Mumbai University — NEP 2020

Computational Theory

Module Notes

Finite Automata & Finite State Machines

DFA · NFA · ε-NFA · Moore & Mealy Machines · DFA Minimization

Table of Contents

1. Introduction to Finite Automata & Finite State Machines

Finite Automata (FA) are the simplest computational models in the Chomsky hierarchy. They serve as the mathematical foundation for understanding how machines can recognize patterns and languages. FA are used extensively in lexical analysis, pattern matching, network protocols, and digital circuit design.

Definition: Finite Automaton

A Finite Automaton (FA) is an abstract machine that reads an input string one symbol at a time, transitions between a finite number of states according to a transition function, and either **accepts** or **rejects** the string based on whether it ends in an accepting (final) state.

1.1 Finite State Machine (FSM)

A Finite State Machine (FSM) is the broader concept encompassing any machine with a finite number of states. It can be either a **recognizer/acceptor** (decides if input belongs to a language) or **transducer** (produces output — Moore/Mealy machines). Every FA is an FSM, but FSMs also include output-generating machines.

Key Intuition:

Think of a traffic light controller — it has a finite number of states (Red, Green, Yellow) and transitions between them based on timers or sensors. It has no 'memory' of past states beyond its current state. This is exactly what a Finite Automaton models.

1.2 Formal Language Theory Connection

Grammar Type	Automaton	Language Class
Type 0 (Unrestricted)	Turing Machine	Recursively Enumerable
Type 1 (Context-Sensitive)	Linear Bounded Automaton	Context-Sensitive
Type 2 (Context-Free)	Pushdown Automaton	Context-Free
Type 3 (Regular)	Finite Automaton	Regular Languages ← Our Focus

Note: Finite Automata recognize exactly the class of Regular Languages (Type 3 in Chomsky Hierarchy).

2. Deterministic Finite Automata (DFA)

Definition: Formal Definition — DFA

A DFA is a 5-tuple **M = (Q, Σ, δ, q_0, F)** where:
- **Q** — finite, non-empty set of states
- Σ — finite input alphabet
- **δ: Q × Σ → Q** — transition function (total function)
- **q_0 ∈ Q** — initial/start state
- **F ⊆ Q** — set of accept/final states

Key Property of DFA: From any state, on reading any symbol, there is *exactly one* next state — hence **deterministic**. The transition function δ is total, meaning it is defined for every (state, symbol) pair.

2.1 Transition Diagram

A transition diagram (state diagram) is a directed graph representing a DFA:
- **Nodes (circles)** represent states
- **Double circles** represent final/accepting states
- **Arrow (→)** into a state indicates the start state
- **Directed edges** labeled with input symbols represent transitions

Example DFA: Accepts strings over {0,1} ending in '1'

Transition Table: M = ({q0, q1}, {0, 1}, δ, q0, {q1})

State	On 0	On 1	Final?
→ q0 (start)	q0	q1	No
* q1	q0	q1	Yes

Transition Diagram Description:
- State q0 (start): on '0' → q0 (self-loop), on '1' → q1
- State q1 (accept): on '0' → q0, on '1' → q1 (self-loop)

The machine stays in q0 while reading 0s, moves to q1 on reading 1, and the last symbol must be 1 to end in q1.

2.2 Extended Transition Function (δ*)

The extended transition function δ*: Q × Σ* → Q processes entire strings:
- **δ*(q, ε) = q** (on empty string, stay in current state)
- **δ*(q, wa) = δ(δ*(q, w), a)** for string w and symbol a ∈ Σ

A string w is **accepted** by DFA M if δ*(q0, w) ∈ F.
The **language of M**: L(M) = { w ∈ Σ* | δ*(q0, w) ∈ F }

2.3 Worked Example — DFA Design

Problem: Design a DFA over Σ = {a, b} that accepts strings containing the substring 'ab'.

States and Logic:

• **q0** (start): Haven't seen 'a' yet

• **q1**: Last symbol was 'a' (potential start of 'ab')

• **q2** (accept): Substring 'ab' has been seen

Transitions:

State	On 'a'	On 'b'	Role
→ q0	q1	q0	Start: no 'a' seen
q1	q1	q2	Last char was 'a'
* q2	q2	q2	Accept: 'ab' found

Trace 'bab': q0 →(b) q0 →(a) q1 →(b) q2 ∈ F → ACCEPTED ✓

Trace 'ba': q0 →(b) q0 →(a) q1 ∉ F → REJECTED ✗

3. Nondeterministic Finite Automata (NFA)

Definition: Formal Definition — NFA

An NFA is a 5-tuple **N = (Q, Σ, δ, q_0, F)** where:
- **Q, Σ, q_0, F** are same as DFA
- **δ: Q × Σ → 2^Q** — transition function returns a *set* of states
- From a state, on reading a symbol, the machine can go to **zero, one, or more** states
- A string is accepted if **at least one** computation path leads to an accept state

3.1 DFA vs NFA Comparison

Feature	DFA	NFA
Transition function	δ: Q × Σ → Q	δ: Q × Σ → 2Q
Next states	Exactly one	Zero, one, or more
Empty string (ε) transitions	Not allowed	Allowed (ε-NFA)
Acceptance condition	Unique path ends in F	Any path ends in F
Implementation	Direct (easy)	Simulation needed
Expressive power	Equal to NFA	Equal to DFA
State complexity	May need more states	Often fewer states

3.2 NFA Example

NFA Example: Accepts strings over {0,1} ending in '01'.

State	On 0	On 1	Final?
→ q0	{q0, q1}	{q0}	No
q1	∅	{q2}	No
* q2	∅	∅	Yes

Trace '001': {q0} →(0) {q0,q1} →(0) {q0,q1} →(1) {q0,q2}. q2 ∈ F → ACCEPTED ✓
Trace '01': {q0} →(0) {q0,q1} →(1) {q0,q2}. q2 ∈ F → ACCEPTED ✓
Trace '10': {q0} →(1) {q0} →(0) {q0,q1}. Neither q0 nor q1 ∈ F → REJECTED ✗

4. ε-NFA and Equivalence of FA Models

Definition: ε-NFA (NFA with ε-transitions)

An ε-NFA is an NFA where the transition function is δ: **$Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$**. It allows transitions on the **empty string** ε — the machine moves to another state *without consuming any input symbol*. This adds convenience but not power.

4.1 ε-Closure

The **ε-closure** of a state q (written ε-CLOSURE(q) or ECLOSE(q)) is the set of all states reachable from q using only ε-transitions (including q itself).

Formal Definition:

- $q \in$ ε-CLOSURE(q) (always includes itself)
- If $p \in$ ε-CLOSURE(q) and $\delta(p, \varepsilon)$ contains r, then $r \in$ ε-CLOSURE(q)
- ε-CLOSURE(S) = $\cup$ { ε-CLOSURE(q) | $q \in$ S } for a set S of states

4.2 ε-NFA Example

Example: ε-NFA accepting (0+1)*1(0+1) — strings whose second-to-last symbol is 1:

State	On ε	On 0	On 1
→ q0	{q1}	∅	∅
q1	∅	{q1}	{q1, q2}
q2	∅	{q3}	{q3}
* q3	∅	∅	∅

4.3 Equivalence of FA Models

All three models — DFA, NFA, and ε-NFA — are **equivalent in power**. They all recognize exactly the class of **Regular Languages**. The following conversions exist:

From	To	Method	State Blowup
ε-NFA	NFA	Eliminate ε-transitions using ε-closure	None
NFA	DFA	Subset Construction Algorithm	Exponential (2■ worst case)
DFA	NFA	DFA is a special case of NFA	None
ε-NFA	DFA	Combine both conversions	Exponential (worst case)

5. NFA to DFA Conversion — Subset Construction

The **Subset Construction Algorithm** (also called powerset construction) converts any NFA to an equivalent DFA. Each DFA state corresponds to a *subset* of NFA states representing all states the NFA could be in simultaneously.

5.1 Algorithm Steps

Step	Action
1	Compute ε-CLOSURE(q0) — this is the DFA start state
2	For each DFA state S (a set of NFA states) and each symbol a ∈ Σ:
	Compute MOVE(S, a) = ∪ { δ(q, a) | q ∈ S }
	Compute ε-CLOSURE(MOVE(S, a)) — this is the next DFA state
3	Repeat step 2 for all new DFA states until no new states are generated
4	A DFA state is accepting if it contains at least one NFA accepting state

5.2 Worked Example

Given NFA N = ({p, q, r}, {0, 1}, δ, p, {r}): Accepts strings ending in '01'

NFA State	On 0	On 1
→ p	{p, q}	{p}
q	∅	{r}
* r	∅	∅

Subset Construction:

Start state: {p} (no ε-transitions, so ε-CLOSURE({p}) = {p})

Process {p}: on 0 → {p,q}, on 1 → {p}
Process {p,q}: on 0 → {p,q} ∪ ∅ = {p,q}, on 1 → {p} ∪ {r} = {p,r}
Process {p,r}: on 0 → {p,q} ∪ ∅ = {p,q}, on 1 → {p} ∪ ∅ = {p}

DFA State	On 0	On 1	Accept? (contains r)
→ A = {p}	B = {p,q}	A = {p}	No
B = {p,q}	B = {p,q}	C = {p,r}	No
* C = {p,r}	B = {p,q}	A = {p}	Yes (contains r)

Result: DFA has 3 states {A, B, C}. The NFA had 3 states — in worst case, n NFA states → up to 2■ DFA states.

6. Minimization of DFA

DFA minimization produces the **unique minimum-state DFA** for a given regular language. The minimized DFA has the fewest possible states and is unique (up to state renaming). The algorithm works by identifying and merging **equivalent (indistinguishable)** states.

Definition: Distinguishable States

Two states p and q are **distinguishable** if there exists some string w such that exactly one of δ*(p,w) and δ*(q,w) is an accepting state. States that are NOT distinguishable are called **equivalent** and can be merged.

6.1 Table-Filling Algorithm (Myhill-Nerode)

Step 1	Remove all unreachable states from the DFA.
Step 2	Create a table of all pairs of states (p, q) where p ≠ q.
Step 3	Mark pair (p,q) as distinguishable if one is final and the other is not.
Step 4	Repeat: For each unmarked pair (p,q) and each symbol a, if the pair (δ(p,a), δ(q,a)) is already marked, then mark (p,q). Continue until no new pairs are marked.
Step 5	All unmarked pairs are equivalent. Merge equivalent states.

6.2 Minimization Example

Given DFA with states {A, B, C, D, E, F}, Σ = {0, 1}, start = A, Final = {C, D, E}:

State	On 0	On 1	Final?
→ A	B	C	No
B	A	D	No
* C	E	F	Yes
* D	F	E	Yes
* E	E	F	Yes
F	F	F	No

Applying Table-Filling:

- **Base:** Mark (Final, Non-Final) pairs: (C,A), (C,B), (C,F), (D,A), (D,B), (D,F), (E,A), (E,B), (E,F) — marked.
- **Check (A,B):** δ(A,0)=B, δ(B,0)=A → (A,B); δ(A,1)=C, δ(B,1)=D → (C,D) unmarked → don't mark yet.
- **Check (C,D):** δ(C,0)=E, δ(D,0)=F → (E,F) is marked → mark (C,D).
- **Check (C,E):** δ(C,0)=E, δ(E,0)=E → (E,E) same state; δ(C,1)=F, δ(E,1)=F → (F,F) same → NOT marked → C≡E.

- **Check (A,B):** $\delta(A,1)=C$, $\delta(B,1)=D \rightarrow (C,D)$ now marked $\rightarrow$ mark (A,B).
- **Result:** Equivalent pairs: {C,E}. Merge C and E $\rightarrow$ Minimized DFA has 5 states: {A, B, {C,E}, D, F}.

7. FSM with Output: Moore and Mealy Machines

Unlike DFA/NFA which only accept or reject strings, **Moore** and **Mealy** machines are FSMs that **produce output** for each input. They are called **transducers**.

7.1 Moore Machine

Definition: Moore Machine

A Moore Machine is a 6-tuple $\mathbf{M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)}$ where:

- **Q** — finite set of states
- Σ — input alphabet
- Δ — output alphabet
- δ**: Q** $\times \Sigma \rightarrow$ **Q** — transition function
- λ**: Q** $\rightarrow \Delta$ — output function (output depends only on *current state*)
- $\mathbf{q_0}$ — start state

Key: Output is associated with STATES. Output is produced when entering a state.

Moore Machine Example: Detect sequence '101' and output 1 when found.

State	Output	On 0	On 1
→ q0	0	q0	q1
q1	0	q2	q1
q2	0	q0	q3
q3	1	q2	q1

For input '101': q0→(1)q1→(0)q2→(1)q3. Output at each state visited = 0,0,0,1. Output string = '0001'. Length = input length + 1 (due to initial state output).

7.2 Mealy Machine

Definition: Mealy Machine

A Mealy Machine is a 6-tuple $\mathbf{M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)}$ where:

- **Q**, Σ, Δ, δ, $\mathbf{q_0}$ — same as Moore machine
- λ**: Q** $\times \Sigma \rightarrow \Delta$ — output function (output depends on *current state AND input*)

Key: Output is associated with TRANSITIONS (edges). Output is produced when a transition is taken.

Mealy Machine Example: Outputs 1 when last two inputs were '01', else 0.

State	On 0 / Output	On 1 / Output

→ q0	q1 / 0	q0 / 0
q1	q1 / 0	q2 / 1
q2	q1 / 0	q0 / 0

For input '0101': q0→(0/0)q1→(1/1)q2→(0/0)q1→(1/1)q2. Output = '0101'. Output length = Input length.

7.3 Moore vs Mealy Comparison

Feature	Moore Machine	Mealy Machine
Output depends on	State only	State and Input
Output function	$\lambda: Q \to \Delta$	$\lambda: Q \times \Sigma \to \Delta$
Output length	Input length + 1	Equal to input length
States needed	Generally more	Generally fewer
Output timing	On entering state	On transition
Equivalence	Can be converted to Mealy	Can be converted to Moore
Use case	Simpler to design, vending machines	Serial adders, code converters

8. Applications and Limitations of Finite Automata

8.1 Applications of Finite Automata

Lexical Analysis (Compilers)	FA form the core of lexers (tokenizers) in compilers. Regular expressions describing tokens (identifiers, keywords, numbers) are converted to DFAs. Tools like 'lex' and 'flex' implement this.
Pattern Matching & Text Search	grep, sed, and regex engines use FA. The Knuth-Morris-Pratt (KMP) algorithm and Aho-Corasick algorithm are FA-based string matching algorithms.
Network Protocols	Packet filtering, stateful firewalls, and intrusion detection systems (IDS) like Snort use FA to describe protocol state machines and detect patterns in network traffic.
Digital Circuit Design	Sequential logic circuits (flip-flops, registers, sequence detectors) are directly modeled as FSMs. Moore/Mealy machines map to hardware implementations.
Vending Machines & Elevators	Control logic for embedded systems with finite states (e.g., vending machine accepting coins, elevator control) is naturally modeled as FSMs.
Natural Language Processing	Morphological analyzers for word form recognition, tokenizers, and finite-state transducers for translation are widely used in NLP.
Software Verification	Model checking tools use FA to verify that software/hardware systems satisfy safety properties by exploring all reachable states.

8.2 Limitations of Finite Automata

Finite Automata have fundamental limitations arising from their **lack of memory** (they can only remember the current state, not the history of computation):

Cannot count unboundedly	Cannot recognize L = {a■b■ | n ≥ 1} because counting requires unbounded memory. FA cannot match an arbitrary number of a's with the same number of b's.

Cannot check palindromes	$L = \{ww^R \mid w \in \{a,b\}^*\}$ requires remembering the first half to compare with the second half — impossible with finite memory.
Cannot match brackets	Balanced parentheses $L = \{(■)■ \mid n \geq 0\}$ is a context-free language, not regular. FA cannot solve this.
Pumping Lemma	The Pumping Lemma for regular languages provides a formal tool to prove that certain languages are NOT regular (i.e., FA cannot recognize them).
No stack/tape	FA cannot simulate a stack or tape. These require Pushdown Automata or Turing Machines respectively.

8.3 Pumping Lemma for Regular Languages

Definition: Pumping Lemma

If L is a regular language, then there exists a constant p (the pumping length) such that for every string $w \in L$ with $|w| \geq p$, w can be divided into three parts $w = xyz$ satisfying:

- $|xy| \leq p$
- $|y| \geq 1$ (y is non-empty)
- For all $i \geq 0$, $xy^iz \in L$

To prove a language is NOT regular: assume it is regular, apply the lemma, and derive a contradiction.

Example: Prove $L = \{0■1■ \mid n \geq 1\}$ is not regular.

Assume L is regular with pumping length p. Choose $w = 0^p1^p \in L$, $|w| = 2p \geq p$.

By the lemma, $w = xyz$ with $|xy| \leq p$, $|y| \geq 1$.

Since $|xy| \leq p$, both x and y consist entirely of 0s. So $y = 0^k$ for some $k \geq 1$.

Pump: $xy^2z = 0^{p+k}1^p$ — this has more 0s than 1s $\rightarrow \notin L$. Contradiction! Therefore L is NOT regular. ✓

Quick Revision — Important Points & Formulae

Concept	Key Fact
DFA Formal Definition	5-tuple: (Q, Σ, δ, q0, F) | δ: Q × Σ → Q (total)
NFA Formal Definition	5-tuple: (Q, Σ, δ, q0, F) | δ: Q × Σ → 2Q
ε-NFA	δ: Q × (Σ ∪ {ε}) → 2Q | ε-moves allowed
ε-CLOSURE	All states reachable via ε-transitions (includes itself)
Language of DFA	L(M) = { w | δ*(q0, w) ∈ F }
NFA Acceptance	String accepted if ANY path leads to final state
NFA → DFA	Subset Construction (Powerset): |DFA states| ≤ 2|NFA states|
DFA Minimization	Table-filling algorithm; merge indistinguishable states
Moore Output	λ: Q → Δ | Output depends on STATE only
Mealy Output	λ: Q × Σ → Δ | Output depends on STATE + INPUT
Moore output length	|Input| + 1 (initial state also produces output)
Mealy output length	|Input| (output on each transition)
Pumping Length p	For regular L: ∀|w| ≥ p, ∃ pumping decomposition w=xyz
Non-regular languages	a■b■, palindromes, balanced parentheses, wwr
Regular = FA recognizable	DFA ≡ NFA ≡ ε-NFA ≡ Regular Grammar ≡ Regular Expression

Sample Exam Questions (Mumbai University Pattern)

1. Define DFA formally. Design a DFA over {a,b} that accepts all strings with an even number of a's. Draw its transition diagram and table.

2. Convert the following NFA to a DFA using subset construction. [NFA given in exam]

3. Explain ε-closure with example. Describe how an ε-NFA is converted to NFA.

4. Minimize the given DFA using the table-filling algorithm. Show all steps.

5. Differentiate Moore and Mealy machines. Design a Mealy machine that outputs 1 whenever the input sequence contains '11'.

6. State and prove the Pumping Lemma for regular languages. Use it to show that L = {a■b■c■ | n ≥ 1} is not regular.

7. Discuss applications and limitations of finite automata with suitable examples.

8. Prove that NFA and DFA are equivalent in expressive power.

Note: *These notes cover the complete Finite Automata module as per Mumbai University NEP 2020 Computational Theory syllabus. Practice designing automata and converting between models for exam success. Always show all steps in conversion/minimization problems.*