CIS 511
Formal Languages And Automata
Models of Computation, Computability
Basics of Recursive Function Theory

Jean Gallier
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104, USA
e-mail: jean@cis.upenn.edu

June 8, 2015

Contents

Chapter 1

Introduction

The theory of computation is concerned with algorithms and algorithmic systems: their design and representation, their completeness, and their complexity.

The purpose of these notes is to introduce some of the basic notions of the theory of computation, including concepts from formal languages and automata theory, and the theory of computability and some basics of recursive function theory. Other topics such as correctness of programs and computational complexity will not be treated here (there just isn't enough time!).

The notes are divided into two parts. The first part is devoted to formal languages and automata. The second part deals with models of computation, recursive functions, and undecidability.

Chapter 2

Regular Languages and Equivalence Relations, The Myhill-Nerode Characterization, State Equivalence

2.1 Morphisms, F-Maps, B-Maps and Homomorphisms of DFA's

It is natural to wonder whether there is a reasonable notion of a mapping between DFA's. It turns out that this is indeed the case and there is a notion of a map between DFA's which is very useful in the theory of DFA minimization (given a DFA, find an equivalent DFA of minimal size). Obviously, a map between DFA's should be a certain kind of graph homomorphism, which means that given two DFA's $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, F_2)$, we have a function, $h\colon Q_1 \to Q_2$, mapping every state, $p \in Q_1$, of D_1, to some state, $q = h(p) \in Q_2$, of D_2 in such a way that for every input symbol, $a \in \Sigma$, the transition on a from p to $\delta_1(p, a)$ is mapped to the transition on a from $h(p)$ to $h(\delta_1(p, a))$, so that

$$h(\delta_1(p, a)) = \delta_2(h(p), a),$$

which can be expressed by the commutativity of the following diagram:

$$\begin{tikzcd}
p \arrow[r, "h"] \arrow[d, "a"'] & h(p) \arrow[d, "a"] \\
\delta_1(p,a) \arrow[r, "h"] & \delta_2(h(p),a)
\end{tikzcd}$$

In order to be useful, a map of DFA's, $h\colon D_1 \to D_2$, should induce a relationship between the languages, $L(D_1)$ and $L(D_2)$, such as $L(D_1) \subseteq L(D_2)$, $L(D_2) \subseteq L(D_1)$ or $L(D_1) = L(D_2)$. This can indeed be achieved by requiring some simple condition on the way final states are related by h.

For any function, $h\colon X \to Y$, and for any two subsets, $A \subseteq X$ and $B \subseteq Y$, recall that

$$h(A) = \{h(a) \in Y \mid a \in A\}$$

is the *(direct) image* of A by h and

$$h^{-1}(B) = \{x \in X \mid h(x) \in B\}$$

is the *inverse image* of B by h, and $h^{-1}(B)$ makes sense even if h is not invertible. The following Definition is adapted from Eilenberg [1] (*Automata, Languages and Machines, Vol A*, Academic Press, 1974; see Chapter III, Section 4).

Definition 2.1.1 Given two DFA's, $D_1 = (Q_1, \Sigma, \delta_1, q_{0,1}, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{0,2}, F_2)$, a *morphism,* $h\colon D_1 \to D_2$*, of DFA's* is a function, $h\colon Q_1 \to Q_2$, satisfying the following conditions:

(1)
$$h(\delta_1(p, a)) = \delta_2(h(p), a), \quad \text{for all } p \in Q_1 \text{ and all } a \in \Sigma,$$

which can be expressed by the commutativity of the following diagram:

$$\begin{tikzcd}
p \arrow[r, "h"] \arrow[d, "a"'] & h(p) \arrow[d, "a"] \\
\delta_1(p, a) \arrow[r, "h"] & \delta_2(h(p), a).
\end{tikzcd}$$

(2) $h(q_{0,1}) = q_{0,2}$.

An *F-map of DFA's*, for short, a *map*, is a morphism of DFA's, $h\colon D_1 \to D_2$, that satisfies the condition

(3a) $h(F_1) \subseteq F_2$.

A *B-map of DFA's* is a morphism of DFA's, $h\colon D_1 \to D_2$, that satisfies the condition

(3b) $h^{-1}(F_2) \subseteq F_1$.

A *proper homomorphism of DFA's*, for short, a *homomorphism*, is an F-map of DFA's that is also a B-map of DFA's namely, a homomorphism satisfies (3a) & (3b).

Now, for any function $f\colon X \to Y$ and any two subsets $A \subseteq X$ and $B \subseteq Y$, recall that

$$f(A) \subseteq B \quad \text{iff} \quad A \subseteq f^{-1}(B).$$

Thus, (3a) & (3b) is equivalent to the condition (3c) below, that is, a homomorphism of DFA's is a morphism satisfying the condition

(3c) $h^{-1}(F_2) = F_1$.

Note that the condition for being a proper homomorphism of DFA's (condition (3c)) is **not** equivalent to

$$h(F_1) = F_2.$$

Condition (3c) forces $h(F_1) = F_2 \cap h(Q_1)$, and furthermore, for every $p \in Q_1$, whenever $h(p) \in F_2$, then $p \in F_1$.

Figure 2.1 shows a map, h, of DFA's, with

$$\begin{aligned} h(A) = h(C) &= 0 \\ h(B) &= 1 \\ h(D) &= 2 \\ h(E) &= 3. \end{aligned}$$

It is easy to check that h is actually a (proper) homomorphism.

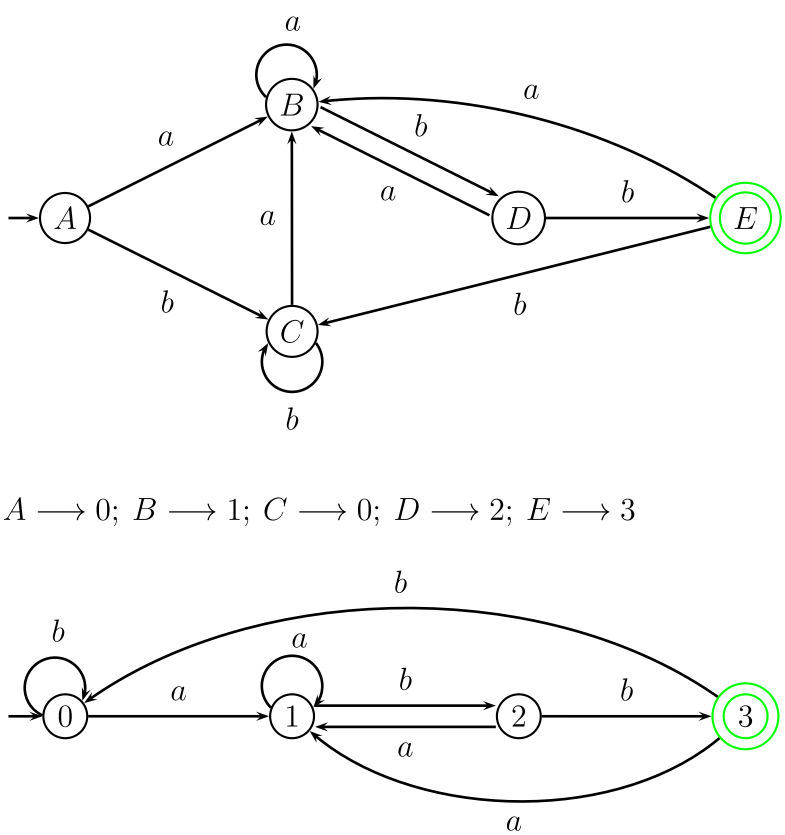

Figure 2.1: A map of DFA's

The reader should check that if $f\colon D_1 \to D_2$ and $g\colon D_2 \to D_3$ are morphisms (resp. F-maps, resp. B-maps), then $g \circ f\colon D_1 \to D_3$ is also a morphism (resp. an F-map, resp. a B-map).

Remark: In previous versions of these notes, an F-map was called simply a *map* and a B-map was called an F^{-1}-*map*. Over the years, the old terminology proved to be confusing. We hope the new one is less confusing!

Note that an F-map or a B-map is a special case of the concept of *simulation* of automata. A proper homomorphism is a special case of a *bisimulation*. Bisimulations play an important role in real-time systems and in concurrency theory.

The main motivation behind these definitions is that when there is an F-map $h\colon D_1 \to D_2$, somehow, D_2 simulates D_1, and it turns out that $L(D_1) \subseteq L(D_2)$.

When there is a B-map $h\colon D_1 \to D_2$, somehow, D_1 simulates D_2, and it turns out that $L(D_2) \subseteq L(D_1)$.

When there is a proper homomorphism $h\colon D_1 \to D_2$, somehow, D_1 bisimulates D_2, and it turns out that $L(D_2) = L(D_1)$.

A DFA morphism $f\colon D_1 \to D_2$, is an *isomorphism* iff there is a DFA morphism, $g\colon D_2 \to D_1$, so that

$$g \circ f = \mathrm{id}_{D_1} \quad \text{and} \quad f \circ g = \mathrm{id}_{D_2}.$$

Similarly an F-map $f\colon D_1 \to D_2$, is an *isomorphism* iff there is an F-map, $g\colon D_2 \to D_1$, so that

$$g \circ f = \mathrm{id}_{D_1} \quad \text{and} \quad f \circ g = \mathrm{id}_{D_2}.$$

Finally, a B-map $f\colon D_1 \to D_2$, is an *isomorphism* iff there is a B-map, $g\colon D_2 \to D_1$, so that

$$g \circ f = \mathrm{id}_{D_1} \quad \text{and} \quad f \circ g = \mathrm{id}_{D_2}.$$

The map g is unique and it is denoted f^{-1}. The reader should prove that if a DFA F-map is an isomorphism, then it is also a proper homomorphism and if a DFA B-map is an isomorphism, then it is also a proper homomorphism.

If $h\colon D_1 \to D_2$ is a morphism of DFA's, it is easily shown by induction on the length of w that

$$h(\delta_1^*(p, w)) = \delta_2^*(h(p), w),$$

for all $p \in Q_1$ and all $w \in \Sigma^*$, which corresponds to the commutativity of the following diagram:

$$\begin{tikzcd}
p \arrow[r, "h"] \arrow[d, "w"'] & h(p) \arrow[d, "w"] \\
\delta_1^*(p, w) \arrow[r, "h"] & \delta_2^*(h(p), w).
\end{tikzcd}$$

This is the generalization of the commutativity of the diagram in condition (1) of Definition 2.1.1, where any arbitrary string $w \in \Sigma^*$ is allowed instead of just a single symbol $a \in \Sigma$.

This is the crucial property of DFA morphisms. It says that for every string, $w \in \Sigma^*$, if we pick any state, $p \in Q_1$, as starting point in D_1, then the image of the path from p on input w in D_1 is the path in D_2 from the image, $h(p) \in Q_2$, of p on the same input, w. In particular, the image, $h(\delta_1^*(p, w))$ of the state reached from p on input w in D_1 is the state, $\delta_2^*(h(p), w)$, in D_2 reached from $h(p)$ on input w. For example, going back to the DFA map shown in Figure 2.1, the image of the path

$$C \xrightarrow{a} B \xrightarrow{b} D \xrightarrow{a} B \xrightarrow{b} D \xrightarrow{b} E$$

from C on input $w = ababb$ in D_1 is the path

$$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

from 0 on input $w = ababb$ in D_2.

As a consequence, we have the following Lemma:

Lemma 2.1.2 *If $h\colon D_1 \to D_2$ is an F-map of DFA's, then $L(D_1) \subseteq L(D_2)$. If $h\colon D_1 \to D_2$ is a B-map of DFA's, then $L(D_2) \subseteq L(D_1)$. Finally, if $h\colon D_1 \to D_2$ is a proper homomorphism of DFA's, then $L(D_1) = L(D_2)$.*

One might think that there may be many DFA morphisms between two DFA's D_1 and D_2, but this is not the case. In fact, if every state of D_1 is reachable from the start state, then there is at most one morphism from D_1 to D_2.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the set Q_r of *accessible or reachable states* is the subset of Q defined as

$$Q_r = \{p \in Q \mid \exists w \in \Sigma^*,\ \delta^*(q_0, w) = p\}.$$

The set Q_r can be easily computed by stages. A DFA is *accessible, or trim* if $Q = Q_r$; that is, if every state is reachable from the start state.

A morphism (resp. F-map, B-map) $h\colon D_1 \to D_2$ is *surjective* if $h(Q_1) = Q_2$.

The following lemma is easy to show:

Lemma 2.1.3 *If D_1 is trim, then there is at most one morphism $h\colon D_1 \to D_2$ (resp. F-map, resp. B-map). If D_2 is also trim and we have a morphism, $h\colon D_1 \to D_2$, then h is surjective.*

It can also be shown that a minimal DFA D_L for L is characterized by the property that there is unique surjective proper homomorphism $h\colon D \to D_L$ from any trim DFA D accepting L to D_L.

Another useful notion is the notion of a congruence on a DFA.

Definition 2.1.4 Given any DFA, $D = (Q, \Sigma, \delta, q_0, F)$, a *congruence* $\equiv$ *on* D is an equivalence relation $\equiv$ on Q satisfying the following conditions: For all $p, q \in Q$ and all $a \in \Sigma$,

(1) If $p \equiv q$, then $\delta(p, a) \equiv \delta(q, a)$.

(2) If $p \equiv q$ and $p \in F$, then $q \in F$.

It can be shown that a proper homomorphism of DFA's $h\colon D_1 \to D_2$ induces a congruence $\equiv_h$ on D_1 defined as follows:

$$p \equiv_h q \quad \text{iff} \quad h(p) = h(q).$$

Given a congruence $\equiv$ on a DFA D, we can define the *quotient DFA* $D/\equiv$, and there is a surjective proper homomorphism $\pi\colon D \to D/\equiv$.

We will come back to this point when we study minimal DFA's.

2.2 The Closure Definition of the Regular Languages

Let $\Sigma = \{a_1, \ldots, a_m\}$ be some alphabet. We would like to define a family of languages, $R(\Sigma)$, by singling out some very basic (atomic) languages, namely the languages $\{a_1\}, \ldots, \{a_m\}$, the empty language, and the trivial language, $\{\epsilon\}$, and then forming more complicated languages by repeatedly forming union, concatenation and Kleene $*$ of previously constructed languages. By doing so, we hope to get a family of languages ($R(\Sigma)$) that is closed under union, concatenation, and Kleene $*$. This means that for any two languages, $L_1, L_2 \in R(\Sigma)$, we also have $L_1 \cup L_2 \in R(\Sigma)$ and $L_1 L_2 \in R(\Sigma)$, and for any language $L \in R(\Sigma)$, we have $L^* \in R(\Sigma)$. Furthermore, we would like $R(\Sigma)$ to be the smallest family with these properties. How do we achieve this rigorously?

First, let us look more closely at what we mean by a family of languages. Recall that a language (over Σ) is *any* subset, L, of Σ^*. Thus, the set of all languages is 2^{Σ^*}, the power set of Σ^*. If Σ is nonempty, this is an uncountable set. Next, we define a *family*, $\mathcal{L}$, of languages to be any subset of 2^{Σ^*}. This time, the set of families of languages is $2^{2^{\Sigma^*}}$. This is a huge set. We can use the inclusion relation on $2^{2^{\Sigma^*}}$ to define a partial order on families of languages. So, $\mathcal{L}_1 \subseteq \mathcal{L}_2$ iff for every language, L, if $L \in \mathcal{L}_1$ then $L \in \mathcal{L}_2$.

We can now state more precisely what we are trying to do. Consider the following properties for a family of languages, $\mathcal{L}$:

(1) We have $\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\} \in \mathcal{L}$, i.e., $\mathcal{L}$ contains the "atomic" languages.

(2a) For all $L_1, L_2 \in \mathcal{L}$, we also have $L_1 \cup L_2 \in \mathcal{L}$.

(2b) For all $L_1, L_2 \in \mathcal{L}$, we also have $L_1 L_2 \in \mathcal{L}$.

(2c) For all $L \in \mathcal{L}$, we also have $L^* \in \mathcal{L}$.

In other words, $\mathcal{L}$ is closed under union, concatenation and Kleene $*$.

Now, what we want is the smallest (w.r.t. inclusion) family of languages that satisfies properties (1) and (2)(a)(b)(c). We can construct such a family using an *inductive definition*. This inductive definition constructs a sequence of families of languages, $(R(\Sigma)_n)_{n\geq 0}$, called the *stages of the inductive definition*, as follows:

$$\begin{aligned} R(\Sigma)_0 &= \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\}, \\ R(\Sigma)_{n+1} &= R(\Sigma)_n \cup \{L_1 \cup L_2,\ L_1L_2,\ L^* \ \mid L_1, L_2, L \in R(\Sigma)_n\}. \end{aligned}$$

Then, we define $R(\Sigma)$ by

$$R(\Sigma) = \bigcup_{n\geq 0} R(\Sigma)_n.$$

Thus, a language L belongs to $R(\Sigma)$ iff it belongs L_n, for some $n \geq 0$. Observe that

$$R(\Sigma)_0 \subseteq R(\Sigma)_1 \subseteq R(\Sigma)_2 \subseteq \cdots R(\Sigma)_n \subseteq R(\Sigma)_{n+1} \subseteq \cdots \subseteq R(\Sigma),$$

so that if $L \in R(\Sigma)_n$, then $L \in R(\Sigma)_p$, for all $p \geq n$. Also, there is some smallest n for which $L \in R(\Sigma)_n$ (the *birthdate* of L!). In fact, all these inclusions are strict. Note that each $R(\Sigma)_n$ only contains a finite number of languages (but some of the languages in $R(\Sigma)_n$ are infinite, because of Kleene $*$). Then we define the *Regular languages, Version 2*, as the family $R(\Sigma)$.

Of course, it is far from obvious that $R(\Sigma)$ coincides with the family of languages accepted by DFA's (or NFA's), what we call the regular languages, version 1. However, this is the case, and this can be demonstrated by giving two algorithms. Actually, it will be slightly more convenient to define a notation system, the *regular expressions*, to denote the languages in $R(\Sigma)$. Then, we will give an algorithm that converts a regular expression, R, into an NFA, N_R, so that $L_R = L(N_R)$, where L_R is the language (in $R(\Sigma)$) denoted by R. We will also give an algorithm that converts an NFA, N, into a regular expression, R_N, so that $L(R_N) = L(N)$.

But before doing all this, we should make sure that $R(\Sigma)$ is indeed the family that we are seeking. This is the content of

Lemma 2.2.1 *The family, $R(\Sigma)$, is the smallest family of languages which contains the atomic languages $\{a_1\}$, $\ldots, \{a_m\}$, $\emptyset$, $\{\epsilon\}$, and is closed under union, concatenation, and Kleene $*$.*

Proof. There are two things to prove.

(i) We need to prove that $R(\Sigma)$ has properties (1) and (2)(a)(b)(c).

(ii) We need to prove that $R(\Sigma)$ is the smallest family having properties (1) and (2)(a)(b)(c).

(i) Since

$$R(\Sigma)_0 = \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\},$$

it is obvious that (1) holds. Next, assume that $L_1, L_2 \in R(\Sigma)$. This means that there are some integers $n_1, n_2 \geq 0$, so that $L_1 \in R(\Sigma)_{n_1}$ and $L_2 \in R(\Sigma)_{n_2}$. Now, it is possible that $n_1 \neq n_2$, but if we let $n = \max\{n_1, n_2\}$, as we observed that $R(\Sigma)_p \subseteq R(\Sigma)_q$ whenever $p \leq q$, we are guaranteed that both $L_1, L_2 \in R(\Sigma)_n$. However, by the definition of $R(\Sigma)_{n+1}$ (that's why we defined it this way!), we have $L_1 \cup L_2 \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. The same argument proves that $L_1L_2 \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. Also, if $L \in R(\Sigma)_n$, we immediately have $L^* \in R(\Sigma)_{n+1} \subseteq R(\Sigma)$. Therefore, $R(\Sigma)$ has properties (1) and (2)(a)(b)(c).

(ii) Let $\mathcal{L}$ be any family of languages having properties (1) and (2)(a)(b)(c). We need to prove that $R(\Sigma) \subseteq \mathcal{L}$. If we can prove that $R(\Sigma)_n \subseteq \mathcal{L}$, for all $n \geq 0$, we are done (since then, $R(\Sigma) = \bigcup_{n\geq 0} R(\Sigma)_n \subseteq \mathcal{L}$). We prove by induction on n that $R(\Sigma)_n \subseteq \mathcal{L}$, for all $n \geq 0$.

The base case $n = 0$ is trivial, since $\mathcal{L}$ has (1), which says that $R(\Sigma)_0 \subseteq \mathcal{L}$. Assume inductively that $R(\Sigma)_n \subseteq \mathcal{L}$. We need to prove that $R(\Sigma)_{n+1} \subseteq \mathcal{L}$. Pick any $L \in R(\Sigma)_{n+1}$. Recall that

$$R(\Sigma)_{n+1} = R(\Sigma)_n \cup \{L_1 \cup L_2,\ L_1L_2,\ L^* \mid L_1, L_2, L \in R(\Sigma)_n\}.$$

If $L \in R(\Sigma)_n$, then $L \in \mathcal{L}$, since $R(\Sigma)_n \subseteq \mathcal{L}$, by the induction hypothesis. Otherwise, there are three cases:

(a) $L = L_1 \cup L_2$, where $L_1, L_2 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1, L_2 \in \mathcal{L}$; since $\mathcal{L}$ has 2(a), we have $L_1 \cup L_2 \in \mathcal{L}$.

(b) $L = L_1L_2$, where $L_1, L_2 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1, L_2 \in \mathcal{L}$; since $\mathcal{L}$ has 2(b), we have $L_1L_2 \in \mathcal{L}$.

(c) $L = L_1^*$, where $L_1 \in R(\Sigma)_n$. By the induction hypothesis, $R(\Sigma)_n \subseteq \mathcal{L}$, so, we get $L_1 \in \mathcal{L}$; since $\mathcal{L}$ has 2(c), we have $L_1^* \in \mathcal{L}$.

Thus, in all cases, we showed that $L \in \mathcal{L}$, and so, $R(\Sigma)_{n+1} \subseteq \mathcal{L}$, which proves the induction step. $\square$

Students should study carefully the above proof. Although simple, it is the prototype of many proofs appearing in the theory of computation.

2.3 Right-Invariant Equivalence Relations on Σ^*

The purpose of this section is to give one more characterization of the regular languages in terms of certain kinds of equivalence relations on strings. Pushing this characterization a bit further, we will be able to show how minimal DFA's can be found.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The DFA D may be redundant, for example, if there are states that are not accessible from the start state. As explained in Section 2.1, the set Q_r of *accessible or reachable states* is the subset of Q defined as

$$Q_r = \{p \in Q \mid \exists w \in \Sigma^*,\ \delta^*(q_0, w) = p\}.$$

If $Q \neq Q_r$, we can "clean up" D by deleting the states in $Q - Q_r$ and restricting the transition function δ to Q_r. This way, we get an equivalent DFA D_r such that $L(D) = L(D_r)$, where all the states of D_r are reachable. From now on, we assume that we are dealing with DFA's such that $D = D_r$, called *trim, or reachable*.

Recall that an *equivalence relation* $\simeq$ on a set A is a relation which is *reflexive*, *symmetric*, and *transitive*. Given any $a \in A$, the set

$$\{b \in A \mid a \simeq b\}$$

is called the *equivalence class of* a, and it is denoted as $[a]_{\simeq}$, or even as $[a]$. Recall that for any two elements $a, b \in A$, $[a] \cap [b] = \emptyset$ iff $a \not\simeq b$, and $[a] = [b]$ iff $a \simeq b$. The set of equivalence classes associated with the equivalence relation $\simeq$ is a *partition* Π of A (also denoted as $A/\simeq$). This means that it is a family of nonempty pairwise disjoint sets whose union is equal to A itself. The equivalence classes are also called the *blocks* of the partition Π. The number of blocks in the partition Π is called the *index* of $\simeq$ (and Π).

Given any two equivalence relations $\simeq_1$ and $\simeq_2$ with associated partitions Π_1 and Π_2,

$$\simeq_1 \subseteq \simeq_2$$

iff every block of the partition Π_1 is contained in some block of the partition Π_2. Then, every block of the partition Π_2 is the union of blocks of the partition Π_1, and we say that $\simeq_1$ is a *refinement* of $\simeq_2$ (and similarly, Π_1 is a refinement of Π_2). Note that Π_2 has at most as many blocks as Π_1 does.

We now define an equivalence relation on strings induced by a DFA. This equivalence is a kind of "observational" equivalence, in the sense that we decide that two strings u, v are equivalent iff, when feeding first u and then v to the DFA, u and v drive the DFA to the same state. From the point of view of the observer, u and v have the same effect (reaching the same state).

Definition 2.3.1 Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, we define the relation $\simeq_D$ on Σ^* as follows: for any two strings $u, v \in \Sigma^*$,

$$u \simeq_D v \quad \text{iff} \quad \delta^*(q_0, u) = \delta^*(q_0, v).$$

We can figure out what the equivalence classes of $\simeq_D$ are for the following DFA:

	a	b
0	1	0
1	2	1
2	0	2

with 0 both start state and (unique) final state. For example

$$\begin{aligned} abbabbb &\simeq_D aa \\ ababab &\simeq_D \epsilon \\ bba &\simeq_D a. \end{aligned}$$

There are three equivalences classes:

$$[\epsilon]_{\simeq}, \quad [a]_{\simeq}, \quad [aa]_{\simeq}.$$

Observe that $L(D) = [\epsilon]_{\simeq}$. Also, the equivalence classes are in one–to–one correspondence with the states of D.

The relation $\simeq_D$ turns out to have some interesting properties. In particular, it is *right-invariant*, which means that for all $u, v, w \in \Sigma^*$, if $u \simeq v$, then $uw \simeq vw$.

Lemma 2.3.2 *Given any (accessible) DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\simeq_D$ is an equivalence relation which is right-invariant and has finite index. Furthermore, if Q has n states, then the index of $\simeq_D$ is n, and every equivalence class of $\simeq_D$ is a regular language. Finally, $L(D)$ is the union of some of the equivalence classes of $\simeq_D$.*

Proof. The fact that $\simeq_D$ is an equivalence relation is a trivial verification. To prove that $\simeq_D$ is right-invariant, we first prove by induction on the length of v that for all $u, v \in \Sigma^*$, for all $p \in Q$,

$$\delta^*(p, uv) = \delta^*(\delta^*(p, u), v).$$

Then, if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, we have

$$\delta^*(q_0, uw) = \delta^*(\delta^*(q_0, u), w) = \delta^*(\delta^*(q_0, v), w) = \delta^*(q_0, vw),$$

which means that $uw \simeq_D vw$. Thus, $\simeq_D$ is right-invariant. We still have to prove that $\simeq_D$ has index n. Define the function $f \colon \Sigma^* \to Q$ such that

$$f(u) = \delta^*(q_0, u).$$

Note that if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, then $f(u) = f(v)$. Thus, the function $f \colon \Sigma^* \to Q$ induces a function $\widehat{f} \colon \Pi \to Q$ defined such that

$$\widehat{f}([u]) = f(u),$$

for every equivalence class $[u] \in \Pi$, where $\Pi = \Sigma^*/\simeq$ is the partition associated with $\simeq_D$. However, the function $\widehat{f}\colon \Pi \to Q$ is injective (one-to-one), since $\widehat{f}([u]) = \widehat{f}([v])$ means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, which means precisely that $u \simeq_D v$, i.e., $[u] = [v]$. Since Q has n states, Π has at most n blocks. Moreover, since every state is accessible, for every $q \in Q$, there is some $w \in \Sigma^*$ so that $\delta^*(q_0, w) = q$, which shows that $\widehat{f}([w]) = f(w) = q$. Consequently, $\widehat{f}$ is also surjective. But then, being injective and surjective, $\widehat{f}$ is bijective and Π has exactly n blocks.

Every equivalence class of Π is a set of strings of the form

$$\{w \in \Sigma^* \mid \delta^*(q_0, w) = p\},$$

for some $p \in Q$, which is accepted by the DFA

$$D_p = (Q, \Sigma, \delta, q_0, \{p\})$$

obtained from D by changing F to $\{p\}$. Thus, every equivalence class is a regular language. Finally, since

$$\begin{aligned} L(D) &= \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\} \\ &= \bigcup_{f \in F} \{w \in \Sigma^* \mid \delta^*(q_0, w) = f\} \\ &= \bigcup_{f \in F} L(D_f), \end{aligned}$$

we see that $L(D)$ is the union of the equivalence classes corresponding to the final states in F. $\square$

The remarkable fact due to Myhill and Nerode, is that Lemma 2.3.2 has a converse.

Lemma 2.3.3 *Given any equivalence relation $\simeq$ on Σ^*, if $\simeq$ is right-invariant and has finite index n, then every equivalence class (block) in the partition Π associated with $\simeq$ is a regular language.*

Proof. Let $C_1, \ldots, C_n$ be the blocks of Π, and assume that $C_1 = [\epsilon]$ is the equivalence class of the empty string.

First, we claim that for every block C_i and every $w \in \Sigma^*$, there is a unique block C_j such that $C_i w \subseteq C_j$, where $C_i w = \{uw \mid u \in C_i\}$.

For every $u \in C_i$, the string uw belongs to one and only one of the blocks of Π, say C_j. For any other string $v \in C_i$, since (by definition) $u \simeq v$, by right invariance, we get $uw \simeq vw$, but since $uw \in C_j$ and C_j is an equivalence class, we also have $vw \in C_j$. This proves the first claim.

We also claim that for every $w \in \Sigma^*$, for every block C_i,

$$C_1 w \subseteq C_i \quad \text{iff} \quad w \in C_i.$$

If $C_1 w \subseteq C_i$, since $C_1 = [\epsilon]$, we have $\epsilon w = w \in C_i$. Conversely, if $w \in C_i$, for any $v \in C_1 = [\epsilon]$, since $\epsilon \simeq v$, by right invariance we have $w \simeq vw$, and thus $vw \in C_i$, which shows that $C_1 w \subseteq C_i$.

For every class C_k, let

$$D_k = (\{1, \ldots, n\}, \Sigma, \delta, 1, \{k\}),$$

where $\delta(i, a) = j$ iff $C_i a \subseteq C_j$. We will prove the following equivalence:

$$\delta^*(i, w) = j \quad \text{iff} \quad C_i w \subseteq C_j.$$

For this, we prove the following two implications by induction on $|w|$:

(a) If $\delta^*(i, w) = j$, then $C_i w \subseteq C_j$, and

(b) If $C_i w \subseteq C_j$, then $\delta^*(i, w) = j$.

The base case ($w = \epsilon$) is trivial for both (a) and (b). We leave the proof of the induction step for (a) as an exercise and give the proof of the induction step for (b) because it is more subtle. Let $w = ua$, with $a \in \Sigma$ and $u \in \Sigma^*$. If $C_i ua \subseteq C_j$, then by the first claim, we know that there is a unique block, C_k, such that $C_i u \subseteq C_k$. Furthermore, there is a unique block, C_h, such that $C_k a \subseteq C_h$, but $C_i u \subseteq C_k$ implies $C_i ua \subseteq C_k a$ so we get $C_i ua \subseteq C_h$. However, by the uniqueness of the block, C_j, such that $C_i ua \subseteq C_j$, we must have $C_h = C_j$. By the induction hypothesis, as $C_i u \subseteq C_k$, we have

$$\delta^*(i, u) = k$$

and, by definition of δ, as $C_k a \subseteq C_j$ $(= C_h)$, we have $\delta(k, a) = j$, so we deduce that

$$\delta^*(i, ua) = \delta(\delta^*(i, u), a) = \delta(k, a) = j,$$

as desired. Then, using the equivalence just proved and the second claim, we have

$$\begin{aligned} L(D_k) &= \{w \in \Sigma^* \mid \delta^*(1, w) \in \{k\}\} \\ &= \{w \in \Sigma^* \mid \delta^*(1, w) = k\} \\ &= \{w \in \Sigma^* \mid C_1 w \subseteq C_k\} \\ &= \{w \in \Sigma^* \mid w \in C_k\} = C_k, \end{aligned}$$

proving that every block, C_k, is a regular language. $\square$

In general it is false that $C_i a = C_j$ for some block C_j, and we can only claim that $C_i a \subseteq C_j$.

We can combine Lemma 2.3.2 and Lemma 2.3.3 to get the following characterization of a regular language due to Myhill and Nerode:

Theorem 2.3.4 *(Myhill-Nerode) A language L (over an alphabet Σ) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation $\simeq$ on Σ^*, which is right-invariant and has finite index.*

Given two DFA's D_1 and D_2, whether or not there is a morphism $h: D_1 \to D_2$ depends on the relationship between $\simeq_{D_1}$ and $\simeq_{D_2}$. More specifically, we have the following lemma:

Lemma 2.3.5 *Given two DFA's D_1 and D_2, with D_1 trim, the following properties hold:*

(1) There is a DFA morphism $h: D_1 \to D_2$ iff

$$\simeq_{D_1} \subseteq \simeq_{D_2} .$$

(2) There is a DFA F-map $h: D_1 \to D_2$ iff

$$\simeq_{D_1} \subseteq \simeq_{D_2} \quad \textit{and} \quad L(D_1) \subseteq L(D_2);$$

(3) There is a DFA B-map $h: D_1 \to D_2$ iff

$$\simeq_{D_1} \subseteq \simeq_{D_2} \quad \textit{and} \quad L(D_2) \subseteq L(D_1).$$

Furthermore, h is surjective iff D_2 is trim.

Theorem 2.3.4 can also be used to prove that certain languages are not regular. A general scheme (not the only one) goes as follows: If L is not regular, then it must be infinite. Now, we argue by contradiction. If L was regular, then by Myhill-Nerode, there would be some equivalence relation, $\simeq$, which is right-invariant and of finite index and such that L is the union of some of the classes of $\simeq$. Because Σ^* is infinite and $\simeq$ has only finitely many equivalence classes, there are strings $x, y \in \Sigma^*$ with $x \neq y$ so that

$$x \simeq y.$$

If we can find a third string, $z \in \Sigma^*$, such that

$$xz \in L \quad \text{and} \quad yz \notin L,$$

then we reach a contradiction. Indeed, by right invariance, from $x \simeq y$, we get $xz \simeq yz$. But, L is the union of equivalence classes of $\simeq$, so if $xz \in L$, then we should also have $yz \in L$, contradicting $yz \notin L$. Therefore, L is not regular.

For example, we prove that $L = \{a^n b^n \mid n \geq 1\}$ is not regular.

Assuming for the sake of contradiction that L is regular, there is some equivalence relation $\simeq$ which is right-invariant and of finite index and such that L is the union of some of the classes of $\simeq$. Since the set

$$\{a, aa, aaa, \ldots, a^i, \ldots\}$$

is infinite and $\simeq$ has a finite number of classes, two of these strings must belong to the same class, which means that $a^i \simeq a^j$ for some $i \neq j$. But since $\simeq$ is right invariant, by concatenating with b^i on the right, we see that $a^i b^i \simeq a^j b^i$ for some $i \neq j$. However $a^i b^i \in L$, and since L is the union of classes of $\simeq$, we also have $a^j b^i \in L$ for $i \neq j$, which is absurd, given the definition of L. Thus, in fact, L is not regular.

Here is another illustration of the use of the Myhill-Nerode Theorem to prove that a language is not regular. We claim that the language,

$$L = \{a^{n!} \mid n \geq 1\},$$

is not regular, where $n!$ (n factorial) is given by $0! = 1$ and $(n+1)! = (n+1)n!$.

Assume L is regular. Then, there is some equivalence relation $\simeq$ which is right-invariant and of finite index and such that L is the union of some of the classes of $\simeq$. Since the sequence

$$a, a^2, \ldots, a^n, \ldots$$

is infinite, two of these strings must belong to the same class, which means that $a^p \simeq a^q$ for some p, q with $1 \leq p < q$. As $q! \geq q$ for all $q \geq 0$ and $q > p$, we can concatenate on the right with $a^{q!-p}$ and we get

$$a^p a^{q!-p} \simeq a^q a^{q!-p},$$

that is,

$$a^{q!} \simeq a^{q!+q-p}.$$

If we can show that

$$q! + q - p < (q+1)!$$

we will obtain a contradiction because then $a^{q!+q-p} \notin L$, yet $a^{q!+q-p} \simeq a^{q!}$ and $a^{q!} \in L$, contradicting Myhill-Nerode. Now, as $1 \leq p < q$, we have $q - p \leq q - 1$, so if we can prove that

$$q! + q - p \leq q! + q - 1 < (q+1)!$$

we will be done. However, $q! + q - 1 < (q+1)!$ is equivalent to

$$q - 1 < (q+1)! - q!,$$

and since $(q+1)! - q! = (q+1)q! - q! = qq!$, we simply need to prove that

$$q - 1 < q \leq qq!,$$

which holds for $q \geq 1$.

There is another version of the Myhill-Nerode Theorem involving congruences which is also quite useful. An equivalence relation, $\simeq$, on Σ^* is *left and right-invariant* iff for all $x, y, u, v \in \Sigma^*$,

$$\text{if} \quad x \simeq y \quad \text{then} \quad uxv \simeq uyv.$$

An equivalence relation, $\simeq$, on Σ^* is a *congruence* iff for all $u_1, u_2, v_1, v_2 \in \Sigma^*$,

$$\text{if} \quad u_1 \simeq v_1 \quad \text{and} \quad u_2 \simeq v_2 \quad \text{then} \quad u_1u_2 \simeq v_1v_2.$$

It is easy to prove that an equivalence relation is a congruence iff it is left and right-invariant, the proof is left as an exercise.

There is a version of Lemma 2.3.2 that applies to congruences and for this we define the relation $\sim_D$ as follows: For any (trim) DFA, $D = (Q, \Sigma, \delta, q_0, F)$, for all $x, y \in \Sigma^*$,

$$x \sim_D y \quad \text{iff} \quad (\forall q \in Q)(\delta^*(q, x) = \delta^*(q, y)).$$

Lemma 2.3.6 *Given any (trim) DFA, $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\sim_D$ is an equivalence relation which is left and right-invariant and has finite index. Furthermore, if Q has n states, then the index of $\sim_D$ is at most n^n and every equivalence class of $\sim_D$ is a regular language. Finally, $L(D)$ is the union of some of the equivalence classes of $\sim_D$.*

Proof. We leave the proof of Lemma 2.3.6 as an exercise. We just make the following remark: Observe that

$$\sim_D \subseteq \simeq_D,$$

since the condition $\delta^*(q, x) = \delta^*(q, y)$ holds for every $q \in Q$, so in particular for $q = q_0$. But then, every equivalence class of $\simeq_D$ is the union of equivalence classes of $\sim_D$ and since, by Lemma 2.3.2, L is the union of equivalence classes of $\simeq_D$, we conclude that L is also the union of equivalence classes of $\sim_D$. $\square$

Using Lemma 2.3.6 and Lemma 2.3.3, we obtain another version of the Myhill-Nerode Theorem.

Theorem 2.3.7 *(Myhill-Nerode, Conguence Version) A language L (over an alphabet Σ) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation $\simeq$ on Σ^*, which is a congruence and has finite index.*

Another useful tool for proving that languages are not regular is the so-called *pumping lemma*.

Lemma 2.3.8 *Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, there is some $m \geq 1$ such that for every $w \in \Sigma^*$, if $w \in L(D)$ and $|w| \geq m$, then there exists a decomposition of w as $w = uxv$, where*

(1) $x \neq \epsilon$,

(2) $ux^iv \in L(D)$, for all $i \geq 0$, and

(3) $|ux| \leq m$.

Moreover, m can be chosen to be the number of states of the DFA D.

Proof. Let m be the number of states in Q, and let $w = w_1 \dots w_n$. Since Q contains the start state q_0, $m \geq 1$. Since $|w| \geq m$, we have $n \geq m$. Since $w \in L(D)$, let $(q_0, q_1, \dots, q_n)$, be the sequence of states in the accepting computation of w (where $q_n \in F$). Consider the subsequence

$$(q_0, q_1, \dots, q_m).$$

This sequence contains $m + 1$ states, but there are only m states in Q, and thus, we have $q_i = q_j$, for some i, j such that $0 \leq i < j \leq m$. Then, letting $u = w_1 \dots w_i$, $x = w_{i+1} \dots w_j$, and $v = w_{j+1} \dots w_n$, it is clear that the conditions of the lemma hold. $\square$

Typically, the pumping lemma is used to prove that a language is not regular. The method is to proceed by contradiction, i.e., to assume (contrary to what we wish to prove) that a language L is indeed regular, and derive a contradiction of the pumping lemma. Thus, it would be helpful to see what the negation of the pumping lemma is, and for this, we first state the pumping lemma as a logical formula. We will use the following abbreviations:

$$\begin{aligned}
\mathit{nat} &= \{0, 1, 2, \dots\}, \\
\mathit{pos} &= \{1, 2, \dots\}, \\
A &\equiv w = uxv, \\
B &\equiv x \neq \epsilon, \\
C &\equiv |ux| \leq m, \\
P &\equiv \forall i\colon \mathit{nat}\, (ux^iv \in L(D)).
\end{aligned}$$

The pumping lemma can be stated as

$$\forall D\colon \mathrm{DFA}\, \exists m\colon \mathit{pos}\, \forall w\colon \Sigma^* \Big((w \in L(D) \land |w| \geq m) \supset (\exists u, x, v\colon \Sigma^*\, A \land B \land C \land P) \Big).$$

Recalling that

$$\neg(A \land B \land C \land P) \equiv \neg(A \land B \land C) \lor \neg P \equiv (A \land B \land C) \supset \neg P$$

and

$$\neg(R \supset S) \equiv R \land \neg S,$$

the negation of the pumping lemma can be stated as

$$\exists D\colon \mathrm{DFA}\, \forall m\colon \mathit{pos}\, \exists w\colon \Sigma^* \Big((w \in L(D) \land |w| \geq m) \land (\forall u, x, v\colon \Sigma^*\, (A \land B \land C) \supset \neg P) \Big).$$

Since

$$\neg P \equiv \exists i\colon \mathit{nat}\, (ux^iv \notin L(D)),$$

in order to show that the pumping lemma is contradicted, one needs to show that for some DFA D, for every $m \geq 1$, there is some string $w \in L(D)$ of length at least m, such that for every possible decomposition $w = uxv$ satisfying the constraints $x \neq \epsilon$ and $|ux| \leq m$, there is some $i \geq 0$ such that $ux^i v \notin L(D)$. When proceeding by contradiction, we have a language L that we are (wrongly) assuming to be regular, and we can use any DFA D accepting L. The creative part of the argument is to pick the right $w \in L$ (not making any assumption on $m \leq |w|$).

As an illustration, let us use the pumping lemma to prove that $L = \{a^n b^n \mid n \geq 1\}$ is not regular. The usefulness of the condition $|ux| \leq m$ lies in the fact that it reduces the number of legal decomposition uxv of w. We proceed by contradiction. Thus, let us assume that $L = \{a^n b^n \mid n \geq 1\}$ is regular. If so, it is accepted by some DFA D. Now, we wish to contradict the pumping lemma. For every $m \geq 1$, let $w = a^m b^m$. Clearly, $w = a^m b^m \in L$ and $|w| \geq m$. Then, every legal decomposition u, x, v of w is such that

$$w = \underbrace{a \ldots a}_{u}\underbrace{a \ldots a}_{x}\underbrace{a \ldots a b \ldots b}_{v}$$

where $x \neq \epsilon$ and x ends within the a's, since $|ux| \leq m$. Since $x \neq \epsilon$, the string $uxxv$ is of the form $a^n b^m$ where $n > m$, and thus $uxxv \notin L$, contradicting the pumping lemma.

We now consider an equivalence relation associated with a language L.

2.4 Finding minimal DFA's

Given any language L (not necessarily regular), we can define an equivalence relation ρ_L which is right-invariant, but not necessarily of finite index. However, when L is regular, the relation ρ_L has finite index. In fact, this index is the size of a smallest DFA accepting L. This will lead us to a construction of minimal DFA's.

Definition 2.4.1 Given any language L (over Σ), we define the relation ρ_L on Σ^* as follows: for any two strings $u, v \in \Sigma^*$,

$$u \rho_L v \quad \text{iff} \quad \forall w \in \Sigma^* (uw \in L \quad \text{iff} \quad vw \in L).$$

It is clear that the relation ρ_L is an equivalence relation, and it is right-invariant. To show right-invariance, argue as follows: if $u \rho_L v$, then for any $w \in \Sigma^*$, since $u \rho_L v$ means that

$$uz \in L \quad \text{iff} \quad vz \in L$$

for all $z \in \Sigma^*$, in particular the above equivalence holds for all z of the form $z = wy$ for any arbitary $y \in \Sigma^*$, so we have

$$uwy \in L \quad \text{iff} \quad vwy \in L$$

for all $y \in \Sigma^*$, which means that $uw \rho_L vw$.

It is also clear that L is the union of the equivalence classes of strings in L. This is because if $u \in L$ and $u \rho_L v$, by letting $w = \epsilon$ in the definition of ρ_L, we get

$$u \in L \quad \text{iff} \quad v \in L,$$

and since $u \in L$, we also have $v \in L$. This implies that if $u \in L$ then $[u]_{\rho_L} \subseteq L$ and so,

$$L = \bigcup_{u \in L} [u]_{\rho_L}.$$

When L is also regular, we have the following remarkable result:

Lemma 2.4.2 *Given any regular language L, for any (accessible) DFA $D = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(D)$, ρ_L is a right-invariant equivalence relation, and we have $\simeq_D \subseteq \rho_L$. Furthermore, if ρ_L has m classes and Q has n states, then $m \leq n$.*

Proof. By definition, $u \simeq_D v$ iff $\delta^*(q_0, u) = \delta^*(q_0, v)$. Since $w \in L(D)$ iff $\delta^*(q_0, w) \in F$, the fact that $u \rho_L v$ can be expressed as

$$\begin{aligned} \forall w \in \Sigma^*(uw \in L \quad \text{iff} \quad vw \in L), \qquad \text{iff}, \\ \forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \quad \text{iff} \quad \delta^*(q_0, vw) \in F), \qquad \text{iff} \\ \forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F), \end{aligned}$$

and if $\delta^*(q_0, u) = \delta^*(q_0, v)$, this shows that $u \rho_L v$. Since the number of classes of $\simeq_D$ is n and $\simeq_D \subseteq \rho_L$, the equivalence relation ρ_L has fewer classes than $\simeq_D$, and $m \leq n$. $\square$

Lemma 2.4.2 shows that when L is regular, the index m of ρ_L is finite, and it is a lower bound on the size of all DFA's accepting L. It remains to show that a DFA with m states accepting L exists. However, going back to the proof of Lemma 2.3.3 starting with the right-invariant equivalence relation ρ_L of finite index m, if L is the union of the classes $C_{i_1}, \ldots, C_{i_k}$, the DFA

$$D_{\rho_L} = (\{1, \ldots, m\}, \Sigma, \delta, 1, \{i_1, \ldots, i_k\}),$$

where $\delta(i, a) = j$ iff $C_i a \subseteq C_j$, is such that $L = L(D_{\rho_L})$. Thus, D_{ρ_L} is a minimal DFA accepting L.

In the next section, we give an algorithm which allows us to find D_{ρ_L}, given any DFA D accepting L. This algorithms finds which states of D are equivalent.

2.5 State Equivalence and Minimal DFA's

The proof of Lemma 2.4.2 suggests the following definition of an equivalence between states:

Definition 2.5.1 Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\equiv$ on Q, called *state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^*(\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

When $p \equiv q$, we say that *p and q are indistinguishable*.

It is trivial to verify that $\equiv$ is an equivalence relation, and that it satisfies the following property:

$$\text{if } p \equiv q \text{ then } \delta(p, a) \equiv \delta(q, a),$$

for all $a \in \Sigma$.

The reader should check that states A and C in the DFA below are equivalent and that no other distinct states are equivalent.

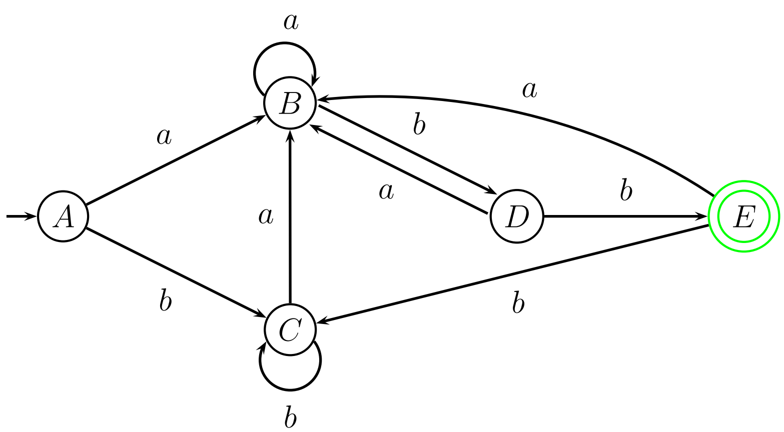

Figure 2.2: A non-minimal DFA for $\{a, b\}^*\{abb\}$

It might be illuminating to express state equivalence as the equality of two languages. Given the DFA $D = (Q, \Sigma, \delta, q_0, F)$, let $D_p = (Q, \Sigma, \delta, p, F)$ be the DFA obtained from D by redefining the start state to be p. Then, it is clear that

$$p \equiv q \quad \text{iff} \quad L(D_p) = L(D_q).$$

This simple observation implies that there is an algorithm to test state equivalence. Indeed, we simply have to test whether the DFA's D_p and D_q accept the same language and this can be done using the cross-product construction. Indeed, $L(D_p) = L(D_q)$ iff $L(D_p) - L(D_q) = \emptyset$ and $L(D_q) - L(D_p) = \emptyset$. Now, if $(D_p \times D_q)_{1-2}$ denotes the cross-product DFA with starting state (p, q) and with final states $F \times (Q - F)$ and $(D_p \times D_q)_{2-1}$ denotes the cross-product DFA also with starting state (p, q) and with final states $(Q - F) \times F$, we know that

$$L((D_p \times D_q)_{1-2}) = L(D_p) - L(D_q) \quad \text{and} \quad L((D_p \times D_q)_{2-1}) = L(D_q) - L(D_p),$$

so all we need to do if to test whether $(D_p \times D_q)_{1-2}$ and $(D_p \times D_q)_{2-1}$ accept the empty language. However, we know that this is the case iff the set of states reachable from (p, q) in $(D_p \times D_q)_{1-2}$ contains no state in $F \times (Q - F)$ and the set of states reachable from (p, q) in $(D_p \times D_q)_{2-1}$ contains no state in $(Q - F) \times F$.

Actually, the graphs of $(D_p \times D_q)_{1-2}$ and $(D_p \times D_q)_{2-1}$ are identical, so we only need to check that no state in $(F \times (Q - F)) \cup ((Q - F) \times F)$ is reachable from (p, q) in that graph. This algorithm to test state equivalence is not the most efficient but it is quite reasonable (it runs in polynomial time).

If $L = L(D)$, the lemma below shows the relationship between ρ_L and $\equiv$ and, more generally, between the DFA, D_{ρ_L}, and the DFA, $D/\equiv$, obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q and defined as follows:

$$D/\equiv = (Q/\equiv, \Sigma, \delta/\equiv, [q_0]_{\equiv}, F/\equiv),$$

where

$$\delta/\equiv([p]_{\equiv}, a) = [\delta(p, a)]_{\equiv}.$$

The DFA $D/\equiv$ is obtained by merging the states in each block of the partition Π associated with $\equiv$, forming states corresponding to the blocks of Π, and drawing a transition on input a from a block C_i to a block C_j of Π iff there is a transition $q = \delta(p, a)$ from any state $p \in C_i$ to any state $q \in C_j$ on input a. The start state is the block containing q_0, and the final states are the blocks consisting of final states.

Lemma 2.5.2 *For any (accessible) DFA $D = (Q, \Sigma, \delta, q_0, F)$ accepting the regular language $L = L(D)$, the function $\varphi\colon \Sigma^* \to Q$ defined such that*

$$\varphi(u) = \delta^*(q_0, u)$$

induces a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$, defined such that

$$\widehat{\varphi}([u]_{\rho_L}) = [\delta^*(q_0, u)]_{\equiv}.$$

Furthermore, we have

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L} \quad \textit{iff} \quad \delta(\varphi(u), a) \equiv \varphi(v).$$

Consequently, $\widehat{\varphi}$ induces an isomorphism of DFA's, $\widehat{\varphi}\colon D_{\rho_L} \to D/\equiv$ (i.e., an invertible F-map whose inverse is also an F-map; we know from a homework problem that such a map, $\widehat{\varphi}$, must be a proper homomorphism whose inverse is also a proper homomorphism).

Proof. Since $\varphi(u) = \delta^*(q_0, u)$ and $\varphi(v) = \delta^*(q_0, v)$, the fact that $\varphi(u) \equiv \varphi(v)$ can be expressed as

$$\begin{aligned} \forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \quad \text{iff} \quad \delta^*(\delta^*(q_0, v), w) \in F), \qquad \text{iff} \\ \forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \quad \text{iff} \quad \delta^*(q_0, vw) \in F), \end{aligned}$$

which is exactly $u\rho_L v$. Therefore,

$$u\rho_L v \quad \text{iff} \quad \varphi(u) \equiv \varphi(v).$$

From the above, we see that the function $\varphi\colon \Sigma^* \to Q$ maps each equivalence class $[u]$ modulo ρ_L to the equivalence class $[\varphi(u)]$ modulo $\equiv$ and so, the function $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$ given by

$$\widehat{\varphi}([u]_{\rho_L}) = [\delta^*(q_0, u)]_{\equiv}$$

is well-defined. Moreover, $\widehat{\varphi}$ is injective, since $\widehat{\varphi}([u]) = \widehat{\varphi}([v])$ iff $\varphi(u) = \varphi(v)$ iff (from above) $u\rho_v v$ iff $[u] = [v]$. Since every state in Q is accessible, for every $q \in Q$, there is some $u \in \Sigma^*$ so that $\varphi(u) = \delta^*(q_0, u) = q$, so $\widehat{\varphi}([u]) = [q]_{\equiv}$ and $\widehat{\varphi}$ is surjective. Therefore, we have a bijection $\widehat{\varphi}\colon \Sigma^*/\rho_L \to Q/\equiv$.

Since $\varphi(u) = \delta^*(q_0, u)$, we have

$$\delta(\varphi(u), a) = \delta(\delta^*(q_0, u), a) = \delta^*(q_0, ua) = \varphi(ua),$$

and thus, $\delta(\varphi(u), a) \equiv \varphi(v)$ can be expressed as $\varphi(ua) \equiv \varphi(v)$. By the previous part, this is equivalent to $ua\rho_L v$, and we claim that this is equivalent to

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

First, if $[u]_{\rho_L} a \subseteq [v]_{\rho_L}$, then $ua \in [v]_{\rho_L}$, that is, $ua\rho_L v$. Conversely, if $ua\rho_L v$, then for every $u' \in [u]_{\rho_L}$, we have $u'\rho_L u$, so by right-invariance we get $u'a\rho_L ua$, and since $ua\rho_L v$, we get $u'a\rho_L v$, that is, $u'a \in [v]_{\rho_L}$. Since $u' \in [u]_{\rho_L}$ is arbitrary, we conclude that $[u]_{\rho_L} a \subseteq [v]_{\rho_L}$. Therefore, we proved that

$$\delta(\varphi(u), a) \equiv \varphi(v) \quad \text{iff} \quad [u]_{\rho_L} a \subseteq [v]_{\rho_L}.$$

It is then easy to check (do it!) that $\widehat{\varphi}$ induces an F-map of DFA's which is an isomorphism (i.e., an invertible F-map whose inverse is also an F-map), $\widehat{\varphi}\colon D_{\rho_L} \to D/\equiv$. $\square$

Lemma 2.5.2 shows that the DFA D_{ρ_L} is isomorphic to the DFA $D/\equiv$ obtained as the quotient of the DFA D modulo the equivalence relation $\equiv$ on Q. Since D_{ρ_L} is a minimal DFA accepting L, so is $D/\equiv$.

There are other characterizations of the regular languages. Among those, the characterization in terms of right derivatives is of particular interest because it yields an alternative construction of minimal DFA's.

Definition 2.5.3 Given any language, $L \subseteq \Sigma^*$, for any string, $u \in \Sigma^*$, the *right derivative of L by u*, denoted L/u, is the language

$$L/u = \{w \in \Sigma^* \mid uw \in L\}.$$

Theorem 2.5.4 *If $L \subseteq \Sigma^*$ is any language, then L is regular iff it has finitely many right derivatives. Furthermore, if L is regular, then all its right derivatives are regular and their number is equal to the number of states of the minimal DFA's for L.*

Proof. It is easy to check that

$$L/u = L/v \qquad \text{iff} \qquad u\rho_L v.$$

The above shows that ρ_L has a finite number of classes, say m, iff there is a finite number of right derivatives, say n, and if so, $m = n$. If L is regular, then we know that the number of equivalence classes of ρ_L is the number of states of the minimal DFA's for L, so the number of right derivatives of L is equal to the size of the minimal DFA's for L.

Conversely, if the number of derivatives is finite, say m, then ρ_L has m classes and by Myhill-Nerode, L is regular. It remains to show that if L is regular then every right derivative is regular.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting L. If $p = \delta^*(q_0, u)$, then let

$$D_p = (Q, \Sigma, \delta, p, F),$$

that is, D with with p as start state. It is clear that

$$L/u = L(D_p),$$

so L/u is regular for every $u \in \Sigma^*$. Also observe that if $|Q| = n$, then there are at most n DFA's D_p, so there is at most n right derivatives, which is another proof of the fact that a regular language has a finite number of right derivatives. $\square$

If L is regular then the construction of a minimal DFA for L can be recast in terms of right derivatives. Let $L/u_1, L/u_2, \ldots, L/u_m$ be the set of all the right derivatives of L. Of course, we may assume that $u_1 = \epsilon$. We form a DFA whose states are the right derivatives, L/u_i. For every state, L/u_i, for every $a \in \Sigma$, there is a transition on input a from L/u_i to $L/u_j = L/(u_i a)$. The start state is $L = L/u_1$ and the final states are the right derivatives, L/u_i, for which $\epsilon \in L/u_i$.

We leave it as an exercise to check that the above DFA accepts L. One way to do this is to recall that $L/u = L/v$ iff $u\rho_L v$ and to observe that the above construction mimics the construction of D_{ρ_L} as in the Myhill-Nerode lemma (Lemma 2.3.3). This DFA is minimal since the number of right derivatives is equal to the size of the minimal DFA's for L.

We now return to state equivalence. Note that if $F = \emptyset$, then $\equiv$ has a single block (Q), and if $F = Q$, then $\equiv$ has a single block (F). In the first case, the minimal DFA is the one state DFA rejecting all strings. In the second case, the minimal DFA is the one state DFA accepting all strings. When $F \neq \emptyset$ and $F \neq Q$, there are at least two states in Q, and $\equiv$ also has at least two blocks, as we shall see shortly. It remains to compute $\equiv$ explicitly. This is done using a sequence of approximations. In view of the previous discussion, we are assuming that $F \neq \emptyset$ and $F \neq Q$, which means that $n \geq 2$, where n is the number of states in Q.

Definition 2.5.5 Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, for every $i \geq 0$, the relation $\equiv_i$ on Q, called *i-state equivalence*, is defined as follows: for all $p, q \in Q$,

$$p \equiv_i q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i\,(\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

When $p \equiv_i q$, we say that *p and q are i-indistinguishable.*

Since state equivalence $\equiv$ is defined such that

$$p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^*(\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F),$$

we note that testing the condition

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

for all strings in Σ^* is equivalent to testing the above condition for all strings of length at most i for all $i \geq 0$, i.e.

$$p \equiv q \quad \text{iff} \quad \forall i \geq 0\, \forall w \in \Sigma^*,\ |w| \leq i\,(\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

Since $\equiv_i$ is defined such that

$$p \equiv_i q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i\,(\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F),$$

we conclude that

$$p \equiv q \quad \text{iff} \quad \forall i \geq 0\,(p \equiv_i q).$$

This identity can also be expressed as

$$\equiv\, = \bigcap_{i \geq 0} \equiv_i\ .$$

If we assume that $F \neq \emptyset$ and $F \neq Q$, observe that $\equiv_0$ has exactly two equivalence classes F and $Q - F$, since ϵ is the only string of length 0, and since the condition

$$\delta^*(p, \epsilon) \in F \quad \text{iff} \quad \delta^*(q, \epsilon) \in F$$

is equivalent to the condition

$$p \in F \quad \text{iff} \quad q \in F.$$

It is also obvious from the definition of $\equiv_i$ that

$$\equiv\, \subseteq \cdots \subseteq\, \equiv_{i+1}\, \subseteq\, \equiv_i\, \subseteq \cdots \subseteq\, \equiv_1\, \subseteq\, \equiv_0\ .$$

If this sequence was strictly decreasing for all $i \geq 0$, the partition associated with $\equiv_{i+1}$ would contain at least one more block than the partition associated with $\equiv_i$ and since we start with a partition with two blocks, the partition associated with $\equiv_i$ would have at least $i+2$ blocks.

But then, for $i = n-1$, the partition associated with $\equiv_{n-1}$ would have at least $n+1$ blocks, which is absurd since Q has only n states. Therefore, there is a smallest integer, $i_0 \leq n-2$, such that

$$\equiv_{i_0+1} = \equiv_{i_0} .$$

Thus, it remains to compute $\equiv_{i+1}$ from $\equiv_i$, which can be done using the following lemma: The lemma also shows that

$$\equiv = \equiv_{i_0} .$$

Lemma 2.5.6 *For any (accessible) DFA $D = (Q, \Sigma, \delta, q_0, F)$, for all $p, q \in Q$, $p \equiv_{i+1} q$ iff $p \equiv_i q$ and $\delta(p, a) \equiv_i \delta(q, a)$, for every $a \in \Sigma$. Furthermore, if $F \neq \emptyset$ and $F \neq Q$, there is a smallest integer $i_0 \leq n-2$, such that*

$$\equiv_{i_0+1} = \equiv_{i_0} = \equiv .$$

Proof. By the definition of the relation $\equiv_i$,

$$p \equiv_{i+1} q \quad \text{iff} \quad \forall w \in \Sigma^*,\ |w| \leq i+1\ (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F).$$

The trick is to observe that the condition

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

holds for all strings of length at most $i+1$ iff it holds for all strings of length at most i and for all strings of length between 1 and $i+1$. This is expressed as

$$\begin{gathered} p \equiv_{i+1} q \quad \text{iff} \\ \forall w \in \Sigma^*,\ |w| \leq i\ (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F) \quad \text{and} \\ \forall w \in \Sigma^*,\ 1 \leq |w| \leq i+1\ (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F). \end{gathered}$$

Obviously, the first condition in the conjunction is $p \equiv_i q$, and since every string w such that $1 \leq |w| \leq i+1$ can be written as au where $a \in \Sigma$ and $0 \leq |u| \leq i$, the second condition in the conjunction can be written as

$$\forall a \in \Sigma\, \forall u \in \Sigma^*,\ |u| \leq i\ (\delta^*(p, au) \in F \quad \text{iff} \quad \delta^*(q, au) \in F).$$

However, $\delta^*(p, au) = \delta^*(\delta(p, a), u)$ and $\delta^*(q, au) = \delta^*(\delta(q, a), u)$, so that the above condition is really

$$\forall a \in \Sigma\, (\delta(p, a) \equiv_i \delta(q, a)).$$

Thus, we showed that

$$p \equiv_{i+1} q \quad \text{iff} \quad p \equiv_i q \quad \text{and} \quad \forall a \in \Sigma\, (\delta(p, a) \equiv_i \delta(q, a)).$$

Thus, if $\equiv_{i+1} = \equiv_i$ for some $i \geq 0$, using induction, we also have $\equiv_{i+j} = \equiv_i$ for all $j \geq 1$. Since

$$\equiv = \bigcap_{i \geq 0} \equiv_i, \quad \equiv_{i+1} \subseteq \equiv_i,$$

and since we know that there is a smallest index say i_0, such that $\equiv_j = \equiv_{i_0}$, for all $j \geq i_0 + 1$, we have $\equiv = \equiv_{i_0}$. $\square$

Using Lemma 2.5.6, we can compute $\equiv$ inductively, starting from $\equiv_0 = (F, Q - F)$, and computing $\equiv_{i+1}$ from $\equiv_i$, until the sequence of partitions associated with the $\equiv_i$ stabilizes.

Note that if $F = Q$ or $F = \emptyset$, then $\equiv = \equiv_0$, and the inductive characterization of Lemma 2.5.6 holds trivially.

There are a number of algorithms for computing $\equiv$, or to determine whether $p \equiv q$ for some given $p, q \in Q$.

A simple method to compute $\equiv$ is described in Hopcroft and Ullman. It consists in forming a triangular array corresponding to all unordered pairs (p, q), with $p \neq q$ (the rows and the columns of this triangular array are indexed by the states in Q, where the entries are below the descending diagonal). Initially, the entry (p, q) is marked iff p and q are **not** 0-equivalent, which means that p and q are not both in F or not both in $Q - F$. Then, we process every unmarked entry on every row as follows: for any unmarked pair (p, q), we consider pairs $(\delta(p, a), \delta(q, a))$, for all $a \in \Sigma$. If any pair $(\delta(p, a), \delta(q, a))$ is already marked, this means that $\delta(p, a)$ and $\delta(q, a)$ are inequivalent, and thus p and q are inequivalent, and we mark the pair (p, q). We continue in this fashion, until at the end of a round during which all the rows are processed, nothing has changed. When the algorithm stops, all marked pairs are inequivalent, and all unmarked pairs correspond to equivalent states.

Let us illustrates the above method. Consider the following DFA accepting $\{a, b\}^*\{abb\}$:

	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

The start state is A, and the set of final states is $F = \{E\}$. (This is the DFA displayed in Figure 2.2.)

The initial (half) array is as follows, using $\times$ to indicate that the corresponding pair (say, (E, A)) consists of inequivalent states, and $\square$ to indicate that nothing is known yet.

B	$\square$			
C	$\square$	$\square$		
D	$\square$	$\square$	$\square$	
E	$\times$	$\times$	$\times$	$\times$
	A	B	C	D

After the first round, we have

B	$\square$			
C	$\square$	$\square$		
D	$\times$	$\times$	$\times$	
E	$\times$	$\times$	$\times$	$\times$
	A	B	C	D

After the second round, we have

B	$\times$			
C	$\square$	$\times$		
D	$\times$	$\times$	$\times$	
E	$\times$	$\times$	$\times$	$\times$
	A	B	C	D

Finally, nothing changes during the third round, and thus, only A and C are equivalent, and we get the four equivalence classes

$$(\{A, C\}, \{B\}, \{D\}, \{E\}).$$

We obtain the minimal DFA showed in Figure 2.3.

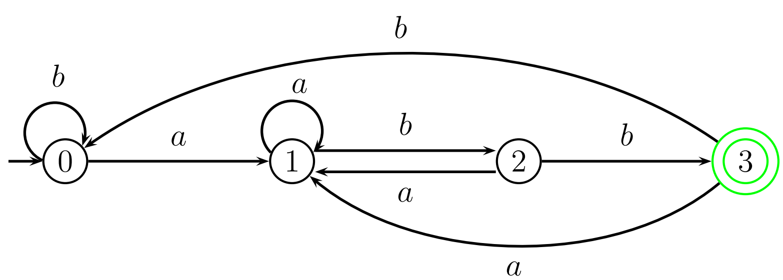

Figure 2.3: A minimal DFA accepting $\{a, b\}^*\{abb\}$

There are ways of improving the efficiency of this algorithm, see Hopcroft and Ullman for such improvements. Fast algorithms for testing whether $p \equiv q$ for some given $p, q \in Q$ also exist. One of these algorithms is based on "forward closures," following an idea of Knuth. Such an algorithm is related to a fast unification algorithm.

2.6 A Fast Algorithm for Checking State Equivalence Using a "Forward-Closure"

Given two states $p, q \in Q$, if $p \equiv q$, then we know that $\delta(p, a) \equiv \delta(q, a)$, for all $a \in \Sigma$. This suggests a method for testing whether two distinct states p, q are equivalent. Starting with the relation $R = \{(p, q)\}$, construct the smallest equivalence relation $R^{\dagger}$ containing R with the property that whenever $(r, s) \in R^{\dagger}$, then $(\delta(r, a), \delta(s, a)) \in R^{\dagger}$, for all $a \in \Sigma$. If we ever encounter a pair (r, s) such that $r \in F$ and $s \in \overline{F}$, or $r \in \overline{F}$ and $s \in F$, then r and s are inequivalent, and so are p and q. Otherwise, it can be shown that p and q are indeed equivalent. Thus, testing for the equivalence of two states reduces to finding an efficient method for computing the "forward closure" of a relation defined on the set of states of a DFA.

Such a method was worked out by John Hopcroft and Richard Karp and published in a 1971 Cornell technical report. This method is based on an idea of Donald Knuth for solving Exercise 11, in Section 2.3.5 of *The Art of Computer Programming*, Vol. 1, second edition, 1973. A sketch of the solution for this exercise is given on page 594. As far as I know, Hopcroft and Karp's method was never published in a journal, but a simple recursive algorithm does appear on page 144 of Aho, Hopcroft and Ullman's *The Design and Analysis of Computer Algorithms*, first edition, 1974. Essentially the same idea was used by Paterson and Wegman to design a fast unification algorithm (in 1978). We make a few definitions.

A relation $S \subseteq Q \times Q$ is a *forward closure* iff it is an equivalence relation and whenever $(r, s) \in S$, then $(\delta(r, a), \delta(s, a)) \in S$, for all $a \in \Sigma$. The *forward closure* of a relation $R \subseteq Q \times Q$ is the smallest equivalence relation $R^{\dagger}$ containing R which is forward closed.

We say that a forward closure S is *good* iff whenever $(r, s) \in S$, then $good(r, s)$, where $good(r, s)$ holds iff either both $r, s \in F$, or both $r, s \notin F$. Obviously, $bad(r, s)$ iff $\neg good(r, s)$.

Given any relation $R \subseteq Q \times Q$, recall that the smallest equivalence relation $R_{\approx}$ containing R is the relation $(R \cup R^{-1})^{*}$ (where $R^{-1} = \{(q, p) \mid (p, q) \in R\}$, and $(R \cup R^{-1})^{*}$ is the reflexive and transitive closure of $(R \cup R^{-1})$). The forward closure of R can be computed inductively by defining the sequence of relations $R_i \subseteq Q \times Q$ as follows:

$$\begin{aligned} R_0 &= R_{\approx} \\ R_{i+1} &= (R_i \cup \{(\delta(r, a), \delta(s, a)) \mid (r, s) \in R_i,\ a \in \Sigma\})_{\approx}. \end{aligned}$$

It is not hard to prove that $R_{i_0+1} = R_{i_0}$ for some least i_0, and that $R^{\dagger} = R_{i_0}$ is the smallest forward closure containing R. The following two facts can also been established.

(a) if $R^{\dagger}$ is good, then

$$R^{\dagger} \subseteq \equiv . \tag{2.1}$$

(b) if $p \equiv q$, then

$$R^{\dagger} \subseteq \equiv,$$

that is, equation (2.1) holds. This implies that $R^{\dagger}$ is good.

As a consequence, we obtain the correctness of our procedure: $p \equiv q$ iff the forward closure $R^{\dagger}$ of the relation $R = \{(p, q)\}$ is good.

In practice, we maintain a partition Π representing the equivalence relation that we are closing under forward closure. We add each new pair $(\delta(r, a), \delta(s, a))$ one at a time, and immediately form the smallest equivalence relation containing the new relation. If $\delta(r, a)$ and $\delta(s, a)$ already belong to the same block of Π, we consider another pair, else we merge the blocks corresponding to $\delta(r, a)$ and $\delta(s, a)$, and then consider another pair.

The algorithm is recursive, but it can easily be implemented using a stack. To manipulate partitions efficiently, we represent them as lists of trees (forests). Each equivalence class C in the partition Π is represented by a tree structure consisting of nodes and parent pointers, with the pointers from the sons of a node to the node itself. The root has a null pointer. Each node also maintains a counter keeping track of the number of nodes in the subtree rooted at that node.

Note that pointers can be avoided. We can represent a forest of n nodes as a list of n pairs of the form $(\mathit{father}, \mathit{count})$. If $(\mathit{father}, \mathit{count})$ is the ith pair in the list, then $\mathit{father} = 0$ iff node i is a root node, otherwise, *father* is the index of the node in the list which is the parent of node i. The number *count* is the total number of nodes in the tree rooted at the ith node.

For example, the following list of nine nodes

$$((0, 3), (0, 2), (1, 1), (0, 2), (0, 2), (1, 1), (2, 1), (4, 1), (5, 1))$$

represents a forest consisting of the following four trees:

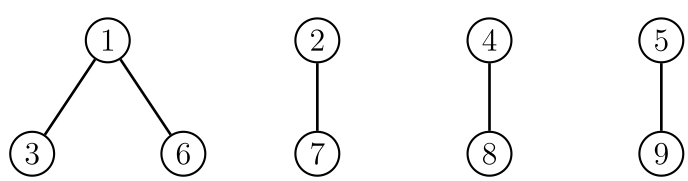

Figure 2.4: A forest of four trees

Two functions *union* and *find* are defined as follows. Given a state p, $\mathit{find}(p, \Pi)$ finds the root of the tree containing p as a node (not necessarily a leaf). Given two root nodes p, q, $\mathit{union}(p, q, \Pi)$ forms a new partition by merging the two trees with roots p and q as follows: if the counter of p is smaller than that of q, then let the root of p point to q, else let the root of q point to p.

For example, given the two trees shown on the left in Figure 2.5, *find*$(6, \Pi)$ returns 3 and *find*$(8, \Pi)$ returns 4. Then *union*$(3, 4, \Pi)$ yields the tree shown on the right in Figure 2.5.

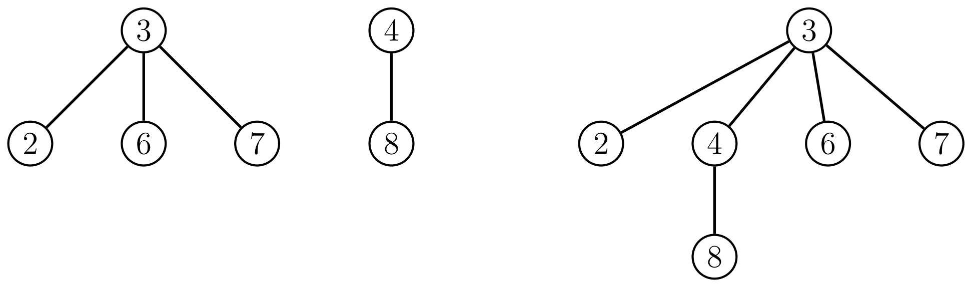

Figure 2.5: Applying the function *union* to the trees rooted at 3 and 4

In order to speed up the algorithm, using an idea due to Tarjan, we can modify *find* as follows: during a call *find*(p, Π), as we follow the path from p to the root r of the tree containing p, we redirect the parent pointer of every node q on the path from p (including p itself) to r (we perform *path compression*). For example, applying *find*$(8, \Pi)$ to the tree shown on the right in Figure 2.5 yields the tree shown in Figure 2.6

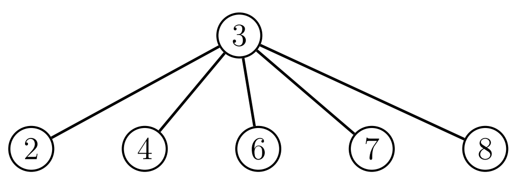

Figure 2.6: The result of applying *find* with path compression

Then, the algorithm is as follows:

function $unif[p, q, \Pi, dd]$: $flag$;
  begin
    $trans := left(dd)$; $ff := right(dd)$; $pq := (p, q)$; $st := (pq)$; $flag := 1$;
    $k := Length(first(trans))$;
    while $st \neq () \wedge flag \neq 0$ **do**
      $uv := top(st)$; $uu := left(uv)$; $vv := right(uv)$;
      $pop(st)$;
      if $bad(ff, uv) = 1$ **then** $flag := 0$
      else
        $u := find(uu, \Pi)$; $v := find(vv, \Pi)$;
        if $u \neq v$ **then**
          $union(u, v, \Pi)$;
          for $i = 1$ **to** k **do**
            $u1 := delta(trans, uu, k - i + 1)$; $v1 := delta(trans, vv, k - i + 1)$;
            $uv := (u1, v1)$; $push(st, uv)$
          endfor
        endif
      endif
    endwhile
  end

The initial partition Π is the identity relation on Q, i.e., it consists of blocks $\{q\}$ for all states $q \in Q$. The algorithm uses a stack st. We are assuming that the DFA dd is specified by a list of two sublists, the first list, denoted $left(dd)$ in the pseudo-code above, being a representation of the transition function, and the second one, denoted $right(dd)$, the set of final states. The transition function itself is a list of lists, where the i-th list represents the i-th row of the transition table for dd. The function $delta$ is such that $delta(trans, i, j)$ returns the j-th state in the i-th row of the transition table of dd. For example, we have the DFA

$$dd = (((2, 3), (2, 4), (2, 3), (2, 5), (2, 3), (7, 6), (7, 8), (7, 9), (7, 6)), (5, 9))$$

consisting of 9 states labeled $1, \ldots, 9$, and two final states 5 and 9 shown in Figure 2.7. Also, the alphabet has two letters, since every row in the transition table consists of two entries. For example, the two transitions from state 3 are given by the pair $(2, 3)$, which indicates that $\delta(3, a) = 2$ and $\delta(3, b) = 3$.

The sequence of steps performed by the algorithm starting with $p = 1$ and $q = 6$ is shown below. At every step, we show the current pair of states, the partition, and the stack.

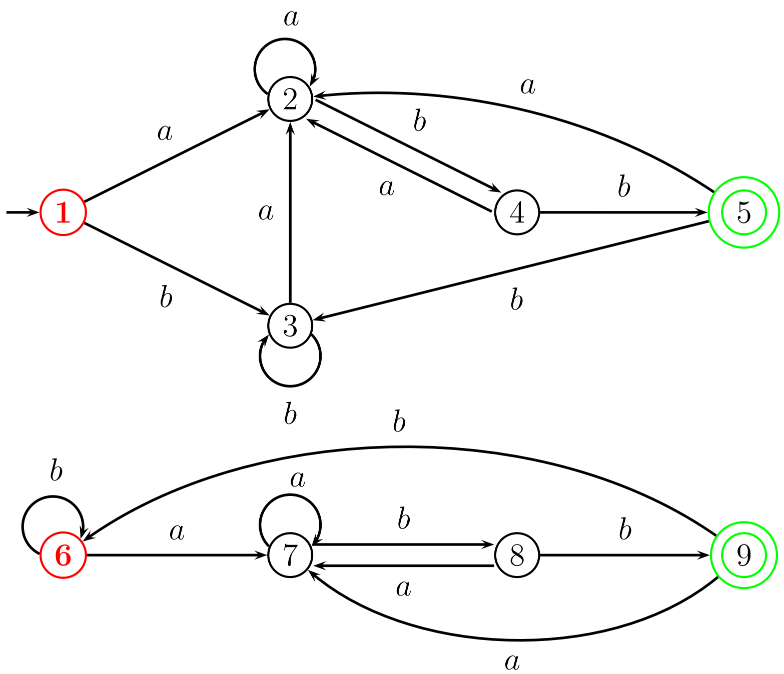

Figure 2.7: Testing state equivalence in a DFA

$p = 1, q = 6,\ \Pi = \{\{1,6\},\{2\},\{3\},\{4\},\{5\},\{7\},\{8\},\{9\}\},\ st = \{\{1,6\}\}$

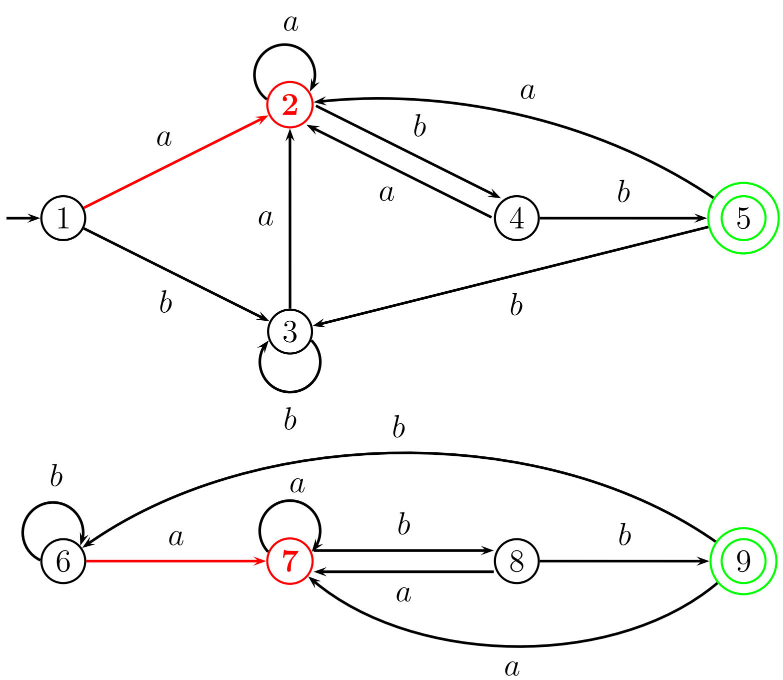

Figure 2.8: Testing state equivalence in a DFA

$p = 2, q = 7,\ \Pi = \{\{1,6\},\{2,7\},\{3\},\{4\},\{5\},\{8\},\{9\}\},\ st = \{\{3,6\},\{2,7\}\}$

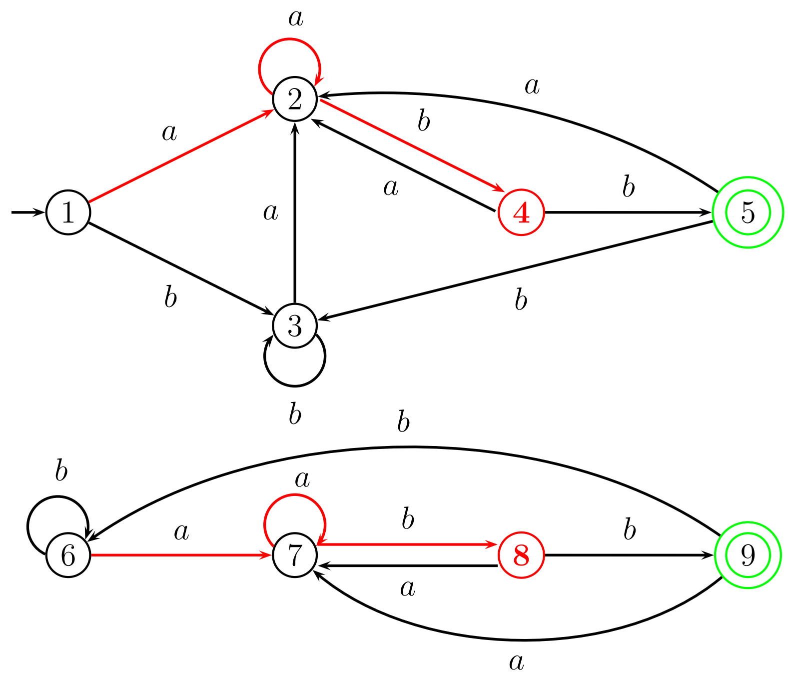

Figure 2.9: Testing state equivalence in a DFA

$p = 4, q = 8,\ \Pi = \{\{1,6\},\{2,7\},\{3\},\{4,8\},\{5\},\{9\}\},\ st = \{\{3,6\},\{4,8\}\}$

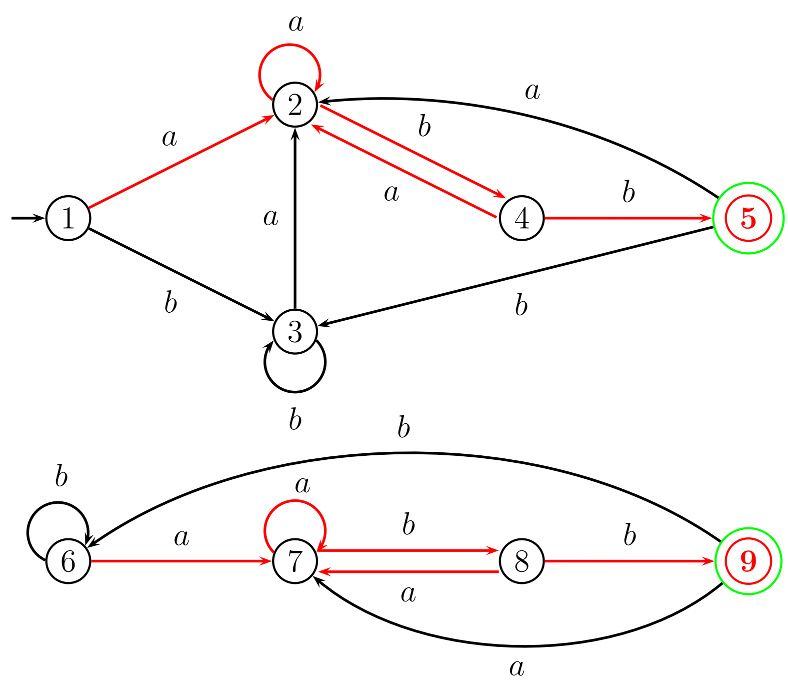

Figure 2.10: Testing state equivalence in a DFA

$p = 5, q = 9,\ \Pi = \{\{1,6\},\{2,7\},\{3\},\{4,8\},\{5,9\}\},\ st = \{\{3,6\},\{5,9\}\}$

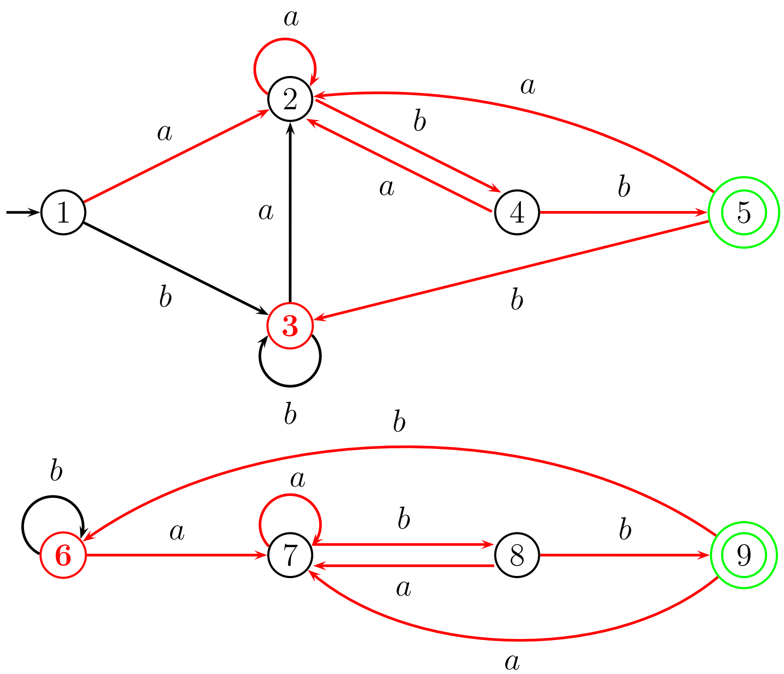

Figure 2.11: Testing state equivalence in a DFA

$p = 3, q = 6,\ \Pi = \{\{1,3,6\},\{2,7\},\{4,8\},\{5,9\}\},\ st = \{\{3,6\},\{3,6\}\}$

Since states 3 and 6 belong to the first block of the partition, the algorithm terminates. Since no block of the partition contains a bad pair, the states $p = 1$ and $q = 6$ are equivalent.

Let us now test whether the states $p = 3$ and $q = 7$ are equivalent.

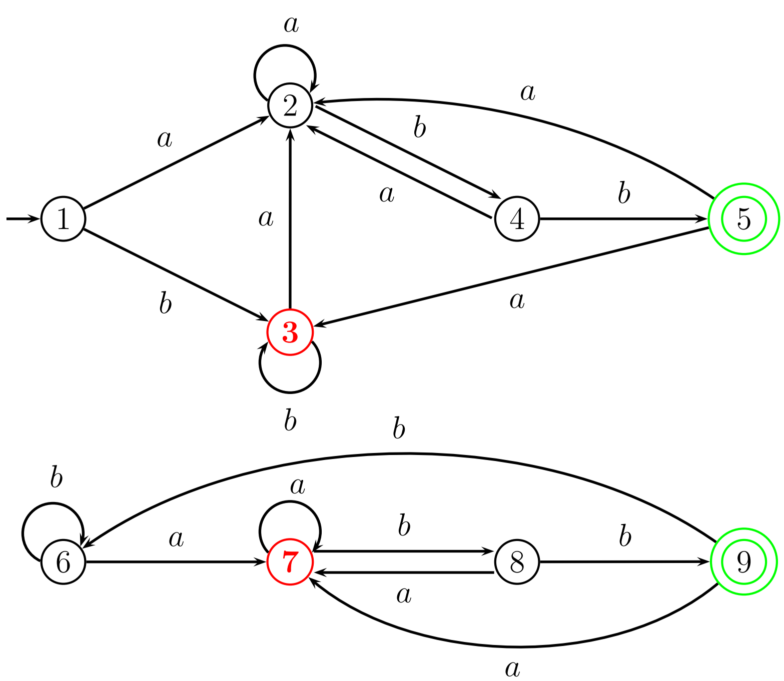

Figure 2.12: Testing state equivalence in a DFA

$p = 3, q = 7,\ \Pi = \{\{1\}, \{2\}, \{3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}, \{9\}\},\ st = \{\{3, 7\}\}$

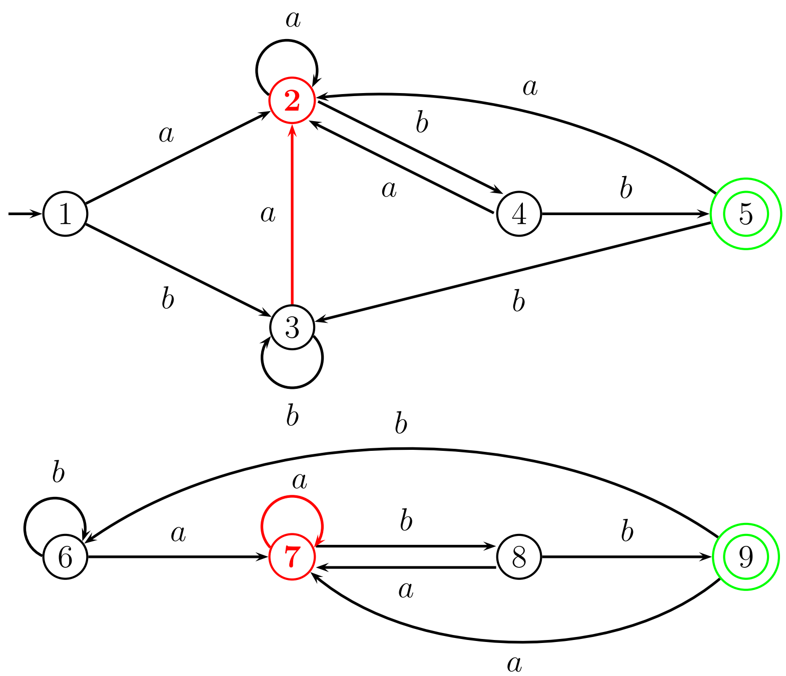

Figure 2.13: Testing state equivalence in a DFA

$p = 2, q = 7,\ \Pi = \{\{1\}, \{2, 3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}, \{9\}\},\ st = \{\{3, 8\}, \{2, 7\}\}$

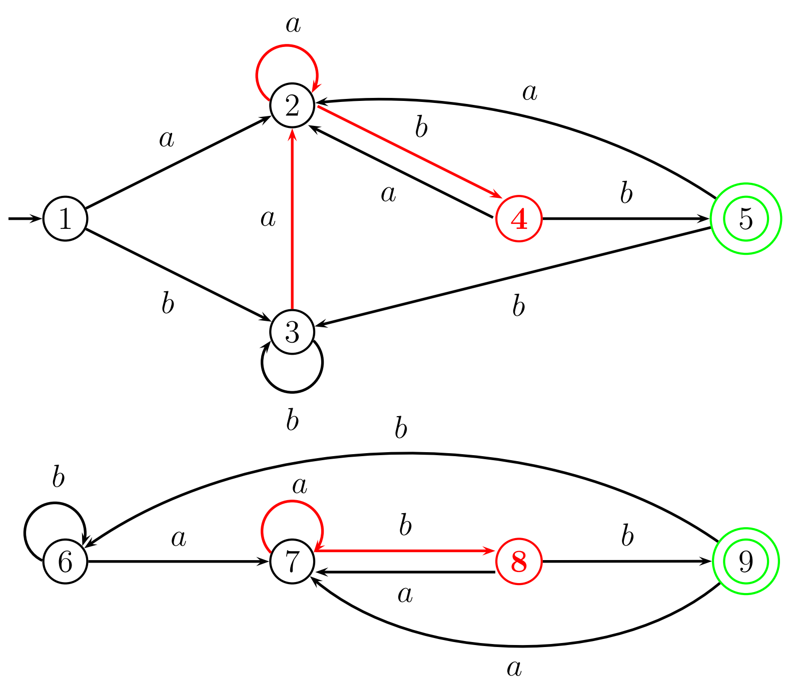

Figure 2.14: Testing state equivalence in a DFA

$p = 4, q = 8,\ \Pi = \{\{1\}, \{2,3,7\}, \{4,8\}, \{5\}, \{6\}, \{9\}\},\ st = \{\{3,8\}, \{4,8\}\}$

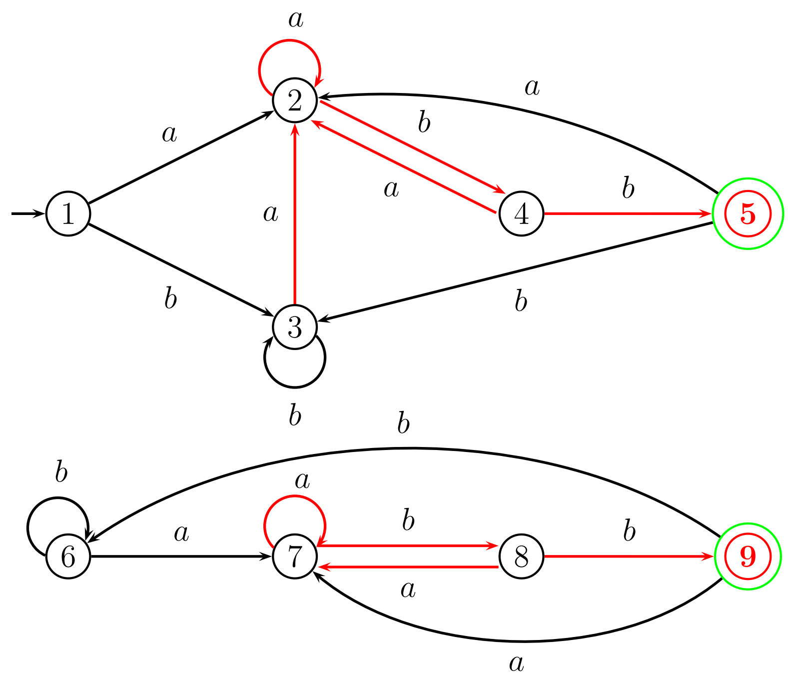

Figure 2.15: Testing state equivalence in a DFA

$p = 5, q = 9,\ \Pi = \{\{1\}, \{2,3,7\}, \{4,8\}, \{5,9\}, \{6\}\},\ st = \{\{3,8\}, \{5,9\}\}$

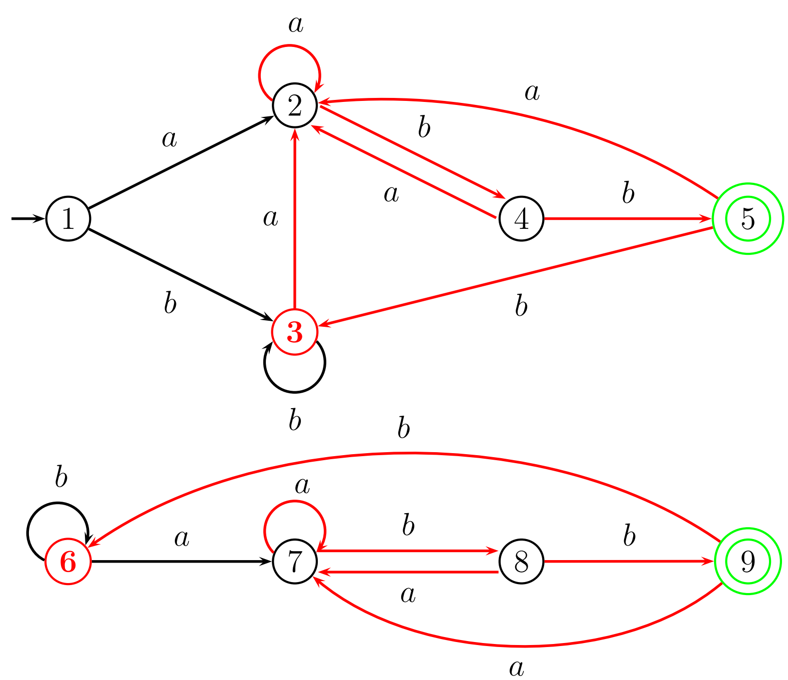

Figure 2.16: Testing state equivalence in a DFA

$p = 3, q = 6,\ \Pi = \{\{1\}, \{2, 3, 6, 7\}, \{4, 8\}, \{5, 9\}\},\ st = \{\{3, 8\}, \{3, 6\}\}$

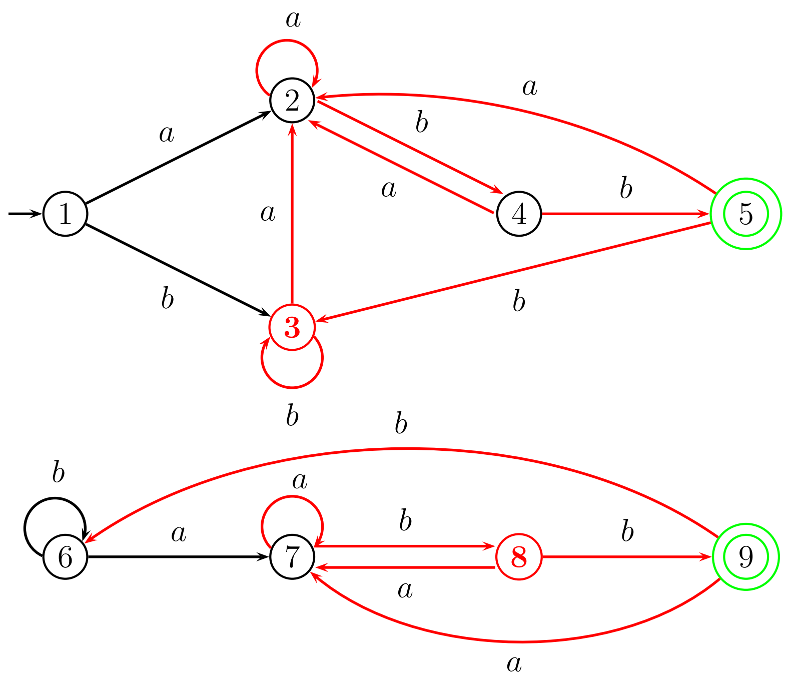

Figure 2.17: Testing state equivalence in a DFA

$p = 3, q = 8,\ \Pi = \{\{1\}, \{2, 3, 4, 6, 7, 8\}, \{5, 9\}\},\ st = \{\{3, 8\}\}$

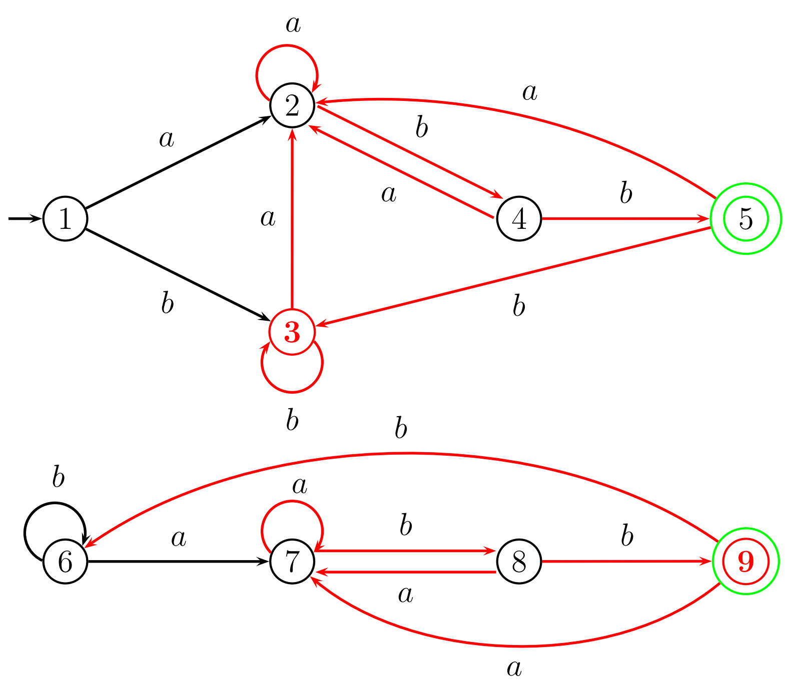

Figure 2.18: Testing state equivalence in a DFA

$p = 3, q = 9,\ \Pi = \{\{1\}, \{2, 3, 4, 6, 7, 8\}, \{5, 9\}\},\ st = \{\{3, 9\}\}$

Since the pair $(3, 9)$ is a bad pair, the algorithm stops, and the states $p = 3$ and $q = 7$ are inequivalent.