TOC
UNIT-1

Vision and Mission of Institute

Vision:

To become a renowned centre of outcome based learning, and work towards academic, professional, cultural and social enrichment of the lives of individuals and communities.

Mission:

- Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.
- Identify areas of focus and provide platform to gain knowledge and solutions based on informed perception of Indian, regional and global needs.
- Offer opportunities for interaction between academia and industry.
- Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

Vision and Mission of Department of Information Technology

Vision:

To establish outcome based excellence in teaching, learning and commitment to support IT Industry.

Mission:

M1: *To provide outcome based education.*

M2: *To provide fundamental & Intellectual knowledge with essential skills to meet current and future need of IT Industry across the globe.*

M3: *To inculcate the philosophy of continuous learning, ethical values & Social Responsibility.*

Syllabus:

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

SYLLABUS

II Year- IV Semester: B.Tech. (Information Technology)

4IT4-06: Theory of Computation

Credit: 3 **Max. Marks: 150(IA:30, ETE:120)**

3L+0T+0P **End Term Exam: 3 Hours**

SN	Contents	Hours
1	**Introduction:** Objective, scope and outcome of the course.	1
2	Finite Automata & Regular Expression: Basic machine, Finite state machine, Transition graph, Transition matrix, Deterministic and non-deterministic finite automation, Equivalence of DFA and NDFA, Decision properties, minimization of finite automata, Mealy & Moore machines. Alphabet, words, Operations, Regular sets, relationship and conversion between Finite automata and regular expression and vice versa, designing regular expressions, closure properties of regular sets, Pumping lemma and regular sets, Myhill- Nerode theorem , Application of pumping lemma, Power of the languages.	7
3	Context Free Grammars (CFG), Derivations and Languages, Relationship between derivation and derivation trees, leftmost and rightmost derivation, sentential forms, parsing and ambiguity, simplification of CFG, normal forms, Greibach and Chomsky Normal form , Problems related to CNF and GNF including membership problem.	8
4	Nondeterministic PDA, Definitions, PDA and CFL, CFG for PDA, Deterministic PDA, and Deterministic PDA and Deterministic CFL , The pumping lemma for CFL's, Closure Properties and Decision properties for CFL, Deciding properties of CFL.	8
5	Turing Machines: Introduction, Definition of Turing Machine, TM language Acceptors and Transducers, Computable Languages and functi Universal TM & Other modification, multiple tracks Turing Machine. Hierarchy of Formal languages: Recursive & recursively enumerable languages, Properties of RL and REL, Introduction of Context sensitive grammers and languages, The Chomsky Hierarchy.	8
6	Tractable and Untractable Problems: P, NP, NP complete and NP hard problems, Un-decidability, examples of these problems like vertex cover problem, Hamiltonian path problem, traveling sales man problem.	8
	Total	**40**

Office of Dean Academic Affairs
Rajasthan Technical University, Kota

Program Educational Objectives (PEO):

1. To enrich students with fundamental knowledge, effective computing, problem solving and communication skills enable them to have successful career in Information Technology.
2. To enable students in acquiring Information Technology's latest tools, technologies and management principles to give them an ability to solve multidisciplinary engineering problems.
3. To impart students with ethical values and commitment towards sustainable development in collaborative mode.
4. To imbibe students with research oriented and innovative approaches which help them to identify, analyze, formulate and solve real life problems and motivates them for lifelong learning.
5. To empower students with leadership quality and team building skills that prepare them for employment, entrepreneurship and to become competent professionals to serve societies and global needs.

Program Outcomes (PO):

1. **Engineering Knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems in IT.
2. **Problem analysis:** Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences in IT.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations using IT.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions using IT.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations in IT.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice using IT.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development in IT.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice using IT.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings in IT.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project Management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage IT projects and in multidisciplinary environments.
12. **Life –long Learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological changes needed in IT.

4IT4-06: Theory of Computation

Course Outcomes (COs)

CO1- Able to design and understand and basic properties of DFA & NDFA and formal languages and formal grammars.

CO2- Able to understand the relation between types of languages and types of finite automata and the Context free languages and grammar's, and also Normalizing CFG

CO3- Able to design & understand the minimization of deterministic and nondeterministic finite automata & the concept of Pushdown automata and its application.

CO4- Able to understand basic properties of Turing machines and computing with Turing machines and concepts of tractability and decidability, the concepts of NP-completeness and NP-hard Problem, the challenges for Theoretical Computer Science and its contribution to other sciences.

CO-PO Mapping:

H=3, M=2, L=1

			PO1	**PO2**	**PO3**	**PO4**	**PO5**	**PO6**	**PO7**	**PO8**	**PO9**	**PO10**	**PO12**	**PO12**
V Sem.	**4IT4-06: Theory of Computation**	**CO-1**	H	M	H	H	M	M	M	-	M	M	L	M
		CO-2	H	H	H	H	M	M	M	-	M	M	L	M
		CO-3	H	M	H	H	M	M	M	-	M	M	L	M
		CO-4	H	M	H	H	M	M	M	-	M	M	M	M

1 Mathematical Preliminaries

1.1 Set Theory

Definition 1 (Set). A *set* is collection of distinct elements, where the order in which the elements are listed does not matter. The size of a set S, denoted $|S|$, is known as its *cardinality* or *order*. The members of a set are referred to as its elements. We denote membership of x in S as $x \in S$. Similarly, if x is not in S, we denote $x \not\in S$.

Example 1. Common examples of sets include the set of real numbers $\mathbb{R}$, the set of rational numbers $\mathbb{Q}$, and the set of integers $\mathbb{Z}$. The sets $\mathbb{R}^+, \mathbb{Q}^+$ and $\mathbb{Z}^+$ denote the strictly positive elements of the reals, rationals, and integers respectively. We denote the set of natural numbers $\mathbb{N} = \{0, 1, \ldots\}$. Let $n \in \mathbb{Z}^+$ and denote $[n] = \{1, \ldots, n\}$.

We now review several basic set operations, as well as the power set. It is expected that students will be familiar with these constructs. Therefore, we proceed briskly, recalling definitions and basic examples intended solely as a refresher.

Definition 2. Set Union Let A, B be sets. Then the *union* of A and B, denoted $A \cup B$ is the set:

$$A \cup B := \{x : x \in A \text{ or } x \in B\}$$

Example 2. Let $A = \{1, 2, 3\}$ and $B = \{4, 5, 6\}$. Then $A \cup B = \{1, 2, 3, 4, 5, 6\}$.

Example 3. Let $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$. So $A \cup B = \{1, 2, 3, 4, 5\}$. Recall that sets do not contain duplicate elements. So even though 3 appears in both A and B, 3 occurs exactly once in $A \cup B$.

Definition 3. Set Intersection Let A, B be sets. Then the *intersection* of A and B, denoted $A \cap B$ is the set:

$$A \cap B := \{x : x \in A \text{ and } x \in B\}$$

Example 4. Let $A = \{1, 2, 3\}$ and $B = \{1, 3, 5\}$. Then $A \cap B = \{1, 3\}$. Now let $C = \{4\}$. So $A \cap C = \emptyset$.

Definition 4 (Symmetric Difference). Let A, B be sets. Then the *symmetric difference* of A and B, denoted $A \triangle B$ is the set:

$$A \triangle B := \{x : x \in A \text{ or } x \in B, \text{ but } x \not\in A \cap B\}$$

Example 5. Let $A = \{1, 2, 3\}$ and $B = \{1, 3, 5\}$. Then $A \triangle B = \{2, 5\}$.

For our next two definitions, we let U be our *universe*. That is, let U be a set. Any sets we consider are subsets of U.

Definition 5 (Set Complementation). Let A be a set contained in our universe U. The *complement* of A, denoted A^C or $\overline{A}$, is the set:

$$\overline{A} := \{x \in U : x \not\in A\}$$

Example 6. Let $U = [5]$, and let $A = \{1, 2, 4\}$. Then $\overline{A} = \{3, 5\}$.

Definition 6 (Set Difference). Let A, B be sets contained in our universe U. The *difference* of A and B, denoted $A \setminus B$ or $A - B$, is the set:

$$A \setminus B = \{x : x \in A \text{ and } x \not\in B\}$$

Example 7. Let $U = [5]$, $A = \{1, 2, 3\}$ and $B = \{1, 2\}$. Then $A \setminus B = \{3\}$.

Remark: The Set Difference operation is frequently known as the *relative complement*, as we are taking the complement of B relative to A rather than with respect to the universe U.

Definition 7 (Cartesian Product). Let A, B be sets. The *Cartesian product* of A and B, denoted $A \times B$, is the set:

$$A \times B := \{(a, b) : a \in A, b \in B\}$$

Example 8. Let $A = \{1, 2, 3\}$ and $B = \{a, b\}$. Then $A \times B = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$.

Definition 8 (Power Set)**.** Let S be a set. The *power set* of S, denoted 2^S or $\mathcal{P}(S)$, is the set of all subsets of S. Formally:

$$2^S := \{A : A \subset S\}$$

Example 9. Let $S = \{1, 2, 3\}$. So $2^S = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Remark: For finite sets S, $|2^S| = 2^{|S|}$; hence, the choice of notation.

Definition 9 (Subset)**.** Let A, B be sets. A is said to be a *subset* of B if for every $x \in A$, we have $x \in B$ as well. This is denoted $A \subset B$ (equivocally, $A \subseteq B$). Note that B is a *superset* of A.

Example 10. Let $A = [3], B = [6], C = \{2, 3, 5\}$. So we have $A \subset B$ and $C \subset B$. However, $A \not\subset C$ as $1 \not\in C$; and $C \not\subset A$, as $5 \not\in A$.

Remark: Let S be a set. The subset relation forms a partial order on 2^S. To show two sets A and B are equal, we must show $A \subset B$ and $B \subset A$. We demonstrate how to prove two sets are equal below.

Proposition 1.1. *Let $A = \{6n : n \in \mathbb{Z}\}, B = \{2n : n \in \mathbb{Z}\}, C = \{3n : n \in \mathbb{Z}\}$. So $A = B \cap C$.*

Proof. We first show that $A \subset B \cap C$. Let $n \in \mathbb{Z}$. So $6n \in A$. We show $6n \in B \cap C$. As 2 is a factor of 6, $6n = 2 \cdot (3n) \in B$. Similarly, as 3 is a factor of 6, $6n = 3 \cdot (2n) \in C$. So $6n \in B \cap C$. We now show that $B \cap C \subset A$. Let $x \in B \cap C$. Let $n_1, n_2 \in \mathbb{Z}$ such that $x = 2n_1 = 3n_2$. As 2 is a factor of x and 3 is a factor of x, it follows that $2 \cdot 3 = 6$ is also a factor of x. Thus, $x = 6n_3$ for some $n_3 \in \mathbb{Z}$. So $x \in A$. Thus, $B \cap C \subset A$. Thus, $A = B \cap C$, as desired. □

Proposition 1.2. *Let A, B, C be sets. Then $A \times (B \cup C) = (A \times B) \cup (A \times C)$.*

Proof. Let $(x, y) \in A \times (B \cup C)$. If $y \in B$, then $(x, y) \in (A \times B)$. Otherwise, $y \in C$ and so $(x, y) \in (A \times C)$. Thus, $A \times (B \cup C) \subset (A \times B) \cup (A \times C)$. Now let $(d, f) \in (A \times B) \cup (A \times C)$. Clearly, $d \in A$. So f must be in either B or C. Thus, $(d, f) \in A \times (B \cup C)$, which implies $(A \times B) \cup (A \times C) \subset A \times (B \cup C)$. We conclude that $A \times (B \cup C) = (A \times B) \cup (A \times C)$. □

1.2 Relations and Functions

Definition 10 (Relation)**.** Let X be a set. A k-ary relation on X is a subset $R \subset X^k$.

Example 11. The notion of equality $=$ over $\mathbb{R}$ is the canonical example of a relation. It is perhaps the most well-known instance of an *equivalence relation*, which will be discussed later.

Intuitively, a k-ary relation R contains k-tuples of elements from X that share common properties. Computer scientists and mathematicians are interested in a number of different relations, including the adjacency relation (graph theory), equivalence relations, orders (such as partial orders), and functions. In this section, functions, asymptotics, and equivalence relations will be discussed.

1.2.1 Functions

The notion of a *function* will be introduced first. Functions are familiar mathematical objects, which appear early on in mathematics education with the notion of an input-output machine. Roughly speaking, a function takes an input and produces an output. Some common examples include the linear equation $f(x) = ax + b$ and the exponential $f(x) = 2^x$.

We denote a function as follows. Let X and Y be sets. A function is a map $f : X \to Y$ such that for every $x \in X$, there is a unique $y \in Y$ where $f(x) = y$. We say that X is the *domain* and Y is the *codomain*. The *range* or *image* is the set $f(X) = \{f(x) : x \in X\}$. More formally, a function is defined as follows:

Definition 11. Function Let X and Y be sets. A function f is a subset (or 1-place relation) of $X \times Y$ such that for every $x \in X$, there exists a unique $y \in Y$ where $(x, y) \in f$.

Let's consider some formal functions and one example of a relation that is not a function.

Example 12. Let $f : \mathbb{R} \to \mathbb{R}$ be given by $f(x) = x$. This is known as the *identity map*.

Example 13. Let $g : \mathbb{R} \to \mathbb{R}$ be given by $g(x) = 3x^2$.

Example 14. Let $h : \mathbb{R} \to \mathbb{R}$ given by:

$$h(x) = \begin{cases} x & : x \neq 3 \\ -3, 2 & : x = 3 \end{cases}$$

Note that h is *not* a function as $(3, -3) \in h$ and $(3, 2) \in h$. The definition of a function states that there must be a *unique* y such that $(3, y) \in h$. If we revise h such that $h(3) = -3$ *only*, then h satisfies the definition of a function.

From a combinatorial perspective, special types of functions known as *injections* and *surjections* are of great importance. The idea is that if we have two sets X and Y and know the cardinality of X, then an injection or surjection from X to Y yields results about Y's cardinality. We can deduce similar results about Y's cardinality.

An injection is also known as a *one-to-one* function. Recall the definition of a function states that the map $f : X \to Y$ maps each $x \in X$ to a unique $y \in Y$. It allows for functions such as $g : \mathbb{R} \to \mathbb{R}$ given by $g(x) = 0$. Clearly, every $x \in \mathbb{R}$ maps to the same y-coordinate: $y = 0$. An injection disallows functions such as these. The idea is that each $y \in Y$ can be paired with at most one $x \in X$, subject to the constraint that each element in X must be mapped to some element from Y. So there can be unmapped elements in Y, but not in X.

We define this formally as follows.

Definition 12 (Injection)**.** A function $f : X \to Y$ is said to be an injection if $f(x_1) = f(x_2) \implies x_1 = x_2$. Equivocally, f is an injection if $x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$.

Let's consider examples of functions that are injections, as well as those that fail to be injections.

Example 15. Let X be a set. Recall the identity map $\text{id} : X \to X$ given by $\text{id}(x) = x$. This function is an injection. Let $\text{id}(x_1) = \text{id}(x_2)$. Then we have $\text{id}(x_1) = x_1 = \text{id}(x_2) = x_2$, which implies that $x_1 = x_2$.

Example 16. Consider the function $g : \mathbb{R} \to \mathbb{R}$ be given by $g(x) = x^2$. Observe that g fails to be an injection. Let $g(x_1) = g(x_2) = 4$. We have $x_1 = 2$ and $x_2 = -2$, both of which map to 4. If we instead consider $h : \mathbb{R}^+ \to \mathbb{R}$ by $h(x) = x^2$, we have an injection since we only consider the positive real numbers. Observe as well that both g and h do not map to any element less than 0.

Remark: Let's reflect on what we know about injections. An injection is a function, in which any mapped element in the codomain is mapped to exactly once. There may be elements in the codomain which remain unmapped. As a result, for two sets X and Y, it is defined that $|X| \leq |Y|$ if there exists an injection $f : X \to Y$. Intuitively speaking, an injection pairs each element from the domain with an element in the codomain, allowing for leftover elements in the codomain. Hence, X has no more elements than Y if there exists an injection $f : X \to Y$.

Surjections or onto functions center around the codomain, rather than the domain. Recall the earlier example of $g : \mathbb{R} \to \mathbb{R}$ given by $g(x) = x^2$, which satisfies $g(x) \geq 0$ for every $x \in \mathbb{R}$. So the negative real numbers will never be maped under g. Surjections exclude functions like g. Intuitively, a function is a surjection if every element in the codomain is mapped. Any element of the codomain can have multiple domain points mapping to it, as long as each has at least one domain point mapping to it. We define this formally as follows.

Definition 13 (Surjection)**.** Let X and Y be sets. A function $f : X \to Y$ is a surjection if for every $y \in Y$, there exists an $x \in X$ such that $f(x) = y$.

We have already seen an example of a function that is not a surjection. Let us now consider a couple examples of functions that are surjections.

Example 17. Recall the identity map $\text{id} : X \to X$. For any $x \in X$, we have $\text{id}(x) = x$. So the identity map is a surjection.

Example 18. Let $X = \{a, b, c, d\}$ and let $Y = \{1, 2, 3\}$. Define $f : X \to Y$ by $f(a) = f(b) = 1, f(c) = 2$ and $f(d) = 3$. This function is a surjection, as each $y \in Y$ is mapped under f. Observe that there are more X elements than Y elements. If X instead had two elements, then f would not be a surjection because at most two of the three elements in Y could be mapped.

Remark: Similarly, let's now reflect upon what we know about surjections. A surjection is a function in which every element of the codomain is mapped at least once. Some elements in the codomain may have multiple elements in the domain mapping to them. Therefore, if there exists a surjection $f : X \to Y$, then $|X| \geq |Y|$. We now introduce the notion of a bijection. A function is a bijection if it is both an injection and a surjection. Intuitively, a bijection matches the elements in the domain and codomain in a one-to-one manner. That is, each element in the domain has precisely one mate in the codomain and vice versa. For this reason, two sets X and Y are defined to have the same cardinality if there exists a bijection $f : X \to Y$. Combinatorialists use bijections to ascertain set cardinalities. The idea is that given sets X and Y, with $|Y|$ known, can we construct a bijection $f : X \to Y$? If the answer is yes, then $|X| = |Y|$.

Definition 14 (Bijection)**.** Let X and Y be sets. A bijection is a function $f : X \to Y$ that is both an injection and a surjection.

Example 19. Some examples of bijections include the identity map, as well as the linear equation $f(x) = mx + b$.

We conclude by showing that the composition of two injective functions are injective, and that the composition of two surjective functions are surjective. This implies that the composition of two bijections is itself a bijection, which is an important fact when working with permutations (which we shall see later).

Proposition 1.3. *Let $f : X \to Y$ and $g : Y \to Z$ be injective functions. Then $g \circ f$ is also injective.*

Proof. Let $x_1, x_2 \in X$ be distinct. As f is injective, $f(x_1) \neq f(x_2)$. Similarly, as g is injective, $g(f(x_1)) \neq g(f(x_2))$. So $(g \circ f)(x_1) \neq (g \circ f)(x_2)$, as desired. As x_1, x_2 were arbitrary, we conclude that $g \circ f$ is injective. $\square$

Proposition 1.4. *Let $f : X \to Y$ and $g : Y \to Z$ be surjective functions. Then $g \circ f$ is also surjective.*

Proof. Let $z \in Z$. As g is surjective, there exists $y \in Y$ such that $g(y) = z$. Now as f is surjective, there exists $x \in X$ such that $f(x) = y$. Thus, $(g \circ f)(x) = z$. As z was arbitrary, it follows that $g \circ f$ is surjective. $\square$

1.2.2 Equivalence Relations

Equivalence relations are of particular importance in mathematics and computer science. Intuitively, an equivalence relation compares which elements in a set X share some common property. The goal is to then partition X into equivalence classes such that all the elements in one of these parts are all equivalent to each other. This allows us to select an arbitrary distinct representative from each equivalence class and consider only that representative.

This idea of partitioning comes up quite frequently. The integers modulo n, denoted $\mathbb{Z}/n\mathbb{Z}$, is a canonical example. Big-Theta is another important equivalence relation. Equivalence relations allow us to prove powerful theorems such as Fermat's Little Theorem from Number Theory and Cauchy's Theorem from Group Theory, as well as to construct a procedure to minimize finite state automata via the Myhill-Nerode Theorem.

In order to guarantee such a partition, an equivalence relation must satisfy three properties: reflexivity, symmetry, and transitivity. We define these formally below, restricting attention to binary relations.

Definition 15 (Reflexive Relation)**.** A relation R on the set X is said to be reflexive if $(a, a) \in R$ for every $a \in X$.

Definition 16 (Symmetric Relation)**.** A relation R on the set X is said to be symmetric if $(a, b) \in R$ if and only if $(b, a) \in R$ for every $a, b \in X$.

Definition 17 (Transitive Relation)**.** A relation R on the set X is said to be transitive if for every $a, b, c \in X$ satisfying $(a, b), (b, c) \in R$, then $(a, c) \in R$.

Definition 18 (Equivalence Relation)**.** An equivalence relation is a reflexive, symmetric, and transitive relation.

DFA's, NFA's, Regular Languages

The family of regular languages is the simplest, yet interesting family of languages.

We give six definitions of the regular languages.

1. Using *deterministic finite automata (DFAs)*.
2. Using *nondeterministic finite automata (NFAs)*.
3. Using a *closure definition* involving, union, concatenation, and Kleene $*$.
4. Using *regular expressions*.
5. Using *right-invariant equivalence relations of finite index* (the Myhill-Nerode characterization).
6. Using *right-linear context-free grammars*.

We prove the equivalence of these definitions, often by providing an *algorithm* for converting one formulation into another.

We find that the introduction of NFA's is motivated by the conversion of regular expressions into DFA's.

To finish this conversion, we also show that every NFA can be converted into a DFA (using the *subset construction*).

So, although NFA's often allow for more concise descriptions, they do not have more expressive power than DFA's.

NFA's operate according to the paradigm: *guess a successful path, and check it in polynomial time*.

This is the essence of an important class of hard problems known as $\mathcal{NP}$, which will be investigated later.

We will also discuss methods for proving that certain languages are not regular (Myhill-Nerode, pumping lemma).

We present algorithms to convert a DFA to an equivalent one with a minimal number of states.

3.1 Deterministic Finite Automata (DFA's)

First we define what DFA's are, and then we explain how they are used to accept or reject strings. Roughly speaking, a DFA is a finite transition graph whose edges are labeled with letters from an alphabet Σ.

The graph also satisfies certain properties that make it deterministic. Basically, this means that given any string w, starting from any node, *there is a unique path in the graph "parsing" the string w*.

Example 1. A DFA for the language

$$L_1 = \{ab\}^+ = \{ab\}^*\{ab\},$$

i.e.,

$$L_1 = \{ab, abab, ababab, \ldots, (ab)^n, \ldots\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_1 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_1 = \{2\}$.

Transition table (function) δ_1:

	a	b
0	1	3
1	3	2
2	1	3
3	3	3

Note that state 3 is a *trap state* or *dead state*.

Here is a graph representation of the DFA specified by the transition function shown above:

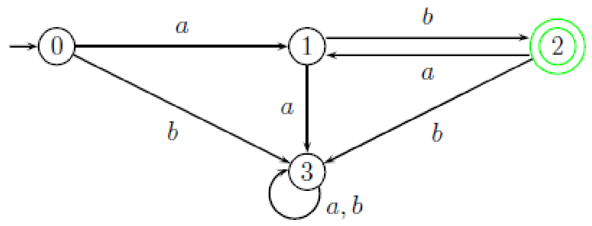

Figure 3.1: DFA for $\{ab\}^+$

Example 2. A DFA for the language

$$L_2 = \{ab\}^* = L_1 \cup \{\epsilon\}$$

i.e.,

$$L_2 = \{\epsilon, ab, abab, ababab, \dots, (ab)^n, \dots\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_2 = \{0, 1, 2\}$.

Start state: 0.

Set of accepting states: $F_2 = \{0\}$.

Transition table (function) δ_2:

	a	b
0	1	2
1	2	0
2	2	2

State 2 is a *trap state* or *dead state*.

Here is a graph representation of the DFA specified by the transition function shown above:

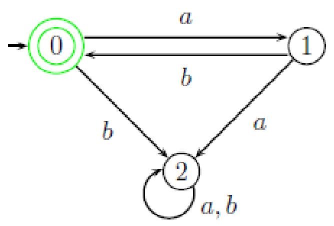

Figure 3.2: DFA for $\{ab\}^*$

Example 3. A DFA for the language

$$L_3 = \{a, b\}^*\{abb\}.$$

Note that L_3 consists of all strings of a's and b's ending in abb.

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_3 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_3 = \{3\}$.

Transition table (function) δ_3:

	a	b
0	1	0
1	1	2
2	1	3
3	1	0

Here is a graph representation of the DFA specified by the transition function shown above:

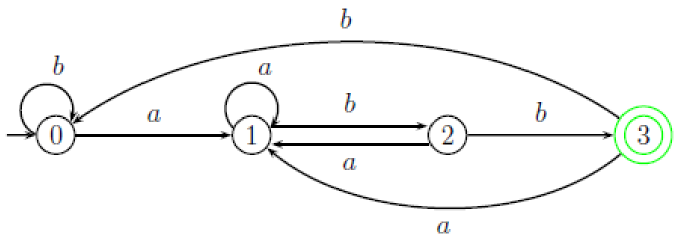

Figure 3.3: DFA for $\{a, b\}^*\{abb\}$

Is this a minimal DFA?

Definition 3.1. A *deterministic finite automaton (or DFA)* is a quintuple $D = (Q, \Sigma, \delta, q_0, F)$, where

- Σ is a finite *input alphabet*;
- Q is a finite set of *states*;

- F is a subset of Q of *final (or accepting) states*;
- $q_0 \in Q$ is the *start state (or initial state)*;
- δ is the *transition function*, a function

$$\delta \colon Q \times \Sigma \to Q.$$

For any state $p \in Q$ and any input $a \in \Sigma$, the state $q = \delta(p, a)$ is uniquely determined.

Thus, it is possible to define the state reached from a given state $p \in Q$ on input $w \in \Sigma^*$, following the path specified by w.

Technically, this is done by defining the extended transition function $\delta^* \colon Q \times \Sigma^* \to Q$.

Definition 3.2. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the *extended transition function* $\delta^* \colon Q \times \Sigma^* \to Q$ is defined as follows:

$$\begin{aligned}\delta^*(p, \epsilon) &= p,\\ \delta^*(p, ua) &= \delta(\delta^*(p, u), a),\end{aligned}$$

where $a \in \Sigma$ and $u \in \Sigma^*$.

It is immediate that $\delta^*(p, a) = \delta(p, a)$ for $a \in \Sigma$.

The meaning of $\delta^*(p, w)$ is that it is the state reached from state p following the path from p specified by w.

We can show (by induction on the length of v) that

$$\delta^*(p, uv) = \delta^*(\delta^*(p, u), v) \quad \text{for all } p \in Q \text{ and all } u, v \in \Sigma^*$$

For the induction step, for $u \in \Sigma^*$, and all $v = ya$ with $y \in \Sigma^*$ and $a \in \Sigma$,

$$\begin{aligned}\delta^*(p, uya) &= \delta(\delta^*(p, uy), a) && \text{by definition of } \delta^*\\ &= \delta(\delta^*(\delta^*(p, u), y), a) && \text{by induction}\\ &= \delta^*(\delta^*(p, u), ya) && \text{by definition of } \delta^*.\end{aligned}$$

We can now define how a DFA accepts or rejects a string.

Definition 3.3. Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the *language $L(D)$ accepted (or recognized) by D* is the language

$$L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

3.3 Nondeteterministic Finite Automata (NFA's)

NFA's are obtained from DFA's by allowing multiple transitions from a given state on a given input. This can be done by defining $\delta(p, a)$ as a **subset** of Q rather than a single state. It will also be convenient to allow transitions on input ϵ.

We let 2^Q denote the set of all subsets of Q, including the empty set. The set 2^Q is the *power set* of Q.

Example 4. A NFA for the language

$$L_3 = \{a, b\}^*\{abb\}.$$

Input alphabet: $\Sigma = \{a, b\}$.

State set $Q_4 = \{0, 1, 2, 3\}$.

Start state: 0.

Set of accepting states: $F_4 = \{3\}$.

Transition table δ_4:

	a	b
0	$\{0, 1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

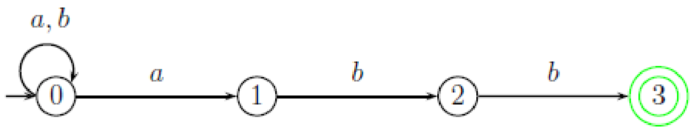

Figure 3.4: NFA for $\{a, b\}^*\{abb\}$

Example 5. Let $\Sigma = \{a_1, \ldots, a_n\}$, let

$$L_n^i = \{w \in \Sigma^* \mid w \text{ contains an odd number of } a_i\text{'s}\},$$

and let

$$L_n = L_n^1 \cup L_n^2 \cup \cdots \cup L_n^n.$$

The language L_n consists of those strings in Σ^* that contain an odd number of some letter $a_i \in \Sigma$.

Equivalently $\Sigma^* - L_n$ consists of those strings in Σ^* with an even number of *every* letter $a_i \in \Sigma$.

It can be shown that every DFA accepting L_n has at least 2^n states.

However, there is an NFA with $2n + 1$ states accepting L_n.

We define NFA's as follows.

Definition 3.5. A *nondeterministic finite automaton (or NFA)* is a quintuple $N = (Q, \Sigma, \delta, q_0, F)$, where

- Σ is a finite *input alphabet*;
- Q is a finite set of *states*;
- F is a subset of Q of *final (or accepting) states*;
- $q_0 \in Q$ is the *start state (or initial state)*;
- δ is the *transition function*, a function

$$\delta \colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q.$$

For any state $p \in Q$ and any input $a \in \Sigma \cup \{\epsilon\}$, the set of states $\delta(p, a)$ is uniquely determined. We write $q \in \delta(p, a)$.

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, we would like to define the language accepted by N.

However, given an NFA N, unlike the situation for DFA's, given a state $p \in Q$ and some input $w \in \Sigma^*$, in general *there is no unique path from p on input w, but instead a tree of computation paths*.

For example, given the NFA shown below,

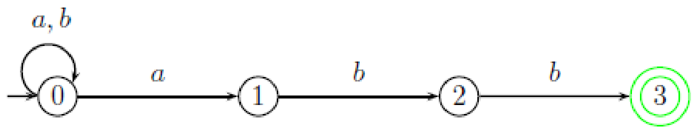

Figure 3.5: NFA for $\{a, b\}^*\{abb\}$

from state 0 on input $w = ababb$ we obtain the following tree of computation paths:

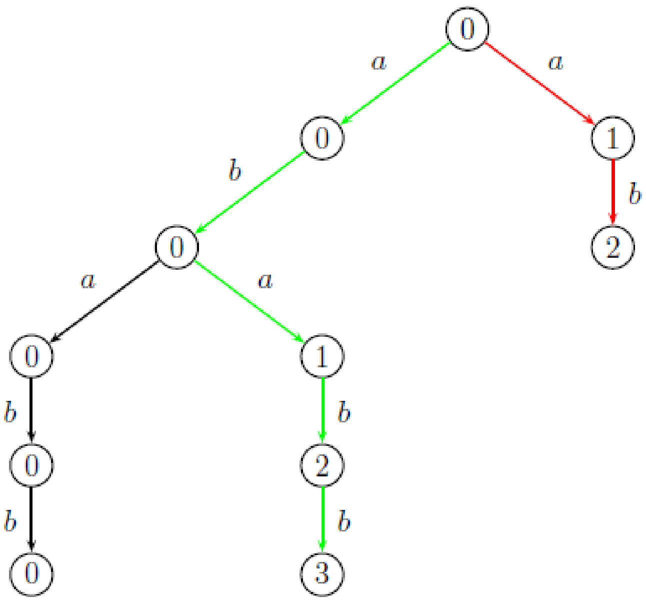

Figure 3.6: A tree of computation paths on input $ababb$

Observe that there are three kinds of computation paths:

1. A path on input w ending in a rejecting state (for example, the lefmost path).
2. A path on some proper prefix of w, along which the computation gets stuck (for example, the rightmost path).
3. A path on input w ending in an accepting state (such as the path ending in state 3).

The acceptance criterion for NFA is *very lenient*: a string w is accepted iff the tree of computation paths contains *some accepting path* (of type (3)).

Thus, all failed paths of type (1) and (2) are ignored. Furthermore, there is *no charge* for failed paths.

A string w is rejected iff all computation paths are failed paths of type (1) or (2).

The "philosophy" of nondeterminism is that an NFA
"guesses" an accepting path and then checks it in polynomial time by following this path. We are only charged for one accepting path (even if there are several accepting paths).

A way to capture this acceptance policy if to extend the transition function $\delta\colon Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ to a function

$$\delta^*\colon Q \times \Sigma^* \to 2^Q.$$

The presence of ϵ-transitions (i.e., when $q \in \delta(p, \epsilon)$) causes technical problems, and to overcome these problems, we introduce the notion of ϵ-closure.

3.4 ϵ-Closure

Definition 3.6. Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ (with ϵ-transitions) for every state $p \in Q$, the *ϵ-closure of p* is set $\epsilon\text{-closure}(p)$ consisting of all states q such that there is a path from p to q whose spelling is ϵ (an *ϵ-path*).

This means that either $q = p$, or that all the edges on the path from p to q have the label ϵ.

We can compute $\epsilon\text{-closure}(p)$ using a sequence of approximations as follows. Define the sequence of sets of states $(\epsilon\text{-clo}_i(p))_{i\geq 0}$ as follows:

$$\begin{aligned}
\epsilon\text{-clo}_0(p) &= \{p\},\\
\epsilon\text{-clo}_{i+1}(p) &= \epsilon\text{-clo}_i(p) \cup\\
&\qquad \{q \in Q \mid \exists s \in \epsilon\text{-clo}_i(p),\ q \in \delta(s, \epsilon)\}.
\end{aligned}$$

Since $\epsilon\text{-clo}_i(p) \subseteq \epsilon\text{-clo}_{i+1}(p)$, $\epsilon\text{-clo}_i(p) \subseteq Q$, for all $i \geq 0$, and Q is finite, it can be shown that there is a smallest i, say i_0, such that

$$\epsilon\text{-clo}_{i_0}(p) = \epsilon\text{-clo}_{i_0+1}(p).$$

It suffices to show that there is some $i \geq 0$ such that $\epsilon\text{-clo}_i(p) = \epsilon\text{-clo}_{i+1}(p)$, because then there is a smallest such i (since every nonempty subset of $\mathbb{N}$ has a smallest element).

Assume by contradiction that

$$\epsilon\text{-clo}_i(p) \subset \epsilon\text{-clo}_{i+1}(p) \quad \text{for all } i \geq 0.$$

Then, I claim that $|\epsilon\text{-clo}_i(p)| \geq i + 1$ for all $i \geq 0$.

This is true for $i = 0$ since $\epsilon\text{-clo}_0(p) = \{p\}$.

Since $\epsilon\text{-clo}_i(p) \subset \epsilon\text{-clo}_{i+1}(p)$, there is some $q \in \epsilon\text{-clo}_{i+1}(p)$ that does not belong to $\epsilon\text{-clo}_i(p)$, and since by induction $|\epsilon\text{-clo}_i(p)| \geq i + 1$, we get

$$|\epsilon\text{-clo}_{i+1}(p)| \geq |\epsilon\text{-clo}_i(p)| + 1 \geq i + 1 + 1 = i + 2,$$

establishing the induction hypothesis.

If $n = |Q|$, then $|\epsilon\text{-clo}_n(p)| \geq n + 1$, a contradiction.

Therefore, there is indeed some $i \geq 0$ such that $\epsilon\text{-clo}_i(p) = \epsilon\text{-clo}_{i+1}(p)$, and for the least such $i = i_0$, we have $i_0 \leq n - 1$.

It can also be shown that

$$\epsilon\text{-closure}(p) = \epsilon\text{-clo}_{i_0}(p),$$

by proving that

1. $\epsilon\text{-clo}_i(p) \subseteq \epsilon\text{-closure}(p)$, for all $i \geq 0$.
2. $\epsilon\text{-closure}(p)_i \subseteq \epsilon\text{-clo}_{i_0}(p)$, for all $i \geq 0$.

where $\epsilon\text{-closure}(p)_i$ is the set of states reachable from p by an ϵ-path of length $\leq i$.

When N has no ϵ-transitions, i.e., when $\delta(p, \epsilon) = \emptyset$ for all $p \in Q$ (which means that δ can be viewed as a function $\delta \colon Q \times \Sigma \to 2^Q$), we have

$$\epsilon\text{-closure}(p) = \{p\}.$$

It should be noted that there are more efficient ways of computing ϵ-closure(p), for example, using a stack (basically, a kind of depth-first search).

We present such an algorithm below. It is assumed that the types *NFA* and *stack* are defined. If n is the number of states of an NFA N, we let

eclotype = **array**$[1..n]$ **of boolean**

```
function eclosure[N: NFA, p: integer]: eclotype;
    begin
      var eclo: eclotype, q, s: integer, st: stack;
      for each $q \in setstates(N)$ do
        $eclo[q] := false$;
      endfor
      $eclo[p] := true$; $st := empty$;
      $trans := deltatable(N)$;
      $st := push(st, p)$;
      while $st \neq emptystack$ do
        $q = pop(st)$;
        for each $s \in trans(q, \epsilon)$ do
          if $eclo[s] = false$ then
            $eclo[s] := true$; $st := push(st, s)$
```

3.5 Converting an NFA into a DFA

The intuition behind the definition of the extended transition function is that $\delta^*(p, w)$ is the set of all states reachable from p by a path whose spelling is w.

Definition 3.7. Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ (with ϵ-transitions), the *extended transition function* $\delta^*\colon Q \times \Sigma^* \to 2^Q$ is defined as follows: for every $p \in Q$, every $u \in \Sigma^*$, and every $a \in \Sigma$,

$$\delta^*(p, \epsilon) = \epsilon\text{-closure}(\{p\}),$$
$$\delta^*(p, ua) = \epsilon\text{-closure}\Bigg(\bigcup_{s \in \delta^*(p,u)} \delta(s, a)\Bigg).$$

In the second equation, if $\delta^*(p, u) = \emptyset$ then

$$\delta^*(p, ua) = \emptyset.$$

The *language $L(N)$ accepted by an NFA N* is the set

$$L(N) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}.$$

Observe that the definition of $L(N)$ conforms to the lenient acceptance policy: a string w is accepted iff $\delta^*(q_0, w)$ contains *some final state*.

We can also extend $\delta^* \colon Q \times \Sigma^* \to 2^Q$ to a function

$$\widehat{\delta} \colon 2^Q \times \Sigma^* \to 2^Q$$

defined as follows: for every subset S of Q, for every $w \in \Sigma^*$,

$$\widehat{\delta}(S, w) = \bigcup_{s \in S} \delta^*(s, w),$$

with

$$\widehat{\delta}(\emptyset, w) = \emptyset.$$

Let $\mathcal{Q}$ be the subset of 2^Q consisting of those subsets S of Q that are ϵ-closed, i.e., such that

$$S = \epsilon\text{-closure}(S).$$

If we consider the restriction

$$\Delta \colon \mathcal{Q} \times \Sigma \to \mathcal{Q}$$

of $\widehat{\delta} \colon 2^Q \times \Sigma^* \to 2^Q$ to $\mathcal{Q}$ and Σ, we observe that Δ is the transition function of a DFA.

Indeed, this is the transition function of a DFA accepting $L(N)$. It is easy to show that Δ is defined directly as follows (on subsets S in $\mathcal{Q}$):

$$\Delta(S, a) = \epsilon\text{-closure}\left(\bigcup_{s \in S} \delta(s, a)\right),$$

with

$$\Delta(\emptyset, a) = \emptyset.$$

Then, the DFA D is defined as follows:

$$D = (\mathcal{Q}, \Sigma, \Delta, \epsilon\text{-closure}(\{q_0\}), \mathcal{F}),$$

where $\mathcal{F} = \{S \in \mathcal{Q} \mid S \cap F \neq \emptyset\}$.

It is not difficult to show that $L(D) = L(N)$, that is, D is a DFA accepting $L(N)$. For this, we show that

$$\Delta^*(S, w) = \widehat{\delta}(S, w).$$

Thus, we have converted the NFA N into a DFA D (and gotten rid of ϵ-transitions).

Since DFA's are special NFA's, the subset construction shows that DFA's and NFA's accept *the same* family of languages, the *regular languages, version 1* (although not with the same complexity).

The states of the DFA D equivalent to N are ϵ-closed subsets of Q. For this reason, the above construction is often called the *subset construction*.

This construction is due to Rabin and Scott.

Although theoretically fine, the method may construct useless sets S that are not reachable from the start state ϵ-closure($\{q_0\}$). A more economical construction is given next.

An Algorithm to convert an NFA into a DFA:
The "subset construction"

Given an input NFA $N = (Q, \Sigma, \delta, q_0, F)$, a DFA $D = (K, \Sigma, \Delta, S_0, \mathcal{F})$ is constructed. It is assumed that K is a linear array of sets of states $S \subseteq Q$, and Δ is a 2-dimensional array, where $\Delta[i, a]$ is the index of the target state of the transition from $K[i] = S$ on input a, with $S \in K$, and $a \in \Sigma$.

```
$S_0 := \epsilon\text{-closure}(\{q_0\})$; $total := 1$; $K[1] := S_0$;
$marked := 0$;
while $marked < total$ do;
  $marked := marked + 1$; $S := K[marked]$;
  for each $a \in \Sigma$ do
    $U := \bigcup_{s \in S} \delta(s, a)$; $T := \epsilon\text{-closure}(U)$;
    if $T \notin K$ then
      $total := total + 1$; $K[total] := T$
    endif;
    $\Delta[marked, a] := \text{index}(T)$
  endfor
endwhile;
$\mathcal{F} := \{S \in K \mid S \cap F \neq \emptyset\}$
```

Let us illustrate the subset construction on the NFA of Example 4.

A NFA for the language

$$L_3 = \{a, b\}^*\{abb\}.$$

Transition table δ_4:

	a	b
0	$\{0,1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

Set of accepting states: $F_4 = \{3\}$.

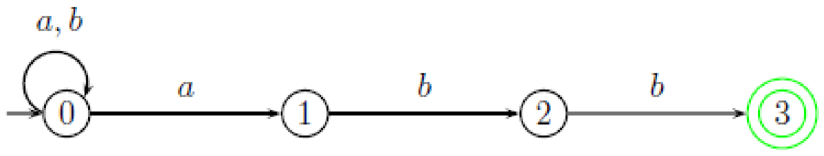

Figure 3.7: NFA for $\{a, b\}^*\{abb\}$

The pointer $\Rightarrow$ corresponds to *marked* and the pointer $\rightarrow$ to *total*.

Initial transition table Δ.

$\Rightarrow$	index	states	a	b
$\rightarrow$	A	$\{0\}$		

Just after entering the while loop

	index	states	a	b
$\Rightarrow\rightarrow$	A	$\{0\}$		

After the first round through the while loop.

	index	states	a	b
$\Rightarrow$	A	$\{0\}$	B	A
$\rightarrow$	B	$\{0,1\}$		

After just reentering the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
$\Rightarrow\rightarrow$	B	$\{0,1\}$		

After the second round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
$\Rightarrow$	B	$\{0,1\}$	B	C
$\rightarrow$	C	$\{0,2\}$		

After the third round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
	B	$\{0,1\}$	B	C
$\Rightarrow$	C	$\{0,2\}$	B	D
$\rightarrow$	D	$\{0,3\}$		

After the fourth round through the while loop.

	index	states	a	b
	A	$\{0\}$	B	A
	B	$\{0,1\}$	B	C
	C	$\{0,2\}$	B	D
$\Rightarrow\rightarrow$	D	$\{0,3\}$	B	A

This is the DFA of Figure 3.3, except that in that example A, B, C, D are renamed $0, 1, 2, 3$.

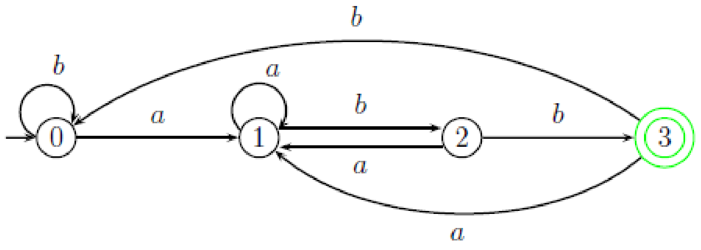

Figure 3.8: DFA for $\{a,b\}^*\{abb\}$

3.6 Finite State Automata With Output: Transducers

So far, we have only considered automata that recognize languages, i.e., automata that do not produce any output on any input (except "accept" or "reject").

It is interesting and useful to consider input/output finite state machines. Such automata are called *transducers*. They compute functions or relations. First, we define a deterministic kind of transducer.

Definition 3.8. A *general sequential machine (gsm)* is a sextuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

(1) Q is a finite set of *states*,

(2) Σ is a finite *input alphabet*,

(3) Δ is a finite *output alphabet*,

(4) $\delta \colon Q \times \Sigma \to Q$ is the *transition function*,

(5) $\lambda \colon Q \times \Sigma \to \Delta^*$ is the *output function* and

(6) q_0 is the *initial* (or *start*) *state*.

If $\lambda(p, a) \neq \epsilon$, for all $p \in Q$ and all $a \in \Sigma$, then M is *nonerasing*. If $\lambda(p, a) \in \Delta$ for all $p \in Q$ and all $a \in \Sigma$, we say that M is a *complete sequential machine (csm)*.

An example of a gsm for which $\Sigma = \{a, b\}$ and $\Delta = \{0, 1, 2\}$ is shown in Figure 3.9. For example aab is converted to 102001.

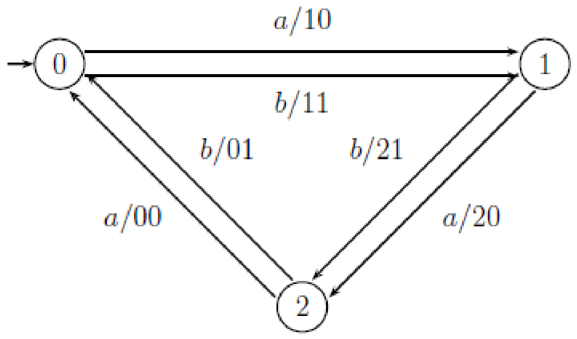

Figure 3.9: Example of a gsm

In order to define how a gsm works, we extend the transition and the output functions. We define $\delta^* \colon Q \times \Sigma^* \to Q$ and $\lambda^* \colon Q \times \Sigma^* \to \Delta^*$ recursively as follows: For all $p \in Q$, all $u \in \Sigma^*$ and all $a \in \Sigma$

$$\begin{aligned}
\delta^*(p, \epsilon) &= p \\
\delta^*(p, ua) &= \delta(\delta^*(p, u), a) \\
\lambda^*(p, \epsilon) &= \epsilon \\
\lambda^*(p, ua) &= \lambda^*(p, u)\lambda(\delta^*(p, u), a).
\end{aligned}$$

For any $w \in \Sigma^*$, we let

$$M(w) = \lambda^*(q_0, w)$$

and for any $L \subseteq \Sigma^*$ and $L' \subseteq \Delta^*$, let

$$M(L) = \{\lambda^*(q_0, w) \mid w \in L\}$$

3.7 An Application of NFA's: Text Search

A common problem in the age of the Web (and on-line text repositories) is the following:

Given a set of words, called the *keywords*, find all the documents that contain one (or all) of those words.

Search engines are a popular example of this process. Search engines use *inverted indexes* (for each word appearing on the Web, a list of all the places where that word occurs is stored).

However, there are applications that are unsuited for inverted indexes, but are good for automaton-based techniques.

Some text-processing programs, such as advanced forms of the UNIX `grep` command (such as `egrep` or `fgrep`) are based on automaton-based techniques.

The characteristics that make an application suitable for searches that use automata are:

(1) The repository on which the search is conducted is rapidly changing.

(2) The documents to be searched cannot be catalogued. For example, Amazon.com creates pages "on the fly" in response to queries.

We can use an NFA to find occurrences of a set of keywords in a text. This NFA signals by entering a final state that it has seen one of the keywords. The form of such an NFA is special.

(1) There is a start state, q_0, with a transition to itself on every input symbol from the alphabet, Σ.

(2) For each keyword, $w = w_1 \cdots w_k$ (with $w_i \in \Sigma$), there are k states, $q_1^{(w)}, \ldots, q_k^{(w)}$, and there is a transition from q_0 to $q_1^{(w)}$ on input w_1, a transition from $q_1^{(w)}$ to $q_2^{(w)}$ on input w_2, and so on, until a transition from $q_{k-1}^{(w)}$ to $q_k^{(w)}$ on input w_k. The state $q_k^{(w)}$ is an accepting state and indicates that the keyword $w = w_1 \cdots w_k$ has been found.

The NFA constructed above can then be converted to a DFA using the subset construction.

Here is an example where $\Sigma = \{a, b\}$ and the set of keywords is

$$\{aba,\ ab,\ ba\}.$$

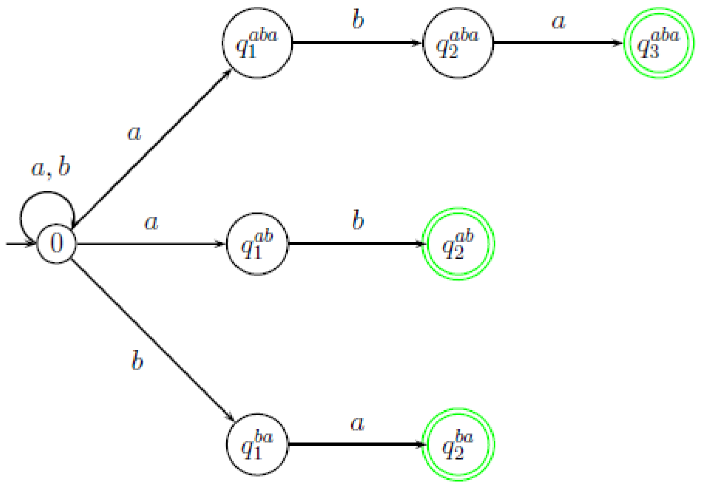

Figure 3.10: NFA for the keywords aba, ab, ba.

Applying the subset construction to the NFA, we obtain the DFA whose transition table is:

		a	b
0	0	1	2
1	$0, q_1^{aba}, q_1^{ab}$	1	3
2	$0, q_1^{ba}$	4	2
3	$0, q_1^{ba}, q_2^{aba}, q_2^{ab}$	5	2
4	$0, q_1^{aba}, q_1^{ab}, q_2^{ba}$	1	3
5	$0, q_1^{aba}, q_1^{ab}, q_2^{ba}, q_3^{aba}$	1	3

The final states are: 3, 4, 5.

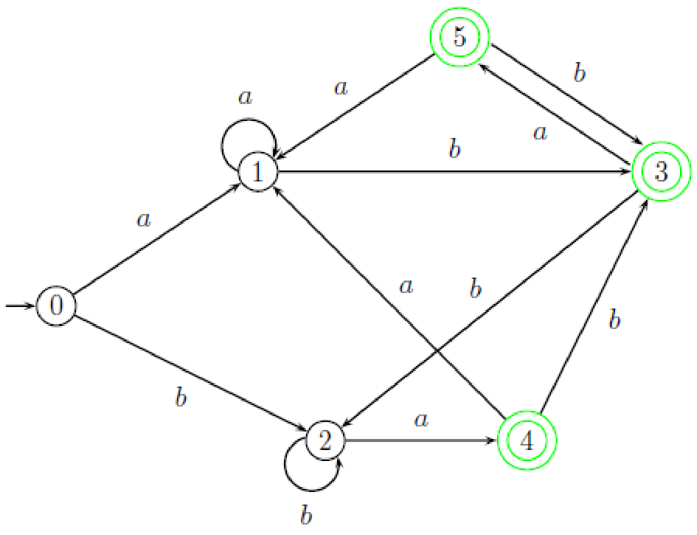

Figure 3.11: DFA for the keywords aba, ab, ba.

The good news news is that, due to the very special structure of the NFA, the number of states of the corresponding DFA is *at most* the number of states of the original NFA!

We find that the states of the DFA are (check it yourself!):

(1) The set $\{q_0\}$, associated with the start state q_0 of the NFA.

(2) For any state $p \neq q_0$ of the NFA reached from q_0 along a path corresponding to a string $u = u_1 \cdots u_m$, the set consisting of:

5.10 Finding minimal DFA's

Given any language L (not necessarily regular), we can define an equivalence relation ρ_L on Σ^* which is right-invariant, but not necessarily of finite index. The equivalence relation ρ_L is such that L is the union of equivalence classes of ρ_L. Furthermore, when L is regular, the relation ρ_L has finite index. In fact, this index is the size of a smallest DFA accepting L. As a consequence, if L is regular, a simple modification of the proof of Proposition 5.9 applied to $\simeq\, = \rho_L$ yields a minimal DFA D_{ρ_L} accepting L.

Then, given any trim DFA D accepting L, the equivalence relation ρ_L can be translated to an equivalence relation $\equiv$ on states, in such a way that for all $u, v \in \Sigma^*$,

$$u \rho_L v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v),$$

where $\varphi\colon \Sigma^* \to Q$ is the function (run the DFA D on u from q_0) given by

$$\varphi(u) = \delta^*(q_0, u).$$

One can then construct a quotient DFA $D/\equiv$ whose states are obtained by merging all states in a given equivalence class of states into a single state, and the resulting DFA $D/\equiv$ is a mininal DFA. Even though $D/\equiv$ appears to depend on D, it is in fact unique, and isomorphic to the abstract DFA D_{ρ_L} induced by ρ_L.

The last step in obtaining the minimal DFA $D/\equiv$ is to give a constructive method to compute the state equivalence relation $\equiv$. This can be done by constructing a sequence of approximations $\equiv_i$, where each $\equiv_{i+1}$ refines $\equiv_i$. It turns out that if D has n states, then there is some index $i_0 \leq n-2$ such that

$$\equiv_j \,=\, \equiv_{i_0} \quad \text{for all } j \geq i_0 + 1,$$

and that

$$\equiv \,=\, \equiv_{i_0}.$$

Furthermore, $\equiv_{i+1}$ can be computed inductively from $\equiv_i$. In summary, we obtain a iterative algorithm for computing $\equiv$ that terminates in at most $n-2$ steps.

Definition 5.10. Given any language L (over Σ), we define the *right-invariant equivalence* ρ_L *associated with* L as the relation on Σ^* defined as follows: for any two strings $u, v \in \Sigma^*$,

$$u \rho_L v \quad \text{iff} \quad \forall w \in \Sigma^*(uw \in L \quad \text{iff} \quad vw \in L).$$

It is clear that the relation ρ_L is an equivalence relation, and it is right-invariant. To show right-invariance, argue as follows: if $u\rho_L v$, then for any $w \in \Sigma^*$, since $u\rho_L v$ means that

$$uz \in L \quad \text{iff} \quad vz \in L$$

for all $z \in \Sigma^*$, in particular the above equivalence holds for all z of the form $z = wy$ for any arbitary $y \in \Sigma^*$, so we have

$$uwy \in L \quad \text{iff} \quad vwy \in L$$

for all $y \in \Sigma^*$, which means that $uw\rho_L vw$.

It is also clear that L is the union of the equivalence classes of strings in L. This is because if $u \in L$ and $u\rho_L v$, by letting $w = \epsilon$ in the definition of ρ_L, we get

$$u \in L \quad \text{iff} \quad v \in L,$$

and since $u \in L$, we also have $v \in L$. This implies that if $u \in L$ then $[u]_{\rho_L} \subseteq L$ and so,

$$L = \bigcup_{u \in L} [u]_{\rho_L}.$$

Example 5.9. For example, consider the regular language

$$L = \{a\} \cup \{b^m \mid m \geq 1\}.$$

We leave it as an exercise to show that the equivalence relation ρ_L consists of the four equivalence classes

$$C_1 = \{\epsilon\},\ C_2 = \{a\},\ C_3 = \{b\}^+,\ C_4 = a\{a,b\}^+ \cup \{b\}^+a\{a,b\}^*$$

encountered earlier in Example 5.6. Observe that

$$L = C_2 \cup C_3.$$

When L is regular, we have the following remarkable result:

Proposition 5.13. *Given any regular language L, for any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(D)$, ρ_L is a right-invariant equivalence relation, and we have $\simeq_D \subseteq\ \rho_L$. Furthermore, if ρ_L has m classes and Q has n states, then $m \leq n$.*

Proof. By definition, $u \simeq_D v$ iff $\delta^*(q_0, u) = \delta^*(q_0, v)$. Since $w \in L(D)$ iff $\delta^*(q_0, w) \in F$, the fact that $u\rho_L v$ can be expressed as

$$\begin{aligned}
&\forall w \in \Sigma^*(uw \in L \quad \text{iff} \quad vw \in L)\\
&\text{iff}\\
&\forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \quad \text{iff} \quad \delta^*(q_0, vw) \in F)\\
&\text{iff}\\
&\forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F),
\end{aligned}$$

and if $\delta^*(q_0, u) = \delta^*(q_0, v)$, this shows that $u\rho_L v$. Since the number of classes of $\simeq_D$ is n and $\simeq_D \subseteq\ \rho_L$, the equivalence relation ρ_L has fewer classes than $\simeq_D$, and $m \leq n$. ∎

Proposition 5.13 shows that when L is regular, the index m of ρ_L is finite, and it is a lower bound on the size of all DFA's accepting L. It remains to show that a DFA with m states accepting L exists.

However, going back to the proof of Proposition 5.9 starting with the right-invariant equivalence relation ρ_L of finite index m, if L is the union of the classes $C_{i_1}, \ldots, C_{i_k}$, the DFA

$$D_{\rho_L} = (\{1, \ldots, m\}, \Sigma, \delta, 1, \{i_1, \ldots, i_k\}),$$

where $\delta(i, a) = j$ iff $C_i a \subseteq C_j$, is such that $L = L(D_{\rho_L})$.

In summary, if L is regular, then the index of ρ_L is equal to the number of states of a minimal DFA for L, and D_{ρ_L} is a minimal DFA accepting L.

Example 5.10. For example, if

$$L = \{a\} \cup \{b^m \mid m \geq 1\}.$$

then we saw in Example 5.9 that ρ_L consists of the four equivalence classes

$$C_1 = \{\epsilon\},\ C_2 = \{a\},\ C_3 = \{b\}^+,\ C_4 = a\{a, b\}^+ \cup \{b\}^+ a\{a, b\}^*,$$

and we showed in Example 5.6 that the transition table of D_{ρ_L} is given by

	a	b
C_1	C_2	C_3
C_2	C_4	C_4
C_3	C_4	C_3
C_4	C_4	C_4

By picking the final states to be C_2 and C_3, we obtain the minimal DFA D_{ρ_L} accepting $L = \{a\} \cup \{b^m \mid m \geq 1\}$.

In the next section, we give an algorithm which allows us to find D_{ρ_L}, given any DFA D accepting L. This algorithms finds which states of D are equivalent.

5.11 State Equivalence and Minimal DFA's

The proof of Proposition 5.13 suggests the following definition of an equivalence between states:

Definition 5.11. Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\equiv$ on Q, called *state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^*(\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F). \qquad (*)$$

When $p \equiv q$, we say that *p and q are indistinguishable*.

It is trivial to verify that $\equiv$ is an equivalence relation, and that it satisfies the following property:

$$\text{if } p \equiv q \text{ then } \delta(p,a) \equiv \delta(q,a), \quad \text{for all } a \in \Sigma.$$

To prove the above, since the condition defining $\equiv$ must hold for all strings $w \in \Sigma^*$, in particular it must hold for all strings of the form $w = au$ with $a \in \Sigma$ and $u \in \Sigma^*$, so if $p \equiv q$ then we have

$$\begin{aligned}
& (\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(p,au) \in F \quad \text{iff} \quad \delta^*(q,au) \in F) \\
\text{iff} \quad & (\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(\delta^*(p,a),u) \in F \quad \text{iff} \quad \delta^*(\delta^*(q,a),u) \in F) \\
\text{iff} \quad & (\forall a \in \Sigma)(\forall u \in \Sigma^*)(\delta^*(\delta(p,a),u) \in F \quad \text{iff} \quad \delta^*(\delta(q,a),u) \in F) \\
\text{iff} \quad & (\forall a \in \Sigma)(\delta(p,a) \equiv \delta(q,a)).
\end{aligned}$$

$$\delta^*(p,\epsilon) \in F \quad \text{iff} \quad \delta^*(q,\epsilon) \in F,$$

which, since $\delta^*(p,\epsilon) = p$ and $\delta^*(q,\epsilon) = q$, is equivalent to

$$p \in F \quad \text{iff} \quad q \in F.$$

Therefore, if two states p, q are equivalent, then either both $p, q \in F$ or both $p, q \in \overline{F}$. This implies that a final state and a rejecting states are *never* equivalent.

Example 5.11. The reader should check that states A and C in the DFA below are equivalent and that no other distinct states are equivalent.

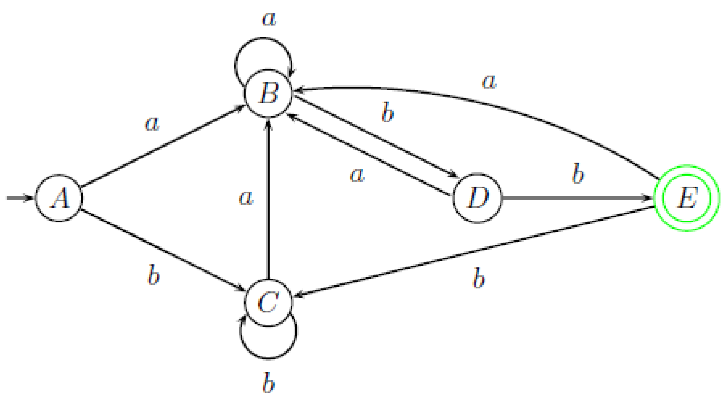

Figure 5.16: A non-minimal DFA for $\{a,b\}^*\{abb\}$

It is illuminating to express state equivalence as the equality of two languages. Given the DFA $D = (Q, \Sigma, \delta, q_0, F)$, let $D_p = (Q, \Sigma, \delta, p, F)$ be the DFA obtained from D by redefining the start state to be p. Then, it is clear that

$$p \equiv q \quad \text{iff} \quad L(D_p) = L(D_q).$$

This simple observation implies that there is an algorithm to test state equivalence. Indeed, we simply have to test whether the DFA's D_p and D_q accept the same language and this can be done using the cross-product construction. Indeed, $L(D_p) = L(D_q)$ iff $L(D_p) - L(D_q) = \emptyset$ and $L(D_q) - L(D_p) = \emptyset$. Now, if $(D_p \times D_q)_{1-2}$ denotes the cross-product DFA with starting state (p, q) and with final states $F \times (Q - F)$ and $(D_p \times D_q)_{2-1}$ denotes the cross-product DFA also with starting state (p, q) and with final states $(Q - F) \times F$, we know that

$$L((D_p \times D_q)_{1-2}) = L(D_p) - L(D_q) \quad \text{and} \quad L((D_p \times D_q)_{2-1}) = L(D_q) - L(D_p),$$

so all we need to do if to test whether $(D_p \times D_q)_{1-2}$ and $(D_p \times D_q)_{2-1}$ accept the empty language. However, we know that this is the case iff the set of states reachable from (p, q) in $(D_p \times D_q)_{1-2}$ contains no state in $F \times (Q - F)$ and the set of states reachable from (p, q) in $(D_p \times D_q)_{2-1}$ contains no state in $(Q - F) \times F$.

Actually, the graphs of $(D_p \times D_q)_{1-2}$ and $(D_p \times D_q)_{2-1}$ are identical, so we only need to check that no state in $(F \times (Q - F)) \cup ((Q - F) \times F)$ is reachable from (p, q) in that graph. This algorithm to test state equivalence is not the most efficient but it is quite reasonable (it runs in polynomial time).

If $L = L(D)$, Theorem 5.14 below shows the relationship between ρ_L and $\equiv$ and, more generally, between the DFA, D_{ρ_L}, and the DFA, $D/\equiv$, obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q.

The minimal DFA $D/\equiv$ is obtained by merging the states in each block C_i of the partition Π associated with $\equiv$, forming states corresponding to the blocks C_i, and drawing a transition on input a from a block C_i to a block C_j of Π iff there is a transition $q = \delta(p, a)$ from any state $p \in C_i$ to any state $q \in C_j$ on input a.

The start state is the block containing q_0, and the final states are the blocks consisting of final states.

Example 5.12. For example, consider the DFA D_1 accepting $L = \{ab, ba\}^*$ shown in Figure 5.17.

This is not a minimal DFA. In fact,

$$0 \equiv 2 \quad \text{and} \quad 3 \equiv 5.$$

Here is the minimal DFA for L:

The minimal DFA D_2 is obtained by merging the states in the equivalence class $\{0, 2\}$ into a single state, similarly merging the states in the equivalence class $\{3, 5\}$ into a single state, and drawing the transitions between equivalence classes. We obtain the DFA shown in Figure 5.18.

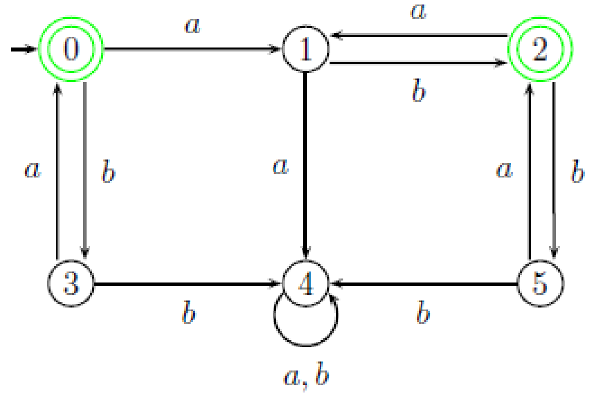

Figure 5.17: A nonminimal DFA D_1 for $L = \{ab, ba\}^*$

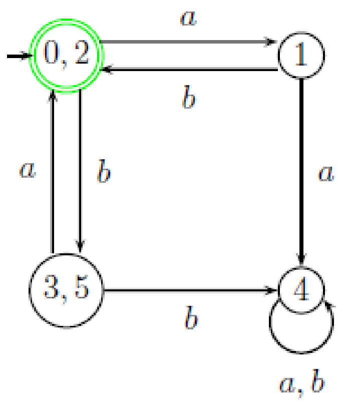

Figure 5.18: A minimal DFA D_2 for $L = \{ab, ba\}^*$

Formally, the quotient DFA $D/\equiv$ is defined such that

$$D/\equiv \; = (Q/\equiv, \Sigma, \delta/\equiv, [q_0]_{\equiv}, F/\equiv),$$

where

$$\delta/\equiv ([p]_{\equiv}, a) = [\delta(p, a)]_{\equiv}.$$

heorem 5.14. *For any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$ accepting the regular langua*
$= L(D)$*, the function $\varphi \colon \Sigma^* \to Q$ defined such that*

$$\varphi(u) = \delta^*(q_0, u)$$

atisfies the property

$$u \rho_L v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v) \quad \text{for all } u, v \in \Sigma^*,$$

nd induces a bijection $\widehat{\varphi} \colon \Sigma^*/\rho_L \to Q/\equiv$, defined such that

$$\widehat{\varphi}([u]_{\rho_L}) = [\delta^*(q_0, u)]_{\equiv}.$$

Furthermore, we have

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L} \quad \textit{iff} \quad \delta(\varphi(u), a) \equiv \varphi(v).$$

Consequently, $\widehat{\varphi}$ induces an isomorphism of DFA's, $\widehat{\varphi}\colon D_{\rho_L} \to D/\equiv$.

Proof. Since $\varphi(u) = \delta^*(q_0, u)$ and $\varphi(v) = \delta^*(q_0, v)$, the fact that $\varphi(u) \equiv \varphi(v)$ can be expressed as

$$\begin{aligned}
&\forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F)\\
&\text{iff}\\
&\forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \quad \text{iff} \quad \delta^*(q_0, vw) \in F),
\end{aligned}$$

which is exactly $u \, \rho_L \, v$. Therefore,

$$u \, \rho_L \, v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v).$$

From the above, we see that the equivalence class $[\varphi(u)]_\equiv$ of $\varphi(u)$ does not depend on the choice of the representative in the equivalence class $[u]_{\rho_L}$ of $u \in \Sigma^*$, since for any $v \in \Sigma^*$, if $u \, \rho_L \, v$ then $\varphi(u) \equiv \varphi(v)$, so $[\varphi(u)]_\equiv = [\varphi(v)]_\equiv$. Therefore, the function $\varphi\colon \Sigma^* \to Q$ maps each equivalence class $[u]_{\rho_L}$ modulo ρ_L to the equivalence class $[\varphi(u)]_\equiv$ modulo $\equiv$, and so the function $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$ given by

$$\widehat{\varphi}([u]_{\rho_L}) = [\varphi(u)]_\equiv = [\delta^*(q_0, u)]_\equiv$$

is well-defined. Moreover, $\widehat{\varphi}$ is injective, since $\widehat{\varphi}([u]) = \widehat{\varphi}([v])$ iff $\varphi(u) \equiv \varphi(v)$ iff (from above) $u\rho_v v$ iff $[u] = [v]$. Since every state in Q is accessible, for every $q \in Q$, there is some $u \in \Sigma^*$ so that $\varphi(u) = \delta^*(q_0, u) = q$, so $\widehat{\varphi}([u]) = [q]_\equiv$ and $\widehat{\varphi}$ is surjective. Therefore, we have a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$.

Since $\varphi(u) = \delta^*(q_0, u)$, we have

$$\delta(\varphi(u), a) = \delta(\delta^*(q_0, u), a) = \delta^*(q_0, ua) = \varphi(ua),$$

and thus, $\delta(\varphi(u), a) \equiv \varphi(v)$ can be expressed as $\varphi(ua) \equiv \varphi(v)$. By the previous part, this is equivalent to $ua\rho_L v$, and we claim that this is equivalent to

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

First, if $[u]_{\rho_L} a \subseteq [v]_{\rho_L}$, then $ua \in [v]_{\rho_L}$, that is, $ua\rho_L v$. Conversely, if $ua\rho_L v$, then for every $u' \in [u]_{\rho_L}$, we have $u'\rho_L u$, so by right-invariance we get $u'a\rho_L ua$, and since $ua\rho_L v$, we get $u'a\rho_L v$, that is, $u'a \in [v]_{\rho_L}$. Since $u' \in [u]_{\rho_L}$ is arbitrary, we conclude that $[u]_{\rho_L} a \subseteq [v]_{\rho_L}$. Therefore, we proved that

$$\delta(\varphi(u), a) \equiv \varphi(v) \quad \text{iff} \quad [u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

The above shows that the transitions of D_{ρ_L} correspond to the transitions of $D/\equiv$. □

Theorem 5.14 shows that the DFA D_{ρ_L} is isomorphic to the DFA $D/\equiv$ obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q. Since D_{ρ_L} is a minimal DFA accepting L, so is $D/\equiv$.

Example 5.13. Consider the following DFA D,

	a	b
1	2	3
2	4	4
3	4	3
4	5	5
5	5	5

with start state 1 and final states 2 and 3. It is easy to see that

$$L(D) = \{a\} \cup \{b^m \mid m \geq 1\}.$$

It is not hard to check that states 4 and 5 are equivalent, and no other pairs of distinct states are equivalent. The quotient DFA $D/\equiv$ is obtained my merging states 4 and 5, and we obtain the following minimal DFA:

	a	b
1	2	3
2	4	4
3	4	3
4	4	4

with start state 1 and final states 2 and 3. This DFA is isomorphic to the DFA D_{ρ_L} of Example 5.10.

There are other characterizations of the regular languages. Among those, the characterization in terms of right derivatives is of particular interest because it yields an alternative construction of minimal DFA's.

5.12 The Pumping Lemma

Another useful tool for proving that languages are not regular is the so-called *pumping lemma.*

Proposition 5.17. *(Pumping lemma) Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, there is some $m \geq 1$ such that for every $w \in \Sigma^*$, if $w \in L(D)$ and $|w| \geq m$, then there exists a decomposition of w as $w = uxv$, where*

(1) $x \neq \epsilon$,

(2) $ux^iv \in L(D)$, for all $i \geq 0$, and

(3) $|ux| \leq m$.

Moreover, m can be chosen to be the number of states of the DFA D.

Proof. Let m be the number of states in Q, and let $w = w_1 \ldots w_n$. Since Q contains the start state q_0, $m \geq 1$. Since $|w| \geq m$, we have $n \geq m$. Since $w \in L(D)$, let $(q_0, q_1, \ldots, q_n)$, be the sequence of states in the accepting computation of w (where $q_n \in F$). Consider the subsequence

$$(q_0, q_1, \ldots, q_m).$$

This sequence contains $m+1$ states, but there are only m states in Q, and thus, we have $q_i = q_j$, for some i, j such that $0 \leq i < j \leq m$. Then, letting $u = w_1 \ldots w_i$, $x = w_{i+1} \ldots w_j$, and $v = w_{j+1} \ldots w_n$, it is clear that the conditions of the proposition hold. $\square$

An important consequence of the pumping lemma is that if a DFA D has m states and if there is some string $w \in L(D)$ such that $|w| \geq m$, then $L(D)$ is infinite.

Indeed, by the pumping lemma, $w \in L(D)$ can be written as $w = uxv$ with $x \neq \epsilon$, and

$$ux^iv \in L(D) \quad \text{for all } i \geq 0.$$

Since $x \neq \epsilon$, we have $|x| > 0$, so for all $i, j \geq 0$ with $i < j$ we have

$$|ux^iv| < |ux^iv| + (j-i)|x| = |ux^jv|,$$

which implies that $ux^iv \neq ux^jv$ for all $i < j$, and the set of strings

$$\{ux^iv \mid i \geq 0\} \subseteq L(D)$$

is an *infinite* subset of $L(D)$, which is itself infinite.

As a consequence, if $L(D)$ is finite, there are *no* strings w in $L(D)$ such that $|w| \geq m$. In this case, since the premise of the pumping lemma is false, the pumping lemma holds vacuously; that is, if $L(D)$ is finite, the pumping lemma yields no information.

Another corollary of the pumping lemma is that there is a test to decide whether a DFA D accepts an infinite language $L(D)$.

Proposition 5.18. *Let D be a DFA with m states, The language $L(D)$ accepted by D is infinite iff there is some string $w \in L(D)$ such that $m \leq |w| < 2m$.*

If $L(D)$ is infinite, there are strings of length $\geq m$ in $L(D)$, but a prirori there is no guarantee that there are "short" strings w in $L(D)$, that is, strings whose length is uniformly bounded by some function of m independent of D. The pumping lemma ensures that there are such strings, and the function is $m \mapsto 2m$.

Typically, the pumping lemma is used to prove that a language is not regular. The method is to proceed by contradiction, i.e., to assume (contrary to what we wish to prove) that a language L is indeed regular, and derive a contradiction of the pumping lemma. Thus, it would be helpful to see what the negation of the pumping lemma is, and for this, we first state the pumping lemma as a logical formula. We will use the following abbreviations:

$$\begin{aligned}
nat &= \{0, 1, 2, \ldots\}, \\
pos &= \{1, 2, \ldots\}, \\
A &\equiv w = uxv, \\
B &\equiv x \neq \epsilon, \\
C &\equiv |ux| \leq m, \\
P &\equiv \forall i \colon nat\, (ux^iv \in L(D)).
\end{aligned}$$

The pumping lemma can be stated as

$$\forall D\colon \text{DFA}\ \exists m\colon pos\ \forall w\colon \Sigma^*\Big((w \in L(D) \wedge |w| \geq m) \supset (\exists u, x, v\colon \Sigma^*\ A \wedge B \wedge C \wedge P)\Big).$$

Recalling that

$$\neg(A \wedge B \wedge C \wedge P) \equiv \neg(A \wedge B \wedge C) \vee \neg P \equiv (A \wedge B \wedge C) \supset \neg P$$

and

$$\neg(R \supset S) \equiv R \wedge \neg S,$$

the negation of the pumping lemma can be stated as

$$\exists D\colon \text{DFA}\ \forall m\colon pos\ \exists w\colon \Sigma^*\Big((w \in L(D) \wedge |w| \geq m) \wedge (\forall u, x, v\colon \Sigma^*\ (A \wedge B \wedge C) \supset \neg P)\Big).$$

Since

$$\neg P \equiv \exists i\colon nat\ (ux^i v \notin L(D)),$$

in order to show that the pumping lemma is contradicted, one needs to show that for some DFA D, for every $m \geq 1$, there is some string $w \in L(D)$ of length at least m, such that for every possible decomposition $w = uxv$ satisfying the constraints $x \neq \epsilon$ and $|ux| \leq m$, there is some $i \geq 0$ such that $ux^i v \notin L(D)$.

When proceeding by contradiction, we have a language L that we are (wrongly) assuming to be regular, and we can use any DFA D accepting L. The creative part of the argument is to pick the right $w \in L$ (not making any assumption on $m \leq |w|$).

As an illustration, let us use the pumping lemma to prove that $L_1 = \{a^n b^n \mid n \geq 1\}$ is not regular. The usefulness of the condition $|ux| \leq m$ lies in the fact that it reduces the number of legal decomposition uxv of w. We proceed by contradiction. Thus, let us assume that $L_1 = \{a^n b^n \mid n \geq 1\}$ is regular. If so, it is accepted by some DFA D. Now, we wish to contradict the pumping lemma. For every $m \geq 1$, let $w = a^m b^m$. Clearly, $w = a^m b^m \in L_1$ and $|w| \geq m$. Then, every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u}\underbrace{a \ldots a}_{x}\underbrace{a \ldots ab \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. Since $x \neq \epsilon$, the string $uxxv$ is of the form $a^n b^m$ where $n > m$, and thus $uxxv \notin L_1$, contradicting the pumping lemma.

Let us consider two more examples. let $L_2 = \{a^m b^n \mid 1 \leq m < n\}$. We claim that L_2 is not regular. Our first proof uses the pumping lemma. For any $m \geq 1$, pick $w = a^m b^{m+1}$. We have $w \in L_2$ and $|w| \geq m$ so we need to contradict the pumping lemma. Every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u}\underbrace{a \ldots a}_{x}\underbrace{a \ldots ab \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. Since $x \neq \epsilon$ and x consists of a's the string $ux^2v = uxxv$ contains at least $m+1$ a's and still $m+1$ b's, so $ux^2v \notin L_2$, contradicting the pumping lemma.

Our second proof uses Myhill-Nerode. Let $\simeq$ be a right-invariant equivalence relation of finite index such that L_2 is the union of classes of $\simeq$. If we consider the infinite sequence

$$a, a^2, \ldots, a^n, \ldots$$

since $\simeq$ has a finite number of classes there are two strings a^m and a^n with $m < n$ such that

$$a^m \simeq a^n.$$

By right-invariance by concatenating on the right with b^n we obtain

$$a^m b^n \simeq a^n b^n,$$

and since $m < n$ we have $a^m b^n \in L_2$ but $a^n b^n \notin L_2$, a contradiction.

Let us now consider the language $L_3 = \{a^m b^n \mid m \neq n\}$. This time let us begin by using Myhill-Nerode to prove that L_3 is not regular. The proof is the same as before, we obtain

$$a^m b^n \simeq a^n b^n,$$

and the contradiction is that $a^m b^n \in L_3$ and $a^n b^n \notin L_3$.

Let use now try to use the pumping lemma to prove that L_3 is not regular. For any $m \geq 1$ pick $w = a^m b^{m+1} \in L_3$. As in the previous case, every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u}\underbrace{a \ldots a}_{x}\underbrace{a \ldots ab \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. However this time we have a problem, namely that we know that x is a nonempty string of a's but we don't know how many, so we can't guarantee that pumping up x will yield exactly the string $a^{m+1}b^{m+1}$. We made the wrong choice for w. There is a choice that will work but it is a bit tricky.

Fortunately, there is another simpler approach. Recall that the regular languages are closed under the boolean operations (union, intersection and complementation). Thus, L_3 is not regular iff its complement $\overline{L}_3$ is not regular. Observe that $\overline{L}_3$ contains $\{a^n b^n \mid n \geq 1\}$, which we showed to be nonregular. But there is another problem, which is that $\overline{L}_3$ contains other strings besides strings of the form $a^n b^n$, for example strings of the form $b^m a^n$ with $m, n > 0$.

Again, we can take care of this difficulty using the closure operations of the regular languages. If we can find a regular language R such that $\overline{L}_3 \cap R$ is not regular, then $\overline{L}_3$ itself is not regular, since otherwise as $\overline{L}_3$ and R are regular then $\overline{L}_3 \cap R$ is also regular. In our case, we can use $R = \{a\}^+\{b\}^+$ to obtain

$$\overline{L}_3 \cap \{a\}^+\{b\}^+ = \{a^n b^n \mid n \geq 1\}.$$