Chapter 1

INTRODUCTION TO THE THEORY OF COMPUTATION

Computer science is a practical discipline. Those who work in it often have a marked preference for useful and tangible problems over theoretical speculation. This is certainly true of computer science students who are interested mainly in working on difficult applications from the real world. Theoretical questions are interesting to them only if they help in finding good solutions. This attitude is appropriate, since without applications there would be little interest in computers. But given this practical orientation, one might well ask "why study theory?"

The first answer is that theory provides concepts and principles that help us understand the general nature of the discipline. The field of computer science includes a wide range of special topics, from machine design to programming. The use of computers in the real world involves a wealth of specific detail that must be learned for a successful application. This makes computer science a very diverse and broad discipline. But in spite of this diversity, there are some common underlying principles. To study these basic principles, we construct abstract models of computers and computation. These models embody the important features that are common to both hardware and software, and that are essential to many of the special and complex constructs we encounter while working with computers. Even

when such models are too simple to be applicable immediately to real-world situations, the insights we gain from studying them provide the foundations on which specific development is based. This approach is of course not unique to computer science. The construction of models is one of the essentials of any scientific discipline, and the usefulness of a discipline is often dependent on the existence of simple, yet powerful, theories and laws.

A second, and perhaps not so obvious answer, is that the ideas we will discuss have some immediate and important applications. The fields of digital design, programming languages, and compilers are the most obvious examples, but there are many others. The concepts we study here run like a thread through much of computer science, from operating systems to pattern recognition.

The third answer is one of which we hope to convince the reader. The subject matter is intellectually stimulating and fun. It provides many challenging, puzzle-like problems that can lead to some sleepless nights. This is problem-solving in its pure essence.

In this book, we will look at models that represent features at the core of all computers and their applications. To model the hardware of a computer, we introduce the notion of an **automaton** (plural, **automata**). An automaton is a construct that possesses all the indispensable features of a digital computer. It accepts input, produces output, may have some temporary storage, and can make decisions in transforming the input into the output. A **formal language** is an abstraction of the general characteristics of programming languages. A formal language consists of a set of symbols and some rules of formation by which these symbols can be combined into entities called sentences. A formal language is the set of all strings permitted by the rules of formation. Although some of the formal languages we study here are simpler than programming languages, they have many of the same essential features. We can learn a great deal about programming languages from formal languages. Finally, we will formalize the concept of a mechanical computation by giving a precise definition of the term **algorithm** and study the kinds of problems that are (and are not) suitable for solution by such mechanical means. In the course of our study, we will show the close connection between these abstractions and investigate the conclusions we can derive from them.

In the first chapter, we look at these basic ideas in a very broad way to set the stage for later work. In Section 1.1, we review the main ideas from mathematics that will be required. While intuition will frequently be our guide in exploring ideas, the conclusions we draw will be based on rigorous arguments. This will involve some mathematical machinery, although these requirements are not extensive. The reader will need a reasonably good grasp of the terminology and of the elementary results of set theory, functions, and relations. Trees and graph structures will be used frequently, although little is needed beyond the definition of a labeled, directed graph. Perhaps the most stringent requirement is the ability to follow proofs and

an understanding of what constitutes proper mathematical reasoning. This includes familiarity with the basic proof techniques of deduction, induction, and proof by contradiction. We will assume that the reader has this necessary background. Section 1.1 is included to review some of the main results that will be used and to establish a notational common ground for subsequent discussion.

In Section 1.2, we take a first look at the central concepts of languages, grammars, and automata. These concepts occur in many specific forms throughout the book. In Section 1.3, we give some simple applications of these general ideas to illustrate that these concepts have widespread uses in computer science. The discussion in these two sections will be intuitive rather than rigorous. Later, we will make all of this much more precise; but for the moment, the goal is to get a clear picture of the concepts with which we are dealing.

1.1 Mathematical Preliminaries and Notation

Sets

A **set** is a collection of elements, without any structure other than membership. To indicate that x is an element of the set S, we write $x \in S$. The statement that x is not in S is written $x \notin S$. A set is specified by enclosing some description of its elements in curly braces; for example, the set of integers 0, 1, 2 is shown as

$$S = \{0, 1, 2\}.$$

Ellipses are used whenever the meaning is clear. Thus, $\{a, b, \ldots, z\}$ stands for all the lower-case letters of the English alphabet, while $\{2, 4, 6, \ldots\}$ denotes the set of all positive even integers. When the need arises, we use more explicit notation, in which we write

$$S = \{i : i > 0, \, i \text{ is even}\} \tag{1.1}$$

for the last example. We read this as "S is set of all i, such that i is greater than zero, and i is even," implying of course that i is an integer.

The usual set operations are **union** ($\cup$), **intersection** ($\cap$), and **difference** ($-$), defined as

$$\begin{aligned} S_1 \cup S_2 &= \{x : x \in S_1 \text{ or } x \in S_2\}, \\ S_1 \cap S_2 &= \{x : x \in S_1 \text{ and } x \in S_2\}, \\ S_1 - S_2 &= \{x : x \in S_1 \text{ and } x \notin S_2\}. \end{aligned}$$

Another basic operation is **complementation.** The complement of a set S, denoted by $\overline{S}$, consists of all elements not in S. To make this

meaningful, we need to know what the **universal set** U of all possible elements is. If U is specified, then

$$\overline{S} = \{x : x \in U, x \notin S\}.$$

The set with no elements, called the **empty set** or the **null set** is denoted by $\varnothing$. From the definition of a set, it is obvious that

$$\begin{aligned} S \cup \varnothing &= S - \varnothing = S, \\ S \cap \varnothing &= \varnothing, \\ \overline{\varnothing} &= U, \\ \overline{\overline{S}} &= S. \end{aligned}$$

The following useful identities, known as the **DeMorgan's laws,**

$$\overline{S_1 \cup S_2} = \overline{S}_1 \cap \overline{S}_2, \tag{1.2}$$

$$\overline{S_1 \cap S_2} = \overline{S}_1 \cup \overline{S}_2, \tag{1.3}$$

are needed on several occasions.

A set S_1 is said to be a **subset** of S if every element of S_1 is also an element of S. We write this as

$$S_1 \subseteq S.$$

If $S_1 \subseteq S$, but S contains an element not in S_1 we say that S_1 is a **proper subset** of S; we write this as

$$S_1 \subset S.$$

If S_1 and S_2 have no common element, that is, $S_1 \cap S_2 = \varnothing$, then the sets are said to be **disjoint.**

A set is said to be finite if it contains a finite number of elements; otherwise it is infinite. The size of a finite set is the number of elements in it; this is denoted by $|S|$.

A given set normally has many subsets. The set of all subsets of a set S is called the **powerset** of S and is denoted by 2^S. Observe that 2^S is a set of sets.

Example 1.1 If S is the set $\{a, b, c\}$, then its powerset is

$$2^S = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$

Here $|S| = 3$ and $\left|2^S\right| = 8$. This is an instance of a general result; if S is finite, then

$$\left|2^S\right| = 2^{|S|}.$$

■

In many of our examples, the elements of a set are ordered sequences of elements from other sets. Such sets are said to be the **Cartesian product** of other sets. For the Cartesian product of two sets, which itself is a set of ordered pairs, we write

$$S = S_1 \times S_2 = \{(x, y) : x \in S_1, y \in S_2\}.$$

Example 1.2 Let $S_1 = \{2, 4\}$ and $S_2 = \{2, 3, 5, 6\}$. Then

$$S_1 \times S_2 = \{(2,2), (2,3), (2,5), (2,6), (4,2), (4,3), (4,5), (4,6)\}.$$

Note that the order in which the elements of a pair are written matters. The pair $(4, 2)$ is in $S_1 \times S_2$, but $(2, 4)$ is not.

The notation is extended in an obvious fashion to the Cartesian product of more than two sets; generally

$$S_1 \times S_2 \times \cdots \times S_n = \{(x_1, x_2, ..., x_n) : x_i \in S_i\}.$$

■

Functions and Relations

A **function** is a rule that assigns to elements of one set a unique element of another set. If f denotes a function, then the first set is called the **domain** of f, and the second set is its **range.** We write

$$f : S_1 \to S_2$$

to indicate that the domain of f is a subset of S_1 and that the range of f is a subset of S_2. If the domain of f is all of S_1, we say that f is a **total function** on S_1; otherwise f is said to be a **partial function.**

In many applications, the domain and range of the functions involved are in the set of positive integers. Furthermore, we are often interested only in the behavior of these functions as their arguments become very large. In such cases an understanding of the growth rates is often sufficient and a common order of magnitude notation can be used. Let $f(n)$ and $g(n)$ be functions whose domain is a subset of the positive integers. If there exists a positive constant c such that for all n

$$f(n) \leq cg(n),$$

we say that f has **order at most** g. We write this as

$$f(n) = O(g(n)).$$

If

$$|f(n)| \geq c\,|g(n)|,$$

then f has **order at least** g, for which we use

$$f(n) = \Omega(g(n)).$$

Finally, if there exist constants c_1 and c_2 such that

$$c_1\,|g(n)| \leq |f(n)| \leq c_2\,|g(n)|,$$

f and g have the **same order of magnitude**, expressed as

$$f(n) = \Theta(g(n)).$$

In this order of magnitude notation, we ignore multiplicative constants and lower order terms that become negligible as n increases.

Example 1.3 Let

$$f(n) = 2n^2 + 3n,$$
$$g(n) = n^3,$$
$$h(n) = 10n^2 + 100.$$

Then

$$f(n) = O(g(n)),$$
$$g(n) = \Omega(h(n)),$$
$$f(n) = \Theta(h(n)).$$

In order of magnitude notation, the symbol = should not be interpreted as equality and order of magnitude expressions cannot be treated like ordinary expressions. Manipulations such as

$$O(n) + O(n) = 2O(n)$$

are not sensible and can lead to incorrect conclusions. Still, if used properly, the order of magnitude arguments can be effective, as we will see in later chapters on the analysis of algorithms. ■

Some functions can be represented by a set of pairs

$$\{(x_1, y_1), (x_2, y_2), \ldots\},$$

where x_i is an element in the domain of the function, and y_i is the corresponding value in its range. For such a set to define a function, each x_i can occur at most once as the first element of a pair. If this is not satisfied, the

set is called a **relation.** Relations are more general than functions: in a function each element of the domain has exactly one associated element in the range; in a relation there may be several such elements in the range.

One kind of relation is that of **equivalence,** a generalization of the concept of equality (identity). To indicate that a pair (x, y) is an equivalence relation, we write

$$x \equiv y.$$

A relation denoted by $\equiv$ is considered an equivalence if it satisfies three rules: the reflexivity rule

$$x \equiv x \text{ for all } x,$$

the symmetry rule

$$\text{if } x \equiv y \text{ then } y \equiv x,$$

and the transitivity rule

$$\text{if } x \equiv y \text{ and } y \equiv z, \text{ then } x \equiv z.$$

Example 1.4 Consider the relation on the set of nonnegative integers defined by

$$x \equiv y,$$

if and only if

$$x \bmod 3 = y \bmod 3.$$

Then $2 \equiv 5$, $12 \equiv 0$, and $0 \equiv 36$. Clearly this is an equivalence relation, as it satisfies reflexivity, symmetry, and transitivity. ■

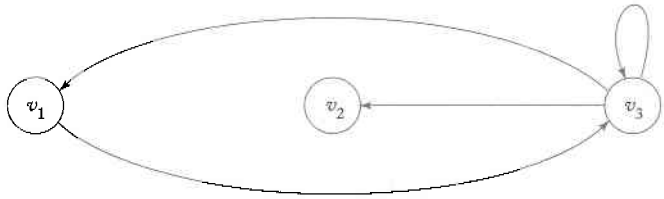
Graphs and Trees

A graph is a construct consisting of two finite sets, the set $V = \{v_1, v_2, ..., v_n\}$ of **vertices** and the set $E = \{e_1, e_2, ..., e_m\}$ of **edges.** Each edge is a pair of vertices from V, for instance

$$e_i = (v_j, v_k)$$

is an edge from v_j to v_k. We say that the edge e_i is an outgoing edge for v_j and an incoming edge for v_k. Such a construct is actually a directed graph (digraph), since we associate a direction (from v_j to v_k) with each edge. Graphs may be labeled, a label being a name or other information associated with parts of the graph. Both vertices and edges may be labeled.

Figure 1.1



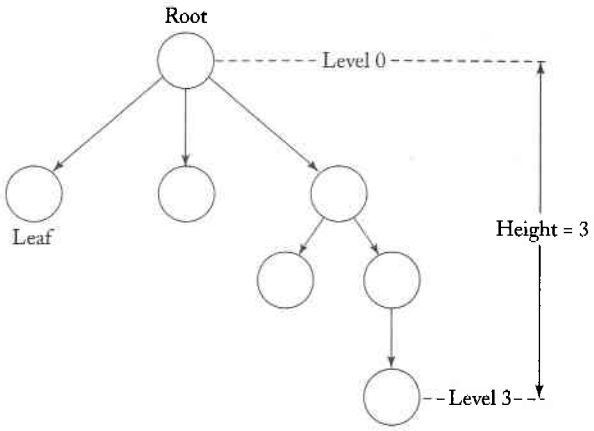
Graphs are conveniently visualized by diagrams in which the vertices are represented as circles and the edges as lines with arrows connecting the vertices. The graph with vertices $\{v_1, v_2, v_3\}$ and edges $\{(v_1, v_3), (v_3, v_1), (v_3, v_2), (v_3, v_3)\}$ is depicted in Figure 1.1.

A sequence of edges $(v_i, v_j), (v_j, v_k), \ldots, (v_m, v_n)$ is said to be a **walk** from v_i to v_n. The length of a walk is the total number of edges traversed in going from the initial vertex to the final one. A walk in which no edge is repeated is said to be a **path;** a path is **simple** if no vertex is repeated. A walk from v_i to itself with no repeated edges is called a **cycle** with **base** v_i. If no vertices other than the base are repeated in a cycle, then it is said to be simple. In Figure 1.1, (v_1, v_3), (v_3, v_2) is a simple path from v_1 to v_2. The sequence of edges (v_1, v_3), (v_3, v_3), (v_3, v_1) is a cycle, but not a simple one. If the edges of a graph are labeled, we can talk about the label of a walk. This label is the sequence of edge labels encountered when the path is traversed. Finally, an edge from a vertex to itself is called a **loop**. In Figure 1.1 there is a loop on vertex v_3.

On several occasions, we will refer to an algorithm for finding all simple paths between two given vertices (or all simple cycles based on a vertex). If we do not concern ourselves with efficiency, we can use the following obvious method. Starting from the given vertex, say v_i, list all outgoing edges (v_i, v_k), $(v_i, v_l), \ldots$. At this point, we have all paths of length one starting at v_i. For all vertices $v_k, v_l, \ldots$ so reached, we list all outgoing edges as long as they do not lead to any vertex already used in the path we are constructing. After we do this, we will have all simple paths of length two originating at v_i. We continue this until all possibilities are accounted for. Since there are only a finite number of vertices, we will eventually list all simple paths beginning at v_i. From these we select those ending at the desired vertex.

Trees are a particular type of graph. A tree is a directed graph that has no cycles, and that has one distinct vertex, called the **root,** such that there is exactly one path from the root to every other vertex. This definition implies that the root has no incoming edges and that there are some vertices without outgoing edges. These are called the **leaves** of the tree. If there is an edge from v_i to v_j, then v_i is said to be the **parent** of v_j, and v_j the **child** of v_i. The **level** associated with each vertex is the number of edges in the path from the root to the vertex. The **height** of the tree is the largest level number of any vertex. These terms are illustrated in Figure 1.2.

Figure 1.2



At times, we want to associate an ordering with the nodes at each level; in such cases we talk about **ordered trees.**

More details on graphs and trees can be found in most books on discrete mathematics.

Proof Techniques

An important requirement for reading this text is the ability to follow proofs. In mathematical arguments, we employ the accepted rules of deductive reasoning, and many proofs are simply a sequence of such steps. Two special proof techniques are used so frequently that it is appropriate to review them briefly. These are **proof by induction** and **proof by contradiction.**

Induction is a technique by which the truth of a number of statements can be inferred from the truth of a few specific instances. Suppose we have a sequence of statements $P_1, P_2, \ldots$ we want to prove to be true. Furthermore, suppose also that the following holds:

1. For some $k \geq 1$, we know that $P_1, P_2, \ldots, P_k$ are true.
2. The problem is such that for any $n \geq k$, the truths of $P_1, P_2, \ldots, P_n$ imply the truth of P_{n+1}.

We can then use induction to show that every statement in this sequence is true.

In a proof by induction, we argue as follows: From Condition 1 we know that the first k statements are true. Then Condition 2 tells us that P_{k+1} also must be true. But now that we know that the first $k+1$ statements are true, we can apply Condition 2 again to claim that P_{k+2} must be true, and so on. We need not explicitly continue this argument because the pattern is clear. The chain of reasoning can be extended to any statement. Therefore, every statement is true.

Chapter 2

FINITE AUTOMATA

Our introduction in the first chapter to the basic concepts of computation, particularly the discussion of automata, was brief and informal. At this point, we have only a general understanding of what an automaton is and how it can be represented by a graph. To progress, we must be more precise, provide formal definitions, and start to develop rigorous results. We begin with finite accepters, which are a simple, special case of the general scheme introduced in the last chapter. This type of automaton is characterized by having no temporary storage. Since an input file cannot be rewritten, a finite automaton is severely limited in its capacity to "remember" things during the computation. A finite amount of information can be retained in the control unit by placing the unit into a specific state. But since the number of such states is finite, a finite automaton can only deal with situations in which the information to be stored at any time is strictly bounded. The automaton in Example 1.16 is an instance of a finite accepter.

2.1 Deterministic Finite Accepters

The first type of automaton we study in detail are finite accepters that are deterministic in their operation. We start with a precise formal definition of deterministic accepters.

Deterministic Accepters and Transition Graphs

Definition 2.1

A **deterministic finite accepter** or **dfa** is defined by the quintuple

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

where

Q is a finite set of **internal states,**
Σ is a finite set of symbols called the **input alphabet,**
$\delta : Q \times \Sigma \to Q$ is a total function called the **transition function,**
$q_0 \in Q$ is the **initial state,**
$F \subseteq Q$ is a set of **final states.**

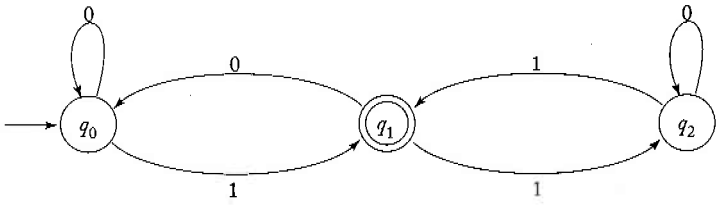
A deterministic finite accepter operates in the following manner. At the initial time, it is assumed to be in the initial state q_0, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, the input mechanism advances one position to the right, so each move consumes one input symbol. When the end of the string is reached, the string is accepted if the automaton is in one of its final states. Otherwise the string is rejected. The input mechanism can move only from left to right and reads exactly one symbol on each step. The transitions from one internal state to another are governed by the transition function δ. For example, if

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

then if the dfa is in state q_0 and the current input symbol is a, the dfa will go into state q_1.

In discussing automata, it is essential to have a clear and intuitive picture to work with. To visualize and represent finite automata, we use **transition graphs,** in which the vertices represent states and the edges represent transitions. The labels on the vertices are the names of the states, while the labels on the edges are the current values of the input symbol. For example, if q_0 and q_1 are internal states of some dfa M, then the graph associated with M will have one vertex labeled q_0 and another labeled q_1. An edge (q_0, q_1) labeled a represents the transition $\delta(q_0, a) = q_1$. The initial

Figure 2.1



state will be identified by an incoming unlabeled arrow not originating at any vertex. Final states are drawn with a double circle.

More formally, if $M = (Q, \Sigma, \delta, q_0, F)$ is a deterministic finite accepter, then its associated transition graph G_M has exactly $|Q|$ vertices, each one labeled with a different $q_i \in Q$. For every transition rule $\delta(q_i, a) = q_j$, the graph has an edge (q_i, q_j) labeled a. The vertex associated with q_0 is called the **initial vertex,** while those labeled with $q_f \in F$ are the **final vertices.** It is a trivial matter to convert from the $(Q, \Sigma, \delta, q_0, F)$ definition of a dfa to its transition graph representation and vice versa.

Example 2.1 The graph in Figure 2.1 represents the dfa

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}),$$

where δ is given by

$$\begin{aligned} \delta(q_0, 0) &= q_0, & \delta(q_0, 1) &= q_1, \\ \delta(q_1, 0) &= q_0, & \delta(q_1, 1) &= q_2, \\ \delta(q_2, 0) &= q_2, & \delta(q_2, 1) &= q_1. \end{aligned}$$

This dfa accepts the string 01. Starting in state q_0, the symbol 0 is read first. Looking at the edges of the graph, we see that the automaton remains in state q_0. Next, the 1 is read and the automaton goes into state q_1. We are now at the end of the string and, at the same time, in a final state q_1. Therefore, the string 01 is accepted. The dfa does not accept the string 00, since after reading two consecutive 0's, it will be in state q_0. By similar reasoning, we see that the automaton will accept the strings 101, 0111, and 11001, but not 100 or 1100. ■

$\delta^* : Q \times \Sigma^* \to Q$

It is convenient to introduce the extended transition function $\delta^* : Q \times \Sigma^* \to Q$. The second argument of δ^* is a string, rather than a single symbol, and its value gives the state the automaton will be in after reading that string. For example, if

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

and

$$\delta(q_1, b) = q_2,$$

then

$$\delta^*(q_0, ab) = q_2.$$

Formally, we can define δ^* recursively by

$$\delta^*(q, \lambda) = q, \tag{2.1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a), \tag{2.2}$$

for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$. To see why this is appropriate, let us apply these definitions to the simple case above. First, we use (2.2) to get

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b). \tag{2.3}$$

But

$$\begin{aligned}\delta^*(q_0, a) &= \delta(\delta^*(q_0, \lambda), a) \\ &= \delta(q_0, a) \\ &= q_1.\end{aligned}$$

Substituting this into (2.3), we get

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2,$$

as expected.

Languages and Dfa's

Having made a precise definition of an accepter, we are now ready to define formally what we mean by an associated language. The association is obvious: the language is the set of all the strings accepted by the automaton.

Definition 2.2

The language accepted by a dfa $M = (Q, \Sigma, \delta, q_0, F)$ is the set of all strings on Σ accepted by M. In formal notation,

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

Note that we require that δ, and consequently δ^*, be total functions. At each step, a unique move is defined, so that we are justified in calling such an automaton deterministic. A dfa will process every string in Σ^* and either accept it or not accept it. Nonacceptance means that the dfa stops in a nonfinal state, so that

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}.$$

Example 2.2 Consider the dfa in Figure 2.2

In drawing Figure 2.2 we allowed the use of two labels on a single edge. Such multiply labeled edges are shorthand for two or more distinct transitions: the transition is taken whenever the input symbol matches any of the edge labels.

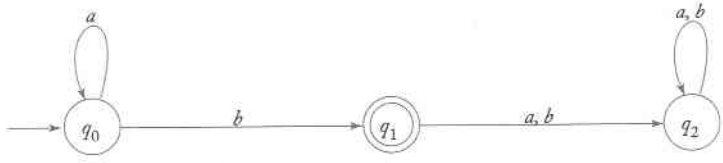
The automaton in Figure 2.2 remains in its initial state q_0 until the first b is encountered. If this is also the last symbol of the input, then the string is accepted. If not, the dfa goes into state q_2, from which it can never escape. The state q_2 is a **trap state.** We see clearly from the graph that the automaton accepts all strings consisting of an arbitrary number of a's, followed by a single b. All other input strings are rejected. In set notation, the language accepted by the automaton is

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

■

These examples show how convenient transition graphs are for working with finite automata. While it is possible to base all arguments strictly on the properties of the transition function and its extension through (2.1) and (2.2), the results are hard to follow. In our discussion, we use graphs, which are more intuitive, as far as possible. To do so, we must of course have some assurance that we are not misled by the representation and that arguments based on graphs are as valid as those that use the formal properties of δ. The following preliminary result gives us this assurance.

Figure 2.2



Theorem 2.1 Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite accepter, and let G_M be its associated transition graph. Then for every q_i, $q_j \in Q$, and $w \in \Sigma^+$, $\delta^*(q_i, w) = q_j$ if and only if there is in G_M a walk with label w from q_i to q_j.

Proof: This claim is fairly obvious from an examination of such simple cases as Example 2.1. It can be proved rigorously using an induction on the length of w. Assume that the claim is true for all strings v with $|v| \leq n$. Consider then any w of length $n+1$ and write it as

$$w = va.$$

Suppose now that $\delta^*(q_i, v) = q_k$. Since $|v| = n$, there must be a walk in G_M labeled v from q_i to q_k. But if $\delta^*(q_i, w) = q_j$, then M must have a transition $\delta(q_k, a) = q_j$, so that by construction G_M has an edge (q_k, q_j) with label a. Thus there is a walk in G_M labeled $va = w$ between q_i and q_j. Since the result is obviously true for $n = 1$, we can claim by induction that, for every $w \in \Sigma^+$,

$$\delta^*(q_i, w) = q_j \tag{2.4}$$

implies that there is a walk in G_M from q_i to q_j labeled w.

The argument can be turned around in a straightforward way to show that the existence of such a path implies (2.4), thus completing the proof. ■

Again, the result of the theorem is so intuitively obvious that a formal proof seems unnecessary. We went through the details for two reasons. The first is that it is a simple, yet typical example of an inductive proof in connection with automata. The second is that the result will be used over and over, so stating and proving it as a theorem lets us argue quite confidently using graphs. This makes our examples and proofs more transparent than they would be if we used the properties of δ^*.

While graphs are convenient for visualizing automata, other representations are also useful. For example, we can represent the function δ as a table. The table in Figure 2.3 is equivalent to Figure 2.2. Here the row label is the current state, while the column label represents the current input symbol. The entry in the table defines the next state.

It is apparent from this example that a dfa can easily be implemented as a computer program; for example, as a simple table-lookup or as a sequence of "if" statements. The best implementation or representation depends on the specific application. Transition graphs are very convenient for the kinds of arguments we want to make here, so we use them in most of our discussions.

In constructing automata for languages defined informally, we employ reasoning similar to that for programming in higher-level languages. But the

Figure 2.3

δ	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

programming of a dfa is tedious and sometimes conceptually complicated by the fact that such an automaton has few powerful features.

Example 2.3 Find a deterministic finite accepter that recognizes the set of all strings on $\Sigma = \{a, b\}$ starting with the prefix ab.

The only issue here is the first two symbols in the string; after they have been read, no further decisions need to be made. We can therefore solve the problem with an automaton that has four states; an initial state, two states for recognizing ab ending in a final trap state, and one nonfinal trap state. If the first symbol is an a and the second is a b, the automaton goes to the final trap state, where it will stay since the rest of the input does not matter. On the other hand, if the first symbol is not an a or the second one is not b, the automaton enters the nonfinal trap state. The simple solution is shown in Figure 2.4. ■

Figure 2.4

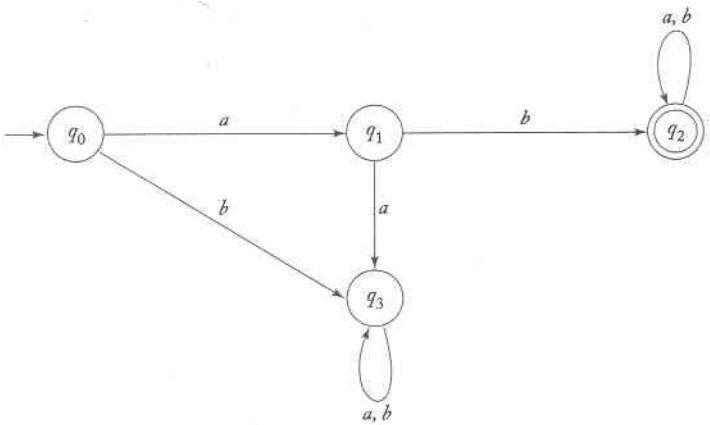
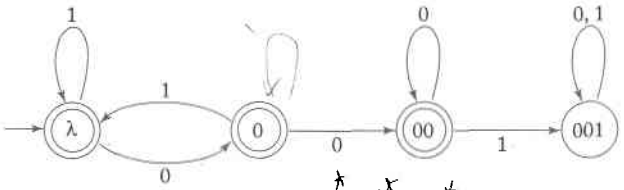


Figure 2.5



Example 2.4 Find a dfa that accepts all the strings on $\{0, 1\}$, except those containing the substring 001.

In deciding whether the substring 001 has occurred, we need to know not only the current input symbol, but we also need to remember whether or not it has been preceded by one or two 0s. We can keep track of this by putting the automaton into specific states and labeling them accordingly. Like variable names in a programming language, state names are arbitrary and can be chosen for mnemonic reasons. For example, the state in which two 0s were the immediately preceding symbols can be labeled simply 00.

If the string starts with 001, then it must be rejected. This implies that there must be a path labeled 001 from the initial state to a nonfinal state. For convenience, this nonfinal state is labeled 001. This state must be a trap state, because later symbols do not matter. All other states are accepting states.

This gives us the basic structure of the solution, but we still must add provisions for the substring 001 occurring in the middle of the input. We must define Q and δ so that whatever we need to make the correct decision is remembered by the automaton. In this case, when a symbol is read, we need to know some part of string to the left, for example, whether or not the two previous symbols were 00. If we label the states with the relevant symbols, it is very easy to see what the transitions must be. For example,

$$\delta(00, 0) = 00,$$

because this situation arises only if there are three consecutive 0s. We are only interested in the last two, a fact we remember by keeping the dfa in the state 00. A complete solution is shown in Figure 2.5. We see from this example how useful mnemonic labels on the states are for keeping track of things. Trace a few strings, such as 100100 and 1010100, to see that the solution is indeed correct. ■

Regular Languages

Every finite automaton accepts some language. If we consider all possible finite automata, we get a set of languages associated with them. We will call such a set of languages a **family.** The family of languages that is accepted by deterministic finite accepters is quite limited. The structure and properties

of the languages in this family will become clearer as our study proceeds; for the moment we will simply attach a name to this family.

Definition 2.3

regular Lan

A language L is called **regular** if and only if there exists some deterministic finite accepter M such that

$$L = L(M).$$

Example 2.5 Show that the language

$$L = \{awa : w \in \{a, b\}^*\}$$

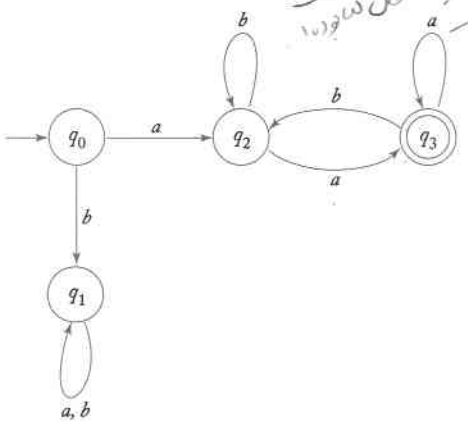
is regular. To show that this or any other language is regular, all we have to do is find a dfa for it. The construction of a dfa for this language is similar to Example 2.3, but a little more complicated. What this dfa must do is check whether a string begins and ends with an a; what is between is immaterial. The solution is complicated by the fact that there is no explicit way of testing the end of the string. This difficulty is overcome by simply putting the dfa into a final state whenever the second a is encountered. If this is not the end of the string, and another b is found, it will take the dfa out of the final state. Scanning continues in this way, each a taking the automaton back to its final state. The complete solution is shown in Figure 2.6. Again, trace a few examples to see why this works. After one or two tests, it will be obvious that the dfa accepts a string if and only if it begins and ends with an a. Since we have constructed a dfa for the language, we can claim that, by definition, the language is regular. ■

Example 2.6 Let L be the language in Example 2.5. Show that L^2 is regular. Again we show that the language is regular by constructing a dfa for it. We can write an explicit expression for L^2, namely,

$$L^2 = \{aw_1aaw_2a : w_1, w_2 \in \{a, b\}^*\}.$$

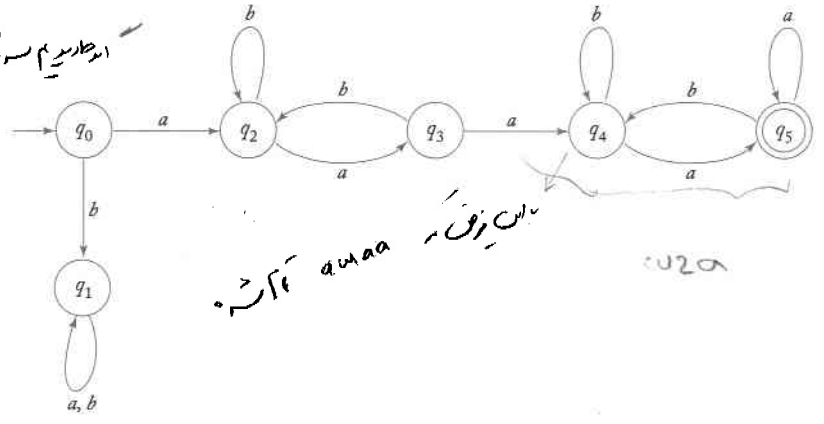
Therefore, we need a dfa that recognizes two consecutive strings of essentially the same form (but not necessarily identical in value). The diagram

Figure 2.6



in Figure 2.6 can be used as a starting point, but the vertex q_3 has to be modified. This state can no longer be final since, at this point, we must start to look for a second substring of the form awa. To recognize the second substring, we replicate the states of the first part (with new names), with q_3 as the beginning of the second part. Since the complete string can be broken into its constituent parts wherever aa occurs, we let the first occurrence of two consecutive a's be the trigger that gets the automaton into its second part. We can do this by making $\delta(q_3, a) = q_4$. The complete solution is in Figure 2.7. This dfa accepts L^2, which is therefore regular.

Figure 2.7



The last example suggests the conjecture that if a language L is regular, so are $L^2, L^3, \ldots$. We will see later that this is indeed correct.

EXERCISES

1. Which of the strings 0001, 01001, 0000110 are accepted by the dfa in Figure 2.1?

2. For $\Sigma = \{a, b\}$, construct dfa's that accept the sets consisting of

   (a) all strings with exactly one a,

   (b) all strings with at least one a,

   (c) all strings with no more than three a's,

   (d) all strings with at least one a and exactly two b's.

   (e) all the strings with exactly two a's and more than two b's.

3. Show that if we change Figure 2.6, making q_3 a nonfinal state and making q_0, q_1, q_2 final states, the resulting dfa accepts $\overline{L}$.

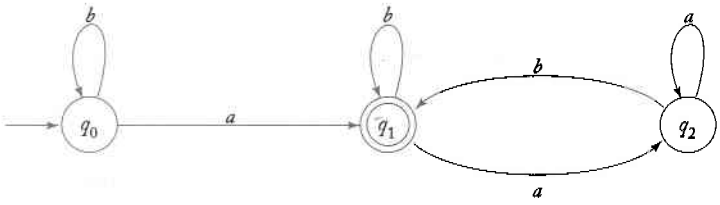
4. Generalize the observation in the previous exercise. Specifically, show that if $M = (Q, \Sigma, \delta, q_0, F)$ and $\widehat{M} = (Q, \Sigma, \delta, q_0, Q - F)$ are two dfa's then $\overline{L(M)} = L\left(\widehat{M}\right)$.

5. Give dfa's for the languages

   (a) $L = \{ab^5wb^4 : w \in \{a, b\}^*\}$

   (b) $L = \{w_1abw_2 : w_1 \in \{a, b\}^*, w_2 \in \{a, b\}^*\}$

6. Give a set notation description of the language accepted by the automaton depicted in the following diagram. Can you think of a simple verbal characterization of the language?



7. Find dfa's for the following languages on $\Sigma = \{a, b\}$.

   (a) $L = \{w : |w| \bmod 3 = 0\}$

   (b) $L = \{w : |w| \bmod 5 \neq 0\}$

   (c) $L = \{w : n_a(w) \bmod 3 > 1\}$

   (d) $L = \{w : n_a(w) \bmod 3 > n_b(w) \bmod 3\}$

(e) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 > 0\}$ Note that $-1 \bmod 3 = 2$.

(f) $L = \{w : |n_a(w) - n_b(w)| \bmod 3 < 2\}$

★8. A run in a string is a substring of length at least two, as long as possible and consisting entirely of the same symbol. For instance, the string *abbbaab* contains a run of b's of length three and a run of a's of length two. Find dfa's for the following languages on $\{a, b\}$.

(a) $L = \{w : w$ contains no runs of length less than four$\}$

(b) $L = \{w :$ every run of a's has length either two or three$\}$

(c) $L = \{w :$ there are at most two runs of a's of length three$\}$

(d) $L = \{w :$ there are exactly two runs of a's of length 3$\}$

9. Consider the set of strings on $\{0, 1\}$ defined by the requirements below. For each construct an accepting dfa.

(a) Every 00 is followed immediately by a 1. For example, the strings 101, 0010, 0010011001 are in the language, but 0001 and 00100 are not.

(b) all strings containing 00 but not 000.

(c) The leftmost symbol differs from the rightmost one.

(d) Every substring of four symbols has at most two 0's. For example, 001110 and 011001 are in the language, but 10010 is not since one of its substrings, 0010, contains three zeros.

(e) All strings of length five or more in which the fourth symbol from the right end is different from the leftmost symbol.

(f) All strings in which the leftmost two symbols and the rightmost two symbols are identical.

★10. Construct a dfa that accepts strings on $\{0, 1\}$ if and only if the value of the string, interpreted as a binary representation of an integer, is zero modulo five. For example, 0101 and 1111, representing the integers 5 and 15, respectively, are to be accepted.

11. Show that the language $L = \{vwv : v, w \in \{a, b\}^*, |v| = 2\}$ is regular.

12. Show that $L = \{a^n : n \geq 4\}$ is regular.

13. Show that the language $L = \{a^n : n \geq 0, n \neq 4\}$ is regular.

14. Show that the language $L = \{a^n : n = i + jk, i, k$ fixed, $j = 0, 1, 2, ...\}$ is regular.

15. Show that the set of all real numbers in C is a regular language.

16. Show that if L is regular, so is $L - \{\lambda\}$.

17. Use (2.1) and (2.2) to show that

$$\delta^*(q, wv) = \delta^*(\delta^*(q, w), v)$$

for all $w, v \in \Sigma^*$.

18. Let L be the language accepted by the automaton in Figure 2.2. Find a dfa that accepts L^2.

19. Let L be the language accepted by the automaton in Figure 2.2. Find a dfa for the language $L^2 - L$.

20. Let L be the language in Example 2.5. Show that L^* is regular.

21. Let G_M be the transition graph for some dfa M. Prove the following.

(a) If $L(M)$ is infinite, then G_M must have at least one cycle for which there is a path from the initial vertex to some vertex in the cycle and a path from some vertex in the cycle to some final vertex.

(b) If $L(M)$ is finite, then no such cycle exists.

22. Let us define an operation *truncate*, which removes the rightmost symbol from any string. For example, *truncate* $(aa\bar{a}ba)$ is $aaab$. The operation can be extended to languages by

$$truncate(L) = \{truncate(w) : w \in L\}.$$

Show how, given a dfa for any regular language L, one can construct a dfa for *truncate* (L). From this, prove that if L is a regular language not containing λ, then *truncate* (L) is also regular.

23. Let $x = a_0a_1 \cdots a_n, y = b_0b_1 \cdots b_n, z = c_0c_1 \cdots c_n$ be binary numbers as defined in Example 1.17. Show that the set of strings of triplets

$$\begin{pmatrix} a_0 \\ b_0 \\ c_0 \end{pmatrix} \begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \cdots \begin{pmatrix} a_n \\ b_n \\ c_n \end{pmatrix},$$

where the a_i, b_i, c_i are such that $x + y = z$ is a regular language.

24. While the language accepted by a given dfa is unique, there are normally many dfa's that accept a language. Find a dfa with exactly six states that accepts the same language as the dfa in Figure 2.4.

2.2 Nondeterministic Finite Accepters

Finite accepters are more complicated if we allow them to act nondeterministically. Nondeterminism is a powerful, but at first sight, unusual idea. We normally think of computers as completely deterministic, and the element of choice seems out of place. Nevertheless, nondeterminism is a useful notion, as we shall see as we proceed.

Definition of a Nondeterministic Accepter

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.

Definition 2.4

A **nondeterministic finite accepter** or **nfa** is defined by the quintuple

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

where Q, Σ, q_0, F are defined as for deterministic finite accepters, but

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q.$$

Note that there are three major differences between this definition and the definition of a dfa. In a nondeterministic accepter, the range of δ is in the powerset 2^Q, so that its value is not a single element of Q, but a subset of it. This subset defines the set of all possible states that can be reached by the transition. If, for instance, the current state is q_1, the symbol a is read, and

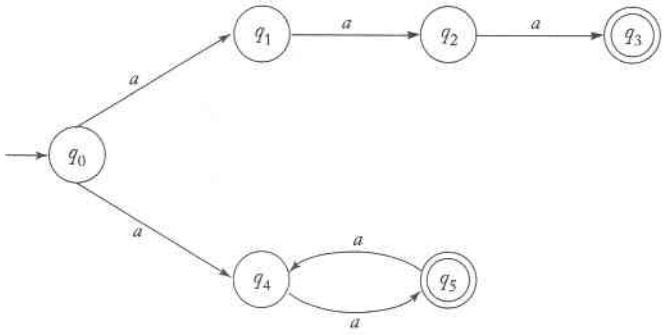
$$\delta(q_1, a) = \{q_0, q_2\},$$

then either q_0 or q_2 could be the next state of the nfa. Also, we allow λ as the second argument of δ. This means that the nfa can make a transition without consuming an input symbol. Although we still assume that the input mechanism can only travel to the right, it is possible that it is stationary on some moves. Finally, in an nfa, the set $\delta(q_i, a)$ may be empty, meaning that there is no transition defined for this specific situation.

Like dfa's, nondeterministic accepters can be represented by transition graphs. The vertices are determined by Q, while an edge (q_i, q_j) with label a is in the graph if and only if $\delta(q_i, a)$ contains q_j. Note that since a may be the empty string, there can be some edges labeled λ.

A string is accepted by an nfa if there is some sequence of possible moves that will put the machine in a final state at the end of the string. A string is rejected (that is, not accepted) only if there is no possible sequence of moves by which a final state can be reached. Nondeterminism can therefore be viewed as involving "intuitive" insight by which the best move can be chosen at every state (assuming that the nfa wants to accept every string).

Figure 2.8



Example 2.7 Consider the transition graph in Figure 2.8. It describes a nondeterministic accepter since there are two transitions labeled a out of q_0. ■

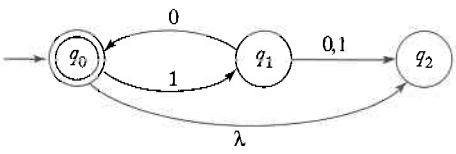
Example 2.8 A nondeterministic automaton is shown in Figure 2.9. It is nondeterministic not only because several edges with the same label originate from one vertex, but also because it has a λ-transition. Some transition, such as $\delta(q_2, 0)$ are unspecified in the graph. This is to be interpreted as a transition to the empty set, that is, $\delta(q_2, 0) = \varnothing$. The automaton accepts strings λ, 1010, and 101010, but not 110 and 10100. Note that for 10 there are two alternative walks, one leading to q_0, the other to q_2. Even though q_2 is not a final state, the string is accepted because one walk leads to a final state.

Again, the transition function can be extended so its second argument is a string. We require of the extended transition function δ^* that if

$$\delta^*(q_i, w) = Q_j,$$

then Q_j is the set of all possible states the automaton may be in, having started in state q_i and having read w. A recursive definition of δ^*, analogous to (2.1) and (2.2), is possible, but not particularly enlightening. A more easily appreciated definition can be made through transition graphs. ■

Figure 2.9



Definition 2.5

For an nfa, the extended transition function is defined so that $\delta^*(q_i, w)$ contains q_j if and only if there is a walk in the transition graph from q_i to q_j labeled w. This holds for all $q_i, q_j \in Q$ and $w \in \Sigma^*$.

Example 2.9

Figure 2.10 represents an nfa. It has several λ-transitions and some undefined transitions such as $\delta(q_2, a)$.

Suppose we want to find $\delta^*(q_1, a)$ and $\delta^*(q_2, \lambda)$. There is a walk labeled a involving two λ-transitions from q_1 to itself. By using some of the λ-edges twice, we see that there are also walks involving λ-transitions to q_0 and q_2. Thus

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}.$$

Since there is a λ-edge between q_2 and q_0, we have immediately that $\delta^*(q_2, \lambda)$ contains q_0. Also, since any state can be reached from itself by making no move, and consequently using no input symbol, $\delta^*(q_2, \lambda)$ also contains q_2.

Therefore

$$\delta^*(q_2, \lambda) = \{q_0, q_2\}.$$

Using as many λ-transitions as needed, you can also check that

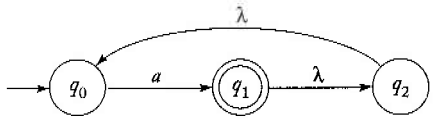
$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}.$$

■

The definition of δ^* through labeled walks is somewhat informal, so it is useful to look at it a little more closely. Definition 2.5 is proper, since between any vertices v_i and v_j there is either a walk labeled w or there is not, indicating that δ^* is completely defined. What is perhaps a little harder to see is that this definition can always be used to find $\delta^*(q_i, w)$.

In Section 1.1, we described an algorithm for finding all simple paths between two vertices. We cannot use this algorithm directly since, as Example 2.9 shows, a labeled walk is not always a simple path. We can modify the simple path algorithm, removing the restriction that no vertex or edge

Figure 2.10



can be repeated. The new algorithm will now generate successively all walks of length one, length two, length three, and so on.

There is still a difficulty. Given a w, how long can a walk labeled w be? This is not immediately obvious. In Example 2.9, the walk labeled a between q_1 and q_2 has length four. The problem is caused by the λ-transitions, which lengthen the walk but do not contribute to the label. The situation is saved by this observation: If between two vertices v_i and v_j there is any walk labeled w, then there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)|w|$, where Λ is the number of λ-edges in the graph. The argument for this is: While λ-edges may be repeated, there is always a walk in which every repeated λ-edge is separated by an edge labeled with a nonempty symbol. Otherwise, the walk contains a cycle labeled λ, which can be replaced by a simple path without changing the label of the walk. We leave a formal proof of this claim as an exercise.

With this observation, we have a method for computing $\delta^*(q_i, w)$. We evaluate all walks of length at most $\Lambda + (1 + \Lambda)|w|$ originating at v_i. We select from them those that are labeled w. The terminating vertices of the selected walks are the elements of the set $\delta^*(q_i, w)$.

As we have remarked, it is possible to define δ^* in a recursive fashion as was done for the deterministic case. The result is unfortunately not very transparent, and arguments with the extended transition function defined this way are hard to follow. We prefer to use the more intuitive and more manageable alternative in Definition 2.5.

As for dfa's, the language accepted by an nfa is defined formally by the extended transition function.

Definition 2.6

The language L accepted by an nfa $M = (Q, \Sigma, \delta, q_0, F)$ is defined as the set of all strings accepted in the above sense. Formally,

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \varnothing\}.$$

In words, the language consists of all strings w for which there is a walk labeled w from the initial vertex of the transition graph to some final vertex.

Example 2.10 What is the language accepted by the automaton in Figure 2.9? It is easy to see from the graph that the only way the nfa can stop in a final state is if the input is either a repetition of the string 10 or the empty string. Therefore the automaton accepts the language $L = \{(10)^n : n \geq 0\}$.

What happens when this automaton is presented with the string $w = 110$? After reading the prefix 11, the automaton finds itself in state q_2, with the transition $\delta(q_2, 0)$ undefined. We call such a situation a **dead configuration,** and we can visualize it as the automaton simply stopping without further action. But we must always keep in mind that such visualizations are imprecise and carry with them some danger of misinterpretation. What we can say precisely is that

$$\delta^*(q_0, 110) = \varnothing.$$

Thus, no final state can be reached by processing $w = 110$, and hence the string is not accepted. ■

Why Nondeterminism?

In reasoning about nondeterministic machines, we should be quite cautious in using intuitive notions. Intuition can easily lead us astray, and we must be able to give precise arguments to substantiate our conclusions. Nondeterminism is a difficult concept. Digital computers are completely deterministic; their state at any time is uniquely predictable from the input and the initial state. Thus it is natural to ask why we study nondeterministic machines at all. We are trying to model real systems, so why include such nonmechanical features as choice? We can answer this question in various ways.

Many deterministic algorithms require that one make a choice at some stage. A typical example is a game-playing program. Frequently, the best move is not known, but can be found using an exhaustive search with backtracking. When several alternatives are possible, we choose one and follow it until it becomes clear whether or not it was best. If not, we retreat to the last decision point and explore the other choices. A nondeterministic algorithm that can make the best choice would be able to solve the problem without backtracking, but a deterministic one can simulate nondeterminism with some extra work. For this reason, nondeterministic machines can serve as models of search-and-backtrack algorithms.

Nondeterminism is sometimes helpful in solving problems easily. Look at the nfa in Figure 2.8. It is clear that there is a choice to be made. The first alternative leads to the acceptance of the string a^3, while the second accepts all strings with an even number of a's. The language accepted by the nfa is $\{a^3\} \cup \{a^{2n} : n \geq 1\}$. While it is possible to find a dfa for this language, the nondeterminism is quite natural. The language is the union of two quite different sets, and the nondeterminism lets us decide at the outset which case we want. The deterministic solution is not as obviously

related to the definition. As we go on, we will see other and more convincing examples of the usefulness of nondeterminism.

In the same vein, nondeterminism is an effective mechanism for describing some complicated languages concisely. Notice that the definition of a grammar involves a nondeterministic element. In

$$S \rightarrow aSb|\lambda$$

we can at any point choose either the first or the second production. This lets us specify many different strings using only two rules.

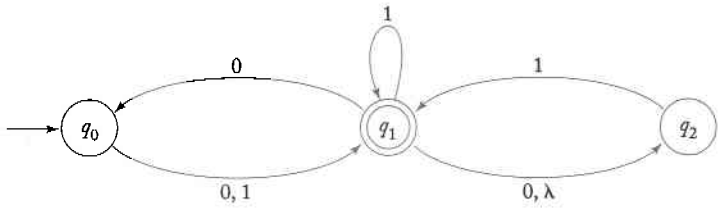
Finally, there is a technical reason for introducing nondeterminism. As we will see, certain results are more easily established for nfa's than for dfa's. Our next major result indicates that there is no essential difference between these two types of automata. Consequently, allowing nondeterminism often simplifies formal arguments without affecting the generality of the conclusion.

EXERCISES

1. Prove in detail the claim made in the previous section that if in a transition graph there is a walk labeled w, there must be some walk labeled w of length no more than $\Lambda + (1 + \Lambda)\,|w|$.
2. Find a dfa that accepts the language defined by the nfa in Figure 2.8.
3. In Figure 2.9, find $\delta^*(q_0, 1011)$ and $\delta^*(q_1, 01)$.
4. In Figure 2.10, find $\delta^*(q_0, a)$ and $\delta^*(q_1, \lambda)$.
5. For the nfa in Figure 2.9, find $\delta^*(q_0, 1010)$ and $\delta^*(q_1, 00)$.
6. Design an nfa with no more than five states for the set $\{abab^n : n \geq 0\} \cup \{aba^n : n \geq 0\}$.
7. Construct an nfa with three states that accepts the language $\{ab, abc\}^*$.
8. Do you think Exercise 7 can be solved with fewer than three states?
9. (a) Find an nfa with three states that accepts the language

$$L = \{a^n : n \geq 1\} \cup \left\{b^m a^k : m \geq 0, k \geq 0\right\}.$$

   (b) Do you think the language in part (a) can be accepted by an nfa with fewer than three states?
10. Find an nfa with four states for $L = \{a^n : n \geq 0\} \cup \{b^n a : n \geq 1\}$.
11. Which of the strings 00, 01001, 10010, 000, 0000 are accepted by the following nfa?



12. What is the complement of the language accepted by the nfa in Figure 2.10?

13. Let L be the language accepted by the nfa in Figure 2.8. Find an nfa that accepts $L \cup \{a^5\}$.

14. Give a simple description of the language in Exercise 12.

15. Find an nfa that accepts $\{a\}^*$ and is such that if in its transition graph a single edge is removed (without any other changes), the resulting automaton accepts $\{a\}$.

16. Can Exercise 15 be solved using a dfa? If so, give the solution; if not, give convincing arguments for your conclusion.

17. Consider the following modification of Definition 2.6. An nfa with multiple initial states is defined by the quintuple

$$M = (Q, \Sigma, \delta, Q_0, F),$$

where $Q_0 \subseteq Q$ is a set of possible initial states. The language accepted by such an automaton is defined as

$$L(M) = \{w : \delta^*(q_0, w) \text{ contains } q_f, \text{ for any } q_0 \in Q_0, q_f \in F\}.$$

Show that for every nfa with multiple initial states there exists an nfa with a single initial state that accepts the same language.

18. Suppose that in Exercise 17 we made the restriction $Q_0 \cap F = \varnothing$. Would this affect the conclusion?

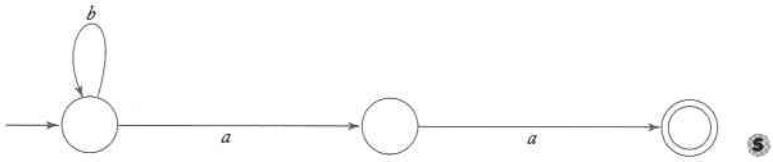
19. Use Definition 2.5 to show that for any nfa

$$\delta^*(q, wv) = \bigcup_{p \in \delta^*(q,w)} \delta^*(p, v),$$

for all $q \in Q$ and all $w, v \in \Sigma^*$.

20. An nfa in which (a) there are no λ-transitions, and (b) for all $q \in Q$ and all $a \in \Sigma$, $\delta(q, a)$ contains at most one element, is sometimes called an **incomplete** dfa. This is reasonable since the conditions make it such that there is never any choice of moves.

For $\Sigma = \{a, b\}$, convert the incomplete dfa below into a standard dfa.



2.3 Equivalence of Deterministic and Nondeterministic Finite Accepters

We now come to a fundamental question. In what sense are dfa's and nfa's different? Obviously, there is a difference in their definition, but this does not imply that there is any essential distinction between them. To explore this question, we introduce the notion of equivalence between automata.

Definition 2.7

Two finite accepters M_1 and M_2 are said to be equivalent if

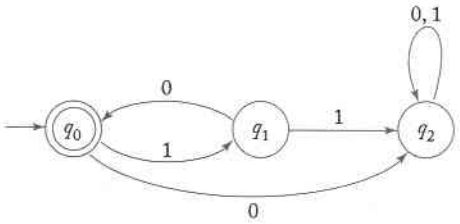
$$L(M_1) = L(M_2),$$

that is, if they both accept the same language.

As mentioned, there are generally many accepters for a given language, so any dfa or nfa has many equivalent accepters.

Example 2.11 The dfa shown in Figure 2.11 is equivalent to the nfa in Figure 2.9 since they both accept the language $\{(10)^n : n \geq 0\}$.

Figure 2.11



When we compare different classes of automata, the question invariably arises whether one class is more powerful than the other. By more powerful we mean that an automaton of one kind can achieve something that cannot be done by any automaton of the other kind. Let us look at this question for finite accepters. Since a dfa is in essence a restricted kind of nfa, it is clear that any language that is accepted by a dfa is also accepted by some nfa. But the converse is not so obvious. We have added nondeterminism, so it is at least conceivable that there is a language accepted by some nfa for which we cannot find a dfa. But it turns out that this is not so. The classes of dfa's and nfa's are equally powerful: For every language accepted by some nfa there is a dfa that accepts the same language.

This result is not obvious and certainly has to be demonstrated. The argument, like most arguments in this book, will be constructive. This means that we can actually give a way of converting any nfa into an equivalent dfa. The construction is not hard to understand; once the principle is clear it becomes the starting point for a rigorous argument. The rationale for the construction is the following. After an nfa has read a string w, we may not know exactly what state it will be in, but we can say that it must be in one state of a set of possible states, say $\{q_i, q_j, ..., q_k\}$. An equivalent dfa after reading the same string must be in some definite state. How can we make these two situations correspond? The answer is a nice trick: label the states of the dfa with a set of states in such a way that, after reading w, the equivalent dfa will be in a single state labeled $\{q_i, q_j, ..., q_k\}$. Since for a set of $|Q|$ states there are exactly $2^{|Q|}$ subsets, the corresponding dfa will have a finite number of states.

Most of the work in this suggested construction lies in the analysis of the nfa to get the correspondence between possible states and inputs. Before getting to the formal description of this, let us illustrate it with a simple example.

Example 2.12 Convert the nfa in Figure 2.12 to an equivalent dfa. The nfa starts in state q_0, so the initial state of the dfa will be labeled $\{q_0\}$. After reading an a, the nfa can be in state q_1 or, by making a λ-transition, in state q_2. Therefore the corresponding dfa must have a state labeled $\{q_1, q_2\}$ and a transition

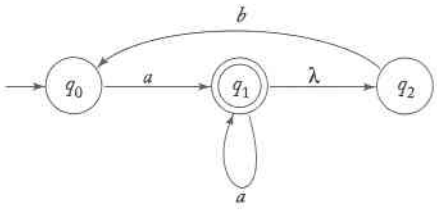
$$\delta(\{q_0\}, a) = \{q_1, q_2\}.$$

In state q_0, the nfa has no specified transition when the input is b, therefore

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

A state labeled $\varnothing$ represents an impossible move for the nfa and, therefore, means nonacceptance of the string. Consequently, this state in the dfa must be a nonfinal trap state.

Figure 2.12



We have now introduced into the dfa the state $\{q_1, q_2\}$, so we need to find the transitions out of this state. Remember that this state of the dfa corresponds to two possible states of the nfa, so we must refer back to the nfa. If the nfa is in state q_1 and reads an a, it can go to q_1. Furthermore, from q_1 the nfa can make a λ-transition to q_2. If, for the same input, the nfa is in state q_2, then there is no specified transition. Therefore

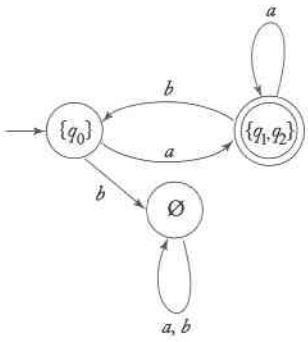
$$\delta(\{q_1, q_2\}, a) = \{q_1, q_2\}.$$

Similarly,

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

At this point, every state has all transitions defined. The result, shown in Figure 2.13, is a dfa, equivalent to the nfa with which we started. The nfa in Figure 2.12 accepts any string for which $\delta^*(q_0, w)$ contains q_1. For the corresponding dfa to accept every such w, any state whose label includes q_1 must be made a final state.

Figure 2.13



Theorem 2.2 Let L be the language accepted by a nondeterministic finite accepter $M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Then there exists a deterministic finite accepter $M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that

$$L = L(M_D).$$

Proof: Given M_N, we use the procedure nfa_to_dfa below to construct the transition graph G_D for M_D. To understand the construction, remember that G_D has to have certain properties. Every vertex must have exactly $|\Sigma|$ outgoing edges, each labeled with a different element of Σ. During the construction, some of the edges may be missing, but the procedure continues until they are all there.

procedure: nfa_to_dfa

1. Create a graph G_D with vertex $\{q_0\}$. Identify this vertex as the initial vertex.

2. Repeat the following steps until no more edges are missing.

   Take any vertex $\{q_i, q_j, ..., q_k\}$ of G_D that has no outgoing edge for some $a \in \Sigma$.

   Compute $\delta^*(q_i, a), \delta^*(q_j, a) ..., \delta^*(q_k, a)$.

   Then form the union of all these δ^*, yielding the set $\{q_l, q_m, ..., q_n\}$.

   Create a vertex for G_D labeled $\{q_l, q_m, ..., q_n\}$ if it does not already exist.

   Add to G_D an edge from $\{q_i, q_j, ..., q_k\}$ to $\{q_l, q_m, ..., q_n\}$ and label it with a.

3. Every state of G_D whose label contains any $q_f \in F_N$ is identified as a final vertex.

4. If M_N accepts λ, the vertex $\{q_0\}$ in G_D is also made a final vertex.

It is clear that this procedure always terminates. Each pass through the loop in Step 2 adds an edge to G_D. But G_D has at most $2^{|Q_N|}|\Sigma|$ edges, so that the loop eventually stops. To show that the construction also gives the correct answer, we argue by induction on the length of the input string.

Assume that for every v of length less than or equal to n, the presence in G_N of a walk labeled v from q_0 to q_i implies that in G_D there is a walk labeled v from $\{q_0\}$ to a state $Q_i = \{..., q_i, ...\}$. Consider now any $w = va$ and look at a walk in G_N labeled w from q_0 to q_l. There must then be a walk labeled v from q_0 to q_i and an edge (or a sequence of edges) labeled a from q_i to q_l. By the inductive assumption, in G_D there will be a walk labeled v from $\{q_0\}$ to Q_i. But by construction, there will be an edge from Q_i to some state whose label contains q_l. Thus the inductive assumption

holds for all strings of length $n+1$. As it is obviously true for $n = 1$, it is true for all n. The result then is that whenever $\delta_N^*(q_0, w)$ contains a final state q_f, so does the label of $\delta_D^*(q_0, w)$. To complete the proof, we reverse the argument to show that if the label of $\delta_D^*(q_0, w)$ contains q_f, so must $\delta_N^*(q_0, w)$. ■

The arguments in this proof, although correct, are admittedly somewhat terse, showing only the major steps. We will follow this practice in the rest of the book, emphasizing the basic ideas in a proof and omitting minor details, which you may want to fill in yourself.

The construction in the above proof is tedious but important. Let us do another example to make sure we understand all the steps.

Example 2.13 Convert the nfa in Figure 2.14 into an equivalent deterministic machine. Since $\delta_N(q_0, 0) = \{q_0, q_1\}$, we introduce the state $\{q_0, q_1\}$ in G_D and add an edge labeled 0 between $\{q_0\}$ and $\{q_0, q_1\}$. In the same way, considering $\delta_N(q_0, 1) = \{q_1\}$ gives us the new state $\{q_1\}$ and an edge labeled 1 between it and $\{q_0\}$.

There are now a number of missing edges, so we continue, using the construction of Theorem 2.2. With $a = 0$, $i = 0$, $j = 1$, we compute

$$\delta_N^*(q_0, 0) \cup \delta_N^*(q_1, 0) = \{q_0, q_1, q_2\}.$$

This gives us the new state $\{q_0, q_1, q_2\}$ and the transition

$$\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1, q_2\}.$$

Then, using $a = 1$, $i = 0$, $j = 1$, $k = 2$,

$$\delta_N^*(q_0, 1) \cup \delta_N^*(q_1, 1) \cup \delta_N^*(q_2, 1) = \{q_1, q_2\}$$

makes it necessary to introduce yet another state $\{q_1, q_2\}$. At this point, we have the partially constructed automaton shown in Figure 2.15. Since there are still some missing edges, we continue until we obtain the complete solution in Figure 2.16. ■

Figure 2.14

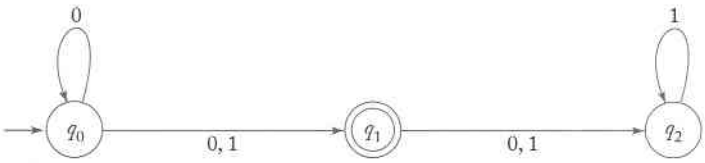


Figure 2.15

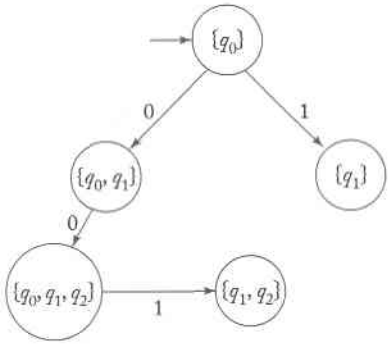
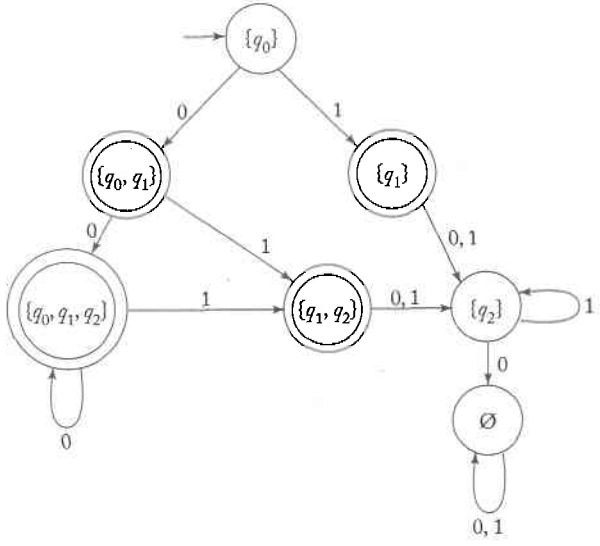


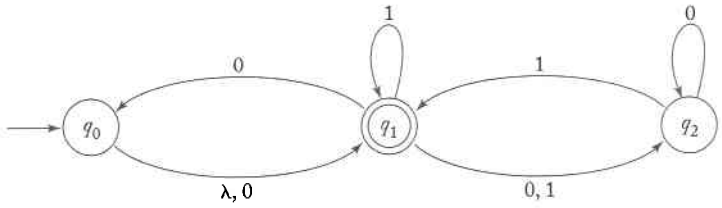
Figure 2.16



One important conclusion we can draw from Theorem 2.2 is that every language accepted by an nfa is regular.

EXERCISES

1. Use the construction of Theorem 2.2 to convert the nfa in Figure 2.10 to a dfa. Can you see a simpler answer more directly?
2. Convert the nfa in Exercise 11, Section 2.2 into an equivalent dfa. ●
3. Convert the following nfa into an equivalent dfa.



4. Carefully complete the arguments in the proof of Theorem 2.2. Show in detail that if the label of $\delta_D^*(q_0, w)$ contains q_f, then $\delta_N^*(q_0, w)$ also contains q_f.
5. Is it true that for any nfa $M = (Q, \Sigma, \delta, q_0, F)$ the complement of $L(M)$ is equal to the set $\{w \in \Sigma^* : \delta^*(q_0, w) \cap F = \varnothing\}$? If so, prove it. If not, give a counterexample.
6. Is it true that for every nfa $M = (Q, \Sigma, \delta, q_0, F)$ the complement of $L(M)$ is equal to the set $\{w \in \Sigma^* : \delta^*(q_0, w) \cap (Q - F) \neq \varnothing\}$? If so, prove it; if not, give a counterexample.
7. Prove that for every nfa with an arbitrary number of final states there is an equivalent nfa with only one final state. Can we make a similar claim for dfa's? ●
8. Find an nfa without λ-transitions and with a single final state that accepts the set $\{a\} \cup \{b^n : n \geq 1\}$. ●
9. ★ Let L be a regular language that does not contain λ. Show that there exists an nfa without λ-transitions and with a single final state that accepts L.
10. Define a dfa with multiple initial states in an analogous way to the corresponding nfa in Exercise 17, Section 2.2. Does there always exist an equivalent dfa with a single initial state?
11. Prove that all finite languages are regular. ●
12. Show that if L is regular, so is L^R.
13. Give a simple verbal description of the language accepted by the dfa in Figure 2.16. Use this to find another dfa, equivalent to the given one, but with fewer states.

★14. Let L be any language. Define $even\,(w)$ as the string obtained by extracting from w the letters in even-numbered positions; that is, if

$$w = a_1a_2a_3a_4\ldots,$$

then

$$even\,(w) = a_2a_4\ldots.$$

Corresponding to this, we can define a language

$$even\,(L) = \{even\,(w) : w \in L\}.$$

Prove that if L is regular, so is $even\,(L)$. ●

15. From a language L we create a new language $chop2\,(L)$ by removing the two leftmost symbols of every string in L. Specifically,

$$chop2\,(L) = \{w : vw \in L, \text{ with } |v| = 2\}.$$

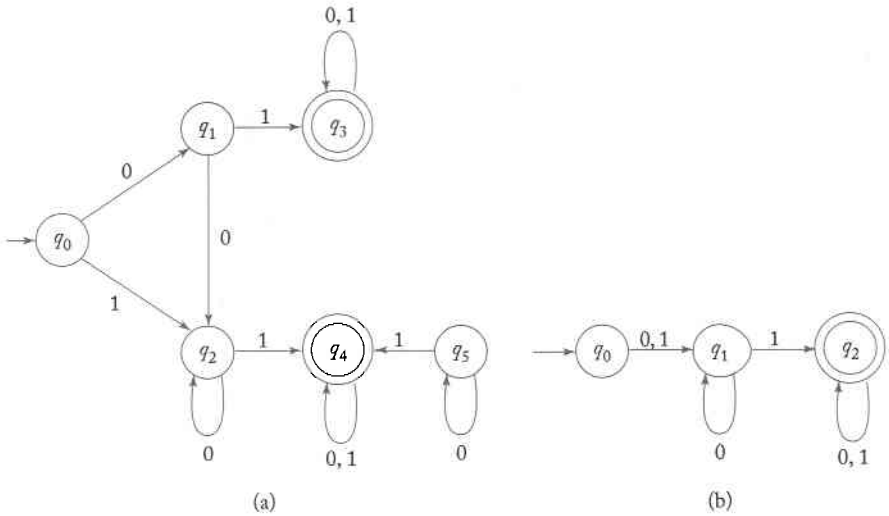
Show that if L is regular then $chop2\,(L)$ is also regular. ●

2.4 Reduction of the Number of States in Finite Automata*

Any dfa defines a unique language, but the converse is not true. For a given language, there are many dfa's that accept it. There may be a considerable difference in the number of states of such equivalent automata. In terms of the questions we have considered so far, all solutions are equally satisfactory, but if the results are to be applied in a practical setting, there may be reasons for preferring one over another.

Example 2.14 The two dfa's depicted in Figure 2.17(a) and 2.17(b) are equivalent, as a few test strings will quickly reveal. We notice some obviously unnecessary features of Figure 2.17(a). The state q_5 plays absolutely no role in the automaton since it can never be reached from the initial state q_0. Such a state is inaccessible, and it can be removed (along with all transitions relating to it) without affecting the language accepted by the automaton. But even after the removal of q_5, the first automaton has some redundant parts. The states reachable subsequent to the first move $\delta\,(q_0, 0)$ mirror those reachable from a first move $\delta\,(q_0, 1)$. The second automaton combines these two options. ■

Figure 2.17



From a strictly theoretical point of view, there is little reason for preferring the automaton in Figure 2.17(b) over that in Figure 2.17(a). However, in terms of simplicity, the second alternative is clearly preferable. Representation of an automaton for the purpose of computation requires space proportional to the number of states. For storage efficiency, it is desirable to reduce the number of states as far as possible. We now describe an algorithm that accomplishes this.

Definition 2.8

Two states p and q of a dfa are called **indistinguishable** if

$$\delta^*(p, w) \in F \text{ implies } \delta^*(q, w) \in F,$$

and

$$\delta^*(p, w) \notin F \text{ implies } \delta^*(q, w) \notin F,$$

for all $w \in \Sigma^*$. If, on the other hand, there exists some string $w \in \Sigma^*$ such that

$$\delta^*(p, w) \in F \text{ and } \delta^*(q, w) \notin F,$$

or vice versa, then the states p and q are said to be **distinguishable** by a string w.

Clearly, two states are either indistinguishable or distinguishable. Indistinguishability has the properties of an equivalence relations: if p and q are indistinguishable and if q and r are also indistinguishable, then so are p and r, and all three states are indistinguishable.

One method for reducing the states of a dfa is based on finding and combining indistinguishable states. We first describe a method for finding pairs of distinguishable states.

procedure: mark

1. Remove all inaccessible states. This can be done by enumerating all simple paths of the graph of the dfa starting at the initial state. Any state not part of some path is inaccessible.

2. Consider all pairs of states (p, q). If $p \in F$ and $q \notin F$ or vice versa, mark the pair (p, q) as distinguishable.

3. Repeat the following step until no previously unmarked pairs are marked.

   For all pairs (p, q) and all $a \in \Sigma$, compute $\delta(p, a) = p_a$ and $\delta(q, a) = q_a$. If the pair (p_a, q_a) is marked as distinguishable, mark (p, q) as distinguishable.

We claim that this procedure constitutes an algorithm for marking all distinguishable pairs.

Theorem 2.3 The procedure *mark*, applied to any dfa $M = (Q, \Sigma, \delta, q_0, F)$, terminates and determines all pairs of distinguishable states.

Proof: Obviously, the procedure terminates, since there are only a finite number of pairs that can be marked. It is also easy to see that the states of any pair so marked are distinguishable. The only claim that requires elaboration is that the procedure finds all distinguishable pairs.

Note first that states q_i and q_j are distinguishable with a string of length n, if and only if there are transitions

$$\delta(q_i, a) = q_k \tag{2.5}$$

and

$$\delta(q_j, a) = q_l, \tag{2.6}$$

for some $a \in \Sigma$, with q_k and q_l distinguishable by a string of length $n - 1$. We use this first to show that at the completion of the nth pass through the loop in step 3, all states distinguishable by strings of length n or less have been marked. In step 2, we mark all pairs indistinguishable by λ, so we have a basis with $n = 0$ for an induction. We now assume that the claim is true

for all $i = 0, 1, ..., n-1$. By this inductive assumption, at the beginning of the nth pass through the loop, all states distinguishable by strings of length up to $n-1$ have been marked. Because of (2.5) and (2.6) above, at the end of this pass, all states distinguishable by strings of length up to n will be marked. By induction then, we can claim that, for any n, at the completion of the nth pass, all pairs distinguishable by strings of length n or less have been marked.

To show that this procedure marks all distinguishable states, assume that the loop terminates after n passes. This means that during the nth pass no new states were marked. From (2.5) and (2.6), it then follows that there cannot be any states distinguishable by a string of length n, but not distinguishable by any shorter string. But if there are no states distinguishable only by strings of length n, there cannot be any states distinguishable only by strings of length $n+1$, and so on. As a consequence, when the loop terminates, all distinguishable pairs have been marked. ■

After the marking algorithm has been executed, we use the results to partition the state set Q of the dfa into disjoint subsets $\{q_i, q_j, ..., q_k\}$, $\{q_l, q_m, ..., q_n\}, ...$, such that any $q \in Q$ occurs in exactly one of these subsets, that elements in each subset are indistinguishable, and that any two elements from different subsets are distinguishable. Using the results sketched in Exercise 11 at the end of this section, it can be shown that such a partitioning can always be found. From these subsets we construct the minimal automaton by the next procedure.

procedure: reduce

Given a dfa $M = (Q, \Sigma, \delta, q_0, F)$, we construct a reduced dfa $\widehat{M} = \left(\widehat{Q}, \Sigma, \widehat{\delta}, \widehat{q}_0, \widehat{F}\right)$ as follows.

1. Use procedure *mark* to find all pairs of distinguishable states. Then from this, find the sets of all indistinguishable states, say $\{q_i, q_j, ..., q_k\}$, $\{q_l, q_m, ..., q_n\}$, etc., as described above.

2. For each set $\{q_i, q_j, ..., q_k\}$ of such indistinguishable states, create a state labeled $ij \cdots k$ for M.

3. For each transition rule of M of the form

$$\delta(q_r, a) = q_p,$$

find the sets to which q_r and q_p belong. If $q_r \in \{q_i, q_j, ..., q_k\}$ and $q_p \in \{q_l, q_m, ..., q_n\}$, add to $\widehat{\delta}$ a rule

$$\widehat{\delta}(ij \cdots k, a) = lm \cdots n.$$

4. The initial state $\widehat{q}_0$ is that state of $\widehat{M}$ whose label includes the 0.

5. $\widehat{F}$ is the set of all the states whose label contains i such that $q_i \in F$.

Example 2.15 Consider the automaton depicted in Figure 2.18.

In step 2, the procedure *mark* will identify distinguishable pairs (q_0, q_4), (q_1, q_4), (q_2, q_4), and (q_3, q_4). In some pass through the step 3 loop, the procedure computes

$$\delta(q_1, 1) = q_4$$

and

$$\delta(q_0, 1) = q_3.$$

Since (q_3, q_4) is a distinguishable pair, the pair (q_0, q_1) is also marked. Continuing this way, the marking algorithm eventually marks the pairs (q_0, q_1), (q_0, q_2), (q_0, q_3), (q_0, q_4), (q_1, q_4), (q_2, q_4) and (q_3, q_4) as distinguishable, leaving the indistinguishable pairs (q_1, q_2), (q_1, q_3) and (q_2, q_3). Therefore, the states q_1, q_2, q_3 are all indistinguishable, and all of the states have been partitioned into the sets $\{q_0\}$, $\{q_1, q_2, q_3\}$ and $\{q_4\}$. Applying steps 2 and 3 of the procedure *reduce* then yields the dfa in Figure 2.19. ■

Figure 2.18

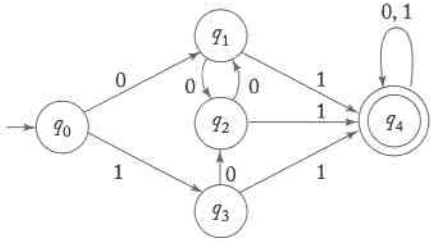
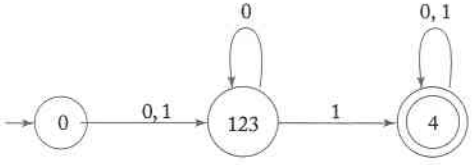


Figure 2.19



Theorem 2.4 Given any dfa M, application of the procedure *reduce* yields another dfa $\widehat{M}$ such that

$$L(M) = L\left(\widehat{M}\right).$$

Furthermore, $\widehat{M}$ is minimal in the sense that there is no other dfa with a smaller number of states which also accepts $L(M)$.

Proof: There are two parts. The first is to show that the dfa created by *reduce* is equivalent to the original dfa. This is relatively easy and we can use inductive arguments similar to those used in establishing the equivalence of dfa's and nfa's. All we have to do is to show that $\delta^*(q_i, w) = q_j$ if and only if the label of $\widehat{\delta}^*(q_i, w)$ is of the form $...j....$ We will leave this as an exercise.

The second part, to show that $\widehat{M}$ is minimal, is harder. Suppose $\widehat{M}$ has states $\{p_0, p_1, p_2, ..., p_m\}$, with p_0 the initial state. Assume that there is an equivalent dfa M_1, with transition function δ_1 and initial state q_0, equivalent to $\widehat{M}$, but with fewer states. Since there are no inaccessible states in $\widehat{M}$, there must be distinct strings $w_1, w_2, ..., w_m$ such that

$$\widehat{\delta}^*(p_0, w_i) = p_i, i = 1, 2, ..., m.$$

But since M_1 has fewer states than $\widehat{M}$, there must be at least two of these strings, say w_k and w_l, such that

$$\delta_1^*(q_0, w_k) = \delta_1^*(q_0, w_l).$$

Since p_k and p_l are distinguishable, there must be some string x such that $\widehat{\delta}^*(p_0, w_k x) = \widehat{\delta}^*(p_k, x)$ is a final state, and $\widehat{\delta}^*(q_0, w_l x) = \widehat{\delta}^*(p_l, x)$ is a nonfinal state (or vice versa). In other words, $w_k x$ is accepted by $\widehat{M}$ and $w_l x$ is not. But note that

$$\begin{aligned}\delta_1^*(q_0, w_k x) &= \delta_1^*(\delta_1^*(q_0, w_k), x) \\ &= \delta_1^*(\delta_1^*(q_0, w_l), x) \\ &= \delta_1^*(q_0, w_l x).\end{aligned}$$

Thus, M_1 either accepts both $w_k x$ and $w_l x$ or rejects both, contradicting the assumption that $\widehat{M}$ and M_1 are equivalent. This contradiction proves that M_1 cannot exist. ■

EXERCISES

1. Minimize the number of states in the dfa in Figure 2.16.

2. Find minimal dfa's for the languages below. In each case prove that the result is minimal.

   (a) $L = \{a^n b^m : n \geq 2, m \geq 1\}$

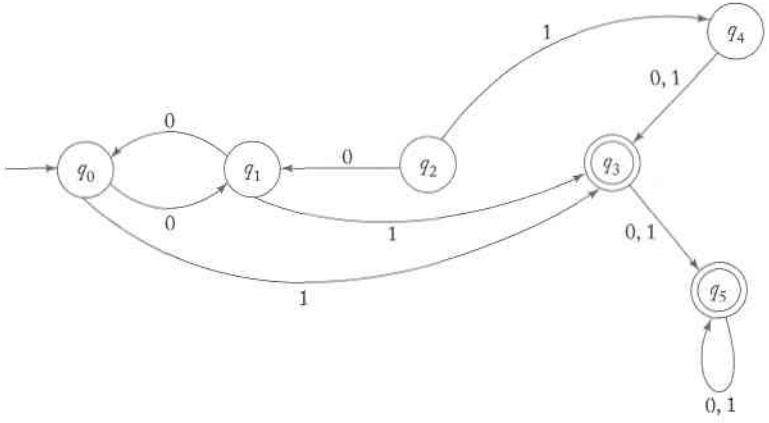
   (b) $L = \{a^n b : n \geq 0\} \cup \{b^n a : n \geq 1\}$

   (c) $L = \{a^n : n \geq 0, n \neq 3\}$ ●

   (d) $L = \{a^n : n \neq 2 \text{ and } n \neq 4\}$.

3. Show that the automaton generated by procedure *reduce* is deterministic.

4. Minimize the states in the dfa depicted in the following diagram.



●

5. Show that if L is a nonempty language such that any w in L has length at least n, then any dfa accepting L must have at least $n + 1$ states.

6. Prove or disprove the following conjecture. If $M = (Q, \Sigma, \delta, q_0, F)$ is a minimal dfa for a regular language L, then $\widehat{M} = (Q, \Sigma, \delta, q_0, Q - F)$ is a minimal dfa for $\overline{L}$. ●

7. Show that indistinguishability is an equivalence relation but that distinguishability is not.

8. Show the explicit steps of the suggested proof of the first part of Theorem 2.4, namely, that $\widehat{M}$ is equivalent to the original dfa.

★★ 9. Write a computer program that produces a minimal dfa for any given dfa.

10. Prove the following: If the states q_a and q_b are indistinguishable, and if q_a and q_c are distinguishable, then q_b and q_c must be distinguishable. ●

11. Consider the following process, to be done after the completion of the procedure *mark*. Start with some state, say, q_0. Put all states not marked distinguishable from q_0 into an equivalence set with q_0. Then take another state, not in the preceding equivalence set, and do the same thing. Repeat until there are no more states available. Then formalize this suggestion to make it an algorithm, and prove that this algorithm does indeed partition the original state set into equivalence sets.