Contents

1. INTRODUCTION TO CONCEPT OF AUTOMATA 1–21

1.1. Historical Aspect of Automata 1
1.2. The Study of Automata Theory is Fruitful and Futuristic 2
1.3. Develop Your Feelings With The Automata 2
1.4. Introduction to Sets 3
1.4.1. Equality of Sets 3
1.4.2. Subset 4
1.4.3. Cardinality 4
1.4.4. Empty Set 4
1.4.5. Universal Set 4
1.4.6. Power Set 4
1.4.7. Set Operations 5
1.5. Logic 8
1.5.1. Propositions 8
1.5.2. Logical Operators 8
1.5.3. Tautology 10
1.5.4. Contradiction 10
1.6. Relations 10
1.6.1. Properties of the Relation 10
1.6.2. Partial Ordering 10
1.6.3. Symmetric Relation 11
1.6.4. Equivalence Relation 11
1.7. Counting principles 11
1.7.1. Sum Rule 11
1.7.2. Product Rule 11
1.8. Alphabets 11
1.9. Strings 12
1.10. Languages 13
1.10.1. Concatenation of Languages 14
1.10.2. Languages in Set Forms 14
1.11. Kleene Closure 14
1.12. Function 15
1.12.1. On-to-function 15
1.12.2. One to One Function 16
1.12.3. One to One On-to Function 16
1.13. Three Fundamental Proof Techniques 16
1.13.1. The Principle of Mathematical Induction 16
1.13.2. The Pigeonhole Principle 17
1.13.3. The Diagonalization Principle 18
1.14. Recursive Definition 18
1.14.1. Recursive Definition of Function 20
■ **Exercise** 20

2. FINITE AUTOMATA 22–76

2.1. Introduction 22
2.2. Definition of Deterministic Finite Automata 24
2.3. Processing of Strings by DFA 24
2.4. Transition Graph 25
2.5. Generalised Transition Graphs 26
2.5.1. Definition of Generalised Transition Graph 26
2.6. Simpler Notations for DFA's 27
2.7. The Language of a DFA 28
2.8. Extending Transition Function to Strings 35
2.9. Nondeterminism 37
2.9.1. Why Nondeterminism 37
2.10. The Language of NFA 43
2.11. Transformation of NFA to DFA 43
2.12. NFA with ∈-transitions 47
2.13. Acceptance of A string by NFA with ∈-transitions 47
2.14. Conversion of NFA with ∈-transitions to NFA without ∈-transitions 48
2.15. Minimization of DFA's 51
2.16. Myhill-Nerode Theorem 55
2.17. Implementation of Myhill-Nerode theorem 56
■ Exercise 74

3. AUTOMATA WITH OUTPUT 77–92

3.1. Introduction 77
3.2. Moore Machine 77
3.2.1. Representation of Moore Machine 78
3.2.2. Processing of the String through Moore Machine 79
3.3. Mealy Machine 82
3.3.1. Representation of Mealy Machine 83
3.3.2. Processing of the String through Mealy Machine 84
3.4. Equivalence of Mealy and Moore Machine 86
3.4.1. Mealy Equivalent of given Moore Machine 86
3.4.2. Moore Equivalent of given Mealy Machine 87
■ Exercise 91

4. REGULAR EXPRESSION AND LANGUAGES 93–110

4.1. Regular Expression 93
4.1.1. The Operators of Regular Expression 93
4.1.2. Definition of Regular Expression 94
4.1.3. Building Regular Expression 94
4.1.4. Inductive Step 95
4.2. Comparative Study of Regular Expression, Regular Sets and Finite Automata 98
4.3. Construction of FA for Regular Expression 98
4.4. Construction of Regular Expression from DFA 101
4.4.1. Ardens's Theorem 102
4.4.2. Use of Arden's Theorem to find Regular Expression of a Deterministic Finite Automata 102
4.5. Algebraic Laws for Regular Expressions 104
4.5.1. Associativity and Commutativity 105
4.5.2. Identities 105
4.5.3. Distributed Law 105
4.5.4. The Idempotent Law 106
4.5.5. Laws for Closure 106
4.5.6. Identities for Regular Expression 106
■ Exercise 109

5. PROPERTIES OF REGULAR LANGUAGES 111–127

5.1. Introduction 111
5.2. Proving Languages not to be Regular 111
5.2.1. The Pumping Lemma for Regular Languages 112
5.2.2. Application of the Pumping Lemma 113
5.3. Closure Properties of Regular Languages 119
5.3.1. Reversal 121
5.3.2. Homomorphism 122
5.3.3. Inverse Homomorphism 123
5.4. Decision Properties of Regular Languages 124
5.4.1. Testing Emptiness of Regular Languages 124
5.4.2. Testing Membership in a Regular Language 125
■ Exercise 126

6. CONTEXT-FREE GRAMMARS AND LANGUAGES 128–163

6.1. Grammars 128
6.2. Context-Free Grammars 129
6.2.1. Derivations 130
6.2.2. Language of Context-Free Grammar 130
6.2.3. Sentential Forms 130
6.2.4. Bacus Naur Form (BNF) 131
6.2.5. Left most and Right most Derivations 146
6.3. Parse Tree 147
6.3.1. Definition 147
6.3.2. The Yield of Parse Tree 148
6.4. Ambiguity in Grammars and Languages 151
6.4.1. Removing Ambiguity from Grammars 155
6.4.2. Inherent Ambiguity 157
■ Exercise 162

7. SIMPLIFIED CONTEXT-FREE GRAMMAR AND IT'S NORMAL FORM 164–189

7.1. Introduction 164
7.2. Reduction of Context Free Grammar 164
7.2.1. Eliminating Use Less Symbols 165
7.2.2. Removal of Unit Production 167
7.2.3. Removal of ∈-productions and Nullable Non-terminal 170
7.3. Chomsky Normal Form 173
7.3.1. Conversion to Chomsky Normal Form 173
7.4. Greibach Normal Form 176
7.4.1. Conversion of CFG into GNF 177
7.5. Regular grammars 181
7.5.1. Equivalence of Regular Grammars and Finite Automata 182
7.5.2. Conversion of Regular Grammar into Finite Automata 183
■ Exercise 188

any day a night 2 hrs everyday turing machine.

8. PUSHDOWN AUTOMATA **190–205**

8.1. Introduction 190
8.2. Definition of Pushdown Automata 191
8.2.1. Explanation for the Moves of Pushdown Automata 192
8.3. Pushdown Automata and Context-Free Grammars 199
8.3.1. Construction of PDA Equivalent of a CFG 199
8.3.2. Construction of CFG Equivalent of a PDA 202
■ **Exercise** 204

9. PROPERTIES OF CONTEXT-FREE LANGUAGES **206–220**

9.1. Introduction 206
9.2. Periodicity Properties 209
9.3. The pumping lemma for CFL's 209
9.4. Decidiability 213
9.4.1. Emptiness 214
9.4.2. Finiteness 216
9.4.3. Membership—The CYK Algorithm 216
■ **Exercise** 219

10. TURING MACHINE **221–274**

10.1. Introduction 221
10.2. Turing Machine as Physical Computing Device 222
10.3. Formal Definition of Turing Machine 223
10.4. Instantaneous Description 224
10.5. Transition Diagram 224
10.6. Turing Thesis 227
10.7. Turing Machine for Computing Functions 227
10.8. Turing Machine as Language Acceptors 230
10.9. Combining Turing Machine 232
10.9.1. Rules for Combining Turing Machines 232
10.10. Extension of Turing Machines 245
10.10.1. Multiple Tapes Turing Machine 246
10.10.2. Two-way Infinite Tape 248
10.10.3. Multiple Heads Turing Machines 249
10.10.4. K-dimensional Turing Machine 250
10.10.5. Non-determinstic Turing Machine 251
10.10.6. Recursively Enumerable and Recursive Languages 252
10.10.7. Turing Machine as Enumerators 253
10.11. Church's Thesis 256
10.12. Universal Turing Machines 256
10.13. The Halting Problem 259
10.14. Undecidability/Decidability 259
10.14.1. Facts about Turing-Decidable and Turing Acceptable Languages 260
10.14.2. Turing Acceptable 261
10.14.3. Turing-acceptable Vs Turing-decidable 261
10.14.4. Reducibility 262
10.14.5. Properties of Recursive and Enumerable Languages 263
10.14.6. The Post Correspondence Problem 266
10.15. Pushdown Automata with Two Stacks 268
10.15.1. Equivalence of Turing Machine and Two Stack Pushdown Automata 269
10.15.2. Two Stack Push-down Automata as an Algorithm 270
■ **Exercise** 273

11. RECURS

11.1. Introduc
11.2. Partial,
11.2.1.
11.2.2.
11.2.3.
11.2.4.
11.2.5.
11.2.6.
11.2.7.
11.2.8.
11.3. Godel N
11.3.1.
■ **Exerc**

12. THE CH

12.1. Phrase-
12.2. Chomsk
12.3. Unrestr
12.4. Context
12.5. Kuroda
12.6. One-sid
12.7. Normal
12.8. Derivati
12.9. Linear B
12.10. Relation
■ **Exerc**

13. LEXICAL

13.1. Introduc
13.2. Lexical A
13.3. Ambigui
13.4. Syntax A
13.4.1.
13.4.2.
13.4.3.
13.4.4.
13.4.5.
■ **Exerc**

14. TRACTA

14.1. Introduc
14.2. Polynom
14.3. Computa
14.4. The Clas
14.4.1.
14.5. The Clas
14.5.1.
14.6. Polynom
14.7. Intractab
14.7.1.
14.7.2.

11. RECURSIVE FUNCTION THEORY 275–282

11.1. Introduction 275
11.2. Partial, Total Recursive and Primitive Recursive Functions 276
11.2.1. Initial Functions For Natural Numbers 276
11.2.2. Initial Functions for Symbols 277
11.2.3. Composition of Functions 277
11.2.4. Recursion 277
11.2.5. Primitive Recursive Function 278
11.2.6. Minimalization 280
11.2.7. Total Recursive Function 280
11.2.8. Partial Recursive Function 280
11.3. Godel Numbering 280
11.3.1. Digression: Penrose's Attack on 'Strong' AI 281
■ **Exercise** 282

12. THE CHOMSKY HIERARCHY 283–296

12.1. Phrase-Structure grammars 283
12.2. Chomsky Hierarchy 284
12.3. Unrestricted (Type-0) Grammars 286
12.4. Context Sensitive Grammar 288
12.5. Kuroda Normal Form 289
12.6. One-sided Context-Sensitive Grammars 290
12.7. Normal Form for Unrestricted Phase Structure Grammars 291
12.8. Derivation Graph 292
12.9. Linear Bounded Automata 293
12.10. Relation Between Classes of Languages 293
■ **Exercise** 294

13. LEXICAL AND SYNTAX ANALYSIS 297–309

13.1. Introduction 297
13.2. Lexical Analysis 298
13.3. Ambiguity and Formal Power Series 298
13.4. Syntax Analysis 301
13.4.1. Top-down Parsing 302
13.4.2. LL(K) Grammar 304
13.4.3. Handling LL(K) grammar with moves like $A \to \in$ 306
13.4.4. Bottom-up Parsing 307
13.4.5. LR(K) Grammar 308
■ **Exercise** 309

14. TRACTABLE AND INTRACTABLE PROBLEMS 310–320

14.1. Introduction 310
14.2. Polynomial-Time Algorithms 311
14.3. Computationally Tractable–Intractable Problems 311
14.4. The Classes P 311
14.4.1. Examples of Problems in Class P 312
14.5. The Class NP 314
14.5.1. Does P = NP? 315
14.6. Polynomial-Time Reducibility 315
14.7. Intractable Classes of Problems—NP-Hard and NP-Complete Problems 316
14.7.1. Hamiltonian Cycle Problem (HCP) 316
14.7.2. Travelling Salesman Problem (TSP) 317

14.7.3. Vertex Cover Problem 317
14.7.4. Partition Problem 317
14.7.5. Why Care About NP-Completeness 318
■ **Exercise** 318

15. INTRODUCTION TO DERIVATION LANGUAGES 321–326

15.1. Rewriting System 321
15.1.1. Composition 321
15.1.2. Juxtaposition 322
15.2. Derivation Words 322
15.3. Derivation Languages 323
15.3.1. Composition of Derivation Words 324
15.4. Canonical Derivations 326
15.5. The Context-Sensitivity of Derivation Languages 326

16. INTRODUCTION TO CELLULAR AUTOMATA 327–336

16.1. Introduction 327
16.2. Cellular Automata 328
16.3. Formal Languages and Cellular Automata 329
16.3.1. Construction of Finite Time Sets 331
16.4. Algebraic Properties of Cellular Automata 332
16.5. Invariants in Cellular Automaton Revolution 333
16.6. Computational Universality and Undecidability in Cellular Automata 334

APPENDIX : TEST YOUR SKILLS OF AUTOMATA 337–358

INDEX 359–363

Chapter 1

Introduction to Concept of Automata

Inside This Chapter

1.1. Historical Aspect of Automata
1.2. The Study of Automata Theory is Fruitful and Futuristic
1.3. Develop your Feelings with the Automata
1.4. Introduction to Sets
1.5. Logic
1.6. Relations
1.7. Counting Principles
1.8. Alphabets
1.9. Strings
1.10. Languages
1.11. Kleene Closure
1.12. Function
1.13. Three Fundamental Proof Techniques
1.14. Recursive Definition

1.1. Historical Aspect of Automata

Automata theory is the study of abstract computing devices, or "machines". Before there were computers, in 1930's, Alen turing introduced an abstract machine that has all the capabilities of today's computers, at least as far as in what they could compute. In the 1940's and 1950's, simpler kinds of machines, which we today call "finite-automata", were studied by a number of researchers. These automata, originally proposed to model brain function, turned out to be extremely useful for a variety of others purposes. Also in the late 1950's the linguist N. Chomsky began the study of formal "grammars". While not strictly machines, these grammars have close relationships to abstract automata and serve today on the basis of some important software components, including parts of compilers.

In 1969, S. Cook extended turing's study of what could and what could not be computed. Cook separated those problems that can be solved, efficiently by computer, from those problems that *m* problems can be solved, but in practice take so much time, that computers are useless for all but very small instances of the problem. The later class of problem's is called "intractable", or NP-hard. All of this theoretical development bears directly on what Computer Scientists do today.

TIPS

In computer science we find many examples of finite state system and the theory of finite state systems, and the theory of finite automata as a useful design tool for these systems. A primary example is a switching circuit. A switching circuit is composed of a finite number of gates, each of which can be in one or two conditions, usually denoted by 0 and 1.

We can assume another situation, in case of electric switch only two situation are possible that is "on" and "off". Let us say 1 for on and 0 for off. We can interchange the situation by pushing the switch, it can be seen in Fig. 1.1.

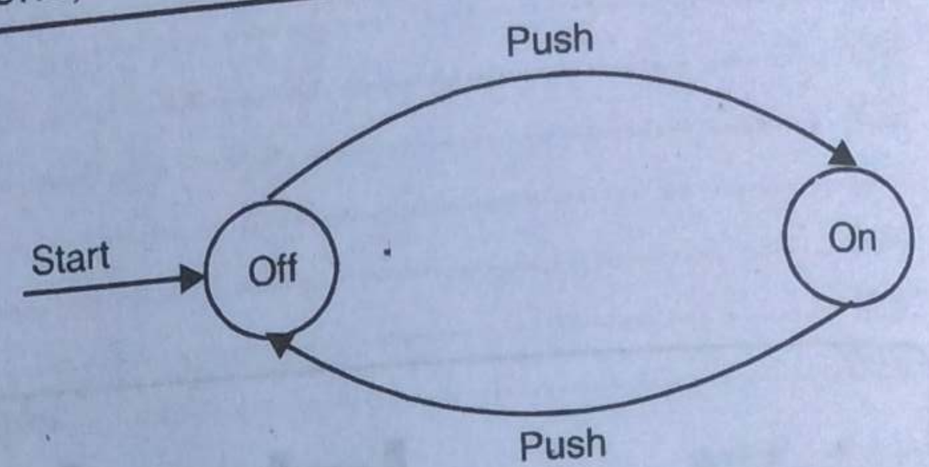


Fig. 1.1.

Automation is also tempting to view the human brain as a finite state system. The number of brain cell or neurons are limited, probably 2^{35} at most. Although there is evidence to the contrary, that the state of each neuron can be described by a small number of bits, if so then finite state theory applies to the brain also.

1.2. The Study of Automata Theory is Fruitful and Futuristic

There are several reasons why the study of automata and complexity is an important and core part of computer science. In this section we will introduce several reasons which will definitely motivate the computer science students to study automata theory. Let us list some important aspects of the automata theory.

(1) Automata theory plays an important role when we are making software for designing and checking the behaviour of digital circuit.

(2) The "Lexical analyser" of the typical compiler, that is the compiler component that breaks the input text into logical units, such as identifiers, keywords and punctuation.

(3) Software for scanning large bodies of text, such as collections of web pages, to find occurance of words, phrases or other patterns.

(4) Automata theory is key to software for verifying systems of all types that have a finite number of distinct states, such as communication protocols or protocol for secure exchange of information.

(5) Automata theory is most useful concept of software for natural language processing.

1.3. Develop Your Feelings With The Automata

Truly speaking it is very necessary to understand the subject like automata theory, that you should develop feelings about this subject.

And moreover you must try to implement it in your day-to-day life. In the whole book you will read about states so let us develop feelings for this term.

Sometime when you are sitting alone thinking about your previous life. Some events you certainly remember, which plays important role in your life. Some of them are not relevent and it is difficult to you to

remember those events. Same theory applies in the case of finite state machines, when we design a system or machine, two kinds of points come into picture, some of them certainly affect the output of the system and others never affect the output of the system. So we can skip these irrelevent points.

TIPS

Those points which affect the output of any system are called states (with the assumption that system can design infinite number of steps). Clearly number of states will be finite. The advantage of having only a finite number of states is that we can implement the system with a fixed number of resources.

1.4. Introduction to Sets

A set can be described in a number of different ways. The simplest is to list up all of its members if that is possible. For example, {1, 2, 3} is the set of three numbers 1, 2, and 3. { indicates the beginning of the set, and } its end. Every object between them separated by commas is a member of the set. Thus {{1, 2}, {{3}, 2}, 2}, {1 }} is the set of the elements {1, 2}, {{3}, 2} and {1}.

A set can also be described by listing the properties that its members must satisfy. For example, { $x \mid 1 \le x \le 2$ and x is a real number. } represents the set of real numbers between 1 and 2, and $\{x \mid x$ is the square of an integer and $x \le 100\}$ represents the set {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100}.

A third way to describe a set is to give a procedure to generate the members of the set. In this representation, first, basic elements of the set are presented. Then a method is given to generate elements of the set from known elements of the set. Thirdly a statement is given that excludes undesirable elements (which may be included in the set otherwise) from the set. for example, the set of natural numbers N can be defined recursively as the set that satisfies the following (1), (2), and (3) :

(1) $0 \in N$.

(2) For any number x if $x \in N$, then $x + 1 \in N$.

(3) Nothing is in N unless it is obtained from (1) and (2).

Following this definition, the set of natural numbers N can be obtained as follows :

First by (1), 0 is put into N.

Then by (2), since 0 is in N, $0 + 1$ (=1) is in N.

Then by (2) again, $1 + 1$ (= 2) is in N.

Proceeding in this manner all the natural numbers are put into N.

Note that if we don't have (3), 0.5, 1.5, 2.5, ... can be included in N, which is not what we want as the set of natural numbers.

1.4.1. Equality of Sets

Two sets are equal if and only if they have the same elements. More formally, for any sets A and B, $A = B$ if and only if $\forall x\ [x \in A \leftrightarrow x \in B]$.

Thus for example, {1, 2, 3} = {3, 2, 1}, that is the order of elements does not matter, and {1, 2, 3} = {3, 2, 1, 1}, that is duplications do not make any difference for sets.

1.4.2. Subset

A set A is a subset of a set B if and only if everything in A is also in B. More formally, for any sets A and B, A is a subset of B, and denoted by $A \subseteq B$, if and only if $\forall x[x \in A \rightarrow x \in B]$.

If $A \subseteq B$, and $A \neq B$, then A is said to be a proper subset of B and it is denoted by $A \subset B$.

For example, $\{1, 2\} \subseteq \{3, 2, 1\}$

Also $\{1, 2\} \subset \{3, 2, 1\}$.

1.4.3. Cardinality

If a set S has n distinct elements for some natural number n, n is the cardinality (size) of S and S is a finite set. The cardinality of S is denoted by $|S|$.

For example, the cardinality of the set $\{3, 1, 2\}$ is 3.

1.4.4. Empty Set

A set which has no elements is called an empty set. More formally, an empty set, denoted by $\emptyset$, is a set that satisfies the following : $\forall x\ x \notin \emptyset$, where $\notin$ means "is not in" or "is not a member of".

Note that $\emptyset$ and $\{\emptyset\}$ are different sets. $\{\emptyset\}$ has one element namely $\emptyset$ in it. So $\{\emptyset\}$ is not empty. But $\emptyset$ has nothing in it.

1.4.5. Universal Set

A set which has all the elements in the universe of discourse is called a universal set. More formally, a universal set, denoted by U, is a set that satisfies the following : $\forall x\ x \in U$.

Three subset relationships involving empty set and universal set are listed below as theorems without proof. Note that the set A in the next four theorems are arbitrary. So A can be an empty set or universal set.

Theorem 1 : For an arbitrary set A $\quad A \subseteq U$.

Theorem 2 : For an arbitrary set A $\quad \emptyset \subseteq A$.

Theorem 3 : For an arbitrary set A $\quad A \subseteq A$.

1.4.6. Power Set

The set of all subsets of a set A is called the power set of A and denoted by 2^A or $\rho(A)$.

For example, for $A = \{1, 2\}$, $\rho(A) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$.

For $B = \{\{1, 2\}, \{\{1\}, 2\}, \emptyset\}$, $\rho(B) = \{\emptyset, \{\{1, 2\}\}, \{\{\{1\}, 2\}\}, \{\emptyset\}, \{\{1, 2\}, \{\{1\}, 2\}\}, \{\{1, 2\}, \emptyset\}, \{\{\{1\}, 2\}, \emptyset\}, \{\{1, 2\}, \{\{1\}, 2\}, \emptyset\}\}$.

Also, $\rho(\emptyset) = \{\emptyset\}$ and $\rho(\{\emptyset\}) = \{\emptyset, \{\emptyset\}\}$.

Theorem 4 : For an arbitrary set A, the number of subsets of A is $2^{|A|}$.

Example 1.1. List the members of the following sets :

(a) $\{ x \mid x$ is a real number such that $x^2 = 4\}$

(b) $\{x \mid x$ is an integer such that $x^2 = 2\}$.

Solution. (a) $\{-2, 2\}$, (b) $\emptyset$

Example 1.2. Determine whether each of the following pairs of sets is equal.

(a) $\{1, 2, 1, 3, 1, 2\}, \{2, 3, 1\}$ (b) $\{\{1\}\}, \{1, \{1\}\}$ (c) $\emptyset, \{\emptyset\}$

Solution. (a) Yes (b) No (c) No

Example 1.3. For each of the following sets, determine whether 1 is an element of that set.

(a) $\{x \in R \mid x$ is an integer greater than 1$\}$

(b) $\{x \in R \mid x$ is the square of an integer$\}$

(c) $\{1, 2, \{1\}\}$

(d) $\{\{1\}, \{\{1\}\}\}$

(e) $\{\{1, 2\}, \{1, \{1\}\}\}$

(f) $\{\{\{1\}\}\}$.

Solution. (a) No (b) Yes (c) Yes (d) No (e) No (f) No

Example 1.4. Suppose that A, B and C are sets such that $A \subset B$ and $B \subseteq C$. Show that $A \subseteq C$.

Solution. Let x be an arbitrary element of A. We are going to show that $x \in C$. Since $A \emptyset B$, this implies that $x \in B$. Since $B \subseteq C$, we see that $x \in C$. Since $x \in A$ implies that $x \in C$, it follows that $A \subseteq C$.

Example 1.5. Find the power set of each of the following sets :

(a) $\{a\}$ (b) $\{a, \{a\}\}$ (c) $\{\emptyset, \{\emptyset\}\}$

Solution.

(a) $\{\emptyset, \{a\}\}$ (b) $\{\emptyset, \{a\}, \{\{a\}\}, \{a, \{a\}\}\}$

(c) $\{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\}$

1.4.7. Set Operations

Sets can be combined in a number of different ways to produce another set. Here some basic operations are introduced and their properties are discussed.

(i) Union of sets

(ii) Intersection of sets

(iii) Difference of sets

(iv) Complement of set

(v) Ordered pair, ordered n-tuple

(vi) Equality of ordered n-tuples

(vii) Cartesian product of sets

1.4.7.1. Union of Sets

The union of sets A and B, denoted by $A \cup B$, is the set defined as

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

For example if $A = \{1, 2, 3\}$ and $B = \{4, 5\}$, then $A \cup B = \{1, 2, 3, 4, 5\}$.
For example if $A = \{1, 2, 3\}$ and $B = \{1, 2, 4, 5\}$, then $A \cup B = \{1, 2, 3, 4, 5\}$.
Note that elements are not repeated in a set.

1.4.7.2. Intersection of Sets

The intersection of sets A and B, denoted by $A \cap B$, is the set defined as

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

For example if $A = \{1, 2, 3\}$ and $B = \{1, 2, 4, 5\}$, then $A \cap B = \{1, 2\}$.
For example if $A = \{1, 2, 3\}$ and $B = \{4, 5\}$, then $A \cap B = \varnothing$.

1.4.7.3. Difference of Sets

The difference of sets A and B, denoted by $A - B$, is the set defined as

$$A - B = \{x \mid x \in A \wedge x \notin B\}$$

For example if $A = \{1, 2, 3\}$ and $B = \{1, 2, 4, 5\}$, then $A - B = \{3\}$ and if $A = \{1, 2, 3\}$ and $B = \{4, 5\}$, then $A - B = \{1, 2, 3\}$.
Note that in general $A - B \neq B - A$.

1.4.7.4. Complement of Sets

For a set A, the difference $U - A$, where U is the universe, is called the complement of A and it is denoted by $\overline{A}$. Thus $\overline{A}$ is the set of everything that is not in A.

The fourth set operation is the Cartesian produce. We first define an ordered pair and Cartesian produce of two sets using it. Then the Cartesian produce of multiple sets is defined using the concept of n-tuple.

1.4.7.5. Ordered Pair

An ordered pair is a pair of objects with an order associated with them. If objects are represented by x and y, then we write the ordered pair as $<x, y>$.

Two ordered pairs $<a, b>$ and $<c, d>$ are equal if and only if $a = c$ and $b = d$. For example, the ordered pair $<1, 2>$ is not equal to the ordered pair $<2, 1>$.

1.4.7.6. Cartesian Product

The set of all ordered pairs $<a, b>$, where a is an element of A and b is an element of B, is called the Cartesian produce of A and B and is denoted by $A \times B$.

Example 1.6. Let $A = \{1, 2, 3\}$ and $B = \{a, b\}$. Then

$$A \times B = \{<1, a>, <1, b>, <2, a>, <2, b>, <3, a>, <3, b>\}.$$

Example 1.7. For the same A and B as in Example 1.6,

$$B \times A = \{<a, 1>, <a, 2>, <a, 3>, <b, 1>, <b, 2>, <b, 3>\}.$$

As you can see in these examples, in general, $A \times B \neq B \times A$ unless $A = \varnothing$, $B = \varnothing$ or $A = B$. Note that $A \times \varnothing = \varnothing \times A = \varnothing$ because there is no element in $\varnothing$ to form ordered pairs with elements of A.

The concept of Cartesian produce can be extended to that of more than two sets. First we are going to define the concept of *ordered n-tuple*.

1.4.7.7. Ordered n-tuple

An ordered n-tuple is a set of n objects with an order associated with them (rigorous definition to be filled in). If n objects are represented by $x_1, x_2, \ldots, x_n$, then we write the ordered n-tuple as $<x_1, x_2, \ldots, x_n>$.

1.4.7.8. Cartesian Product

Let $A_1, \ldots, A_n$ be n sets. Then the set of all ordered n-tuples $<x_1, \ldots, x_n>$, where $x_i \in a_i$ for all i, $1 \leq i \leq n$, is called the Cartesian produce of $A_1, \ldots, A_n$, and is denoted by $A_1 \times \ldots \times A_n$.

Example 1.8. Let $A = \{1, 2\}$, $B = \{a, b\}$ and $C = \{5, 6\}$. Then

$$A \times B \times C = \{<1, a, 5>, <1, a, 6>, <1, b, 5>, <1, b, 6>, <2, a, 5>, <2, a, 6>, <2, b, 5>, <2, b, 6>\}.$$

Equality of n-tuples : Two ordered n-tuples $<x_1, \ldots, x_n>$ and $<y_1, \ldots, y_n>$ are equal if and only if $x_i = y_i$ for all i, $1 \leq i \leq n$.

For example the ordered 3-tuple $<1, 2, 3>$ is not equal to the ordered n-tuple $<2, 3, 1>$.

Example 1.9. Let A be a set. Show that $\varnothing \times A = A \times \varnothing = \varnothing$.

Solution. $\varnothing \times A = \{(x, y) \mid x \in \varnothing \text{ and } y \in A\} = \varnothing = \{(x, y) \mid x \in A \text{ and } y \in \varnothing\} = A \times \varnothing$

Example 1.10. How many different elements does $A \times B$ have if A has m elements and B has n elements ?

Solution. mn.

Example 1.11. Let $A = \{1, 2, 3, 4\}$ and $b = \{0, 3, 5\}$. Find

(a) $A \cup B$ **(b) $A \cap B$**
(c) $A - B$ **(d) $B - A$**

Solution.

(a) $\{0, 1, 2, 3, 4, 5\}$ (b) $\{3\}$
(c) $\{1, 2, 4\}$ (d) $\{0, 5\}$

Example 1.12. What can you say about the sets A and B if the following are true ?

(a) $A \cup B = A$ **(b) $A - B = A$**

Solution. (a) $B \subseteq A$. For since $A \cup B = A$ every element of B is also in A. But A may have more elements than B.

(b) $A \cap B = \varnothing$. For $A - B = A$ means that subtracting elements of B does not change A. Hence no elements of B are in A. Hence A and B have no elements in common.

Example 1.13. Let A and B be subsets of a universal set U. Show that $A \subseteq B$ if and only if $\overline{B} \subseteq \overline{A}$.

Solution.

$$A \subseteq B \Leftrightarrow \forall x\,(x \in A \rightarrow x \in B) \Leftrightarrow \forall x(x \notin B \rightarrow x \notin A) \Leftrightarrow \forall x(x \in \overline{B} \rightarrow x \in \overline{A}) \Leftrightarrow \overline{B} \subseteq \overline{A}$$

1.4.7.9. Properties of Set Operation

1. $A \cup \emptyset = A$ ...Identity Laws
   $A \cap U = A$
2. $A \cup U = U$ ...Domination Laws
   $A \cap \emptyset = \emptyset$
3. $A \cup A = A$ ...Idempotent Laws
   $A \cap A = A$
4. $A \cup B = B \cup A$ ...Commutative Laws
   $A \cap B = B \cap A$
5. $(A \cup B) \cup C = A \cup (B \cup C)$ ...Associative Laws
   $(A \cap B) \cap C = A \cap (B \cap C)$
6. $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ ...Distributive Laws
   $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
7. If $A \subseteq B$ and $C \subseteq D$, then $A \cup C \subseteq B \cup D$, and $A \cap C \subseteq B \cap D$.
8. If $A \in B$, then $A \cup B = B$ and $A \cap B = A$.
9. $A \cup (B - A) = A \cup B$.
10. $A \cap (B - A) = \emptyset$.
11. $A - (B \cup C) = (A - B) \cap (A - C)$ (cf. $\overline{A \cup B} = \overline{A} \cap \overline{B}$)
    $A - (B \cap C) = (A - B) \cup (A - C)$ (cf. $\overline{A \cap B} = \overline{A} \cup \overline{B}$) ...DeMorgan's Laws
12. $B = A$ if and only if $A \cup B = U$ and $A \cap B = \emptyset$
13. $\overline{A} = A$.

Additional Properties :

14. $A \subseteq A \cup B$
15. $A \cap B \subseteq A$

1.5. Logic

In this section, we will introduce some of the basic concepts of mathematical logic. These logical arguments will usually be applied to mathematics, they employ the same technique that are used by a doctor in clinic or a by lawyer in a courtroom. Another reason for the importance of this section is that the circuits that make up digital computers are designed using the same algebra of propositions that we will be discussing.

1.5.1. Propositions

A proposition is a sentence to which one and only one of the terms true or false can be meaningfully applied.

For example :

"Five is odd" → True

"$5 \in \{1, 3, 2\}$ → False

and "$45 \geq 22$" → True

are propositions.

1.5.2. Logical Operators

There are several ways in which we commonly combine simple statements into compound ones. To avoid any confusion, we will precisely define each

one's meaning and introduce its standard symbol. Let us discuss these logical operators as follows :

1. If p and q are propostions, **their conjunction (AND)**, p and q (denoted $p \wedge q$) is defined in following truth table :

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

2. If p and q are propositions, their **disjunction (OR)** is p or q, denoted $p \vee q$, and is defined in following truth table.

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

3. If p is a proposition, its **negation (NOT)**, not p, is denoted $\sim p$ is defined in following truth table.

p	$\sim p$
0	1
1	0

4. The **conditional operator if p then q**, denoted $p \rightarrow q$, is defined by following truth table :

p	q	$p \rightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

The only case in which a conditional proposition is false is when the condition is true and the conclusion is false.

5. If p and q are propositions, the **Biconditional operator (If and only if) p if and only if q**, denoted $p \leftrightarrow q$, is defined by following :

p	q	$p \leftrightarrow q$
0	0	1
0	1	0
1	0	0
1	1	1

$p \leftrightarrow q$ is true when p and q have the same truth values.

1.5.3. Tautology

An expression involving logical variables that is true in all cases is called a tautology. For example following are tautologies.

(*i*) $\sim(p \wedge q) \leftrightarrow (\sim p) \vee (\sim q)$

(*ii*) $p \vee \sim p$

(*iii*) $(p \wedge q) \rightarrow p$

(*iv*) $q \rightarrow (p \vee q)$

(*v*) $(p \vee q) \leftrightarrow (q \vee p)$

1.5.4. Contradiction

An expression involving logical variables that is false for all cases of its truth table is called a contradiction. For example $p \wedge \sim p$ and $(p \vee q) \wedge (\sim p) \wedge (\sim q)$ are contradictions.

1.6. Relations

Complete understanding of a set is possible only if the structure of the set is made clear by the inter-relationships between its elements. For example, the individuals in crowd can be compared by height, by nationality, or through several other criteria. **In mathematics, such comparisons are called relations.**

"Let A and B be set. A relation from A into B is any subset of $A \times B$"

For example, let $A = \{1, 2, 3\}$ and $B = \{4, 5\}$ then

$\{(1, 4), (2, 4), (3, 5)\}$ is a relation from A into B.

Let $A = \{2, 3, 5, 6\}$ and define a relation R from A into A by $(x, y) \in R$ if and only if x divides evenly into y. So $R = \{(2, 2), (3, 3), (5, 5), (6, 6), (2, 6), (3, 6)\}$.

1.6.1. Properties of the Relation

There are following three basic properties of Relation :

(*a*) Reflexive (*b*) Antisymmetric (*c*) Transitive.

Let A be a set and R is a relation on A then

(*i*) R is called **reflexive** if and only if ${}_aR_a$ for all $a \in A$.

(*ii*) R is called **antisymmetric** if and only if whenever ${}_aR_b$ and $a \neq b$ then ${}_bR_a$ is false; or equivalently whenever ${}_aR_b$ and ${}_bR_a$ then $a = b$.

(*iii*) R is called **transitive** if and only if whenever ${}_aR_b$ and ${}_bR_c$ then ${}_aR_c$.

1.6.2. Partial Ordering

A relation on a set A **which is reflexive, antisymmetric and transitive** is called a partial ordering on A. A set on which there is a partial ordering defined is called a **Partial order set** or **poset.**

For example, for a set A, $\rho(A)$ together with the relation $\subseteq$ is a poset, can be prove by showing following three properties :

(*i*) Let $B \in \rho(A)$. We must show that $\rho \subseteq B$. This is true by the definition of subset. hence the relation is reflexive.

(*ii*) Let $B_1, B_2 \in \rho(A)$ and assume that $B_1 \subseteq B_2$ and $B_1 \neq B_2$ clearly $B_2 \nsubseteq B_1$ so relation is antisymmetric.

(*iii*) Let $B_1, B_2, B_3 \in \rho(A)$ and assume that $B_1 \subseteq B_2$ and $B_2 \subseteq B_3$. So by the defition $B_1 \subseteq B_3$ so relation is transitive.

Above three properties are enough to show that for a set A, power set of A ($\rho(A)$) with relation $\subseteq$ is a poset.

1.6.3. Symmetric Relation

Let R be a relation on a set A. R is called symmetric if and only if whenever ${}_aR_b$, it follows ${}_bR_a$ for $a, b \in A$.

For example consider the relation = defined on R. Certainly $a = b \Rightarrow b = a$, so = is a symmetric relation on R.

1.6.4. Equivalence Relation

A relation R on a set A is called an equivalence relation if and only if it is reflexive, symmetric and transitive.

The classic example of an equivalence relation is the relation = on R.

Let Z^* be the set of non-zero integers. One of the most basic equivalence relation in mathematics is the relation q on $Z \times Z^*$ defined by $(a, b)q\ (c, d)$ if and only if $ad = bc$.

1.7. Counting Principles

It is frequently necessary to count how many ways certain choices can be made. In computer science, for instance, we may want to estimate the number of times a certain procedure is called during the execution of a program, in order to assess the program's efficiency, or in theory of games, one might want to access the probability of a certain outcome given the current position, and this usually involves counting the number of ways in which certain combinations can arise.

In this section we collect some of the basic methods for counting the number of ways in which events can happen. We begin with rules of sum and product.

1.7.1. Sum Rule

We say **events are mutually exclusive or incompatible** if they cannot occur simultaneously. the rule of sum then says that the number of ways in which either of two mutually exclusive events can occur is equal to the sum of the number of ways in which each can occur separately. In set theoretical terms this says that if A and B are disjoint sets then $|A \cup B| = |A| + |B|$.

1.7.2. Product Rule

The rule of product says that number of ways in which two independent events E_1 and E_2 can occur is the product of the number of ways in which E_1 and E_2 can occur separately.

1.8. Alphabets

Everything in mathematics is based on symbols. This is also true for automata theory of computation. The symbols are generally letters and digits.

Definition "Alphabets are defined as a finite set of symbols"

An example of an alphabet is a set of decimal numbers.

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Here Σ is representation for the alphabet.

Another example is a binary alphabet that is :

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

Similarly $\Sigma = \{A, B, \ldots Z\}$ is a alphabet.

$\Sigma = \{a, b, \ldots z\}$ is also a alphabet}.

1.9. STRINGS

Definition "A string (or sometimes word) is a finite sequence of symbols selected from some alphabet". For example, if Σ $\{a, b\}$ is an alphabet then *abab* is string over alphabet Σ. A string is generally denoted by w.

The **"length"** of the string is denoted by $| w |$ and it is, the number of positions for the symbol in the string. For instance string

$$w = 01101 \text{ has length}$$

$$| w | = 5.$$

TIPS

It is common to say that the length of the string is "the number of symbols" in the string; this statement is generally accepted but not strictly correct. Thus, there are only two symbols, 0 and 1, in the string 01101, but there are five positions for symbols, and its length is 5.

The **"empty"** string is the string with zero occurance of symbols. This string is represented by $\in$, or e or λ, or γ **but we will follow $\in$ in this particular book.**

The set of strings, including the empty, over an alphabet Σ is denoted by Σ^*, that is suppose $\Sigma = \{0, 1\}$ then Σ^* is a set containing empty and all combinations of 0's and 1's. Thus

$$\Sigma^* = \{\in, 0, 1, 01, 10, 000, 010, 0000, 1111000 \ldots\}$$

Similarly $\{0\}^* = \{\in, 0, 00, 000 \ldots\}$,

$$\{1\}^* = \{\in, 1, 11, 111 \ldots\}$$

Note that $\in$ is in Σ^*, regardless of what alphabet Σ is.

That is $\in$ is only string whose length is 0.

if $\Sigma = \{0, 1\}$, then $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$,

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

and so on. Note that there is slight confusion between Σ and Σ^1. The formal is an alphabet; its members 0 and 1 are symbols. The latter is a set of strings; its members are the strings 0 and 1, each of which is of length 1. We shall not try to use seprate notations for the two sets, relying on context to make it clear whether {0, 1} or similar sets are alphabets or set of strings.

Sometimes, we wish to exclude the empty string from the set of strings. The set of non-empty strings from alphabet Σ is denoted by Σ^+. Thus, two appropriate equivalence are :

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$$

$$\Sigma^* = \Sigma^+ \cup \{\in\}.$$

Concatenation of Strings. Let w_1 and w_2 be two strings. Then $w_1\ w_2$ denotes the concatenation of w_1 and w_2, that is, the string formed by making a copy of w_1 and followed it by a copy of w_2.

if $w_1 = abc$

$$w_2 = xyz$$

then $w_1\ w_2 = abc\ xyz$

In the **set form string** w_2 is defined as :

$$w_2 = \begin{cases} \in & \text{for } i = 0 \\ w^{i-1}\ w & \text{for } i \geq 1 \end{cases}$$

that is if $w = aba$

then $w^3 = (aba)^3$

$$w^3 = aba\ aba\ aba = (aba)^2\ aba.$$

Reverse of the String. Reverse of the string can be achieved by flipping over last symbols :

if $w = abc$

then $w^R = cba$, where w^R is reverse of w, $| w | = | w^R |$

1.10. LANGUAGES

Definition "A set of strings all of which are chosen from some Σ^*, where Σ is a particular alphabet, is called a language". If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is said to be language over alphabet Σ.

Notice that a language over Σ need not include strings with all the symbols of Σ, so once we have established that L is a language over Σ, we also know it is a language over any alphabet which is a superset of Σ.

An example is english, where the collection of legal english words is a set of strings over the alphabet that consists of all the letters. Another example is C or any other programming language, where the legal programs are a subset of the possible strings that can be formed from the alphabet of the language. This alphabet is a subset of the ASCII characters. The exact alphabet may differ slightly among different programming languages, but generally includes the upper-and lower-case letters; the digits, punctuation, and mathematical symbols.

However, there may be some other languages that appear when we study automata. Some examples are as follows :

(*a*) The language of all strings consisting of n 0's followed by n 1's, for some $n \geq 0$: $\{\in, 01, 0011, 000111, \ldots\}$

(*b*) The set of strings over 0's and 1's with equal number of each :
{∈, 01, 10, 0011, 0101, 1001 ...}.

(*c*) The set of binary numbers whose value is a prime :
{10, 11, 101, 111, 1011, ...}

(*d*) Σ^* is a language for any alphabet Σ.

(*e*) ϕ, the empty language, is a language over any alphabet.

(*f*) {∈}, the language consisting of only the empty string, is also a language over any alphabet. Notice that $\phi \neq \{\in\}$; the former has no strings and latter has one string.

The only important constraint on what can be a language is that all alphabet are finite. Thus languages, although they can have an infinite number of strings, are restricted to consist of strings drawn from one fixed, finite alphabet.

1.10.1. Concatenation of Languages

Concatenation of languages can be defined as :

if L_1 and L_2 are two languages, their concatenation

$L = L_1 . L_2$ where $L = \{w/w = xy, x \in L_1, y \in L_2\}$.

That is, the string in L_1 L_2 are formed by choosing a string in L_1 and followed by a string in L_2, in all possible combinations.

1.10.2. Languages in Set Forms

It is very common to describe a language using a "set-form" :

{*w*/some logical view about *w*}.

This expression is read "the set of words *w* such that (whatever is said about *w* to the right of the vertical bar)." For example :

(*a*) {*w*/*w* consists an equal number of 0's and 1's}

(*b*) $L = \{w/w$ is a binary integer that is prime$\}$

It is common to replace *w* by some expression with parameters and describe the strings in the language by stating conditions on the parameters. Here are some examples :

$\{a^n b^n / n \geq 1\}$. Read "the set of a to the *n*, *b* to the *n* such that *n* is greater than or equal to 1," this language consists of the strings {ab, aabb, aaabbb, ...}. Notice that as with alphabet, we can raise a single symbol to a power *n* in order to represent *n* copies of that symbol.

1.11. Kleene Closure

Given an alphabet, we wish to define a language in which any string of letters form Σ is a word, even the null string. This language we shall call the closure of the alphabet. It is denoted by writing a star, after the name of the alphabet as a superscript :

$$\Sigma^*$$

This notation is sometimes known as the "kleene star" after the name of logician who was one of the founders of this subject.

For example if $\Sigma = \{a\}$

then $\Sigma^* = \{\in, a, aa, aaa, aaaa, ...\}$

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

then $\Sigma^* = \{\in, 0, 1, 00, 01, 10, 11, 000, 001 ...\}$

Definition "If *S* is a set of words then by S^* we mean the set of all finite strings formed by concatenating words from *S*, where any word may be used as often we like, and where the null string is also included".

We can think of the kleene star as an operation that makes an infinite language of strings of letters out of an alphabet. When we say "infinite language", we mean infinitely many words, each of which of finite length.

1.12. Function

A function is an association of each object of one kind with a unique object other kinds.

"A function from a set S_1 to a set S_2 is a binary relation R on S_1 and S_2 with the following property : for each $x \in S_1$ there is **exactly one** ordered pair in *R* with first component of S_2.

Let us see a example :

Let S_1 be set of cities and S_2 is the set of states and we are defining two relations as follows :

$R_1 = \{(x, y) : x \in S_1, y \in S_2$, and *x* is a city in state *y*$\}$,

$R_2 = \{(x, y) : x \in S_2, y \in S_1$, and *y* is a city in state *x*$\}$.

Then relation R_1 is a function, since each city is in one and only one state, but R_2 is not a function, since some states have more than one city.

Generally function is represented as

$$f : S_1 \rightarrow S_2$$

that is it indicate *f* is a function from S_1 to S_2. We call S_1 the domain and S_2 as range. If $x \in S_1$. We write $f(x)$ for the element *y* of S_2 such that $(x, y) \in f$; since *f* is function, there is exactly one $y \in S_2$ with this property so $f(x)$ denotes a unique object. The object $f(x)$ is called the image of *x* under *f*.

Now let us discuss various types of functions :

1.12.1. On-to-function

A function from set S_1 to S_2 is said to be an on to function if every element of S_2 is the image of one or more element of S_1.

Let $S_1 = \{a, b, c, d, e\}$

$S_2 = \{\alpha, \beta, \gamma\}$

Here *f* is defined as follows :

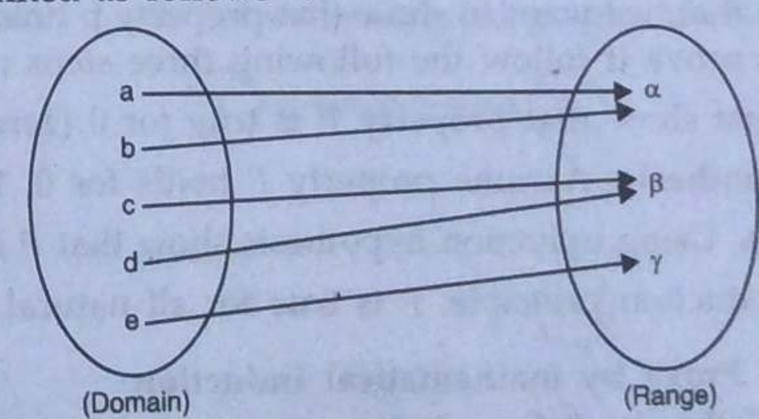


Fig. 1.2.

then *f* is onto function.

1.12.2. One to One Function

A function from S_1 to S_2 is said to be a one to one function if no two elements of S_1 have the same image in S_2.

Let $S_1 = \{a, b, c, d, e\}$

and $S_2 = \{\alpha, \beta, \gamma, \delta, \psi\}$

and function f is defined as follows :

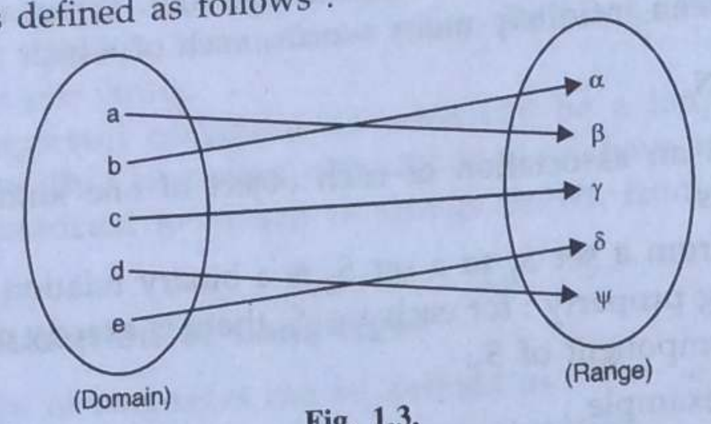


Fig. 1.3.

Here f is said to be one to one function.

1.12.3. One to One On-to Function

A function from set S_1 to set S_2 is said to be one to one onto function if it is both onto and one to one functions.

In the literature an on-to function is also called a surjection. A one to one function is also called an injection. In the similar fashion A one to one onto function is also called bijection.

1.13. Three Fundamental Proof Techniques

There are three basic fundamental technique to proof. These three fundamental principles, are important to proof any problem.

1.13.1. The Principle of Mathematical Induction

The principle of mathematical induction states that any set of natural numbers containing zero, and with the property that it contains $n + 1$ whenever if contains all numbers up to end including n, must in fact be the set of all natural numbers.

We can use this induction method as follows :

Let us assume that, we want to show that property p holds for all natural numbers. Then to prove it follow the following three steps :

Basic Step. First show that property P is true for 0 (zero).

Induction Hypothesis. Assume property P holds for 0, 1, 2, ... n.

Induction Step. Using induction hypothesis show that P is true for $n + 1$.

Then by the induction principle, P is true for all natural numbers.

Example 1.14. Prove by mathematical induction $n^4 - 4n^2$ is divisible by 3 for $n \geq 0$.

Solution. Basic Step. For $n = 0$

$$n^4 - 4n^2 = 0 \text{ which is divisible by 3.}$$

Induction hypothesis. Let $n^4 - 4n^2$ is divisible by 3

Induction Step. $(n + 1)^4 - 4(n + 1)^2$

$$= ((n + 1)^2)^2 - 4(n + 1)^2$$
$$= (n^2 + 2n + 1)^2 - (2n + 2)^2$$
$$= (n^2 + 2n + 1 + 2n + 2)\ (n^2 + 2n + 1 - 2n - 2)$$
$$= (n^2 + 4n + 3)\ (n^2 - 1)$$
$$= n^4 + 4n^3 + 3n^2 - 3 - 4n - n^2$$
$$= (n^4 - 4n^2) + 6n^2 - 3 + 4(n^3 - x)$$

Now we have to prove that $(n^3 - n)$ is divisible by 3, Let us again use Induction :

Basic Step. For $n = 0$

$n^3 - n = 0$ which is divisible by 3.

Induction hypothesis. Let $n^3 - n$ is divisible by 3.

Induction Step. $(n + 1)^3 - (n + 1)$

$$= (n^3 + 3n^2 + 3n + 1) - (n + 1)$$
$$= (n^3 + 3n^2 + 3n + 1 - n - 1$$
$$= n^3 + 3n^2 + 2n$$
$$= (n^3 - n) + 3n^2 + 3n \text{ which is divisible by 3.}$$

So we can say that $(n^4 - 4n^2) + 6n^2 - 3 + 4\ (n^3 - n)$ is also divisible by 3. Hence proved.

1.13.2. The Pigeonhole Principle

If S_1 and S_2 are non-empty finite sets and $| S_1 | > | S_2 |$, then there is no one-to-one function from S_1 to S_2 (that is there is no bijection).

In other words, if we attempt to pair off the element of S_1 (the "pigeons") with element of S_2 (the "pigeonholes"), sooner or later we will have to put more than one pigeon in a pigeonhole.

For example

if $S_1 = \{a_1, a_2, a_3\}$ then $| S_1 | = 3$

if $S_2 = \{(a_1, a_2), (a_2, a_3)\}$ then $| S_2 | = 2$

This would only holds for an undirected graph that is no cycles.

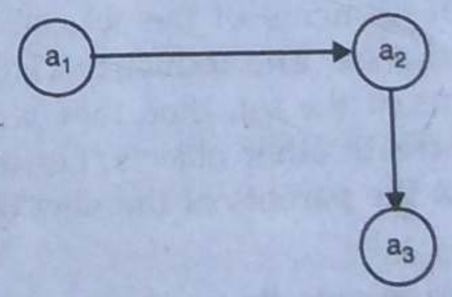


Fig. 1.4.

1.13.3. The Diagonalization Principle

Consider set S_i with members $k \in N$ as indicated (➤) below :

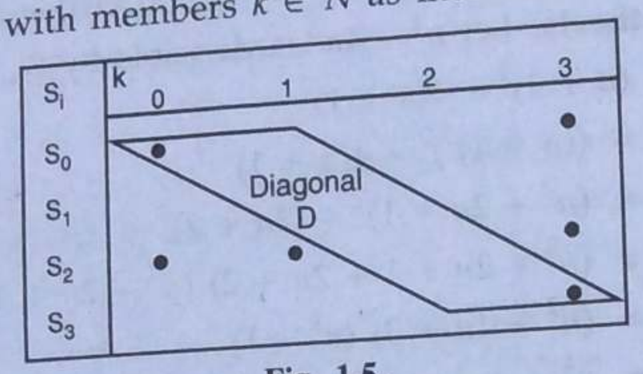


Fig. 1.5.

Diagonal set $D = \{0, 3, ...\}$ and

$\overline{D}$ (Complement of diagonal) $= \{1, 2, ...\}$

Now we can define the diagonal principle as follows :

" Let R be a binary relation on a set S, and let D, the diagonal set of R, be $\{x : x \in S \text{ and } (x, x) \notin R\}$. For each $x \in S$; let $R_x = \{b : b \in S \text{ and } (x, y) \in R\}$. Then D is distinct from each R_x.

1.14. Recursive Definition

Sets which have too many elements to list them up, and for which there are no convenient or obvious predicates to specify their elements can often be defined using a recursive definition (also called inductive definition). It essentially gives a procedure to generate the members of the set one by one starting with some subset of its elements. In this type of definition, first a collection of elements to be included initially in the set is specified. These elements can be viewed as the seeds of the set being defined. Next, the rules to be used to generate elements of the set from elements already known to be in the set (initially the seeds) are given. These rules provide a method to construct the set element by element starting with the seeds. These rules can also be used to test elements for the membership in the set.

A recursive definition of a set always consists of three distinct clauses :

1. The basis clause (or simply basis) of the definition establishes that certain objects are in the set. This part of the definition specifies the "seeds" of the set from which the elements of the set are generated using the methods given in the inductive clause. The set of elements specified here is called basis of the set being defined.
2. The inductive clause (or simply induction) of the definition establishes the ways in which elements of the set can be combined to produce new elements of the set. The inductive clause always asserts that if objects are elements of the set, then they can be combined in certain specified ways to create other objects. Let us call the objects used to create a new object the parents of the new object, and the new object is their child .
3. The extremal clause asserts that unless an object can be shown to be a member of the set by applying the basis and inductive clauses a finite number of times, the object is not a member of the set.

Let us see some of the examples of Recursive Definition of Set

Example 1.15. Definition of the Set of Natural Numbers N.

Solution. The set N is the set that satisfies the following three clauses:

Basis Clause. $0 \in N$.

Inductive Clause. For any element x in N, $x + 1$ is in N.

Extremal Clause. Nothing is in N unless it is obtained from the Basis and Inductive Clauses.

The basis for this set N is $\{0\}$. The $x + 1$ in the Inductive Clause is the parent of x, and x is the child of $x + 1$.

Following this definition, the set of natural numbers N can be obtained as follows :

First by the Basis Clause, 0 is put into N.

Then by the Inductive Clause, since 0 is in N, $0 + 1$ (= 1) is in N. 0 is the parent of 1, and 1 is the child of 0.

Then by the Inductive Clause again, $1 + 1$ (= 2) is in N. 1 is the parent of 2, and 2 is the child of 1.

Proceeding in this manner all the "natural numbers" are put into N.

Note that if we don't have the Extremal Clause, 0.5, 1.5, 2.5, ... can be included in N, which is not what we want as the set of natural numbers.

Example 1.16. Definition of the Set of Nonnegative Even Numbers *NE*.

Solution. The set NE is the set that satisfies the following three clauses:

Basis Clause. $0 \in NE$

Inductive Clause. For any element x in NE, $x + 2$ is in NE.

Extremal Clause. Nothing is in NE unless it is obtained from the Basis and Inductive Clauses.

Example 1.17. Definition of the Set of Even Integers *EI*.

Solution. The set EI is the set that satisfies the following three clauses :

Basis Clause. $0 \in EI$

Inductive Clause. For any element x in EI, $x + 2$, and $x - 2$ are in EI.

Extremal Clause. Nothing is in EI unless it is obtained from the Basis and Inductive Clauses.

Example 1.18. Definition of the Set of Strings *S* over the alphabet {*a*, *b*} excepting empty string. This is the set of strings consisting of *a*'s and *b*'s such as *abbab*, *bbabaa*, etc.

Solution. The set S is the set that satisfies the following three clauses :

Basis Clause. $a \in S$, and $b \in S$.

Inductive Clause. For any element x in S, $ax \in S$, and $bx \in S$.

Here ax means the concatenation of a with x.

Extremal Clause. Nothing is in S unless it is obtained from the Basis and Inductive Clauses.

1.14.1. Recursive Definition of Function

Some functions can also be defined recursively.

Condition : The domain of the function you wish to define recursively must be a set defined recursively.

How to define function recursively : First the values of the function for the basis elements of the domain are specified. Then the value of the function at an element, say x, of the domain is defined using its value at the parent(s) of the element x.

A few examples are given below.

They are all on functions from integer to integer except the last one.

Example 1.19. The function $f(n) = n!$ for natural numbers n can be defined recursively as follows.

Solution. The function f is the function that satisfies the following two clauses:

Basis Clause. $f(0) = 0! = 1$

Inductive Clause. For all natural number n, $f(n + 1) = (n + 1) f(n)$.

Note that here Extremal Clause is not necessary, because the set of natural numbers can be defined recursively and that has the extremal clause in it. So there is no chance of other elements to come into the function being defined.

Using this definition, 3 ! can be found as follows:

Since 0 ! = 1, 1 ! = 1 * 0 ! = 1 * 1 = 1,

Hence 2 ! = 2 * 1 ! = 2 * 1 = 2

Hence 3 ! = 3 * 2 ! = 3 * 2 * 1 = 6.

Example 1.20. The function $f(n) = 2n + 1$ for natural numbers n can be defined recursively as follows.

Solution. The function f is the function that satisfies the following two clauses :

Basis Clause. $f(0) = 1$

Inductive Clause. For all natural number n, $f(n + 1) = f(n) + 2$.

See above for the extremal clause.

EXERCISE

1. Let $L = \{ab, aa, baa\}$. Which of the following strings are in L^* : *abaab aaabaa, aaaabaaaa, baaaaabaaaab, baaaaabaa.*
2. Give some examples of strings in, and not in these sets, where $\Sigma = \{a, b\}$.
   (*a*) $\{w$: for sum $x \in \Sigma$, $w = x\, x^R x\}$
   (*b*) $\{w : ww = www\}$
   (*c*) $\{w$: for some $\mu, v \in \Sigma^*$, $\mu vw = wv\mu\}$
   (*d*) $\{w$: for some $\mu \in \Sigma^*$, $www = \mu\mu\}$.

3. Which of the following are true? Explain.
   (*a*) $baa \in a^* b^* a^* b^*$ (*b*) $b^* a^* \cap a^* b^* = a^* \cup b^*$
   (*c*) $a^* b^* \cap b^* c^* = \phi$ (*d*) $abcd \in (a(cd)^* b)^*$.
4. Write the following explicitly :
   (*i*) $\{1\} \times \{1, 2\} \times \{1, 2, 3\}$
   (*ii*) $\phi \times \{1, 2\}$.
5. Let $f : A \to B$ and $g : B \to C$. Let $h : A \to C$ be their composition. In each of the following cases state necessary and sufficient conditions on f and g for h to be as specified.
   (*i*) On to.
   (*ii*) One to one.
   (*iii*) A bijection.
6. Prove the following by principle of induction :
   (*i*) $\sum_{k=1}^{n} K^2 = \frac{n(n+1)(2n+1)}{6}$
   (*ii*) $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{(n+1)}$.
7. Prove the following by the principle of induction :
   (*i*) $2i > i$ for all $i > 1$
   (*ii*) $1 + 4 + 7 + \ldots + (3n - 2) = \frac{n(3n-1)}{2}$
8. If $X = \{a, b\}$ and $Y = \{b, c\}$, find
   (*i*) $(X \cup Y)^*$ (*ii*) $(X \cap Y)^*$
   (*iii*) $(X^* \cup Y^*)$ (*iv*) $X^* \cap Y^*$
   (*v*) $X - Y$ (*vi*) $(X - Y)^*$
   (*vii*) $(Y - X)^*$.
9. Let $f : \{a, b\}^* \to \{a, b\}^*$ be given by $f(x) = ax$ for every $x \in \{a, b\}^*$. Show that f is one to one not on to.
10. Prove the following by induction $2^x \geq x^2$ if $x \geq 4$.

*****

Chapter 2

Finite Automata

2.1. Introduction

In this book, we are discussing the mathematical model of the computers and algorithms. In next two chapter we are going to define powerful models of computation, more and more sophisticated device for accepting the generating languages, which are restricted model of the actual computers, called finite automata or finite state machine. These machines are very similar to the Central Processing Unit of a computer. Absence of the memory makes these machines more restricted model.

Computer is also deterministic, by which we mean that, on reading one particular input instruction, the machine converts itself from the state it was, into some particular other state, where the resultant state is completely determined by the prior state and the input instruction. Some sequence of instruction may lead to success and some may not. Success is entirely determined by the sequence of inputs. Either program will work or not.

Inside This Chapter

2.1. Introduction
2.2. Definition of Deterministic Finite Automata
2.3. Processing of strings by DFA
2.4. Transition Graph
2.5. Generalised Transition Graphs
2.6. Simpler Notations for DFA's
2.7. The Language of DFA
2.8. Extended Transition Function to Strings
2.9. Nondeterminism
2.10. The Language of NFA
2.11. Transformation of NFA to DFA
2.12. NFA with ∈-transitions
2.13. Acceptance of a string by NFA with ∈-transitions
2.14. Conversion of NFA with ∈-transitions to NFA without ∈-transitions
2.15. Minimization of DFA's
2.16. Myhill-Nerode Theorem
2.17. Implementation of Myhill-Nerode Theorem

Finite automation is called "finite" because number of possible states and number of letter in the alphabet are both finite, and "automation" because the change of the state is totally governed by the input. It is deterministic, since, what state is next is automatic not will-full, just as the motion of the hands of the clock is automatic, while the motion of hands of a human is presumably the result of desire and thought.

Before discussing the mathematical model let us discuss the pictorial representation of finite machine. Strings are fed into device by means of an input tape, which is divided into square, with one symbol in each square. The main part of the machine is a "black box", which is responsible for all proceedings. Let us say "black box" is the finite control, finite control can serve, that what symbol is written at any position on the input tape by means of a movable reading head. Initially the head is placed at the left most square of the tape and the finite control is set is designated initial state.

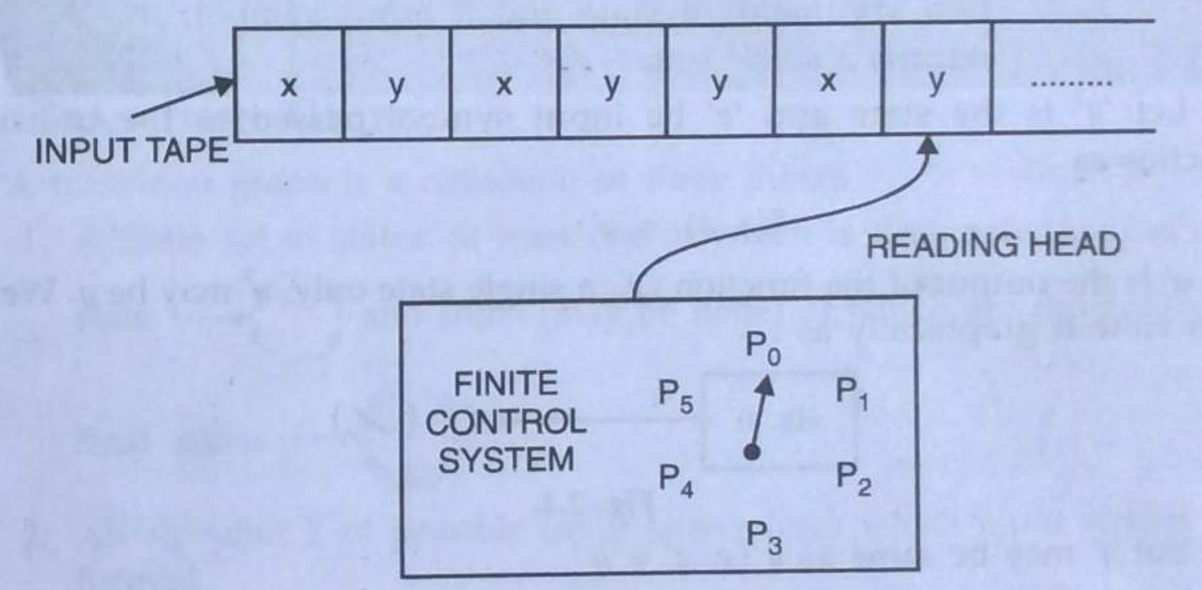


Fig. 2.1.

$P_0, P_1, P_2, P_3, P_4, P_5$ are states in finite control system x and y are input symbols.

At regular interval the automation reads one symbol from the input tape and then enters in a new state that depends only on the current state and the symbol just read.

After reading an input symbol, reading head moves one square to the right on the input tape, so that on the next move, it will read the symbol in next tape square. This process is repeated again and again.

The automation then indicates approval or disapproval.

If it winds up *in one of a set of final states* the input strings is considered to be accepted. The language accepted by the machine is the set of strings, it accepts.

2.2. Definition of Deterministic Finite Automata

Definition When we analyse above discussion mathematically the following formal version obtained.

A deterministic finite automata is a quintuple

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

where

Q : Is a non-empty finite set of states presents in the finite control. $(q_0, q_1, q_2, ...)$

Σ : Is a non-empty finite set of input symbols which can be passed to the finite state machine. $(a, b, c, d, e, ...)$

q_0 : Is a starting state, one of the state in Q.

F : Is a non-empty set of final states or accepting states, set of final states belongs to Q.

δ : Is a function called transition function that takes two arguments a state and a input symbol, it returns a *single* state.

Let 'q' is the state and 'a' be input symbol passed to the transition function as :

$$\delta(q, a) = q'$$

q' is the output of the function *i.e.*, a single state only, q' may be q. We can also view it graphically as :

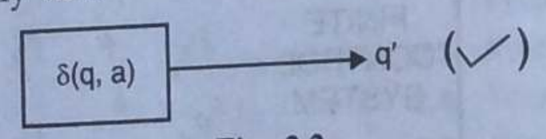


Fig. 2.2.

but q' may be same as q *i.e.* $q' = q$

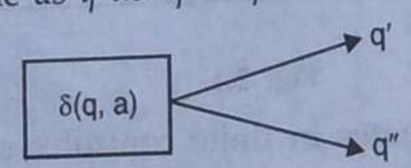


Fig. 2.3.

case discussed in Fig. 2.3 is not valid case in deterministic finite automata.

TIPS

Some people prefer to call this just a finite acceptor because its sole job is to accept certain input strings and rejects others. It does not do anything like printing machine. Sometime it is also referred as language recognizer because FA merely recognizes whether the input strings is in the language as we can also recognize that some body is speaking *Russian* or *English* without necessarily understanding what it means.

2.3. Processing of Strings by DFA

Suppose $a_1, a_2, a_3, ..., a_n$ is a sequence of input symbols. We start out with deterministic finite automata having $q_0, q_1, q_2, q_3, ..., q_n$ states where q_0 is the initial state and q_n is the final state and transition function and processed as

$$\delta(q_0, a_1) = q_1$$
$$\delta(q_1, a_2) = q_2$$
$$\delta(q_2, a_3) = q_3$$
$$\vdots \qquad \vdots$$
$$\delta(q_{n-1}, q_n) = q_n$$

Input $a_1, a_2, ..., a_n$ is said to be "**accepted**" since q_n is the member of the final states, and if not then it is "**rejected**". By adopting this approach we can define the language of a particular DFA, and can also check that particular string is accepted or rejected by given FA.

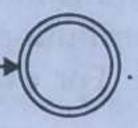
TIPS

"So a string is said to be accepted by the given FA only when it is processed by transition function δ in such a manner that it ends at the final state".

Before discussing the formal notations for the FA, let us know about the transition graph :

2.4. Transition Graph

A transition graph is a collection of three things :

1. A finite set of states, at least one of which is designated as the start state and some (may be none) of which are designated as final states  .
2. An alphabet Σ of possible input letters from which input strings are formed.
3. A finite set of transitions (edge labels) that show how to go from some states to some others, based on reading specified substrings of input letters (possibly even the null string).

Transition graph were invented by John Myhill in 1957. A *successful path* through a transition graph is a series of edges forming a path beginning at some start state (there may be several start states) and ending at a final state. If we concatenate in order the strings of letters that label each edge in the path, we produce a word that is accepted by the this machine.

Let us consider some examples of TG's :

(*a*)

This picture above represents a TG that accepts nothing, not even the null string, to be accept any thing, it must be final state.

(*b*) The machine

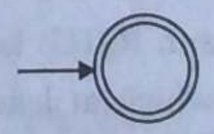


Fig. 2.4.

accept null string.

(c)

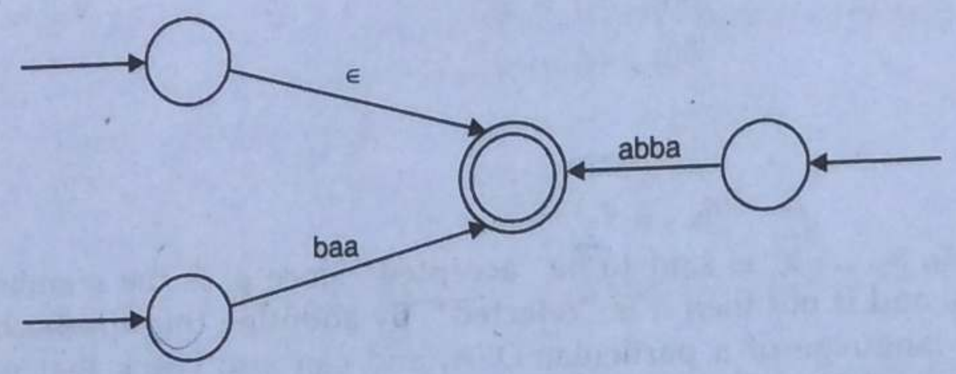


Fig. 2.5.

above TG accepts ∈, *baa* and *abba*.

(d)

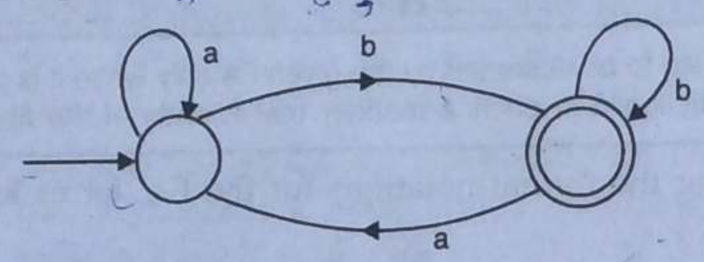


Fig. 2.6.

The language accepted by the TG is all words ending in *b* over alphabet $\Sigma = \{a, b\}$.

2.5. Generalised Transition Graphs

The ultimate step liberating state-to-state transition is to allow the input to progress from one place to another by contributing a substring restricted to being a word in a predetermined language. For example :

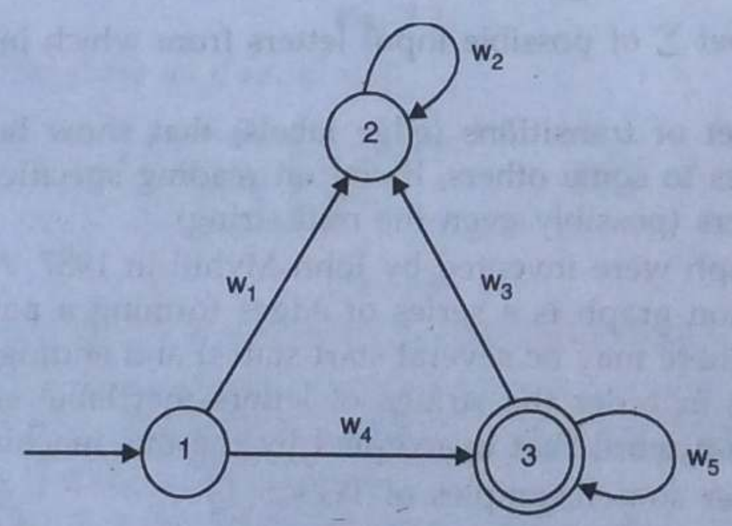


Fig. 2.7.

We can travel from start to state 2 by reading any (of course finite) word from the (possibly infinite) set of choices w_1 and, similarly, between all other states.

2.5.1. Definition of Generalised Transition Graph

A Generalised Transition Graph (GTG) is a collection of three things :

1. A finite set of states, of which at least one is a start state and some (may be none) are final states.

2. An alphabet Σ of input letters.
3. Directed edges connecting some pair of states, each labelled with a regular expression.

Example :

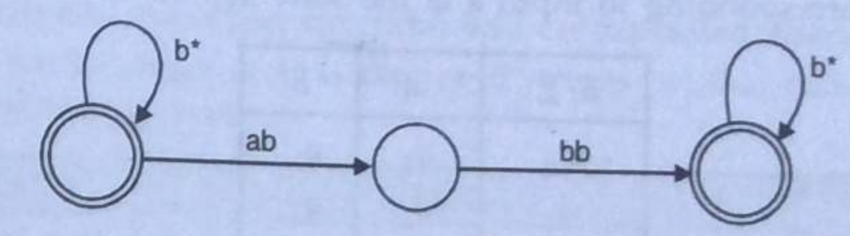


Fig. 2.8.

This machine accepts all strings without double *a*.

(Concept of Regular expression will be discussed in next chapter)

2.6. Simpler Notations for DFA's

There are two preferred notations for describing automata :

1. Transition Diagram.
2. Transition Table.

1. Transition Diagram Notations. A transition diagram for DFA, $M = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows :

(a) For each state in Q there is a node represented by the circle.

(b) For each state q in Q and each input symbol a in Σ, let $\delta(q, a) = P$. Then transition diagram has an arc from node q to node P, labelled a. If there are several input symbols that cause transition from q to P, then the transition diagram can have one arc, labelled by the list of these symbols as :

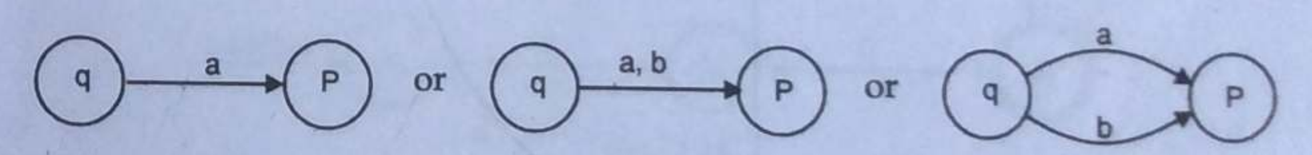


Fig. 2.9.

(c) If any state q in Q is the starting state then it represented by the circle with arrow as

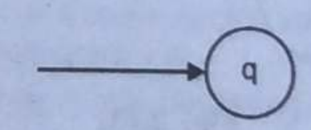


Fig. 2.10.

(d) Nodes corresponding to accepting states (those are in F) are marked by a double circle. States not in F have a single circle.

Fig. 2.11.

2. Transition Tables. A transition table is a Conventional, tabular representation of a function like δ that takes two arguments and returns a state. The rows of the table correspond to the states, and the columns correspond to the input. The entry for one row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$. For example :

δ/Σ	a	b
$\rightarrow q_0$	q_1	q_2
q_1	q_2	q_0
$*q_2$	q_2	q_2

Fig. 2.12.

q_0 is the starting state and q_2 is the final state,

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

Let us draw the transition diagram for the transition table shown in Fig. 2.12.

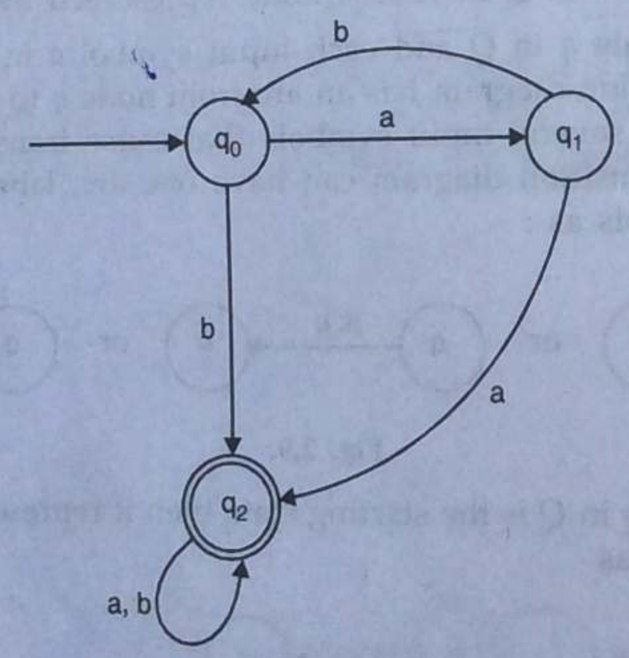


Fig. 2.13.

So clearly transition diagram can be find from transition table and vice-versa is also true.

2.7. The Language of a DFA

It has been explained informally that DFA defines a language : The set of all strings that result in a sequence of state transition from start state to an accepting state.

Now we can define the language of a DFA $M = (Q, \Sigma, \delta, q_0, F)$. This language is denoted by $L(M)$, and defined by

$$L(M) = \left\{w / \hat{\delta}(q_0, w) \text{ is in } F\right\}$$

* $\hat{\delta}$ (extended transition function) will be explained afterwords.

That is the language of M is the set of strings (w) that take the start q_0 to one of the accepting states.

TIPS

If L is $L(M)$ for some deterministic finite automata, then we say L is a regular language.

Now for practical understanding of FA, let us discuss the some examples.

Example 2.1. Design a FA that accepts set of strings such that every string ends in 00, over alphabet

$$\{0, 1\} \textit{ i.e., } \Sigma = \{0, 1\}$$

Solution. Let $M = (Q, \Sigma, \delta, q_0, F)$ be the DFA

q_0 : initial state.

Σ : $\{0, 1\}$ is given.

So according to the problem we have to design a FA which accepts strings $w = w'$ 00.

Where w' can be any combination of 0's and 1's for example 011100, 111100, 00, 100, 01010100 etc.

Now first we will decide the approach to design FA. It is not an easy task to think FA as a whole. So first we have to fulfill minimum condition *i.e.* every string end in 00.

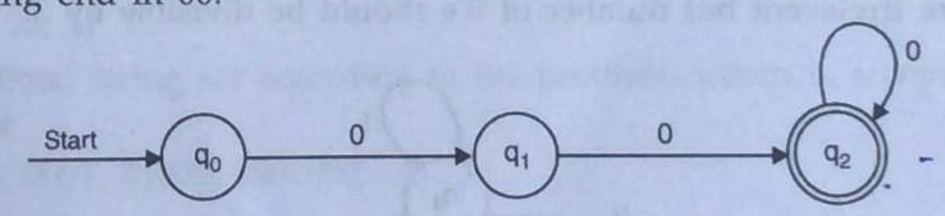


Fig. 2.14.

TIPS

Being the DFA we must check every input symbol for output state from every state. So we have to decide output state at symbol 1 from q_0, q_1 and q_2. Then it will be complete FA.

It is important to know that in the designing procedure, FA must accept all those strings which are in the given language but should not accept those strings which are not in the language of Finite Automata.

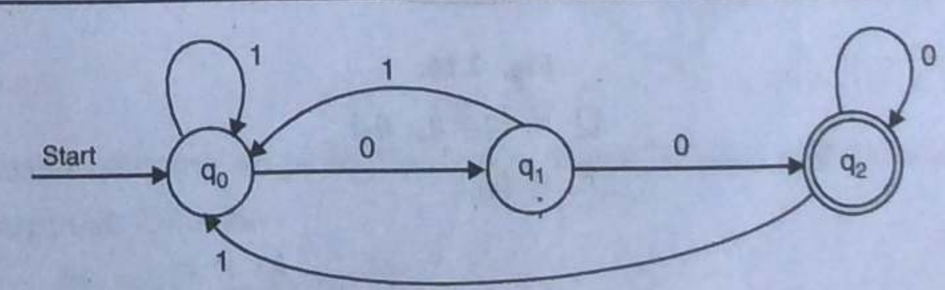


Fig. 2.15.

So Fig. 2.14 satisfied the minimum condition and Fig. 2.15 is the transition diagram of required FA.

$$Q = \{q_0, q_1, q_2\}$$
$$F = \{q_2\}$$

Facts in Designing Procedure of FA

On the basis of example 2.1, we can decide some *facts or observation* for FA.

(*a*) In the designing of the FA, first of all we have to analyse the set of strings *i.e.* language which is accepted by the FA.

(*b*) Make sure that every state is check for the output state for every input symbol of Σ.

(*c*) Make sure that no state must have two different output state for a single input symbol.

(*d*) Make sure, that there is one initial state and at least one final state in transition diagram of FA.

Consideration of above discussed points in the designing, will minimize the error.

Example 2.2. Construct a FA that accepts set of strings where the number of 0's in every string is multiple of three over alphabet $\Sigma = \{0, 1\}$.

Solution. Multiple of three means numbers of 0's in the strings may be 0, 3, 6, 9, 12, ... in the number at any position.

Let FA be $M = (Q, \Sigma, \delta, q_0, F)$

Here it is necessary to consider acceptance of Null ($\in$) string since it contains number of 0's zero times which is divisible by three so it must be very clear that here initial state *i.e.* q_0 is also final state. Here positions of 0's and 1's are irrelevent but number of 0's should be divisible by 3.

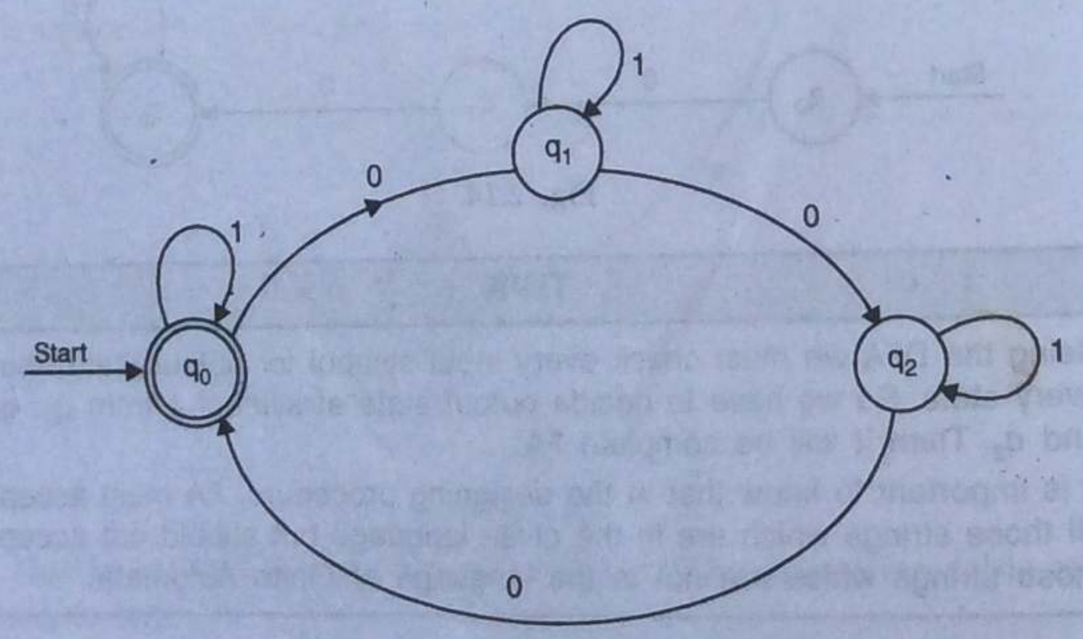


Fig. 2.16.

$$Q = \{q_0, q_1, q_2\}$$
$$F = \{q_0\}$$

Example 2.3. Design a FA which accepts set of strings containing exactly four 1's in every string over alphabet $\Sigma = \{0, 1\}$.

Solution. DFA should accepts the strings 1111, 0101011, 0110110000 ..... .

Let DFA be $M = (Q, \Sigma, \delta, q_0, F)$

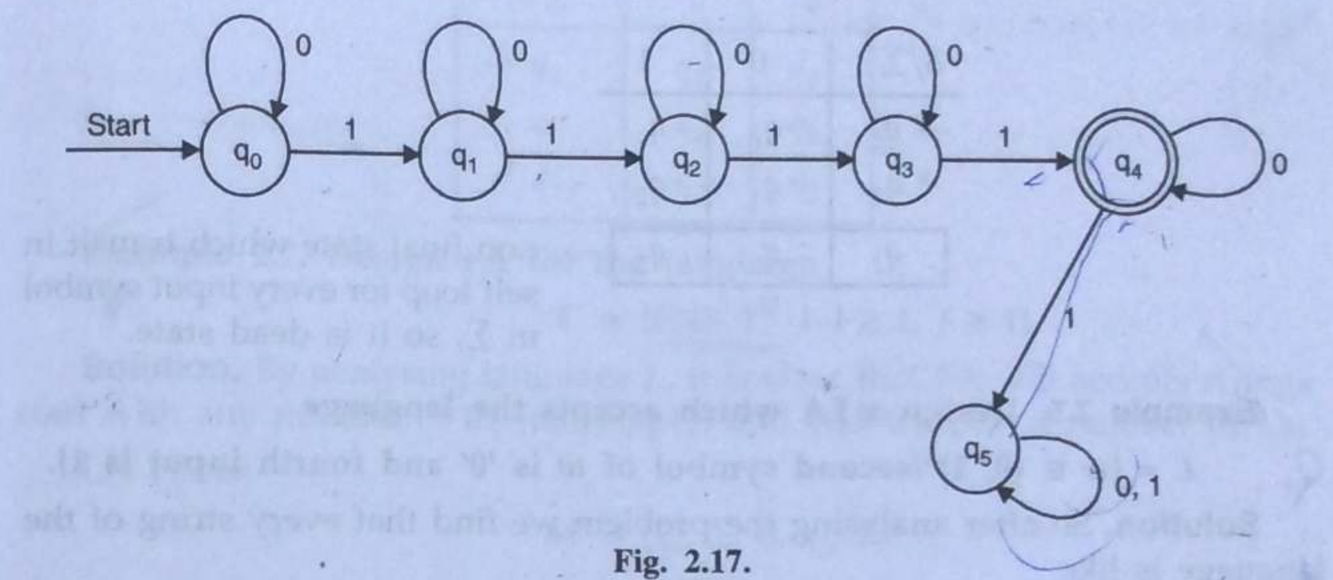


Fig. 2.17.

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$$
$$F = \{q_4\}$$

From q_0 to q_4 transitions four 1's are already there and q_4 is final state so 1 from q_4 should go on a trap path, that is why q_5 is there so q_5 is called trap or dead state.

"Dead states are those non final states which transits in itself for all the input symbol. So q_5 is the example of dead state."

Example 2.4. Design a FA that accepts strings containing exactly 1 over alphabet {0, 1}

Solution. String set according to the problem, which is accepted by the FA is like

1, 0001, 01000, 0001000, ...

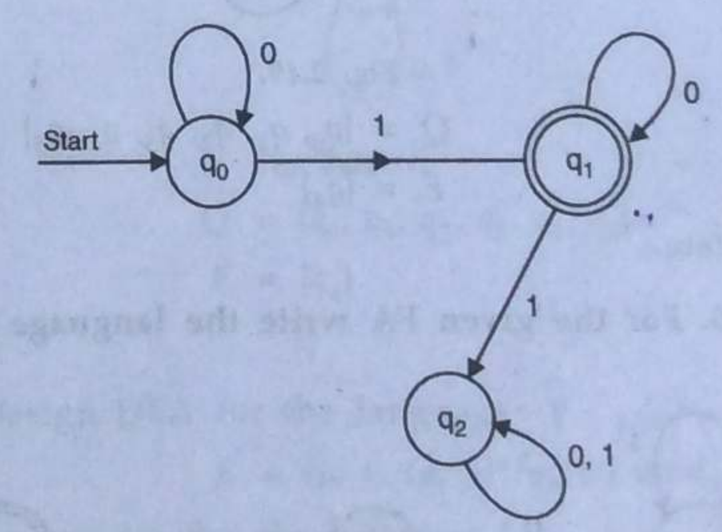


Fig. 2.18.

So clearly starting state to final state there is only one transition of 1.

Let suppose DFA be

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

$Q = \{q_0, q_1, q_2\}$

$F = \{q_1\}$

q_2 is the dead state.

Transition table for above FA

δ/Σ	0	1
$\rightarrow q_0$	q_0	q_1
$* q_1$	q_1	q_2
q_2	q_2	q_2

$\rightarrow$ non final state which transit in self loop for every input symbol in Σ, so it is dead state.

Example 2.5. Design a FA which accepts the language

$$L = \{w \in (0, 1)^*/\text{second symbol of } w \text{ is '0' and fourth input is 1}\}.$$

Solution. So after analysing the problem we find that every string of the language is like

0001, 1011, 10110000, ...

Let FA is $M = (Q, \Sigma, \delta, q_0, F)$

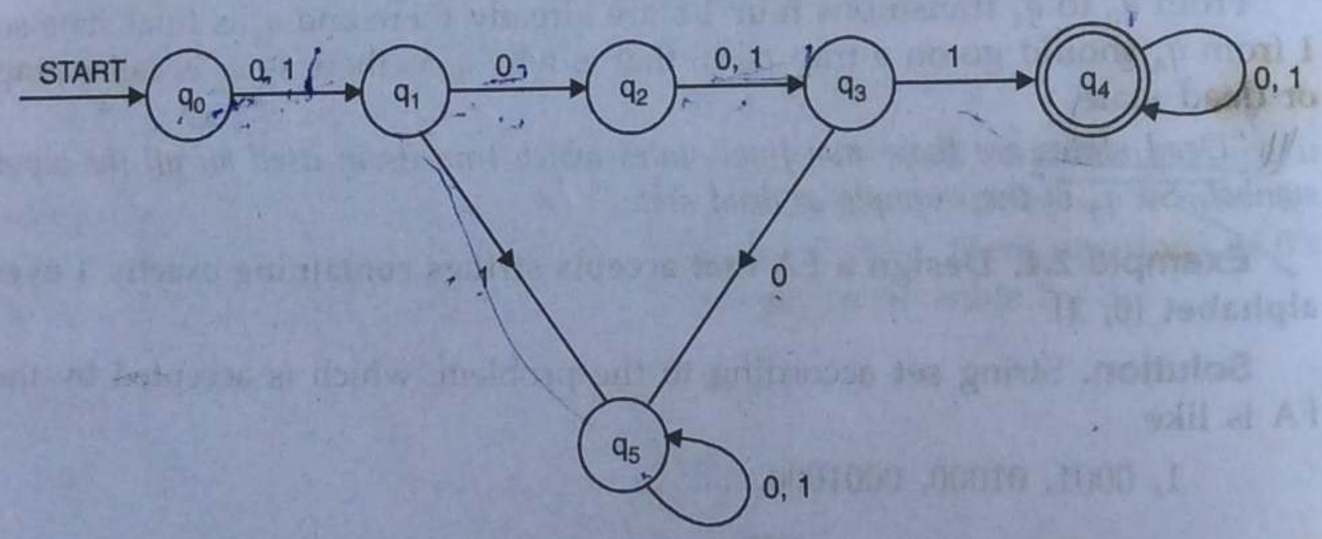


Fig. 2.19.

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$F = \{q_4\}$

q_5 is dead state.

Example 2.6. For the given FA write the language and also give the transition table

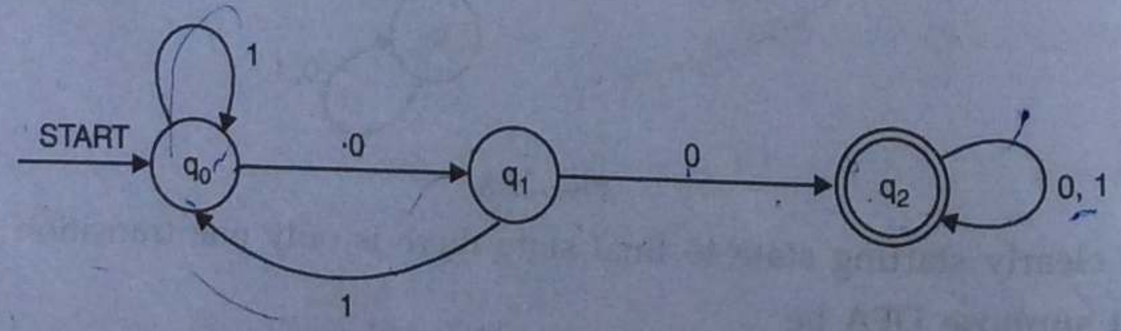


Fig. 2.20.

Solution. Let the language be L for FA shown in Fig. 2.20.

$L = \{w \in (0, 1)^*/\text{every string } w \text{ of the language containing 00 as substring}\}$.

Transition table :

δ/Σ	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_0
$* q_2$	q_2	q_2

Example 2.7. Design FA for the language

$$L = \{(01)^i\, 1^{2j} \mid i \geq 1, J \geq 1\}.$$

Solution. By analysing language L, it is clear that FA will accepts strings start with any number of 01 (not empty) and end with even number of 1's.

Let FA be

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

q : initial state.

δ : is transition function.

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

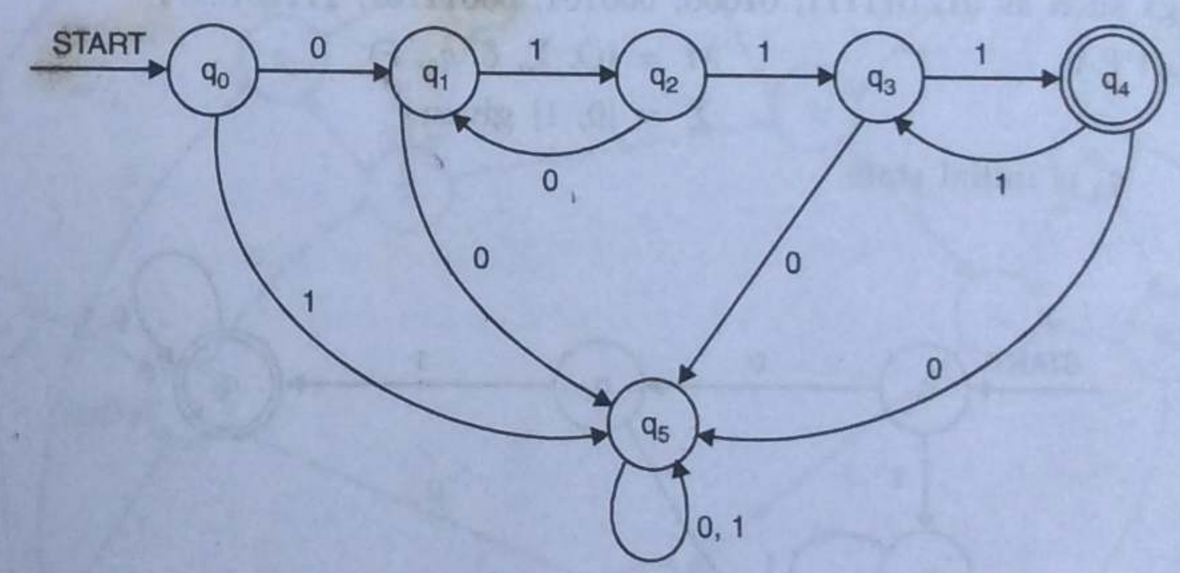


Fig. 2.21.

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$F = \{q_4\}$

q_5 is dead state.

Example 2.8. Design DFA for the language

$$L = \{w \in (a, b)^*/n_b(w) \bmod 3 > 1\}.$$

Solution. Let us first analyse the language L.

$n_b(w)$: number of b's in every string of language.

$L_b(w)$ mod 3 : number of b's in string is divided by three and remainder is result of $n_b(w)$ mod 3.

$n_b(w)$ mod 3 > 1 means that remainder is only 2 since when any number is divided by 3 then remainder may be only 0, 1 and 2.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be FA

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

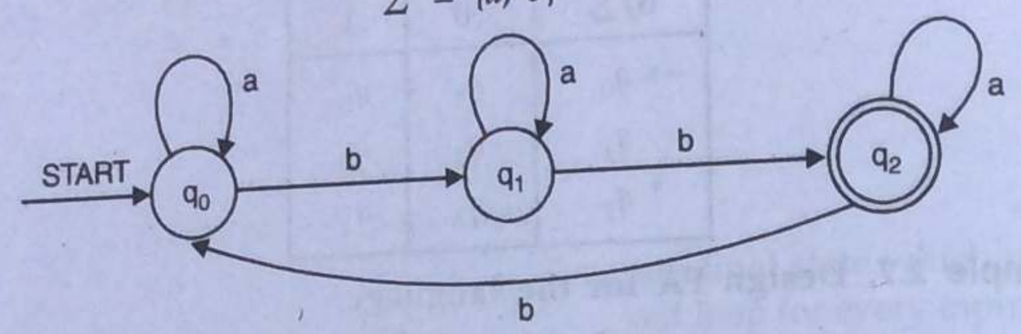


Fig. 2.22.

It will accept string, having $2b$'s, $5b$'s and $8b$'s so remainder is always two.

$Q = \{q_0, q_1, q_2\}$

$F = \{q_2\}$

Example 2.9. Design a FA over alphabet $\Sigma = \{0, 1\}$, which accepts the set of strings either start with 01 or end with 01.

Solution. By the analysis of problem, it is clear that FA will accepts the strings such as 01, 011111, 01000, 000101, 00011101, 111101..... .

Let FA $M = \{Q, \Sigma, \delta, q_0, F)$

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

q_0 is initial state.

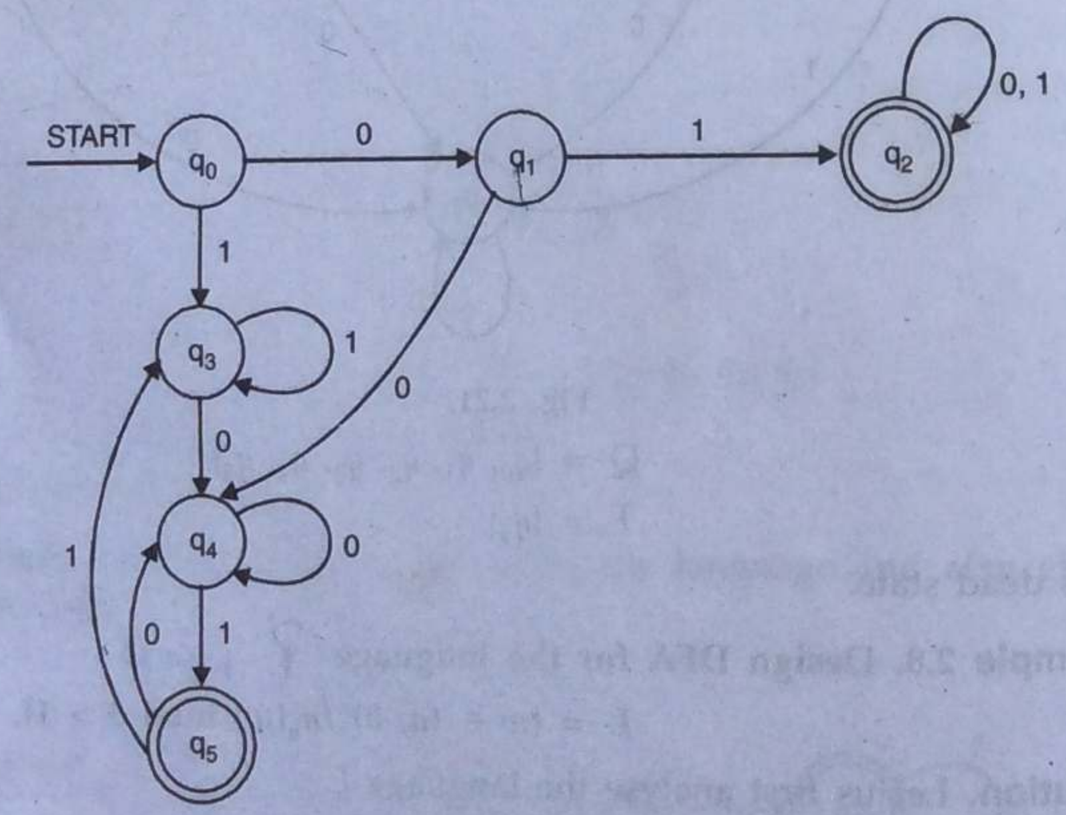


Fig. 2.23.

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$F = \{q_2, q_5\}$

Example 2.10. Give the DFA accepting the set of strings over alphabet $\Sigma = \{0, 1\}$, such that in each string number of 0's is divisible by five *and* number of 1's is divisible by 3.

Solution. Here two conditions *i.e.* number of 0's is divisible by five *and* number of 1's is divisible by 3 must satisfied simultaneously. So clearly only those state of FA will be final which satisfied both the condition simultaneously.

Let initial state is q_0, number of 0's passed at q_0 are zero which is divisible by five similarly number of 1's passed at q_0 are zero which is also divisible by three so clearly q_0 is final state.

Let FA be

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

$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}\}$

$F = \{q_0\}$

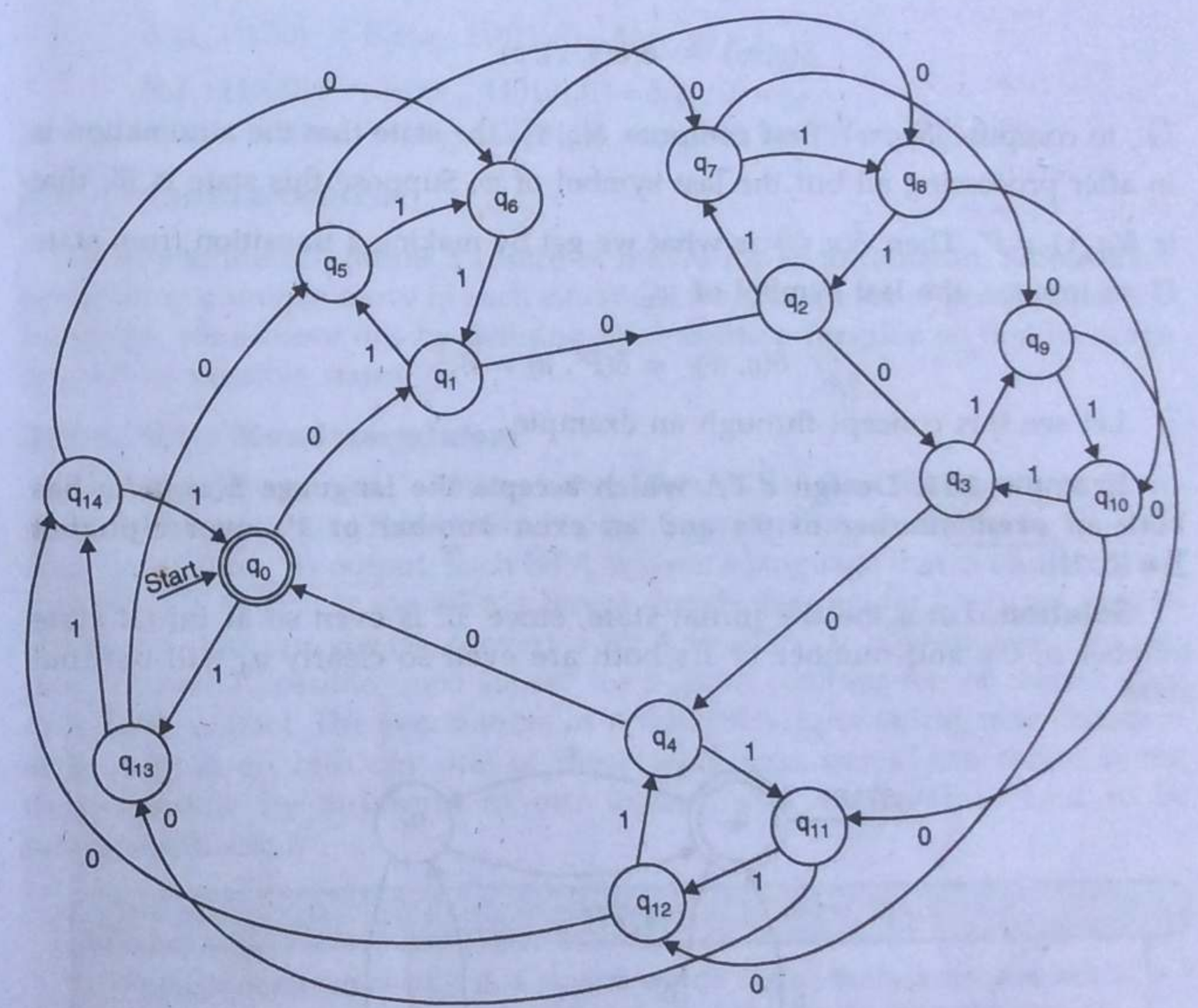


Fig. 2.24.

2.8. Extending Transition Function to Strings

In terms of the transition diagram, the language of a DFA is the set of labels along the paths that lead from the start state to any accepting state.

Now, we need to make the notation of the language of FA precise. To do so we define extended transition function that describes what happens when we start in any state and follow any sequence of inputs. If δ is our transition function, the extended function constructed from δ will be called $\hat{\delta}$.

The extended transition function is the function that takes a state q and a string w and returns a state P, the state that automation reaches when starting in state q and processing the sequence of inputs w. We define $\hat{\delta}$ by induction on the length of the input string as follows :

Basis : $\hat{\delta}(q, \in) = q$. That is, if we are in state q and read no input, then we are still in state q.

Induction. Suppose w is a string of the form xa, that is a is the last symbol of w and x is the substring of w, consisting of all except the last symbol 'a'. For example

$w = 1101$ is broken into $x = 110$ and $a = 1$ then

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$$

i.e., to compute $\hat{\delta}(q, w)$, first compute $\hat{\delta}(q, x)$, the state that the automation is in after processing all but the last symbol of w. Suppose this state is P'; that is $\hat{\delta}(q, x) = P'$. Then $\hat{\delta}(q, w)$ is what we get by making a transition from state P' on input a, the last symbol of w.

$$\hat{\delta}(q, w) = \delta(P', a) = P$$

Let see this concept through an example.

Example 2.11. Design a FA which accepts the language $L = \{w/w$ has both an even number of 0's and an even number of 1's over alphabet $\Sigma = \{0, 1\}\}$.

Solution. Let q_0 be the initial state, since '0' is even so at initial state number of 0's and number of 1's both are even so clearly q_0 will be final state

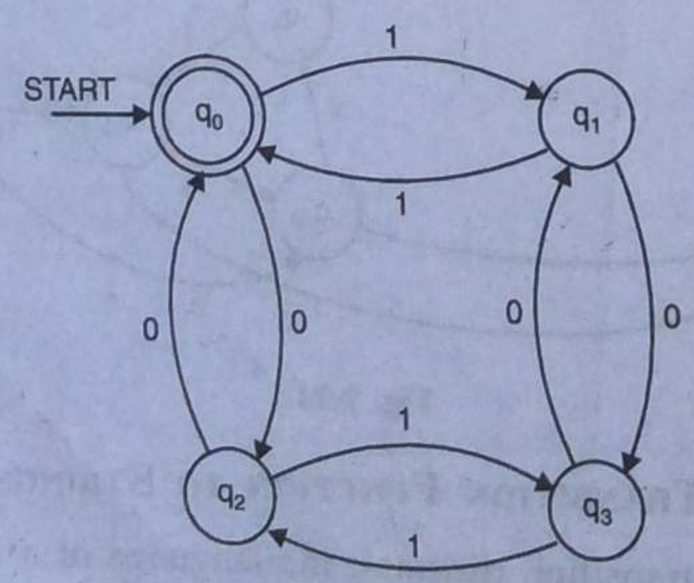


Fig. 2.25.

Let FA be $M = (Q, \Sigma, \delta, q_0, F)$

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

q_0 : both 1 and 0's are even

q_1 : 0's are even and 1's are odd

q_2 : 1's are even and 0's are odd

q_3 : both are odd.

Check 110101 string on above FA *i.e.* it is accepted by FA or rejected. Suppose extended transition function is $\hat{\delta}$,

$$\hat{\delta}(q_0, \in) = q_0$$

$$\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \in), 1) = \delta(q_0, 1) = q_1$$

$$\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$$

$$\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$$

$$\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_3$$

$$\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 0) = q_1$$

$$\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 0) = q_0$$

and q_0 is final state so string 110101 is accepted by FA in Fig. 2.25.

2.9. NONDETERMINISM

Nondeterminism means a choice of moves for an automaton. Rather than prescribing a unique move in each situation, we allow a set of possible moves. Formally, we achieve this by defining the transition function so that its range is a set of possible states.

2.9.1. Why Nondeterminism

A "Nondeterministic" Finite Automation (NFA) has the power to be in several states at once. This ability is often expressed as an ability to "guess", something about its output. Each NFA accepts a language that is also accepted by some DFA. That is the NFA's accept exactly the regular language, just as DFA's do. We can always convert a NFA to DFA. It is clear now, we can permit several possible "next states" for a given combination of current state and input symbol. The automation, as it reads the input string, may choose at each step to go into any one of these legal next states, the choice is not deterministic by anything in our model, and is therefore said to be nondeterminstic.

TIPS

Though nondeterministic is a feature we do not normally associate with real computers, it arises naturally as an extension of the behaviour of deterministic finite automata and in some cases greatly simplifies the description of these automata. Moreover, nondeterminism is an essential feature of FA, every NFA is equivalent to FA.

Thus, we profit from the greater flexibility of nondeterminism, when we wish to design language recognizers without actually extending the class of language we are able to describe.

Definition Now let me introduce the formal notions associated with nondeterministic finite automata. The difference between DFA's and NFA's will be pointed out as we do.

Let M be NFA then M is defined as

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

where Q : is a non empty finite set of states in NFA.

Σ : is a non-empty finite set of input symbols.

q_0 : initial state of NFA and it is the member of Q.

F : a non-empty finite set of final states in Q.

δ : is the transition function that takes a states and Q and an input symbol in Σ as arguments and returns a subset of Q.

that is $Q \times (\Sigma \cup \{\in\}) \rightarrow 2^Q$

So in nondeterministic accepter, the range of δ is in the power set 2^Q, so its value is not a single element of Q, but a subset of it. This subset defines the set of possible states that can be reached by transition.

Let q be an state of Q and $a \in \Sigma$ then

We can transit from a state q on same input a to different states $q_0, q_1, \ldots, q_n$ in Q. This not possible in the case of FA.

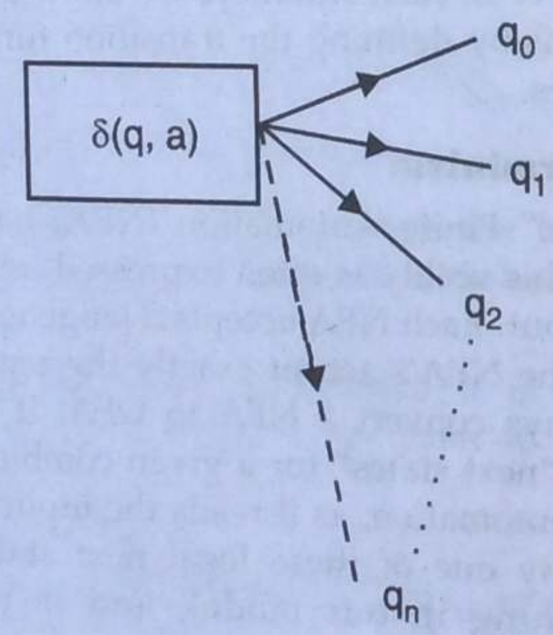


Fig. 2.26.

Example 2.12. Design a NFA for the language L = all strings over {0, 1} that have at least two consecutive 0's or 1's.

Solution. By the analysis it is clear that NFA will accepts the strings of the pattern 00, 11, 101000, 101100, ...

Let NFA be

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

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

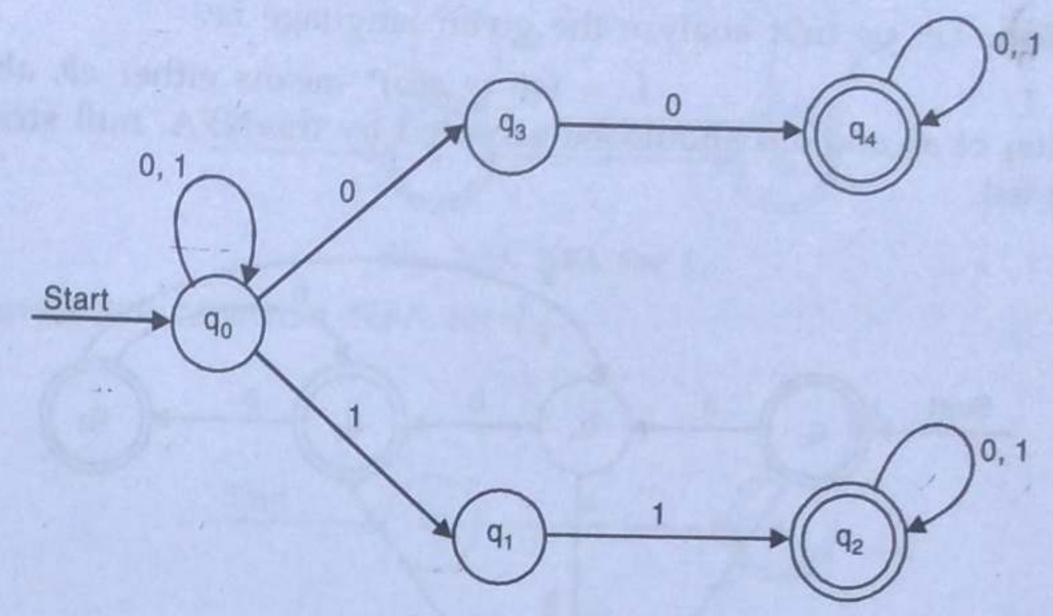


Fig. 2.27.

Note that state q_3 does not go anywhere for an input of 1. **We use the terminology that the path "dies" if in q_3 getting an input 1.**

The tree of states this NFA is in for the input 0100011.

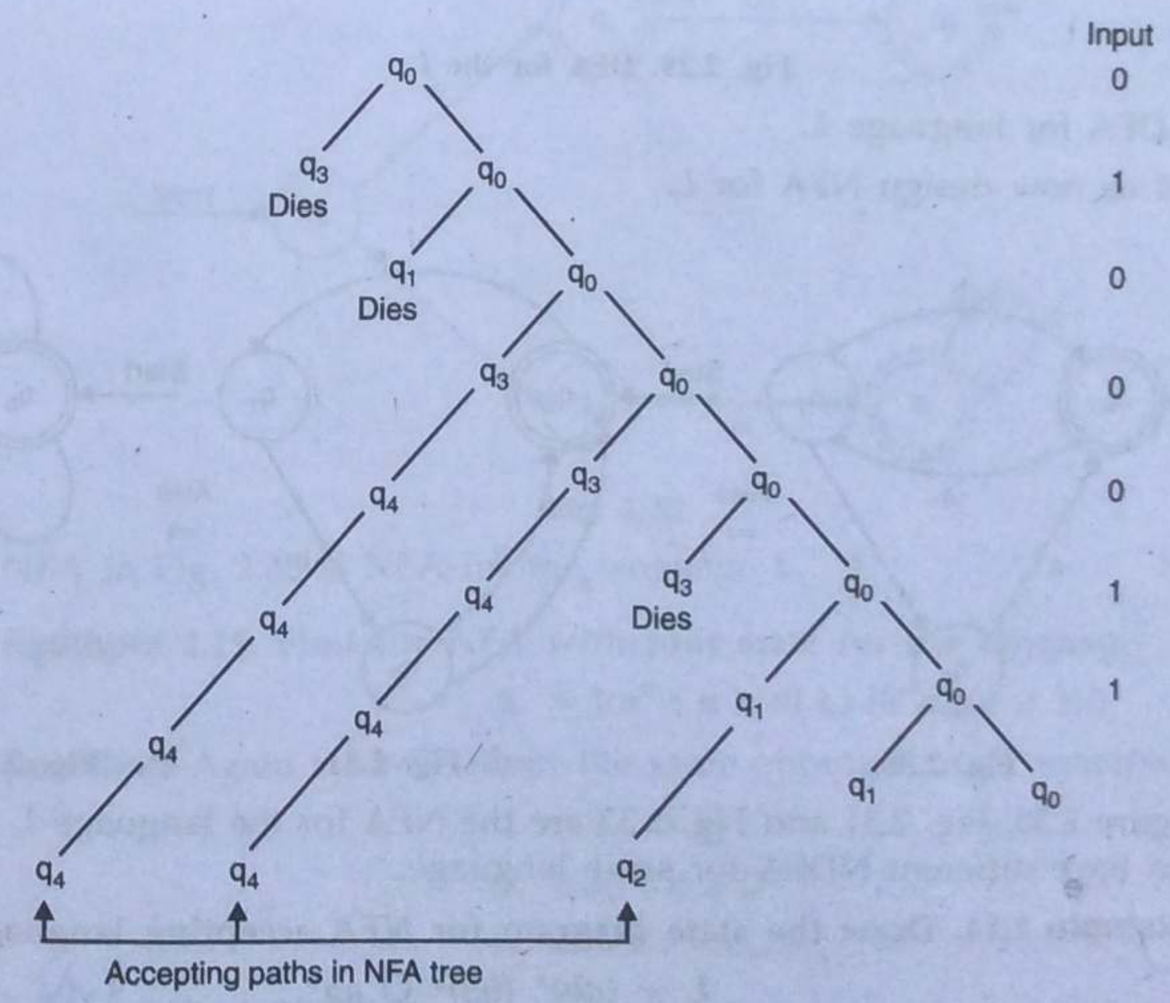


Fig. 2.28.

So it is clear from figure 2.28 that for a particular string there may be various paths, some may lead it to accepted states, and some may lead it to non accepted states and some may lead it to the blocked paths as $q_0 \rightarrow q_3$ as q_3 dies.

Example 2.13. Design a NFA for the language $L = (ab \cup aba)^*$.

Solution. Let us first analyse the given language *i.e.*

$L = (ab \cup aba)^*$ means either *ab*, *aba* or any combination of *ab* and *aba* should be accepted by the NFA. null string ($\in$) is also accepted.

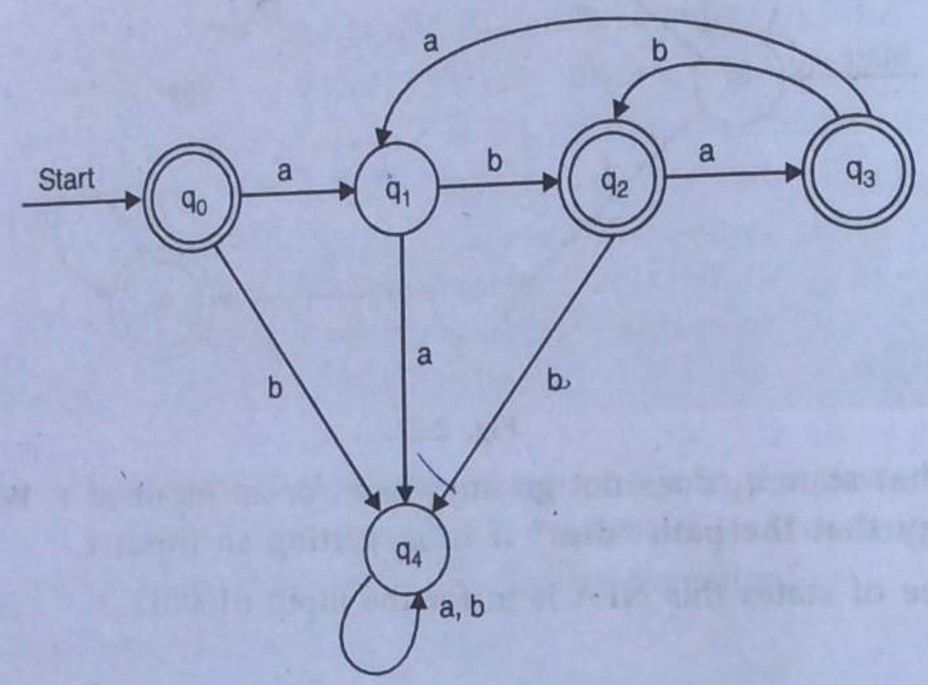


Fig. 2.29. DFA for the *L*.

If DFA for language *L*.

Let us now design NFA for *L*.

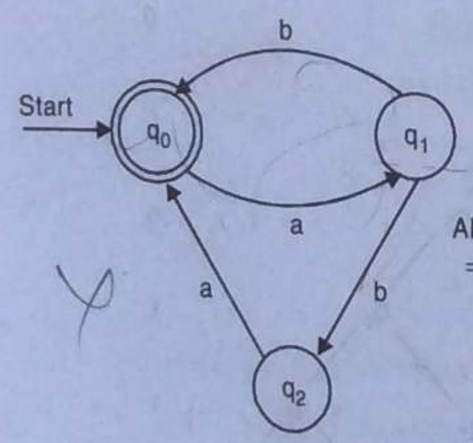


Fig. 2.30.

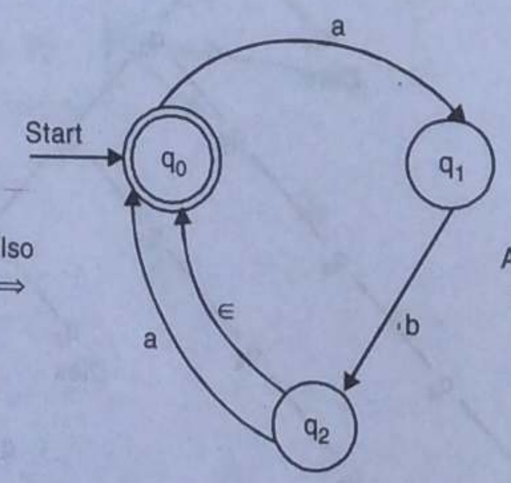


Fig. 2.31.

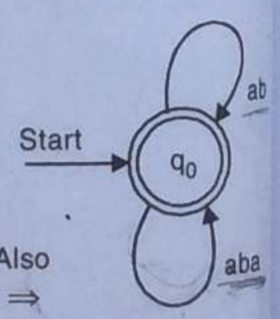


Fig. 2.32.

Figure 2.30, Fig. 2.31 and Fig. 2.32 are the NFA for the language *L*. Thus we can have different NDFA for same language.

Example 2.14. Draw the state diagram for NFA accepting language

$$L = (ab)^* (ba)^* \cup aa^*.$$

Solution. We can construct NFA for the language *L* in two paths *i.e.*

$$L = L_1 \cup L_2$$
$$L_1 = (ab)^* (ba)^*$$
$$L_2 = aa^*$$

First we can construct NFA for L_1

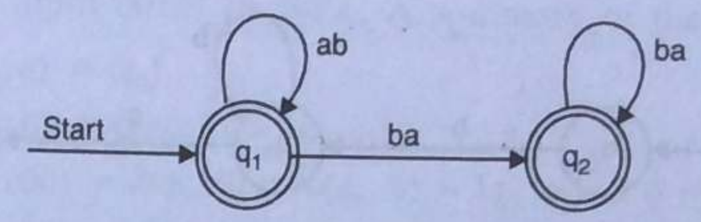


Fig. 2.33. NFA for L_1.

Secondly we construct NFA for L_2.

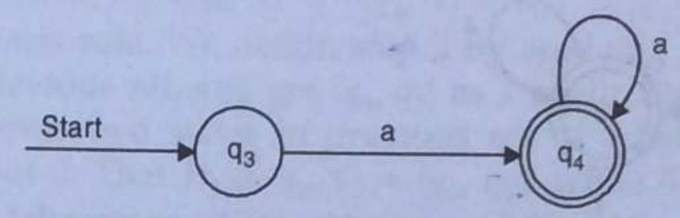


Fig. 2.34.

Now we can combine NFA in Fig. 2.33 and 2.34. For combining we will introduce another state q_0 and will connect both NFA with Null transitions as follows :

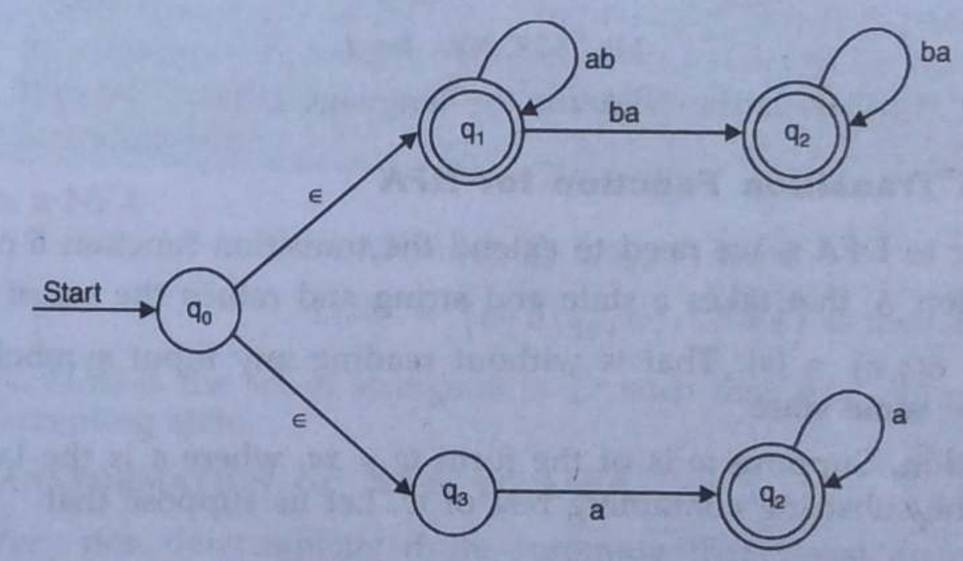


Fig. 2.35.

NFA in Fig. 2.35 is NFA for the language *L*.

Example 2.15. Find the NFA with four state for the language

$$L = \{(a^n : n \geq 0) \cup (b^n a : n \geq 1)\}.$$

Solution. Again we will adopt the same approach as in example 2.14.

$$L = L_1 \cup L_2$$
$$L_1 = a^n : n \geq 0$$
$$L_2 = b^n a : n \geq 1$$

NFA for the language L_1 is

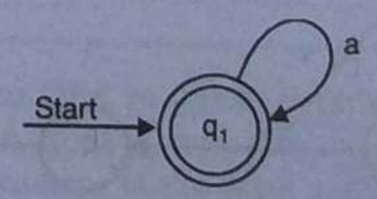


Fig. 2.36.

NFA for the language L_2 is

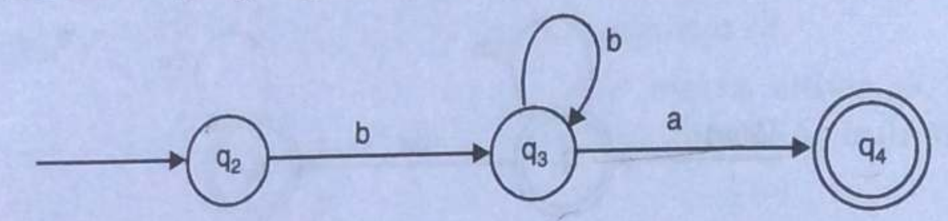


Fig. 2.37.

Now combine NFA in Fig. 2.36 and Fig. 2.37 as

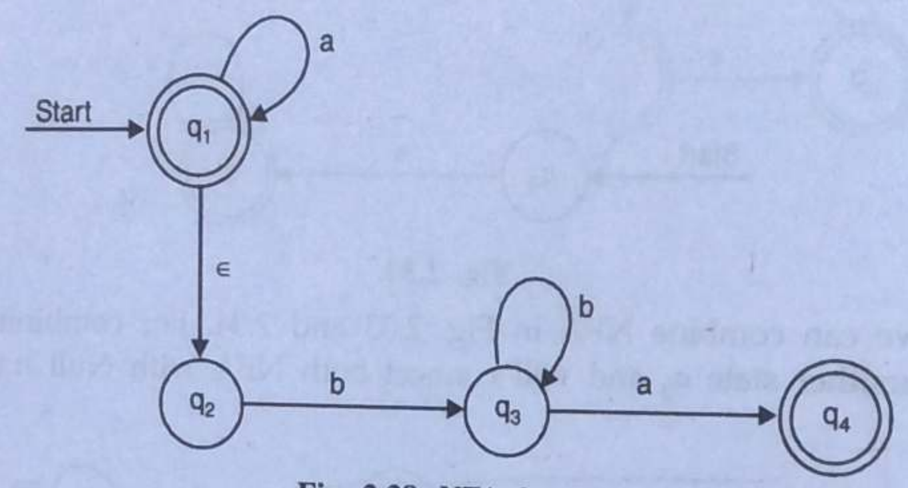


Fig. 2.38. NFA for *L*.

Finally Fig. 2.38 is the NFA for the language L.

Extended Transition Function for NFA

Similar to DFA's, we need to extend the transition function δ of an NFA to a function $\hat{\delta}$ that takes a state and string and return the set of states.

Basis. $\hat{\delta}(q,\in) = \{q\}$. That is without reading any input symbols, we are only in the same state.

Induction. Suppose w is of the form $w = xa$, where a is the last symbol and x is the substring containing rest of w. Let us suppose that

$$\hat{\delta}(q,x) = \{p_1, p_2, \dots p_k\},$$

Let

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, r_3, \dots, r_m\}$$

Then $\hat{\delta}(q, w) = \{r_1, r_2, \dots r_m\}$. Less formally, we compute $\hat{\delta}(q, w)$ by first computing $\hat{\delta}(q, x)$, and by then following any transition from any of these states that is labeled a. Let us see an example for better understanding of the concept.

Example 2.16. Consider NFA in Fig. 2.39, and describe the processing of string 00101 by it.

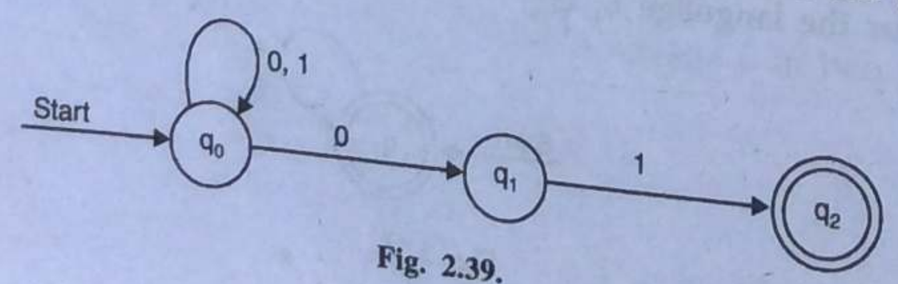


Fig. 2.39.

Solution. It is NFA accepting all strings end in 01. Let us use $\hat{\delta}$ to describe the processing of input 00101 by NFA. A summary of the steps is.

Step 1. $\hat{\delta}(q_0, \in) = \{q_0\}$

Step 2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$

Step 3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \phi = \{q_0, q_1\}$

Step 4. $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Step 5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \phi = \{q_0, q_1\}$

Step 6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

Step 1 is the basis rule. We obtain step 2 by applying δ to the done state q_0, that is in the previous set, and get $\{q_0, q_1\}$ as a result. Step 3 is obtained by taking the union over two states in previous set of what we get when we apply δ to the input 0. That is, $\delta(q_0, 0) = \{q_0, q_1\}$, while $\delta\{q_1, 0\} = \phi$.

For step 4, we take union of $\delta(q_0, 1) = q_0$ and $\delta(q_1, 1) = \{q_2\}$. So step 5 and step 6 are obtained in similar way.

2.10. The Language of NFA

Definition As we discussed, an NFA accepts a string w if it is possible to make any sequence of choice of next state while reading the characters of w, and go from the start state to any accepting state.

If M is a NFA

$$M = (Q, \Sigma, \delta, q_0, F) \text{ then}$$

$$L(M) = \{w / \hat{\delta}(q_0, w) \cap F \neq \phi\}$$ is the Language of M *i.e.* NFA. $L(M)$ is the set of strings w in Σ^* such that $\hat{\delta}(q_0, w)$ contains at least one accepting state.

2.11. Transformation of NFA to DFA

For every non deterministic finite automata there exist an equivalent deterministic finite automata. The equivalence is defined in terms of language acceptance. Since a NFA is a nothing but a finite automata in which zero, one or more transition on an input symbol is permitted, we can always construct a finite automata which will simulate all the moves of NFA on a particular input symbol in parallel, then get a finite automata in which there will be exactly one transition on every input symbol, hence it will be DFA equivalent to NFA.

TIPS

Since the DFA equivalent of NFA is to simulate the moves in parallel state of a DFA will be combination of one or more states of NFA, hence every state of a DFA will be represented by some subset of set of states of NFA, and therefor the transition of NFA to DFA is normally called a subset construction.

So transformation of NFA to DFA in value, finding subset of set of states of NFA. The conversion of an NFA to DFA involves constructing all subsets of the states of NFA. The subset constructing starts from an NFA $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$. Its goal is the description of DFA $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ such that $L(M_1) = L(M_2)$.

The input alphabets of two automata are the same, and start state of M_2 is the set containing only the start state of M_1. The other component of M_2 are constructed as follows :

(*i*) Q_2 is the set of subsets of Q_1, *i.e.* Q_2 is the power set of Q_1. If Q_1 has n states then Q_2 will have 2^n states. Often not all these states are accessible from the start state of Q_2. In accessible states can be thrown away, so effectively the number of states of M_2 may be much smaller then 2^n.

(*ii*) F_2 is all sets of M_1's states that include at least one accepting state of F_1.

(*iii*) For each subset S of Q_1 and for each input symbol a in Σ,

$$\delta_2(S, a) = U_{P \text{ in } S}\ \delta_2(P, a)$$
$$S = \{q_1, q_2, ..., q_i\} \in Q_1$$

That is to compute $\delta_2(S, a)$, we look at all the states in P in S, see what states M_1 goes from P on input a and take the union of all those states.

When the NFA M_1 has n states the corresponding DFA M_2 has 2^n states. However we need not construct M_2 for all these 2^n states, but only for those states reachable from starting state. So we start the construction of M_2 from q_0. We continue by considering only states appearing earlier under input columns and constructing M_2 for such states. We halt when no more states appear under the input columns.

Let us implement this concept in the form of set of steps as algorithm through an example.

Example 2.17. Convert the following NFA into DFA.

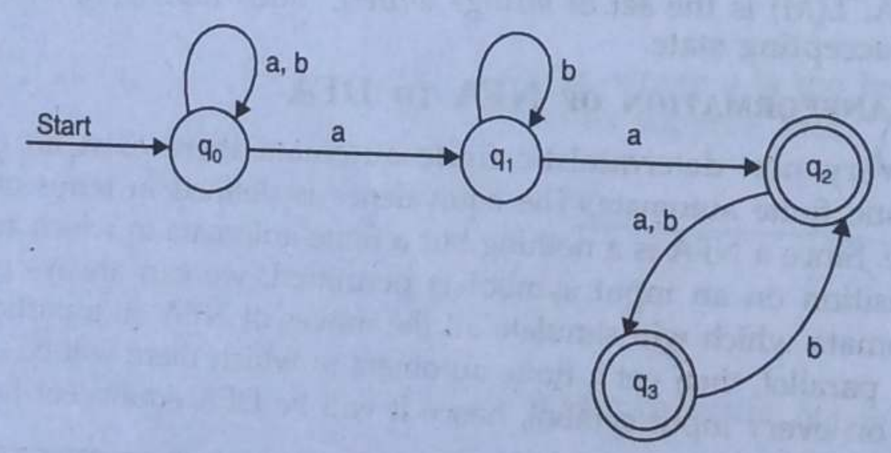


Fig. 2.40.

Solution.

Step 1. Seek all the transition from starting state q_0 for every symbol in Σ *i.e.* (a, b). If you get a set of states for same input then consider that set as new single state as :

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

in Fig. 2.40 q_0 transit to both q_0 and q_1 for input a.

$$\delta(q_0, b) = \{q_0\}$$

Step 2. In step 1 we are getting a new state $\{q_0, q_1\}$. Now repeat step 1 for this new state only *i.e.* check all transitions of a and b (that is Σ) from $\{q_0, q_1\}$ as :

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

Step 3. Repeat step 2 till you are getting any new state. All those states which consists at least one accepting state of given NFA as member state will be considered as final states.

$[q_0, q_1, q_2]$ is new state.

$$\delta(\{q_0, q_1, q_2\}, a) = \delta(q_0, a) \cup \delta(q_1, a) \cup \delta(q_2, a)$$
$$= \{q_0, q_1\} \cup \{q_2\} \cup \{q_3\}$$
$$= \{q_0, q_1, q_2, q_3\} \quad \text{(New state)}$$
$$\delta(\{q_0, q_1, q_2\}, b) = \delta(q_0, b) \cup (q_1, b)\ \delta(q_2, b)$$
$$= q_0 \cup q_1 \cup q_3 = \{q_0, q_1, q_3\} \quad \text{(New state)}$$
$$\delta(\{q_0, q_1, q_2, q_3\}, a) = \{q_0, q_1, q_2, q_3\} \quad \text{(Old state)}$$
$$\delta(\{q_0, q_1, q_2, q_3\}, b) = \{q_0, q_1, q_3, q_2\} \quad \text{(Old state)}$$
$$\delta(\{q_0, q_1, q_3\}, a) = \{q_0, q_1, q_2, q_3\} \quad \text{(Old state)}$$
$$\delta(\{q_0, q_1, q_3\}, b) = \{q_0, q_1, q_3, q_2\} \quad \text{(Old state)}$$

Let draw transition table

δ/Σ	a	b
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_3\}$
$*\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$
$*\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_2, q_3\}$	$\{q_0, q_1, q_2, q_3\}$

Let us say

$$q_0 \rightarrow A$$
$$\{q_0, q_1\} \rightarrow B$$
$$\{q_0, q_1, q_2\} \rightarrow C$$
$$\{q_0, q_1, q_2, q_3\} \rightarrow D$$
$$\{q_0, q_1, q_3\} \rightarrow E$$

A is initial state and C, D and E are final states since they contain q_2 and q_3 as member which are final states of NFA.

So

δ/Σ	a	b
$\rightarrow A$	B	A
B	C	B
$*C$	D	E
$*D$	D	D
$*E$	D	D