Chapter 2

Finite Automata

This chapter introduces the class of languages known as "regular languages." These languages are exactly the ones that can be described by finite automata, which we sampled briefly in Section 1.1.1. After an extended example that will provide motivation for the study to follow, we define finite automata formally.

As was mentioned earlier, a finite automaton has a set of states, and its "control" moves from state to state in response to external "inputs." One of the crucial distinctions among classes of finite automata is whether that control is "deterministic," meaning that the automaton cannot be in more than one state at any one time, or "nondeterministic," meaning that it may be in several states at once. We shall discover that adding nondeterminism does not let us define any language that cannot be defined by a deterministic finite automaton, but there can be substantial efficiency in describing an application using a nondeterministic automaton. In effect, nondeterminism allows us to "program" solutions to problems using a higher-level language. The nondeterministic finite automaton is then "compiled," by an algorithm we shall learn in this chapter, into a deterministic automaton that can be "executed" on a conventional computer.

We conclude the chapter with a study of an extended nondeterministic automaton that has the additional choice of making a transition from one state to another spontaneously, i.e., on the empty string as "input." These automata also accept nothing but the regular languages. However, we shall find them quite important in Chapter 3, when we study regular expressions and their equivalence to automata.

The study of the regular languages continues in Chapter 3. There, we introduce another important way to describe regular languages: the algebraic notation known as regular expressions. After discussing regular expressions, and showing their equivalence to finite automata, we use both automata and regular expressions as tools in Chapter 4 to show certain important properties of the regular languages. Examples of such properties are the "closure" properties, which allow us to claim that one language is regular because one or more other languages are known to be regular, and "decision" properties. The latter are algorithms to answer questions about automata or regular expressions, e.g., whether two automata or expressions represent the same language.

2.1 An Informal Picture of Finite Automata

In this section, we shall study an extended example of a real-world problem whose solution uses finite automata in an important role. We investigate protocols that support "electronic money" — files that a customer can use to pay for goods on the internet, and that the seller can receive with assurance that the "money" is real. The seller must know that the file has not been forged, nor has it been copied and sent to the seller, while the customer retains a copy of the same file to spend again.

The nonforgeability of the file is something that must be assured by a bank and by a cryptography policy. That is, a third player, the bank, must issue and encrypt the "money" files, so that forgery is not a problem. However, the bank has a second important job: it must keep a database of all the valid money that it has issued, so that it can verify to a store that the file it has received represents real money and can be credited to the store's account. We shall not address the cryptographic aspects of the problem, nor shall we worry about how the bank can store and retrieve what could be billions of "electronic dollar bills." These problems are not likely to represent long-term impediments to the concept of electronic money, and examples of its small-scale use have existed since the late 1990's.

However, in order to use electronic money, protocols need to be devised to allow the manipulation of the money in a variety of ways that the users want. Because monetary systems always invite fraud, we must verify whatever policy we adopt regarding how money is used. That is, we need to prove the only things that can happen are things we intend to happen — things that do not allow an unscrupulous user to steal from others or to "manufacture" money. In the balance of this section, we shall introduce a very simple example of a (bad) electronic-money protocol, model it with finite automata, and show how constructions on automata can be used to verify protocols (or, in this case, to discover that the protocol has a bug).

2.1.1 The Ground Rules

There are three participants: the customer, the store, and the bank. We assume for simplicity that there is only one "money" file in existence. The customer may decide to transfer this money file to the store, which will then redeem the file from the bank (i.e., get the bank to issue a new money file belonging to the store rather than the customer) and ship goods to the customer. In addition, the customer has the option to cancel the file. That is, the customer may ask the bank to place the money back in the customer's account, making the money no longer spendable. Interaction among the three participants is thus limited to five events:

1. The customer may decide to *pay*. That is, the customer sends the money to the store.

2. The customer may decide to *cancel*. The money is sent to the bank with a message that the value of the money is to be added to the customer's bank account.

3. The store may *ship* goods to the customer.

4. The store may *redeem* the money. That is, the money is sent to the bank with a request that its value be given to the store.

5. The bank may *transfer* the money by creating a new, suitably encrypted money file and sending it to the store.

2.1.2 The Protocol

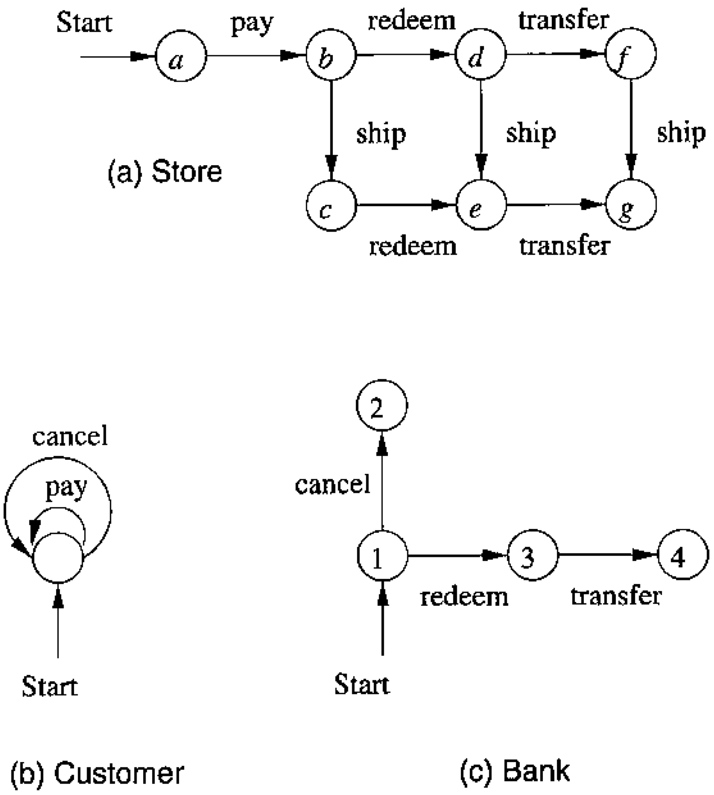
The three participants must design their behaviors carefully, or the wrong things may happen. In our example, we make the reasonable assumption that the customer cannot be relied upon to act responsibly. In particular, the customer may try to copy the money file, use it to pay several times, or both pay and cancel the money, thus getting the goods "for free."

The bank must behave responsibly, or it cannot be a bank. In particular, it must make sure that two stores cannot both redeem the same money file, and it must not allow money to be both canceled and redeemed. The store should be careful as well. In particular, it should not ship goods until it is sure it has been given valid money for the goods.

Protocols of this type can be represented as finite automata. Each state represents a situation that one of the participants could be in. That is, the state "remembers" that certain important events have happened and that others have not yet happened. Transitions between states occur when one of the five events described above occur. We shall think of these events as "external" to the automata representing the three participants, even though each participant is responsible for initiating one or more of the events. It turns out that what is important about the problem is what sequences of events can happen, not who is allowed to initiate them.

Figure 2.1 represents the three participants by automata. In that diagram, we show only the events that affect a participant. For example, the action *pay* affects only the customer and store. The bank does not know that the money has been sent by the customer to the store; it discovers that fact only when the store executes the action *redeem*.

Let us examine first the automaton (c) for the bank. The start state is state 1; it represents the situation where the bank has issued the money file in question but has not been requested either to redeem it or to cancel it. If a



(a) Store

(b) Customer

(c) Bank

Figure 2.1: Finite automata representing a customer, a store, and a bank

cancel request is sent to the bank by the customer, then the bank restores the money to the customer's account and enters state 2. The latter state represents the situation where the money has been cancelled. The bank, being responsible, will not leave state 2 once it is entered, since the bank must not allow the same money to be cancelled again or spent by the customer.[1]

Alternatively, when in state 1 the bank may receive a *redeem* request from the store. If so, it goes to state 3, and shortly sends the store a *transfer* message, with a new money file that now belongs to the store. After sending the transfer message, the bank goes to state 4. In that state, it will neither accept *cancel* or *redeem* requests nor will it perform any other actions regarding this particular money file.

Now, let us consider Fig. 2.1(a), the automaton representing the actions of the store. While the bank always does the right thing, the store's system has some defects. Imagine that the shipping and financial operations are done by

[1] You should remember that this entire discussion is about one single money file. The bank will in fact be running the same protocol with a large number of electronic pieces of money, but the workings of the protocol are the same for each of them, so we can discuss the problem as if there were only one piece of electronic money in existence.

separate processes, so there is the opportunity for the *ship* action to be done either before, after, or during the redemption of the electronic money. That policy allows the store to get into a situation where it has already shipped the goods and then finds out the money was bogus.

The store starts out in state a. When the customer orders the goods by performing the *pay* action, the store enters state b. In this state, the store begins both the shipping and redemption processes. If the goods are shipped first, then the store enters state c, where it must still redeem the money from the bank and receive the *transfer* of an equivalent money file from the bank. Alternatively, the store may send the *redeem* message first, entering state d. From state d, the store might next ship, entering state e, or it might next receive the transfer of money from the bank, entering state f. From state f, we expect that the store will eventually ship, putting the store in state g, where the transaction is complete and nothing more will happen. In state e, the store is waiting for the *transfer* from the bank. Unfortunately, the goods have already been shipped, and if the *transfer* never occurs, the store is out of luck.

Last, observe the automaton for the customer, Fig. 2.1(b). This automaton has only one state, reflecting the fact that the customer "can do anything." The customer can perform the *pay* and *cancel* actions any number of times, in any order, and stays in the lone state after each action.

2.1.3 Enabling the Automata to Ignore Actions

While the three automata of Fig. 2.1 reflect the behaviors of the three participants independently, there are certain transitions that are missing. For example, the store is not affected by a *cancel* message, so if the *cancel* action is performed by the customer, the store should remain in whatever state it is in. However, in the formal definition of a finite automaton, which we shall study in Section 2.2, whenever an input X is received by an automaton, the automaton must follow an arc labeled X from the state it is in to some new state. Thus, the automaton for the store needs an additional arc from each state to itself, labeled *cancel*. Then, whenever the *cancel* action is executed, the store automaton can make a "transition" on that input, with the effect that it stays in the same state it was in. Without these additional arcs, whenever the *cancel* action was executed the store automaton would "die"; that is, the automaton would be in no state at all, and further actions by that automaton would be impossible.

Another potential problem is that one of the participants may, intentionally or erroneously, send an unexpected message, and we do not want this action to cause one of the automata to die. For instance, suppose the customer decided to execute the *pay* action a second time, while the store was in state e. Since that state has no arc out with label *pay*, the store's automaton would die before it could receive the transfer from the bank. In summary, we must add to the automata of Fig. 2.1 loops on certain states, with labels for all those actions that must be ignored when in that state; the complete automata are shown in Fig. 2.2. To save space, we combine the labels onto one arc, rather than showing several arcs with the same heads and tails but different labels. The two kinds of actions that must be ignored are:

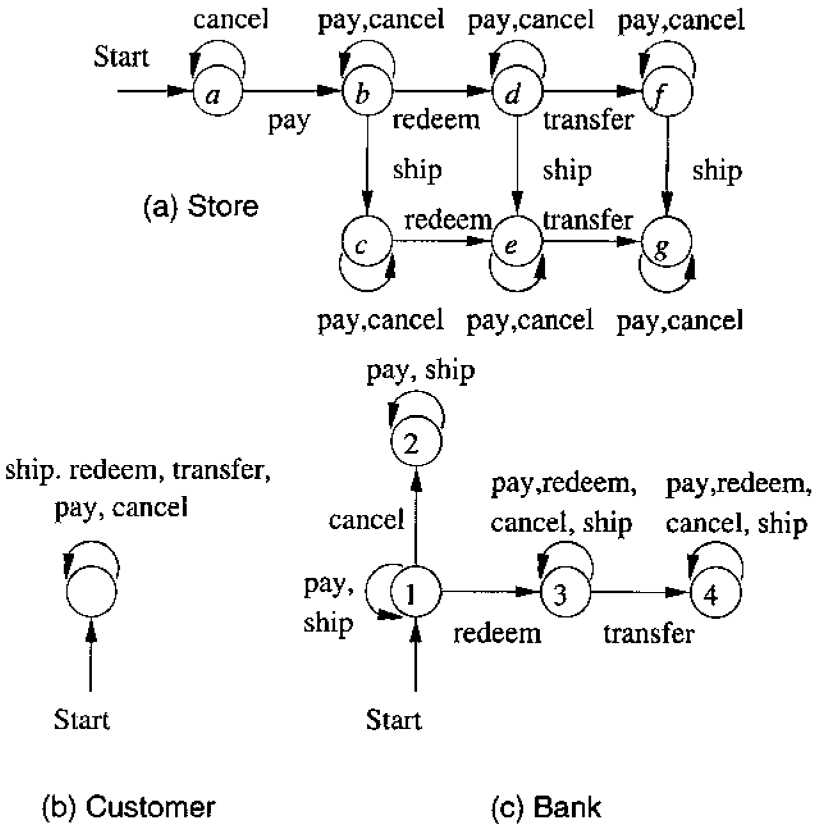


Figure 2.2: The complete sets of transitions for the three automata

1. *Actions that are irrelevant to the participant involved.* As we saw, the only irrelevant action for the store is *cancel*, so each of its seven states has a loop labeled *cancel*. For the bank, both *pay* and *ship* are irrelevant, so we have put at each of the bank's states an arc labeled *pay, ship*. For the customer, *ship*, *redeem* and *transfer* are all irrelevant, so we add arcs with these labels. In effect, it stays in its one state on any sequence of inputs, so the customer automaton has no effect on the operation of the overall system. Of course, the customer is still a participant, since it is the customer who initiates the *pay* and *cancel* actions. However, as we mentioned, the matter of who initiates actions has nothing to do with the behavior of the automata.

2. *Actions that must not be allowed to kill an automaton.* As mentioned, we must not allow the customer to kill the store's automaton by executing *pay*

again, so we have added loops with label *pay* to all but state a (where the *pay* action is expected and relevant). We have also added loops with labels *cancel* to states 3 and 4 of the bank, in order to prevent the customer from killing the bank's automaton by trying to cancel money that has already been redeemed. The bank properly ignores such a request. Likewise, states 3 and 4 have loops on *redeem*. The store should not try to redeem the same money twice, but if it does, the bank properly ignores the second request.

2.1.4 The Entire System as an Automaton

While we now have models for how the three participants behave, we do not yet have a representation for the interaction of the three participants. As mentioned, because the customer has no constraints on behavior, that automaton has only one state, and any sequence of events lets it stay in that state; i.e., it is not possible for the system as a whole to "die" because the customer automaton has no response to an action. However, both the store and bank behave in a complex way, and it is not immediately obvious in what combinations of states these two automata can be.

The normal way to explore the interaction of automata such as these is to construct the *product* automaton. That automaton's states represent a pair of states, one from the store and one from the bank. For instance, the state $(3, d)$ of the product automaton represents the situation where the bank is in state 3, and the store is in state d. Since the bank has four states and the store has seven, the product automaton has $4 \times 7 = 28$ states.

We show the product automaton in Fig. 2.3. For clarity, we have arranged the 28 states in an array. The row corresponds to the state of the bank and the column to the state of the store. To save space, we have also abbreviated the labels on the arcs, with P, S, C, R, and T standing for pay, ship, cancel, redeem, and transfer, respectively.

To construct the arcs of the product automaton, we need to run the bank and store automata "in parallel." Each of the two components of the product automaton independently makes transitions on the various inputs. However, it is important to notice that if an input action is received, and one of the two automata has no state to go to on that input, then the product automaton "dies"; it has no state to go to.

To make this rule for state transitions precise, suppose the product automaton is in state (i, x). That state corresponds to the situation where the bank is in state i and the store in state x. Let Z be one of the input actions. We look at the automaton for the bank, and see whether there is a transition out of state i with label Z. Suppose there is, and it leads to state j (which might be the same as i if the bank loops on input Z). Then, we look at the store and see if there is an arc labeled Z leading to some state y. If both j and y exist, then the product automaton has an arc from state (i, x) to state (j, y), labeled Z. If either of states j or y do not exist (because the bank or store has no arc

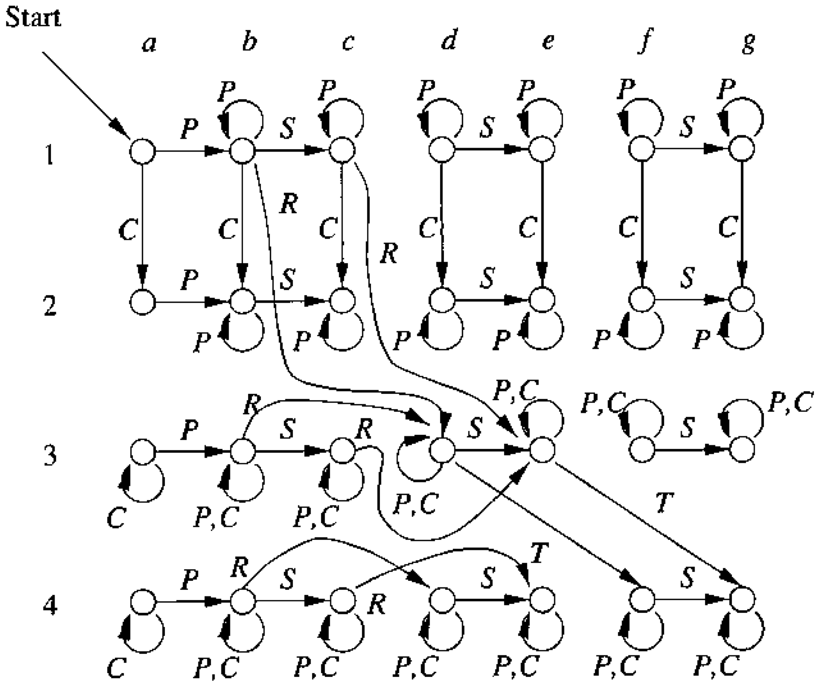


Figure 2.3: The product automaton for the store and bank

out of i or x, respectively, for input Z), then there is no arc out of (i, x) labeled Z.

We can now see how the arcs of Fig. 2.3 were selected. For instance, on input *pay*, the store goes from state a to b, but stays put if it is in any other state besides a. The bank stays in whatever state it is in when the input is *pay*, because that action is irrelevant to the bank. This observation explains the four arcs labeled P at the left ends of the four rows in Fig. 2.3, and the loops labeled P on other states.

For another example of how the arcs are selected, consider the input *redeem*. If the bank receives a *redeem* message when in state 1, it goes to state 3. If in states 3 or 4, it stays there, while in state 2 the bank automaton dies; i.e., it has nowhere to go. The store, on the other hand, can make transitions from state b to d or from c to e when the *redeem* input is received. In Fig. 2.3, we see six arcs labeled *redeem*, corresponding to the six combinations of three bank states and two store states that have outward-bound arcs labeled R. For example, in state $(1, b)$, the arc labeled R takes the automaton to state $(3, d)$, since *redeem* takes the bank from state 1 to 3 and the store from b to d. As another example, there is an arc labeled R from $(4, c)$ to $(4, e)$, since *redeem* takes the bank from state 4 back to state 4, while it takes the store from state c to state e.

2.1.5 Using the Product Automaton to Validate the Protocol

Figure 2.3 tells us some interesting things. For instance, of the 28 states, only ten of them can be reached from the start state, which is $(1, a)$ — the combination of the start states of the bank and store automata. Notice that states like $(2, e)$ and $(4, d)$ are not *accessible*, that is, there is no path to them from the start state. Inaccessible states need not be included in the automaton, and we did so in this example just to be systematic.

However, the real purpose of analyzing a protocol such as this one using automata is to ask and answer questions that mean "can the following type of error occur?" In the example at hand, we might ask whether it is possible that the store can ship goods and never get paid. That is, can the product automaton get into a state in which the store has shipped (that is, the state is in column c, e, or g), and yet no transition on input T was ever made or will be made?

For instance, in state $(3, e)$, the goods have shipped, but there will eventually be a transition on input T to state $(4, g)$. In terms of what the bank is doing, once it has gotten to state 3, it has received the *redeem* request and processed it. That means it must have been in state 1 before receiving the *redeem* and therefore the *cancel* message had not been received and will be ignored if received in the future. Thus, the bank will eventually perform the transfer of money to the store.

However, state $(2, c)$ is a problem. The state is accessible, but the only arc out leads back to that state. This state corresponds to a situation where the bank received a *cancel* message before a *redeem* message. However, the store received a *pay* message; i.e., the customer was being duplicitous and has both spent and canceled the same money. The store foolishly shipped before trying to redeem the money, and when the store does execute the *redeem* action, the bank will not even acknowledge the message, because it is in state 2, where it has canceled the money and will not process a *redeem* request.

2.2 Deterministic Finite Automata

Now it is time to present the formal notion of a finite automaton, so that we may start to make precise some of the informal arguments and descriptions that we saw in Sections 1.1.1 and 2.1. We begin by introducing the formalism of a deterministic finite automaton, one that is in a single state after reading any sequence of inputs. The term "deterministic" refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state. In contrast, "nondeterministic" finite automata, the subject of Section 2.3, can be in several states at once. The term "finite automaton" will refer to the deterministic variety, although we shall use "deterministic" or the abbreviation *DFA* normally, to remind the reader of which kind of automaton we are talking about.

2.2.1 Definition of a Deterministic Finite Automaton

A *deterministic finite automaton* consists of:

Capital Letter for Set
Small Letter for elements

1. A finite set of *states*, often denoted Q.
2. A finite set of *input symbols*, often denoted Σ.
3. A *transition function* that takes as arguments a state and an input symbol and returns a state. The transition function will commonly be denoted δ. In our informal graph representation of automata, δ was represented by arcs between states and the labels on the arcs. If q is a state, and a is an input symbol, then $\delta(q, a)$ is that state p such that there is an arc labeled a from q to p.[2]
4. A *start state*, one of the states in Q.
5. A set of *final* or *accepting* states F. The set F is a subset of Q.

A deterministic finite automaton will often be referred to by its acronym: *DFA*. The most succinct representation of a DFA is a listing of the five components above. In proofs we often talk about a DFA in "five-tuple" notation:

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

where A is the name of the DFA, Q is its set of states, Σ its input symbols, δ its transition function, q_0 its start state, and F its set of accepting states.

2.2.2 How a DFA Processes Strings

The first thing we need to understand about a DFA is how the DFA decides whether or not to "accept" a sequence of input symbols. The "language" of the DFA is the set of all strings that the DFA accepts. Suppose $a_1 a_2 \cdots a_n$ is a sequence of input symbols. We start out with the DFA in its start state, q_0. We consult the transition function δ, say $\delta(q_0, a_1) = q_1$ to find the state that the DFA A enters after processing the first input symbol a_1. We process the next input symbol, a_2, by evaluating $\delta(q_1, a_2)$; let us suppose this state is q_2. We continue in this manner, finding states $q_3, q_4, \ldots, q_n$ such that $\delta(q_{i-1}, a_i) = q_i$ for each i. If q_n is a member of F, then the input $a_1 a_2 \cdots a_n$ is accepted, and if not then it is "rejected."

Example 2.1: Let us formally specify a DFA that accepts all and only the strings of 0's and 1's that have the sequence 01 somewhere in the string. We can write this language L as:

Solution-> Figure 2.4, Page 48

$$\{w \mid w \text{ is of the form } x01y \text{ for some strings } x \text{ and } y \text{ consisting of 0's and 1's only}\}$$

[2] More accurately, the graph is a picture of some transition function δ, and the arcs of the graph are constructed to reflect the transitions specified by δ.

Another equivalent description, using parameters x and y to the left of the vertical bar, is:

$$\{x01y \mid x \text{ and } y \text{ are any strings of 0's and 1's}\}$$

Examples of strings in the language include 01, 11010, and 100011. Examples of strings *not* in the language include ϵ, 0, and 111000.

What do we know about an automaton that can accept this language L? First, its input alphabet is $\Sigma = \{0, 1\}$. It has some set of states, Q, of which one, say q_0, is the start state. This automaton has to remember the important facts about what inputs it has seen so far. To decide whether 01 is a substring of the input, A needs to remember:

1. Has it already seen 01? If so, then it accepts every sequence of further inputs; i.e., it will only be in accepting states from now on.

2. Has it never seen 01, but its most recent input was 0, so if it now sees a 1, it will have seen 01 and can accept everything it sees from here on?

3. Has it never seen 01, but its last input was either nonexistent (it just started) or it last saw a 1? In this case, A cannot accept until it first sees a 0 and then sees a 1 immediately after.

These three conditions can each be represented by a state. Condition (3) is represented by the start state, q_0. Surely, when just starting, we need to see a 0 and then a 1. But if in state q_0 we next see a 1, then we are no closer to seeing 01, and so we must stay in state q_0. That is, $\delta(q_0, 1) = q_0$.

However, if we are in state q_0 and we next see a 0, we are in condition (2). That is, we have never seen 01, but we have our 0. Thus, let us use q_2 to represent condition (2). Our transition from q_0 on input 0 is $\delta(q_0, 0) = q_2$.

Now, let us consider the transitions from state q_2. If we see a 0, we are no better off than we were, but no worse either. We have not seen 01, but 0 was the last symbol, so we are still waiting for a 1. State q_2 describes this situation perfectly, so we want $\delta(q_2, 0) = q_2$. If we are in state q_2 and we see a 1 input, we now know there is a 0 followed by a 1. We can go to an accepting state, which we shall call q_1, and which corresponds to condition (1) above. That is, $\delta(q_2, 1) = q_1$.

Finally, we must design the transitions for state q_1. In this state, we have already seen a 01 sequence, so regardless of what happens, we shall still be in a situation where we've seen 01. That is, $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.

Thus, $Q = \{q_0, q_1, q_2\}$. As we said, q_0 is the start state, and the only accepting state is q_1; that is, $F = \{q_1\}$. The complete specification of the automaton A that accepts the language L of strings that have a 01 substring, is

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

where δ is the transition function described above. □

2.2.3 Simpler Notations for DFA's

Specifying a DFA as a five-tuple with a detailed description of the δ transition function is both tedious and hard to read. There are two preferred notations for describing automata:

1. A *transition diagram*, which is a graph such as the ones we saw in Section 2.1.

2. A *transition table*, which is a tabular listing of the δ function, which by implication tells us the set of states and the input alphabet.

Transition Diagrams

A *transition diagram* for a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

a) For each state in Q there is a node.

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

c) There is an arrow into the start state q_0, labeled *Start*. This arrow does not originate at any node.

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

Example 2.2: Figure 2.4 shows the transition diagram for the DFA that we designed in Example 2.1. We see in that diagram the three nodes that correspond to the three states. There is a *Start* arrow entering the start state, q_0, and the one accepting state, q_1, is represented by a double circle. Out of each state is one arc labeled 0 and one arc labeled 1 (although the two arcs are combined into one with a double label in the case of q_1). The arcs each correspond to one of the δ facts developed in Example 2.1. □

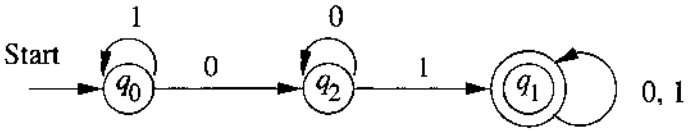


Figure 2.4: The transition diagram for the DFA accepting all strings with a substring 01

Transition Tables

A *transition table* is a conventional, tabular representation of a function like δ that takes two arguments and returns a value. The rows of the table correspond to the states, and the columns correspond to the inputs. The entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$.

Example 2.3: The transition table corresponding to the function δ of Example 2.1 is shown in Fig. 2.5. We have also shown two other features of a transition table. The start state is marked with an arrow, and the accepting states are marked with a star. Since we can deduce the sets of states and input symbols by looking at the row and column heads, we can now read from the transition table all the information we need to specify the finite automaton uniquely. □

Symbols

States

	0	1
$\rightarrow q_0$	q_2	q_0
$*q_1$	q_1	q_1
q_2	q_2	q_1

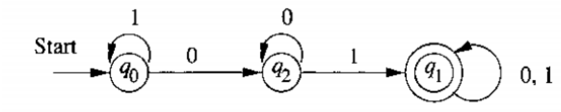


Figure 2.5: Transition table for the DFA of Example 2.1

2.2.4 Extending the Transition Function to Strings

We have explained informally that the DFA defines a language: the set of all strings that result in a sequence of state transitions from the start state to an accepting state. In terms of the transition diagram, the language of a DFA is the set of labels along all the paths that lead from the start state to any accepting state.

Now, we need to make the notion of the language of a DFA precise. To do so, we define an *extended transition function* that describes what happens when we start in any state and follow any sequence of inputs. If δ is our transition function, then the extended transition function constructed from δ will be called $\hat{\delta}$. The extended transition function is a function that takes a state q and a string w and returns a state p — the state that the automaton reaches when starting in state q and processing the sequence of inputs w. We define $\hat{\delta}$ by induction on the length of the input string, as follows:

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

INDUCTION: Suppose w is a string of the form xa; that is, a is the last symbol of w, and x is the string consisting of all but the last symbol.[3] For example, $w = 1101$ is broken into $x = 110$ and $a = 1$. Then

$$\hat{\delta}(q, w) = \delta\big(\hat{\delta}(q, x), a\big) \tag{2.1}$$

Now (2.1) may seem like a lot to take in, but the idea is simple. To compute $\hat{\delta}(q, w)$, first compute $\hat{\delta}(q, x)$, the state that the automaton is in after processing all but the last symbol of w. Suppose this state is p; that is, $\hat{\delta}(q, x) = p$. Then $\hat{\delta}(q, w)$ is what we get by making a transition from state p on input a, the last symbol of w. That is, $\hat{\delta}(q, w) = \delta(p, a)$.

Example 2.4: Let us design a DFA to accept the language

$$L = \{w \mid w \text{ has both an even number of 0's and an even number of 1's}\}$$

It should not be surprising that the job of the states of this DFA is to count both the number of 0's and the number of 1's, but count them modulo 2. That is, the state is used to remember whether the number of 0's seen so far is even or odd, and also to remember whether the number of 1's seen so far is even or odd. There are thus four states, which can be given the following interpretations:

q_0: Both the number of 0's seen so far and the number of 1's seen so far are even.

q_1: The number of 0's seen so far is even, but the number of 1's seen so far is odd.

q_2: The number of 1's seen so far is even, but the number of 0's seen so far is odd.

q_3: Both the number of 0's seen so far and the number of 1's seen so far are odd.

State q_0 is both the start state and the lone accepting state. It is the start state, because before reading any inputs, the numbers of 0's and 1's seen so far are both zero, and zero is even. It is the only accepting state, because it describes exactly the condition for a sequence of 0's and 1's to be in language L.

We now know almost how to specify the DFA for language L. It is

$$A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

[3] Recall our convention that letters at the beginning of the alphabet are symbols, and those near the end of the alphabet are strings. We need that convention to make sense of the phrase "of the form xa."

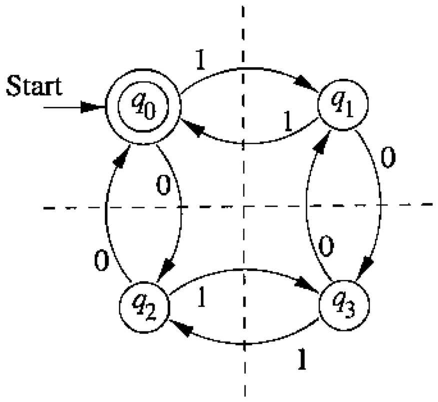


Figure 2.6: Transition diagram for the DFA of Example 2.4

where the transition function δ is described by the transition diagram of Fig. 2.6. Notice how each input 0 causes the state to cross the horizontal, dashed line. Thus, after seeing an even number of 0's we are always above the line, in state q_0 or q_1 while after seeing an odd number of 0's we are always below the line, in state q_2 or q_3. Likewise, every 1 causes the state to cross the vertical, dashed line. Thus, after seeing an even number of 1's, we are always to the left, in state q_0 or q_2, while after seeing an odd number of 1's we are to the right, in state q_1 or q_3. These observations are an informal proof that the four states have the interpretations attributed to them. However, one could prove the correctness of our claims about the states formally, by a mutual induction in the spirit of Example 1.23.

We can also represent this DFA by a transition table. Figure 2.7 shows this table. However, we are not just concerned with the design of this DFA; we want to use it to illustrate the construction of $\hat{\delta}$ from its transition function δ. Suppose the input is 110101. Since this string has even numbers of 0's and 1's both, we expect it is in the language. Thus, we expect that $\hat{\delta}(q_0, 110101) = q_0$, since q_0 is the only accepting state. Let us now verify that claim.

	0	1
$* \rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Figure 2.7: Transition table for the DFA of Example 2.4

The check involves computing $\hat{\delta}(q_0, w)$ for each prefix w of 110101, starting at ϵ and going in increasing size. The summary of this calculation is:

Standard Notation and Local Variables

After reading this section, you might imagine that our customary notation is required; that is, you *must* use δ for the transition function, use A for the name of a DFA, and so on. We tend to use the same variables to denote the same thing across all examples, because it helps to remind you of the types of variables, much the way a variable i in a program is almost always of integer type. However, we are free to call the components of an automaton, or anything else, anything we wish. Thus, you are free to call a DFA M and its transition function T if you like.

Moreover, you should not be surprised that the same variable means different things in different contexts. For example, the DFA's of Examples 2.1 and 2.4 both were given a transition function called δ. However, the two transition functions are each local variables, belonging only to their examples. These two transition functions are very different and bear no relationship to one another.

- $\hat{\delta}(q_0, \epsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta\big(\hat{\delta}(q_0, \epsilon), 1\big) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta\big(\hat{\delta}(q_0, 1), 1\big) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta\big(\hat{\delta}(q_0, 11), 0\big) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0, 1101) = \delta\big(\hat{\delta}(q_0, 110), 1\big) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta\big(\hat{\delta}(q_0, 1101), 0\big) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta\big(\hat{\delta}(q_0, 11010), 1\big) = \delta(q_1, 1) = q_0$.

□

2.2.5 The Language of a DFA

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

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ is in } F\}$$

That is, the language of A is the set of strings w that take the start state q_0 to one of the accepting states. If L is $L(A)$ for some DFA A, then we say L is a *regular language*.

Example 2.5: As we mentioned earlier, if A is the DFA of Example 2.1, then $L(A)$ is the set of all strings of 0's and 1's that contain a substring 01. If A is instead the DFA of Example 2.4, then $L(A)$ is the set of all strings of 0's and 1's whose numbers of 0's and 1's are both even. □

2.2.6 Exercises for Section 2.2

Exercise 2.2.1: In Fig. 2.8 is a marble-rolling toy. A marble is dropped at A or B. Levers x_1, x_2, and x_3 cause the marble to fall either to the left or to the right. Whenever a marble encounters a lever, it causes the lever to reverse after the marble passes, so the next marble will take the opposite branch.

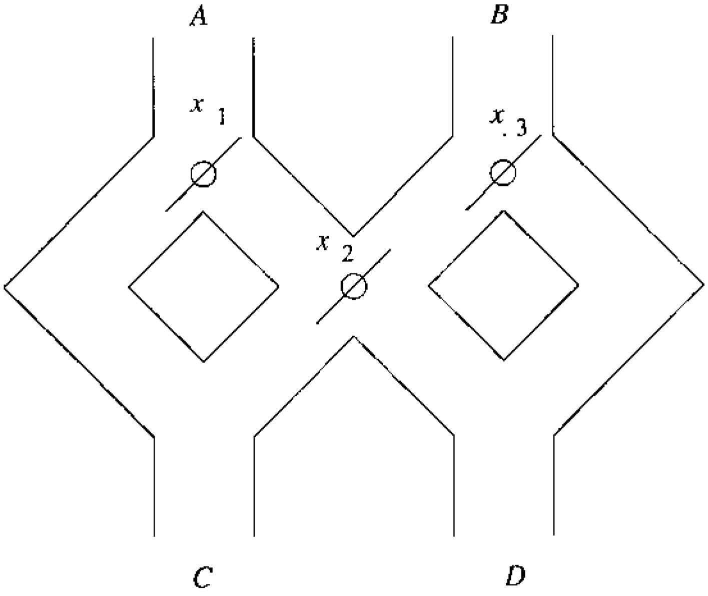


Figure 2.8: A marble-rolling toy

* a) Model this toy by a finite automaton. Let the inputs A and B represent the input into which the marble is dropped. Let acceptance correspond to the marble exiting at D; nonacceptance represents a marble exiting at C.

! b) Informally describe the language of the automaton.

c) Suppose that instead the levers switched *before* allowing the marble to pass. How would your answers to parts (a) and (b) change?

*! **Exercise 2.2.2:** We defined $\hat{\delta}$ by breaking the input string into any string followed by a single symbol (in the inductive part, Equation 2.1). However, we informally think of $\hat{\delta}$ as describing what happens along a path with a certain

string of labels, and if so, then it should not matter how we break the input string in the definition of $\hat{\delta}$. Show that in fact, $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ for any state q and strings x and y. *Hint*: Perform an induction on $|y|$.

! **Exercise 2.2.3:** Show that for any state q, string x, and input symbol a, $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$. *Hint*: Use Exercise 2.2.2.

Exercise 2.2.4: Give DFA's accepting the following languages over the alphabet $\{0, 1\}$:

* a) The set of all strings ending in 00.

b) The set of all strings with three consecutive 0's (not necessarily at the end).

c) The set of strings with 011 as a substring.

! **Exercise 2.2.5:** Give DFA's accepting the following languages over the alphabet $\{0, 1\}$:

a) The set of all strings such that each block of five consecutive symbols contains at least two 0's.

b) The set of all strings whose tenth symbol from the right end is a 1.

c) The set of strings that either begin or end (or both) with 01.

d) The set of strings such that the number of 0's is divisible by five, and the number of 1's is divisible by 3.

!! **Exercise 2.2.6:** Give DFA's accepting the following languages over the alphabet $\{0, 1\}$:

* a) The set of all strings beginning with a 1 that, when interpreted as a binary integer, is a multiple of 5. For example, strings 101, 1010, and 1111 are in the language; 0, 100, and 111 are not.

b) The set of all strings that, when interpreted *in reverse* as a binary integer, is divisible by 5. Examples of strings in the language are 0, 10011, 1001100, and 0101.

Exercise 2.2.7: Let A be a DFA and q a particular state of A, such that $\delta(q, a) = q$ for all input symbols a. Show by induction on the length of the input that for all input strings w, $\hat{\delta}(q, w) = q$.

Exercise 2.2.8: Let A be a DFA and a a particular input symbol of A, such that for all states q of A we have $\delta(q, a) = q$.

a) Show by induction on n that for all $n \geq 0$, $\hat{\delta}(q, a^n) = q$, where a^n is the string consisting of n a's.

b) Show that either $\{a\}^* \subseteq L(A)$ or $\{a\}^* \cap L(A) = \emptyset$.

***! Exercise 2.2.9:** Let $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ be a DFA, and suppose that for all a in Σ we have $\delta(q_0, a) = \delta(q_f, a)$.

a) Show that for all $w \neq \epsilon$ we have $\hat{\delta}(q_0, w) = \hat{\delta}(q_f, w)$.

b) Show that if x is a nonempty string in $L(A)$, then for all $k > 0$, x^k (i.e., x written k times) is also in $L(A)$.

***! Exercise 2.2.10:** Consider the DFA with the following transition table:

	0	1
$\rightarrow A$	A	B
$*B$	B	A

Informally describe the language accepted by this DFA, and prove by induction on the length of an input string that your description is correct. *Hint*: When setting up the inductive hypothesis, it is wise to make a statement about what inputs get you to each state, not just what inputs get you to the accepting state.

! Exercise 2.2.11: Repeat Exercise 2.2.10 for the following transition table:

	0	1
$\rightarrow *A$	B	A
$*B$	C	A
C	C	C

2.3 Nondeterministic Finite Automata

A "nondeterministic" finite automaton (*NFA*) has the power to be in several states at once. This ability is often expressed as an ability to "guess" something about its input. For instance, when the automaton is used to search for certain sequences of characters (e.g., keywords) in a long text string, it is helpful to "guess" that we are at the beginning of one of those strings and use a sequence of states to do nothing but check that the string appears, character by character. We shall see an example of this type of application in Section 2.4.

Before examining applications, we need to define nondeterministic finite automata and show that each one accepts a language that is also accepted by some DFA. That is, the NFA's accept exactly the regular languages, just as DFA's do. However, there are reasons to think about NFA's. They are often more succinct and easier to design than DFA's. Moreover, while we can always convert an NFA to a DFA, the latter may have exponentially more states than the NFA; fortunately, cases of this type are rare.

2.3.1 An Informal View of Nondeterministic Finite Automata

Like the DFA, an NFA has a finite set of states, a finite set of input symbols, one start state and a set of accepting states. It also has a transition function, which we shall commonly call δ. The difference between the DFA and the NFA is in the type of δ. For the NFA, δ is a function that takes a state and input symbol as arguments (like the DFA's transition function), but returns a set of zero, one, or more states (rather than returning exactly one state, as the DFA must). We shall start with an example of an NFA, and then make the definitions precise.

Example 2.6: Figure 2.9 shows a nondeterministic finite automaton, whose job is to accept all and only the strings of 0's and 1's that end in 01. State q_0 is the start state, and we can think of the automaton as being in state q_0 (perhaps among other states) whenever it has not yet "guessed" that the final 01 has begun. It is always possible that the next symbol does not begin the final 01, even if that symbol is 0. Thus, state q_0 may transition to itself on both 0 and 1.

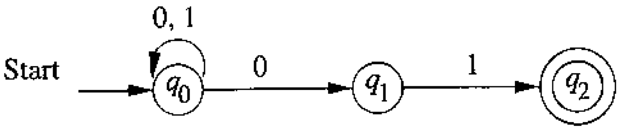


Figure 2.9: An NFA accepting all strings that end in 01

However, if the next symbol is 0, this NFA also guesses that the final 01 has begun. An arc labeled 0 thus leads from q_0 to state q_1. Notice that there are two arcs labeled 0 out of q_0. The NFA has the option of going either to q_0 or to q_1, and in fact it does both, as we shall see when we make the definitions precise. In state q_1, the NFA checks that the next symbol is 1, and if so, it goes to state q_2 and accepts.

Notice that there is no arc out of q_1 labeled 0, and there are no arcs at all out of q_2. In these situations, the thread of the NFA's existence corresponding to those states simply "dies," although other threads may continue to exist. While a DFA has exactly one arc out of each state for each input symbol, an NFA has no such constraint; we have seen in Fig. 2.9 cases where the number of arcs is zero, one, and two, for example.

Figure 2.10 suggests how an NFA processes inputs. We have shown what happens when the automaton of Fig. 2.9 receives the input sequence 00101. It starts in only its start state, q_0. When the first 0 is read, the NFA may go to either state q_0 or state q_1, so it does both. These two threads are suggested by the second column in Fig. 2.10.

Then, the second 0 is read. State q_0 may again go to both q_0 and q_1. However, state q_1 has no transition on 0, so it "dies." When the third input, a

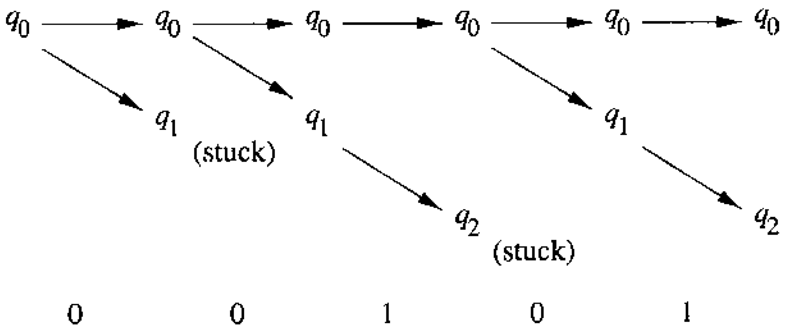


Figure 2.10: The states an NFA is in during the processing of input sequence 00101

1, occurs, we must consider transitions from both q_0 and q_1. We find that q_0 goes only to q_0 on 1, while q_1 goes only to q_2. Thus, after reading 001, the NFA is in states q_0 and q_2. Since q_2 is an accepting state, the NFA accepts 001.

However, the input is not finished. The fourth input, a 0, causes q_2's thread to die, while q_0 goes to both q_0 and q_1. The last input, a 1, sends q_0 to q_0 and q_1 to q_2. Since we are again in an accepting state, 00101 is accepted. □

2.3.2 Definition of Nondeterministic Finite Automata

Now, let us introduce the formal notions associated with nondeterministic finite automata. The differences between DFA's and NFA's will be pointed out as we do. An NFA is represented essentially like a DFA:

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

where:

1. Q is a finite set of *states.*

2. Σ is a finite set of *input symbols.*

3. q_0, a member of Q, is the *start state.*

4. F, a subset of Q, is the set of *final* (or *accepting*) states.

5. δ, the *transition function* is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q. Notice that the only difference between an NFA and a DFA is in the type of value that δ returns: a set of states in the case of an NFA and a single state in the case of a DFA.

Example 2.7: The NFA of Fig. 2.9 can be specified formally as

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

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

Figure 2.11: Transition table for an NFA that accepts all strings ending in 01

where the transition function δ is given by the transition table of Fig. 2.11. □

Notice that transition tables can be used to specify the transition function for an NFA as well as for a DFA. The only difference is that each entry in the table for the NFA is a set, even if the set is a *singleton* (has one member). Also notice that when there is no transition at all from a given state on a given input symbol, the proper entry is $\emptyset$, the empty set.

2.3.3 The Extended Transition Function

As for DFA's, we need to extend the transition function δ of an NFA to a function $\hat{\delta}$ that takes a state q and a string of input symbols w, and returns the set of states that the NFA is in if it starts in state q and processes the string w. The idea was suggested by Fig. 2.10; in essence $\hat{\delta}(q, w)$ is the column of states found after reading w, if q is the lone state in the first column. For instance, Fig. 2.10 suggests that $\hat{\delta}(q_0, 001) = \{q_0, q_2\}$. Formally, we define $\hat{\delta}$ for an NFA's transition function δ by:

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

INDUCTION: Suppose w is of the form $w = xa$, where a is the final symbol of w and x is the rest of w. Also suppose that $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$. Let

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

Then $\hat{\delta}(q, w) = \{r_1, r_2, \ldots, r_m\}$. Less formally, we compute $\hat{\delta}(q, w)$ by first computing $\hat{\delta}(q, x)$, and by then following any transition from any of these states that is labeled a.

Example 2.8: Let us use $\hat{\delta}$ to describe the processing of input 00101 by the NFA of Fig. 2.9. A summary of the steps is:

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

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

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

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

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

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

Line (1) is the basis rule. We obtain line (2) by applying δ to the lone state, q_0, that is in the previous set, and get $\{q_0, q_1\}$ as a result. Line (3) is obtained by taking the union over the two states in the previous set of what we get when we apply δ to them with input 0. That is, $\delta(q_0, 0) = \{q_0, q_1\}$, while $\delta(q_1, 0) = \emptyset$. For line (4), we take the union of $\delta(q_0, 1) = \{q_0\}$ and $\delta(q_1, 1) = \{q_2\}$. Lines (5) and (6) are similar to lines (3) and (4). □

2.3.4 The Language of an NFA

As we have suggested, an NFA accepts a string w if it is possible to make any sequence of choices of next state, while reading the characters of w, and go from the start state to any accepting state. The fact that other choices using the input symbols of w lead to a nonaccepting state, or do not lead to any state at all (i.e., the sequence of states "dies"), does not prevent w from being accepted by the NFA as a whole. Formally, if $A = (Q, \Sigma, \delta, q_0, F)$ is an NFA, then

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

That is, $L(A)$ is the set of strings w in Σ^* such that $\hat{\delta}(q_0, w)$ contains at least one accepting state.

Example 2.9: As an example, let us prove formally that the NFA of Fig. 2.9 accepts the language $L = \{w \mid w \text{ ends in } 01\}$. The proof is a mutual induction of the following three statements that characterize the three states:

1. $\hat{\delta}(q_0, w)$ contains q_0 for every w.

2. $\hat{\delta}(q_0, w)$ contains q_1 if and only if w ends in 0.

3. $\hat{\delta}(q_0, w)$ contains q_2 if and only if w ends in 01.

To prove these statements, we need to consider how A can reach each state; i.e., what was the last input symbol, and in what state was A just before reading that symbol?

Since the language of this automaton is the set of strings w such that $\hat{\delta}(q_0, w)$ contains q_2 (because q_2 is the only accepting state), the proof of these three statements, in particular the proof of (3), guarantees that the language of this NFA is the set of strings ending in 01. The proof of the theorem is an induction on $|w|$, the length of w, starting with length 0.

BASIS: If $|w| = 0$, then $w = \epsilon$. Statement (1) says that $\hat{\delta}(q_0, \epsilon)$ contains q_0, which it does by the basis part of the definition of $\hat{\delta}$. For statement (2), we know that ϵ does not end in 0, and we also know that $\hat{\delta}(q_0, \epsilon)$ does not contain q_1, again by the basis part of the definition of $\hat{\delta}$. Thus, the hypotheses of both directions of the if-and-only-if statement are false, and therefore both directions of the statement are true. The proof of statement (3) for $w = \epsilon$ is essentially the same as the above proof for statement (2).

INDUCTION: Assume that $w = xa$, where a is a symbol, either 0 or 1. We may assume statements (1) through (3) hold for x, and we need to prove them for w. That is, we assume $|w| = n + 1$, so $|x| = n$. We assume the inductive hypothesis for n and prove it for $n + 1$.

1. We know that $\hat{\delta}(q_0, x)$ contains q_0. Since there are transitions on both 0 and 1 from q_0 to itself, it follows that $\hat{\delta}(q_0, w)$ also contains q_0, so statement (1) is proved for w.

2. (If) Assume that w ends in 0; i.e., $a = 0$. By statement (1) applied to x, we know that $\hat{\delta}(q_0, x)$ contains q_0. Since there is a transition from q_0 to q_1 on input 0, we conclude that $\hat{\delta}(q_0, w)$ contains q_1.

   (Only-if) Suppose $\hat{\delta}(q_0, w)$ contains q_1. If we look at the diagram of Fig. 2.9, we see that the only way to get into state q_1 is if the input sequence w is of the form $x0$. That is enough to prove the "only-if" portion of statement (2).

3. (If) Assume that w ends in 01. Then if $w = xa$, we know that $a = 1$ and x ends in 0. By statement (2) applied to x, we know that $\hat{\delta}(q_0, x)$ contains q_1. Since there is a transition from q_1 to q_2 on input 1, we conclude that $\hat{\delta}(q_0, w)$ contains q_2.

   (Only-if) Suppose $\hat{\delta}(q_0, w)$ contains q_2. Looking at the diagram of Fig. 2.9, we discover that the only way to get to state q_2 is for w to be of the form $x1$, where $\hat{\delta}(q_0, x)$ contains q_1. By statement (2) applied to x, we know that x ends in 0. Thus, w ends in 01, and we have proved statement (3).

□

2.3.5 Equivalence of Deterministic and Nondeterministic Finite Automata

Although there are many languages for which an NFA is easier to construct than a DFA, such as the language (Example 2.6) of strings that end in 01, it is a surprising fact that every language that can be described by some NFA can also be described by some DFA. Moreover, the DFA in practice has about as many states as the NFA, although it often has more transitions. In the worst case, however, the smallest DFA can have 2^n states while the smallest NFA for the same language has only n states.

The proof that DFA's can do whatever NFA's can do involves an important "construction" called the ***subset construction*** because it involves constructing all subsets of the set of states of the NFA. In general, many proofs about automata involve constructing one automaton from another. It is important for us to observe the subset construction as an example of how one formally describes one automaton in terms of the states and transitions of another, without knowing the specifics of the latter automaton.

The subset construction starts from an NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Its goal is the description of a DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(D) = L(N)$. Notice that the input alphabets of the two automata are the same, and the start state of D is the set containing only the start state of N. The other components of D are constructed as follows.

- Q_D is the set of subsets of Q_N; i.e., Q_D is the ***power set*** of Q_N. Note that if Q_N has n states, then Q_D will have 2^n states. Often, not all these states are accessible from the start state of Q_D. Inaccessible states can be "thrown away," so effectively, the number of states of D may be much smaller than 2^n.

- F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N's states that include at least one accepting state of N.

- For each set $S \subseteq Q_N$ and for each input symbol a in Σ,

$$\delta_D(S, a) = \bigcup_{p \text{ in } S} \delta_N(p, a)$$

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

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

Figure 2.12: The complete subset construction from Fig. 2.9

Example 2.10: Let N be the automaton of Fig. 2.9 that accepts all strings that end in 01. Since N's set of states is $\{q_0, q_1, q_2\}$, the subset construction produces a DFA with $2^3 = 8$ states, corresponding to all the subsets of these three states. Figure 2.12 shows the transition table for these eight states; we shall show shortly the details of how some of these entries are computed.

Notice that this transition table belongs to a deterministic finite automaton. Even though the entries in the table are sets, the states of the constructed DFA *are* sets. To make the point clearer, we can invent new names for these states, e.g., A for $\emptyset$, B for $\{q_0\}$, and so on. The DFA transition table of Fig 2.13 defines exactly the same automaton as Fig. 2.12, but makes clear the point that the entries in the table are single states of the DFA.

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Figure 2.13: Renaming the states of Fig. 2.12

Of the eight states in Fig. 2.13, starting in the start state B, we can only reach states B, E, and F. The other five states are inaccessible from the start state and may as well not be there. We often can avoid the exponential-time step of constructing transition-table entries for every subset of states if we perform "lazy evaluation" on the subsets, as follows.

BASIS: We know for certain that the singleton set consisting only of N's start state is accessible.

INDUCTION: Suppose we have determined that set S of states is accessible. Then for each input symbol a, compute the set of states $\delta_D(S, a)$; we know that these sets of states will also be accessible.

For the example at hand, we know that $\{q_0\}$ is a state of the DFA D. We find that $\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ and $\delta_D(\{q_0\}, 1) = \{q_0\}$. Both these facts are established by looking at the transition diagram of Fig. 2.9 and observing that on 0 there are arcs out of q_0 to both q_0 and q_1, while on 1 there is an arc only to q_0. We thus have one row of the transition table for the DFA: the second row in Fig. 2.12.

One of the two sets we computed is "old"; $\{q_0\}$ has already been considered. However, the other — $\{q_0, q_1\}$ — is new and its transitions must be computed. We find $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ and $\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$. For instance, to see the latter calculation, we know that

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

We now have the fifth row of Fig. 2.12, and we have discovered one new state of D, which is $\{q_0, q_2\}$. A similar calculation tells us

$$\delta_D(\{q_0, q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$$
$$\delta_D(\{q_0, q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \emptyset = \{q_0\}$$

These calculations give us the sixth row of Fig. 2.12, but it gives us only sets of states that we have already seen.

Thus, the subset construction has converged; we know all the accessible states and their transitions. The entire DFA is shown in Fig. 2.14. Notice that it has only three states, which is, by coincidence, exactly the same number of states as the NFA of Fig. 2.9, from which it was constructed. However, the DFA of Fig. 2.14 has six transitions, compared with the four transitions in Fig. 2.9. □

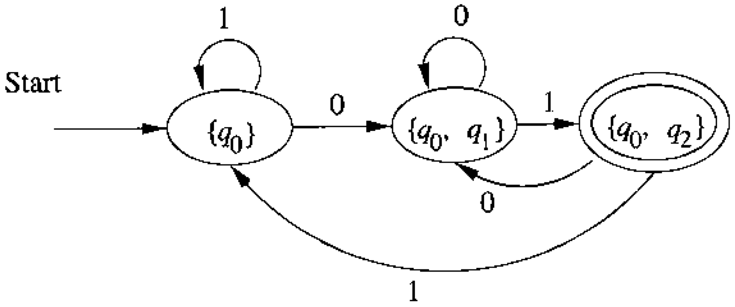


Figure 2.14: The DFA constructed from the NFA of Fig 2.9

We need to show formally that the subset construction works, although the intuition was suggested by the examples. After reading sequence of input symbols w, the constructed DFA is in one state that is the set of NFA states that the NFA would be in after reading w. Since the accepting states of the DFA are those sets that include at least one accepting state of the NFA, and the NFA also accepts if it gets into at least one of its accepting states, we may then conclude that the DFA and NFA accept exactly the same strings, and therefore accept the same language.

Theorem 2.11: If $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ is the DFA constructed from NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ by the subset construction, then $L(D) = L(N)$.

PROOF: What we actually prove first, by induction on $|w|$, is that

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Notice that each of the $\hat{\delta}$ functions returns a set of states from Q_N, but $\hat{\delta}_D$ interprets this set as one of the states of Q_D (which is the power set of Q_N), while $\hat{\delta}_N$ interprets this set as a subset of Q_N.

BASIS: Let $|w| = 0$; that is, $w = \epsilon$. By the basis definitions of $\hat{\delta}$ for DFA's and NFA's, both $\hat{\delta}_D(\{q_0\}, \epsilon)$ and $\hat{\delta}_N(q_0, \epsilon)$ are $\{q_0\}$.

INDUCTION: Let w be of length $n + 1$, and assume the statement for length n. Break w up as $w = xa$, where a is the final symbol of w. By the inductive hypothesis, $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$. Let both these sets of N's states be $\{p_1, p_2, \ldots, p_k\}$.

The inductive part of the definition of $\hat{\delta}$ for NFA's tells us

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

The subset construction, on the other hand, tells us that

$$\delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

Now, let us use (2.3) and the fact that $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ in the inductive part of the definition of $\hat{\delta}$ for DFA's:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D\big(\hat{\delta}_D(\{q_0\}, x), a\big) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.4}$$

Thus, Equations (2.2) and (2.4) demonstrate that $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. When we observe that D and N both accept w if and only if $\hat{\delta}_D(\{q_0\}, w)$ or $\hat{\delta}_N(q_0, w)$, respectively, contain a state in F_N, we have a complete proof that $L(D) = L(N)$. □

Theorem 2.12: A language L is accepted by some DFA if and only if L is accepted by some NFA.

PROOF: (If) The "if" part is the subset construction and Theorem 2.11.

(Only-if) This part is easy; we have only to convert a DFA into an identical NFA. Put intuitively, if we have the transition diagram for a DFA, we can also interpret it as the transition diagram of an NFA, which happens to have exactly one choice of transition in any situation. More formally, let $D = (Q, \Sigma, \delta_D, q_0, F)$ be a DFA. Define $N = (Q, \Sigma, \delta_N, q_0, F)$ to be the equivalent NFA, where δ_N is defined by the rule:

- If $\delta_D(q, a) = p$, then $\delta_N(q, a) = \{p\}$.

It is then easy to show by induction on $|w|$, that if $\hat{\delta}_D(q_0, w) = p$ then

$$\hat{\delta}_N(q_0, w) = \{p\}$$

We leave the proof to the reader. As a consequence, w is accepted by D if and only if it is accepted by N; i.e., $L(D) = L(N)$. □

2.3.6 A Bad Case for the Subset Construction

In Example 2.10 we found that the DFA had no more states than the NFA. As we mentioned, it is quite common in practice for the DFA to have roughly the same number of states as the NFA from which it is constructed. However, exponential growth in the number of states is possible; all the 2^n DFA states that we could construct from an n-state NFA may turn out to be accessible. The following example does not quite reach that bound, but it is an understandable way to reach 2^n states in the smallest DFA that is equivalent to an $n+1$-state NFA.

Example 2.13 : Consider the NFA N of Fig. 2.15. $L(N)$ is the set of all strings of 0's and 1's such that the nth symbol from the end is 1. Intuitively, a DFA D that accepts this language must remember the last n symbols it has read. Since any of 2^n subsets of the last n symbols could have been 1, if D has fewer than 2^n states, then there would be some state q such that D can be in state q after reading two different sequences of n bits, say $a_1a_2\cdots a_n$ and $b_1b_2\cdots b_n$.

Since the sequences are different, they must differ in some position, say $a_i \neq b_i$. Suppose (by symmetry) that $a_i = 1$ and $b_i = 0$. If $i = 1$, then q must be both an accepting state and a nonaccepting state, since $a_1a_2\cdots a_n$ is accepted (the nth symbol from the end is 1) and $b_1b_2\cdots b_n$ is not. If $i > 1$, then consider the state p that D enters after reading $i-1$ 0's. Then p must be both accepting and nonaccepting, since $a_ia_{i+1}\cdots a_n00\cdots0$ is accepted and $b_ib_{i+1}\cdots b_n00\cdots0$ is not.

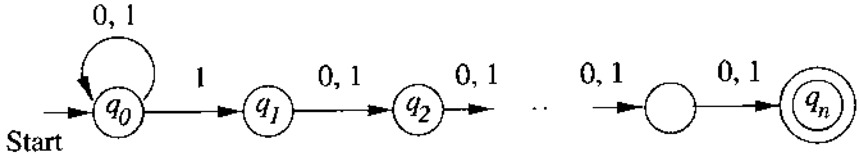


Figure 2.15: This NFA has no equivalent DFA with fewer than 2^n states

Now, let us see how the NFA N of Fig. 2.15 works. There is a state q_0 that the NFA is always in, regardless of what inputs have been read. If the next input is 1, N may also "guess" that this 1 will be the nth symbol from the end, so it goes to state q_1 as well as q_0. From state q_1, any input takes N to q_2, the next input takes it to q_3, and so on, until $n-1$ inputs later, it is in the accepting state q_n. The formal statement of what the states of N do is:

1. N is in state q_0 after reading any sequence of inputs w.
2. N is in state q_i, for $i = 1, 2, \ldots, n$, after reading input sequence w if and only if the ith symbol from the end of w is 1; that is, w is of the form $x1a_1a_2\cdots a_{i-1}$, where the a_j's are each input symbols.

We shall not prove these statements formally; the proof is an easy induction on $|w|$, mimicking Example 2.9. To complete the proof that the automaton

The Pigeonhole Principle

In Example 2.13 we used an important reasoning technique called the *pigeonhole principle.* Colloquially, if you have more pigeons than pigeonholes, and each pigeon flies into some pigeonhole, then there must be at least one hole that has more than one pigeon. In our example, the "pigeons" are the sequences of n bits, and the "pigeonholes" are the states. Since there are fewer states than sequences, one state must be assigned two sequences.

The pigeonhole principle may appear obvious, but it actually depends on the number of pigeonholes being finite. Thus, it works for finite-state automata, with the states as pigeonholes, but does not apply to other kinds of automata that have an infinite number of states.

To see why the finiteness of the number of pigeonholes is essential, consider the infinite situation where the pigeonholes correspond to integers $1, 2, \ldots$. Number the pigeons $0, 1, 2, \ldots$, so there is one more pigeon than there are pigeonholes. However, we can send pigeon i to hole $i + 1$ for all $i \geq 0$. Then each of the infinite number of pigeons gets a pigeonhole, and no two pigeons have to share a pigeonhole.

accepts exactly those strings with a 1 in the nth position from the end, we consider statement (2) with $i = n$. That says N is in state q_n if and only if the nth symbol from the end is 1. But q_n is the only accepting state, so that condition also characterizes exactly the set of strings accepted by N. □

2.3.7 Exercises for Section 2.3

* **Exercise 2.3.1:** Convert to a DFA the following NFA:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\emptyset$
$*s$	$\{s\}$	$\{s\}$

Exercise 2.3.2: Convert to a DFA the following NFA:

	0	1
$\rightarrow p$	$\{q, s\}$	$\{q\}$
$*q$	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
$*s$	$\emptyset$	$\{p\}$

Dead States and DFA's Missing Some Transitions

We have formally defined a DFA to have a transition from any state, on any input symbol, to exactly one state. However, sometimes, it is more convenient to design the DFA to "die" in situations where we know it is impossible for any extension of the input sequence to be accepted. For instance, observe the automaton of Fig. 1.2, which did its job by recognizing a single keyword, **then**, and nothing else. Technically, this automaton is not a DFA, because it lacks transitions on most symbols from each of its states.

However, such an automaton is an NFA. If we use the subset construction to convert it to a DFA, the automaton looks almost the same, but it includes a ***dead state***, that is, a nonaccepting state that goes to itself on every possible input symbol. The dead state corresponds to $\emptyset$, the empty set of states of the automaton of Fig. 1.2.

In general, we can add a dead state to any automaton that has *no more* than one transition for any state and input symbol. Then, add a transition to the dead state from each other state q, on all input symbols for which q has no other transition. The result will be a DFA in the strict sense. Thus, we shall sometimes refer to an automaton as a DFA if it has *at most* one transition out of any state on any symbol, rather than if it has *exactly one* transition.

! **Exercise 2.3.3:** Convert the following NFA to a DFA and informally describe the language it accepts.

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r, s\}$	$\{t\}$
r	$\{p, r\}$	$\{t\}$
$*s$	$\emptyset$	$\emptyset$
$*t$	$\emptyset$	$\emptyset$

! **Exercise 2.3.4:** Give nondeterministic finite automata to accept the following languages. Try to take advantage of nondeterminism as much as possible.

* a) The set of strings over alphabet $\{0, 1, \ldots, 9\}$ such that the final digit has appeared before.

b) The set of strings over alphabet $\{0, 1, \ldots, 9\}$ such that the final digit has *not* appeared before.

c) The set of strings of 0's and 1's such that there are two 0's separated by a number of positions that is a multiple of 4. Note that 0 is an allowable multiple of 4.

Exercise 2.3.5: In the only-if portion of Theorem 2.12 we omitted the proof by induction on $|w|$ that if $\hat{\delta}_D(q_0, w) = p$ then $\hat{\delta}_N(q_0, w) = \{p\}$. Supply this proof.

! **Exercise 2.3.6:** In the box on "Dead States and DFA's Missing Some Transitions," we claim that if N is an NFA that has at most one choice of state for any state and input symbol (i.e., $\delta(q, a)$ never has size greater than 1), then the DFA D constructed from N by the subset construction has exactly the states and transitions of N plus transitions to a new dead state whenever N is missing a transition for a given state and input symbol. Prove this contention.

Exercise 2.3.7: In Example 2.13 we claimed that the NFA N is in state q_i, for $i = 1, 2, \ldots, n$, after reading input sequence w if and only if the ith symbol from the end of w is 1. Prove this claim.

2.4 An Application: Text Search

In this section, we shall see that the abstract study of the previous section, where we considered the "problem" of deciding whether a sequence of bits ends in 01, is actually an excellent model for several real problems that appear in applications such as Web search and extraction of information from text.

2.4.1 Finding Strings in Text

A common problem in the age of the Web and other on-line text repositories is the following. Given a set of words, find all documents that contain one (or all) of those words. A search engine is a popular example of this process. The search engine uses a particular technology, called *inverted indexes*, where for each word appearing on the Web (there are 100,000,000 different words), a list of all the places where that word occurs is stored. Machines with very large amounts of main memory keep the most common of these lists available, allowing many people to search for documents at once.

Inverted-index techniques do not make use of finite automata, but they also take very large amounts of time for crawlers to copy the Web and set up the indexes. There are a number of related applications that are unsuited for inverted indexes, but are good applications for 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. For example:

   (a) Every day, news analysts want to search the day's on-line news articles for relevant topics. For example, a financial analyst might search for certain stock ticker symbols or names of companies.

   (b) A "shopping robot" wants to search for the current prices charged for the items that its clients request. The robot will retrieve current catalog pages from the Web and then search those pages for words that suggest a price for a particular item.

2. The documents to be searched cannot be cataloged. For example, Amazon.com does not make it easy for crawlers to find all the pages for all the books that the company sells. Rather, these pages are generated "on the fly" in response to queries. However, we could send a query for books on a certain topic, say "finite automata," and then search the pages retrieved for certain words, e.g., "excellent" in a review portion.

2.4.2 Nondeterministic Finite Automata for Text Search

Suppose we are given a set of words, which we shall call the ***keywords***, and we want to find occurrences of any of these words. In applications such as these, a useful way to proceed is to design a nondeterministic finite automaton, which signals, by entering an accepting state, that it has seen one of the keywords. The text of a document is fed, one character at a time to this NFA, which then recognizes occurrences of the keywords in this text. There is a simple form to an NFA that recognizes a set of keywords.

1. There is a start state with a transition to itself on every input symbol, e.g. every printable ASCII character if we are examining text. Intuitively, the start state represents a "guess" that we have not yet begun to see one of the keywords, even if we have seen some letters of one of these words.

2. For each keyword $a_1a_2\cdots a_k$, there are k states, say $q_1, q_2, \ldots, q_k$. There is a transition from the start state to q_1 on symbol a_1, a transition from q_1 to q_2 on symbol a_2, and so on. The state q_k is an accepting state and indicates that the keyword $a_1a_2\cdots a_k$ has been found.

Example 2.14: Suppose we want to design an NFA to recognize occurrences of the words `web` and `ebay`. The transition diagram for the NFA designed using the rules above is in Fig. 2.16. State 1 is the start state, and we use Σ to stand for the set of all printable ASCII characters. States 2 through 4 have the job of recognizing `web`, while states 5 through 8 recognize `ebay`. □

Of course the NFA is not a program. We have two major choices for an implementation of this NFA.

1. Write a program that simulates this NFA by computing the set of states it is in after reading each input symbol. The simulation was suggested in Fig. 2.10.

2. Convert the NFA to an equivalent DFA using the subset construction. Then simulate the DFA directly.

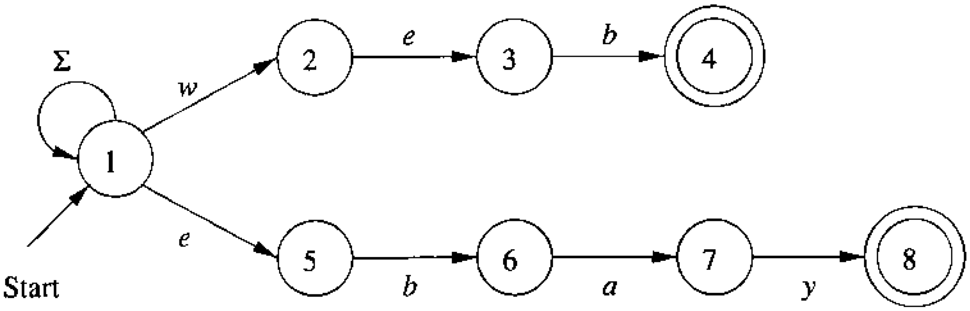


Figure 2.16: An NFA that searches for the words web and ebay

Some text-processing programs, such as advanced forms of the UNIX **grep** command (**egrep** and **fgrep**) actually use a mixture of these two approaches. However, for our purposes, conversion to a DFA is easy and is guaranteed not to increase the number of states.

2.4.3 A DFA to Recognize a Set of Keywords

We can apply the subset construction to any NFA. However, when we apply that construction to an NFA that was designed from a set of keywords, according to the strategy of Section 2.4.2, we find that the number of states of the DFA is never greater than the number of states of the NFA. Since in the worst case the number of states exponentiates as we go to the DFA, this observation is good news and explains why the method of designing an NFA for keywords and then constructing a DFA from it is used frequently. The rules for constructing the set of DFA states is as follows.

a) If q_0 is the start state of the NFA, then $\{q_0\}$ is one of the states of the DFA.

b) Suppose p is one of the NFA states, and it is reached from the start state along a path whose symbols are $a_1a_2\cdots a_m$. Then one of the DFA states is the set of NFA states consisting of:

   1. q_0.

   2. p.

   3. Every other state of the NFA that is reachable from q_0 by following a path whose labels are a suffix of $a_1a_2\cdots a_m$, that is, any sequence of symbols of the form $a_ja_{j+1}\cdots a_m$.

Note that in general, there will be one DFA state for each NFA state p. However, in step (b), two states may actually yield the same set of NFA states, and thus become one state of the DFA. For example, if two of the keywords begin with the same letter, say a, then the two NFA states that are reached from q_0 by an arc labeled a will yield the same set of NFA states and thus get merged in the DFA.

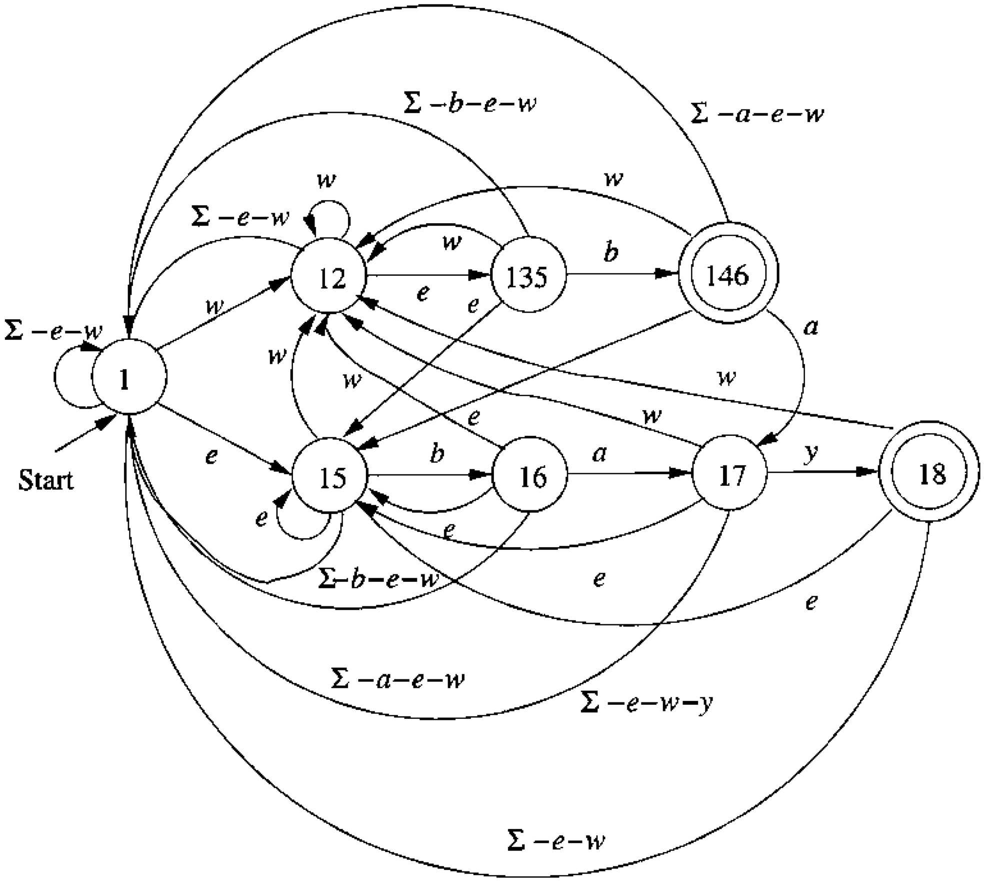


Figure 2.17: Conversion of the NFA from Fig. 2.16 to a DFA

Example 2.15 : The construction of a DFA from the NFA of Fig. 2.16 is shown in Fig. 2.17. Each of the states of the DFA is located in the same position as the state p from which it is derived using rule (b) above. For example, consider the state 135, which is our shorthand for $\{1, 3, 5\}$. This state was constructed from state 3. It includes the start state, 1, because every set of the DFA states does. It also includes state 5 because that state is reached from state 1 by a suffix, e, of the string we that reaches state 3 in Fig. 2.16.

The transitions for each of the DFA states may be calculated according to the subset construction. However, the rule is simple. From any set of states that includes the start state q_0 and some other states $\{p_1, p_2, \ldots, p_n\}$, determine, for each symbol x, where the p_i's go in the NFA, and let this DFA state have a transition labeled x to the DFA state consisting of q_0 and all the targets of the

p_i's on symbol x. On all symbols x such that there are no transitions out of any of the p_i's on symbol x, let this DFA state have a transition on x to that state of the DFA consisting of q_0 and all states that are reached from q_0 in the NFA following an arc labeled x.

For instance, consider state 135 of Fig. 2.17. The NFA of Fig. 2.16 has transitions on symbol b from states 3 and 5 to states 4 and 6, respectively. Therefore, on symbol b, 135 goes to 146. On symbol e, there are no transitions of the NFA out of 3 or 5, but there is a transition from 1 to 5. Thus, in the DFA, 135 goes to 15 on input e. Similarly, on input w, 135 goes to 12.

On every other symbol x, there are no transitions out or 3 or 5, and state 1 goes only to itself. Thus, there are transitions from 135 to 1 on every symbol in Σ other than b, e, and w. We use the notation $\Sigma - b - e - w$ to represent this set, and use similar representations of other sets in which a few symbols are removed from Σ. □

2.4.4 Exercises for Section 2.4

Exercise 2.4.1: Design NFA's to recognize the following sets of strings.

* a) abc, abd, and aacd. Assume the alphabet is $\{a, b, c, d\}$.

b) 0101, 101, and 011.

c) ab, bc, and ca. Assume the alphabet is $\{a, b, c\}$.

Exercise 2.4.2: Convert each of your NFA's from Exercise 2.4.1 to DFA's.

2.5 Finite Automata With Epsilon-Transitions

We shall now introduce another extension of the finite automaton. The new "feature" is that we allow a transition on ϵ, the empty string. In effect, an NFA is allowed to make a transition spontaneously, without receiving an input symbol. Like the nondeterminism added in Section 2.3, this new capability does not expand the class of languages that can be accepted by finite automata, but it does give us some added "programming convenience." We shall also see, when we take up regular expressions in Section 3.1, how NFA's with ϵ-transitions, which we call *ϵ-NFA's*, are closely related to regular expressions and useful in proving the equivalence between the classes of languages accepted by finite automata and by regular expressions.

2.5.1 Uses of ϵ-Transitions

We shall begin with an informal treatment of ϵ-NFA's, using transition diagrams with ϵ allowed as a label. In the examples to follow, think of the automaton as accepting those sequences of labels along paths from the start state to an accepting state. However, each ϵ along a path is "invisible"; i.e., it contributes nothing to the string along the path.

Example 2.16: In Fig. 2.18 is an ϵ-NFA that accepts decimal numbers consisting of:

1. An optional + or − sign,
2. A string of digits,
3. A decimal point, and
4. Another string of digits. Either this string of digits, or the string (2) can be empty, but at least one of the two strings of digits must be nonempty.

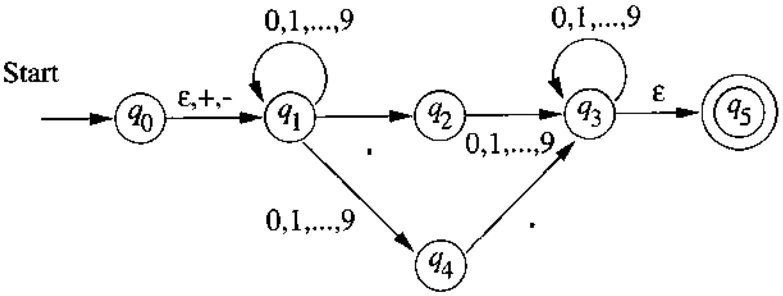


Figure 2.18: An ϵ-NFA accepting decimal numbers

Of particular interest is the transition from q_0 to q_1 on any of ϵ, +, or −. Thus, state q_1 represents the situation in which we have seen the sign if there is one, and perhaps some digits, but not the decimal point. State q_2 represents the situation where we have just seen the decimal point, and may or may not have seen prior digits. In q_4 we have definitely seen at least one digit, but not the decimal point. Thus, the interpretation of q_3 is that we have seen a decimal point and at least one digit, either before or after the decimal point. We may stay in q_3 reading whatever digits there are, and also have the option of "guessing" the string of digits is complete and going spontaneously to q_5, the accepting state. □

Example 2.17: The strategy we outlined in Example 2.14 for building an NFA that recognizes a set of keywords can be simplified further if we allow ϵ-transitions. For instance, the NFA recognizing the keywords web and ebay, which we saw in Fig. 2.16, can also be implemented with ϵ-transitions as in Fig. 2.19. In general, we construct a complete sequence of states for each keyword, as if it were the only word the automaton needed to recognize. Then, we add a new start state (state 9 in Fig. 2.19), with ϵ-transitions to the start-states of the automata for each of the keywords. □

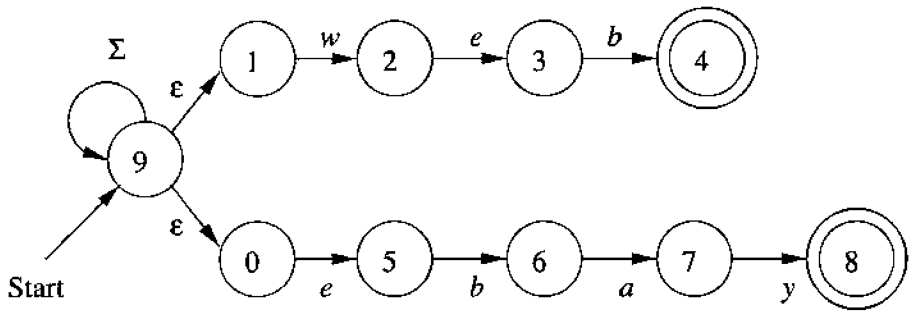


Figure 2.19: Using ϵ-transitions to help recognize keywords

2.5.2 The Formal Notation for an ϵ-NFA

We may represent an ϵ-NFA exactly as we do an NFA, with one exception: the transition function must include information about transitions on ϵ. Formally, we represent an ϵ-NFA A by $A = (Q, \Sigma, \delta, q_0, F)$, where all components have their same interpretation as for an NFA, except that δ is now a function that takes as arguments:

1. A state in Q, and

2. A member of $\Sigma \cup \{\epsilon\}$, that is, either an input symbol, or the symbol ϵ. We require that ϵ, the symbol for the empty string, cannot be a member of the alphabet Σ, so no confusion results.

Example 2.18: The ϵ-NFA of Fig. 2.18 is represented formally as

$$E = (\{q_0, q_1, \ldots, q_5\}, \{., +, -, 0, 1, \ldots, 9\}, \delta, q_0, \{q_5\})$$

where δ is defined by the transition table in Fig. 2.20. □

	ϵ	$+, -$	$.$	$0, 1, \ldots, 9$
q_0	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Figure 2.20: Transition table for Fig. 2.18

2.5.3 Epsilon-Closures

We shall proceed to give formal definitions of an extended transition function for ϵ-NFA's, which leads to the definition of acceptance of strings and languages by these automata, and eventually lets us explain why ϵ-NFA's can be simulated by DFA's. However, we first need to learn a central definition, called the *ϵ-closure* of a state. Informally, we ϵ-close a state q by following all transitions out of q that are labeled ϵ. However, when we get to other states by following ϵ, we follow the ϵ-transitions out of those states, and so on, eventually finding every state that can be reached from q along any path whose arcs are all labeled ϵ. Formally, we define the ϵ-closure ECLOSE(q) recursively, as follows:

BASIS: State q is in ECLOSE(q).

INDUCTION: If state p is in ECLOSE(q), and there is a transition from state p to state r labeled ϵ, then r is in ECLOSE(q). More precisely, if δ is the transition function of the ϵ-NFA involved, and p is in ECLOSE(q), then ECLOSE(q) also contains all the states in $\delta(p, \epsilon)$.

Example 2.19: For the automaton of Fig. 2.18, each state is its own ϵ-closure, with two exceptions: ECLOSE$(q_0) = \{q_0, q_1\}$ and ECLOSE$(q_3) = \{q_3, q_5\}$. The reason is that there are only two ϵ-transitions, one that adds q_1 to ECLOSE(q_0) and the other that adds q_5 to ECLOSE(q_3).

A more complex example is given in Fig. 2.21. For this collection of states, which may be part of some ϵ-NFA, we can conclude that

$$\text{ECLOSE}(1) = \{1, 2, 3, 4, 6\}$$

Each of these states can be reached from state 1 along a path exclusively labeled ϵ. For example, state 6 is reached by the path $1 \to 2 \to 3 \to 6$. State 7 is not in ECLOSE(1), since although it is reachable from state 1, the path must use the arc $4 \to 5$ that is not labeled ϵ. The fact that state 6 is also reached from state 1 along a path $1 \to 4 \to 5 \to 6$ that has non-ϵ transitions is unimportant. The existence of one path with all labels ϵ is sufficient to show state 6 is in ECLOSE(1). □

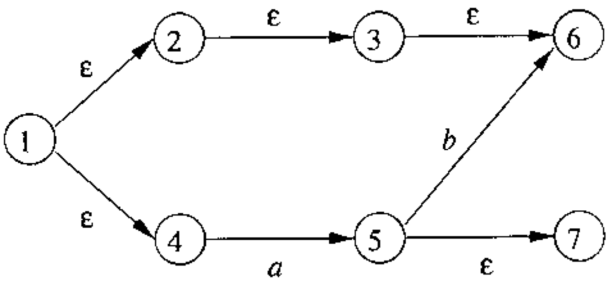


Figure 2.21: Some states and transitions

2.5.4 Extended Transitions and Languages for ϵ-NFA's

The ϵ-closure allows us to explain easily what the transitions of an ϵ-NFA look like when given a sequence of (non-ϵ) inputs. From there, we can define what it means for an ϵ-NFA to accept its input.

Suppose that $E = (Q, \Sigma, \delta, q_0, F)$ is an ϵ-NFA. We first define $\hat{\delta}$, the extended transition function, to reflect what happens on a sequence of inputs. The intent is that $\hat{\delta}(q, w)$ is the set of states that can be reached along a path whose labels, when concatenated, form the string w. As always, ϵ's along this path do not contribute to w. The appropriate recursive definition of $\hat{\delta}$ is:

BASIS: $\hat{\delta}(q, \epsilon) = \text{ECLOSE}(q)$. That is, if the label of the path is ϵ, then we can follow only ϵ-labeled arcs extending from state q; that is exactly what ECLOSE does.

INDUCTION: Suppose w is of the form xa, where a is the last symbol of w. Note that a is a member of Σ; it cannot be ϵ, which is not in Σ. We compute $\hat{\delta}(q, w)$ as follows:

1. Let $\{p_1, p_2, \ldots, p_k\}$ be $\hat{\delta}(q, x)$. That is, the p_i's are all and only the states that we can reach from q following a path labeled x. This path may end with one or more transitions labeled ϵ, and may have other ϵ-transitions, as well.

2. Let $\bigcup_{i=1}^{k} \delta(p_i, a)$ be the set $\{r_1, r_2, \ldots, r_m\}$. That is, follow all transitions labeled a from states we can reach from q along paths labeled x. The r_j's are *some* of the states we can reach from q along paths labeled w. The additional states we can reach are found from the r_j's by following ϵ-labeled arcs in step (3), below.

3. Then $\hat{\delta}(q, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$. This additional closure step includes all the paths from q labeled w, by considering the possibility that there are additional ϵ-labeled arcs that we can follow after making a transition on the final "real" symbol, a.

Example 2.20: Let us compute $\hat{\delta}(q_0, 5.6)$ for the ϵ-NFA of Fig. 2.18. A summary of the steps needed are as follows:

- $\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$.

- Compute $\hat{\delta}(q_0, 5)$ as follows:

  1. First compute the transitions on input 5 from the states q_0 and q_1 that we obtained in the calculation of $\hat{\delta}(q_0, \epsilon)$, above. That is, we compute $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.

  2. Next, ϵ-close the members of the set computed in step (1). We get $\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$. That set is $\hat{\delta}(q_0, 5)$. This two-step pattern repeats for the next two symbols.

- Compute $\hat{\delta}(q_0, \texttt{5.})$ as follows:

  1. First compute $\delta(q_1, \texttt{.}) \cup \delta(q_4, \texttt{.}) = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$.
  2. Then compute

     $$\hat{\delta}(q_0, \texttt{5.}) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$$

- Compute $\hat{\delta}(q_0, \texttt{5.6})$ as follows:

  1. First compute $\delta(q_2, \texttt{6}) \cup \delta(q_3, \texttt{6}) \cup \delta(q_5, \texttt{6}) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$.
  2. Then compute $\hat{\delta}(q_0, \texttt{5.6}) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$.

□

Now, we can define the language of an ϵ-NFA $E = (Q, \Sigma, \delta, q_0, F)$ in the expected way: $L(E) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$. That is, the language of E is the set of strings w that take the start state to at least one accepting state. For instance, we saw in Example 2.20 that $\hat{\delta}(q_0, \texttt{5.6})$ contains the accepting state q_5, so the string 5.6 is in the language of that ϵ-NFA.

2.5.5 Eliminating ϵ-Transitions

Given any ϵ-NFA E, we can find a DFA D that accepts the same language as E. The construction we use is very close to the subset construction, as the states of D are subsets of the states of E. The only difference is that we must incorporate ϵ-transitions of E, which we do through the mechanism of the ϵ-closure.

Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. Then the equivalent DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

is defined as follows:

1. Q_D is the set of subsets of Q_E. More precisely, we shall find that all accessible states of D are *ϵ-closed* subsets of Q_E, that is, sets $S \subseteq Q_E$ such that $S = \text{ECLOSE}(S)$. Put another way, the ϵ-closed sets of states S are those such that any ϵ-transition out of one of the states in S leads to a state that is also in S. Note that $\emptyset$ is an ϵ-closed set.

2. $q_D = \text{ECLOSE}(q_0)$; that is, we get the start state of D by closing the set consisting of only the start state of E. Note that this rule differs from the original subset construction, where the start state of the constructed automaton was just the set containing the start state of the given NFA.

3. F_D is those sets of states that contain at least one accepting state of E. That is, $F_D = \{S \mid S \text{ is in } Q_D \text{ and } S \cap F_E \neq \emptyset\}$.

4. $\delta_D(S, a)$ is computed, for all a in Σ and sets S in Q_D by:

(a) Let $S = \{p_1, p_2, \ldots, p_k\}$.

(b) Compute $\bigcup_{i=1}^{k} \delta_E(p_i, a)$; let this set be $\{r_1, r_2, \ldots, r_m\}$.

(c) Then $\delta_D(S, a) = \bigcup_{j=1}^{m}$ ECLOSE(r_j).

Example 2.21: Let us eliminate ϵ-transitions from the ϵ-NFA of Fig. 2.18, which we shall call E in what follows. From E, we construct an DFA D, which is shown in Fig. 2.22. However, to avoid clutter, we omitted from Fig. 2.22 the dead state $\emptyset$ and all transitions to the dead state. You should imagine that for each state shown in Fig. 2.22 there are additional transitions from any state to $\emptyset$ on any input symbols for which a transition is not indicated. Also, the state $\emptyset$ has transitions to itself on all input symbols.

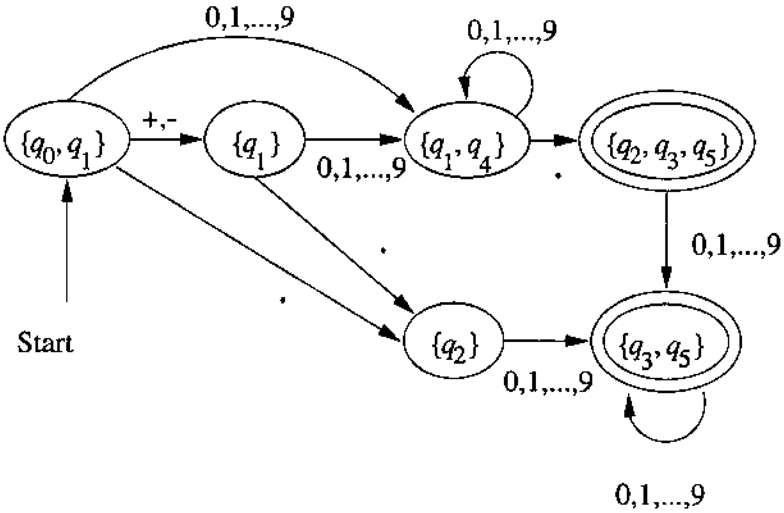


Figure 2.22: The DFA D that eliminates ϵ-transitions from Fig. 2.18

Since the start state of E is q_0, the start state of D is ECLOSE(q_0), which is $\{q_0, q_1\}$. Our first job is to find the successors of q_0 and q_1 on the various symbols in Σ; note that these symbols are the plus and minus signs, the dot, and the digits 0 through 9. On $+$ and $-$, q_1 goes nowhere in Fig. 2.18, while q_0 goes to q_1. Thus, to compute $\delta_D(\{q_0, q_1\}, +)$ we start with $\{q_1\}$ and ϵ-close it. Since there are no ϵ-transitions out of q_1, we have $\delta_D(\{q_0, q_1\}, +) = \{q_1\}$. Similarly, $\delta_D(\{q_0, q_1\}, -) = \{q_1\}$. These two transitions are shown by one arc in Fig. 2.22.

Next, we need to compute $\delta_D(\{q_0, q_1\}, .)$. Since q_0 goes nowhere on the dot, and q_1 goes to q_2 in Fig. 2.18, we must ϵ-close $\{q_2\}$. As there are no ϵ-transitions out of q_2, this state is its own closure, so $\delta_D(\{q_0, q_1\}, .) = \{q_2\}$.

Finally, we must compute $\delta_D(\{q_0, q_1\}, 0)$, as an example of the transitions from $\{q_0, q_1\}$ on all the digits. We find that q_0 goes nowhere on the digits, but q_1 goes to both q_1 and q_4. Since neither of those states have ϵ-transitions out, we conclude $\delta_D(\{q_0, q_1\}, 0) = \{q_1, q_4\}$, and likewise for the other digits.

We have now explained the arcs out of $\{q_0, q_1\}$ in Fig. 2.22. The other transitions are computed similarly, and we leave them for you to check. Since q_5 is the only accepting state of E, the accepting states of D are those accessible states that contain q_5. We see these two sets $\{q_3, q_5\}$ and $\{q_2, q_3, q_5\}$ indicated by double circles in Fig. 2.22. □

Theorem 2.22: A language L is accepted by some ϵ-NFA if and only if L is accepted by some DFA.

PROOF: (If) This direction is easy. Suppose $L = L(D)$ for some DFA. Turn D into an ϵ-DFA E by adding transitions $\delta(q, \epsilon) = \emptyset$ for all states q of D. Technically, we must also convert the transitions of D on input symbols, e.g., $\delta_D(q, a) = p$ into an NFA-transition to the set containing only p, that is $\delta_E(q, a) = \{p\}$. Thus, the transitions of E and D are the same, but E explicitly states that there are no transitions out of any state on ϵ.

(Only-if) Let $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ be an ϵ-NFA. Apply the modified subset construction described above to produce the DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

We need to show that $L(D) = L(E)$, and we do so by showing that the extended transition functions of E and D are the same. Formally, we show $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ by induction on the length of w.

BASIS: If $|w| = 0$, then $w = \epsilon$. We know $\hat{\delta}_E(q_0, \epsilon) = \text{ECLOSE}(q_0)$. We also know that $q_D = \text{ECLOSE}(q_0)$, because that is how the start state of D is defined. Finally, for a DFA, we know that $\hat{\delta}(p, \epsilon) = p$ for any state p, so in particular, $\hat{\delta}_D(q_D, \epsilon) = \text{ECLOSE}(q_0)$. We have thus proved that $\hat{\delta}_E(q_0, \epsilon) = \hat{\delta}_D(q_D, \epsilon)$.

INDUCTION: Suppose $w = xa$, where a is the final symbol of w, and assume that the statement holds for x. That is, $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x)$. Let both these sets of states be $\{p_1, p_2, \ldots, p_k\}$.

By the definition of $\hat{\delta}$ for ϵ-NFA's, we compute $\hat{\delta}_E(q_0, w)$ by:

1. Let $\{r_1, r_2, \ldots, r_m\}$ be $\bigcup_{i=1}^{k} \delta_E(p_i, a)$.

2. Then $\hat{\delta}_E(q_0, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$.

If we examine the construction of DFA D in the modified subset construction above, we see that $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ is constructed by the same two steps (1) and (2) above. Thus, $\hat{\delta}_D(q_D, w)$, which is $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ is the same set as $\hat{\delta}_E(q_0, w)$. We have now proved that $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ and completed the inductive part. □

2.5.6 Exercises for Section 2.5

* **Exercise 2.5.1:** Consider the following ϵ-NFA.

	ϵ	a	b	c
$\rightarrow p$	$\emptyset$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\emptyset$
$*r$	$\{q\}$	$\{r\}$	$\emptyset$	$\{p\}$

a) Compute the ϵ-closure of each state.

b) Give all the strings of length three or less accepted by the automaton.

c) Convert the automaton to a DFA.

Exercise 2.5.2: Repeat Exercise 2.5.1 for the following ϵ-NFA:

	ϵ	a	b	c
$\rightarrow p$	$\{q, r\}$	$\emptyset$	$\{q\}$	$\{r\}$
q	$\emptyset$	$\{p\}$	$\{r\}$	$\{p, q\}$
$*r$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Exercise 2.5.3: Design ϵ-NFA's for the following languages. Try to use ϵ-transitions to simplify your design.

a) The set of strings consisting of zero or more a's followed by zero or more b's, followed by zero or more c's.

! b) The set of strings that consist of either 01 repeated one or more times or 010 repeated one or more times.

! c) The set of strings of 0's and 1's such that at least one of the last ten positions is a 1.

2.6 Summary of Chapter 2

✦ *Deterministic Finite Automata*: A DFA has a finite set of states and a finite set of input symbols. One state is designated the start state, and zero or more states are accepting states. A transition function determines how the state changes each time an input symbol is processed.

✦ *Transition Diagrams*: It is convenient to represent automata by a graph in which the nodes are the states, and arcs are labeled by input symbols, indicating the transitions of that automaton. The start state is designated by an arrow, and the accepting states by double circles.

- *Language of an Automaton*: The automaton accepts strings. A string is accepted if, starting in the start state, the transitions caused by processing the symbols of that string one-at-a-time lead to an accepting state. In terms of the transition diagram, a string is accepted if it is the label of a path from the start state to some accepting state.

- *Nondeterministic Finite Automata*: The NFA differs from the DFA in that the NFA can have any number of transitions (including zero) to next states from a given state on a given input symbol.

- *The Subset Construction*: By treating sets of states of an NFA as states of a DFA, it is possible to convert any NFA to a DFA that accepts the same language.

- *ϵ-Transitions*: We can extend the NFA by allowing transitions on an empty input, i.e., no input symbol at all. These extended NFA's can be converted to DFA's accepting the same language.

- *Text-Searching Applications*: Nondeterministic finite automata are a useful way to represent a pattern matcher that scans a large body of text for one or more keywords. These automata are either simulated directly in software or are first converted to a DFA, which is then simulated.

2.7 References for Chapter 2

The formal study of finite-state systems is generally regarded as originating with [2]. However, this work was based on a "neural nets" model of computing, rather than the finite automaton we know today. The conventional DFA was independently proposed, in several similar variations, by [1], [3], and [4]. The nondeterministic finite automaton and the subset construction are from [5].

1. D. A. Huffman, "The synthesis of sequential switching circuits," *J. Franklin Inst.* **257**:3-4 (1954), pp. 161–190 and 275–303.

2. W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervious activity," *Bull. Math. Biophysics* **5** (1943), pp. 115–133.

3. G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal* **34**:5 (1955), pp. 1045–1079.

4. E. F. Moore, "Gedanken experiments on sequential machines," in [6], pp. 129–153.

5. M. O. Rabin and D. Scott, "Finite automata and their decision problems," *IBM J. Research and Development* **3**:2 (1959), pp. 115–125.

6. C. E. Shannon and J. McCarthy, *Automata Studies*, Princeton Univ. Press, 1956.