NARSIMHA REDDY ENGINEERING COLLEGE
UGC AUTONOMOUS INSTITUTION
Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India
Accredited by NBA & NAAC with 'A' Grade
Approved by AICTE
Permanently affiliated to JNTUH

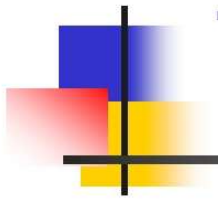
FORMAL LANGUAGES AND AUTOMATA THEORY

(23CY503)

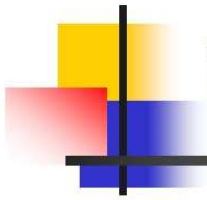
Prepared by,
M. Rajeshwari, Asst.Prof
Department Of CSE (CS)

FORMAL LANGUAGES AND AUTOMATA THEORY

UNIT 1



Introduction to Automata Theory



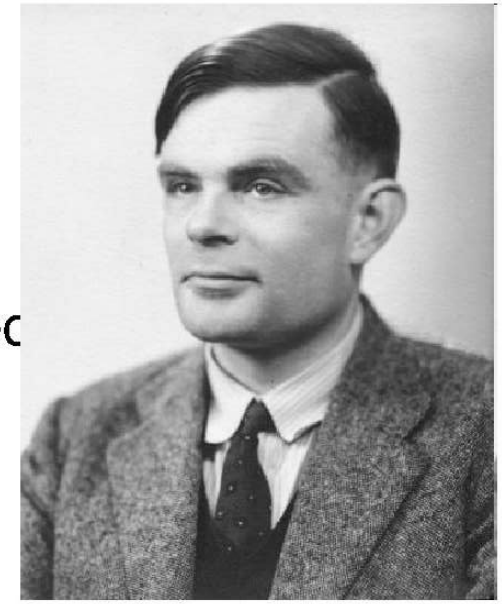
What is Automata Theory?

- *Study of abstract computing devices, or "machines"*
- Automaton = an abstract computing device
  - Note: A "device" need not even be a physical hardware!
- A fundamental question in computer science:
  - Find out what different models of machines can do and cannot do
  - The *theory of computation*
- Computability vs. Complexity

(A pioneer of automata theory)

Alan Turing (1912-1954)

- Father of Modern Computer Science
- English mathematician
- Studied abstract machines called ***Turing machines*** even before computers existed
- Heard of the Turing test?



Languages & Grammars

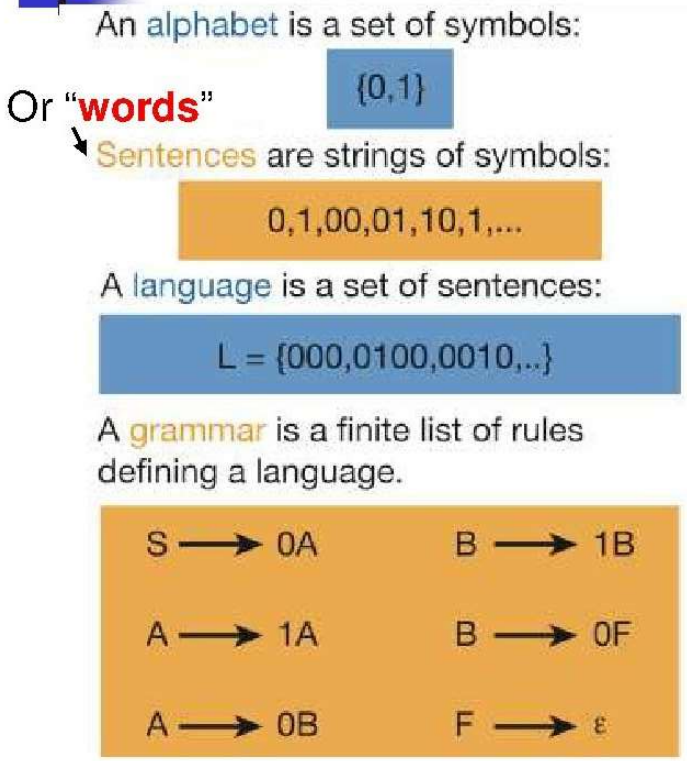


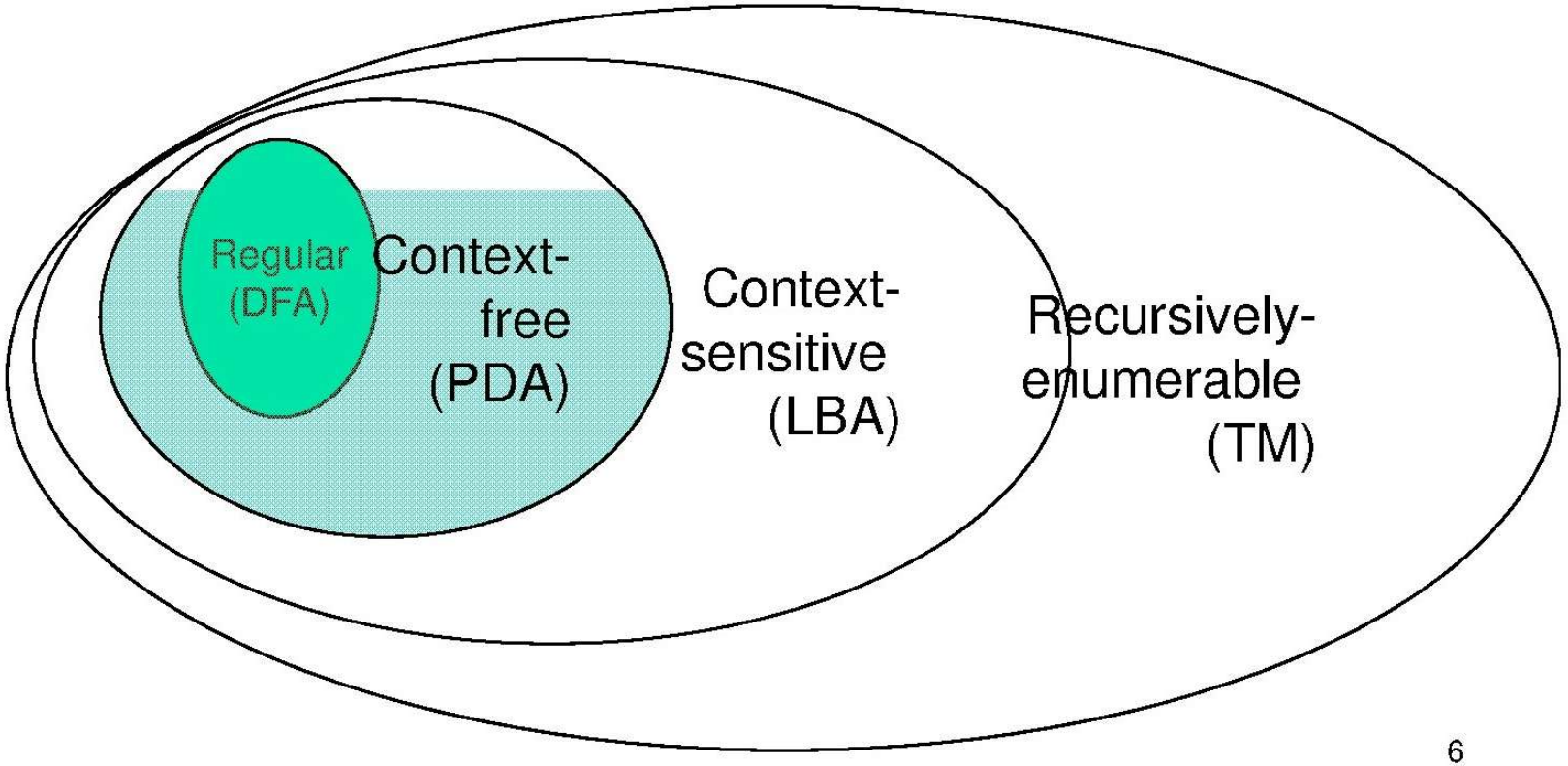
Image source: Nowak et al. Nature, vol 417, 2002

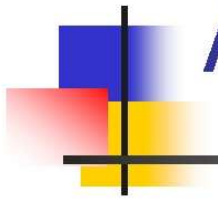
- Languages: "*A language is a collection of sentences of finite length all constructed from a finite alphabet of symbols*"
- Grammars: "*A grammar can be regarded as a device that enumerates the sentences of a language*" - nothing more, nothing less

- *N. Chomsky, Information and Control, Vol 2, 1959*

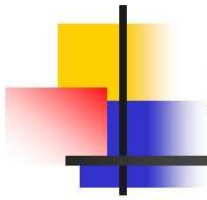
The Chomsky Hierachy

- A containment hierarchy of classes of formal languages





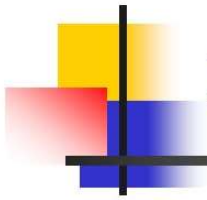
The Central Concepts of Automata Theory



Alphabet

An alphabet is a finite, non-empty set of symbols

- We use the symbol ∑ (sigma) to denote an alphabet
- Examples:
  - Binary: ∑ = {0,1}
  - All lower case letters: ∑ = {a,b,c,..z}
  - Alphanumeric: ∑ = {a-z, A-Z, 0-9}
  - DNA molecule letters: ∑ = {a,c,g,t}
  - …



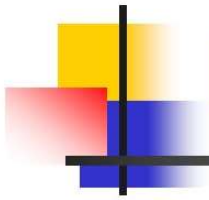
Strings

A string or word is a finite sequence of symbols chosen from ∑

- ***Empty string is ε (or "epsilon")***

- Length of a string *w*, denoted by "|*w*|", is equal to the *number of (non-ε) characters in the string*
  - *E.g., x = 010100* |*x*| = 6
  - *x = 01 ε 0 ε 1 ε 00 ε* |*x*| = ?

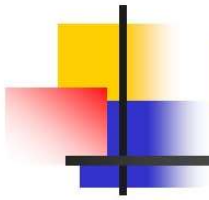
- *xy* = *c*oncatentation of two strings *x* and *y*



Powers of an alphabet

Let $\sum$ be an alphabet.

- $\sum^k$ = the set of all strings of length k
- $\sum^* = \sum^0 \cup \sum^1 \cup \sum^2 \cup \ldots$
- $\sum^+ = \sum^1 \cup \sum^2 \cup \sum^3 \cup \ldots$



Languages

*L is a said to be a language over alphabet ∑, only if L ⊆ ∑**

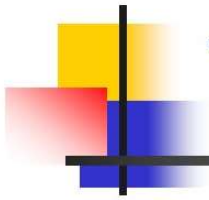
➔ this is because ∑* is the set of all strings (of all possible length including 0) over the given alphabet ∑

Examples:

1. Let L be *the* language of all strings consisting of *n* 0's followed by *n* 1's:
   L = {ε,01,0011,000111,…}
2. Let L be *the* language of all strings of with equal number of 0's and 1's:
   L = {ε,01,10,0011,1100,0101,1010,1001,…}

Definition: Ø denotes the Empty language

- Let L = {ε}; Is L=Ø? NO



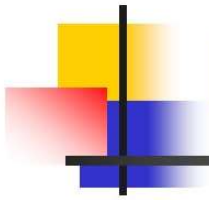
The Membership Problem

Given a string $w \in \sum^*$ and a language L over $\sum$, decide whether or not $w \in L$.

Example:

Let w = 100011

Q) Is w $\in$ the language of strings with equal number of 0s and 1s?

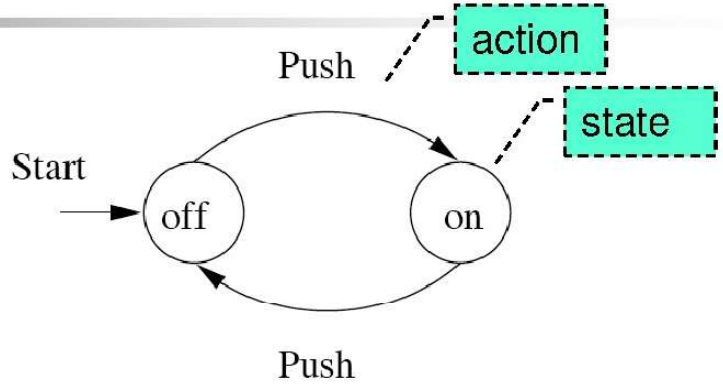


Finite Automata

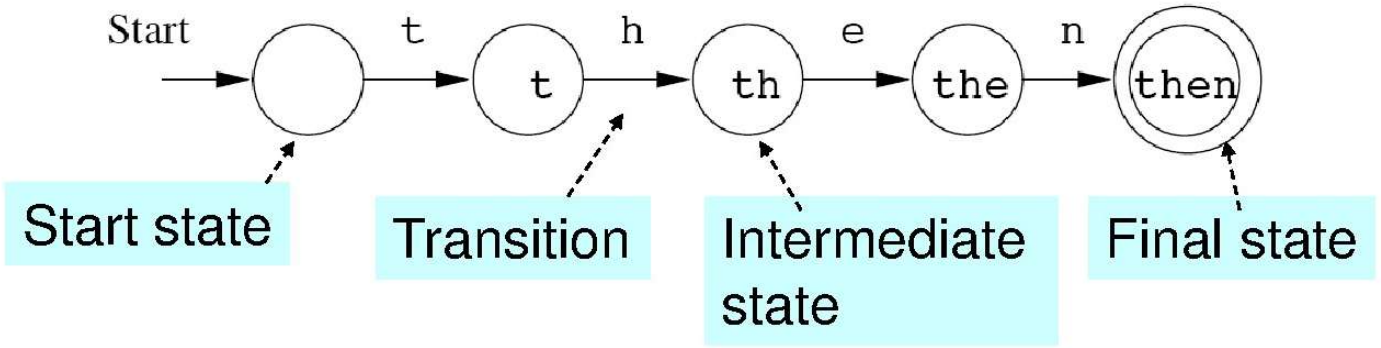
- Some Applications
  - Software for designing and checking the behavior of digital circuits
  - Lexical analyzer of a typical compiler
  - Software for scanning large bodies of text (e.g., web pages) for pattern finding
  - Software for verifying systems of all types that have a finite number of states (e.g., stock market transaction, communication/network protocol)

Finite Automata : Examples

- On/Off switch

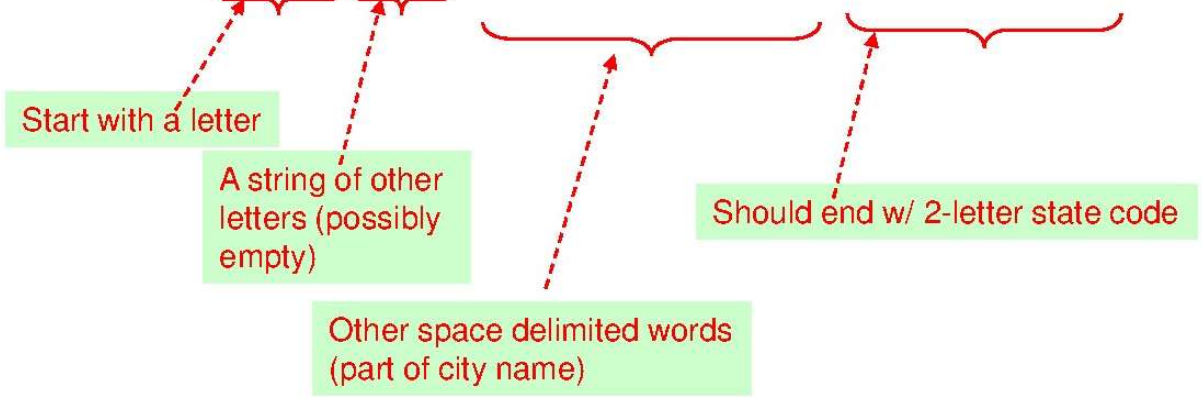


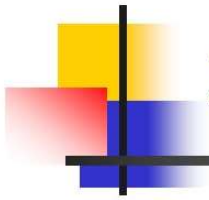
- Modeling recognition of the word "*then*"



Structural expressions

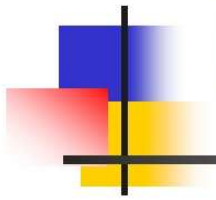
- Grammars
- Regular expressions
  - E.g., unix style to capture city names such as “Palo Alto CA”:
    - [A-Z][a-z]*([ ][A-Z][a-z]*)*[ ][A-Z][A-Z]



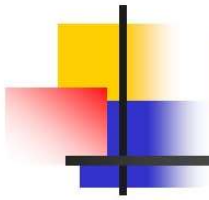


Summary

- Automata theory & a historical perspective
- Chomsky hierarchy
- Finite automata
- Alphabets, strings/words/sentences, languages
- Membership problem



Finite Automata



Finite Automaton (FA)

- Informally, a state diagram that comprehensively captures all possible states and transitions that a machine can take while responding to a stream or sequence of input symbols
- Recognizer for “Regular Languages”

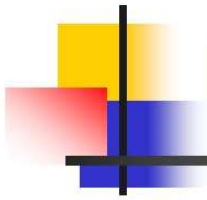
- Deterministic Finite Automata (DFA)
  - The machine can exist in only one state at any given time
- Non-deterministic Finite Automata (NFA)
  - The machine can exist in multiple states at the same time

Deterministic Finite Automata - Definition

- A Deterministic Finite Automaton (DFA) consists of:
  - Q ==> a finite set of states
  - $\sum$ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of final states
  - δ ==> a transition function, which is a mapping between Q x $\sum$ ==> Q
- A DFA is defined by the 5-tuple:
  - {Q, $\sum$, q_0,F, δ }

What does a DFA do on reading an input string?

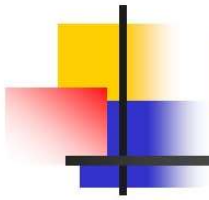
- Input: a word w in $\sum^*$
- Question: Is w acceptable by the DFA?
- Steps:
  - Start at the "start state" q_0
  - For every input symbol in the sequence w do
    - Compute the next state from the current state, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed, the current state is one of the final states (F) then *accept w;*
  - Otherwise, *reject w.*



Regular Languages

- Let L(A) be a language *recognized* by a DFA A.
  - Then L(A) is called a "*Regular Language*".

- Locate regular languages in the Chomsky Hierarchy



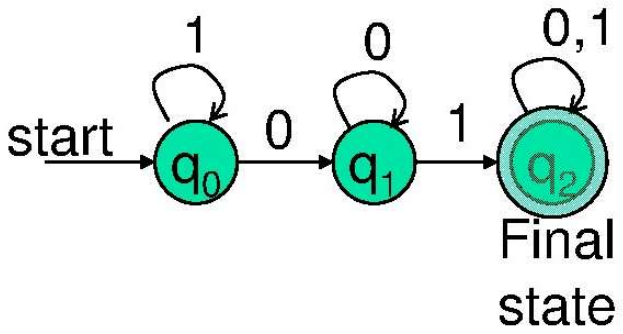
Example #1

- Build a DFA for the following language:
  - L = {w | w is a binary string that contains 01 as a substring}
- Steps for building a DFA to recognize L:
  - ∑ = {0,1}
  - Decide on the states: Q
  - Designate start state and final state(s)
  - δ: Decide on the transitions:
- Final states == same as “accepting states”
- Other states == same as “non-accepting states”

Regular expression: (0+1)*01(0+1)*

DFA for strings containing 01

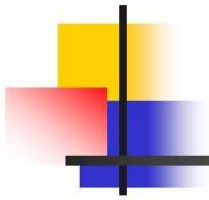
- What makes this DFA deterministic?



- What if the language allows empty strings?

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

	symbols	
states δ	**0**	**1**
→ **q_0**	q_1	q_0
q_1	q_1	q_2
***q_2**	q_2	q_2



Example #2

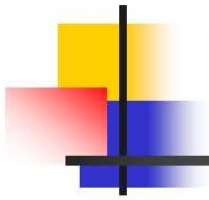
Clamping Logic:

- A clamping circuit waits for a ”1” input, and turns on forever. However, to avoid clamping on spurious noise, we’ll design a DFA that waits for *two consecutive 1s* in a row before clamping on.

- Build a DFA for the following language:

  L = { w | w is a bit string which contains the substring 11}

- State Design:
  - q_0 : start state (initially off), also means the most recent input was not a 1
  - q_1: has never seen 11 but the most recent input was a 1
  - q_2: has seen 11 at least once



Example #3

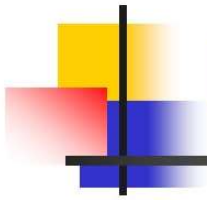
- Build a DFA for the following language:

  L = { w | w is a binary string that has even number of 1s and even number of 0s}

- ?

Extension of transitions (δ) to Paths ($\hat{\delta}$)

- $\hat{\delta}$ *(q,w) = destination state* from state *q* on input string *w*
- $\hat{\delta}$ *(q,wa)* = δ ($\hat{\delta}$*(q,w), a)*
  - Work out example #3 using the input sequence w=10010, a=1:
    - $\hat{\delta}$ (q_0,wa) = ?



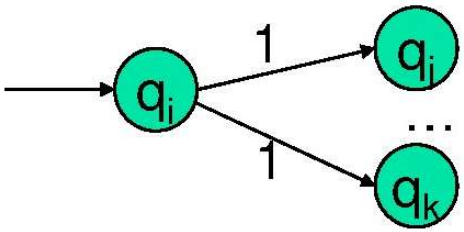
Language of a DFA

A DFA A accepts string *w* if there is a path from q_0 to an accepting (or final) state that is labeled by *w*

- *i.e.,* $L(A) = \{ w \mid \hat{\delta}(q_0,w) \in F \}$
- *I.e., L(A) = all strings that lead to a final state from* q_0

Non-deterministic Finite Automata (NFA)

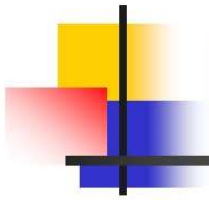
- A Non-deterministic Finite Automaton (NFA)
  - is of course "non-deterministic"
    - Implying that the machine can exist in more than one state at the same time
    - Transitions could be non-deterministic



• Each transition function therefore maps to a set of states

Non-deterministic Finite Automata (NFA)

- A Non-deterministic Finite Automaton (NFA) consists of:
  - Q ==> a finite set of states
  - $\sum$ ==> a finite set of input symbols (alphabet)
  - q_0 ==> a start state
  - F ==> set of final states
  - δ ==> a transition function, which is a mapping between Q x $\sum$ ==> subset of Q
- An NFA is also defined by the 5-tuple:
  - {Q, $\sum$, q_0,F, δ }



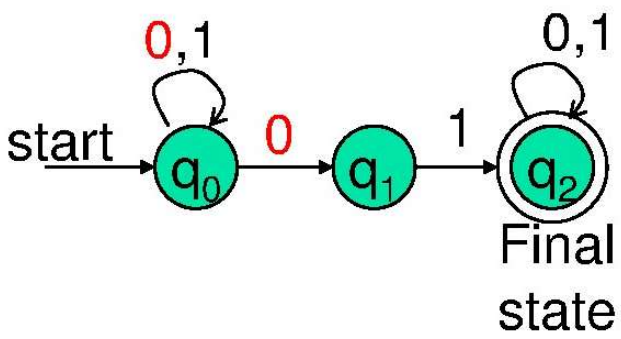
How to use an NFA?

- Input: a word w in $\sum^*$
- Question: Is w acceptable by the NFA?
- Steps:
  - Start at the "start state" q_0
  - For every input symbol in the sequence w do
    - Determine all possible next states from all current states, given the current input symbol in w and the transition function
  - If after all symbols in w are consumed and if at least one of the current states is a final state then *accept w;*
  - Otherwise, *reject w.*

Regular expression: (0+1)*01(0+1)*

NFA for strings containing 01

Why is this non-deterministic?



What will happen if at state q_1 an input of 0 is received?

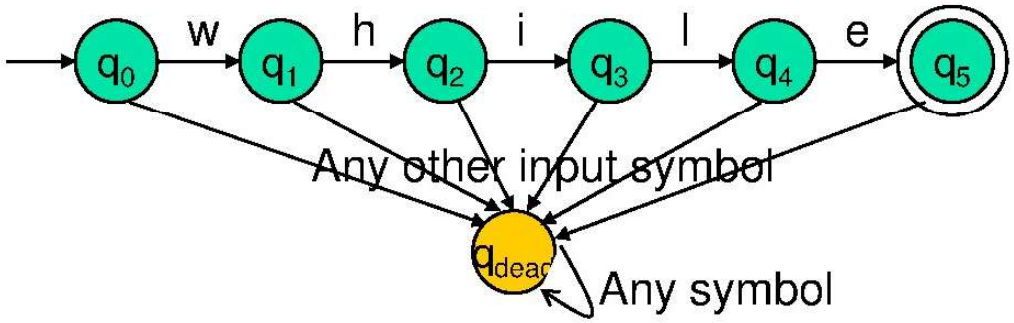
- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0,1\}$
- start state = q_0
- $F = \{q_2\}$
- Transition table

	symbols	
states / δ	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	Φ	$\{q_2\}$
$*q_2$	$\{q_2\}$	$\{q_2\}$

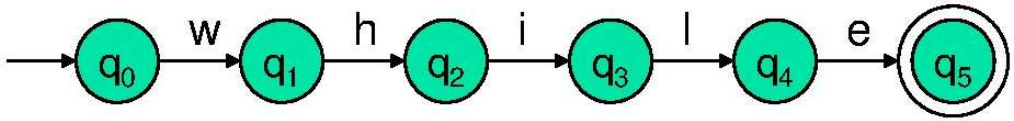
Note: Explicitly specifying dead states is just a matter of design convenience
(one that is generally followed in NFAs), and
this feature does not make a machine deterministic or non-deterministic.

What is a "dead state"?

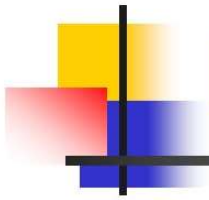
- A DFA for recognizing the key word "*while*"



- An NFA for the same purpose:



Transitions into a dead state are implicit

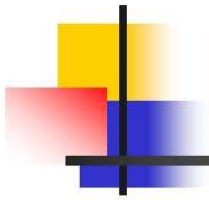


Example #2

- Build an NFA for the following language:

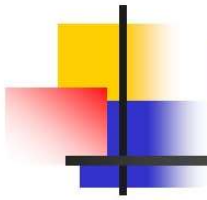
  L = { w | w ends in 01}

- ?
- Other examples
  - Keyword recognizer (e.g., if, then, else, while, for, include, etc.)
  - Strings where the first symbol is present somewhere later on at least once



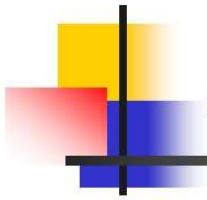
Extension of δ to NFA Paths

- Basis: $\hat{\delta}(q,\varepsilon) = \{q\}$
- Induction:
  - Let $\hat{\delta}(q_0,w) = \{p_1,p_2\ldots,p_k\}$
  - $\delta(p_i,a) = S_i$ for $i=1,2\ldots,k$
  - *Then,* $\hat{\delta}(q_0,wa) = S_1 \cup S_2 \cup \ldots \cup S_k$



Language of an NFA

- An NFA accepts *w* if *there exists at least one* path from the start state to an accepting (or final) state that is labeled by *w*
- $L(N) = \{ w \mid \hat{\delta}(q_0, w) \cap F \neq \Phi \}$



Advantages & Caveats for NFA

- Great for modeling regular expressions
  - String processing - e.g., grep, lexical analyzer
- Could a non-deterministic state machine be implemented in practice?
  - A parallel computer could exist in multiple “states” at the same time
  - Probabilistic models could be viewed as extensions of non-deterministic state machines (e.g., toss of a coin, a roll of dice)

But, DFAs and NFAs are equivalent in their power to capture langauges !!

Differences: DFA vs. NFA

- **DFA**

1. All transitions are deterministic
   - Each transition leads to exactly one state
2. For each state, transition on all possible symbols (alphabet) should be defined
3. Accepts input if the last state is in F
4. Sometimes harder to construct because of the number of states
5. Practical implementation is feasible

- **NFA**

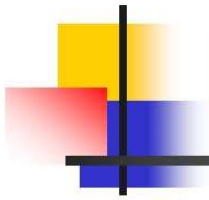
1. Some transitions could be non-deterministic
   - A transition could lead to a subset of states
2. Not all symbol transitions need to be defined explicitly (if undefined will go to a dead state – this is just a design convenience, not to be confused with "non-determinism")
3. Accepts input if *one of* the last states is in F
4. Generally easier than a DFA to construct
5. Practical implementation has to be deterministic (convert to DFA) or in the form of parallelism

Equivalence of DFA & NFA

- Theorem:
  - A language L is accepted by a DFA *if and only if* it is accepted by an NFA.

Should be true for any L

- Proof:
  1. If part:
     - Prove by showing every NFA can be converted to an equivalent DFA (in the next few slides…)
  2. Only-if part is trivial:
     - Every DFA is a special case of an NFA where each state has exactly one transition for every input symbol. Therefore, if L is accepted by a DFA, it is accepted by a corresponding NFA. □



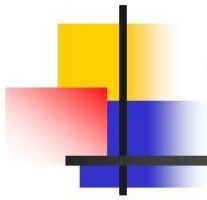
Proof for the if-part

- If-part: A language L is accepted by a DFA if it is accepted by an NFA
- rephrasing…
- Given any NFA N, we can construct a DFA D such that L(N)=L(D)

- How to convert an NFA into a DFA?
  - Observation: In an NFA, each transition maps to a *subset* of states
  - Idea: Represent:

    each "subset of NFA_states" ➔ a single "DFA_state"

Subset construction



NFA to DFA by subset construction

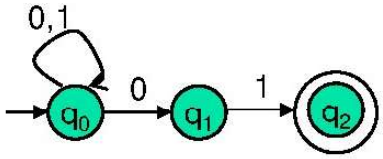
- Let $N = \{Q_N, \Sigma, \delta_N, q_0, F_N\}$
- Goal: Build $D=\{Q_D, \Sigma, \delta_D, \{q_0\}, F_D\}$ s.t. $L(D)=L(N)$
- Construction:
  1. Q_D= all subsets of Q_N (i.e., power set)
  2. F_D=set of subsets S of Q_N s.t. $S \cap F_N \neq \Phi$
  3. δ_D: for each subset S of Q_N and for each input symbol a in Σ:
     - $\delta_D(S,a) = \bigcup_{p \text{ in } s} \delta_N(p,a)$

Idea: To avoid enumerating all of power set, do "lazy creation of states"

NFA to DFA construction: Example

- *L = {w | w ends in 01}*

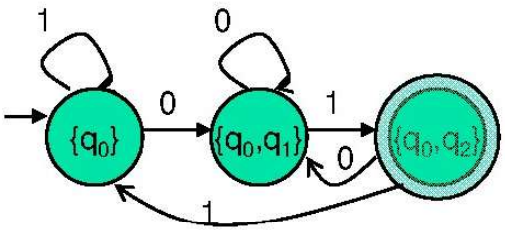
NFA:



δ_N	0	1
q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

DFA:

δ_D
~~$\emptyset$~~
$\{q_0\}$
~~$\{q_1\}$~~
~~$*\{q_2\}$~~
$\{q_0,q_1\}$
$*\{q_0,q_2\}$
~~$\{q_1,q_2\}$~~
~~$*\{q_0,q_1,q_2\}$~~



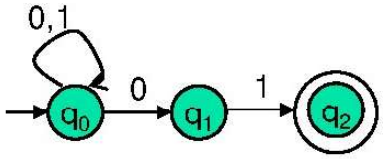
δ_D	0	1
$\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$*\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

0. Enumerate all possible subsets
1. Determine transitions
2. Retain only those states reachable from $\{q_0\}$

NFA to DFA: Repeating the example using *LAZY CREATION*

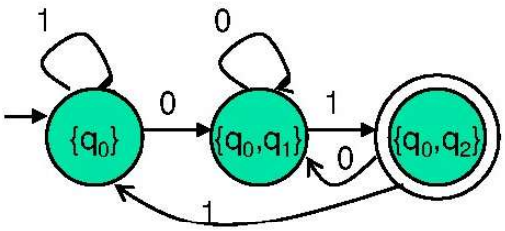
- *L = {w | w ends in 01}*

NFA:



δ_N	0	1
→ q_0	$\{q_0,q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

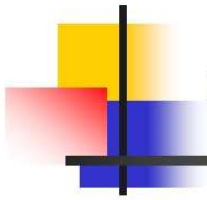
DFA:



δ_D	0	1
→ $\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$

Main Idea:

Introduce states as you go (on a need basis)



Correctness of subset construction

Theorem: If D is the DFA constructed from NFA N by subset construction, then L(D)=L(N)

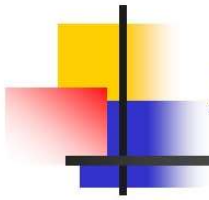
- Proof:
  - Show that $\hat{\delta}_D(\{q_0\},w) \equiv \hat{\delta}_N(q_0,w\}$, for all w
  - Using induction on w's length:
    - Let w = xa
    - $\hat{\delta}_D(\{q_0\},xa) \equiv \delta_D(\hat{\delta}_N(q_0,x\}, a) \equiv \hat{\delta}_N(q_0,w\}$

A bad case where #states(DFA)>>#states(NFA)

- L = {w | w is a binary string s.t., the k^{th} symbol from its end is a 1}
  - NFA has k+1 states
  - But an equivalent DFA needs to have at least 2^k states

(Pigeon hole principle)

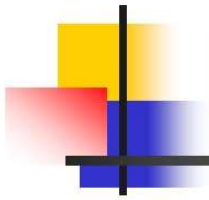
- *m* holes and >*m* pigeons
  - => at least one hole has to contain two or more pigeons



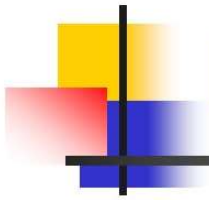
Applications

- Text indexing
  - inverted indexing
  - For each unique word in the database, store all locations that contain it using an NFA or a DFA
- Find pattern P in text T
  - Example: Google querying
- Extensions of this idea:
  - PATRICIA tree, suffix tree

A few subtle properties of DFAs and NFAs



- The machine never really terminates.
  - It is always waiting for the next input symbol or making transitions.
- The machine decides when to consume the next symbol from the input and when to ignore it.
  - (but the machine can never skip a symbol)
- => A transition can happen even *without* really consuming an input symbol (think of consuming ε as a free token)
- A single transition cannot consume more than one symbol.



FA with ε-Transitions

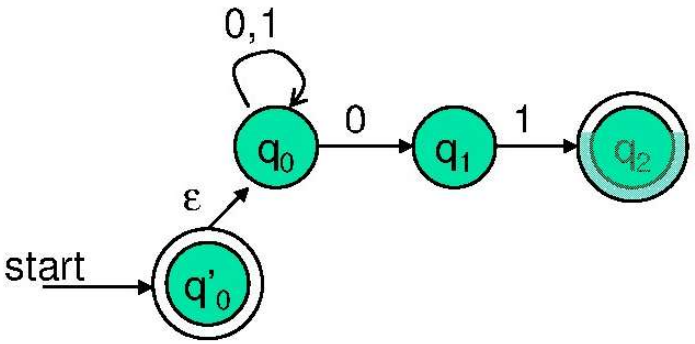
- We can allow <u>explicit</u> ε-transitions in finite automata
  - i.e., a transition from one state to another state without consuming any additional input symbol
  - Makes it easier sometimes to construct NFAs

<u>Definition:</u> ε -NFAs are those NFAs with at least one explicit ε-transition defined.

- ε -NFAs have one more column in their transition table

Example of an ε-NFA

L = {w | w is empty, or if non-empty will end in 01}



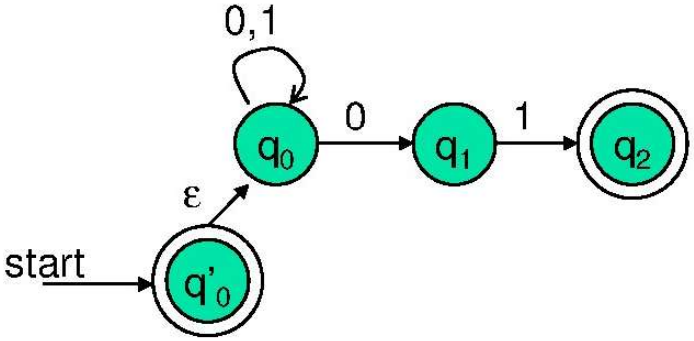
δ_E	0	1	ε	
→ $*q'_0$	∅	∅	$\{q'_0, q_0\}$	ECLOSE(q'_0)
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$	ECLOSE(q_0)
q_1	∅	$\{q_2\}$	$\{q_1\}$	ECLOSE(q_1)
$*q_2$	∅	∅	$\{q_2\}$	ECLOSE(q_2)

- ε-closure of a state q, ***ECLOSE(q)***, is the set of all states (including itself) that can be *reached* from q by repeatedly making an arbitrary number of ε-transitions.

To simulate any transition:
Step 1) Go to all immediate destination states.
Step 2) From there go to all their ε-closure states as well.

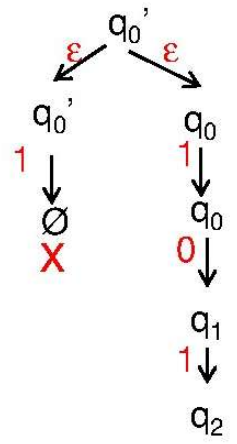
Example of an ε-NFA

L = {w | w is empty, or if non-empty will end in 01}



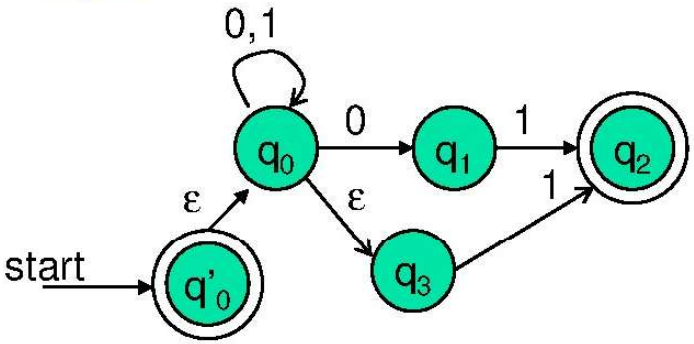
δ_E	0	1	ε	
→ $*q'_0$	∅	∅	$\{q'_0, q_0\}$	ECLOSE(q'_0)
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0\}$	ECLOSE(q_0)
q_1	∅	$\{q_2\}$	$\{q_1\}$	
$*q_2$	∅	∅	$\{q_2\}$	

Simulate for w=101:



To simulate any transition:
Step 1) Go to all immediate destination states.
Step 2) From there go to all their ε-closure states as well.

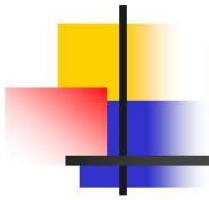
Example of another ε-NFA



δ_E	0	1	ε
→ $*q'_0$	Ø	Ø	$\{q'_0, q_0, q_3\}$
q_0	$\{q_0, q_1\}$	$\{q_0\}$	$\{q_0, q_3\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$*q_2$	Ø	Ø	$\{q_2\}$
q_3	Ø	$\{q_2\}$	$\{q_3\}$

Simulate for w=101:

?



Equivalency of DFA, NFA, ε-NFA

- Theorem: A language L is accepted by some ε-NFA if and only if L is accepted by some DFA

- Implication:
  - DFA ≡ NFA ≡ ε-NFA
  - (all accept Regular Languages)

Eliminating ε-transitions

Let $E = \{Q_E, \Sigma, \delta_E, q_0, F_E\}$ be an ε-NFA

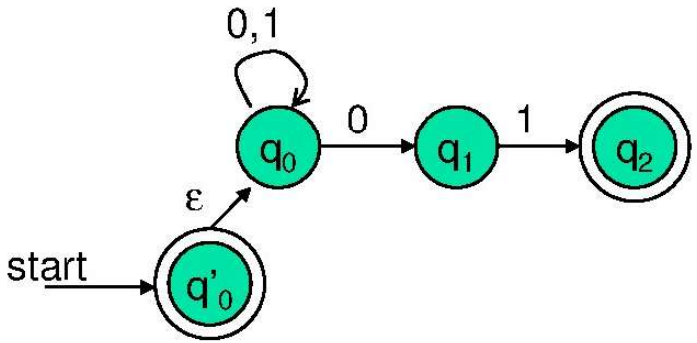
Goal: To build DFA $D=\{Q_D, \Sigma, \delta_D, \{q_D\}, F_D\}$ s.t. $L(D)=L(E)$

Construction:

1. Q_D= all reachable subsets of Q_E factoring in ε-closures
2. $q_D = \text{ECLOSE}(q_0)$
3. F_D=subsets S in Q_D s.t. $S \cap F_E \neq \Phi$
4. δ_D: for each subset S of Q_E and for each input symbol $a \in \Sigma$:
   - Let $R = \bigcup_{p \text{ in } s} \delta_E(p,a)$ // go to destination states
   - $\delta_D(S,a) = \bigcup_{r \text{ in } R} \text{ECLOSE}(r)$ // from there, take a union of all their ε-closures

Example: ε-NFA ➔ DFA

L = {w | w is empty, or if non-empty will end in 01}

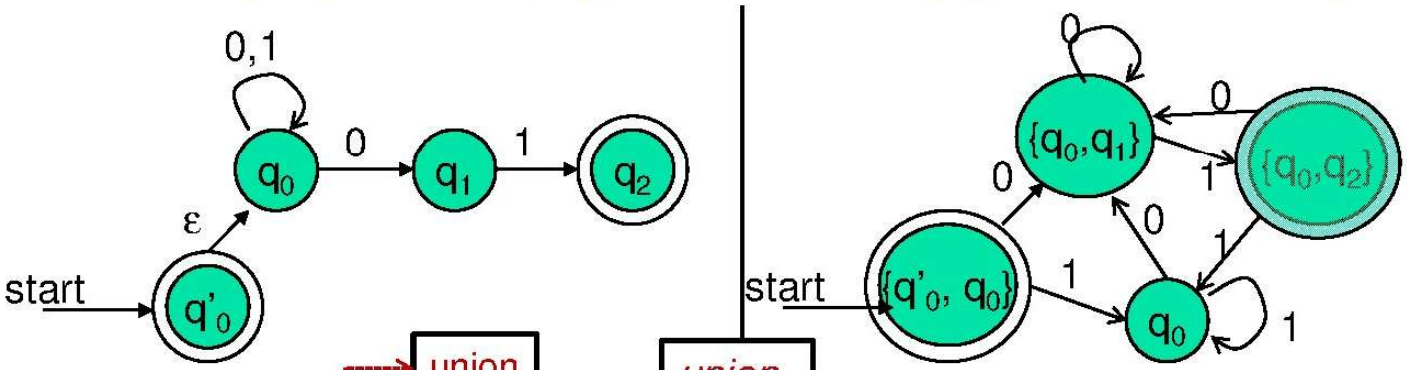


δ_E	0	1	ε
→ $*q'_0$	Ø	Ø	$\{q'_0,q_0\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	Ø	$\{q_2\}$	$\{q_1\}$
$*q_2$	Ø	Ø	$\{q_2\}$

δ_D	0	1
→ $*\{q'_0,q_0\}$		
…		

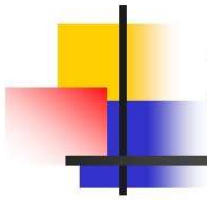
Example: ε-NFA ➔ DFA

L = {w | w is empty, or if non-empty will end in 01}



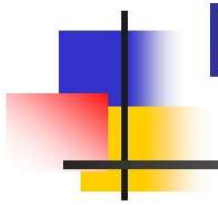
δ_E	0	1	ε
→ *q'_0	∅	∅	$\{q'_0,q_0\}$
q_0	$\{q_0,q_1\}$	$\{q_0\}$	$\{q_0\}$
q_1	∅	$\{q_2\}$	$\{q_1\}$
*q_2	∅	∅	$\{q_2\}$

δ_D	0	1
→ *$\{q'_0,q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$\{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
*$\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$

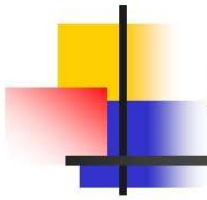


Summary

- DFA
  - Definition
  - Transition diagrams & tables
- Regular language
- NFA
  - Definition
  - Transition diagrams & tables
- DFA vs. NFA
- NFA to DFA conversion using subset construction
- Equivalency of DFA & NFA
- Removal of redundant states and including dead states
- ε-transitions in NFA
- Pigeon hole principles
- Text searching applications



Equivalence & Minimization of DFAs



Applications of interest

- Comparing two DFAs:
  - $L(DFA_1) == L(DFA_2)$?

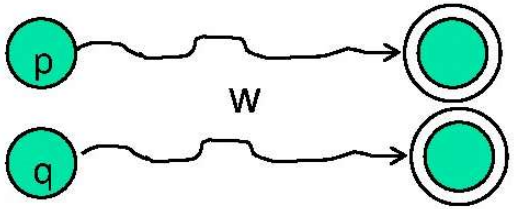
- How to minimize a DFA?
  1. Remove unreachable states
  2. Identify & condense equivalent states into one

When to call two states in a DFA "equivalent"?

Past doesn't matter - only future does!

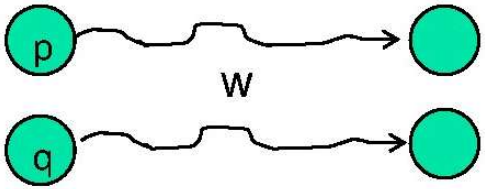
Two states p and q are said to be *equivalent* iff:

i) Any string w accepted by starting at p is also accepted by starting at q;



AND

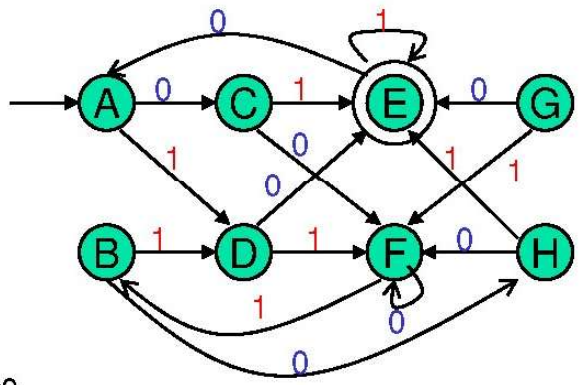
i) Any string w rejected by starting at p is also rejected by starting at q.



➔ p≡q

Computing equivalent states in a DFA

Table Filling Algorithm



A	=							
B	=	=						
C	x	x	=					
D	x	x	x	=				
E	x	x	x	x	=			
F	x	x	x	x	x	=		
G	x	x	x	=	x	x	=	
H	x	x	=	x	x	x	x	=
	A	B	C	D	E	F	G	H

Pass #0

1. Mark accepting states ≠ non-accepting states

Pass #1

1. Compare every pair of states
2. Distinguish by one symbol transition
3. Mark = or ≠ or blank(tbd)

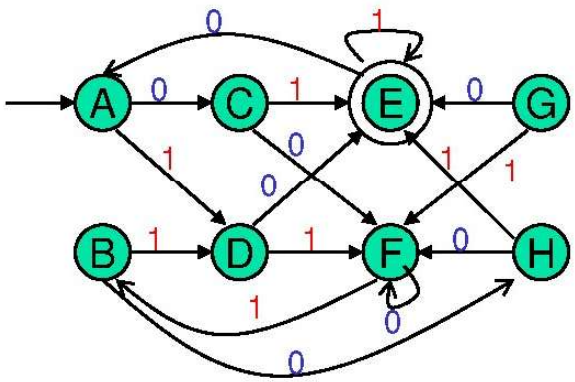
Pass #2

1. Compare every pair of states
2. Distinguish by up to two symbol transitions (until different or same or tbd)

....

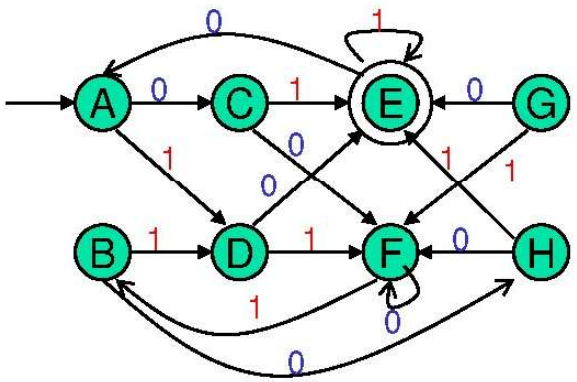
(keep repeating until table complete)

Table Filling Algorithm - step by step



	A	B	C	D	E	F	G	H
A	=							
B		=						
C			=					
D				=				
E					=			
F						=		
G							=	
H								=

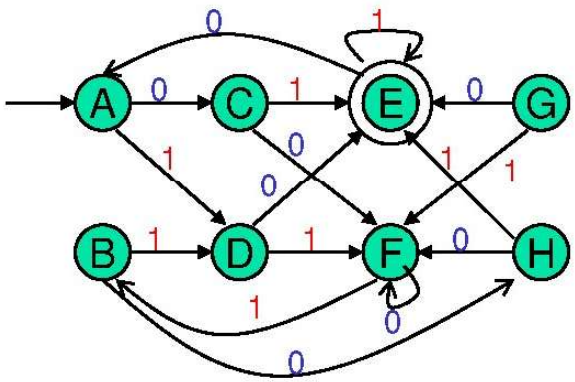
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state

	A	B	C	D	E	F	G	H
A	=							
B		=						
C			=					
D				=				
E	X	X	X	X	=			
F					X	=		
G					X		=	
H					X			=

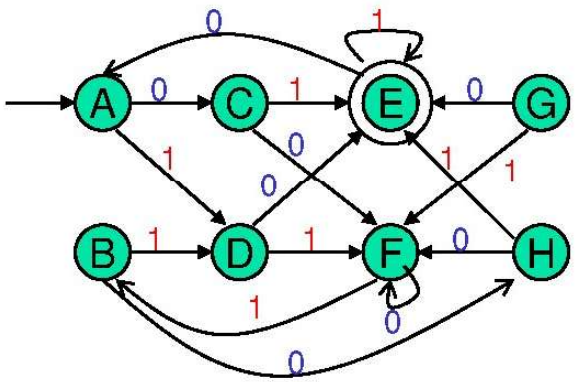
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

	A	B	C	D	E	F	G	H
A	=							
B		=						
C	X		=					
D	X			=				
E	X	X	X	X	=			
F					X	=		
G	X				X		=	
H	X				X			=

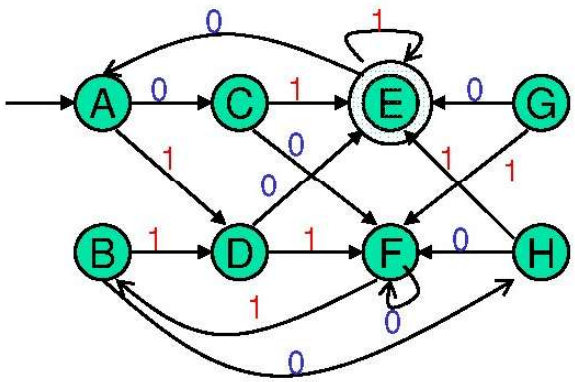
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

	A	B	C	D	E	F	G	H
A	=							
B		=						
C	X	X	=					
D	X	X		=				
E	X	X	X	X	=			
F					X	=		
G	X	X			X		=	
H	X	X			X			=

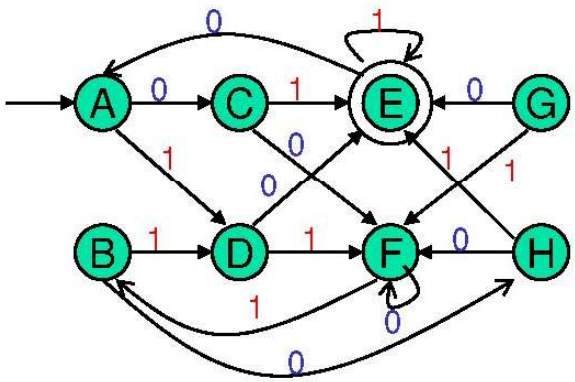
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

	A	B	C	D	E	F	G	H
A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X		X	=		
G	X	X	X		X		=	
H	X	X	=		X			=

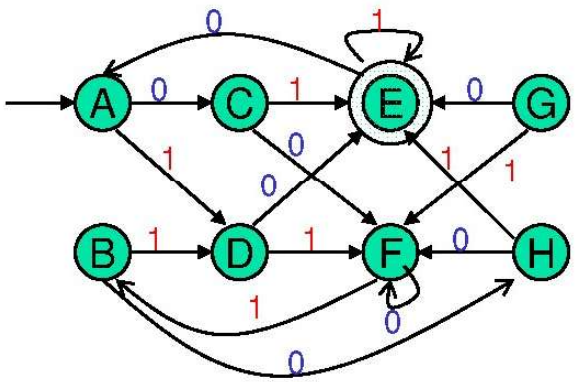
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X		=	
H	X	X	=	X	X			=
	A	B	C	D	E	F	G	H

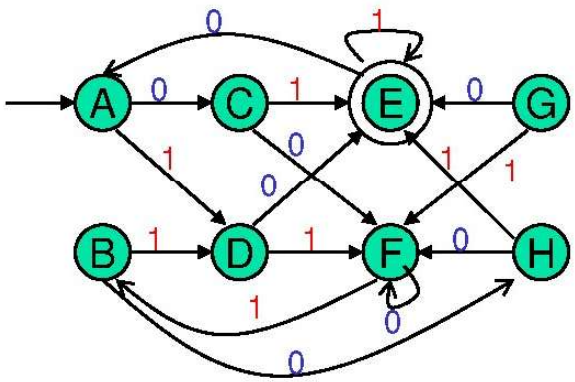
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

	A	B	C	D	E	F	G	H
A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X		=

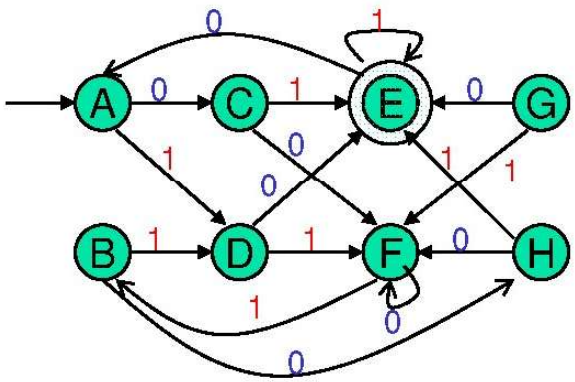
Table Filling Algorithm - step by step



1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings

A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F			X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=
	A	B	C	D	E	F	G	H

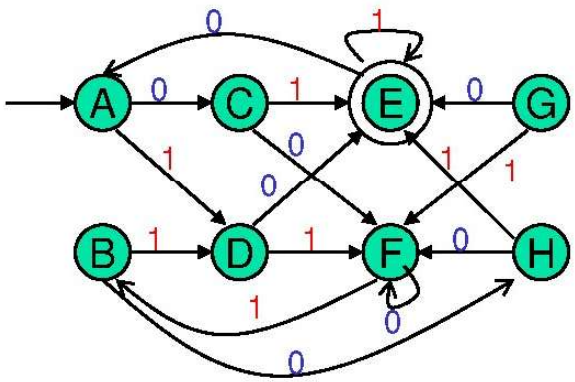
Table Filling Algorithm - step by step



A	=							
B	**=**	=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F	**X**	**X**	X	X	X	=		
G	X	X	X	**=**	X	X	=	
H	X	X	**=**	X	X	X	X	=
	A	B	C	D	E	F	G	H

1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings
3. Look 2-hops away for distinguishing states or strings

Table Filling Algorithm - step by step

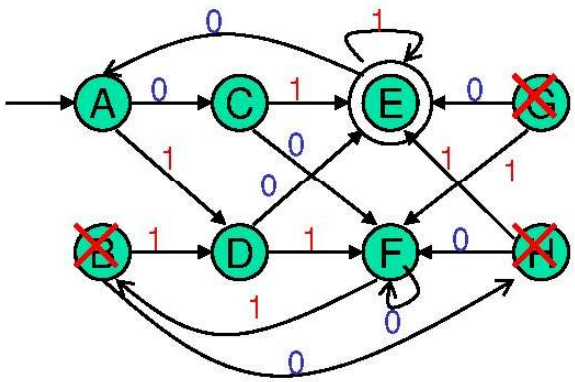


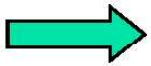
1. Mark X between accepting vs. non-accepting state
2. Look 1- hop away for distinguishing states or strings
3. Look 2-hops away for distinguishing states or strings

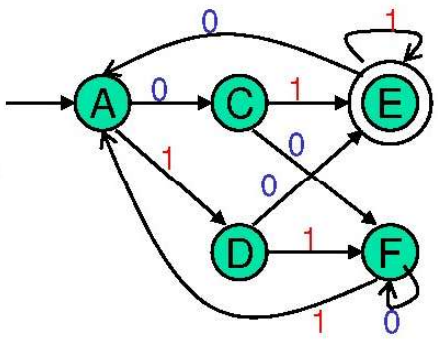
	A	B	C	D	E	F	G	H
A	=							
B		=						
C	X	X	=					
D	X	X	X	=				
E	X	X	X	X	=			
F	X	X	X	X	X	=		
G	X	X	X	=	X	X	=	
H	X	X	=	X	X	X	X	=

Equivalences:
- A=B
- C=H
- D=G

Table Filling Algorithm - step by step



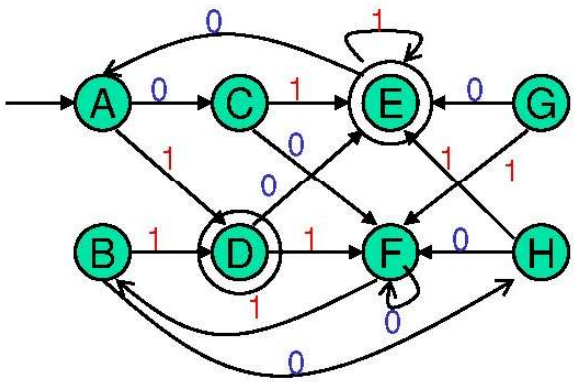




Retrain only one copy for each equivalence set of states

Equivalences:
- A=B
- C=H
- D=G

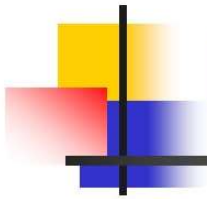
Table Filling Algorithm – special case



	A	B	C	D	E	F	G	H
A	=							
B		=						
C			=					
D				=				
E				?	=			
F						=		
G							=	
H								=

Q) What happens if the input DFA has more than one final state?
Can all final states initially be treated as equivalent to one another?

Putting it all together …



How to minimize a DFA?

- Goal: Minimize the number of states in a DFA
- Algorithm:

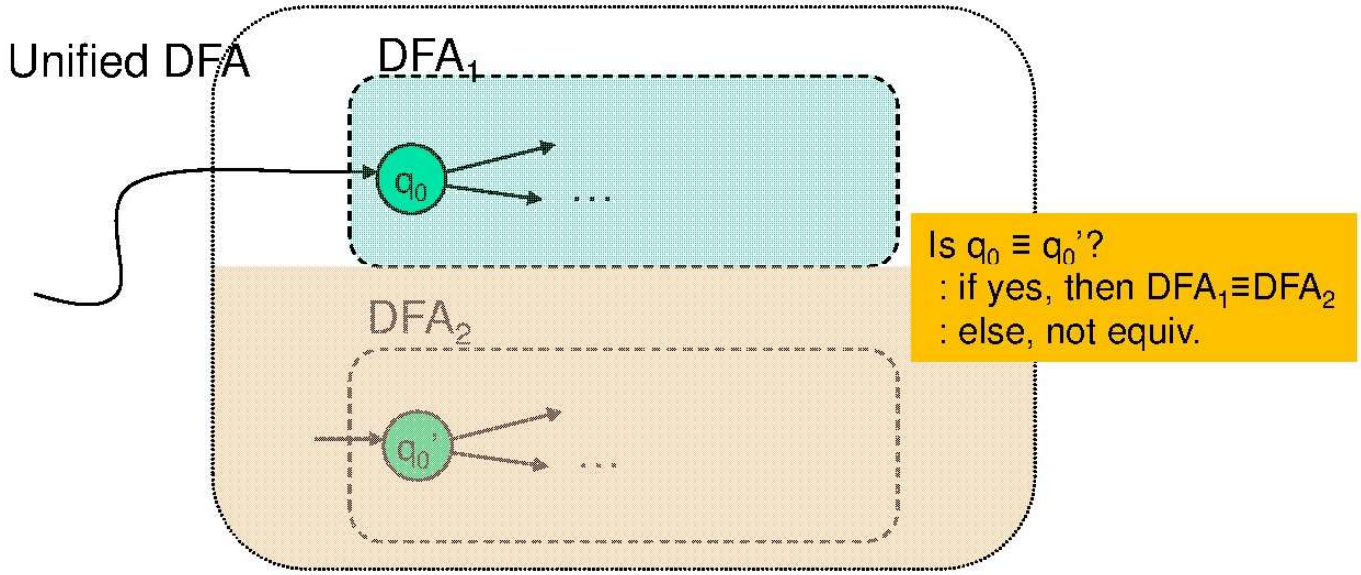
Depth-first traversal from the start state

1. Eliminate states unreachable from the start state

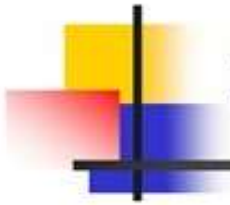
Table filling algorithm

2. Identify and remove equivalent states
3. Output the resultant DFA

Are Two DFAs Equivalent?



1. Make a new dummy DFA by just putting together both DFAs
2. Run table-filling algorithm on the unified DFA
3. *IF* the start states of both DFAs are found to be equivalent,
   THEN: $DFA_1 \equiv DFA_2$
   ELSE: different



Summary

- Simplification of DFAs
  - How to remove unreachable states?
  - How to identify and collapse equivalent states?
  - How to minimize a DFA?
  - How to tell whether two DFAs are equivalent?