5. REGULAR AND NONREGULAR LANGUAGES

5.1 A Criterion for Regularity

- Suppose that a language over Σ is specified in some way that involves neither a regular expression nor an FA.

  How can we tell whether the language is regular?
  If the language is regular, how can we find either a regular expression describing it or an FA accepting it?

- Theorem 3.2 provides a partial answer to the first question: If there are infinitely many strings that are "pairwise distinguishable" with respect to L, then L cannot be regular.

- It is useful to reformulate this condition. According to Definition 3.5, L/x denotes the set $\{y \in \Sigma^* \mid xy \in L\}$. Let I_L be the indistinguishability relation on Σ^*, defined by

  $x\ I_L\ y$ if and only if $L/x = L/y$

- It is possible to show that I_L is an equivalence relation on Σ^*. If two strings are distinguishable with respect to L, they are in different equivalence classes of I_L.

- Theorem 3.2 implies that if L is regular, then the set of equivalence classes for the relation I_L is finite. It turns out that the converse is also true: if the set of equivalence classes is finite, then L is regular.

- This result has some useful consequences:

  It provides a way to test a language L is regular.
  If L turns out to be regular, the equivalence classes of I_L can be used to construct an FA that accepts L.
  This FA is the most natural one to accept L, in that it has the fewest possible states.

  A by-product of the result will be a method for taking any FA and simplifying it as much as possible.

A Criterion for Regularity (Continued)

- Let L be a regular language that is accepted by the FA $M = (Q, \Sigma, q_0, A, \delta)$. For each state $q \in Q$, let

  $L_q = \{x \in \Sigma^* \mid \delta^*(q_0, x) = q\}$

- There are two natural partitions of Σ^* formed by L and M: the one defined by the equivalence relation I_L and the one formed by the sets L_q for $q \in Q$.

- Lemma 3.1 describes the relationship between the two partitions: If x and y are in the same L_q (in other words, if $\delta^*(q_0, x) = \delta^*(q_0, y)$), then $L/x = L/y$, so that x and y are in the same equivalence class of I_L.

- Each set L_q must be a subset of a single equivalence class, and therefore that every equivalence class of I_L is the union of one or more of the L_q's.

- There can be no fewer of the L_q's than there are equivalence classes of I_L. If the two numbers are the same, then each set L_q is precisely one of the equivalence classes of I_L, and M is an FA with the fewest possible states recognizing L.

- This suggests that an FA could be constructed for L by letting each state be a set of strings (one of the equivalence classes of I_L).

- Filling in the remaining details is surprisingly easy:

  One of the strings that cause an FA to be in the initial state is Λ, so the initial state will be the equivalence class containing Λ.

  So that the FA will accept L, the accepting states will be those equivalence classes containing elements of L.

  The value of the transition function will be computed by taking a string in the present state (equivalence class) and concatenating it with the input symbol. The resulting string determines the new equivalence class.

A Criterion for Regularity (Continued)

- There is a potential problem with the definition of the transition function. Consider the transition from the equivalence class containing the string x on input symbol a:

  $\delta([x], a) = [xa]$

  If y is another string in $[x]$, then it must also be the case that

  $\delta([y], a) = [ya]$

  However, it is not clear that $[xa]$ and $[ya]$ are the same state. If they are not, then the δ function is not well-defined. The following lemma shows that this potential problem never arises.

- **Lemma 5.1.** I_L is *right invariant* with respect to concatenation. In other words, for any $x, y \in \Sigma^*$ and any $a \in \Sigma$, if $x\ I_L\ y$, then $xa\ I_L\ ya$. Equivalently, if $[x] = [y]$, then $[xa] = [ya]$.

  Proof: Suppose $x\ I_L\ y$ and $a \in \Sigma$. Then $L/x = L/y$, so that for any $z' \in \Sigma^*$, $xz'$and yz' are either both in L or both not in L. Therefore, for any $z \in \Sigma^*$, xaz and yaz are either both in L or both not in L (because the previous statement can be applied with $z' = az$), and thus $xa\ I_L\ ya$.

A Criterion for Regularity (Continued)

- **Theorem 5.1.** Let $L \subseteq \Sigma^*$, and let Q_L be the set of equivalence classes of the relation I_L on Σ^*. (Each element of Q, therefore, is a set of strings.) If Q_L is a finite set, then $M_L = (Q_L, \Sigma, q_0, A_L, \delta)$ is a finite automaton accepting L, where $q_0 = [\Lambda]$, $A_L = \{q \in Q_L \mid q \cap L \neq \varnothing\}$, and $\delta : Q_L \times \Sigma \rightarrow Q_L$ is defined by the formula $\delta([x], a) = [xa]$. Furthermore, M_L has the fewest states of any FA accepting L.

  Proof: According to Lemma 5.1, the formula $\delta([x], a) = [xa]$ is a meaningful function definition and thus the 5-tuple M_L is a valid FA. In order to verify that M_L recognizes L, we need the formula

  $\delta^*([x], y) = [xy]$

  for $x, y \in \Sigma^*$. The proof is by structural induction on y. The basis step is to show that $\delta^*([x], \Lambda) = [x]$ for every x. The left side is $[x]$ because of the definition of δ^* in an FA (Definition 3.3), and the right side is $[x]$ because $x\Lambda = x$.

  For the induction step, suppose that for some y, $\delta^*([x], y) = [xy]$ for every string x, and consider $\delta^*([x], ya)$ for $a \in \Sigma$:

  $$\begin{aligned} \delta^*([x], ya) &= \delta(\delta^*([x], y), a) \quad \text{(by definition of } \delta^*) \\ &= \delta([xy], a) \quad \text{(by the induction hypothesis)} \\ &= [xya] \quad \text{(by definition of } \delta) \end{aligned}$$

  From this formula it follows that $\delta^*(q_0, x) = \delta^*([\Lambda], x) = [x]$. The definition of A_L says, therefore, that x is accepted by M_L if and only if $[x] \cap L \neq \varnothing$. But the two statements $[x] \cap L \neq \varnothing$ and $x \in L$ are the same. One direction is obvious: If $x \in L$, then $[x] \cap L \neq \varnothing$, since $x \in [x]$. In the other direction, if $[x]$ contains an element y of L, then x must be in L. Otherwise the string Λ would distinguish x and y with respect to L, and x and y could not both be elements of $[x]$. Therefore, M_L accepts L.

  Finally, if there are n equivalence classes of I_L, then we can get a set of n strings that are pairwise distinguishable by choosing one string from each equivalence class. Theorem 3.2 implies that any FA accepting L must have at least n states. Since M_L has exactly n, it has the fewest possible.

A Criterion for Regularity (Continued)

- **Corollary 5.1.** L is a regular language if and only if the set of equivalence classes of I_L is finite.

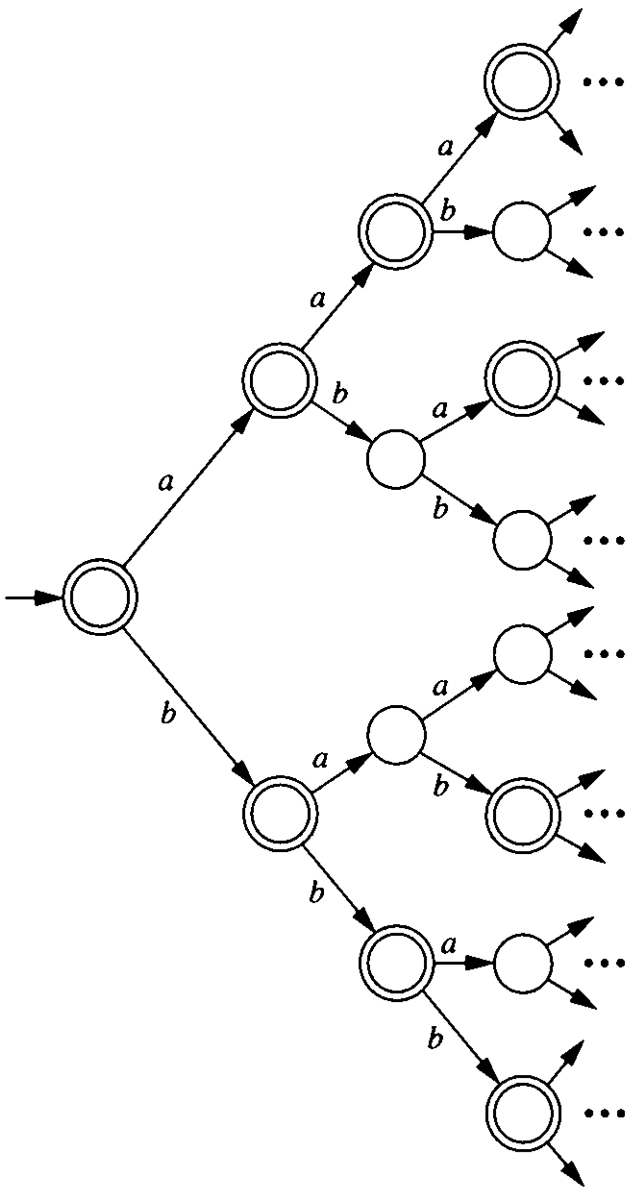
  Proof: Theorem 5.1 says that if the set of equivalence classes is finite, there is an FA accepting L; and Theorem 3.2 says that if the set is infinite, there can be no such FA.

- Corollary 5.1 was proved by Myhill and Nerode, and it is often called the Myhill-Nerode theorem.

A Criterion for Regularity (Continued)

- The construction of M_L in Theorem 5.1 can be carried out for any language L, even one that is not regular. However, if L is not regular, then Q_L will not be finite and M_L will not be an FA.

- If L is the language *pal* of all palindromes over $\{a, b\}$, each equivalence class in Q_L contains exactly one string, giving M_L the following appearance:



Applying the Myhill-Nerode Theorem

- **Example 5.1:** Applying Theorem 5.1 to {0, 1}*{10}

  Let L be the following language from Example 3.12:

  $L = \{x \in \{0, 1\}^* \mid x \text{ ends with } 10\}$

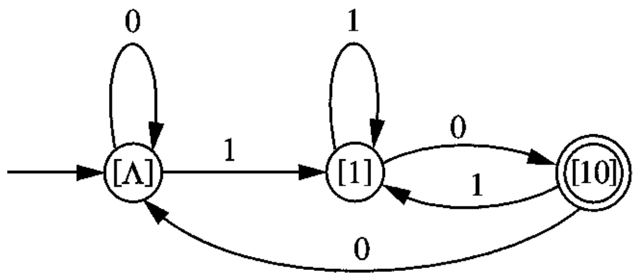
  Any two of the strings Λ, 1, and 10 are distinguishable with respect to L: The string Λ distinguishes Λ and 10, and also 1 and 10, while the string 0 distinguishes Λ and 1. Therefore, the equivalence classes [Λ], [1], and [10] are distinct.

  Any string y is equivalent to one of these strings. If y ends in 10, then y is equivalent to 10; if y ends in 1, y is equivalent to 1; otherwise (if $y = \Lambda$, $y = 0$, or y ends with 00), y is equivalent to Λ. Therefore, these three equivalence classes are the only ones.

  Let $M_L = (Q_L, \Sigma, q_0, A_L, \delta)$ be the FA constructed in Theorem 5.1. Then

$$\begin{array}{rcl} \delta([\Lambda], 0) = [\Lambda] & \text{and} & \delta([\Lambda], 1) = [1] \\ \delta([1], 0) = [10] & \text{and} & \delta([\Lambda], 1) = [1] \\ \delta([10], 0) = [\Lambda] & \text{and} & \delta([10], 1) = [1] \end{array}$$

  The transition diagram for M_L:



Applying the Myhill-Nerode Theorem (Continued)

- Theorem 3.2 (the “only if” part of Theorem 5.1) was previously used to show that the language of palindromes over {0, 1} is nonregular. It can be used to exhibit a number of other nonregular languages.

- **Example 5.2:** The Equivalence Classes of I_L for $L = \{0^n1^n \mid n > 0\}$

  Let $L = \{0^n1^n \mid n > 0\}$. Using Theorem 5.1 to prove that L is not regular is done by showing that there are infinitely many distinct equivalence classes of I_L. In fact, it is not hard to describe the equivalence classes exactly:

  1. *The set of all strings that are not prefixes of any elements of L.* Examples include 1, 011, and 010.

  2. *The set of all strings in L.* For any string $x \in L$, Λ is the only string that can follow x so as to produce an element of L.

  3. *For each $i \geq 0$, the set* $\{0^i\}$. If $i \neq j$, the strings 0^i and 0^j are distinguished by the string 1^i, because $0^i1^i \in L$ and $0^j1^i \notin L$. For any string x other than 0^i, the string 01^{i+1} distinguishes 0^i and x (because $0^i01^{i+1} \in L$ and $x01^{i+1} \notin L$).

  4. *For each $k > 0$, the set* $\{0^{j+k}1^j \mid j > 0\}$. All strings of the form $0^{j+k}1^j$ for a particular value of k are equivalent, because there is exactly one string z for which $0^{j+k}1^jz \in L$: the string $z = 1^k$. No string other than one of these can be equivalent to $0^{j+k}1^j$.

  Since there are infinitely many distinct equivalence classes, L is not regular.

Applying the Myhill-Nerode Theorem (Continued)

- **Example 5.3:** Simple Algebraic Expressions

  Let L be the set of all legal algebraic expressions involving the identifier a, the operator +, and left and right parentheses. The string

  $((\ldots(a)\ldots))$

  is in L if and only if the numbers of left and right parentheses are the same. Consider the set $S = \{(^n \mid n \geq 0\}$. For $0 \leq m < n$, the string $a)^m$ distinguishes $(^m$ and $(^n$, and so any two elements of S are distinguishable with respect to L. Therefore L is not regular.

- **Example 5.4:** The Set of Strings of the Form *ww*

  For another example where the set $S = \{0^n \mid n \geq 0\}$ can be used to prove a language nonregular, let L be the language

  $\{ww \mid w \in \{0, 1\}^*\}$

  The string $z = 1^n 0^n 1^n$ distinguishes 0^n and 0^m when $m \neq n$, because $0^n z$ is in L and $0^m z$ is not.

- **Example 5.5:** Another Nonregular Language from Theorem 5.1

  Let $L = \{0, 011, 011000, 0110001111, \ldots\}$. A string in L consists of groups of 0's alternated with groups of 1's. It begins with a single 0, and each subsequent group of identical symbols is one symbol longer than the previous group.

  The infinite set L itself can be shown to be pairwise distinguishable with respect to L, and therefore that L is not regular. Let x and y be two distinct elements of L. Suppose x and y both end with groups of 0's, for example, x with 0^j and y with 0^k. Then $x1^{j+1} \in L$, but $y1^{j+1} \notin L$. Similar arguments work in the other three cases.

5.2 Minimal Finite Automata

- Finding a minimum-state FA for a language L is easy to do once the equivalence classes of I_L have been identified. Unfortunately, it is not clear how to find these classes in general.

- If L is already described by an FA, however, then there is an algorithm for modifying it so that the resulting machine has the fewest possible states. These states correspond exactly to the equivalence classes of I_L.

- Suppose that L is accepted by the finite automaton $M = (Q, \Sigma, q_0, A, \delta)$. Consider the two partitions of Σ^* described earlier, one in which the subsets are the sets L_q and one in which the subsets are the equivalence classes of I_L. If the two partitions are the same, then M is already a minimum-state FA.

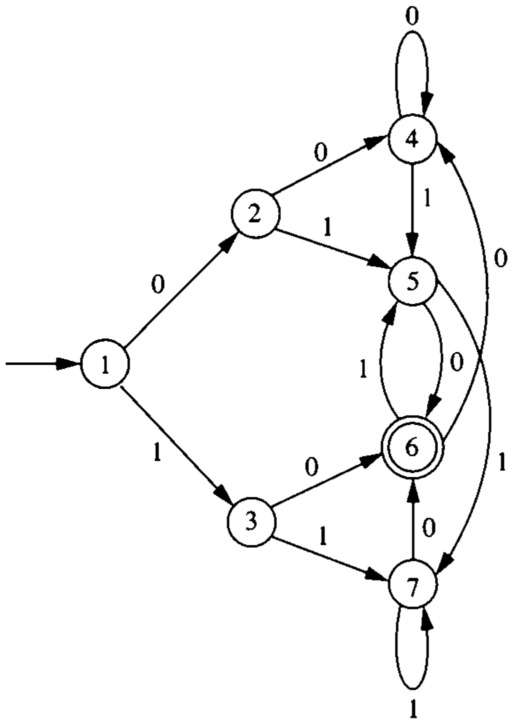
- If not, then the first partition is finer than the second (one subset from the second partition might be the union of several from the first). All that is necessary is to determine which sets L_q can be combined to obtain an equivalence class.

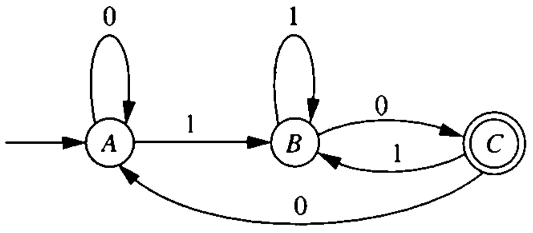
- Before proceeding to this step, however, M should be modified by eliminating the states q for which $L_q = \varnothing$ (states that are unreachable from q_0).

- It is easy to formulate a recursive definition of the set of reachable states of M and then use that definition to obtain an algorithm that finds all reachable states. If all the others are eliminated, the resulting FA still recognizes L.

Minimal Finite Automata (Continued)

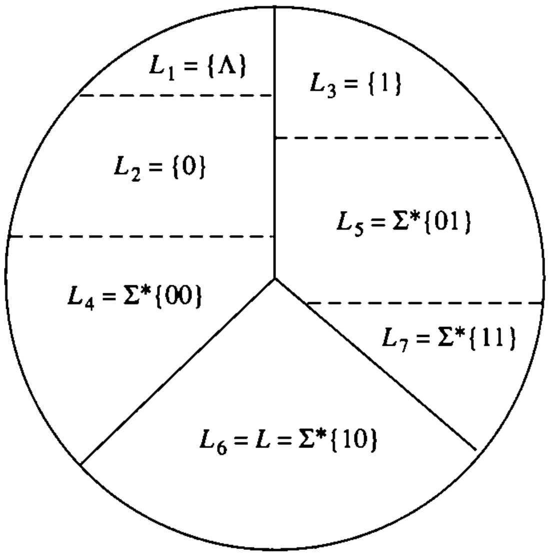
- The original FA for the language in Example 3.12 appears on the left. On the right is the corresponding minimum-state FA from Example 5.1.

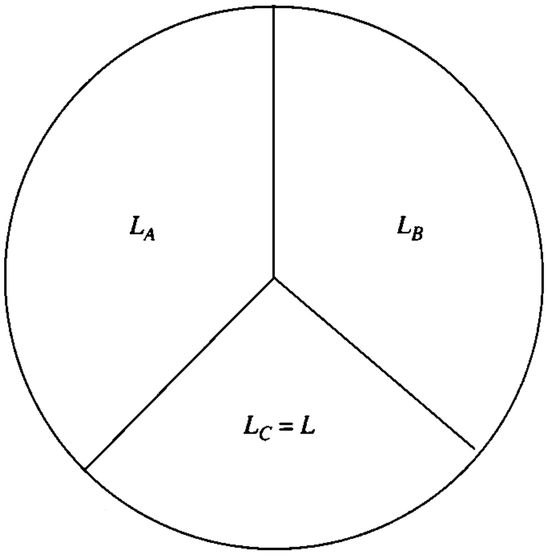




(*b*)

The partition corresponding to the original FA is shown on the left. The right figure shows the equivalence classes of I_L (the sets L_q for the minimum-state FA).





Minimal Finite Automata (Continued)

- The problem of minimizing the number of states in the FA M involves identifying the *pairs* (p, q) of states for which L_p and L_q are subsets of the same equivalence class (written $p \equiv q$).

- The algorithm for minimizing the number of states in an FA M actually solves the opposite problem: identify those pairs (p, q) for which $p \not\equiv q$.

- The following lemma expresses the statement $p \equiv q$ in a slightly different way.

- **Lemma 5.2.** Suppose $p, q \in Q$, and x and y are strings with $x \in L_p$ and $y \in L_q$ (in other words, $\delta^*(q_0, x) = p$ and $\delta^*(q_0, y) = q$). Then these three statements are all equivalent:

  1. $p \equiv q$.
  2. $L/x = L/y$ (i.e., $x\ I_L\ y$, or x and y are indistinguishable with respect to L).
  3. For any $z \in \Sigma^*$, $\delta^*(p, z) \in A \Leftrightarrow \delta^*(q, z) \in A$ (i.e., $\delta^*(p, z)$ and $\delta^*(q, z)$ are either both in A or both not in A).

  Proof: To see that statements 2 and 3 are equivalent, consider the formulas

  $$\delta^*(p, z) = \delta^*(\delta^*(q_0, x), z) = \delta^*(q_0, xz)$$
  $$\delta^*(q, z) = \delta^*(\delta^*(q_0, y), z) = \delta^*(q_0, yz)$$

  Saying that $L/x = L/y$ means that a string z is in one set if and only if it is in the other, or that $xz \in L$ if and only if $yz \in L$; since M accepts L, this is exactly the same as statement 3.

  Now if statement 1 is true, then L_p and L_q are both subsets of the same equivalence class. This means that x and y are equivalent, which is statement 2. The converse is also true, because if L_p and L_q are not both subsets of the same equivalence class, then they are subsets of different equivalence classes, so that statement 2 does not hold.

Minimal Finite Automata (Continued)

- According to the lemma, if $p \not\equiv q$, then for some z, exactly one of the two states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A. The simplest way this can happen is with $z = \Lambda$, so that only one of the states p and q is in A.

- Once a pair (p, q) with $p \not\equiv q$ has been found, suppose that $r, s \in Q$, and for some $a \in \Sigma$, $\delta(r, a) = p$ and $\delta(s, a) = q$. Then

  $$\delta^*(r, az) = \delta^*(\delta^*(r, a), z) = \delta^*(\delta(r, a), z) = \delta^*(p, z)$$

  and similarly, $\delta^*(s, az) = \delta^*(q, z)$. Since $p \not\equiv q$, then for some z, exactly one of the states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A; therefore, exactly one of $\delta^*(r, az)$ and $\delta^*(s, az)$ is in A, and $r \not\equiv s$.

- These observations suggest the following recursive definition of a set S, which will turn out to be the set of all pairs (p, q) with $p \not\equiv q$.

  1. For any p and q for which exactly one of p and q is in A, (p, q) is in S.
  2. For any pair $(p, q) \in S$, if (r, s) is a pair for which $\delta(r, a) = p$ and $\delta(s, a) = q$ for some $a \in \Sigma$, then (r, s) is in S.
  3. No other pairs are in S.

Minimal Finite Automata (Continued)

- It is not difficult to see that for any pair $(p, q) \in S$, $p \not\equiv q$. On the other hand, Lemma 5.2 makes it possible to show that S contains all such pairs by establishing the following statement:

  For any string $z \in \Sigma^*$, every pair of states (p, q) for which only one of the states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A is an element of S.

  The proof is by structural induction on z.

- For the basis step, if only one of $\delta^*(p, \Lambda)$ and $\delta^*(q, \Lambda)$ is in A, then only one of the two states p and q is in A, and $(p, q) \in S$ because of statement 1 of the definition.

- Now suppose that for some z, all pairs (p, q) for which only one of $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A are in S. Consider the string az, where $a \in \Sigma$, and suppose that (r, s) is a pair for which only one of $\delta^*(r, az)$ and $\delta^*(s, az)$ is in A. Letting $p = \delta(r, a)$ and $q = \delta(s, a)$, then

  $\delta^*(r, az) = \delta^*(\delta(r, a), z) = \delta^*(p, z)$
  $\delta^*(s, az) = \delta^*(\delta(s, a), z) = \delta^*(q, z)$

  The assumption about r and s is that only one of the states $\delta^*(r, az)$ and $\delta^*(s, az)$ is in A, and therefore only one of the states $\delta^*(p, z)$ and $\delta^*(q, z)$ is in A. The induction hypothesis therefore implies that $(p, q) \in S$, and it then follows from statement 2 in the recursive definition that $(r, s) \in S$.

Minimal Finite Automata (Continued)

- Now it is a simple matter to convert this recursive definition into an algorithm to identify all the pairs (p, q) for which $p \not\equiv q$.

- **Algorithm 5.1 (For Identifying the Pairs (p, q) with $p \not\equiv q$)** List all (unordered) pairs of states (p, q) for which $p \neq q$. Make a sequence of passes through these pairs. On the first pass, mark each pair of which exactly one element is in A. On each subsequent pass, mark any pair (r, s) if there is an $a \in \Sigma$ for which $\delta(r, a) = p$, $\delta(s, a) = q$, and (p, q) is already marked. After a pass in which no new pairs are marked, stop. The marked pairs (p, q) are precisely those for which $p \not\equiv q$.

- When the algorithm terminates, any pair (p, q) that remains unmarked represents two states that can be merged into one.

- Finding the total number of equivalence classes, or the minimum number of states, can be done by making one final pass through the states of M.

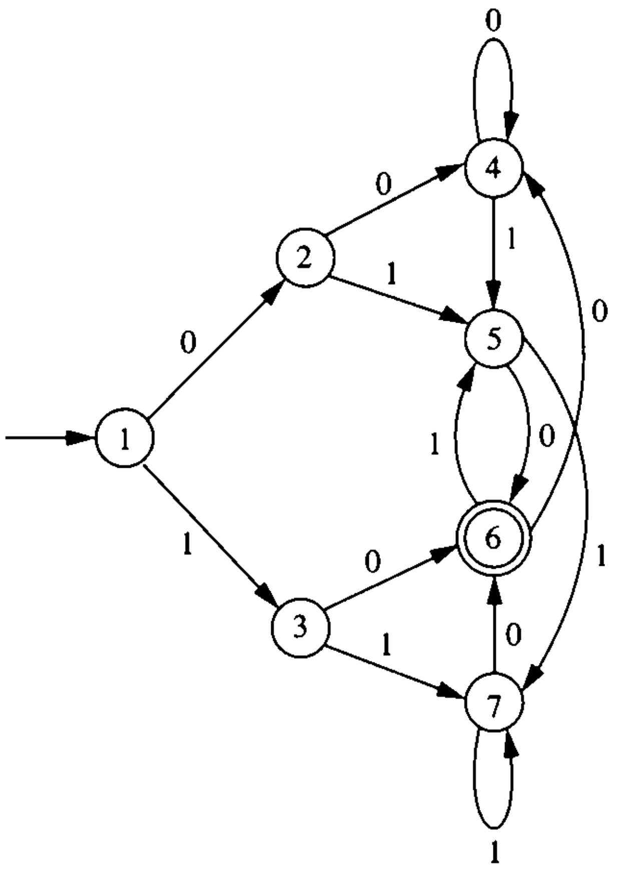
  The first state of M to be considered corresponds to one equivalence class.
  For each subsequent state q of M, q represents a new equivalence class only if the pair (p, q) was marked by Algorithm 5.1 for every previous state p of M.

- Once the states in the minimum-state FA are known, determining the transitions is straightforward.

Minimal Finite Automata (Continued)

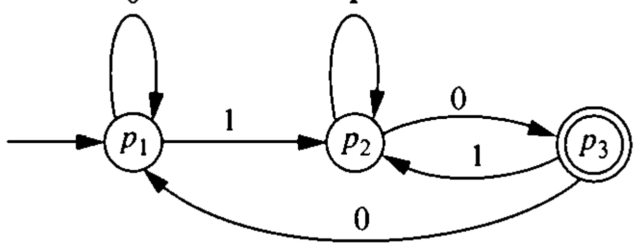
- **Example 5.6:** Minimizing the FA in Figure 5.3a

  The FA to be minimized:



The table at left shows all unordered pairs (*p*, *q*) with *p* ≠ *q*. Numbers indicate the pass on which each pair was marked. The resulting FA is shown on the right.

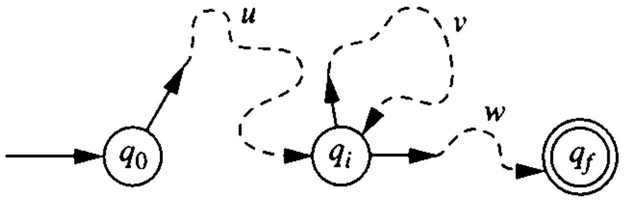
2						
3	2	2				
4			2			
5	2	2		2		
6	1	1	1	1	1	
7	2	2		2		1
	1	2	3	4	5	6



5.3 The Pumping Lemma for Regular Languages

- The fact that every regular language can be accepted by a finite automaton leads to a property shared by all regular languages. Showing that a language does not have this property will be another way of showing that the language is not regular.

- Suppose $M = (Q, \Sigma, q_0, A, \delta)$ is an FA recognizing a language L. An input string $x \in L$ requiring M to enter some state twice corresponds to a path that starts at q_0, ends at some accepting state q_f, and contains a loop:



  Any other path obtained from this one by changing the number of traversals of the loop will correspond to a different element of L.

- Suppose that Q has n elements. For any string x in L with length at least n, if we write $x = a_1a_2\ldots a_ny$, then the sequence of $n + 1$ states

  $q_0 = \delta^*(q_0, \Lambda)$
  $q_1 = \delta^*(q_0, a_1)$
  $q_2 = \delta^*(q_0, a_1a_2)$
  …
  $q_n = \delta^*(q_0, a_1a_2\ldots a_n)$

  must contain some state at least twice, by the pigeonhole principle. Suppose $q_i = q_{i+p}$, where $0 \le i < i + p \le n$. Then

  $\delta^*(q_0, a_1a_2\ldots a_i) = q_i$
  $\delta^*(q_i, a_{i+1}a_{i+2}\ldots a_{i+p}) = q_i$
  $\delta^*(q_i, a_{i+p+1}a_{i+p+2}\ldots a_ny) = q_f \in A$

  Let $u = a_1a_2\ldots a_i$, $v = a_{i+1}a_{i+2}\ldots a_{i+p}$, and $w = a_{i+p+1}a_{i+p+2}\ldots a_ny$.

  Since $\delta^*(q_i, v) = q_i$, it must be the case that $\delta^*(q_i, v^m) = q_i$ for every $m \ge 0$, and it follows that $\delta^*(q_0, uv^mw) = q_f$ for every $m \ge 0$. Since $p > 0$ and $i + p \le n$, we have proved the following result.

The Pumping Lemma for Regular Languages (Continued)

- **Theorem 5.2.** Suppose L is a regular language recognized by a finite automaton with n states. For any $x \in L$ with $|x| \geq n$, x may be written as $x = uvw$ for some strings u, v, and w satisfying

  $|uv| \leq n$
  $|v| > 0$
  for any $m \geq 0$, $uv^m w \in L$

- This result is often referred to as the Pumping Lemma for Regular Languages, because it says that any sufficiently long string in L can be "pumped" by inserting additional copies of the substring v.

- Theorem 5.2 is a bit tricky to use. A slightly weaker version is sufficient for most applications.

- **Theorem 5.2a (The Pumping Lemma for Regular Languages).** Suppose L is a regular language. Then there is an integer n so that for any $x \in L$ with $|x| \geq n$, there are strings u, v, and w so that

  $x = uvw$ (5.1)
  $|uv| \leq n$ (5.2)
  $|v| > 0$ (5.3)
  for any $m \geq 0$, $uv^m w \in L$ (5.4)

- Using the pumping lemma to show that a language L is not regular involves assuming that L is regular and then deriving a contradiction.

- If L is regular, the theorem says that a particular integer n exists, but doesn't say what the value of n is. The goal is to find a string x with $|x| \geq n$ so that the statements involving x in the theorem will lead to a contradiction. The choice of x must be made without knowing the value of n.

- To prove that L is not regular, it will be necessary to show that any choice of u, v, and w satisfying (5.1)–(5.4) produces a contradiction.

Applying the Pumping Lemma

- **Example 5.7:** Application of the Pumping Lemma

  Let $L = \{0^i 1^i \mid i \geq 0\}$. Suppose that L is regular, and let n be the integer in Theorem 5.2a. Let $x = 0^n 1^n$. Since $|x| \geq n$, the theorem says that $x = uvw$ for some u, v, and w satisfying (5.2)–(5.4). The fact that (5.2) is true implies that $uv = 0^k$ for some k, and it follows from (5.3) that $v = 0^j$ for some $j > 0$. Equation (5.4) says that $uv^m w \in L$ for every $m \geq 0$. If $m = 2$, however, the string uv^2w contains j extra 0's in the first part ($uv^2w = 0^{n+j}1^n$), and cannot be in L because $j > 0$. Because of this contradiction, L cannot be regular.

  Another way to choose x would have been to let it be $0^m 1^m$ for some $m \geq n/2$. The proof would be more complicated, because uv would not necessarily consist of 0's. There would be two other cases: $v = 0^j 1^j$ (causing uv^2w to contain the substring 10, a contradiction) or $v = 1^j$ (causing uv^2w to contain more 1's than 0's, also a contradiction).

  In general, it is best to choose x so that getting the contradiction is as simple as possible.

  The initial choice of x (but not the second choice) could also be used to show that the larger language $L_1 = \{x \in \{0, 1\}^* \mid n_0(x) = n_1(x)\}$ is not regular.

  Choosing $x = (01)^n$ in an attempt to show that L_1 is not regular would fail. Possible values for v include $(01)^j$ and $1(01)^j0$, neither of which leads to a contradiction.

Applying the Pumping Lemma (Continued)

- **Example 5.8:** Another Application of the Pumping Lemma

  Consider the language

  $L = \{0^i x \mid i \geq 0, x \in \{0, 1\}^* \text{ and } |x| \leq i\}$

  Assume that L is regular, and let n be the integer in Theorem 5.2a. Let $x = 0^n 1^n$. Then if conditions (5.1)–(5.4) hold, it follows that $v = 0^j$ for some $j > 0$. Inserting additional copies of v will not lead to a contradiction, unfortunately. However, (5.4) also says that $uv^0 w \in L$. This does produce a contradiction, because $uv^0 w = uw = 0^{n-j} 1^n \notin L$.Therefore, L is not regular.

- **Example 5.9:** Application of the Pumping Lemma to *pal*

  Let L be *pal*, the languages of palindromes over $\{0, 1\}$. Suppose that L is regular, and let n be the integer in the statement of the pumping lemma. Let $x = 0^n 1 0^n$.

  If conditions (5.1)–(5.4) are true, then v is a substring of the form 0^j (with $j > 0$) from the first part of x. A contradiction can be obtained for either $m = 0$ or $m > 1$. In the first case, $uv^m w = 0^{n-j} 1 0^n$, and in the second case, if $m = 2$ for example, $uv^m w = 0^{n+j} 1 0^n$. Neither of these is a palindrome, and it follows that L cannot be regular.

Weaker Forms of the Pumping Lemma

- It is often possible to get by with a weakened form of the pumping lemma.

- **Theorem 5.3 (Weak Form of Pumping Lemma).** Suppose L is an infinite regular language. Then there are strings u, v, and w so that $|v| > 0$ and $uv^m w \in L$ for every $m \geq 0$.

  Proof: Follows immediately from Theorem 5.2a. No matter how big the integer n in the statement of that theorem is, L must contain a string at least that long, because L has infinitely many elements.

- Theorem 5.3 would be sufficient for Example 5.7, but not Examples 5.8 or 5.9.

- **Theorem 5.4 (Even Weaker Form of Pumping Lemma).** Suppose L is an infinite regular language. There are integers p and q, with $q > 0$, so that for every $m \geq 0$, L contains a string of length $p + mq$. In other words, the set of integers

  $lengths(L) = \{|x| \mid x \in L\}$

  contains the "arithmetic progression" of all integers $p + mq$ (where $m \geq 0$).

  Proof: Follows from Theorem 5.3, by taking $p = |u| + |w|$ and $q = |v|$.

- Theorem 5.4 would not be enough to show that the language in Example 5.7 is not regular. The next example shows a language for which it might be used.

- **Example 5.10:** An Application of Theorem 5.4

  Let $L = \{0^n \mid n \text{ is prime}\} = \{0^2, 0^3, 0^5, 0^7, 0^{11}, \ldots\}$. To prove that L is not regular using Theorem 5.4, it is necessary to show that, for any $p \geq 0$ and any $q > 0$, there is an integer m so that $p + mq$ is not prime.

  Let $m = p + 2q + 2$. Then

  $$\begin{aligned} p + mq &= p + (p + 2q + 2)q \\ &= (p + 2q) + (p + 2q)q \\ &= (p + 2q)(1 + q) \end{aligned}$$

  which is clearly not prime.

The Pumping Lemma Cannot Show a Language Is Regular

- Corollary 5.1 gives a condition involving a language that is necessary *and* sufficient for the language to be regular.

- Theorem 5.2a gives a condition that is necessary but not sufficient. Showing that the conclusions of the theorem hold is not enough to show that a language is regular.

- **Example 5.11:** The Pumping Lemma Cannot Show a Language Is Regular

  Let

  $L = \{a^i b^j c^j \mid i \geq 1 \text{ and } j \geq 0\} \cup \{b^j c^k \mid j, k \geq 0\}$

  To show that the conclusions of Theorem 5.2a hold, take n to be 1, and suppose that $x \in L$ and $|x| \geq n$. There are two cases to consider.

  1. If $x = a^i b^j c^j$, where $i > 0$, then define

  $u = \Lambda \quad v = a \quad w = a^{i-1} b^j c^j$

  Any string of the form $uv^m w$ is still of the form $a^l b^j c^j$ and is therefore an element of L (whether or not l is 0).

  2. If $x = b^i c^j$, then again let $u = \Lambda$ and let v be the first symbol in x. It is still true that $uv^m w \in L$ for every $m \geq 0$.

  However, L is not regular, which can be shown using Corollary 5.1.

5.4 Decision Problems

- The computational problems that a finite automaton can solve are limited to *decision* problems: problems that can be answered yes or no, like:

  Given a string x of a’s and b’s, does x contain an occurrence of the substring *baa*?
  Given a regular expression r and a string x, does x belong to the language corresponding to r?

- A decision problem of this type consists of a set of specific *instances*.

  An instance of the first problem is a string x of a’s and b’s, and the set of possible instances is the entire set $\{a, b\}^*$.
  An instance of the second is a pair (r, x), where r is a regular expression and x is a string.

- The generic decision problem that can be solved by a particular finite automaton is the *membership problem* for the corresponding regular language L: Given a string x, is x an element of L? An instance of this problem is a string x.

- The *membership problem for regular languages* is: Given a finite automaton M and a string x, is x accepted by M? An instance of the problem is a pair (M, x), where M is an FA and x is a string.

- The latter problem has an easy solution: give the string x to M as input and see whether M ends up in an accepting state.

Decision Problems (Continued)

- A partial list of other decision problems having to do with finite automata and regular languages:

  1. Given a regular expression r and a string x, does x belong to the language corresponding to r?
  2. Given a finite automaton M, is there a string that it accepts? (Alternatively, given an FA M, is $L(M) = \emptyset$)?
  3. Given an FA M, is $L(M)$ finite?
  4. Given two finite automata M_1 and M_2, are there any strings that are accepted by both?
  5. Given two FAs M_1 and M_2, do they accept the same language? In other words, is $L(M_1) = L(M_2)$?
  6. Given two FAs M_1 and M_2, is $L(M_1)$ a subset of $L(M_2)$?
  7. Given two regular expressions r_1 and r_2, do they correspond to the same language?
  8. Given an FA M, is it a minimum-state FA accepting the language $L(M)$?

- Chapter 4 provides an algorithm to take an arbitrary regular expression and produce an FA M accepting the corresponding language, so problem 1 can be reduced to the problem of testing whether M accepts x.

- Section 5.2 gives a decision algorithm for problem 8: Apply the minimization algorithm to M, and see if the number of states is reduced.

- An algorithm to solve problem 2 could be used to solve problems 4 through 7.

  Problem 4 could be solved by first using the algorithm presented in Section 3.5 to construct a finite automaton M recognizing $L(M_1) \cap L(M_2)$, and then applying to M the algorithm for problem 2.

  Problem 6 could be solved the same way, with $L(M_1) \cap L(M_2)$ replaced by $L(M_1) - L(M_2)$, because $L(M_1) \subseteq L(M_2)$ if and only if $L(M_1) - L(M_2) = \emptyset$.

  Problem 5 can be reduced to problem 6, since two sets are equal precisely when each is a subset of the other.

  A solution to problem 6 would also solve problem 7, using the algorithm for finding a finite automaton corresponding to a given regular expression.

Decision Problems (Continued)

- With regard to problem 2, an FA will fail to accept any strings if none of its accepting states is reachable from the initial state.

- T_k, the set of states that can be reached from q_0 by using strings of length k or less, is defined as follows:

$$T_k = \begin{cases} \{q_0\} & \text{if } k = 0 \\ T_{k-1} \cup \{\delta(q, a) | q \in T_{k-1} \text{ and } a \in \Sigma\} & \text{if } k > 0 \end{cases}$$

- **Decision Algorithm for Problem 2 (Given an FA *M*, is *L*(*M*) = ∅?)** Compute the set T_k for each $k \geq 0$, until either T_k contains an accepting state or until $k > 0$ and $T_k = T_{k-1}$. In the first case $L(M) \neq \emptyset$, and in the second case $L(M) = \emptyset$.

- If n is the number of states of M, then one of the two outcomes of the algorithm must occur by the time T_k has been computed. This implies that the following (less efficient) algorithm would also work:

  Begin testing all input strings, in nondecreasing order of length, for acceptance by M. If no strings of length n or less are accepted, then $L(M) = \emptyset$.

Decision Problems (Continued)

- The pumping lemma is useful for solving problem 3.
- Suppose that M is an FA with n states that accepts a language L. If L contains any strings whose length is at least n, then the pumping lemma implies that L must be infinite. In particular, if there is a string $x \in L$ with $n \leq |x| < 2n$, then L is infinite.
- On the other hand, if there are strings in L of length at least n, it is impossible for the shortest such string x to have length $2n$ or greater—because there would then have to be a shorter string $y \in L$ whose length is at least n. Therefore, if L is infinite, there must be a string $x \in L$ with $n \leq |x| < 2n$.
- **Decision Algorithm for Problem 3 (Given an FA *M*, Is *L*(*M*) finite?)** Test input strings beginning with those of length n (where n is the number of states of M), in nondecreasing order of length. If there is a string x with $n \leq |x| < 2n$ that is accepted, then $L(M)$ is infinite; otherwise, $L(M)$ is finite.

Decision Problems (Continued)

- There are at least two reasons for discussing, and trying to solve, decision problems.
- One is the obvious fact that solutions may be useful. In this case, finding an efficient solution is at least as important as determining whether there is a solution in principle.
- There is another reason for considering these decision problems: not all decision problems can be solved.
- In the 1930s, mathematician Alan Turing described a type of abstract machine, now called a Turing machine. He showed that the membership problem for Turing machines (given a Turing machine M and a string x, does M accept x?) is unsolvable.
- Turing machines turn out to be a general model of computation, making it possible to formulate the idea of an algorithm precisely and to say exactly what "unsolvable" means.
- Showing the existence of unsolvable problems—particularly ones that arise naturally and are easy to state—was a significant development in the theory of computation.

5.5 Regular Languages and Computers

- Regular languages have limited usefulness in pratical computer science. In particular, programming languages are not regular.

  In the C language, the string `main(){`^{m}`}`^{n} is a valid program if and only if $m = n$. It is easy to use Corollary 5.1 or the pumping lemma to show that the set of valid C programs is not regular.

- However, regular expressions are often used to describe the *tokens* of a programming language (including identifiers, literals, operators, reserved words, and punctuation).

- In particular, the input to `lex` (a Unix lexical-analyzer generator) is a set of regular expressions specifying the structure of tokens. The output is a software version of an FA that can be incorporated into a compiler.

- Regular expressions come up in Unix in other ways as well.

  Unix text editors allow searches for text that matches a regular expression.

  Commands such as `grep` (global regular expression print) and `egrep` (extended global regular expression print) can search a file for lines containing strings that match a specified regular expression.

- In theory, a computer could be modeled as a finite automaton. As a way of understanding a computer, however, this observation is not helpful; there is hardly any practical difference between a huge number of possible states and an infinite number.

- The situation is similar with regard to languages. A programming language is effectively regular because the set of programs that could actually be written is finite. However, finite sets are not always easier to deal with than infinite sets, because they may have no underlying structure.

- Later chapters introduce abstract models that resemble a computer more closely. Studying these models is the best way to understand both the potential and the limitations of the physical machines that approximate the models.