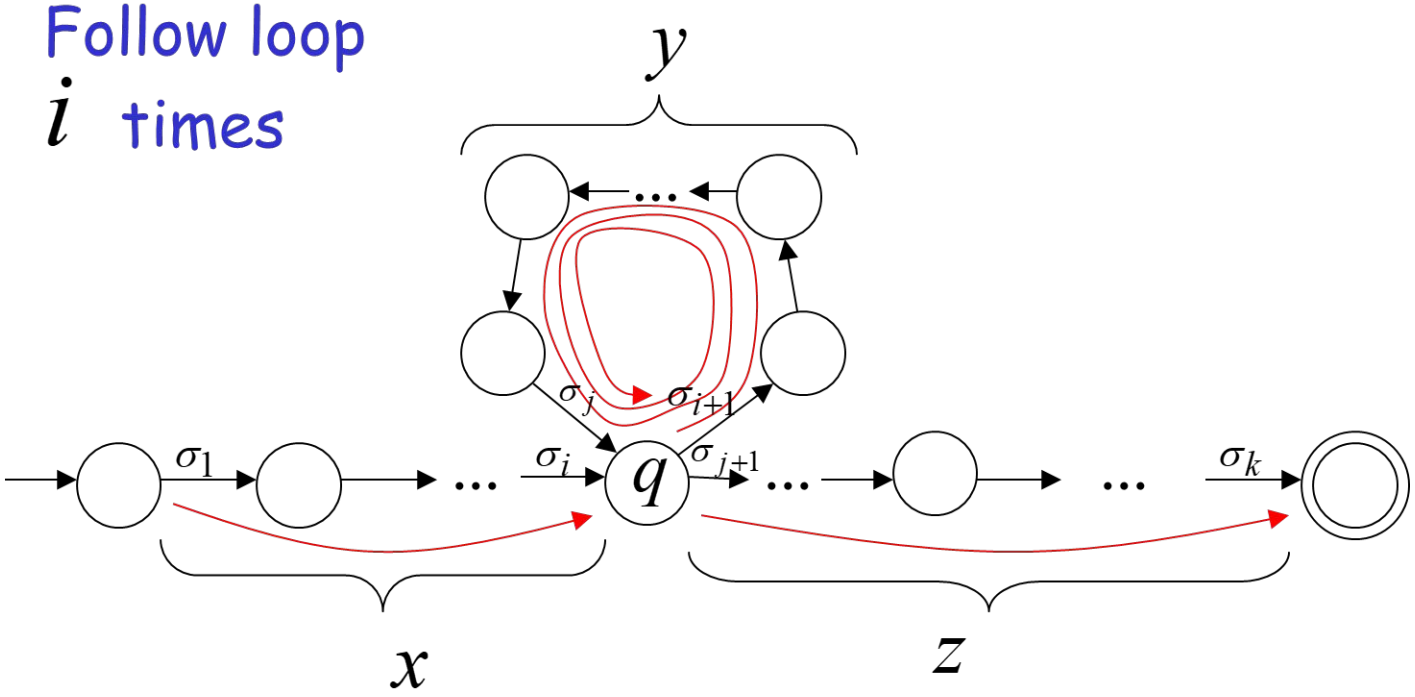


In General:

The string $x\, y^i\, z$

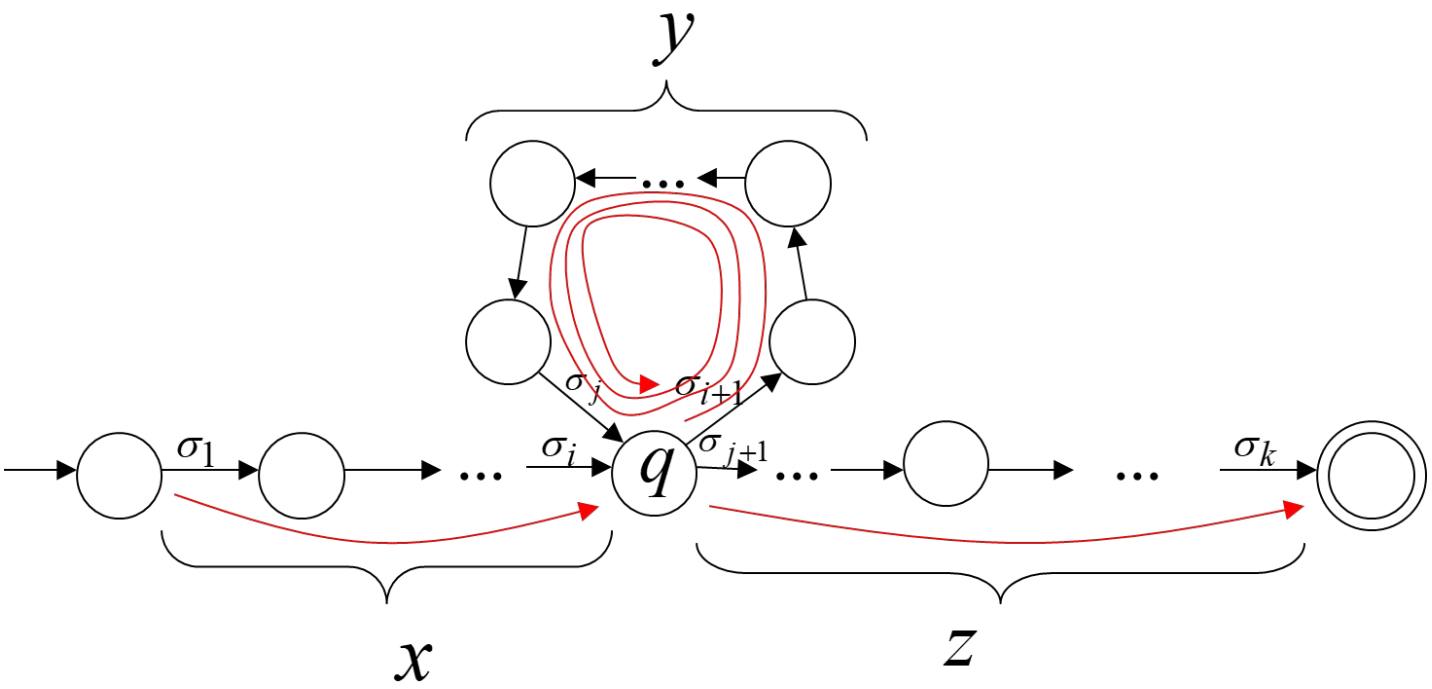
is accepted $i = 0, 1, 2, \ldots$

Follow loop i times



Therefore: $x\ y^i\ z\ \in L \qquad i = 0, 1, 2, \ldots$

Language accepted by the DFA



The Pumping Lemma:

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

The Pumping Lemma:

- Given a infinite regular language L
- there exists an integer m (critical length)
- for any string $w \in L$ with length $|w| \geq m$
- we can write $w = x\ y\ z$
- with $|x\ y| \leq m$ and $|y| \geq 1$
- such that: $x\ y^i\ z \in L \qquad i = 0, 1, 2, \ldots$

Applications of the Pumping Lemma

Observation:

Every language of finite size has to be regular

(we can easily construct an NFA
that accepts every string in the language)

Therefore, every non-regular language
has to be of infinite size

(contains an infinite number of strings)

Suppose you want to prove that
An infinite language L is not regular

1. Assume the opposite: L is regular

2. The pumping lemma should hold for L

3. Use the pumping lemma to obtain a contradiction

4. Therefore, L is not regular

Applications of the Pumping Lemma

Theorem: The language $L = \{a^n b^n : n \geq 0\}$ is not regular

Proof: Use the Pumping Lemma

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

Assume for contradiction
that L is a regular language

Since L is infinite
we can apply the Pumping Lemma

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

Let m be the critical length for L

Pick a string w such that: $w \in L$

and length $|w| \geq m$

We pick $w = a^m b^m$

From the Pumping Lemma:

we can write $w = a^m b^m = x\ y\ z$

with lengths $|x\ y| \leq m, \quad |y| \geq 1$

$$w = xyz = a^m b^m = \overbrace{a\ldots a}^{x}\overbrace{a\ldots a}^{y}\overbrace{a\ldots a\,b\ldots b}^{z}$$

(m a's, then m b's)

Thus: $y = a^k, \quad 1 \leq k \leq m$

$x\,y\,z = a^m b^m \qquad y = a^k,\ \ 1 \le k \le m$

From the Pumping Lemma: $x\,y^i\,z \in L$

$i = 0, 1, 2, \ldots$

Thus: $x\,y^2\,z \in L$

$x\,y\,z = a^m b^m \qquad y = a^k,\ \ 1 \le k \le m$

From the Pumping Lemma: $x\,y^2\,z \in L$

$$xy^2z = \overbrace{a\ldots a}^{m+k}\overbrace{a\ldots a}^{m}\overbrace{a\ldots a}^{m}\overbrace{a\ldots a}^{m}\overbrace{b\ldots b}^{m} \in L$$

$x \quad y \quad y \quad z$

Thus: $a^{m+k} b^m \in L$

$$a^{m+k}b^m \in L \qquad k \geq 1$$

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

$$a^{m+k}b^m \notin L$$

CONTRADICTION!!!