Exercise 4.3.4: Give an algorithm to tell whether two regular languages L_1 and L_2 have at least one string in common.

Exercise 4.3.5: Give an algorithm to tell, for two regular languages L_1 and L_2 over the same alphabet Σ, whether there is any string in Σ^* that is in neither L_1 nor L_2.

4.4 Equivalence and Minimization of Automata

In contrast to the previous questions — emptiness and membership — whose algorithms were rather simple, the question of whether two descriptions of two regular languages actually define the same language involves considerable intellectual mechanics. In this section we discuss how to test whether two descriptors for regular languages are *equivalent*, in the sense that they define the same language. An important consequence of this test is that there is a way to minimize a DFA. That is, we can take any DFA and find an equivalent DFA that has the minimum number of states. In fact, this DFA is essentially unique: given any two minimum-state DFA's that are equivalent, we can always find a way to rename the states so that the two DFA's become the same.

4.4.1 Testing Equivalence of States

We shall begin by asking a question about the states of a single DFA. Our goal is to understand when two distinct states p and q can be replaced by a single state that behaves like both p and q. We say that states p and q are *equivalent* if:

- For all input strings w, $\hat{\delta}(p, w)$ is an accepting state if and only if $\hat{\delta}(q, w)$ is an accepting state.

Less formally, it is impossible to tell the difference between equivalent states p and q merely by starting in one of the states and asking whether or not a given input string leads to acceptance when the automaton is started in this (unknown) state. Note we do *not* require that $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ are the *same* state, only that either both are accepting or both are nonaccepting.

If two states are not equivalent, then we say they are *distinguishable*. That is, state p is distinguishable from state q if there is at least one string w such that one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and the other is not accepting.

Example 4.18: Consider the DFA of Fig. 4.8, whose transition function we shall refer to as δ in this example. Certain pairs of states are obviously not equivalent. For example, C and G are not equivalent because one is accepting and the other is not. That is, the empty string distinguishes these two states, because $\hat{\delta}(C, \epsilon)$ is accepting and $\hat{\delta}(G, \epsilon)$ is not.

Consider states A and G. String ϵ doesn't distinguish them, because they are both nonaccepting states. String 0 doesn't distinguish them because they go to

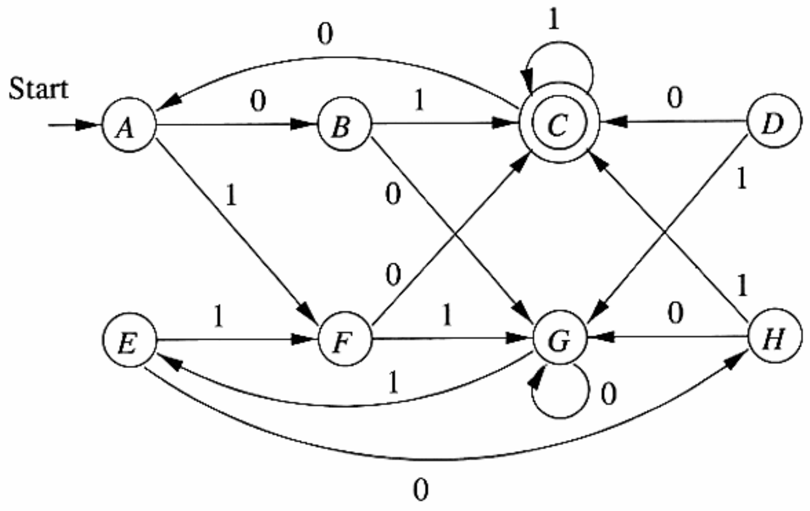

Figure 4.8: An automaton with equivalent states

states B and G, respectively on input 0, and both these states are nonaccepting. Likewise, string 1 doesn't distinguish A from G, because they go to F and E, respectively, and both are nonaccepting. However, 01 distinguishes A from G, because $\hat{\delta}(A, 01) = C$, $\hat{\delta}(G, 01) = E$, C is accepting, and E is not. Any input string that takes A and G to states only one of which is accepting is sufficient to prove that A and G are not equivalent.

In contrast, consider states A and E. Neither is accepting, so ϵ does not distinguish them. On input 1, they both go to state F. Thus, no input string that begins with 1 can distinguish A from E, since for any string x, $\hat{\delta}(A, 1x) = \hat{\delta}(E, 1x)$.

Now consider the behavior of states A and E on inputs that begin with 0. They go to states B and H, respectively. Since neither is accepting, string 0 by itself does not distinguish A from E. However, B and H are no help. On input 1 they both go to C, and on input 0 they both go to G. Thus, all inputs that begin with 0 will fail to distinguish A from E. We conclude that no input string whatsoever will distinguish A from E; i.e., they are equivalent states. □

To find states that are equivalent, we make our best efforts to find pairs of states that are distinguishable. It is perhaps surprising, but true, that if we try our best, according to the algorithm to be described below, then any pair of states that we do not find distinguishable are equivalent. The algorithm, which we refer to as the *table-filling algorithm*, is a recursive discovery of distinguishable pairs in a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

BASIS: If p is an accepting state and q is nonaccepting, then the pair $\{p, q\}$ is distinguishable.

INDUCTION: Let p and q be states such that for some input symbol a, $r = \delta(p, a)$ and $s = \delta(q, a)$ are a pair of states known to be distinguishable. Then

$\{p, q\}$ is a pair of distinguishable states. The reason this rule makes sense is that there must be some string w that distinguishes r from s; that is, exactly one of $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$ is accepting. Then string aw must distinguish p from q, since $\hat{\delta}(p, aw)$ and $\hat{\delta}(q, aw)$ is the same pair of states as $\hat{\delta}(r, w)$ and $\hat{\delta}(s, w)$.

Example 4.19: Let us execute the table-filling algorithm on the DFA of Fig 4.8. The final table is shown in Fig. 4.9, where an x indicates pairs of distinguishable states, and the blank squares indicate those pairs that have been found equivalent. Initially, there are no x's in the table.

B	x						
C	x	x					
D	x	x	x				
E		x	x	x			
F	x	x	x		x		
G	x	x	x	x	x	x	
H	x		x	x	x	x	x
	A	B	C	D	E	F	G

Figure 4.9: Table of state inequivalences

For the basis, since C is the only accepting state, we put x's in each pair that involves C. Now that we know some distinguishable pairs, we can discover others. For instance, since $\{C, H\}$ is distinguishable, and states E and F go to H and C, respectively, on input 0, we know that $\{E, F\}$ is also a distinguishable pair. In fact, all the x's in Fig. 4.9 with the exception of the pair $\{A, G\}$ can be discovered simply by looking at the transitions from the pair of states on either 0 or on 1, and observing that (for one of those inputs) one state goes to C and the other does not. We can show $\{A, G\}$ is distinguishable on the next round, since on input 1 they go to F and E, respectively, and we already established that the pair $\{E, F\}$ is distinguishable.

However, then we can discover no more distinguishable pairs. The three remaining pairs, which are therefore equivalent pairs, are $\{A, E\}$, $\{B, H\}$, and $\{D, F\}$. For example, consider why we can not infer that $\{A, E\}$ is a distinguishable pair. On input 0, A and E go to B and H, respectively, and $\{B, H\}$ has not yet been shown distinguishable. On input 1, A and E both go to F, so there is no hope of distinguishing them that way. The other two pairs, $\{B, H\}$ and $\{D, F\}$ will never be distinguished because they each have identical transitions on 0 and identical transitions on 1. Thus, the table-filling algorithm stops with the table as shown in Fig. 4.9, which is the correct determination of equivalent and distinguishable states. $\Box$

Theorem 4.20: If two states are not distinguished by the table-filling algorithm, then the states are equivalent.

PROOF: Let us again assume we are talking of the DFA $A = (Q, \Sigma, \delta, q_0, F)$. Suppose the theorem is false; that is, there is at least one pair of states $\{p, q\}$ such that

1. States p and q are distinguishable, in the sense that there is some string w such that exactly one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting, and yet

2. The table-filling algorithm does not find p and q to be distinguished.

Call such a pair of states a *bad pair.*

If there are bad pairs, then there must be some that are distinguished by the shortest strings among all those strings that distinguish bad pairs. Let $\{p, q\}$ be one such bad pair, and let $w = a_1a_2 \cdots a_n$ be a string as short as any that distinguishes p from q. Then exactly one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is accepting.

Observe first that w cannot be ϵ, since if ϵ distinguishes a pair of states, then that pair is marked by the basis part of the table-filling algorithm. Thus, $n \geq 1$.

Consider the states $r = \delta(p, a_1)$ and $s = \delta(q, a_1)$. States r and s are distinguished by the string $a_2a_3 \cdots a_n$, since this string takes r and s to the states $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$. However, the string distinguishing r from s is shorter than any string that distinguishes a bad pair. Thus, $\{r, s\}$ cannot be a bad pair. Rather, the table-filling algorithm must have discovered that they are distinguishable.

But the inductive part of the table-filling algorithm will not stop until it has also inferred that p and q are distinguishable, since it finds that $\delta(p, a_1) = r$ is distinguishable from $\delta(q, a_1) = s$. We have contradicted our assumption that bad pairs exist. If there are no bad pairs, then every pair of distinguishable states is distinguished by the table-filling algorithm, and the theorem is true. $\square$

4.4.2 Testing Equivalence of Regular Languages

The table-filling algorithm gives us an easy way to test if two regular languages are the same. Suppose languages L and M are each represented in some way, e.g., one by a regular expression and one by an NFA. Convert each representation to a DFA. Now, imagine one DFA whose states are the union of the states of the DFA's for L and M. Technically, this DFA has two start states, but actually the start state is irrelevant as far as testing state equivalence is concerned, so make any state the lone start state.

Now, test if the start states of the two original DFA's are equivalent, using the table-filling algorithm. If they are equivalent, then $L = M$, and if not, then $L \neq M$.

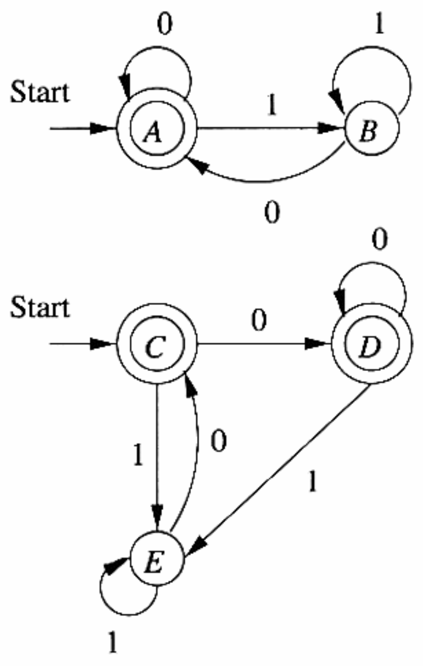

Figure 4.10: Two equivalent DFA's

Example 4.21: Consider the two DFA's in Fig. 4.10. Each DFA accepts the empty string and all strings that end in 0; that is the language of regular expression $\epsilon + (\mathbf{0} + \mathbf{1})^*\mathbf{0}$. We can imagine that Fig. 4.10 represents a single DFA, with five states A through E. If we apply the table-filling algorithm to that automaton, the result is as shown in Fig. 4.11.

B	x			
C		x		
D		x		
E	x		x	x
	A	B	C	D

Figure 4.11: The table of distinguishabilities for Fig. 4.10

To see how the table is filled out, we start by placing x's in all pairs of states where exactly one of the states is accepting. It turns out that there is no more to do. The four remaining pairs, $\{A, C\}$, $\{A, D\}$, $\{C, D\}$, and $\{B, E\}$ are all equivalent pairs. You should check that no more distinguishable pairs are discovered in the inductive part of the table-filling algorithm. For instance, with the table as in Fig. 4.11, we cannot distinguish the pair $\{A, D\}$ because on 0 they go to themselves, and on 1 they go to the pair $\{B, E\}$, which has not yet been distinguished. Since A and C are found equivalent by this test, and those states were the start states of the two original automata, we conclude that these DFA's do accept the same language. □

The time to fill out the table, and thus to decide whether two states are equivalent is polynomial in the number of states. If there are n states, then there are $\binom{n}{2}$, or $n(n-1)/2$ pairs of states. In one round, we consider all pairs of states, to see if one of their successor pairs has been found distinguishable, so a round surely takes no more than $O(n^2)$ time. Moreover, if on some round, no additional x's are placed in the table, then the algorithm ends. Thus, there can be no more than $O(n^2)$ rounds, and $O(n^4)$ is surely an upper bound on the running time of the table-filling algorithm.

However, a more careful algorithm can fill the table in $O(n^2)$ time. The idea is to initialize, for each pair of states $\{r, s\}$, a list of those pairs $\{p, q\}$ that "depend on" $\{r, s\}$. That is, if $\{r, s\}$ is found distinguishable, then $\{p, q\}$ is distinguishable. We create the lists initially by examining each pair of states $\{p, q\}$, and for each of the fixed number of input symbols a, we put $\{p, q\}$ on the list for the pair of states $\{\delta(p, a), \delta(q, a)\}$, which are the successor states for p and q on input a.

If we ever find $\{r, s\}$ to be distinguishable, then we go down the list for $\{r, s\}$. For each pair on that list that is not already distinguishable, we make that pair distinguishable, and we put the pair on a queue of pairs whose lists we must check similarly.

The total work of this algorithm is proportional to the sum of the lengths of the lists, since we are at all times either adding something to the lists (initialization) or examining a member of the list for the first and last time (when we go down the list for a pair that has been found distinguishable). Since the size of the input alphabet is considered a constant, each pair of states is put on $O(1)$ lists. As there are $O(n^2)$ pairs, the total work is $O(n^2)$.

4.4.3 Minimization of DFA's

Another important consequence of the test for equivalence of states is that we can "minimize" DFA's. That is, for each DFA we can find an equivalent DFA that has as few states as any DFA accepting the same language. Moreover, except for our ability to call the states by whatever names we choose, this minimum-state DFA is unique for the language. The algorithm is as follows:

1. First, eliminate any state that cannot be reached from the start state.

2. Then, partition the remaining states into blocks, so that all states in the same block are equivalent, and no pair of states from different blocks are equivalent. Theorem 4.24, below, shows that we can always make such a partition.

Example 4.22: Consider the table of Fig. 4.9, where we determined the state equivalences and distinguishabilities for the states of Fig. 4.8. The partition of the states into equivalent blocks is $(\{A,E\},\ \{B,H\},\ \{C\},\ \{D,F\},\ \{G\})$. Notice that the three pairs of states that are equivalent are each placed in a block together, while the states that are distinguishable from all the other states are each in a block alone.

For the automaton of Fig. 4.10, the partition is $(\{A,C,D\},\ \{B,E\})$. This example shows that we can have more than two states in a block. It may appear fortuitous that A, C, and D can all live together in a block, because every pair of them is equivalent, and none of them is equivalent to any other state. However, as we shall see in the next theorem to be proved, this situation is guaranteed by our definition of "equivalence" for states. $\square$

Theorem 4.23: The equivalence of states is transitive. That is, if in some DFA $A = (Q, \Sigma, \delta, q_0, F)$ we find that states p and q are equivalent, and we also find that q and r are equivalent, then it must be that p and r are equivalent.

PROOF: Note that transitivity is a property we expect of any relationship called "equivalence." However, simply calling something "equivalence" doesn't make it transitive; we must prove that the name is justified.

Suppose that the pairs $\{p,q\}$ and $\{q,r\}$ are equivalent, but pair $\{p,r\}$ is distinguishable. Then there is some input string w such that exactly one of $\hat{\delta}(p,w)$ and $\hat{\delta}(r,w)$ is an accepting state. Suppose, by symmetry, that $\hat{\delta}(p,w)$ is the accepting state.

Now consider whether $\hat{\delta}(q,w)$ is accepting or not. If it is accepting, then $\{q,r\}$ is distinguishable, since $\hat{\delta}(q,w)$ is accepting, and $\hat{\delta}(r,w)$ is not. If $\hat{\delta}(q,w)$ is nonaccepting, then $\{p,q\}$ is distinguishable for a similar reason. We conclude by contradiction that $\{p,r\}$ was not distinguishable, and therefore this pair is equivalent. $\square$

We can use Theorem 4.23 to justify the obvious algorithm for partitioning states. For each state q, construct a block that consists of q and all the states that are equivalent to q. We must show that the resulting blocks are a partition; that is, no state is in two distinct blocks.

First, observe that all states in any block are mutually equivalent. That is, if p and r are two states in the block of states equivalent to q, then p and r are equivalent to each other, by Theorem 4.23.

Suppose that there are two overlapping, but not identical blocks. That is, there is a block B that includes states p and q, and another block C that includes p but not q. Since p and q are in a block together, they are equivalent. Consider how the block C was formed. If it was the block generated by p, then q would be in C, because those states are equivalent. Thus, it must be that there is some third state s that generated block C; i.e., C is the set of states equivalent to s.

We know that p is equivalent to s, because p is in block C. We also know that p is equivalent to q because they are together in block B. By the transitivity of Theorem 4.23, q is equivalent to s. But then q belongs in block C, a contradiction. We conclude that equivalence of states partitions the states; that is, two states either have the same set of equivalent states (including themselves), or their equivalent states are disjoint. To conclude the above analysis:

Theorem 4.24: If we create for each state q of a DFA a *block* consisting of q and all the states equivalent to q, then the different blocks of states form a *partition* of the set of states.[5] That is, each state is in exactly one block. All members of a block are equivalent, and no pair of states chosen from different blocks are equivalent. $\square$

We are now able to state succinctly the algorithm for minimizing a DFA $A = (Q, \Sigma, \delta, q_0, F)$.

1. Use the table-filling algorithm to find all the pairs of equivalent states.

2. Partition the set of states Q into blocks of mutually equivalent states by the method described above.

3. Construct the minimum-state equivalent DFA B by using the blocks as its states. Let γ be the transition function of B. Suppose S is a set of equivalent states of A, and a is an input symbol. Then there must exist one block T of states such that for all states q in S, $\delta(q, a)$ is a member of block T. For if not, then input symbol a takes two states p and q of S to states in different blocks, and those states are distinguishable by Theorem 4.24. That fact lets us conclude that p and q are not equivalent, and they did not both belong in S. As a consequence, we can let $\gamma(S, a) = T$. In addition:

   (a) The start state of B is the block containing the start state of A.

   (b) The set of accepting states of B is the set of blocks containing accepting states of A. Note that if one state of a block is accepting, then all the states of that block must be accepting. The reason is that any accepting state is distinguishable from any nonaccepting state, so you can't have both accepting and nonaccepting states in one block of equivalent states.

Example 4.25: Let us minimize the DFA from Fig. 4.8. We established the blocks of the state partition in Example 4.22. Figure 4.12 shows the minimum-state automaton. Its five states correspond to the five blocks of equivalent states for the automaton of Fig. 4.8.

The start state is $\{A, E\}$, since A was the start state of Fig. 4.8. The only accepting state is $\{C\}$, since C is the only accepting state of Fig. 4.8. Notice that the transitions of Fig. 4.12 properly reflect the transitions of Fig. 4.8. For instance, Fig. 4.12 has a transition on input 0 from $\{A, E\}$ to $\{B, H\}$. That

[5] You should remember that the same block may be formed several times, starting from different states. However, the partition consists of the *different* blocks, so this block appears only once in the partition.

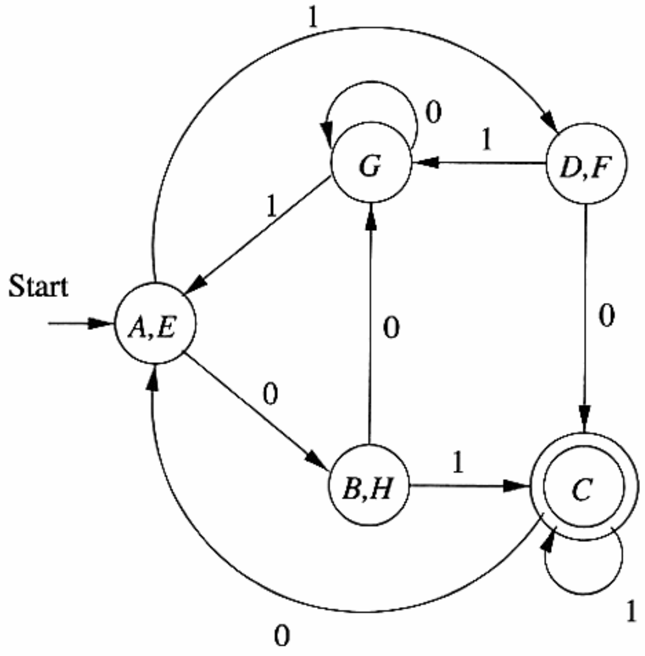

Figure 4.12: Minimum-state DFA equivalent to Fig. 4.8

makes sense, because in Fig. 4.8, A goes to B on input 0, and E goes to H. Likewise, on input 1, $\{A, E\}$ goes to $\{D, F\}$. If we examine Fig. 4.8, we find that both A and E go to F on input 1, so the selection of the successor of $\{A, E\}$ on input 1 is also correct. Note that the fact neither A nor E goes to D on input 1 is not important. You may check that all of the other transitions are also proper. □

4.4.4 Why the Minimized DFA Can't Be Beaten

Suppose we have a DFA A, and we minimize it to construct a DFA M, using the partitioning method of Theorem 4.24. That theorem shows that we can't group the states of A into fewer groups and still have an equivalent DFA. However, could there be another DFA N, unrelated to A, that accepts the same language as A and M, yet has fewer states than M? We can prove by contradiction that N does not exist.

First, run the state-distinguishability process of Section 4.4.1 on the states of M and N together, as if they were one DFA. We may assume that the states of M and N have no names in common, so the transition function of the combined automaton is the union of the transition rules of M and N, with no interaction. States are accepting in the combined DFA if and only if they are accepting in the DFA from which they come.

The start states of M and N are indistinguishable because $L(M) = L(N)$. Further, if $\{p, q\}$ are indistinguishable, then their successors on any one input

Minimizing the States of an NFA

You might imagine that the same state-partition technique that minimizes the states of a DFA could also be used to find a minimum-state NFA equivalent to a given NFA or DFA. While we can, by a process of exhaustive enumeration, find an NFA with as few states as possible accepting a given regular language, we cannot simply group the states of some given NFA for the language.

An example is in Fig. 4.13. None of the three states are equivalent. Surely accepting state B is distinguishable from nonaccepting states A and C. However, A and C are distinguishable by input 0. The successors of C are A alone, which does not include an accepting state, while the successors of A are $\{A, B\}$, which does include an accepting state. Thus, grouping equivalent states does not reduce the number of states of Fig. 4.13.

However, we can find a smaller NFA for the same language if we simply remove state C. Note that A and B alone accept all strings ending in 0, while adding state C does not allow us to accept any other strings.

symbol are also indistinguishable. The reason is that if we could distinguish the successors, then we could distinguish p from q.

Neither M nor N could have an inaccessible state, or else we could eliminate that state and have an even smaller DFA for the same language. Thus, every state of M is indistinguishable from at least one state of N. To see why, suppose p is a state of M. Then there is some string $a_1a_2\cdots a_k$ that takes the start state of M to state p. This string also takes the start state of N to some state q. Since we know the start states are indistinguishable, we also know that their successors under input symbol a_1 are indistinguishable. Then, the successors of those states on input a_2 are indistinguishable, and so on, until we conclude

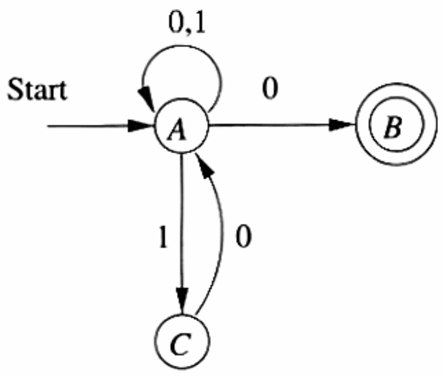

Figure 4.13: An NFA that cannot be minimized by state equivalence

that p and q are indistinguishable.

Since N has fewer states than M, there are two states of M that are indistinguishable from the same state of N, and therefore indistinguishable from each other. But M was designed so that all its states *are* distinguishable from each other. We have a contradiction, so the assumption that N exists is wrong, and M in fact has as few states as any equivalent DFA for A. Formally, we have proved:

Theorem 4.26: If A is a DFA, and M the DFA constructed from A by the algorithm described in the statement of Theorem 4.24, then M has as few states as any DFA equivalent to A. □

In fact we can say something even stronger than Theorem 4.26. There must be a one-to-one correspondence between the states of any other minimum-state N and the DFA M. The reason is that we argued above how each state of M must be equivalent to one state of N, and no state of M can be equivalent to two states of N. We can similarly argue that no state of N can be equivalent to two states of M, although each state of N must be equivalent to one of M's states. Thus, the minimum-state DFA equivalent to A is unique except for a possible renaming of the states.

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

Figure 4.14: A DFA to be minimized

4.4.5 Exercises for Section 4.4

* **Exercise 4.4.1:** In Fig. 4.14 is the transition table of a DFA.

a) Draw the table of distinguishabilities for this automaton.

b) Construct the minimum-state equivalent DFA.

Exercise 4.4.2: Repeat Exercise 4.4.1 for the DFA of Fig 4.15.

!! **Exercise 4.4.3:** Suppose that p are q are distinguishable states of a given DFA A with n states. As a function of n, what is the tightest upper bound on how long the shortest string that distinguishes p from q can be?

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

Figure 4.15: Another DFA to minimize

4.5 Summary of Chapter 4

✦ *The Pumping Lemma*: If a language is regular, then every sufficiently long string in the language has a nonempty substring that can be "pumped," that is, repeated any number of times while the resulting strings are also in the language. This fact can be used to prove that many different languages are *not* regular.

✦ *Operations That Preserve the Property of Being a Regular Language*: There are many operations that, when applied to regular languages, yield a regular language as a result. Among these are union, concatenation, closure, intersection, complementation, difference, reversal, homomorphism (replacement of each symbol by an associated string), and inverse homomorphism.

✦ *Testing Emptiness of Regular Languages*: There is an algorithm that, given a representation of a regular language, such as an automaton or regular expression, tells whether or not the represented language is the empty set.

✦ *Testing Membership in a Regular Language*: There is an algorithm that, given a string and a representation of a regular language, tells whether or not the string is in the language.

✦ *Testing Distinguishability of States*: Two states of a DFA are distinguishable if there is an input string that takes exactly one of the two states to an accepting state. By starting with only the fact that pairs consisting of one accepting and one nonaccepting state are distinguishable, and trying to discover addition pairs of distinguishable states by finding pairs whose successors on one input symbol are distinguishable, we can discover all pairs of distinguishable states.