Only if part
If $L(A)$ is infinite, there exists $x \in L(A)$ such that $|x| \geq n$. Decompose $x$ as $x_1 x_2 x_3$ as in the above “if part”. If both $|x_1 x_3| < n$ and $|x_2| \leq n$, the thesis is proved. Otherwise, suppose first that $|x_1 x_3| \geq n$. If $|x_1 x_3| < 2n$ the thesis is proved because $x_1 x_3 \in L(A)$. Otherwise, the same decomposition is applied to $x_1 x_3$ and so on until a string $y$ is found such that $y = y_1 y_2 y_3$, with $\delta^*(q_0, y_3) = q_1$, $\delta^*(q_1, y_2) = q_1$, $\delta^*(q_1, y_3) \in F$ and $|y_1 y_3| < n$. If $|y_2| \leq n$, the thesis is proved. Otherwise $y_2$ can again be decomposed as $z_1 z_2 z_3$ such that $z_2 \neq \epsilon$ and $\delta^*(q_1, z_1 z_3) = q_1$. The procedure continues until an $y'_2$ such that $\delta^*(q_1, y'_2) = q_1$ and $|y'_2| \leq n$ is actually found.

The previous theorems are based on the fact that a FA can present loops in its graphlike representation. If it does not, the accepted language is finite; if it does, the accepted language is infinite. The possible presence of loops justifies also the following theorem, which describes an interesting property of languages recognized by FAs and somewhat generalizes Theorems 1.1 and 1.2.

**Theorem 1.3 (Pumping Lemma)**
Let $A$ be a FA. There exists a constant $k$ such that if $x \in L(A)$ and $|x| \geq k$, then $x$ can be written as $y w z$, where $1 \leq |w| \leq k$ and $y w z \in L$, for every $i \geq 0$.

The proof is left to the reader as an exercise. Hint: $k$ is the number of states in $A$.

Now we turn to another class of problems concerning FAs. First of all, let us observe that there exist more than one FA that can recognize a given language. For example, Figure 1.14 shows a FA that recognizes exactly the same language that was recognized by the FA shown in Figure 1.7.

Thus, two questions naturally arise. How can we decide whether two FAs recognize exactly the same language? Second, can we define some "canonical-form automaton," for example, a minimum-state automaton, and determine a procedure that transforms any FA into such canonical form? In the sequel, we give theorems that answer these two questions, starting from the latter.

\[\text{FIGURE 1.14 A FA accepting Pascal identifiers.}\]
Finite-State Automata

**Theorem 1.4**
Let \( \mathcal{A} \) be the set of finite-state acceptors that recognize a given language \( L \subseteq I^* \). Let \( \tilde{A} = \langle \tilde{Q}, I, \tilde{\delta}, \tilde{q}_0, \tilde{F} \rangle \) be a minimum-state automaton of \( \mathcal{A} \), that is, such that \(|\tilde{Q}| \leq |Q|\) for each \( A = \langle Q, I, \delta, q_0, F \rangle \). Then \( \tilde{A} \) is unique up to an isomorphism, i.e., a renaming of states.

**Proof**
In order to prove this theorem we need to investigate some interesting algebraic properties of languages and of FA's. A natural equivalence relation \( E_L \) can be associated to any language \( L \subseteq I^* \). \( xE_L y \) holds for any \( x \) and \( y \in I^* \) if and only if either both or neither of \( x \) and \( y \) belong to \( L \). It is immediate to verify that \( E_L \) is actually an equivalence relation on \( I^* \). Now modify \( E_L \) into a new relation \( R_L \) defined as \( xR_L y \) if and only if, for any \( z \) in \( I^* \), either both or neither of \( xz \) and \( yz \) are in \( L \). It is immediate to realize that \( R_L \) is an equivalence relation as well, and that \( R_L \subseteq E_L \), that is, \( R_L \) implies \( E_L \). Furthermore, \( R_L \) is a right congruence with respect to concatenation, that is, \( xR_L y \) implies \( xR_L yz \) for any \( z \in I^* \).

Consider now a FA \( A \) and define the relation \( R_A \) on \( I^* \) as \( xR_A y \) if and only if \( \delta^*(q_0, x) = \delta^*(q_0, y) \), that is, \( x \) and \( y \) lead \( A \) into the same state, when starting from \( q_0 \). It is immediate to realize that \( R_A \) is an equivalence relation and a right congruence as well as \( R_L \). Furthermore, \( xR_A y \) clearly implies \( xR_L(A) y \) since \( A \) reaches the same state for \( x \) as for \( y \), which either is in \( F \) or not. Thus we obtain the following statement.

**Statement 1.5**
For any FA \( A \) the relation \( R_A \) is a refinement of relation \( R_L(A) \), that is, \( R_A \subseteq R_L(A) \). Furthermore since \( R_A \) 's index, that is, its number of equivalence classes, is equal to \(|\tilde{Q}|\), for any language \( L \) accepted by some FA, \( R_L \) is of finite index. In general, \( \text{index}(R_L(A)) \leq \text{index}(R_A) \).

The converse result holds as well.

**Statement 1.6**
Consider any language \( L \) such that \( R_L \) is of finite index. A FA \( A_R = \langle Q_R, I, \delta_R, q_{0R}, F_R \rangle \) can be built such that \( L = L(A_R) \) and \(|Q_R| = \text{index}(R_L)\).

In order to prove the statement we define
- \( Q_R = \{ [x] | x \in I^* \} \)
- \( \delta_R([x], i) = [xi] \) for any \( [x] \in Q_R, i \in I \)
- \( q_{0R} = [\epsilon] \)
- \( F_R = \{ [x] | x \in L \} \)

It is easy to verify that
- The above definition is consistent as if \( [x] = [y] \) for some \( x \) and \( y \in I^* \), \( \delta_R([x], i) = \delta_R([y], i) \) for any \( i \), since \( R_L \) is a right congruence and either both \( [x], [y] \) are in \( F_R \) or neither.
- \( L(A_R) = L \), since \( \delta_R^*([\epsilon], x) = [x] \), which is in \( F_R \) if and only if \( x \in L \).
The collection of Statements 1.5 and 1.6 is also known as the Myhill–Nerode theorem.

At this point we can complete the proof of Theorem 1.4 by observing that $A_R$ is the minimum state automaton $\bar{A}$, up to a suitable renaming. In fact, let $\bar{q}$ be a state of $\bar{A}$. There must exist an $x$ such that $\delta^*(\bar{q}_0, x) = \bar{q}$ since otherwise $\bar{q}$ could be removed from $\bar{Q}$ without altering $L(\bar{A})$, thus contradicting the minimality of $\bar{A}$. Thus rename $\bar{q}$ as $[x]$ for such an $x$ and verify the consistency of the definition through the fact that if $\delta^*(\bar{q}_0, x) = \bar{q} = \delta^*(\bar{q}_0, y)$ then $xR_{\bar{A}}y$ and therefore $xR_Ly$ because of Statement 1.5. Thus $[x] = [y]$.

The minimality of $\bar{A} = A_R$ follows then from the fact that $|Q_R| = \text{index}(R_L) \leq \text{index}(R_A) = |Q|$ for any FA $A$ recognizing $L$.

The FA $A_R$ is called the canonical acceptor recognizing $L$ since it is unique up to a renaming of its states. Note, however, that the proof of Theorem 1.4 states the existence of a minimal automaton without providing an explicit procedure to build it. In order to get a constructive proof of Theorem 1.4 we can proceed as follows. Let $A$ be any FA.

i. First, eliminate all useless states from $Q$. A useless state $q$ is such that either there is no $x$ such that $\delta^*(q_0, x) = q$ or there is no $y$ such that $\delta^*(q, y) \in F$.

It is an easy exercise to derive an algorithm that performs the elimination.

ii. Then define an equivalence relation $D$ on $Q$ as $D = D'$ if and only if for any $x$ either both or neither of $\delta^*(q, x), \delta^*(q', x)$ are in $F$. $D$ can also be easily tested by means of an algorithm since, as usual, if for some $q$ and $q'$, an $x$ exists for which $\delta^*(q, x) \in F, \delta^*(q', x) \notin F$, then also an $x'$ exists with the same property and with $|x'| < |Q|.$

iii. At this point, define the automaton $A' = \langle Q', I, \delta', q_0', F' \rangle$ with $Q' = \{[q]\}$, that is, the set of equivalence classes of $Q$ with respect to $D, \delta'([q], a) = [\delta(q, a)],$ and $F' = \{[q] | q \in F\}$.

As a simple exercise, one can verify that

- $A'$ is well defined as the definition of $\delta'$ does not depend on the choice of the particular $q$ in $[q]$.
- $L(A') = L(A)$
- $Q' \leq \text{index}(R_L(A))$. In fact, by contradiction suppose that $Q' > \text{index}(R_L(A)).$ Then, since by Statement 1.5 $R_A$ is a refinement of $R_L(A)$, there should exist $[q] \neq [q']$ and $x \neq y$, such that $xR_{L(A)}y$ and $\delta^*([q_0], x) = [q], \delta^*([q_0], y) = [q']$. However, supposing $[q] \neq [q']$ (i.e., $D'$ does not hold) there should exist a $w$ such that $\delta^*(q_0, xw) = \delta^*(q, w) \in F$ and $\delta^*(q_0, yw) \notin F$, or vice-versa. But this contradicts $xR_{L(A)}y$ because $R_L$ is a right congruence.

**Example 1.8**

Consider the FA of Figure 1.14. It is immediate to realize that $q_1Dq_2$ since $\delta^*(q_1, x) \in F$ as well as $\delta^*(q_2, x)$ for any $x \in \{A, \ldots, Z, 0, 1, \ldots, 9\}$. However, $q_0Dq_1$ does not hold because $\delta^*(q_0, \epsilon) \notin F$ while $\delta^*(q_1, \epsilon) \in F$. Thus, by applying the above $\alpha$ coincides with the FA...
applying the above construction, we obtain the FA of Figure 1.15, which coincides with the FAs of Figure 1.7 up to a renaming of states.

**Exercises**

1.11 Minimize the FA subjacent to the FT of Figure 1.10.
1.12 Minimize the FA of Figure 1.16.
1.13 Devise a procedure to minimize FTs and build a program implementing it.

1.14 Minimize the FT of Figure 1.10.

**Corollary 1.7**
Given two FAs $A_1$ and $A_2$, $L(A_1) = L(A_2)$ if and only if their associated minimal automata $A_1^*$ and $A_2^*$ are identical up to a renaming of states. □

Theorem 1.4 and Corollary 1.7 are examples that show how the application of mathematical properties of a model can provide results of practical interest.

The following theorem shows that the class of languages accepted by FAs is closed under set operations; that is, the combination of languages in the class yields a language in the same class. Additional closure properties will be proved later.

**Theorem 1.8**
The class of languages accepted by finite-state automata is closed under

a. Intersection.
b. Complement with respect to $I^*$.
c. Union.

**Outline of the proof**

a. Intersection.

Let $A_1 = \langle Q_1, I_1, \delta_1, q_{01}, F_1 \rangle$ and $A_2 = \langle Q_2, I_2, \delta_2, q_{02}, F_2 \rangle$ be two FAs. Assume that $I_1 = I_2 = I$ and $Q_1, Q_2$ are disjoint. This does not cause any loss of generality since we can always let $I = I_1 \cup I_2$, and suitably rename the states of $A_1$ obtaining two automata equivalent to $A_1$ and $A_2$, respectively, and satisfying the assumption. The FA $A = \langle Q, I, \delta, q_0, F \rangle$ accepting $L(A_1) \cap L(A_2)$ is constructed as follows.

- $Q = Q_1 \times Q_2$
- $q_0 = \langle q_{01}, q_{02} \rangle$
- $F = \{ \langle q', q'' \rangle \mid q' \in F_1, q'' \in F_2 \}$
- $\delta(\langle q_1, q_2 \rangle, a) = \langle q'_1, q'_2 \rangle$ if and only if $\delta_1(q_1, a) = q'_1, \delta_2(q_2, a) = q'_2$

b. Complement.

Let $A = \langle Q, I, \delta, q_0, F \rangle$ be a FA. First, construct a FA $A'$ by adding a new state $\overline{q}$ to $A$ such that the next-state function of $A'$ leads to $\overline{q}$ whenever it is undefined in $A$. Furthermore, it remains in $\overline{q}$ for every input symbol. The FA accepting $L(A) = I^* - L(A)$ is identical to $A'$, except that final and non-final states are interchanged.

c. Union.

Let $A_1, A_2$ be two FAs. Then $L = L(A_1) \cup L(A_2) = \overline{\overline{L(A_1)}} \cap \overline{\overline{L(A_2)}}$. Thus, according to (a) and (b), $L$ is accepted by a FA.
by (11.4) or (11.7) might have a large number of states, it follows that the DFA constructed by (11.8) might have a large number of states. In the next few paragraphs we'll see that any DFA can be automatically transformed into a DFA with the minimum number of states. Then, putting everything together, we'll have an automatic process for constructing the most efficient DFA for any regular expression.

11.3.3 Minimum-State DFAs

One way to try and simplify the DFA for some regular expression is to algebraically transform the regular expression into a simpler one before starting construction of the DFA. For example, from the properties (11.1) we have

\[ \lambda + a + aaa^* = a^*. \]

If we use our wits, most of us can construct a simpler DFA for \( a^* \) than for \( \lambda + a + aaa^* \). If we use the algorithms, we can also obtain a simpler DFA for \( a^* \) than for \( \lambda + a + aaa^* \). But we still might not have obtained the simplest DFA.

It's nice to know that no matter what DFA we come up with, we can always transform it into a DFA with the minimum number of states that recognizes the same language. The basic result is given by the following theorem, which is named after Myhill [1957] and Nerode [1958]:

**Theorem**

Every regular expression has a unique minimum-state DFA.  \( (11.9) \)

The word "unique" in (11.9) means that the only difference that can occur between any two minimum-state DFAs for a regular expression is not in the number of states, but rather in the names given to the states. So we could rename the states of one DFA so that it becomes the same as the other DFA.

We already know how to transform a regular expression into an NFA and then into a DFA. Now let's see how to transform a DFA into a minimum-state DFA. The key idea is to define two states \( s \) and \( t \) to be *equivalent* if for every string \( w \), the transitions

\[ T(s, w) \quad \text{and} \quad T(t, w) \]

are either both final or both nonfinal.

In other words, to say that \( s \) and \( t \) are equivalent means that whenever the execution of the DFA reaches either \( s \) or \( t \) with the same string \( w \) left to consume, then the DFA will consume \( w \) and, in either case, enter the same type of state—either both reject or both accept.

It's easy to see that equivalence is an equivalence relation. Stop and check it out. So once we know the equivalent pairs of states, we can partition the states of the DFA into a collection of subsets, where each subset contains states that are equivalent to each other. These subsets become the states of the new minimum-state DFA.
Before we present the algorithm, let’s try to make the idea more precise with a very simple example. Suppose we’re given the four-state DFA in Figure 11.7, which is represented in graphical form and as a transition table.

It’s pretty easy to see that states 1 and 2 are equivalent. For example, if the DFA is in either state 1 or 2 with any string starting with $a$, then the DFA will consume $a$ and enter state 3. From this point the DFA stays in state 3. On the other hand, if the DFA is in either state 1 or 2 with any string starting with $b$, then the DFA will consume $b$ and it will stay in state 1 or 2 as long as $b$ is present at the beginning of the resulting string. So for any string $w$, both $T(1, w)$ and $T(2, w)$ are either both final or both nonfinal.

It’s also pretty easy to see that no other distinct pairs of states are equivalent. For example, states 0 and 1 are not equivalent because $T(0, a) = 1$, which is a reject state, and $T(1, a) = 3$, which is an accept state. Since the only distinct equivalent states are 1 and 2, we can partition the states of the DFA into the subsets

$$\{0\}, \{1, 2\}, \text{ and } \{3\}.$$  

These three subsets form the states of the minimum-state DFA. This minimum-state DFA can be represented by either one of the two forms shown in Figure 11.8.

**Figure 11.7** DFA as graph and table.

<table>
<thead>
<tr>
<th></th>
<th>$T$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>final</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 11.8** Minimum-state DFA.
Partitioning the States

There are several methods to compute the equivalence relation and its corresponding partition. The method that we'll present is easy to understand, and it can be programmed. We start the process by forming the set

\[ E_0 \]

of distinct pairs of the form \( \{s, t\} \), where \( s \) and \( t \) are either both final or both nonfinal. This collection contains the possible equivalent pairs.

Next we construct a new collection \( E_1 \) from \( E_0 \) by throwing away any pair \( \{s, t\} \) if there is some letter \( a \) such that \( \{T(s, a), T(t, a)\} \) is a distinct pair that does not occur in \( E_0 \). This means that the pair \( \{T(s, a), T(t, a)\} \) contains two states of different types. So we must throw \( \{s, t\} \) away.

The process continues by constructing a new collection \( E_2 \) from \( E_1 \) by throwing away \( \{s, t\} \) if there is some letter \( a \) such that \( \{T(s, a), T(t, a)\} \) is a distinct pair that does not occur in \( E_1 \). This means that there is a string of length 2 such that the DFA, if started from either \( s \) or \( t \), consumes the string and enters two different types of states. So we must throw \( \{s, t\} \) out of \( E_1 \).

We continue the process by constructing a descending sequence

\[ E_0 \supseteq E_1 \supseteq E_2 \supseteq \cdots \supseteq E_n \supseteq \cdots . \]

Each set \( E_n \) in the sequence has been constructed to have the property that for each pair \( \{s, t\} \) in \( E_n \) and for any string of length less than or equal to \( n \), the DFA, if started from either \( s \) or \( t \), will consume the string and enter the same type of states—either both reject or both accept. Since \( E_0 \) is a finite set, the sequence of sets must eventually stop with some set \( E_k \) such that

\[ E_{k+1} = E_k . \]

This means \( E_k \) is the desired set of equivalent pairs of states, because for any pair \( \{s, t\} \) in \( E_k \) and any length string, the DFA, if started from either \( s \) or \( t \), will consume the string and enter the same type of states—either both reject or both accept.

For example, from the DFA in Figure 11.7 we start with

\[ E_0 = \{ \{0, 1\}, \{0, 2\}, \{1, 2\} \} \]

To construct \( E_1 \) from \( E_0 \), we throw away \( \{0, 1\} \) because

\[ \{T(0, a), T(1, a)\} = \{1, 3\}, \]

which is not in \( E_0 \). We must also throw away \( \{0, 2\} \) because

\[ \{T(0, a), T(2, a)\} = \{1, 3\}, \]
which is not in $E_0$. This leaves us with the set

$$E_1 = \{(1, 2)\}.$$

We can't throw away any pairs from $E_1$. Therefore, $E_2 = E_1$, which says that the desired set of equivalent pairs is $E_1 = \{(1, 2)\}$. Notice that $\{1, 2\}$ is a state in the minimum-state DFA shown in Figure 11.8.

Now we're ready to present the actual algorithm to transform a DFA into a minimum-state DFA.

### Algorithm to Construct a Minimum-State DFA (11.10)

Given: A DFA with set of states $S$ and transition table $T$. Assume that all states that cannot be reached from the start state have already been thrown away.

Output: A minimum-state DFA recognizing the same regular language as the input DFA.

1. Construct the equivalent pairs of states by calculating the descending sequence of sets of pairs $E_0 \supseteq E_1 \supseteq \cdots$ defined as follows:

   $$E_0 = \{\{s, t\} \mid s \text{ and } t \text{ are distinct and either both states are final or both states are nonfinal}\}.$$

   $$E_{i+1} = \{\{s, t\} \mid \{s, t\} \in E_i \text{ and for every } a \in A \text{ either } T(s, a) = T(t, a) \text{ or } \{T(s, a), T(t, a)\} \in E_i\}.$$

   The computation stops when $E_k = E_{k+1}$ for some index $k$. $E_k$ is the desired set of equivalent pairs.

2. Use the equivalence relation generated by the pairs in $E_k$ to partition $S$ into a set of equivalence classes. These equivalence classes are the states of the new DFA.

3. The start state is the equivalence class containing the start state of the input DFA.

4. A final state is any equivalence class containing a final state of the input DFA.

5. The transition table $T_{\text{min}}$ for the minimum-state DFA is defined as follows, where $[s]$ denotes the equivalence class containing $s$ and $a$ is any letter: $T_{\text{min}}([s], a) = [T(s, a)]$.
Example 11.17  A Minimum-State DFA Construction

We'll compute the minimum-state DFA for the following DFA.

![DFA Diagram]

The set of states is \( S = \{0, 1, 2, 3, 4\} \). For Step 1 we'll start by calculating \( E_0 \) as the set of pairs \( \{s, t\} \), where \( s \) and \( t \) are both final or both nonfinal:

\[
E_0 = \{\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}\}.
\]

To calculate \( E_1 \) we throw away \( \{0, 3\} \) because \( T(\{0, b\}, T(3, b)) = \{1, 4\} \), which is not in \( E_0 \). We also throw away \( \{1, 3\} \) and \( \{2, 3\} \). That leaves us with

\[
E_1 = \{\{0, 1\}, \{0, 2\}, \{1, 2\}\}.
\]

To calculate \( E_2 \) we throw away \( \{0, 2\} \) because \( T(\{0, a\}, T(2, a)) = \{2, 3\} \), which is not in \( E_1 \). That leaves us with

\[
E_2 = \{\{1, 2\}\}.
\]

To calculate \( E_3 \) we don't throw anything away from \( E_2 \). So we stop with

\[
E_3 = E_2 = \{\{1, 2\}\}.
\]

So the only distinct equivalence pair is \( \{1, 2\} \). Therefore, the set \( S \) of states is partitioned into the following four equivalence classes:

\[
\{0\}, \{1, 2\}, \{3\}, \{4\}.
\]

These are the states for the new DFA. The start state is \( 0 \), and the final state is \( 4 \). Using equivalence class notation we have

\[
[0] = \{0\}, \quad [1] = [2] = \{1, 2\}, \quad [3] = \{3\}, \quad \text{and} \quad [4] = \{4\}.
\]

Thus we can apply Step 5 to construct the table for \( T_{\text{min}} \). For example, we'll compute \( T_{\text{min}}(\{0\}, a) \) and \( T_{\text{min}}(\{1, 2\}, b) \) as follows:

\[
T_{\text{min}}(\{0\}, a) = T_{\text{min}}(\{0\}, a) = [T(0, a)] = [2] = \{1, 2\},
\]

\[
T_{\text{min}}(\{1, 2\}, b) = T_{\text{min}}(\{1\}, b) = [T(1, b)] = [1] = \{1, 2\}.
\]
Similar computations yield the table for $T_{\text{min}}$, which is listed as follows:

<table>
<thead>
<tr>
<th>$T_{\text{min}}$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start ${0}$</td>
<td>${1, 2}$</td>
<td>${1, 2}$</td>
</tr>
<tr>
<td>${1, 2}$</td>
<td>${3}$</td>
<td>${1, 2}$</td>
</tr>
<tr>
<td>${3}$</td>
<td>${3}$</td>
<td>${4}$</td>
</tr>
<tr>
<td>final ${4}$</td>
<td>${4}$</td>
<td>${4}$</td>
</tr>
</tbody>
</table>

We can simplify the table by assigning a single number to each state. For example, assign 0, 1, 2, and 3 to the states $\{0\}$, $\{1, 2\}$, $\{3\}$, and $\{4\}$. Then the preceding table can be written in the following familiar form.

<table>
<thead>
<tr>
<th>$T_{\text{min}}$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start 0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>final 3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Be sure to draw a picture of this minimum-state DFA.

**Example 11.18 A Minimum-State DFA Construction**

We'll compute the minimum-state DFA for the following DFA.

<table>
<thead>
<tr>
<th>$T$</th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start 0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>final 4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>final 5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

The set of states is $S = \{0, 1, 2, 3, 4, 5\}$. For Step 1 we get the following sequence of relations:

$E_0 = \{\{0, 1\}, \{0, 2\}, \{0, 3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{4, 5\}\}$,

$E_1 = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{4, 5\}\}$,

$E_2 = E_1$.

Therefore, the equivalence relation is generated by the four equivalent pairs $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$, and $\{4, 5\}$. Thus we obtain a partition of $S$ into the following three equivalence classes:

$\{0\}$, $\{1, 2, 3\}$, $\{4, 5\}$. 
These are the states for the new DFA. The start state is \{0\}, and the final state is \{4, 5\}. Using the standard notation for equivalence classes, we have

\[ [0] = \{0\}, \quad [1] = [2] = [3] = \{1, 2, 3\}, \quad \text{and} \quad [4] = [5] = \{4, 5\}. \]

So we can apply Step 5 to construct the table for \( T_{\text{min}} \). For example, we can compute \( T_{\text{min}}(\{1, 2, 3\}, b) \) and \( T_{\text{min}}(\{4, 5\}, a) \) as follows:

\[
T_{\text{min}}(\{1, 2, 3\}, b) = T_{\text{min}}([1], b) = [T(1, b)] = [1] = \{1, 2, 3\},
\]

\[
T_{\text{min}}(\{4, 5\}, a) = T_{\text{min}}([4], a) = [T(4, a)] = [4] = \{4, 5\}.
\]

Similar computations will yield the table for \( T_{\text{min}} \). We’ll leave these calculations as an exercise.

Exercises

Construction Algorithms

1. Use (11.7) to build an NFA for each of the following regular expressions.
   a. \( a^* b^* \)
   b. \( (a + b)^* \)
   c. \( a^* + b^* \)

2. Construct a DFA table for the following NFA in two ways using (11.8):

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( \Lambda )</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0 ( \emptyset )</td>
<td>{1, 2}</td>
<td>{1}</td>
</tr>
<tr>
<td></td>
<td>1 {2}</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>final</td>
<td>2 ( \emptyset )</td>
<td>{2}</td>
<td>{1}</td>
</tr>
</tbody>
</table>

   a. Take unions of lambda closures of the NFA entries.
   b. Take lambda closures of unions of the NFA entries.

3. Suppose we are given the following NFA over the alphabet \( \{a, b\} \):

   ![NFA Diagram]

   a. Find a regular expression for the language accepted by the NFA.
   b. Write down the transition table for the NFA.
   c. Use (11.8) to transform the NFA into a DFA.
   d. Draw a picture of the resulting DFA.