Regular Languages

Sheng Yu

Regular languages and finite automata are among the oldest topics in formal language theory. The formal study of regular languages and finite automata can be traced back to the early forties, when finite state machines were used to model neuron nets by McCulloch and Pitts [83]. Since then, regular languages have been extensively studied. Results of early investigations are, for example, Kleene's theorem establishing the equivalence of regular expressions and finite automata [69], the introduction of automata with output by Mealy [86] and Moore [88], the introduction of nondeterministic finite automata by Rabin and Scott [99], and the characterization of regular languages by congruences of finite index by Myhill [90] and Nerode [91].

Regular languages and finite automata have had a wide range of applications. Their most celebrated application has been lexical analysis in programming language compilation and user-interface translations [1, 2]. Other notable applications include circuit design [21], text editing, and pattern matching [70]. Their application in the recent years has been further extended to include parallel processing [3, 37, 50], image generation and compression [9, 28, 29, 33, 116], type theory for object-oriented languages [92], DNA computing [31, 53], etc.

Since the late seventies, many have believed that everything of interest about regular languages is known except for a few very hard problems, which could be exemplified by the six open problems Brzozowski presented at the International Symposium on Formal Language Theory in 1979 [18]. It appeared that not much further work could be done on regular languages. However, contrary to the widespread belief, many new and interesting results on regular languages have kept coming out in the last fifteen years. Besides the fact that three of the six open problems, i.e., the restricted star height problem [52], the regularity of noncounting classes problem [36], and the optimality of prefix codes problem [117], have been solved, there have also been many other interesting new results [65, 82, 102, 111, 120, 124], which include results on measuring or quantifying operations on regular languages. For example, it is shown in [65] that the the "DFA to minimal NFA" problem is PSPACE-complete.

There is a huge amount of established research on regular languages over the span of a half century. One can find a long list of excellent books that include chapters dedicated to regular languages, e.g., [54, 106, 84, 41, 57, 107,

G. Rosenberg et al. (eds.), *Handbook of Formal Languages*
© Springer-Verlag Berlin Heidelberg 1997

123, 98]. Many results, including many recent results, on regular languages are considered to be highly important and very interesting. However, only a few of them can be included in this chapter. In choosing the material for the chapter, besides the very basic results, we tend to select those relatively recent results that are of general interest and have not been included in the standard texts. We choose, for instance, some basic results on alternating finite automata and complexities of operations on regular languages.

This chapter contains the following five sections: 1. Preliminaries, 2. Finite Automata, 3. Regular Expressions, 4. Properties of Regular Languages, and 5. Complexity Issues.

In the first section, we give basic notations and definitions.

In Section 2, we describe three basic types of finite automata: deterministic finite automata, nondeterministic finite automata, and alternating finite automata. We show that the above three models accept exactly the same family of languages. Alternating finite automata are a natural and succinct representation of regular languages. A particularly nice feature of alternating finite automata is that they are backwards deterministic and, thus, can be used practically [50]. We also describe briefly several models of finite automata with output, which include Moore machines, Mealy machines, and finite transducers. Finite transducers are used later in Section 4 for proving various closure properties of regular languages.

In Section 3, we define regular expressions and describe the transformation between regular expressions and finite automata. We present the well-known star height problem and the extended star height problem. At the end of the section, we give a characterization of regular languages having a polynomial density using regular expressions of a special form.

In Section 4, we describe four pumping lemmas for regular languages. The first two give necessary conditions for regularity; and the other two give both sufficient and necessary conditions for regularity. All the four lemmas are stated in a simple and understandable form. We give example languages that satisfy the first two pumping conditions but are nonregular. We show that there are uncountably many such languages. In this section, we also discuss various closure properties of regular languages. We describe the Myhill-Nerode Theorem and discuss minimization of DFA as well as AFA. We also give a lower bound on the number of states of an NFA accepting a given language.

In the final section, we discuss two kinds of complexity issues. The first kind considers the number of states of a minimal DFA for a language resulting from some operation, as a function of the numbers of states for the operand languages. This function is called the state complexity of the operation. We describe the state complexity for several basic operations on regular languages. The state complexity gives a clear and objective measurement for each operation. It also gives a lower bound on the time required for the operation. The second kind of complexity issue that we consider is the time and

space complexity of various problems for finite automata and regular expressions. We list a number of problems, mostly decision problems, together with their time or space complexity to conclude the section as well as the chapter.

1. Preliminaries

An *alphabet* is a finite nonempty set of symbols. A *word* or a *string* over an alphabet Σ is a finite sequence of symbols taken from Σ. The *empty word*, i.e., the word containing zero symbols, is denoted λ. In this chapter, a, b, c, 0, and 1 are used to denote symbols, while u, v, w, x, y, and z are used to denote words.

The *catenation* of two words is the word formed by juxtaposing the two words together, i.e., writing the first word immediately followed by the second word, with no space in between. Let $\Sigma = \{a, b\}$ be an alphabet and $x = aab$ and $y = ab$ be two words over Σ. Then the catenation of x and y, denoted xy, is $aabab$.

Denote by Σ^* the set of all words over the alphabet Σ. Note that Σ^* is a free monoid with catenation being the associative binary operation and λ being the identity element. So, we have

$$\lambda x = x\lambda = x$$

for each $x \in \Sigma^*$. The *length* of a word x, denoted $|x|$, is the number of occurrences of symbols in x.

Let n be a nonnegative integer and x a word over an alphabet Σ. Then x^n is a word over Σ defined by

(i) $x^0 = \lambda$,
(ii) $x^n = xx^{n-1}$, for $n > 0$.

Let $x = a_1 \dots a_n$, $n \geq 0$, be a word over Σ. The reversal of x, denoted x^R, is the word $a_n \dots a_1$. Formally, it is defined inductively by

(i) $x^R = x$, if $x = \lambda$;
(ii) $x^R = y^R a$, if $x = ay$ for $a \in \Sigma$ and $y \in \Sigma^*$.

Let x and y be two words over Σ. We say that x is a *prefix* of y if there exists $z \in \Sigma^*$ such that $xz = y$. Similarly, x is a *suffix* of y if there exists $z \in \Sigma^*$ such that $zx = y$, and x is a *subword* of y if there exist $u, v \in \Sigma^*$ such that $uxv = y$.

A *language* L over Σ is a set of words over Σ. The *empty language* is denoted $\emptyset$. The *universal language* over Σ, which is the language consisting of all words over Σ, is Σ^*. For a language L, we denote by $|L|$ the cardinality of L.

The *catenation* of two languages $L_1, L_2 \subseteq \Sigma^*$, denoted L_1L_2, is the set

$$L_1L_2 = \{w_1w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}.$$

For an integer $n \geq 0$ and a language L, the n^{th} power of L, denoted L^n, is defined by

(i) $L^0 = \{\lambda\}$,
(ii) $L^n = L^{n-1}L$, for $n > 0$.

The star (Kleene closure) of a language L, denoted L^*, is the set

$$\bigcup_{i=0}^{\infty} L^i$$

Similarly, we define

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

Then, the notation Σ^* is consistent with the above definition. The reversal of a language L, denoted L^R, is the set

$$L^R = \{w^R \mid w \in L\}.$$

Note that we often denote a singleton language, i.e., a language containing exactly one word, by the word itself when no confusion will be caused. For example, by xLy, where $x, y \in \Sigma^*$ and $L \subseteq \Sigma^*$, we mean $\{x\}L\{y\}$.

Let Σ and Δ be two finite alphabets. A mapping $h : \Sigma^* \to \Delta^*$ is called a *morphism* if

(1) $h(\lambda) = \lambda$ and
(2) $h(xy) = h(x)h(y)$ for all $x, y \in \Sigma^*$.

Note that condition (1) follows from condition (2), Therefore, condition (1) can be deleted.

For a set S, let 2^S denote the power set of S, i.e., the collection of all subsets of S. A mapping $\varphi : \Sigma^* \to 2^{\Delta^*}$ is called a *substitution* if

(1) $\varphi(\lambda) = \{\lambda\}$ and
(2) $\varphi(xy) = \varphi(x)\varphi(y)$.

Clearly, a morphism is a special kind of substitution where each word is associated with a singleton set. Note that because of the second condition of the definition, morphisms and substitutions are usually defined by specifying only the image of each letter in Σ under the mapping. We extend the definitions of h and φ, respectively, to define

$$h(L) = \{h(w) \mid w \in L\}$$

and

$$\varphi(L) = \bigcup_{w \in L} \varphi(w)$$

for $L \subseteq \Sigma^*$.

Example 1.1. Let $\Sigma = \{a, b, c\}$ and $\Delta = \{0, 1\}$. We define a morphism $h : \Sigma^* \to \Delta^*$ by

$$h(a) = 01, \quad h(b) = 1, \quad h(c) = \lambda.$$

Then, $h(baca) = 10101$. We define a substitution $\varphi : \Sigma^* \to 2^{\Delta^*}$ by

$$\varphi(a) = \{01, 001\}, \quad \varphi(b) = \{1^i \mid i > 0\}, \quad \varphi(c) = \{\lambda\}.$$

Then, $\varphi(baca) = \{1^i0101, 1^i01001, 1^i00101, 1^i001001 \mid i > 0\}$. □

A morphism $h : \Sigma^* \to \Delta^*$ is said to be λ-free if $h(a) \neq \lambda$ for all $a \in \Sigma$. A substitution $\varphi : \Sigma^* \to 2^{\Delta^*}$ is said to be λ-free if $\lambda \notin \varphi(a)$ for all $a \in \Sigma$. And φ is called a *finite substitution* if, for each $a \in \Sigma$, $\varphi(a)$ is a finite subset of Δ^*.

Let $h : \Sigma^* \to \Delta^*$ be a morphism. The inverse of the morphism h is a mapping $h^{-1} : \Delta^* \to 2^{\Sigma^*}$ defined by, for each $y \in \Delta^*$,

$$h^{-1}(y) = \{x \in \Sigma^* \mid h(x) = y\}.$$

Similarly, for a substitution $\varphi : \Sigma^* \to 2^{\Delta^*}$, the inverse of the substitution φ is a mapping $\varphi^{-1} : \Delta^* \to 2^{\Sigma^*}$ defined by, for each $y \in \Delta^*$,

$$\varphi^{-1}(y) = \{x \in \Sigma^* \mid y \in \varphi(x)\}.$$

2. Finite automata

In formal language theory in general, there are two major types of mechanisms for defining languages: acceptors and generators. For regular languages in particular, the acceptors are finite automata and the generators are regular expressions and right (left) linear grammars, etc.

In this section, we describe three types of finite automata (FA): deterministic finite automata (DFA), nondeterministic finite automata (NFA), and alternating finite automata (AFA). We show that all the three types of abstract machines accept exactly the same family of languages. We describe the basic operations of union, intersection, catenation, and complementation on the family of languages implemented using these different mechanisms.

2.1 Deterministic finite automata

A finite automaton consists of a finite set of internal states and a set of rules that govern the change of the current state when reading a given input symbol. If the next state is always uniquely determined by the current state and the current input symbol, we say that the automaton is deterministic.

As an informal explanation, we consider the following example[1]. Let A_0 be an automaton that reads strings of 0's and 1's and recognizes those strings

[1] A similar example is given in [98].

which, as binary numbers, are congruent to 2 (mod 3). We use $\nu_3(x)$ to denote the value, modulo 3, of the binary string x. For example, $\nu(100) = 1$ and $\nu_3(1011) = 2$. Consider an arbitrary input string $w = a_1 \cdots a_n$ to A_0 where each a_i, $1 \leq i \leq n$, is either 0 or 1. It is clear that for each i, $1 \leq i \leq n$, the string $a_1 \cdots a_i$ falls into one of the three cases: (0) $\nu_3(a_1 \cdots a_i) = 0$, (1) $\nu_3(a_1 \cdots a_i) = 1$ and (2) $\nu_3(a_1 \cdots a_i) = 2$. No other cases are possible. So, A_0 needs only three states which correspond to the above three cases (and the initial state corresponds to the case (0)). We name those three states (0), (1), and (2), respectively. The rules that govern the state changes should be defined accordingly. Note that

$$\nu_3(a_1 \cdots a_{i+1}) \equiv 2 * \nu_3(a_1 \cdots a_i) + a_{i+1} \pmod 3.$$

So, if the current state is (1) and the current input symbol is 1, then the next state is (0) since $2 * 1 + 1 \equiv 0 \pmod 3$. The states and their transition rules are shown in Figure 1.

Clearly, each step of state transition is uniquely determined by the current state and the current input symbol. We distinguish state (2) as the final state and define that A_0 accepts an input w if A_0 is in state (2) after reading the last symbol of w. A_0 is an example of a deterministic finite automaton.

Formally, we define a deterministic finite automaton as follows:

A *deterministic finite automaton* (DFA) A is a quintuple $(Q, \Sigma, \delta, s, F)$, where

Q is the finite set of states;
Σ is the input alphabet;
$\delta : Q \times \Sigma \to Q$ is the state transition function;
$s \in Q$ is the starting state; and
$F \subseteq Q$ is the set of final states.

Note that, in general, we do not require the transition function δ to be total, i.e., to be defined for every pair in $Q \times \Sigma$. If δ is total, then we call A a *complete DFA*.

In the above definition, we also do not require that a DFA is connected if we view a DFA as a directed graph where states are nodes and transitions between states are arcs between nodes. A DFA such that every state is reachable from the starting state and reaches a final state is called a *reduced* DFA. A reduced DFA may not be a complete DFA.

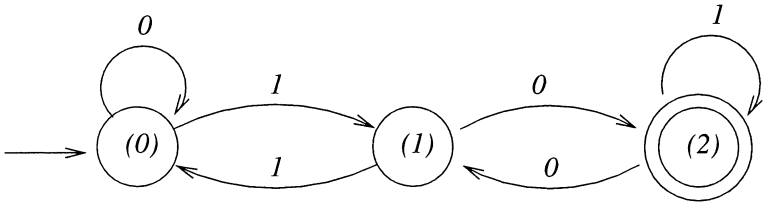


Fig. 1. The states and transition rules of A_0

Example 2.1. A DFA $A_1 = (Q_1, \Sigma_1, \delta_1, s_1, F_1)$ is shown in Figure 2, where $Q_1 = \{0,1,2,3\}$, $\Sigma_1 = \{a,b\}$, $s_1 = 0$, $F_1 = \{3\}$, and δ_1 is defined as follows:

$$\begin{aligned} \delta_1(0,a) &= 1, & \delta_1(0,b) &= 0, \\ \delta_1(1,a) &= 1, & \delta_1(1,b) &= 2, \\ \delta_1(2,a) &= 1, & \delta_1(2,b) &= 3, \\ \delta_1(3,a) &= 3, & \delta_1(3,b) &= 3. \end{aligned}$$

The DFA A_1 is reduced and complete. Note that in a state transition diagram, we always represent final states with double circles and non-final states with single circles. □

A *configuration* of $A = (Q, \Sigma, \delta, s, F)$ is a word in $Q\Sigma^*$, i.e., a state $q \in Q$ followed by a word $x \in \Sigma^*$ where q is the current state of A and x is the remaining part of the input. The *starting configuration* of A for an input word $x \in \Sigma^*$ is sx. *Accepting configurations* are defined to be elements of F (followed by the empty word λ).

A computation step of A is a transition from a configuration α to a configuration β, denoted by $\alpha \vdash_A \beta$, where $\vdash_A$ is a binary relation on the set of configurations of A. The relation $\vdash_A$ is defined by: for $px, qy \in Q\Sigma^*$, $px \vdash_A qy$ if $x = ay$ for some $a \in \Sigma$ and $\delta(p,a) = q$. For example, $0abb \vdash_{A_1} 1bb$ for the DFA A_1. We use $\vdash$ instead of $\vdash_A$ if there is no confusion. The kth power of $\vdash$, denoted $\vdash^k$, is defined by $\alpha \vdash^0 \alpha$ for all configurations $\alpha \in Q\Sigma^*$; and $\alpha \vdash^k \beta$, for $k > 0$ and $\alpha, \beta \in Q\Sigma^*$, if there exists $\gamma \in Q\Sigma^*$ such that $\alpha \vdash^{k-1} \gamma$ and $\gamma \vdash \beta$. The transitive closure and the reflexive and transitive closure of $\vdash$ are denoted $\vdash^+$ and $\vdash^*$, respectively.

A *configuration sequence* of A is a sequence of configurations $C_1, \ldots, C_n$, of A, for some $n \geq 1$, such that $C_i \vdash_A C_{i+1}$ for each i, $1 \leq i \leq n-1$. A configuration sequence is said to be an *accepting configuration sequence* if it starts with a starting configuration and ends with an accepting configuration.

The language accepted by a DFA $A = (Q, \Sigma, \delta, s, F)$, denoted $L(A)$, is defined as follows:

$$L(A) = \{\, w \mid sw \vdash^* f \text{ for some } f \in F \,\}.$$

For convenience, we define the extension of δ, $\delta^* : Q \times \Sigma^* \to Q$, inductively as follows. We set $\delta^*(q, \lambda) = q$ and $\delta^*(q, xa) = \delta(\delta^*(q,x), a)$, for $q \in Q$, $a \in \Sigma$, and $x \in \Sigma^*$. Then, we can also write

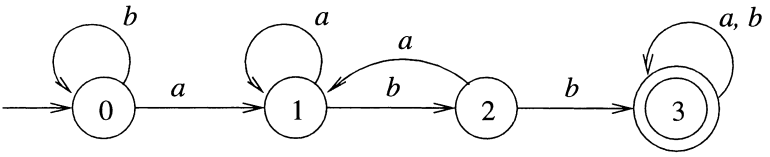


Fig. 2. A deterministic finite automaton A_1

$$L(A) = \{ w \mid \delta^*(s, w) = f \text{ for some } f \in F \}.$$

The collection of all languages accepted by DFA is denoted $\mathcal{L}_{DFA}$. We call it the *family of DFA languages.* We will show later that the families of languages accepted by deterministic, nondeterministic, and alternating finite automata are the same. This family is again the same as the family of languages denoted by regular expressions. It is called the family of regular languages.

In the remaining of this subsection, we state several basic properties of DFA languages. More properties of DFA languages can be found in Section 4..

Lemma 2.1. *For each $L \in \mathcal{L}_{DFA}$, there is a complete DFA that accepts L.*

Proof. Let $L \in \mathcal{L}_{DFA}$. Then there is a DFA $A = (Q, \Sigma, \delta, s, F)$ such that $L = L(A)$. If A is complete, then we are done. Otherwise, we construct a DFA A' which is the same as A except that there is one more state d and all transitions undefined in A go to d in A'. More precisely, we define $A' = (Q', \Sigma, \delta', s, F)$ such that $Q' = Q \cup \{d\}$, where $d \notin Q$, and $\delta' : Q' \times \Sigma \to Q'$ is defined by

$$\delta'(p, a) = \begin{cases} \delta(p, a), & \text{if } \delta(p, a) \text{ is defined;} \\ d, & \text{otherwise} \end{cases}$$

for $p \in Q'$ and $a \in \Sigma$. It is clear that the new state d and the new state transitions do not change the acceptance of a word. Therefore, $L(A) = L(A')$. □

Theorem 2.1. *The family of DFA languages, $\mathcal{L}_{DFA}$, is closed under union and intersection.*

Proof. Let $L_1, L_2 \subseteq \Sigma^*$ be two arbitrary DFA languages such that $L_1 = L(A_1)$ and $L_2 = L(A_2)$ for some complete DFA $A_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $A_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$.

First, we show that there exists a DFA A such that $L(A) = L_1 \cup L_2$. We construct $A = (Q, \Sigma, \delta, s, F)$ as follows:

$Q = Q_1 \times Q_2$,

$s = (s_1, s_2)$,

$F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$, and

$\delta : Q_1 \times Q_2 \to Q_1 \times Q_2$ is defined by $\delta((p_1, p_2), a) = (\delta_1(p_1, a), \delta_2(p_2, a))$.

The intuitive idea of the construction is that, for each input word, A runs A_1 and A_2 in parallel, starting from both the starting states. Having finished reading the input word, A accepts the word if either A_1 or A_2 accepts it. Therefore, $L(A) = L(A_1) \cup L(A_2)$.

For intersection, the construction is the same except that $F = F_1 \times F_2$. □

Note that, in the above proof, the condition that A_1 and A_2 are complete is not necessary in the case of intersection. However, if either A_1 or A_2 is incomplete, the resulting automaton is incomplete.

Theorem 2.2. *$\mathcal{L}_{DFA}$ is closed under complementation.*

Proof. Let $L \in \mathcal{L}_{DFA}$. By Lemma 2.1, there is a complete DFA $A = (Q, \Sigma, \delta, s, F)$ such that $L = L(A)$. Then, clearly, the complement of L, denoted $\overline{L}$, is accepted by $\overline{A} = (Q, \Sigma, \delta, s, Q - F)$. □

2.2 Nondeterministic finite automata

Nondeterministic finite automata (NFA) are a generalization of DFA where, for a given state and an input symbol, the number of possible transitions can be greater than one. An NFA is shown in Figure 3, where there are two possible transitions for state 0 and input symbol a: to state 0 or to state 1.

Formally, a *nondeterministic finite automaton* A is a quintuple $(Q, \Sigma, \delta, s, F)$ where Q, Σ, s, and F are defined exactly the same way as for a DFA, and $\delta : Q \times \Sigma \to 2^Q$ is the transition function, where 2^Q denotes the power set of Q.

For example, the transition function for the NFA A_2 of Figure 3 is the following:

$$\begin{aligned} \delta(0,a) &= \{0,1\}, & \delta(0,b) &= \{0\}, \\ \delta(1,a) &= \emptyset, & \delta(1,b) &= \{2\}, \\ \delta(2,a) &= \emptyset, & \delta(2,b) &= \{3\}, \\ \delta(3,a) &= \{3\}, & \delta(3,b) &= \{3\}. \end{aligned}$$

A DFA can be considered an NFA, where each value of the transition function is either a singleton or the empty set.

The computation relation $\vdash_A$: $Q\Sigma^* \times Q\Sigma^*$ of an NFA A is defined by setting $px \vdash_A qy$ if $x = ay$ and $q \in \delta(p, a)$ for $p, q \in Q$, $x, y \in \Sigma^*$, and $a \in \Sigma$. Then the language accepted by A is

$$L(A) = \{\, w \mid sw \vdash_A^* f, \text{ for some } f \in F \,\}.$$

Two automata are said to be *equivalent* if they accept exactly the same language. The NFA A_2 which is shown in Figure 3 accepts exactly the same language as A_1 of Figure 2. Thus, A_2 is equivalent to A_1.

Denote by $\mathcal{L}_{NFA}$ the family of languages accepted by NFA. We show that $\mathcal{L}_{DFA} = \mathcal{L}_{NFA}$.

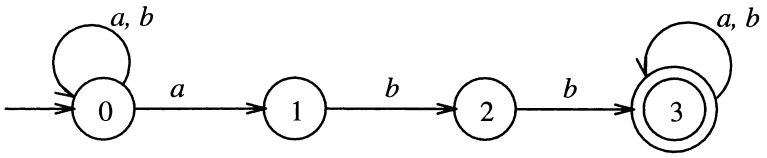


Fig. 3. A nondeterministic finite automaton A_2

Lemma 2.2. *For each NFA A of n states, there exists a complete DFA A' of at most 2^n states such that $L(A') = L(A)$.*

Proof. Let $A = (Q, \Sigma, \delta, s, F)$ be an NFA such that $|Q| = n$. We construct a DFA A' such that each state of A' is a subset of Q and the transition function is defined accordingly. More precisely, we define $A' = (Q', \Sigma, \delta', s', F')$ where $Q' = 2^Q$; $\delta' : Q' \times \Sigma \to Q'$ is defined by, for $P_1, P_2 \in Q'$ and $a \in \Sigma$, $\delta'(P_1, a) = P_2$ if

$$P_2 = \{q \in Q \mid \text{there exists } p \in P_1 \text{ such that } q \in \delta(p, a)\};$$

$s' = \{s\}$; and $F' = \{P \in Q' \mid P \cap F \neq \emptyset\}$. Note that A' has 2^n states.

In order to show that $L(A) = L(A')$, we first prove the following claim.

Claim. For an arbitrary word $x \in \Sigma^*$, $sx \vdash_A^t p$, for some $p \in Q$, if and only if $s'x \vdash_{A'}^t P$ for some $P \in Q'$ (i.e., $P \subseteq Q$) such that $p \in P$.

We prove the claim by induction on t, the number of transitions. If $t = 0$, then the statement is trivially true since $s' = \{s\}$. We hypothesize that the statement is true for $t - 1$, $t > 0$. Now consider the case of t, $t > 0$. Let $x = x_0 a$, $x_0 \in \Sigma^*$ and $a \in \Sigma$, and $sx_0 a \vdash_A^{t-1} qa \vdash_A p$ for some $p, q \in Q$. Then, by the induction hypothesis, $s'x_0 \vdash_{A'}^{t-1} P'$ for some $P' \in Q'$ such that $q \in P'$. Since $p \in \delta(q, a)$, we have $\delta'(P', a) = P$ for some $P \in Q'$ such that $p \in P$ by the definition of δ'. So, we have $s'x \vdash_{A'}^{t-1} P'a \vdash_{A'} P$ and $p \in P$. Conversely, let $s'x_0 a \vdash_{A'}^{t-1} P'a \vdash_{A'} P$ and $p \in P$. Then $\delta'(P', a) = P$ and, therefore, there exists $p' \in P'$ such that $p \in \delta(p', a)$ by the definition of δ'. By the induction hypothesis, we have $sx_0 \vdash_A^{t-1} p'$. Thus, $sx \vdash_A^{t-1} p'a \vdash_A p$. This completes the proof of the claim.

Due to the above claim, we have $sw \vdash_A^* f$, for some $f \in F$, if and only if $s'w \vdash_{A'}^* P$, for some $P \in Q'$, such that $P \cap F \neq \emptyset$, i.e., $P \in F'$. Therefore, $L(A) = L(A')$. □

The method used above is called the *subset construction*. In the worst case, all the subsets of Q are necessary. Then the resulting DFA would consist of 2^n states if n is the number of states of the corresponding NFA. Note that if the resulting DFA is not required to be a complete DFA, the empty subset of Q is not needed. So, the resulting DFA consists of $2^n - 1$ states in the worst case. In Section 5., we will show that such cases exist. However, in most cases, not all the subsets are necessary. Thus, it suffices to construct only those subsets that are reachable from $\{s\}$. As an example, we construct a DFA A_6 which is equivalent to NFA A_2 of Figure 3 as follows:

State	a	b
$\{0\}$	$\{0,1\}$	$\{0\}$
$\{0,1\}$	$\{0,1\}$	$\{0,2\}$
$\{0,2\}$	$\{0,1\}$	$\{0,3\}$
$\{0,3\}$	$\{0,1,3\}$	$\{0,3\}$
$\{0,1,3\}$	$\{0,1,3\}$	$\{0,2,3\}$
$\{0,2,3\}$	$\{0,1,3\}$	$\{0,3\}$

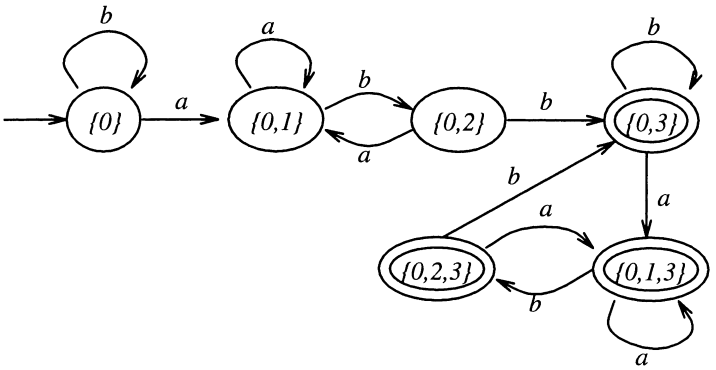


Fig. 4. A DFA A_6 equivalent to NFA A_2

The state transition diagram for A_6 is shown in Figure 4. Only six out of the total sixteen subsets are used in the above example. The other ten subsets of $\{0,1,2,3\}$ are not reachable from $\{0\}$ and, therefore, useless. Note that the resulting DFA can be further minimized into one of only four states. Minimization of DFA is one of the topics in Section 4.

NFA can be further generalized to have state transitions without reading any input symbol. Such transitions are called λ-transitions in the following definition.

A *nondeterministic finite automaton with λ-transitions* (λ-*NFA*) A is a quintuple $(Q, \Sigma, \delta, s, F)$ where Q, Σ, s, and F are the same as for an NFA; and $\delta : Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ is the transition function.

Figure 5 shows the transition diagram of a λ-NFA, where the transition function δ can also be written as follows:

$$\begin{array}{ll} \delta(0,a) = \{0\}, & \delta(0,\lambda) = \{1\}, \\ \delta(1,b) = \{1\}, & \delta(1,\lambda) = \{2\}, \\ \delta(2,c) = \{2\}. & \end{array}$$

and $\delta(q, X) = \emptyset$ in all other cases.

For a λ-NFA $A = (Q, \Sigma, \delta, s, F)$, the binary relation $\vdash_A$: $Q\Sigma^* \times Q\Sigma^*$ is defined by that $px \vdash_A qy$, for $p, q \in Q$ and $x, y \in \Sigma^*$, if $x = ay$ and $q \in \delta(p, a)$ or if $x = y$ and $q \in \delta(p, \lambda)$. The language accepted by A is again defined as

$$L(A) = \{w \mid sw \vdash_A^* f, \text{ for some } f \in F\}.$$

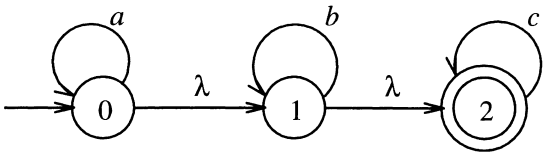


Fig. 5. A λ-NFA A_3

For example, the language accepted by A_3 of Figure 5 is

$$L(A_3) = \{a^i b^j c^k \mid i, j, k \geq 0\}.$$

We will show that for each λ-NFA, there exists an NFA that accepts exactly the same language. First, we give the following definition.

Let $A = (Q, \Sigma, \delta, s, F)$ be a λ-NFA. The *λ-closure* of a state $q \in Q$, denoted λ-closure(q), is the set of all states that are reachable from q by zero or more λ-transitions, i.e.,

$$\lambda\text{-closure}(q) = \{p \in Q \mid q \vdash_A^* p\}.$$

Theorem 2.3. *For each λ-NFA A, there exists an NFA A' such that $L(A) = L(A')$.*

Proof. Let $A = (Q, \Sigma, \delta, s, F)$ be a λ-NFA. We construct an NFA $A' = (Q, \Sigma, \delta', s, F')$ where for each $q \in Q$ and $a \in \Sigma$,

$$\delta'(p, a) = \delta(p, a) \cup \bigcup_{q \in \lambda\text{-closure}(p)} \delta(q, a),$$

and

$$F' = \{q \mid \lambda\text{-closure}(q) \cap F \neq \emptyset\}.$$

The reader can verify that $L(A) = L(A')$. □

Consider λ-NFA A_3 which is shown in Figure 5. We have λ-closure(0) = $\{0, 1, 2\}$, λ-closure(1) = $\{1, 2\}$, and λ-closure(2) = $\{2\}$. An equivalent NFA is shown in Figure 6, which is obtained by following the construction specified in the above proof.

Let $M_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ be two λ-NFA and assume that $Q_1 \cap Q_2 = \emptyset$. Then it is straightforward to construct λ-NFA $M_1{+}M_2$, M_1M_2, and M_1^* such that $L(M_1{+}M_2) = L(M_1) \cup L(M_2)$, $L(M_1M_2) = L(M_1)L(M_2)$, and $L(M_1^*) = (L(M_1))^*$, respectively. The constructions are illustrated by the diagrams in Figure 7. Formal definitions of the λ-NFA are listed below:

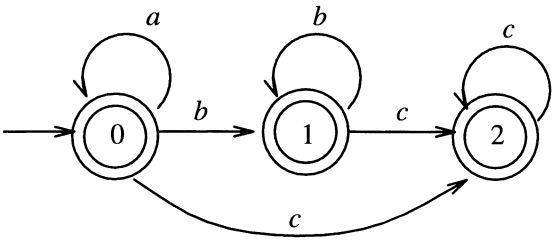
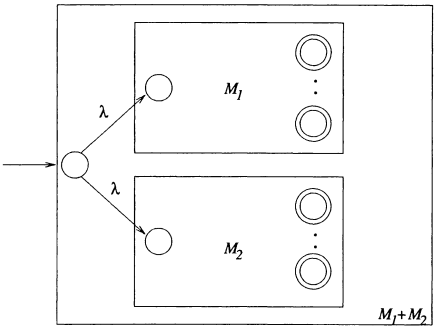
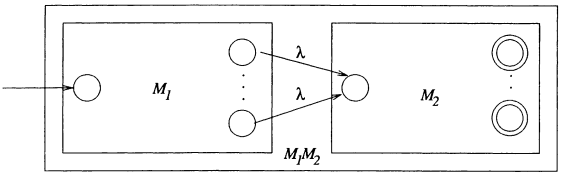


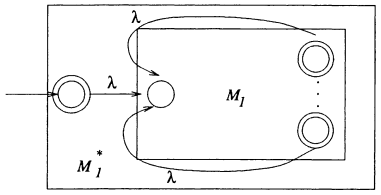
Fig. 6. An NFA A_3'



(a) Union



(b) Catenation



(c) Star

Fig. 7. Union, Catenation, and Star operations on λ-NFA

- **Union** $M_1+M_2 = (Q, \Sigma, \delta, s, F)$ such that $L(M_1+M_2) = L(M_1) \cup L(M_2)$, where $Q = Q_1 \cup Q_2 \cup \{s\}$, $s \notin Q_1 \cup Q_2$, $F = F_1 \cup F_2$, and
  $\delta(s, \lambda) = \{s_1, s_2\}$,
  $\delta(q, a) = \delta_1(q, a)$ if $q \in Q_1$ and $a \in \Sigma \cup \{\lambda\}$,
  $\delta(q, a) = \delta_2(q, a)$ if $q \in Q_2$ and $a \in \Sigma \cup \{\lambda\}$.
- **Catenation** $M_1M_2 = (Q, \Sigma, \delta, s, F)$ such that $L(M_1M_2) = L(M_1)L(M_2)$, where $Q = Q_1 \cup Q_2$, $s = s_1$, $F = F_2$, and
  $\delta(q, a) = \delta_1(q, a)$ if $q \in Q_1$ and $a \in \Sigma$ or $q \in Q_1 - F_1$ and $a = \lambda$,
  $\delta(q, \lambda) = \delta_1(q, \lambda) \cup \{s_2\}$ if $q \in F_1$,
  $\delta(q, a) = \delta_2(q, a)$ if $q \in Q_2$ and $a \in \Sigma \cup \{\lambda\}$.
- **Star** $M_1^* = (Q, \Sigma, \delta, s, F)$ such that $L(M_1^*) = (L(M_1))^*$, where $Q = Q_1 \cup \{s\}$, $s \notin Q_1$, $F = F_1 \cup \{s\}$, and
  $\delta(s, \lambda) = \{s_1\}$,
  $\delta(q, \lambda) = \delta_1(q, \lambda) \cup \{s_1\}$ if $q \in F_1$,
  $\delta(q, a) = \delta_1(q, a)$ if $q \in Q_1$ and $a \in \Sigma$ or $q \in Q_1 - F_1$ and $a = \lambda$.

Intersection and complementation are more convenient to do using the DFA representation.

Another form of generalization of NFA is defined in the following.

A *NFA with nondeterministic starting state* (NNFA) $A = (Q, \Sigma, \delta, S, F)$ is an NFA except that there is a set of starting states S rather than exactly one starting state. Thus, for an input word, the computation of A starts from a nondeterministically chosen starting state.

Clearly, for each NNFA A, we can construct an equivalent λ-NFA A' by adding to A a new state s and a λ-transition from s to each of the starting states in S, and defining s to be the starting state of A'. Thus, NNFA accept exactly the same family of languages as NFA (or DFA or λ-NFA). Each NNFA can also be transformed directly to an equivalent DFA using a subset construction, which is similar to the one for transforming an NFA to a DFA except that the starting state of the resulting DFA is the set of all the starting states of the NNFA. So, we have the following:

Theorem 2.4. *For each NNFA A of n states, we can construct an equivalent DFA A' of at most 2^n states.* □

Each NNFA has a matrix representation defined as follows [107]: Let $A = (Q, \Sigma, \delta, S, F)$ be an NNFA and assume that $Q = \{q_1, q_2, \ldots, q_n\}$. A mapping h of Σ into the set of $n \times n$ Boolean matrices is defined by setting the (i, j)th entry in the matrix $h(a)$, $a \in \Sigma$, to be 1 if $q_j \in \delta(q_i, a)$, i.e., there is an a-transition from q_i to q_j. We extend the domain of h from Σ to Σ^* by

$$h(w) = \begin{cases} I & \text{if } w = \lambda, \\ h(w_0)h(a) & \text{if } w = w_0 a, \end{cases}$$

where I is the $n \times n$ identity matrix and the juxtaposition of two matrices denotes the multiplication of the two Boolean matrices, where $\wedge$ and $\vee$ are the basic operations. A row vector π of n entries is defined by setting the ith entry to 1 if $q_i \in S$. A column vector ξ of n entries is defined by setting the ith entry to 1 if $q_i \in F$. The following theorem has been proved in [107].

Theorem 2.5. *Let $w \in \Sigma^*$. Then $w \in L(A)$ if and only if $\pi h(w) \xi = 1$.* □

2.3 Alternating finite automata

The notion of alternation is a natural generalization of nondeterminism. It received its first formal treatment by Chandra, Kozen, and Stockmeyer in 1976 [22, 23, 71]. Various types of alternating Turing machines (ATM) and alternating pushdown machines and their relationship to complexity classes have been studied [24, 37, 61, 62, 79, 72, 94, 103, 38, 55]. Such machines are useful for a better understanding of many questions in complexity theory. For alternating finite automata (AFA – not to be confused with *abstract families of acceptors* defined in [47]), it is proved in [23] that they are precisely as

powerful as deterministic finite automata as far as language recognition is concerned. It is also shown in [23] that there exist k-state AFA such that any equivalent complete DFA has at least 2^{2^k} states. A more detailed treatment of alternating finite automata and their operations can be found in [45].

The study of *Boolean automata* was initiated by Brzozowski and Leiss [19] at almost the same time period as AFA were introduced. Boolean automata are essentially AFA except that they allow multiple initial states instead of exactly one initial state in the case of an AFA. In that seminal paper, they also introduced a new type of system of language equations, which can be used to give a clear and comprehensible representation of a Boolean automaton. Boolean automata and the systems of language equations have been further studied in [73, 75, 76, 77].

In the following, we will describe results obtained from both of the above mentioned sources. However, we will use only the term *alternating finite automaton* (AFA). Our basic definitions of AFA follow those in [23]. The equational representation is from [19, 44], and the operations of AFA are from [45].

2.3.1 AFA – the definition

AFA are a natural extension of NFA. In an NFA, if there are two or more possible transitions for the current state and the current input symbol, the outcomes of all the possible computations for the remaining input word are logically **OR**ed. Consider the NFA A_4 shown in Figure 8 with the input *abbb*. When starting at state 0 and reading a, the automaton has two possible moves: to state 1 or to state 2. If we denote by a Boolean variable x_0 whether there is a successful computation for *abbb* from state 0, and by x_1 and x_2 whether there is a successful computation for the remaining of the input *bbb* from state 1 and state 2, respectively, then the relation of the computations can be described by the equation

$$x_0 = x_1 \vee x_2.$$

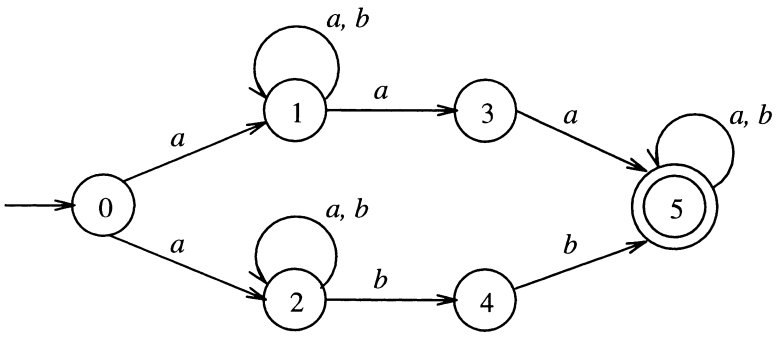


Fig. 8. An NFA A_4

This relation, represented by the equation, captures the essence of nondeterminism. The definition of AFA extends this idea and allows arbitrary Boolean operations in place of the "$\vee$" operation. For example, we may specify that

$$x_0 = (\neg x_1) \wedge x_2.$$

It means that there is a successful computation for *abbb* from state 0 if and only if there is no successful computation for *bbb* from state 1 and there is a successful computation for *bbb* from state 2.

More specifically, an AFA works in the following way: When the automaton reads an input symbol a in a given state q, it will activate all states of the automaton to work on the remaining part of the input in parallel. Once the states have completed their tasks, q will compute its value by applying a Boolean function on those results and pass on the resulting value to the state by which it was activated. A word w is accepted if the starting state computes the value of 1. It is rejected otherwise. We now formalize this idea.

Denote by the symbol B the two-element Boolean algebra $B = (\{0,1\}, \vee, \wedge, \neg, 0, 1)$. Let Q be a set. Then B^Q is the set of all mappings of Q into B. Note that $u \in B^Q$ can be considered as a vector of $|Q|$ entries, indexed by elements of Q, with each entry being from B. For $u \in B^Q$ and $q \in Q$, we write u_q to denote the image of q under u. If P is a subset of Q then $u|_P$ is the restriction of u to P.

An *alternating finite automaton* (AFA) is a quintuple $A = (Q, \Sigma, s, F, g)$ where

Q is the finite set of *states;*
Σ is the *input alphabet;*
$s \in Q$ is the *starting state;*
$F \subseteq Q$ is the set of *final states;*
g is a function of Q into the set of all functions of $\Sigma \times B^Q$ into B.

Note that for each state $q \in Q$, $g(q)$ is a function from $\Sigma \times B^Q$ into B, which we will often denote by g_q in the sequel. For each state $q \in Q$ and $a \in \Sigma$, we define $g_q(a)$ to be the Boolean function $B^Q \to B$ such that

$$g_q(a)(u) = g_q(a, u), \quad u \in B^Q.$$

Thus, for $u \in B^Q$, the value of $g_q(a)(u)$, also $g_q(a, u)$, is either 1 or 0.

We define the function $g_Q : \Sigma \times B^Q \to B^Q$ by putting together the $|Q|$ functions $g_q : \Sigma \times B^Q \to B$, $q \in Q$, as follows. For $a \in \Sigma$ and $u, v \in B^Q$, $g_Q(a, u) = v$ if and only if $g_q(a, u) = v_q$ for each $q \in Q$. For convenience, we will write $g(a, u)$ instead of $g_Q(a, u)$ in the following.

Example 2.2. We define an AFA $A_5 = (Q, \Sigma, s, F, g)$ where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $s = q_0$, $F = \{q_2\}$, and g is given by

State	a	b
q_0	$q_1 \wedge q_2$	0
q_1	q_2	$q_1 \wedge q_2$
q_2	$\overline{q_1} \wedge q_2$	$q_1 \vee \overline{q_2}$

Note that we use $\overline{q}$ instead of $\neg q$ for convenience. □

We define $f \in B^Q$ by the condition

$$f_q = 1 \iff q \in F,$$

and we call f the *characteristic vector* of F. The characteristic vector for F of A_5 is $f = (f_{q_0}, f_{q_1}, f_{q_2}) = (0, 0, 1)$.

We extend g to a function of Q into the set of all functions $\Sigma^* \times B^Q \to B$ as follows:

$$g_q(w, u) = \begin{cases} u_q, & \text{if } w = \lambda, \\ g_q(a, g(w', u)), & \text{if } w = aw' \text{ with } a \in \Sigma \text{ and } w' \in \Sigma^*, \end{cases}$$

where $w \in \Sigma^*$ and $u \in B^Q$.

Now we define the acceptance of a word and the acceptance of a language by an AFA.

Let $A = (Q, \Sigma, s, F, g)$ be an AFA. A word $w \in \Sigma^*$ is *accepted* by A if and only if $g_s(w, f) = 1$, where f is the characteristic vector of F. The *language accepted* by A is the set

$$L(A) = \{w \in \Sigma^* \mid g_s(w, f) = 1\}.$$

Let $w = aba$. Then w is accepted by A_5 of Example 2.2 as follows:

$$\begin{aligned} & g_{q_0}(aba, f) \\ = \; & g_{q_1}(ba, f) \wedge g_{q_2}(ba, f) \\ = \; & (g_{q_1}(a, f) \wedge g_{q_2}(a, f)) \wedge (g_{q_1}(a, f) \vee \overline{g_{q_2}(a, f)}) \\ = \; & (g_{q_2}(\lambda, f) \wedge (\overline{g_{q_1}(\lambda, f)} \wedge g_{q_2}(\lambda, f))) \wedge (g_{q_2}(\lambda, f) \vee \\ & \overline{\overline{g_{q_1}(\lambda, f)} \wedge g_{q_2}(\lambda, f)}) \\ = \; & (f_{q_2} \wedge (\overline{f_{q_1}} \wedge f_{q_2})) \wedge (f_{q_2} \vee \overline{\overline{f_{q_1}} \wedge f_{q_2}}) \\ = \; & (1 \wedge (\overline{0} \wedge 1)) \wedge (1 \vee \overline{\overline{0} \wedge 1}) \\ = \; & 1 \end{aligned}$$

If we denote each $u \in B^Q$ by a vector $(u_{q_0}, u_{q_1}, u_{q_2})$ and write $f = (0, 0, 1)$, then we can rewrite the above:

$$\begin{aligned} & g_{q_0}(aba, f) \\ = \; & g_{q_0}(a, g(ba, f)) \\ = \; & g_{q_0}(a, g(b, g(a, f))) \\ = \; & g_{q_0}(a, g(b, g(a, (0, 0, 1)))) \\ = \; & g_{q_0}(a, g(b, (0, 1, 1))) \\ = \; & g_{q_0}(a, (0, 1, 1)) \\ = \; & 1 \end{aligned}$$

2.3.2 Systems of equations – representations of AFA

Consider again the example of AFA A_5. We may use the following system of equations instead of a table to represent the transitions of A_5:

$$\begin{cases} X_0 &= a \cdot (X_1 \wedge X_2) + b \cdot 0 \\ X_1 &= a \cdot X_2 + b \cdot (X_1 \wedge X_2) \\ X_2 &= a \cdot (\overline{X_1} \wedge X_2) + b \cdot (X_1 \vee \overline{X_2}) + \lambda \end{cases}$$

where a variable X_i represents state q_i, $0 \leq i \leq 2$, respectively; and λ appearing in the third equation specifies that q_2 is a final state.

In general, an AFA $A = (Q, \Sigma, s, F, g)$ can be represented by

$$X_q = \sum_{a \in \Sigma} a \cdot g_q(a, X) + \varepsilon_q, \quad q \in Q \tag{1}$$

where X is the vector of variables X_q, $q \in Q$, and

$$\varepsilon_q = \begin{cases} \lambda & \text{if } q \in F, \\ 0 & \text{otherwise,} \end{cases}$$

for each $q \in Q$. Note that all the terms of the form $a \cdot 0$ or 0, $a \in \Sigma$, can be omitted.

For each AFA A, we call such a system of equations the *equational representation* of A. At this moment, we consider the system of equations solely as an alternative form to present the definition of an AFA.

NFA are a special case of AFA. The NFA A_2 of Figure 3 can be represented by

$$\begin{cases} X_0 &= a \cdot (X_0 \vee X_1) + b \cdot X_0 \\ X_1 &= b \cdot X_2 \\ X_2 &= b \cdot X_3 \\ X_3 &= a \cdot X_3 + b \cdot X_3 + \lambda \end{cases}$$

Let Σ be an alphabet. We define the *L-interpretation* as follows:

Notation	Interpretation
0	$\emptyset$
1	Σ^*
$\wedge$	$\cap$
$\vee$	$\cup$
$\neg$	complement
a, $a \in \Sigma$	$\{a\}$
λ	$\{\lambda\}$
$\cdot$	set catenation
$+$	$\cup$
$=$	language equivalence

Under this interpretation, the systems of equations defined above become systems of language equations. Systems of language equations of a different form were studied by Salomaa in [106], where the operations are restricted to catenation, union, and star. The systems of language equations we are considering can be viewed as an extension of the systems of language equations of Salomaa.

Formally, a *system of language equations* over an alphabet Σ is a system of equations of the following form under the L-interpretation:

$$X_i = \sum_{a \in \Sigma} a \cdot f_i^{(a)}(X) + \varepsilon_i, \quad i = 0, \ldots, n \tag{2}$$

for some $n \geq 0$, where $X = (X_0, \ldots, X_n)$; for each $a \in \Sigma$ and $i \in \{0, \ldots, n\}$, $f_i^{(a)}(X)$ is a Boolean function; and $\varepsilon_i = \lambda$ or 0.

The following result has been proved in [19].

Theorem 2.6. *Any system of language equations of the form (2.3.2) has a unique solution for each X_i, $i = 0, \ldots, n$. Furthermore, the solution for each X_i is regular.* □

The following results can be found in [44].

Theorem 2.7. *Let A be an AFA and E the equational representation of A. Assume that the variable X_0 corresponds to the starting state of A. Then the solution for X_0 in E under the L-interpretation is exactly $L(A)$.* □

Theorem 2.8. *For each system of language equations of the form* (2.3.2), *there is an AFA A such that the solution for X_0 is equal to $L(A)$.* □

It is easy to observe that an AFA is a DFA if and only if each function $g_q(a, X)$, $q \in Q$ and $a \in \Sigma$, in its equational representation (2.3.2) is either a single variable or empty. An AFA is an NFA if and only if each function in its equational representation (2.3.2) is defined using only the $\vee$ operation.

Such systems of language equations and their solutions have been further studied in [74, 76, 77]. Naturally, one may view that each such system of language equations corresponds directly to a set of solutions in the form of extended regular expressions (which will be defined in Section 3.4). However, it remains open how we can solve such a general system of language equations by directly manipulating extended regular expressions without resorting to transformations of the corresponding AFA.

2.3.3 Normal forms

The following results have been proved in [45].

Theorem 2.9. *For any k-state AFA A, $k > 0$, there exists an equivalent k-state AFA A' with at most one final state. More precisely, A' has no final state if $\lambda \notin L(A)$ and A' has one final state otherwise. In the latter case, the starting state is the unique final state.* □

The proof of this theorem relies on the usage of the negation operation in AFA.

Theorem 2.10. *For each AFA $A = (Q, \Sigma, s, F, g)$, one can construct an equivalent AFA $A' = (Q', \Sigma, s', F', g')$ with $|Q'| \leq 2|Q|$ such that $g'_q(a)$ is defined with only the $\wedge$ and $\vee$ operations, for each $q \in Q'$ and $a \in \Sigma$. In other words, A' is an AFA without negations.* □

Theorem 2.11. *Let A be a k-state AFA without negations. One can construct an equivalent $(k+1)$-state AFA without negations that has one final state if $\lambda \notin L(A)$ and at most two final states otherwise.* □

In the following, we define a special type of AFA, which we call an s-AFA.

An s-AFA $A = (Q, \Sigma, s, F, g)$ is an AFA such that the value of $g_q(a)$, for any $q \in Q$ and $a \in \Sigma$, does not depend on the status of s, that is, in the equational representation of A, the variable X_s does not appear on the righthand side of any equation.

Example 2.3. The following is an equational representation of a 4-state s-AFA which accepts all words over $\{a, b\}$ that do not contain 6 consecutive occurrences of a. We use the convention that the operator $\wedge$ has precedence over $\vee$.

$$\begin{cases} X_0 &= a \cdot (\overline{X_1} \vee \overline{X_2}) + b \cdot (\overline{X_1} \vee \overline{X_2} \vee \overline{X_3}) + \lambda, \\ X_1 &= a \cdot (X_1 \vee X_2 \wedge X_3) + b \cdot (X_1 \wedge X_2 \wedge X_3), \\ X_2 &= a \cdot (X_1 \wedge X_2 \vee X_2 \wedge \overline{X_3} \vee \overline{X_2} \wedge X_3) + b \cdot (X_1 \wedge X_2 \wedge X_3), \\ X_3 &= a \cdot (X_1 \wedge X_2 \vee X_1 \wedge \overline{X_3} \vee X_2 \wedge \overline{X_3}) + b \cdot 1 + \lambda. \end{cases}$$ □

It is clear that for any AFA, there exists an equivalent s-AFA having at most one additional state.

2.3.4 AFA to NFA – the construction

Let $A = (Q, \Sigma, s, F, g)$ be an AFA and f the characteristic vector of F. We construct an NNFA

$$A_v = (Q_v, \Sigma, \delta_v, S_v, F_v)$$

where

$Q_v = B^Q$,

$S_v = \{u \in B^Q \mid u_s = 1\}$;

$F_v = \{f\}$,

$\delta_v : Q_v \times \Sigma \to 2^{Q_v}$ is defined by $\delta_v(u, a) = \{u' \mid g(a, u') = u\}$, for each $u \in Q_v$ and $a \in \Sigma$.

Claim. $L(A_v) = L(A)$.

Proof. We first prove that for $u \in Q_v$ $(= B^Q)$ and $x \in \Sigma^*$,

$$ux \vdash^*_{A_v} f \iff g(x, f) = u \tag{3}$$

by induction on the length of x.

For $x = \lambda$, one has $u = f$ and $g(\lambda, f) = f$. Now assume that the statement holds for all words up to a length l, and let $x = ax_0$ with $a \in \Sigma$ and $x_0 \in \Sigma^l$.

Let $u = g(x, f)$. Then we have $u = g(a, g(x_0, f))$. Let $u' = g(x_0, f)$. By the definition of δ_v, we have $u' \in \delta_v(u, a)$. We also have $u'x_0 \vdash_{A_v}^* f$ by the induction hypothesis. Therefore,

$$u\,x = u\,ax_0 \vdash_{A_v} u'\,x_0 \vdash_{A_v}^* f\ .$$

For the converse, let $u\,x \vdash_{A_v}^* f$. Then

$$u\,x = u\,ax_0 \vdash_{A_v}^* u'\,x_0 \vdash_{A_v}^* f$$

for some $u' \in Q_v$. Thus, $u' = g(x_0, f)$ by the induction hypothesis and $u = g(a, u')$ by the definition of δ_v. Therefore, $u = g(a, u') = g(a, g(x_0, f)) = g(x, f)$. Thus, (3) holds.

By (2.3.4) and the definition of S_v, we have $L(A_v) = L(A)$. □

In the above construction of A_v, the state set is $Q_v = B^Q$, i.e., each state of the NNFA A_v is a Boolean vector indexed by the states of the given AFA A. If the number of states of A is n, then the number of states of A_v is 2^n. Also notice that a computation of an AFA can be viewed as a sequence of calculations of Boolean vectors starting with f, the characteristic vector of F, as the initial vector and proceeding backwards with respect to the input word. At each step of this process, an input symbol is read and a new vector is calculated. Note that at each step, the new vector is uniquely determined. The process terminates when the first input symbol is read. Then the input word is accepted if and only if the resulting vector has a value 1 at the entry that is indexed by the starting state. We have the following results.

Theorem 2.12. *If L is accepted by an n-state AFA, then it is accepted by an NNFA with at most 2^n states.* □

Theorem 2.13. *If L is accepted by an n-state AFA, then L^R is accepted by a DFA with at most 2^n states.* □

2.3.5 NFA or DFA to AFA

NFA and DFA are special cases of AFA. So, the transformations are straightforward.

Let $A = (Q, \Sigma, \delta, s, F)$ be an NFA. We can construct an equivalent AFA $A' = (Q, \Sigma, s, F, g)$, where g is defined as follows: for each $q \in Q$, $a \in \Sigma$, and $u \in B^Q$,

$$g_q(a, u) = 0 \iff u_p = 0 \text{ for all } p \in \delta(q, a)\ .$$

More intuitively, the equational representation of A' is

$$X_q = \sum_{a \in \Sigma} a \cdot \bigvee_{p \in \delta(q,a)} X_p + \varepsilon_q, \qquad \text{for } q \in Q,$$

where $\varepsilon_q = \lambda$ if $q \in F$ and $\varepsilon_q = 0$ otherwise. A proof for $L(A) = L(A')$ can be found in [45].

Theorem 2.14. *L is accepted by a complete 2^k-state DFA if and only if L^R is accepted by a $(k+1)$-state s-AFA.*

Proof. The "if"-part is implied by Theorem 2.13. In the following, we describe the construction of an s-AFA for a given DFA but do not give a proof of its correctness. For a detailed proof, the reader can refer to [73, 44]. Let $D = (Q_D, \Sigma, \delta, s_D, F_D)$ be the given 2^k-state complete DFA and $L = L(D)$. We construct a $(k+1)$-state s-AFA $A = (Q_A, \Sigma, s_A, F_A, g)$ as follows. The main idea of the construction is that each of the 2^k states is encoded by a k-bit Boolean vector and each of the k bits is represented by a state of the AFA. In addition to these k states, the s-AFA has one more state, the starting state.

Let $K = \{1, \ldots, k\}$ and $K_0 = K \cup \{0\}$. Then we define K_0 to be the state set of the AFA A, where 0 is the starting state. We define an arbitrary bijection π between Q_D and B^K. The bijection π can be considered as an encoding scheme such that each state in Q_D is encoded by a distinct k-bit vector. For convenience, we simply use $\pi(q)$, i.e., the k-bit vector, to denote q in the following. In particular, we use the vector $(0, \ldots, 0)$ to denote the starting state s_D of D. Note that one can choose any of k-bit vector to encode s_D. We choose $(0, \ldots, 0)$ purely for notational conveniece. Then, we define a $(k+1)$-state s-AFA A as follows: $A = (Q_A, \Sigma, s_A, F_A, g)$ where

$Q_A = K_0$,

$s_A = 0$, and

$$F_A = \begin{cases} \{0\} & \text{if } s_D \in F_D, \\ \emptyset & \text{otherwise.} \end{cases}$$

The function g is defined by setting, for $a \in \Sigma$ and $u \in B^{Q_A}$,

$$g_0(a, u) = \begin{cases} 1 & \text{if } \delta(u|_K, a) \in F_D, \\ 0 & \text{otherwise} \end{cases}$$

and $v = g(a, u)$, for some $v \in B^{Q_A}$, if and only if $\delta(u|_K, a) = v|_K$. More precisely, we define $g_i(a, u)$, for $i \in K$ and $u \in B^{Q_A}$, in the following. Note that $\theta_z(x)$ denotes either x or $\overline{x}$ depending on the value of z, i.e., $\theta_z(x) = x$ if $z = 1$ and $\theta_z(x) = \overline{x}$ if $z = 0$. Then, for $i \in K$,

$$g_i(a, u) = \bigvee_{v \in B^K} (\delta(v, a)_i \wedge \theta_{v_1}(u_1) \wedge \ldots \wedge \theta_{v_k}(u_k))$$

and

$$g_0(a, u) = \bigvee_{v \in F_D} \theta_{v_1}(g_1(a, u)) \wedge \ldots \wedge \theta_{v_k}(g_k(a, u)). \qquad \square$$

Corollary 2.1. *Let A be an n-state DFA and $L = L(A)$. Then L^R is accepted by an s-AFA with at most $\lceil \log n \rceil + 1$ states.* $\square$

As an example, we construct a 3-state s-AFA A which is equivalent to the 4-state DFA A_1 of Figure 2 as follows:

$A = (Q_A, \Sigma, s_A, F_A, g)$ where $Q_A = \{0,1,2\}$, $s_A = 0$, $F_A = \emptyset$. The encoding of the states of A_1 is shown in the following. Note that we denote a 2-bit Boolean vector as a 2-bit binary number, i.e., we write X_1X_2 instead of (X_1, X_2).

State of A_1	0	1	2	3
Encoding X_1X_2	00	01	10	11

In order to explain intuitively how the function g is defined, we first write $g_1(a, X)$ informally (and in unnecessary detail) as follows:

$$\begin{aligned} g_1(a,X) &= (\delta(00,a)_1 \wedge \overline{X_1} \wedge \overline{X_2}) \vee (\delta(01,a)_1 \wedge \overline{X_1} \wedge X_2) \\ &\quad \vee (\delta(10,a)_1 \wedge X_1 \wedge \overline{X_2}) \vee (\delta(11,a)_1 \wedge X_1 \wedge X_2) \\ &= ((01)_1 \wedge \overline{X_1} \wedge \overline{X_2}) \vee ((01)_1 \wedge \overline{X_1} \wedge X_2) \vee ((01)_1 \wedge X_1 \wedge \overline{X_2}) \\ &\quad \vee ((11)_1 \wedge X_1 \wedge X_2) \\ &= (0 \wedge \overline{X_1} \wedge \overline{X_2}) \vee (0 \wedge \overline{X_1} \wedge X_2) \vee (0 \wedge X_1 \wedge \overline{X_2}) \vee (1 \wedge X_1 \wedge X_2) \\ &= X_1 \wedge X_2 \end{aligned}$$

Then we have

$$\begin{aligned} &g_1(a,X) = X_1 \wedge X_2, \\ &g_1(b,X) = (\overline{X_1} \wedge X_2) \vee (X_1 \wedge \overline{X_2}) \vee (X_1 \wedge X_2) = (\overline{X_1} \wedge X_2) \vee X_1 \\ &\qquad = X_1 \vee X_2, \\ &g_2(a,X) = (\overline{X_1} \wedge \overline{X_2}) \vee (\overline{X_1} \wedge X_2) \vee (X_1 \wedge \overline{X_2}) \vee (X_1 \wedge X_2) = 1, \\ &g_2(b,X) = ((X_1 \wedge \overline{X_2}) \vee (X_1 \wedge X_2) = X_1, \\ &g_0(a,X) = g_1(a,X) \wedge g_2(a,X) = (X_1 \wedge X_2) \wedge 1 = X_1 \wedge X_2, \\ &g_0(b,X) = g_1(b,X) \wedge g_2(b,X) = (X_1 \vee X_2) \wedge X_1 = X_1. \end{aligned}$$

So, the equational representation of A is

$$\begin{cases} X_0 &= a \cdot (X_1 \wedge X_2) + b \cdot (X_1) \\ X_1 &= a \cdot (X_1 \wedge X_2) + b \cdot (X_1 \vee X_2) \\ X_2 &= a \cdot 1 + b \cdot (X_1) \end{cases}$$

and the characteristic vector of F_A is $f = (0,0,0)$.

2.3.6 Basic operations

Let

$$A^{(1)} = (Q^{(1)}, \Sigma, s^{(1)}, F^{(1)}, g^{(1)})$$

be an $(m+1)$-state s-AFA and

$$A^{(2)} = (Q^{(2)}, \Sigma, s^{(2)}, F^{(2)}, g^{(2)})$$

be an $(n+1)$-state s-AFA. Assume that $Q^{(1)} \cap Q^{(2)} = \emptyset$.

We construct an $(m+n+1)$-state AFA $A = (Q, \Sigma, s, F, g)$ such that $L(A) = L(A^{(1)}) \cup L(A^{(2)})$ as follows:

$Q = (Q^{(1)} - \{s^{(1)}\}) \cup (Q^{(2)} - \{s^{(2)}\}) \cup \{s\}$,
$s \notin Q^{(1)} \cup Q^{(2)}$,

$$F = \begin{cases} F^{(1)} \cup F^{(2)} & \text{if } s^{(1)} \notin F^{(1)} \text{ and } s^{(2)} \notin F^{(2)}, \\ (F^{(1)} \cup F^{(2)} \cup \{s\}) \cap Q & \text{otherwise.} \end{cases}$$

We define g as follows. For $a \in \Sigma$ and $u \in B^Q$,

$$g_s(a,u) = g^{(1)}_{s^{(1)}}(a,u) \vee g^{(2)}_{s^{(2)}}(a,u),$$

and for $q \in Q - \{s\}$,

$$g_q(a,u) = \begin{cases} g^{(1)}_q(a,u) & \text{if } q \in Q^{(1)}, \\ g^{(2)}_q(a,u) & \text{if } q \in Q^{(2)}. \end{cases}$$

An $(m+n+1)$-state AFA $A = (Q, \Sigma, s, F, g)$ such that $L(A) = L(A^{(1)}) \cap L(A^{(2)})$ is constructed as above except the following:

$$g_s(a,u) = g^{(1)}_{s^{(1)}}(a,u) \wedge g^{(2)}_{s^{(2)}}(a,u)$$

and s is in F if and only if both $s^{(1)} \in F^{(1)}$ and $s^{(2)} \in F^{(2)}$.

For complementation, we construct an m-state s-AFA

$$A = (Q^{(1)}, \Sigma^{(1)}, s^{(1)}, F', g)$$

such that $L(A) = \overline{L(A^{(1)})}$, where the function g is the same as $g^{(1)}$ except that $g_{s^{(1)}}(a,u) = \overline{g^{(1)}_{s^{(1)}}(a,u)}$; and $F' = \{s^{(1)}\} \cup F^{(1)}$ if $s^{(1)} \notin F^{(1)}$ and $F' = F^{(1)} - \{s^{(1)}\}$ otherwise.

Let $L_1 = L(A^{(1)})$ and $L_2 = L(A^{(2)})$. We can easily construct an AFA to accept a language which is obtained by an arbitrary combination of Boolean operations on L_1 and L_2, e.g., $L = (L_1 \cup L_2) \cap \overline{(L_1 \cap L_2)}$. The only essential changes are the functions for s and whether s is in the final state set, which are all determined by the respective Boolean operations.

Other AFA operations, e.g., catenation, star, and shuffle, have been described in [45, 44].

2.3.7 Implementation and r-AFA

Although alternation is a generalization of nondeterminism, the reader may notice that AFA are backwards deterministic. We have also shown that a language L is accepted by a 2^n-state DFA if and only if it is accepted by an s-AFA of $n+1$ states reversely (i.e., words are read from right to left). Due to the above observation, we introduce a variation of s-AFA which we call r-AFA. The definition of an r-AFA is exactly the same as an s-AFA except that the input word is to be read reversely. Therefore, an r-AFA is forward deterministic. Then, for each L that is accepted by a DFA with n states, we can construct an equivalent r-AFA with at most $\lceil \log n \rceil + 1$ states.

An r-AFA $A = (Q, \Sigma, s, F, g)$ can be represented by a *system of right language equations* [19] of the following form:

$$X_q = \sum_{a \in \Sigma} g_q(a, X) \cdot a + \varepsilon_q, \quad q \in Q$$

where X is the vector of variables X_q, $q \in Q$, and

$$\varepsilon_q = \begin{cases} \lambda & \text{if } q \in F, \\ 0 & \text{otherwise,} \end{cases}$$

for each $q \in Q$.

In the following, we present a scheme such that Boolean functions of an r-AFA can be represented by Boolean vectors, and the computation of a Boolean function can be done with bitwise vector operations. Note that for a DFA of n states, its corresponding r-AFA has at most $\lceil \log n \rceil + 1$ states. So, for all practical problems, i.e., those using DFA of up to 2^{31} states, each Boolean vector can be stored in one word. In many cases, this can save space tremendously in comparison to symbolic representations of AFA. Also, each bitwise vector operation can be done with one instruction. So, AFA computations can be done efficiently.

We represent each Boolean function $g_q(a)$, $q \in Q$ and $a \in \Sigma$, in disjunctive normal form. The disjunctive normal form consists of a disjunction of formulas of the type $(Y_1 \wedge \ldots \wedge Y_m)$ where each Y_i is a variable X_i or the negation of a variable, $\overline{X_i}$. We call each such formula of the type $(Y_1 \wedge \ldots \wedge Y_m)$ a term. For example, let $X = (X_1, \ldots, X_8)$. The following Boolean function in disjunctive normal form

$$\mu(X) = (X_2 \wedge \overline{X_4} \wedge X_7) \vee (\overline{X_1} \wedge X_2) \vee (X_3 \wedge X_4 \wedge \overline{X_6})$$

has three terms. We name them $t^{(1)}(X)$, $t^{(2)}(X)$, and $t^{(3)}(X)$, respectively. Each term $t^{(i)}(X)$ can be represented by two 8-bit Boolean vectors $\alpha^{(i)}$ and $\beta^{(i)}$ and the value of $t_i(X)$ can be computed with two bitwise operations. The two Boolean vectors are defined as follows:

$$\alpha_k^{(i)} = 1 \text{ iff } X_k \text{ or } \overline{X_k} \text{ appears in } t^{(i)}$$

and

$$\beta_k^{(i)} = 1 \text{ iff } X_k \text{ appears in } t^{(i)}.$$

For example, the two vectors for $t^{(1)}(X)$ are

$$\alpha^{(1)} = (0,1,0,1,0,0,1,0),$$
$$\beta^{(1)} = (0,1,0,0,0,0,1,0).$$

Then, for any instance u of X, $t^{(1)}(u) = 1$ iff $(u \,\&\, \alpha^{(1)}) \uparrow \beta^{(1)} = 0$, where $\&$ and $\uparrow$ are bitwise AND and XOR, respectively, and 0 denotes the all-0 vector.

The above idea is based on the observation that a term $t(X)$ has a value 1 iff all the variables of the form X_i in $t(X)$ have a value 1 and all those of the form $\overline{X_i}$ in $t(X)$ have a value 0. For an instance u of X, $t(u)$ is evaluated with the above defined vectors α and β as follows. First, the vector α changes each u_i such that the variable X_i does not appear in $t(X)$ to 0 and keeps all others unchanged. Then the vector β changes each u_i such that X_i (rather than $\overline{X_i}$) is in $t(X)$ to $\overline{u_i}$, i.e., 1 if $u_i = 0$ and 0 if $u_i = 1$. Finally, $t(u)$ is 1 iff u becomes an all-0 vector.

Note that each term can be evaluated in parallel and each Boolean function of an r-AFA can be evaluated in parallel as well.

2.4 Finite automata with output

In the previous subsections, we have described three basic forms of finite automata: DFA, NFA, and AFA. They are all considered to be language acceptors. In this subsection, we consider several models of finite automata with output, which are not only language acceptors but also language transformers.

A *Moore* machine, informally, is a DFA where each state is associated with an output letter [88, 57]. Formally, a Moore machine A is a 6-tuple $(Q, \Sigma, \Delta, \delta, \sigma, s)$ where Q, Σ, δ, and s are defined as in a DFA; Δ is the output alphabet; and $\sigma : Q \to \Delta$ is the output function. For an input word $a_1 \cdots a_n$, if the state transition sequence is

$$s = q_0, q_1, \ldots, q_n$$

then the output of A in response to $a_1 \cdots a_n$ is

$$\sigma(q_0)\sigma(q_1)\cdots\sigma(q_n).$$

A *Mealy machine* is a DFA where an output symbol is associated to each transition rather than to each state [86, 57]. Formally, a Mealy machine A is a 5-tuple $(Q, \Sigma, \Delta, \sigma, s)$ where Q, Σ, and s are defined as in a DFA; Δ is the output alphabet; and $\sigma : Q \times \Sigma \to Q \times \Delta$ is the transition-and-output function. For an input word $x = a_1 \cdots a_n$, $a_1, \ldots, a_n \in \Sigma$, if $\sigma(s, a_1) = (q_1, b_1)$, $\sigma(q_1, a_2) = (q_2, b_2)$, ..., $\sigma(q_{n-1}, a_n) = (q_n, b_n)$, i.e.,

$$s \xrightarrow{a_1/b_1} q_1 \xrightarrow{a_2/b_2} \cdots\cdots \xrightarrow{a_n/b_n} q_n,$$

then the output of A in response to x is $b_1 \cdots b_n$.

For the above two models, we do not define final states. Final states can be defined such that only those input words that are accepted, i.e., reaching a final state, are associated to an output word. Then the models without final states are only a special case of the corresponding models with final states in the sense that all states are final states.

In the above definitions, we do not require that the transition functions are total. If an input word cannot be completely read, then there is no output word associated to this input word.

Another important model, the *finite transducer* model, is a generalization of the Mealy machines. Many closure properties of regular languages can be easily proved by using various finite transducers. See Section 4.2 for details.

A finite transducer T is a 6-tuple $(Q, \Sigma, \Delta, \sigma, s, F)$ where

Q is the finite set of states;
Σ is the input alphabet;
Δ is the output alphabet;
σ is the transition-and-output function from a finite subset of $Q \times \Sigma^*$ to finite subsets of $Q \times \Delta^*$;

$s \in Q$ is the starting state;
$F \subseteq Q$ is the set of final states.

An example of a finite transducer $T = (\{\mathbf{0}, \mathbf{1}, \mathbf{2}\}, \{a, b\}, \{0, 1\}, \sigma, \mathbf{0}, \{\mathbf{2}\})$ is shown in Figure 9. The arc from state **0** to state **1** with the label $b/101$ specifies that $(\mathbf{1}, 101) \in \sigma(\mathbf{0}, b)$.

For a given word $u \in \Sigma^*$, we say that $v \in \Delta^*$ is an output of T for u if there exists a state transition sequence of T, $(q_1, v_1) \in \sigma(s, u_1)$, $(q_2, v_2) \in \sigma(q_1, u_2)$, $\ldots$, $(q_n, v_n) \in \sigma(q_{n-1}, u_n)$, and $q_n \in F$, i.e.,

$$s \xrightarrow{u_1/v_1} q_1 \xrightarrow{u_2/v_2} \cdots\cdots \xrightarrow{u_n/v_n} q_n \in F$$

such that $u = u_1 \cdots u_n$, $u_1, \ldots, u_n \in \Sigma^*$, and $v = v_1 \cdots v_n$, $v_1, \ldots, v_n \in \Delta^*$. We write that $v \in T(u)$, where $T(u)$ denotes the set of all outputs of T for the input word u. Note that $s \in F$ implies that $\lambda \in T(\lambda)$.

T is said to be single-valued if for each input word u, T has at most one distinct output in response to u, i.e., $|T(u)| \leq 1$ for each $u \in \Sigma^*$.

A finite transducer $T = (Q, \Sigma, \Delta, \sigma, s, F)$ is called a *generalized sequential machine* (GSM) if σ is a function from $Q \times \Sigma$ to finite subsets of $Q \times \Delta^*$, i.e., T reads exactly one symbol at each transition. The GSM T is said to be deterministic if its underlying finite automaton (i.e., T without output) is a DFA, i.e., σ is a (partial) function from $Q \times \Sigma$ to $Q \times \Delta^*$. The definition of a GSM is not standardized in the literature. Some authors define GSMs with no final states [51].

Each finite transducer $T = (Q, \Sigma, \Delta, \sigma, s, F)$ defines a *finite transduction* $T : \Sigma^* \to 2^{\Delta^*}$. Note that for an input word $w \in \Sigma^*$, $T(w)$, which is the set of all output words in response to w, may be finite or infinite. $T(w) = \emptyset$ if T cannot reach a final state by reading w. Also note that we use T to denote both the finite transducer and the finite transduction it defines since this clearly will not cause any confusion. For a language $L \subseteq \Sigma^*$, we define

$$T(L) = \bigcup_{w \in L} T(w).$$

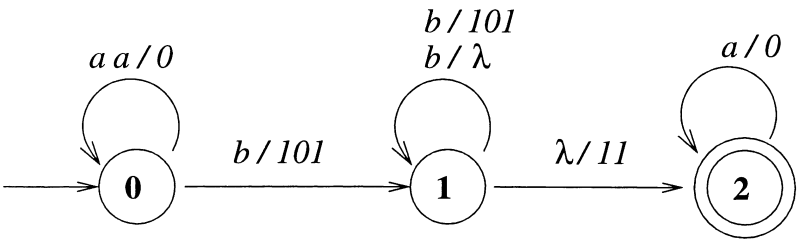


Fig. 9. A finite transducer T

Example 2.4. Let us consider the transducer T of Figure 9. We have

$$T(aabb) = \{010111, 010110111\},$$

$$T(bbba) = \{101110, 101101110, 101101101110\},$$

$$T(\lambda) = \emptyset, \qquad T(aaab) = \emptyset,$$

$$T(\{b, ba\}) = \{10111, 101110\}.$$

Let $L = \{a^i ba^j \mid i, j \geq 0\}$. Then

$$T(L) = \{0^k 101110^l \mid k, l \geq 0\}. \qquad \square$$

A finite transduction T can also be viewed as a relation $R_T \subseteq \Sigma^* \times \Delta^*$ defined by

$$R_T = \{(u, v) \mid v \in T(u)\}.$$

Relations induced by finite transducers are also called *rational relations* in the literature, e.g., [41]. The following is Nivat's Representation Theorem for finite transductions [93].

Theorem 2.15. *Let Σ and Δ be finite alphabets. $R \subseteq \Sigma^* \times \Delta^*$ is a rational relation iff there are a finite alphabet Γ, a regular language $L \subseteq \Gamma^*$ and morphisms $g : \Gamma^* \to \Sigma^*$ and $h : \Gamma^* \to \Delta^*$ such that*

$$R = \{(g(w), h(w)) \mid w \in L\}. \qquad \square$$

Two finite transducers are said to be *equivalent* if they define exactly the same finite transduction. The equivalence problem for finite transducers is undecidable [60]. This holds even for nondeterministic GSMs. However, the equivalence problem for single-valued finite transducers is decidable [114, 32]. This implies that the equivalence problem for deterministic GSMs (DGSMs) is also decidable.

From the above definitions, it is easy to see that morphisms can be characterized by one-state complete DGSMs. By a complete GSM, we mean that its transition-and-output function is a total function. Also, finite substitutions can be characterized by one-state (nondeterministic) GSMs. In both cases, the sole state is both the starting state and the final state.

For a function $T : \Sigma^* \to 2^{\Delta^*}$ (relation $R_T \subseteq \Sigma^* \times \Delta^*$), we define $T^{-1} : \Delta^* \to 2^{\Sigma^*}$ by $T^{-1}(y) = \{x \mid y \in T(x)\}$ ($R_T^{-1} \subseteq \Delta^* \times \Sigma^*$ by $R_T^{-1} = \{(y, x) \mid (x, y) \in R_T\}$). Then, clearly, T (R_T) is a finite transduction (rational relation) iff T^{-1} (R_T^{-1}) is a finite transduction (rational relation). This can be shown by simply interchanging the input and the output of the finite transducer. Then, we have the following:

Theorem 2.16. *Let $T : \Sigma^* \to 2^{\Delta^*}$ be a finite transduction. Then the inverse of T, i.e., $T^{-1} : \Delta^* \to 2^{\Sigma^*}$, is also a finite transduction.* $\square$

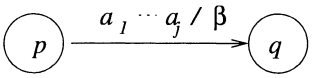
We define the following standard form for finite transducers.

A finite transducer $T = (Q, \Sigma, \Delta, \sigma, s, F)$ is said to be in the *standard form* if σ is a function from $Q \times (\Sigma \cup \{\lambda\})$ to $2^{Q \times (\Delta \cup \{\lambda\})}$. Intuitively, the standard form restricts the input and output of each transition to be only a single letter or λ.

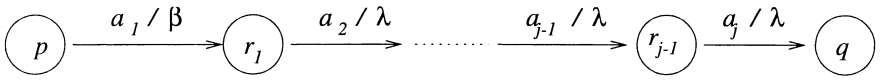
Theorem 2.17. *Each finite transducer can be transformed into an equivalent finite transducer in the standard form.* □

The transformation of an arbitrary finite transducer to an equivalent one in the standard form consists of two steps: First, each transition that reads more than one letter is transformed into several transitions reading exactly one letter. Second, each transition that has a string of more than one letter as output is transformed into several transitions such that each of them has exactly one letter as output.

More specifically, in the first step, we replace each transition of the form

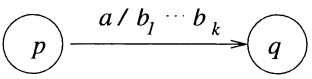


where $p, q \in Q$, $a_1, \ldots, a_j \in \Sigma$, $j \geq 2$, and $\beta \in \Delta^*$, by the following

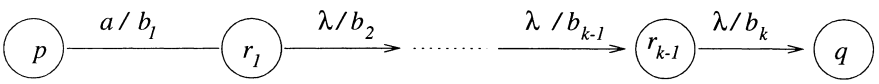


where $r_1, \ldots, r_{j-1}$ are new states.

For the second step, each of the transitions of the following form



where $p, q \in Q$, $a \in \Sigma \cup \{\lambda\}$, and $b_1, \ldots, b_k \in \Delta$, $k \geq 2$, is replaced by



where $r_1, \ldots, r_{k-1}$ are new states. It is clear that the two-step transformation results in an equivalent finite transducer in the standard form.

In many cases, the use of the standard form of finite transducers can result in much simpler proofs than the use of the general form. In Section 4.2, we

will use the standard form in proving that the family of regular languages is closed under finite transduction.

3. Regular expressions

In the previous section, we have defined languages that are recognized by finite automata. Finite automata in various forms are easy to implement by computer programs. For example, a DFA can be implemented by a case or switch statement; an NFA can be expressed as a matrix and manipulated by corresponding matrix operations; and an AFA can be represented as Boolean vectors and computed by bitwise Boolean operations. However, finite automata in any of the above mentioned forms are not convenient to be specified sequentially by users. For instance, when we specify a string pattern to be matched or define a token for certain identifiers, it is quite cumbersome to write a finite automaton definition for the purpose. In this case, a succinct and comprehensible expression in sequential form would be better suited than a finite automaton definition. For example, the language accepted by the finite automaton A_2 of Figure 3 can be expressed as $(a+b)^*abb(a+b)^*$. Such expressions are called regular expressions and they were originally introduced by Kleene [69]. In practice, regular expressions are often used as user interfaces for specifying regular languages. In contrast, finite automata are better suited as computer internal representations for storing regular languages.

3.1 Regular expressions – the definition

We define, inductively, a *regular expression* e over an alphabet Σ and the language $L(e)$ it denotes as follows:

(1) $e = \emptyset$ is a regular expression denoting the language $L(e) = \emptyset$.
(2) $e = \lambda$ is a regular expression denoting the language $L(e) = \{\lambda\}$.
(3) $e = a$, for $a \in \Sigma$, is a regular expression denoting the languge $L(e) = \{a\}$. Let e_1 and e_2 be regular expressions and $L(e_1)$ and $L(e_2)$ the languages they denote, respectively. Then
(4) $e = (e_1 + e_2)$ is a regular expression denoting the language $L(e) = L(e_1) \cup L(e_2)$.
(5) $e = (e_1 \cdot e_2)$ is a regular expression denoting the language $L(e) = L(e_1)L(e_2)$.
(6) $e = e_1^*$ is a regular expression denoting the language $(L(e_1))^*$.

We assume that $*$ has higher precedence than $\cdot$ and $+$, and $\cdot$ has higher precedence than $+$. A pair of parentheses may be omitted whenever the omission would not cause any confusion. Also, we usually omit the symbol $\cdot$ in regular expressions.

Example 3.1. Let $\Sigma = \{a, b, c\}$ and $L \subseteq \Sigma^*$ be the set of all words that contain *abcc* as a subword. Then L can be denoted by the regular expression $(a + b + c)^*abcc(a + b + c)^*$. □

Example 3.2. Let $L \subseteq \{0, 1\}^*$ be the set of all words that do not contain two consecutive 1's. Then L is denoted by $(10 + 0)^*(1 + \lambda)$. □

Example 3.3. Let $\Sigma = \{a, b\}$ and $L = \{w \in \Sigma^* \mid |w|_b \text{ is odd}\}$. Then L can be denoted by $(a^*ba^*b)^*a^*ba^*$. □

Two regular expressions e_1 and e_2 over Σ are said to be equivalent, denoted $e_1 = e_2$, if $L(e_1) = L(e_2)$. The languages that are denoted by regular expressions are called *regular languages*. The family of regular languages is denoted $\mathcal{L}_{REG}$.

In [69], Kleene has shown that the family of regular languages and the family of DFA languages are exactly the same, i.e., regular expressions are equivalent to finite automata in terms of the languages they define. There are various algorithms for transforming a regular expression to an equivalent finite automaton and vice versa. In the following, we will describe two approaches for the transformation from a regular expression to an equivalent finite automaton and one from a finite automaton to an equivalent regular expression.

3.2 Regular expressions to finite automata

There are three major approaches for transforming regular expressions into finite automata. The first approach, due to Thompson [121], is to transform a regular expression into a λ-NFA. This approach is simple and intuitive, but may generate many λ-transitions. Thus, the resulting λ-NFA can be unnecessarily large and the further transformation of it into a DFA can be rather time and space consuming. The second approach transforms a regular expression into an NFA without λ-transitions. This approach is due to Berry and Sethi [7], whose algorithm is based on Brzozowski's theory of derivatives [16] and McNaughton and Yamada's marked expression algorithm. Berry and Sethi's algorithm has been further improved by Brüggemann-Klein [13] and Chang and Paige [25]. The third approach is to transform a regular expression directly into an equivalent DFA [16, 2]. This approach is very involved and can be replaced by two separate steps: (1) regular expressions to NFA using one of the above approaches and (2) NFA to DFA.

In the following, we give a very brief description of the first approach and give an intuitive idea of the marked expression algorithm [7] that forms the basis of the second approach. Here we will not discuss the above mentioned third approach.

3.2.1 Regular expressions to λ-NFA

The following construction can be found in many introductory books on automata and formal language theory, e.g., [57, 68, 78, 123]. Our approach is different from that of Thompson's [121, 2, 57] in that the number of final states is not restricted to one.

Let e be a regular expression over the alphabet Σ. Then a λ-NFA M_e is constructed recursively as follows:

(*i*) If $e = \emptyset$, then $M_e = (\{s\}, \Sigma, \delta, s, \emptyset)$ where $\delta(s, a) = \emptyset$ for any $a \in \Sigma \cup \{\lambda\}$.
(*ii*) If $e = \lambda$, then $M_e = (\{s\}, \Sigma, \delta, s, \{s\})$ where δ is the same as in *i*).
(*iii*) If $e = a$, for some $a \in \Sigma$, then $M_e = (\{s, f\}, \Sigma, \delta, s, \{f\})$ where $\delta(s, a) = \{f\}$ is the only defined transition.
(*iv*) If $e = e_1 + e_2$ where e_1 and e_2 are regular expressions and M_{e_1} and M_{e_2} are λ-NFA constructed for e_1 and e_2, respectively, i.e., $L(M_{e_1}) = L(e_1)$ and $L(M_{e_2}) = L(e_2)$, then $M_e = M_{e_1} + M_{e_2}$, where $M_{e_1} + M_{e_2}$ is defined in Subsection 2.2.
Similarly, if $e = e_1 e_2$, then $M_e = M_{e_1} M_{e_2}$; and if $e = e_1^*$, then $M_e = M_{e_1}^*$, where $M_{e_1} M_{e_2}$ and $M_{e_1}^*$ are defined in Subsection 2.2.

Example 3.4. Following the above approach, the regular expression $a(a + b)a^*b$ would be transformed into the λ-NFA shown in Figure 10. □

3.2.2 Regular expressions to NFA without λ-transitions

The following presentation is a modification of the one given in [14]. An informal description is presented in Figure 11.

Let e be a regular expression over Σ. We define an NFA $\mathcal{M}_e$ inductively as follows:

(0) $\mathcal{M}_\emptyset = (\{s\}, \Sigma, \delta, s, \emptyset)$ where $\delta(s, a) = \emptyset$ for all $a \in \Sigma$.
(1) $\mathcal{M}_\lambda = (\{s\}, \Sigma, \delta, s, \{s\})$ where $\delta(s, a) = \emptyset$ for all $a \in \Sigma$.
(2) For $a \in \Sigma$, $\mathcal{M}_a = (\{s, f\}, \Sigma, \delta, s, \{f\})$ where $\delta(s, a) = \{f\}$ is the only transition.
(3) Assume that $\mathcal{M}_{e_1} = (Q_1, \Sigma, \delta_1, s_1, F_1)$, $\mathcal{M}_{e_2} = (Q_2, \Sigma, \delta_2, s_2, F_2)$, and $Q_1 \cap Q_2 = \emptyset$.

(3.1) $\mathcal{M}_{e_1 + e_2} = (Q, \Sigma, \delta, s_1, F)$ where
$Q = Q_1 \cup (Q_2 - \{s_2\})$ (merging s_1 and s_2 into s_1),

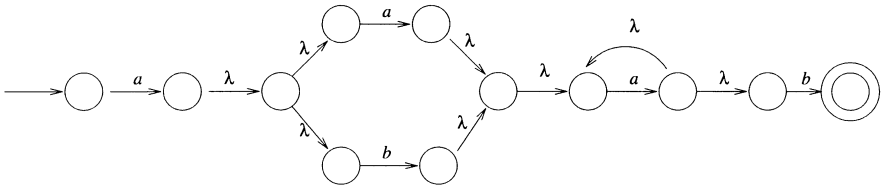
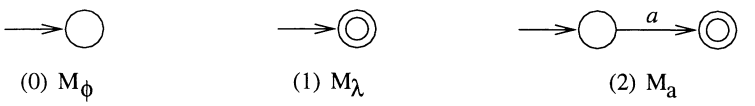
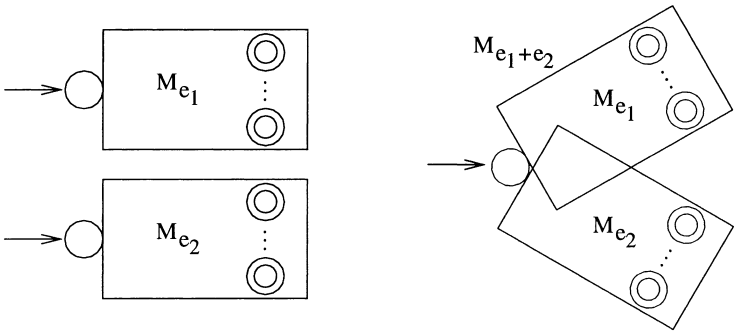


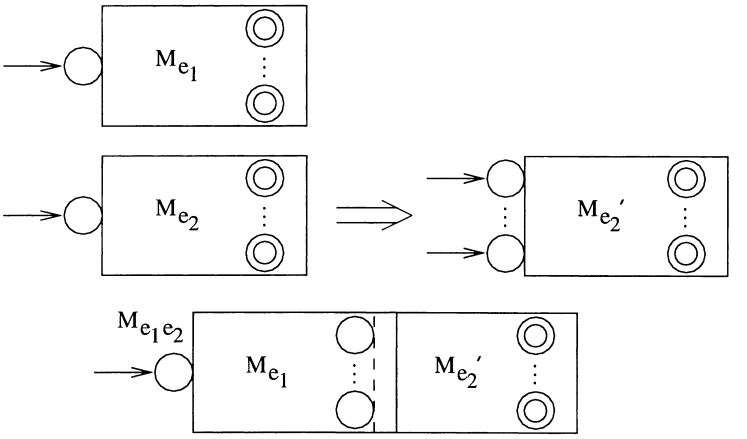
Fig. 10. A λ-NFA constructed for $a(a + b)a^*b$



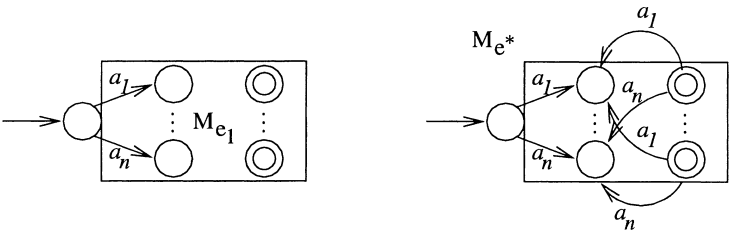
(0) M_ϕ (1) M_λ (2) M_a



(3.1) $M_{e_1+e_2}$



(3.2) $M_{e_1e_2}$



(3.3) M_{e^*}

Fig. 11. Regular expression to an NFA without λ-transitions

$$F = \begin{cases} F_1 \cup F_2 & \text{if } s_2 \notin F_2, \\ F_1 \cup (F_2 - \{s_2\}) \cup \{s_1\} & \text{otherwise;} \end{cases}$$
for $q \in Q$ and $a \in \Sigma$,

$$\delta(q,a) = \begin{cases} \delta_1(s_1,a) \cup \delta_2(s_2,a), & \text{if } q = s_1, \\ \delta_1(q,a) & \text{if } q \in Q_1, \\ \delta_2(q,a) & \text{if } q \in Q_2; \end{cases}$$

(3.2) $\mathcal{M}_{e_1 e_2} = (Q, \Sigma, \delta, s_1, F)$ where
$Q = Q_1 \cup (Q_2 - \{s_2\})$ (merging each state in F_1 with a copy of s_2),

$$F = \begin{cases} F_2 & \text{if } s_2 \notin F_2, \\ F_1 \cup (F_2 - \{s_2\}) & \text{if } s_2 \in F_2; \end{cases} \quad \text{for } q \in Q \text{ and } a \in \Sigma,$$

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{if } q \in Q_1 - F_1, \\ \delta_1(q,a) \cup \delta_2(s_2,a) & \text{if } q \in F_1, \\ \delta_2(q,a) & \text{if } q \in Q_2 - \{s_2\}; \end{cases}$$

(3.3) $\mathcal{M}_{e_1^*} = (Q, \Sigma, \delta, s_1, F)$ where
$Q = Q_1$,
$F = F_1 \cup \{s_1\}$,
for $q \in Q$ and $a \in \Sigma$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & \text{if } q \in Q_1 - F_1, \\ \delta_1(q,a) \cup \delta_1(s_1,a) & \text{if } q \in F_1. \end{cases}$$

Such NFA are called Glushkov automata in [14] and were first defined by Glushkov in [48]. Note that Glushkov automata have the property that the starting state has no incoming transitions. One may observe that the automaton constructed in step (0), (1), or (2) has no incoming transitions, and each operation in step (3) preserves the property.

A detailed proof of the following result can be found in [7].

Theorem 3.1. *Let e be an arbitrary regular expression over Σ. Then $L(e) = L(\mathcal{M}_e)$.* □

A regular expression e is said to be *deterministic* [14] if $\mathcal{M}_e$ is a DFA.

3.3 Finite automata to regular expressions

Here, we show that for a given finite automaton A, we can construct a regular expression e such that e denotes the language accepted by A. The construction uses *extended finite automata* where a transition between a pair of states is labeled by a regular expression. The technique we will describe in the following is called the *state elimination technique* [123]. For a given finite automaton, the state elimination technique deletes a state at each step and changes the transitions accordingly. This process continues until the FA contains only the starting state, a final state, and the transition between them. The regular expression labeling the transition specifies exactly the language accepted by A.

Let R_Σ denote the set of all regular expressions over the alphabet Σ. An *extended finite automaton* (EFA) is formally defined as follows:

Definition 3.1. *An EFA A is a quintuple $(Q, \Sigma, \delta, s, F)$ where*
Q is the finite set of states;
Σ is the input alphabet;
$\delta : Q \times Q \to R_\Sigma$ is the labeling function of the state transitions;
$s \in Q$ is the starting state;
$F \subseteq Q$ is the set of final states.

Note that we assume $\delta(p, q) = \emptyset$ if the transition from p to q is not explicitly defined.

A word $w \in \Sigma^*$ is said to be accepted by A if $w = w_1 \cdots w_n$, for $w_1, \ldots, w_n \in \Sigma^*$, and there is a state sequence $q_0, q_1, \ldots, q_n$, $q_0 = s$ and $q_n \in F$, such that $w_1 \in L(\delta(q_0, q_1))$, ..., $w_n \in L(\delta(q_{n-1}, q_n))$. The language accepted by A is defined accordingly.

First we describe the pivotal step of the algorithm, i.e., the elimination of one non-starting and non-final state. Then we give the complete state-elimination algorithm which repeatedly applies the above step and eventually transforms the given EFA to an equivalent regular expression.

Let $A = (Q, \Sigma, \delta, s, F)$ be an EFA. Denote by e_{pq} the regular expression $\delta(p, q)$, i.e., the label of the transition from state p to state q. Let q be a state in Q such that $q \neq s$ and $q \notin F$. Then an equivalent EFA $A' = (Q', \Sigma, \delta', s, F)$ such that $Q' = Q - \{q\}$, i.e., q is eliminated, is defined as follows: For each pair of states p and r in $Q' = Q - \{q\}$,

$$\delta'(p, r) = e_{pr} + e_{pq} e_{qq}^* e_{qr}.$$

We illustrate this step by the diagram in Figure 12, where state 1 is eliminated from the given EFA.

Now, we describe the complete algorithm.

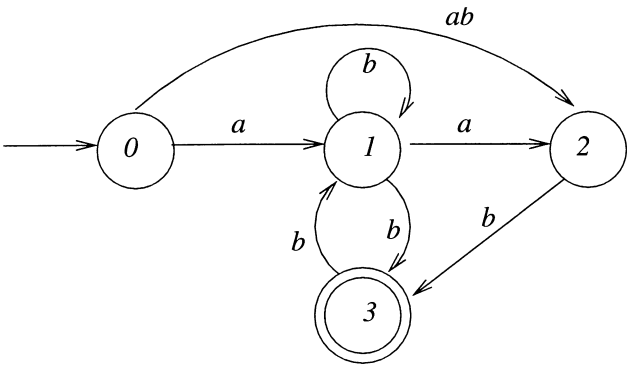
Let $A = (Q, \Sigma, \delta, s, F)$ be an EFA.

(1) (a) If the starting state is a final state or it has an incoming transition, i.e., $s \in F$ or $\delta(q, s) \neq \emptyset$ for some $q \in Q$, then add a new state s' to the state set and define $\delta(s', s) = \lambda$. Also define s' to be the starting state.

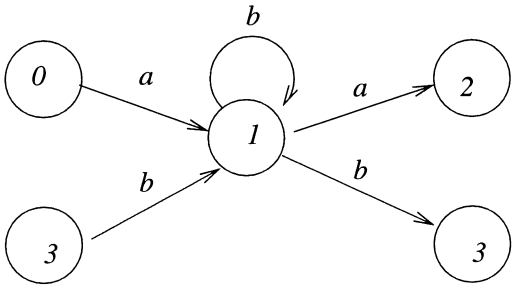
(b) If there are more than one final states, i.e., $|F| > 1$, then add a new state f' and new transitions $\delta(q, f') = \lambda$ for each q in F. Then, redefine the final state set to be $\{f'\}$.

Let $A' = (Q', \Sigma, \delta', s', F')$ denote the EFA after the above steps.

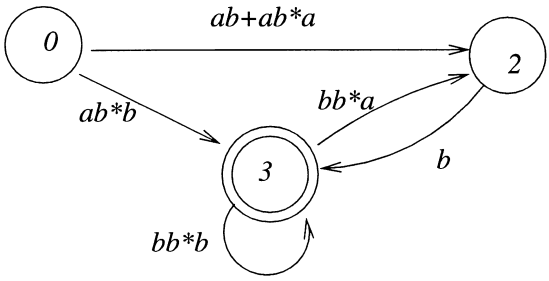
(2) If Q' consists only of s' and f', then the resulting regular expression is $e_{s'f'} e_{f'f'}^*$, where $e_{s'f'} = \delta'(s', f')$ and $e_{f'f'} = \delta'(f', f')$, and the algorithm terminates. Otherwise, continue to (3).



(a) Given EFA



(b) Working sheet for deleting State 1



(c) Resulting EFA after deleting State 1

Fig. 12. Deletion of a state from an EFA

(3) Choose $q \in Q'$ such that $q \neq s'$ and $q \neq f'$. Eliminate q from A' following the above description. Then the new state set will be $Q' - \{q\}$ and δ' is changed accordingly. Continue to (2).

Note that every DFA, NFA, or λ-NFA is an EFA. So, the above algorithm applies to all of them.

3.4 Star height and extended regular expressions

Among the three operators of regular expressions, the star operator is perhaps the most essential one. Regular expressions without the star operator define only finite languages. One natural measurement of the complexity of a regular expression is the number of nested stars in the expression, which is called the star height of the expression. Questions concerning star height were considered among the most fascinating problems in formal language theory. Some unsolved problems are still attracting researchers. In the following, we first give the basic definitions and then describe several of the most well known problems and results concerning star height. The interested reader may refer to [98] or [107] for details on the topic.

The star height of a regular expression e over the alphabet Σ, denoted $H(e)$, is a nonnegative integer defined recursively as follows:

(1) $H(e) = 0$, if $e = \emptyset$, λ, or a for $a \in \Sigma$.
(2) $H(e) = \max(H(e_1), H(e_2))$, if $e = (e_1 + e_2)$ or $e = (e_1 e_2)$, where e_1 and e_2 are regular expressions over Σ.
(3) $H(e) = H(e_1) + 1$, if $e = e_1^*$ and e_1 is a regular expression over Σ.

The star height of a regular language R, denoted $H(R)$, is the least integer h such that $H(e) = h$ for some regular expression e denoting R.

Example 3.5. Let $e_1 = (ab(cbc)^*(ca^* + c)^*)^* + b(ca^* + c)^*$. Then $H(e_1) = 3$. Let $e_2 = a(aaa^*)^*$ and $L = L(e_2)$. Then $H(e_2) = 2$ but $H(L) = 1$ because L is denoted also by $a + aaaa^*$ and L is of at least star height one since it is infinite. □

Concerning the star height of regular languages, one of the central questions is whether there exist languages of arbitrary star height. This question was answered by Eggan in 1963 [39]. He showed that for each integer $h \geq 0$ there exists a regular language R_h such that $H(R_h) = h$. However, in his proof the size of the alphabet for R_h grows with h. Solutions with a two-letter alphabet were given by McNaughton (unpublished notes mentioned in [18]) and later by Dejean and Schützenberger [35] in 1966.

Theorem 3.2. *For each integer $i \geq 0$, there exists a regular language R_i over a two-letter alphabet such that $H(R_i) = i$.* □

The language R_i, for each $i \geq 0$, is given by a regular expression e_i defined recursively as follows:

$$\begin{aligned} e_0 &= \lambda, \\ e_{i+1} &= (a^{2^i} e_i b^{2^i} e_i)^*, \ i \geq 0. \end{aligned}$$

Thus, for example,

$e_1 = (ab)^*$,
$e_2 = (a^2(ab)^*b^2(ab)^*)^*$,
$e_3 = (a^4(a^2(ab)^*b^2(ab)^*)^*b^4(a^2(ab)^*b^2(ab)^*)^*)^*$.

Clearly, $H(e_i) = i$. This implies that $H(R_i) \leq i$. The proof showing that $H(R_i)$ is at least i is quite involed. Detailed proofs can be found, e.g., in [106, 107].

Since there exist regular languages of arbitrary star height, one may naturally ask the following question: Does there exist an algorithm for determining the star height of a given regular language? This problem, often refered to as "the star height problem", was among the most well-known open problems on regular languages [18]. It had been open for more than two decades until it was solved by Hashiguchi [52] in 1988. The proof of the result is more than 40 pages long. The result by Hashiguchi can be stated as follows.

Theorem 3.3 (The Star Height). *There exists an algorithm which, for any given regular expression e, determines the star height of the language denoted by e.*

Generally speaking, almost all natural important properties are decidable for regular languages. The star height is an example of a property such that, although it is decidable, the proof of decidability is highly nontrivial.

In the following, we discuss the extended regular expressions as well as the extended star height problem.

An *extended regular expression* is one which allows the intersection $\cap$ and the complement $\neg$ operators in addition to the union, catenation, and star operators of a normal regular expression. We specify that the languages denoted by the expressions $(e_1 \cap e_2)$ and $\neg e_1$, respectively, are $L(e_1 \cap e_2) = L(e_1) \cap L(e_2)$ and $L(\neg e_1) = \overline{L(e_1)}$. We assume that $\cap$ has higher precedence than $+$ but lower precedence than $\cdot$ and $*$; and $\neg$ has the lowest precedence. For convenience, we use $\overline{e}$ to denote $\neg e$ in the following. A pair of parentheses may be omitted whenever the omission would not cause any confusion.

For instance, $\overline{\emptyset}$, $\overline{\lambda}$, and $a(a+b)^* \cap \overline{(a+b)^*bb(a+b)^*}$ are all valid extended regular expressions over $\Sigma = \{a, b\}$ denoting, respectively, Σ^*, Σ^+, and the set of all words that start with an a and contain no consecutive b's. Clearly, extended regular expressions denote exactly the family of regular languages.

The definition for the star height of an extended regular expression has the following two additions to the definition for a standard regular expression:

(4) $H(e) = \max(H(e_1), H(e_2))$, if $e = (e_1 \cap e_2)$;
(5) $H(e) = H(e_1)$, if $e = \overline{e_1}$;

where e_1 and e_2 are extended regular expressions over Σ. Similarly, the extended star height of a regular language R, denoted $H(R)$, is the least integer h such that $H(e) = h$ for some extended regular expression e denoting R.

The *star-free languages*, i.e., languages of extended star height zero, form the lowest level of the extended star height language hierarchy. It has been shown that there exist regular languages of extended star height one. However, the following problem which was raised in the sixties and formulated by Brzozowski [18] in 1979 remains open.

Open Problem *Does there exist a regular language of extended star height two or higher?*

Special attention has been paid to the family of star-free languages. The study of star-free languages was initiated by McNaughton [84, 85]. An interesting characterization theorem for star-free languages using noncounting (aperiodic) sets was proved by Schützenberger [113]. A set $S \subseteq \Sigma^*$ is said to be noncounting (aperiodic) if there exists an integer $n > 0$ such that for all $x, y, z \in \Sigma^*$, $xy^n z \in S$ iff $xy^{n+1}z \in S$. We state the characterization theorem below. The reader may refer to [113] or [98] for a detailed proof.

Theorem 3.4. *A regular language is star-free iff it is noncounting (aperiodic).* □

It appears that extended regular expressions correspond to AFA directly. It can be shown that the family of star-free languages can also be characterized by the family of languages accepted by a special subclass of AFA, which we call loop-free AFA. An AFA is said to be *loop-free* if there is a total order $<$ on the states of the AFA such that any state j does not depend on any state i such that $i < j$ or state j itself. The following result can be found in [110].

Theorem 3.5. *A regular language is star-free iff it is accepted by a loop-free AFA.* □

A special sublass of star-free languages which has attracted much attention is the *locally testable languages* [20, 41, 84]. Informally, for a locally testable language L, one can decide whether a word w is in L by looking at all subwords of w of a previously given length k.

For $k \geq 0$ and $x \in \Sigma^*$ such that $|x| \geq k$, denote by $pre_k(x)$ and $suf_k(x)$, respectively, the prefix and the suffix of length k of x, and by $int_k(x)$ (interior words) the set of all subwords of length k of x that occur in x in a position other than the prefix or the suffix. A language $L \subseteq \Sigma^*$ is said to be *k-testable* iff, for any words $x, y \in \Sigma^*$, the conditions $pre_k(x) = pre_k(y)$, $suf_k(x) = suf_k(y)$, and $int_k(x) = int_k(y)$ imply that $x \in L$ iff $y \in L$. A language is said to be *locally testable* if it is k-testable for some integer $k \geq 1$.

Many useful locally testable languages belong to a smaller class of languages, which are called *locally testable languages in the strict sense* [84]. A language $L \subseteq \Sigma^*$ is k-testable in the strict sense if there are finite sets $P, S, I \subset \Sigma^*$ such that, for all $x \in \Sigma^*$ of length at least k, $x \in L$ iff $pre_k(x) \in P$, $suf_k(x) \in S$, and $int_k(x) \subseteq I$. A language is said to be locally testable in the strict sense if it is k-testable in the strict sense for some integer $k > 0$. There are languages that are locally testable but not locally testable in the strict sense. For example, let L be the set of all words over $\{0, 1\}$ that contain either 000 or 111 as an interior word but not both. Then L is locally testable but not locally testable in the strict sense. The class of locally testable languages is closed under Boolean operations. This is not true for the class of languages that are locally testable in the strict sense.

More properties of locally testable languages can be found in [20, 41, 84, 98, 125].

3.5 Regular expressions for regular languages of polynomial density

Given a regular language, it is often useful to know how many words of a certain length are in the language, i.e., the density of the language. The study of densities of regular languages has a long history, see, e.g., [112, 41, 109, 10, 120]. Here, we consider the relationship between the densities of regular languages and the forms of the regular expressions denoting those languages. In particular, we consider the forms of regular expressions that denote regular languages of polynomial density.

For each language $L \subseteq \Sigma^*$, we define the *density function* of L

$$\rho_L(n) = |L \cap \Sigma^n|,$$

where $|S|$ denotes the cardinality of the set S. In other words, $\rho_L(n)$ counts the number of words of length n in L. If $\rho_L(n) = O(1)$, we say that L has a constant density; and if $\rho_L(n) = O(n^k)$ for some integer $k \geq 0$, we say that L has a polynomial density. Languages of constant density are called *slender languages* [34, 115]. Languages that have at most one word for each length are called *thin languages* [34].

The first theorem below characterizes regular languages of polynomial density with regular expressions of a specific form. A detailed proof can be found in [120]. Similar results can be found in [112, 41, 109, 10].

Theorem 3.6. *A regular language R over Σ has a density in $O(n^k)$, $k \geq 0$, iff R can be denoted by a finite union of regular expressions of the following form:*

$$xy_1^* z_1 \ldots y_{k+1}^* z_{k+1} \tag{4}$$

where $x, y_1, z_1, \ldots, y_{k+1}, z_{k+1} \in \Sigma^*$. □

The following result ([120]) shows that the number of states of a finite automaton A may restrict the order of the density function of $L(A)$.

Theorem 3.7. *Let R be a regular language accepted by a DFA of k states. If R has a polynomial density, then the function $\rho_R(n)$ is $O(n^{k-1})$.* □

Theorem 3.6 is a powerful tool in proving various properties of regular languages of polynomial density. As an application of Theorem 3.6, we show the following closure properties:

Theorem 3.8. *Let L_1 and L_2 be regular languages over Σ with $\rho_{L_1}(n) = \Theta(n^k)$ and $\rho_{L_2}(n) = \Theta(n^l)$. Then the following statements hold:*

(a) *If $L = prefix(L_1) = \{x \mid xy \in L_1$ for some $y \in \Sigma^*\}$, then $\rho_L(n) = \Theta(n^k)$.*
(b) *If $L = infix(L_1) = \{y \mid xyz \in L_1$ for some $x, z \in \Sigma^*\}$, then $\rho_L(n) = \Theta(n^k)$.*
(c) *If $L = suffix(L_1) = \{z \mid xz \in L_1$ for some $x \in \Sigma^*\}$, then $\rho_L(n) = \Theta(n^k)$.*
(d) *If $L = L_1 \cup L_2$, then $\rho_L(n) = \Theta(n^{\max(k,l)})$.*
(e) *If $L = L_1 \cap L_2$, then $\rho_L(n) = O(n^{\min(k,l)})$.*
(f) *If $L = L_1 L_2$, then $\rho_L(n) = O(n^{k+l})$.*
(g) *If $L = h(L_1)$ where h is an arbitrary morphism[30], then $\rho_L(n) = O(n^k)$.*
(h) *If $L = \frac{1}{m}(L_1) = \{x_1 \mid x_1 \ldots x_m \in L_1$, for $x_1, \ldots, x_m \in \Sigma^*$, and $|x_1| = \ldots = |x_m|\}$, then $\rho_L(n) = \Theta(n^k)$.*

Proof. We only prove (a) as an example. The rest can be similarly proved.

Since $\rho_{L_1}(n) = \Theta(n^k)$, by Theorem 3.6, L_1 can be specified as a finite union of regular expressions of the form:

$$xy_1^* z_1 \ldots y_{k+1}^* z_{k+1} \tag{5}$$

where $x, y_1, z_1, \ldots, y_{k+1}, z_{k+1} \in \Sigma^*$. Then clearly, L, where $L = prefix(L_1)$, can be specified as a finite union of regular expressions of the following forms:

$$\begin{array}{ll} x', & x' \text{ is a prefix of } x, \\ xy_1^* z_1 \ldots y_i^* y_i', & y_i' \text{ is a prefix of } y_i,\ 1 \le i \le k+1, \\ xy_1^* z_1 \ldots y_i^* z_i', & z_i' \text{ is a prefix of } z_i,\ 1 \le i \le k+1. \end{array}$$

Then, by Theorem 3.6, the density function $\rho_L(n)$ is in $O(n^k)$. Since L is a superset of L_1, we have $\rho_L(n) \ge \rho_{L_1}(n)$, i.e., $\rho_L(n) = \Omega(n^k)$. Thus, $\rho_L(n) = \Theta(n^k)$. □

It is clear that all regular languages with polynomial densities are of star height one. But, not all star-height one languages are of polynomial density. For example, the language $(ab+b)^*(a+\epsilon)$ is of exponential density. However, there is a relation between these two subclasses of regular languages, which is stated in the following theorem.

Theorem 3.9. *A regular language is of star height one if and only if it is the image of a regular language of polynomial density under a finite substitution.*

Proof. The *if* part is obvious. For the *only if* part, let E be a regular expression of star height one over an alphabet Σ. Denote by X the set of all regular expressions e (over Σ) such that e^* is a subexpression of E. Choose $\Delta = \Sigma \cup \hat{X}$, where $\hat{X} = \{\hat{e} \mid e \in X\}$. Let $\hat{E}$ be the regular expression over Δ that is obtained from E by replacing each subexpression of the form e^*, $e \in X$, by $\hat{e}^*$. By Theorem 3.6, $L(\hat{E})$ is a regular language of polynomial density. We define a finite substitution $\pi : \Delta^* \to 2^{\Sigma^*}$ as follows. For each $a \in \Sigma$, $\pi(a) \to \{a\}$ and for each $\hat{e} \in \hat{X}$, $\pi(\hat{e}) = L(e)$. Then clearly $\pi(L(\hat{E})) = L(E)$. □

It is clear that $\rho_\emptyset(n) = 0$ and $\rho_{\Sigma^*}(n) = |\Sigma|^n$. For each $L \subseteq \Sigma^*$, we have $\rho_\emptyset(n) \leq \rho_L(n) \leq \rho_{\Sigma^*}(n)$ for all $n \geq 0$. It turns out that there exist functions between $\rho_\emptyset(n)$ and $\rho_{\Sigma^*}(n)$ which are not the density function of any regular language. The following two theorems [120] show that, for the densities of regular languages, there is a gap between $\Theta(n^k)$ and $\Theta(n^{k+1})$, for each integer $k \geq 0$; and there is a gap between polynomial functions and exponential functions of the order $2^{\Theta(n)}$. For example, there is no regular language that has a density of the order $\sqrt{n}$, $n \log n$, or $2^{\sqrt{n}}$.

Theorem 3.10. *For any integer $k \geq 0$, there does not exist a regular language R such that $\rho_R(n)$ is neither $O(n^k)$ nor $\Omega(n^{k+1})$.* □

Theorem 3.11. *There does not exist a regular language R such that $\rho_R(n)$ is not $O(n^k)$, for any integer $k \geq 0$, and not of the order $2^{\Omega(n)}$.* □

It is not difficult to show that, for each nonnegative integer k, we can construct a regular language R such that $\rho_R(n)$ is exactly n^k. Therefore, for each polynomial function $f(n)$, there exists a regular language R such that $\rho_R(n)$ is $\Theta(f(n))$; and for each regular language R, either there exists a polynomial function $f(n)$ such that $\rho_R(n) = \Theta(f(n))$, or $\rho_R(n)$ is of the order $2^{\Theta(n)}$.

4. Properties of regular languages

4.1 Four pumping lemmas

There are many ways to show that a language is regular; for example, this can be done by demonstrating that the language is accepted by a finite automaton, specified by a regular expression, or generated by a right-linear grammar. To prove that a language is **not** regular, the most commonly used tools are the *pumping properties* of regular languages, which are usually stated as "pumping lemmas". The term "pumping" intuitively describes the property that any sufficiently long word of the language has a nonempty subword

that can be "pumped". This means that if the subword is replaced by an arbitrary number of copies of the same subword, the resulting word is still in the language.

There are many versions of pumping lemmas for regular languages. The "standard" version, which has appeared in many introductory books on the theory of computation, is a necessary but not sufficient condition for regularity, i.e., every regular language satisfies these conditions, but those conditions do not necessarily imply regularity. The first necessary and sufficient pumping lemma for regular languages was introduced by Jaffe [63]. Another necessary and sufficient pumping lemma, which is called "block pumping", was established by Ehrenfeucht, Parikh, and Rozenberg [40]. In contrast, for context-freeness of languages, only some necessary pumping conditions are known, but no conditions are known to be also sufficient.

In the following, we describe four pumping lemmas for regular languages: two necessary pumping lemmas and two necessary and sufficient pumping lemmas. We will give a proof for the first and the third, but omit the proofs for the second and the fourth. Examples will also be given to show how these lemmas can be used to prove the nonregularity of certain languages.

The first pumping lemma below was originally formulated in [5] and has appeared in many introductory books, see, e.g., [57, 108, 123, 27, 58].

Lemma 4.1. *Let R be a regular language over Σ. Then there is a constant k, depending on R, such that for each $w \in R$ with $|w| \geq k$ there exist $x, y, z \in \Sigma^*$ such that $w = xyz$ and*

(1) $|xy| \leq k$,
(2) $|y| \geq 1$,
(3) $xy^t z \in R$ *for all* $t \geq 0$.

Proof. Let R be accepted by a DFA $A = (Q, \Sigma, \delta, s, F)$ and k be the number of states of A, i.e., $k = |Q|$. For a word $w = a_1 \dots a_n \in R$, $a_1, \dots, a_n \in \Sigma$, we denote the computation of A on w by the following sequence of transitions:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n$$

where $q_0, \dots, q_n \in Q$, $q_0 = s$, $q_n \in F$, and $\delta(q_i, a_{i+1}) = q_{i+1}$ for all i, $0 \leq i < n$.

If $n \geq k$, the above sequence has states q_i and q_j, $0 \leq i < j \leq k$, such that $q_i = q_j$. Then for each $t \geq 0$, we have the following transition sequence:

$$s = q_0 \xrightarrow{a_1} \cdots \xrightarrow{a_i} \{q_i \xrightarrow{a_{i+1}} \cdots \xrightarrow{a_j}\}^t q_j \xrightarrow{a_{j+1}} \cdots \xrightarrow{a_n} q_n$$

where $\{\alpha\}^t$ denotes that α is being repeated t times. Let $x = a_1 \dots a_i$, $y = a_{i+1} \dots a_j$, and $z = a_{j+1} \dots a_n$. Then $xy^t z \in R$ for all $t \geq 0$, where $|xy| \leq k$ and $|y| \geq 1$. □

The lemma states that every regular language possesses the above pumping property. Therefore, any language that does not possess the property is

not a regular language. For example, one can easily show that the language $L = \{a^i b^i \mid i \geq 0\}$ is not regular using the above lemma. The arguments are as follows: Assume that L is regular and let k be the constant for the lemma. Choose $w = a^k b^k$ in L. Clearly, $|w| \geq k$. By the pumping lemma, $w = xyz$ for some $x, y, z \in \Sigma^*$ such that (1) $|xy| \leq k$, (2) $|y| \geq 1$, and (3) $xy^t z \in R$ for all $t \geq 0$. By (1) and (2), we have $y = a^m$, $1 \leq m \leq k$. But $xy^0 z = xz = a^{k-m} b^k$ is not in L. Thus, (3) does not hold. Therefore, L does not satisfy the pumping property of Lemma 4.1.

The pumping lemma has been used to show the nonregularity of many languages, e.g., the set of all binary numbers whose value is a prime [57], the set of all palindromes over a finite alphabet [58], and the set of all words of length i^2 for $i \geq 0$ [123].

However, not only regular languages but also some nonregular languages satisfy the pumping property of Lemma 4.1. Consider the following example.

Example 4.1. Let $L \subseteq \Sigma^*$ be an arbitrary *nonregular* language and

$$L_{\#} = (\#^+ L) \cup \Sigma^*$$

where $\# \notin \Sigma$. Then $L_{\#}$ satisfies the conditions of Lemma 4.1 with the constant k being 1. For any word $w \in \#^+ L$, we can choose $x = \lambda$ and $y = \#$, and for any word $w \in \Sigma^*$, we choose $x = \lambda$ and y to be the first letter of w. However, $L_{\#}$ is not regular, which can be shown as follows. Let h be a morphism defined by $h(a) = a$ for each $a \in \Sigma$ and $h(\#) = \lambda$. Then

$$L = h(L_{\#} \cap \#^+ \Sigma^*).$$

Clearly, $\#^+ \Sigma^*$ is regular. Assume that $L_{\#}$ is regular. Then L is regular since regular languages are closed under intersection and morphism (which will be shown in Section 4.2). This contradicts the assumption. Thus, $L_{\#}$ is not regular. □

Note that $L_{\#}$ is at the same level of the Chomsky hierarchy as L. So, there are languages at all levels of the Chomsky hierarchy, even non-recursively enumerable languages, that satisfy Lemma 4.1.

Note also that, for each language $L \subseteq \Sigma^*$, we can construct a distinct language $L_{\#} \subseteq (\Sigma \cup \{\#\})^*$ that satisfies Lemma 4.1. Consequently, there are uncountably many nonregular languages that satisfy the pumping lemma.

Below, we give two more examples of nonregular languages that satisfy the pumping condition of Lemma 4.1. They are quite simple and interesting.

Example 4.2. Let $L \subseteq b^*$ be an arbitrary nonregular language. Then the following languages are nonregular, but satisfy the pumping condition of Lemma 4.1:

(1) $a^+ L \cup b^*$,
(2) $b^* \cup aL \cup aa^+ \{a, b\}^*$.

Note that the first example above is just a simplified version of the language given in Example 4.1, with the alphabet Σ being a singleton. □

Lemma 4.2. *Let R be a regular language over Σ. Then there is a constant k, depending on R, such that for all $u, v, w \in \Sigma^*$, if $|w| \geq k$ then there exist $x, y, z \in \Sigma^*$, $y \neq \lambda$ such that $w = xyz$ and for all $t \geq 0$*

$$[uxy^t zv \in L \text{ iff } uwv \in L. \qquad \square$$

Any language that satisfies the pumping condition of Lemma 4.2 satisfies also the pumping condition of Lemma 4.1. This follows by setting $u = \lambda$ and $|w| = k$ in the condition of Lemma 4.2. However, the converse is not true. We can show that there exist languages that satisfy the pumping condition of Lemma 4.1, but do not satisfy that of Lemma 4.2. For example, let $L = \{a^i b^i \mid i \geq 0\}$ and consider the language $L_{\#} = \#^+ L \cup \{a, b\}^*$ as in Example 4.1. Clearly, $L_{\#}$ satisfies the pumping condition of Lemma 4.1. However, if we choose $u = \#$, $v = \lambda$, and $w = a^k b^k$ for Lemma 4.2 where k is the constant (corresponding to $L_{\#}$), it is clear that there do not exist x, y, z as required by the lemma. Therefore, the set of languages that satisfy the pumping condition of Lemma 4.2 is a proper subset of the set of languages that satisfy the condition of Lemma 4.1. In other words, Lemma 4.2 can rule out more nonregular languages. In this sense, we say that Lemma 4.2 is a stronger pumping lemma for regular languages than Lemma 4.1.

Nevertheless, Lemma 4.2 still does not give a sufficient condition for regularity. We show in the following that there exist nonregular languages that satisfy the pumping condition of Lemma 4.2. In fact, the number of such languages is uncountable. A different proof was given in [40].

Example 4.3. Let L be an arbitrary nonregular language over Σ and $\$ \notin \Sigma$. Define

$$\begin{aligned} L_{\$} = {} & \{\$^+ a_1 \$^+ a_2 \$^+ \ldots \$^+ a_m \$^+ \mid a_1 a_2 \ldots a_m \in L,\ a_1, a_2, \ldots, a_m \in \Sigma, m \geq 0\} \\ & \cup \{\$^+ x_1 \$^+ x_2 \$^+ \ldots \$^+ x_n \$^+ \mid x_1, x_2, \ldots, x_n \in \Sigma^*,\ n \geq 0, |x_i| \neq 1 \\ & \text{for some } i, 1 \leq i \leq n\}. \end{aligned}$$

We can easily prove that $L_{\$}$ is nonregular. Let $\Sigma_{\$}$ denote $\Sigma \cup \{\$\}$. We now show that $L_{\$}$ satisfies the pumping condition of Lemma 4.2. Let $k = 3$ be the constant for the pumping lemma. To establish the nontrivial implication of the statement of the lemma, it suffices to show that for any $u, w, v \in \Sigma_{\$}^*$ with $uwv \in L$ and $|w| \geq 3$, there exist $x, y, z \in \Sigma_{\$}^*$ with $w = xyz$ and $y \neq \lambda$ such that $uxy^i zv \in L_{\$}$ for all $i \geq 0$. We can choose $y = \$$ if w contains a $\$$ and $y = a$ for some $a \in \Sigma$ if w does not contain any symbol $\$$. $\square$

The next pumping lemma, introduced by Jaffe [63], gives a necessary and sufficient condition for regularity. A detailed proof of the following lemma can be found also in [108].

Lemma 4.3. *A language* $\mathrm{L} \in \Sigma^*$ *is regular* **iff** *there is a constant* $\mathrm{k} > 0$ *such that for all* $\mathrm{w} \in \Sigma^*$, *if* $|w| \geq k$ *then there exist* $\mathrm{x}, \mathrm{y}, \mathrm{z} \in \Sigma^*$ *such that* $\mathrm{w} = \mathrm{xyz}$ *and* $\mathrm{y} \neq \lambda$, *and for all* $\mathrm{i} \geq 0$ *and all* $\mathrm{v} \in \Sigma^*$, $\mathrm{wv} \in \mathrm{L}$ *iff* $\mathrm{xy^izv} \in \mathrm{L}$.

Proof. The *only if* part is relatively straightforward. Let A be a complete DFA which accepts L and k the number of states of A. For any word w of length $l \geq k$, i.e., $w = a_1 a_2 \cdots a_l$, let the state transition sequence of A on w be the following:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_l} q_l,$$

where q_0 is the starting state. Since there are at most k distinct states among $q_0, q_1, \ldots, q_l$ and $k < l+1$, it follows that $q_i = q_j$ for some i, j, $0 \leq i < j \leq l$. This implies that the transition from q_i to q_j is a loop back to the same state. Let $x = a_1 \cdots a_i$, $y = a_{i+1} \cdots a_j$, and $z = a_{j+1} \cdots a_l$ ($x = \lambda$ if $i = 1$ and $z = \lambda$ if $j = l$). Then, for all $i \geq 0$,

$$\delta^*(q_0, xy^iz) = q_l,$$

i.e., A is in the same state q_l after reading each word xy^iz, $i \geq 0$. Therefore, for all $i \geq 0$ and for all $v \in \Sigma^*$, $xy^izv \in L$ iff $wv \in L$.

For the *if* part, let L be a language which satisfies the pumping condition of the lemma and k be the constant. We prove that L is regular by constructing a DFA A_L using the pumping property of L and then proving that $L(A_L) = L$.

The DFA $A_L = (Q, \Sigma, \delta, s, F)$ is defined as follows. Each state in Q corresponds to a string w, in Σ^*, of length less than k, i.e.,

$$Q = \{q_w \mid w \in \Sigma^* \text{ and } |w| \leq k-1\},$$

$s = q_\lambda$ and $F = \{q_w \in Q \mid w \in L\}$. The transition function δ is defined as follows:

(1) If $|w| < k-1$, then for each $a \in \Sigma$,

$$\delta(q_w, a) = q_{wa}.$$

(2) If $|w| = k-1$, then by the pumping property of L, for each $a \in \Sigma$, wa can be decomposed into xyz, $y \neq \lambda$, such that for all $v \in \Sigma^*$, $xyzv \in L$ iff $xzv \in L$. There may be a number of such decompositions. We choose the one such that xy is the shortest (and y is the shortest if there is a tie). Then define

$$\delta(q_w, a) = q_{xz}.$$

Now we show that the language accepted by A_L is exactly L. We prove this by induction on the length of a word $w \in \Sigma^*$. It is clear that for all words w such that $|w| < k$, $w \in L(A_L)$ iff $w \in L$ by the definition of A_L. We hypothesize that for all words shorter than n, $n \geq k$, $w \in L(A_L)$ iff $w \in L$. Consider a word w with $|w| = n$. Let $w = w_0 v$ where $|w_0| = k$. By the construction of A_L, we have $\delta^*(s, w_0) = \delta^*(s, xz) = q_{xz}$ for some $x, z \in \Sigma^*$

where $w_0 = xyz$, $y \in \Sigma^+$, and for any $v' \in \Sigma^*$, $w_0v' \in L$ iff $xzv' \in L$. We replace the arbitrary v' by v, then we have that $w \in L$ iff $xzv \in L$. Since xz and w_0 reach the same state in A_L, xzv and $w = w_0v$ will reach the same state, i.e., $w \in L(A_L)$ iff $xzv \in L(A_L)$. Notice that $|xzv| < n$. By the hypothesis, $xzv \in L(A_L)$ iff $xzv \in L$. So, we conclude that $w \in L(A_L)$ iff $w \in L$. □

Example 4.4. Let $L = \{a^ib^i \mid i \geq 0\}$ and $L_\# = (\#^+L) \cup \{a, b\}^*$. We have shown that $L_\#$ satisfies the pumping condition of Lemma 4.1. Now we demonstrate that $L_\#$ does not satisfy the pumping condition of Lemma 4.3. Assume the contrary. Let $k > 0$ be the constant of Lemma 4.3 for $L_\#$. Consider the word $w = \#a^kb^k$ and any decomposition $w = xyz$ such that $y \neq \lambda$. If y does not contain the symbol #, i.e., $y \in a^+$, $y \in b^+$, or $y \in a^+b^+$, then let $v = \lambda$ and, clearly, $wv \in L_\#$ but $xy^2zv \notin L_\#$. If y contains the symbol #, then let $v = a$ and we have $wv = xyzv \notin L_\#$ but $xzv \in L_\#$. So, $L_\#$ does not satisfy the pumping condition of Lemma 4.3. □

Notice that Lemma 4.3 requires a decomposition $w = xyz$ that works for **all** wv, $v \in \Sigma^*$. Another necessary and sufficient pumping lemma for regularity, which does not require this type of global condition, was given by Ehrenfeucht, Parikh, and Rozenberg [40]. The latter is called the block pumping lemma, which is very similar to Lemma 4.2 except that the decomposition of w into xyz has to be along the given division of w into subwords (blocks) $w_1, \ldots, w_k$, i.e., each of x, y, and z has to be a catenation of those subwords.

Lemma 4.4. *(Block pumping) $L \subseteq \Sigma^*$ is regular iff there is a constant $k > 0$ such that for all $u, v, w \in \Sigma^*$, if $w = w_1 \cdots w_k$, $w_1, \ldots, w_k \in \Sigma^*$, then there exist m, n, $1 \leq m < n \leq k$, such that $w = xyz$ with $y = w_{m+1} \ldots w_n$, $x, z \in \Sigma^*$, and for all $i \geq 0$,*

$$uwv \in L \text{ iff } uxy^izv \in L.$$ □

Example 4.5. Let $L = \{a^ib^i \mid i \geq 0\}$ and let $L_\$$ be defined as in Example 4.3. We have shown in Example 4.3 that $L_\$$ satisfies the pumping property of Lemma 4.2. Here we show that $L_\$$ does not satisfy the pumping property of Lemma 4.4. Assume the contrary. Let k be the constant in the lemma and choose $u = \lambda$, $w_1 = \$a$, $w_2 = \$a$, ..., $w_k = \$a$, $v = (\$b)^k\$$, and $w = w_1 \cdots w_k$. Then $uwv \in L_\$$. But, clearly, there do not exist m, n, $1 \leq m < n \leq k$, such that $y = w_{m+1} \ldots w_n$, $w = xyz$, and $uxzv = uw_1 \cdots w_m w_{n+1} \cdots w_k v = (\$a)^{k-n+m}(\$b)^k\$ \in L_\$$. □

In Lemma 4.4, the pumping condition is sufficient for the regularity of L even if we change the statement "for all $i \geq 0$" to "for $i = 0$". Then the pumping property becomes a cancellation property. It has been shown that the pumping and cancellation properties are equivalent [40]. A similar result can also be obtained for Lemma 4.3.

4.2 Closure properties

The following theorem has been established in Section 2. and 3..

Theorem 4.1. *The family of regular languages is closed under the following operations:* (1) *union,* (2) *intersection,* (3) *complementation,* (4) *catenation,* (5) *star, and* (6) *reversal.* □

The next theorem is a remarkably powerful tool for proving other properties of regular languages.

Theorem 4.2. *The family of regular languages is closed under finite transduction.*

Proof. Let L be an arbitrary regular language accepted by a DFA $A = (Q_A, \Sigma, \delta, s, F)$ and $T = (Q_T, \Sigma, \Delta, \sigma_T, s_T, F_T)$ a finite transducer in the standard form. We show that $T(L)$ is regular.

Construct a λ-NFA $R = (Q_R, \Delta, \delta_R, s_R, F_R)$ where
$Q_R = Q_A \times Q_T$;
$s_R = (s_A, s_T)$;
$F_R = F_A \times F_T$;
δ_R is defined by, for $(p, q) \in Q_R$ and $b \in \Delta \cup \{\lambda\}$,

$$\delta_R((p,q),b) = \{(p',q') \mid \text{there exists } a \in \Sigma \text{ such that}$$

$$\delta_A(p,a) = p' \ \&\ (q',b) \in \sigma_T(q,a), \text{or } (q',b) \in \sigma_T(q,\lambda) \ \&\ p = p'\}.$$

Now we show that $L(R) = T(L)$.

Let w be accepted by R. Then there is a state transition sequence of R

$$(s_A, s_T) \xrightarrow{b_1} (p_1, q_1) \xrightarrow{b_2} \cdots\cdots \xrightarrow{b_n} (p_n, q_n)$$

where $w = b_1 \cdots b_n$, $b_1, \ldots, b_n \in \Delta \cup \{\lambda\}$, and $p_n \in F_A$, $q_n \in F_T$. By the definition of R, there exist $a_1, \ldots, a_n \in \Sigma \cup \{\lambda\}$ such that

$$s_T \xrightarrow{a_1/b_1} q_1 \xrightarrow{a_2/b_2} \cdots\cdots \xrightarrow{a_n/b_n} q_n.$$

Let $a_{i_1}, \ldots, a_{i_m}$ be the non-λ subsequence of $a_1, \ldots, a_n$, i.e., $a_{i_1}, \ldots, a_{i_m} \in \Sigma$ and $u = a_{i_1} \cdots a_{i_m} = a_1 \cdots a_n$. Note that if $a_k = \lambda$, then $p_{k-1} = p_k$ (assuming $p_0 = s_A$). Thus, we have

$$s_A \xrightarrow{a_{i_1}} p_{i_1} \xrightarrow{a_{i_2}} \cdots\cdots \xrightarrow{a_{i_m}} p_{i_m} = p_n.$$

So, u is accepted by A and $w \in T(u)$. Therefore, $w \in T(L)$.

Let $u \in L(A)$ and $T(u) = w$. We prove that $w \in L(R)$. Since $T(u) = w$, there is a state transition sequence of T

$$s_T = q_0 \xrightarrow{a_1/b_1} q_1 \xrightarrow{a_2/b_2} \cdots\cdots \xrightarrow{a_n/b_n} q_n$$

for $u = a_1 \cdots a_n$, $a_1, \ldots, a_n \in \Sigma \cup \{\lambda\}$, $w = b_1 \cdots b_n$, $b_1, \ldots, b_n \in \Delta \cup \{\lambda\}$, and $q_n \in F_T$. Let $a_{i_1}, \ldots, a_{i_m}$ be the non-λ subsequence of $a_1, \ldots, a_n$, i.e., $a_{i_1} \cdots a_{i_m} = a_1 \cdots a_n = u$. Since $u \in L(A)$, we have

$$s_A = p_0 \xrightarrow{a_{i_1}} p_1 \xrightarrow{a_{i_2}} \cdots\cdots \xrightarrow{a_{i_m}} p_m,$$

where $p_m \in F_A$. Then, by the construction of R, there exists a state transition sequence of R

$$r_0 \xrightarrow{b_1} r_1 \xrightarrow{b_2} \cdots\cdots \xrightarrow{b_n} r_n$$

where $r_0 = s_R = (p_0, q_0)$ and for each j, $1 \leq j \leq n$, $r_j = (p_k, q_j)$ if $a_j \neq \lambda$ and $j = i_k$; $r_j = (p_{k-1}, q_j)$ if $a_j = \lambda$ and $i_{k-1} < j < i_k$, $1 \leq k \leq m$. Thus, $w \in L(R)$. □

Many operations can be implemented by finite transducers. Thus, the fact that regular languages are closed under those operations follows immediately by the above theorem. We list some of the operations in the next theorem.

Theorem 4.3. *The family of regular languages is closed under the following operations (assuming that $L \subseteq \Sigma^*$):*

(1) $\text{prefix}(L) = \{x \mid xy \in L,\ x, y \in \Sigma^*\}$,
(2) $\text{suffix}(L) = \{y \mid xy \in L,\ x, y \in \Sigma^*\}$,
(3) $\text{infix}(L) = \{y \mid xyz \in L,\ x, y, z \in \Sigma^*\}$,
(4) morphism,
(5) finite substitution,
(6) inverse morphism,
(7) inverse finite substitution.

Proof. (4) and (5) are obvious since they are only special cases of finite transductions: morphisms can be represented as one-state deterministic finite transducers (DGSMs) and finite substitutions can be represented as one-state (nondeterministic) finite transducers without λ-transitions. Note that, in both cases, the sole state is both the starting state and the final state.

(6) and (7) are immediate since, by Theorem 2.16, an inverse finite transduction is again a finite transduction.

Each of the operations (1)–(3) can be realized by a finite transducer given below. We omit the proof showing, in each case, the equality of the transduction and the operation in question. Figure 13 gives the transducers in the case where $\Sigma = \{a, b\}$.

(1) $T_{pre} = (Q_1, \Sigma, \Sigma, \sigma_1, s_1, F_1)$ where $Q_1 = \{1, 2\}$, $s_1 = 1$, $F_1 = Q_1$, and σ_1:
$\sigma_1(1, a) = \{(1, a), (2, \lambda)\}$, for each $a \in \Sigma$;
$\sigma_1(2, a) = \{(2, \lambda)\}$, for each $a \in \Sigma$.
(2) $T_{suf} = (Q_2, \Sigma, \Sigma, \sigma_2, s_2, F_2)$ where $Q_2 = \{0, 1\}$, $s_2 = 0$, $F_2 = Q_2$, and σ_2:
$\sigma_2(0, a) = \{(0, \lambda), (1, a)\}$, for each $a \in \Sigma$;
$\sigma_2(1, a) = \{(1, a)\}$, for each $a \in \Sigma$.

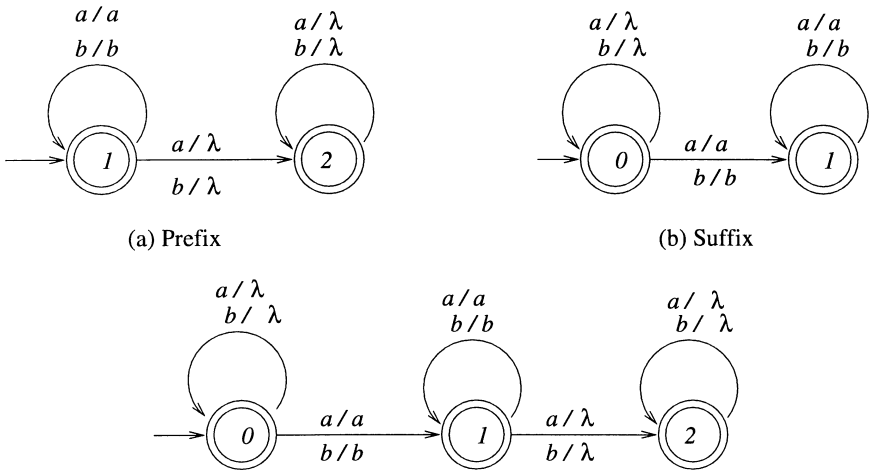


Fig. 13. Finite transducers realizing the prefix, suffix, and infix operations

(3) $T_{inf} = (Q_3, \Sigma, \Sigma, \sigma_3, s_3, F_3)$ where $Q_3 = \{0, 1, 2\}$, $s_3 = 0$, $F_3 = Q_3$, and σ_3:
$\sigma_3(0, a) = \{(0, \lambda), (1, a)\}$, for each $a \in \Sigma$;
$\sigma_3(1, a) = \{(1, a), (2, \lambda)\}$, for each $a \in \Sigma$;
$\sigma_3(2, a) = \{(2, \lambda)\}$, for each $a \in \Sigma$. □

A substitution $\varphi : \Sigma^* \to 2^{\Delta^*}$ is called a regular substitution if, for each $a \in \Sigma$, $\varphi(a)$ is a regular language. The reader can verify that each regular substitution can be specified by a finite transduction. Thus, we have the following:

Theorem 4.4. *The family of regular languages is closed under regular substitution and inverse regular substitution.* □

Let L be an arbitrary language over Σ. For each $x \in \Sigma^*$, the left-quotient of L by x is the set

$$x\backslash L = \{y \in \Sigma^* \mid xy \in L\},$$

and for a language $L_0 \subseteq \Sigma^*$, the left-quotient of L by L_0 is the set

$$L_0\backslash L = \bigcup_{x \in L_0} x\backslash L = \{y \mid xy \in L,\ x \in L_0\}.$$

Similarly, the right-quotient of L by a word $x \in \Sigma^*$ is the set

$$L/x = \{y \in \Sigma^* \mid yx \in L\},$$

and the right-quotient of L by a language $L_0 \subseteq \Sigma^*$ is

$$L/L_0 = \bigcup_{x \in L_0} L/x = \{y \mid yx \in L,\ x \in L_0\}.$$

It is clear by the above definition that

$$(L_1 \backslash L) \cup (L_2 \backslash L) = (L_1 \cup L_2) \backslash L$$

for any $L_1, L_2 \subseteq \Sigma^*$. This implies that $\{x, y\} \backslash L = x \backslash L \cup y \backslash L$. Similar equalities hold for right-quotient of languages.

For $L \subseteq \Sigma^*$, we define the following operations:

- $\min(L) = \{w \in L \mid$ there does not exist $x \in L$ such that x is a proper prefix of $w\}$.
- $\max(L) = \{w \in L \mid$ there does not exist $x \in L$ such that w is a propper prefix of $x\}$.

Theorem 4.5. *The family of regular languages is closed under (1) left-quotient by an arbitrary language, (2) right quotient by an arbitrary language, (3) min, and (4) max.*

Proof. Let $L \subseteq \Sigma^*$ be a regular language accepted by a DFA $A = (Q, \Sigma, \delta, s, F)$. We define, for each $q \in Q$, DFA $A_q = (Q, \Sigma, \delta, s, \{q\})$ and $A^{(q)} = (Q, \Sigma, \delta, q, F)$. For each of the four operations, we prove that the resulting language is regular by constructing a finite automaton to accept it. We leave the verifications of the constructions to the reader.

For (1), let $L_0 \subseteq \Sigma^*$ be an arbitrary language. Then $L_0 \backslash L$ is accepted by the NNFA $A_1 = (Q, \Sigma, \delta, S_1, F)$ where Q, δ, and F are the same as in A; and

$$S_1 = \{q \in Q \mid L(A_q) \cap L_0 \neq \emptyset\}$$

is the set of starting states of the NNFA.

For (2), we construct a DFA $A_2 = (Q, \Sigma, \delta, s, F_2)$ where Q, δ, and s are the same as in A; and $F_2 = \{q \in Q \mid L(A^{(q)}) \cap L_0 \neq \emptyset\}$.

For (3), we define $A_3 = (Q, \Sigma, \delta_3, s, F)$ where δ_3 is the same as δ except that all transitions from each final state are deleted.

For (4), we define $A_4 = (Q, \Sigma, \delta, s, F_4)$ where $F_4 = \{f \in F \mid \delta^*(f, x) \notin F$ for all $x \in \Sigma^+\}$. □

Let m and n be two natural numbers such that $m < n$. Then, for a language $L \subseteq \Sigma^*$, $\frac{m}{n}(L)$ is defined to be the language

$$\begin{aligned} \frac{m}{n}(L) \quad = \quad & \{w_1 \cdots w_m \mid w_1 \cdots w_m w_{m+1} \cdots w_n \in \\ & L, w_1, \ldots, w_n \in \Sigma^*, |w_1| = \cdots = |w_n|\}. \end{aligned}$$

Note that the above definition requires that the division of a word into n parts has to be exact. Then, the operations $\frac{m}{n}$ and $\frac{cm}{cn}$ are not equivalent for an integer $c > 1$. For example, let $L = \{\lambda, a, ba, aab, bbab\}$; then $\frac{1}{2}(L) = \{\lambda, b, bb\}$, but $\frac{2}{4}(L) = \{\lambda, bb\}$. We show that the family of regular languages is closed under the $\frac{m}{n}$ operation.

Theorem 4.6. *Let $L \subseteq \Sigma^*$ be a regular language and m, n be two natural numbers such that $m < n$. Then $\frac{m}{n}(L)$ is regular.*

Proof. Let L be accepted by a DFA $A = (Q, \Sigma, \delta, s, F)$. For each $q \in Q$, we construct a variant of an NFA $A(q)$ which reads m symbols at each transition. Such a variant can clearly be transformed into an equivalent standard NFA. More specifically, $A(q) = (Q', \Sigma, \delta', s_q, F_q)$ where $Q' = Q \times Q$; $\delta : Q' \times \Sigma^m \to 2^{Q'}$ is defined, for $a_1, \ldots, a_m \in \Sigma$,

$$\delta'((p_1, p_2), a_1 \cdots a_m) = \{(p_1', p_2') \mid \delta^*(p_1, a_1 \cdots a_m) = p_1' \text{ and}$$

$$\text{there exists } x \in \Sigma^{n-m} \text{ such that } \delta^*(p_2, x) = p_2'\};$$

$s_q = (s, q)$; and $F_q = \{(q, f) \mid f \in F\}$.

Intuitively, $A(q)$ operates on two tracks, starting with the states s and q, respectively. A word u with $|u| = cm$, for some nonnegative integer c, is accepted by $A(q)$ if $A(q)$, working on the first track, can reach q by reading u and, simultaneously working on the second track, can reach a final state of A from the state q by reading a "phantom" input of length $c(n - m)$.

It is easy to see that $\frac{m}{n}(L) = \cup_{q \in Q} L(A(q))$. We omit the details of the proof. □

4.3 Derivatives and the Myhill-Nerode theorem

The notion of derivatives was introduced in [99, 100, 43] (under different names) and was first applied to regular expressions by Brzozowski in [16].

We define derivatives using the left-quotient operation. Let $L \subseteq \Sigma^*$ and $x \in \Sigma^*$. The *derivative* of L with respect to x, denoted $D_x L$, is

$$D_x L = x \backslash L = \{y \mid xy \in L\}.$$

For $L \subseteq \Sigma^*$, we define a relation $\equiv_L \subseteq \Sigma^* \times \Sigma^*$ by

$$x \equiv_L y \text{ iff } D_x L = D_y L$$

for each pair $x, y \in \Sigma^*$. Clearly, $\equiv_L$ is an equivalence relation. It partitions Σ^* into equivalence classes. The number of equivalence classes of $\equiv_L$ is called the *index* of $\equiv_L$. We denote the equivalence class that contains x by $[x]_{\equiv_L}$, i.e.,

$$[x]_{\equiv_L} = \{y \in \Sigma^* \mid y \equiv_L x\}.$$

Clearly, $x \equiv_L y$ iff $[x]_L = [y]_L$. We simply write $[x]$ instead of $[x]_{\equiv_L}$ if there is no confusion.

A relation $R \subseteq \Sigma^* \times \Sigma^*$ is said to be *right-invariant* with respect to catenation if $x \, R \, y$ implies $xz \, R \, yz$, for any $z \in \Sigma^*$. It is clear that the relation $\equiv_L$ is right-invariant.

Lemma 4.5. *Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA and $L = L(A)$. For each $q \in Q$, let $A_q = (Q, \Sigma, \delta, s, \{q\})$. Then, for all $x, y \in \Sigma^*$ and $q \in Q$, $x, y \in L(A_q)$ implies $x \equiv_L y$.*

Proof. Let $x, y \in L(A_q)$. Define $A^{(q)} = (Q, \Sigma, \delta, q, F)$. Then, clearly, $D_x L = L(A^{(q)}) = D_y L$. Thus, $x \equiv_L y$ by the definition of $\equiv_L$. □

The following is a variant of the theorem which is called the Myhill-Nerode Theorem in [57]. The result was originally given by Myhill [90] and Nerode [91]. A similar result on regular expressions was obtained by Brzozowski [16].

Theorem 4.7. *A language $L \subseteq \Sigma^*$ is regular iff $\equiv_L$ has a finite index.*

Proof. *Only if:* Let L be accepted by a complete DFA $A = (Q, \Sigma, \delta, s, F)$. As in Lemma 4.5, we define $A_q = (Q, \Sigma, \delta, s, \{q\})$ for each $q \in Q$. Since A is a complete DFA, we have

$$\bigcup_{q \in Q} L(A_q) = \Sigma^*.$$

Thus, $\pi_A = \{L(A_q) \mid q \in Q\}$ is a partition of Σ^*. By Lemma 4.5, for each $q \in Q$, $x, y \in L(A_q)$ implies $x \equiv_L y$, i.e., $L(A_q) \subseteq [x]$ for some $x \in \Sigma^*$. This means that π_A refines the partition induced by $\equiv_L$. Since π_A is a finite partition, the number of the equivalence classes of $\equiv_L$ is finite.

If: We construct a DFA $A' = (Q', \Sigma, \delta', s', F')$ where the elements of Q' are the equivalence classes of $\equiv_L$, i.e., $Q' = \{[x] \mid x \in \Sigma^*\}$; δ' is defined by $\delta'([x], a) = [xa]$, for all $[x] \in Q$ and $a \in \Sigma$; $s' = [\lambda]$; and $F' = \{[x] \mid x \in L\}$. Note that δ' is well-defined because $\equiv_L$ is right-invariant. It is easy to verify that $\delta'(s', x) = [x]$ for each $x \in \Sigma^*$ (by induction on the length of x). Then $x \in L$ iff $\delta'(s', x) = [x] \in F'$. Therefore, $L(A') = L$. □

Theorem 4.8. *Let L be a regular language. The minimal number of states of a complete DFA that accepts L is equal to the index of $\equiv_L$.*

Proof. Let the index of $\equiv_L$ be k. In the proof of Theorem 4.7, it is shown that there is a k-state complete DFA that accepts L. We now prove that k is minimum. Suppose that L is accepted by a complete DFA $A = (Q, \Sigma, \delta, s, F)$ of k' states where $k' < k$. Then, for some $q \in Q$, $L(A_q)$ contains words from two distinct equivalence classes of $\equiv_L$, i.e., $x \not\equiv_L y$ for some $x, y \in L(A_q)$. This contradicts Lemma 4.5. □

Corollary 4.1. *Let $A = (Q, \Sigma, \delta, s, F)$ be a complete DFA and $L = L(A)$. A is a minimum-state complete DFA accepting L iff, for each $q \in Q$, $L(A_q) = [x]$ for some $x \in \Sigma^*$.*

Proof. The *if* part follows immediately from Theorem 4.8. For the converse implication, assume that A is a minimum-state complete DFA. By Lemma 4.5, the partition of Σ^* into the languages $L(A_q)$, $q \in Q$, is a refinement of $\equiv_L$. By Theorem 4.8, $|Q|$ equals to the index of $\equiv_L$. Hence, each language $L(A_q)$, $q \in Q$, has to coincide with some class of the relation $\equiv_L$. □

From the above arguments, one can observe that for minimizing a given DFA A that accepts L, we need just to merge into one state all the states q such that the corresponding languages $L(A_q)$ are in the same equivalence class of $\equiv_L$. Transitions from states to states are also merged accordingly. This can be done because of the right-invariant property of $\equiv_L$.

More formally, for a DFA $A = (Q, \Sigma, \delta, s, F)$, we define an equivalence relation $\approx_A$ on Q as follows:

$$p \approx_A q \text{ iff } L(A^{(p)}) = L(A^{(q)})$$

for $p, q \in Q$. Note that $\approx_A$ is right-invariant in the sense that if $p \approx q$ then $\delta^*(p, x) \approx_A \delta^*(q, x)$, for any given $x \in \Sigma^*$. It is clear that each equivalence class of $\approx_A$ corresponds exactly to an equivalence class of $\equiv_{L(A)}$. Then we can present the following DFA minimization scheme:

(1) Partition Q into equivalence classes of $\approx_A$:

$$\Pi = \{[q] \mid q \in Q\}.$$

(2) Construct $A' = (Q', \Sigma, \delta', s', F')$ where $Q' = \Pi$, $s' = [s']$, $F' = \{[f] \mid f \in F\}$, and $\delta'([p], a) = [q]$ if $\delta(p, a) = q$, for all $p, q \in Q$ and $a \in \Sigma$.

Note that the right-invariant property of $\approx_A$ guarantees that δ' is well defined.

The major part of the scheme is at the step (1), i.e., finding the partition of Q. A straightforward algorithm for step (1) is that we check whether $p \approx_A q$ by simply testing whether $L(A^{(p)}) = L(A^{(q)})$. However, the complexity of this algorithm is too high ($\Omega(n^4)$ where n is the number of states of A).

Many partition algorithms have been developed, see, e.g., [56, 49, 57, 11, 12]. The algorithm by Hopcroft [56], which was redescribed later by Gries [49] in a more understandable way, is so far the most efficient algorithm. A rather complete list of DFA minimization algorithms can be found in [122].

An interesting observation is that, for a given DFA, if we construct an NFA that is the reversal of the given DFA and then transform it to a DFA by the standard subset construction technique (constructing only those states that are reachable from the new starting state), then the resulting DFA is a minimum-state DFA [15, 67, 81, 12]. We state this more formally in the following. First, we define two operations γ and τ on automata. For a DFA $A = (Q, \Sigma, \delta, s, F)$, $\gamma(A)$ is the NNFA $A^R = (Q, \Sigma, \delta^R, F, \{s\})$ where $\delta^R : Q \to 2^Q$ is defined by $\delta^R(p, a) = \{q \mid \delta(q, a) = p\}$; and for an NNFA $M = (Q, \Sigma, \eta, S, F)$, $\tau(M)$ is the DFA $M' = (Q', \Sigma, \eta', s', F')$ where $s' = S$; η' and F' are defined by the standard subset construction technique; and $Q' \subseteq 2^Q$ consists of only those subsets of Q that are reachable from s'.

Theorem 4.9. *Let $A = (Q, \Sigma, \delta, s, F)$ be a DFA with the property that all states in Q are reachable from s. Then $L(\tau(\gamma(A))) = L^R$ and $\tau(\gamma(A))$ is a minimum-state DFA.*

Proof. Let $\gamma(A)$ be the NNFA $A^R = (Q, \Sigma, \delta^R, F, \{s\})$ and $\tau(A^R)$ be the DFA $A' = (Q', \Sigma, \delta', s', F')$ as defined above. Obviously, $L(A') = L^R$. To prove that A' is minimum, it suffices to show that, for any $p', q' \in Q'$, $p' \approx_{A'} q'$ implies $p' = q'$. Notice that p' and q' are both subsets of Q. If $p' \approx_{A'} q'$, then $L(A'^{(p')}) = L(A'^{(q')})$. Let $r \in p'$. Since $\delta^*(s, x) = r$ for some $x \in \Sigma^*$, we have $s \in (\delta^R)^*(r, x)$ and, thus, $x \in L(A^{(p')})$. This implies that $x \in L(A^{(q')})$ and, thus, there exist $t \in q'$ such that $\delta^*(s, x) = t$. Since δ is a deterministic transition function, $r = t$, i.e., $r \in q'$. So, $p' \subseteq q'$. Similarly, we can prove that $q' \subseteq p'$. Therefore, $p' = q'$. □

From the above idea, a conceptually simple algorithm for DFA minimization can be obtained as follows. Given a DFA A, we compute $A' = \tau(\gamma(\tau(\gamma(A))))$. Then A' is a minimum-state DFA which is equivalent to A. The algorithm is descriptively very simple. However, the time and space complexities of the algorithm are very high; they are both of the order of 2^n in the worst case, where n is the number of states of A. This algorithm was originally given by Brzozowski in [15]. Descriptions of the algorithm can also be found in [81, 67, 12, 118, 122]. Watson wrote an interesting paragraph on the origin of the algorithm in [122] (on pages 195–196).

Theorem 4.8 gives a tight lower bound on the number of states of a DFA. Can we get similar results for AFA and NFA? The following result for AFA follows immediately from Theorem 2.14 and Theorem 4.8.

Theorem 4.10. *Let L be a regular language and $k > 1$ be the minimum number of states of a DFA that accepts L^R, i.e., the reversal of L. Then the minimum number of states of an s-AFA accepting L is $\lceil \log k \rceil + 1$.* □

Note that there can be many different minimum-state AFA accepting a given language and they are not necessarily identical or equivalent under a renaming of the states.

NFA are a special case of AFA. Any lower bound on the number of states of an AFA would also be a lower bound on the number of states of an NFA.

Corollary 4.2. *Let L be a regular language and $k > 1$ be the minimum number of states of a DFA that accepts L^R. Then the minimum number of states of an NFA accepting L is greater than or equal to $\lceil \log k \rceil + 1$.*

The above lower bound is reached for some languages, e.g., the languages accepted by the automata shown in Figure 18.

Also by Lemma 2.2, we have the following:

Theorem 4.11. *Let L be a regular language. Let k and k' be the numbers of states of the minimal DFA accepting L and L^R, respectively. Then the number of states of any NFA accepting L is greater than or equal to* $\max(\lceil \log k \rceil, \lceil \log k' \rceil)$. □

Observe that minimum-state NNFA that accept L and L^R, respectively, have exactly the same number of states. A minimum-state NFA requires at

most one more state than a minimum-state NNFA equivalent to it. So, this gives another proof for the above lower bound.

5. Complexity issues

In the previous sections, we studied various representations, operations, and properties of regular languages. When we were considering the operations on regular languages, we were generally satisfied with knowing what can be done and what cannot be done, but did not measure the complexity of the operations. In this section, we consider two kinds of measurements: (1) state complexity and (2) time and space complexity. One possibility would have been to discuss these complexity issues together with the various operations in the previous sections. Since this topic has usually not been at all included in earlier surveys, we feel that devoting a separate section for the complexity issues is justified.

5.1 State complexity issues

By the *state complexity* of a regular language, we mean the minimal number of states of a DFA representing the language. By the state complexity of an operation on regular languages we mean a function that associates the sizes of the DFA representing the operands of the operation to the minimal number of states of the DFA representing the resulting language. Note that in this section, by a DFA we always mean a complete DFA.

State complexity is a natural measurement of operations on regular languages. It also gives a lower bound for the time and space complexity of those operations. State complexity is of central importance especially for applications using implementations of finite automata. However, questions of state complexity have rarely been the object of a systematic investigation. Examples of early studies concentrated on this topic are [104, 105] by Salomaa and [89] by Moore. Some recent results can be found in [6, 102, 101, 124, 111]. Most of the results presented in this section are from [111].

By an n-state DFA language, we mean a regular language that is accepted by an n-state DFA. Here, we consider only the worst-case state complexity. For example, for an arbitrary m-state DFA language and an arbitrary n-state DFA language, the state complexity of the catenation of the two languages is $m2^n - 2^{n-1}$. This means that

(1) there exist an m-state DFA language and an n-state DFA language such that any DFA accepting the catenation of the two languages needs at least $m2^n - 2^{n-1}$ states; and
(2) the catenation of an m-state DFA language and an n-state DFA language can always be accepted by a DFA using $m2^n - 2^{n-1}$ states or less.

So, it is a tight lower bound and upper bound. In the following, we first summarize the state complexity of various operations on regular languages. Then we give some details for certain operations. For each operation we consider, we give an exact function rather than the order of the function.

Let Σ be an alphabet, L_1 and L_2 be an m-state DFA language and an n-state DFA language over Σ, respectively. A list of operations on L_1 and L_2 and their state complexity are the following:

- L_1L_2 : $m2^n - 2^{n-1}$;
- $(L_2)^*$: $2^{n-1} + 2^{n-2}$;
- $L_1 \cap L_2$: mn;
- $L_1 \cup L_2$: mn;
- $L \backslash L_2$: $2^n - 1$, where L is an arbitrary language;
- L_2/L : n, where L is an arbitrary language;
- L_2^R : 2^n.

The state complexity of some of the above operations is much lower if we consider only the case when $|\Sigma| = 1$. For unary alphabets, we have

- $(L_2)^*$: $(n-1)^2 + 1$;
- L_1L_2 : mn (if $(m,n) = 1$).

5.1.1 Catenation

We first show that for any $m \geq 1$ and $n > 1$ there exist an m-state DFA A and an n-state DFA B such that any DFA accepting $L(A)L(B)$ needs at least $m2^n - 2^{n-1}$ states. Then we show that for any pair of m-state DFA A and n-state DFA B defined on the same alphabet Σ, there exists a DFA with at most $m2^n - 2^{n-1}$ states that accepts $L(A)L(B)$.

Theorem 5.1. *For any integers $m \geq 1$ and $n \geq 2$, there exist a DFA A of m states and a DFA B of n states such that any DFA accepting $L(A)L(B)$ needs at least $m2^n - 2^{n-1}$ states.*

Proof. We first consider the cases when $m = 1$ and $n \geq 2$. Let $\Sigma = \{a, b\}$. Since $m = 1$, A is a one-state DFA accepting Σ^*. Choose $B = (P, \Sigma, \delta_B, p_0, F_B)$ (Figure 14) where $P = \{p_0, \ldots, p_{n-1}\}$, $F_B = \{ p_{n-1} \}$, and $\delta_B(p_0, a) = p_0$, $\delta_B(p_0, b) = p_1$, $\delta_B(p_i, a) = p_{i+1}$, $1 \leq i \leq n-2$, $\delta_B(p_{n-1}, a) = p_1$, $\delta_B(p_i, b) = p_i$, $1 \leq i \leq n-1$.

It is easy to see that

$$L(A)L(B) = \{w \in \Sigma^* \mid w = ubv, |v|_a \equiv n-2 \ mod \ (n-1)\}.$$

Let $(i_1, \ldots, i_{n-1}) \in \{0,1\}^{n-1}$ and denote

$$w(i_1, \ldots, i_{n-1}) = b^{i_1}ab^{i_2} \ldots ab^{i_{n-1}}.$$

Then, for every $j \in \{0, \ldots, n-2\}$, $w(i_1, \ldots, i_{n-1})a^j \in L(A)L(B)$ iff $i_{j+1} = 1$. Thus a DFA accepting $L(A)L(B)$ needs at least 2^{n-1} states.

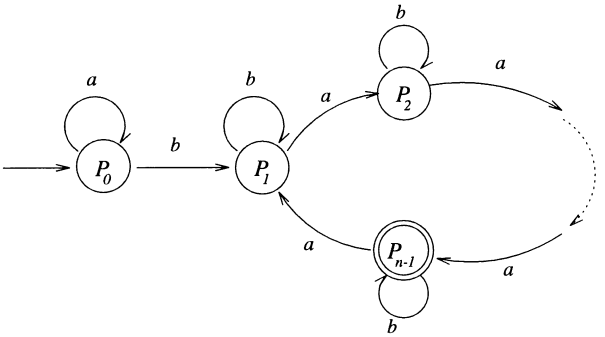


Fig. 14. DFA B

Now we consider the cases when $m \geq 2$ and $n \geq 2$.

Let $\Sigma = \{a, b, c\}$. Define $A = (Q, \Sigma, \delta_A, q_0, F_A)$ where $Q = \{q_0, \ldots, q_{m-1}\}$; $F_A = \{ q_{m-1} \}$; for each i, $0 \leq i \leq m-1$,

$$\delta_A(q_i, X) = \begin{cases} q_j,\ j = (i+1) \bmod m, & \text{if } X = a, \\ q_0, & \text{if } X = b, \\ q_i, & \text{if } X = c. \end{cases}$$

Define $B = (P, \Sigma, \delta_B, p_0, F_B)$ where $P = \{p_0, \ldots, p_{n-1}\}$; $F_B = \{ p_{n-1} \}$; and for each i, $0 \leq i \leq n-1$,

$$\delta_B(p_i, X) = \begin{cases} p_j,\ j = (i+1) \bmod n, & \text{if } X = b, \\ p_i, & \text{if } X = a, \\ p_1, & \text{if } X = c. \end{cases}$$

The DFA A and B are shown in Figure 15 and Figure 16, respectively. The reader can verify that

$$L(A) = \{ xy \mid x \in (\Sigma^*\{b\})^*,\ y \in \{a, c\}^* \ \&\ |y|_a = m - 1 \bmod m \},$$

and

$$L(B) \cap \{a, b\}^* = \{ x \in \{a, b\}^* \mid |x|_b = n - 1 \bmod n \}.$$

Now we consider the catenation of $L(A)$ and $L(B)$, i.e., $L(A)L(B)$.

Fact 5.1. For $m > 1$, $L(A) \cap \Sigma^*\{b\} = \emptyset$. □

For each $x \in \{a, b\}^*$, we define

$$S(x) = \{ i \mid x = uv \text{ such that } u \in L(A), \text{ and } i = |v|_b \bmod n \}.$$

Consider $x, y \in \{a, b\}^*$ such that $S(x) \neq S(y)$. Let $k \in S(x) - S(y)$ (or $S(y) - S(x)$). Then it is clear that $xb^{n-1-k} \in L(A)L(B)$ but $yb^{n-1-k} \notin L(A)L(B)$. So, x and y are in different equivalence classes of $\equiv_{L(A)L(B)}$ where $\equiv_L$ is defined in Section 4.3.

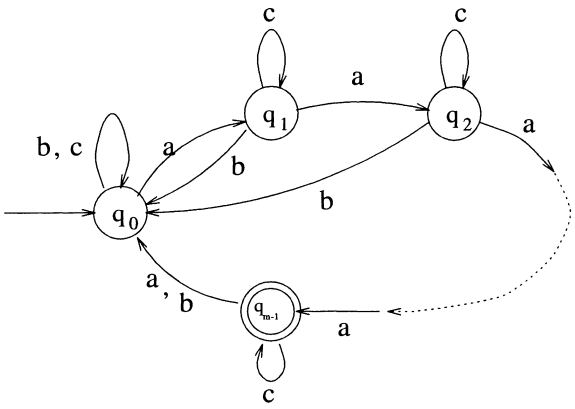


Fig. 15. DFA A

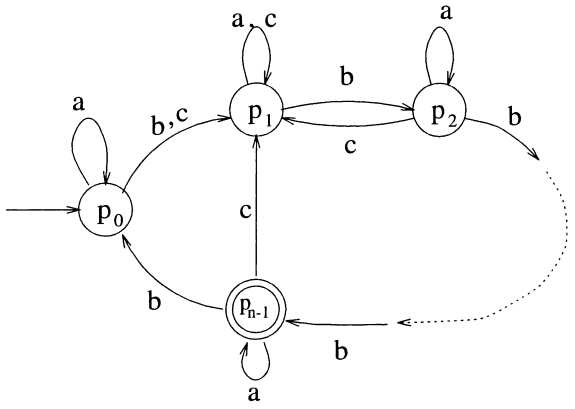


Fig. 16. DFA B

For each $x \in \{a,b\}^*$, define $T(x) = max\{|z| \mid x = yz \;\&\; z \in a^*\}$. Consider $u, v \in \{a,b\}^*$ such that $S(u) = S(v)$ and $T(u) > T(v) \bmod m$. Let $i = T(u) \bmod m$ and $w = ca^{m-1-i}b^{n-1}$. Then clearly $uw \in L(A)L(B)$ but $vw \notin L(A)L(B)$. Notice that there does not exist a word $w \in \Sigma^*$ such that $0 \notin S(w)$ and $T(w) = m-1$, since the fact that $T(w) = m-1$ guarantees that $0 \in S(w)$.

For each subset $s = \{i_1, \ldots, i_t\}$ of $\{0, \ldots, n-1\}$, where $i_1 > \ldots > i_t$, and each integer $j \in \{0, \ldots, m-1\}$ except the case when both $0 \notin s$ and $j = m-1$ are true, there exists a word

$$x = a^{m-1}b^{i_1-i_2}a^{m-1}b^{i_2-i_3}a^{m-1}\ldots a^{m-1}b^{i_t+n}a^j$$

such that $S(x) = s$ and $T(x) = j$. Thus, the relation $\equiv_{L(A)L(B)}$ has at least $m2^n - 2^{n-1}$ distinct equivalence classes. □

The next theorem gives an upper bound which coincides exactly with the above lower bound result. Therefore, the bound is tight.

Theorem 5.2. *Let A and B be two complete DFA defined on the same alphabet, where A has m states and B has n states, and let A have k final states, $0 < k < m$. Then there exists a $(m2^n - k2^{n-1})$-state DFA which accepts $L(A)L(B)$.*

Proof. Let $A = (Q, \Sigma, \delta_A, q_0, F_A)$ and $B = (P, \Sigma, \delta_B, p_0, F_B)$. Construct $C = (R, \Sigma, \delta_C, r_0, F_C)$ such that

$R = Q \times 2^P - F_A \times 2^{P-\{p_0\}}$ where 2^X denotes the power set of X;
$r_0 =< q_0, \emptyset >$ if $q_0 \notin F_A$, $r_0 =< q_0, \{p_0\} >$ otherwise;
$F_C = \{< q, T > \in R \mid T \cap F_B \neq \emptyset\}$;
$\delta_C(< q, T >, a) =< q', T' >$, for $a \in \Sigma$, where $q' = \delta_A(q, a)$ and $T' = \delta_B(T, a) \cup \{p_0\}$ if $q' \in F_A$, $T' = \delta_B(T, a)$ otherwise.

Intuitively, R is a set of pairs such that the first component of each pair is a state in Q and the second component is a subset of P. R does not contain those pairs whose first component is a final state of A and whose second component does not contain the initial state of B. Clearly, C has $m2^n - k2^{n-1}$ states. The reader can easily verify that $L(C) = L(A)L(B)$. □

We still need to consider the cases when $m \geq 1$ and $n = 1$. We have the following result.

Theorem 5.3. *The number of states that is sufficient and necessary in the worst case for a DFA to accept the catenation of an m-state DFA language and a 1-state DFA language is m.*

Proof. Let Σ be an alphabet and $a \in \Sigma$. Clearly, for any integer $m > 0$, the language $L = \{w \in \Sigma^* \mid |w|_a \equiv m - 1 \bmod m\}$ is accepted by an m-state DFA. Note that Σ^* is accepted by a one-state DFA. It is easy to see that any DFA accepting $L\ \Sigma^* = \{w \in \Sigma^* \mid \#_a(w) \geq m - 1\}$ needs at least m states. So, we have proved the necessary condition.

Let A and B be an m-state DFA and a 1-state DFA, respectively. Since B is a complete DFA, $L(B)$ is either $\emptyset$ or Σ^*. We need to consider only the case $L(B) = \Sigma^*$. Let $A = (Q, \Sigma, \delta_A, q_0, F_A)$. Define $C = (Q, \Sigma, \delta_C, q_0, F_A)$ where, for any $X \in \Sigma$ and $q \in Q$,

$$\delta_C(q, X) = \begin{cases} \delta_A(q, X), & \text{if } q \notin F_A, \\ q, & \text{if } q \in F_A. \end{cases}$$

The automaton C is exactly as A except that final states are made to be sink-states: when the computation has reached some final state q, it remains there. Now it is clear that $L(C) = L(A)\Sigma^*$. □

5.1.2 Star operation (Kleene closure)

Here we prove that the state complexity of the star operation of an n-state DFA language is $2^{n-1} + 2^{n-2}$.

Theorem 5.4. *For any n-state DFA $A = (Q, \Sigma, \delta, q_0, F)$ such that $|F - \{q_0\}| = k \geq 1$ and $n > 1$, there exists a DFA of at most $2^{n-1} + 2^{n-k-1}$ states that accepts $(L(A))^*$.*

Proof. Let $A = (Q, \Sigma, \delta, q_0, F)$ and $L = L(A)$. Denote $F - \{q_0\}$ by F_0. Then $|F_0| = k \geq 1$. We construct a DFA $A' = (Q', \Sigma, \delta', q_0', F')$ where

$q_0' \notin Q$ is a new start state;
$Q' = \{q_0'\} \cup \{P | P \subseteq (Q - F_0) \& P \neq \emptyset\} \cup \{R | R \subseteq Q \& q_0 \in R \& R \cap F_0 \neq \emptyset\}$;
$\delta'(q_0', a) = \{\delta(q_0, a)\}$, for any $a \in \Sigma$, and

$$\delta'(R, a) = \begin{cases} \delta(R, a) & \text{if } \delta(R, a) \cap F_0 = \emptyset, \\ \delta(R, a) \cup \{q_0\} & \text{otherwise,} \end{cases}$$

for $R \subseteq Q$ and $a \in \Sigma$;
$F' = \{q_0'\} \cup \{R \mid R \subseteq Q \ \& \ R \cap F \neq \emptyset\}$.

The reader can verify that $L(A') = L^*$. Now we consider the number of states in Q'. Notice that in the second term of the union for Q', there are $2^{n-k} - 1$ states. In the third term, there are $(2^k - 1)2^{n-k-1}$ states. So, $|Q'| = 2^{n-1} + 2^{n-k-1}$. □

Note that if q_0 is the only final state of A, i.e., $k = 0$, then $(L(A))^* = L(A)$. So, the worst-case state complexity of the star operation occurs when $k = 1$.

Corollary 5.1. *For any n-state DFA A, $n > 1$, there exists a DFA A' of at most $2^{n-1} + 2^{n-2}$ states such that $L(A') = (L(A))^*$.* □

Theorem 5.5. *For any integer $n \geq 2$, there exists a DFA A of n states such that any DFA accepting $(L(A))^*$ needs at least $2^{n-1} + 2^{n-2}$ states.*

Proof. For $n = 2$, it is clear that $L = \{w \in \{a, b\}^* \mid |w|_a \text{ is odd}\}$ is accepted by a two-state DFA, and $L^* = \{\varepsilon\} \cup \{w \in \{a, b\}^* \mid |w|_a \geq 1\}$ cannot be accepted by a DFA with less than 3 states.

For $n > 2$, we give the following construction: $A_n = (Q_n, \Sigma, \delta_n, 0, \{n-1\})$ where $Q_n = \{0, \ldots, n-1\}$; $\Sigma = \{a, b\}$; $\delta(i, a) = (i + 1) \bmod n$ for each $0 \leq i < n$, $\delta(i, b) = (i + 1) \bmod n$ for each $1 \leq i < n$ and $\delta(0, b) = 0$. A_n is shown in Figure 17.

We construct the DFA $A_n' = (Q_n', \Sigma, \delta_n', q_0', F_n')$ from A_n exactly as described in the proof of the previous theorem. We need to show that (I) every state is reachable from the start state and (II) each state defines a distinct equivalence class of $\equiv_{L(A_n)^*}$.

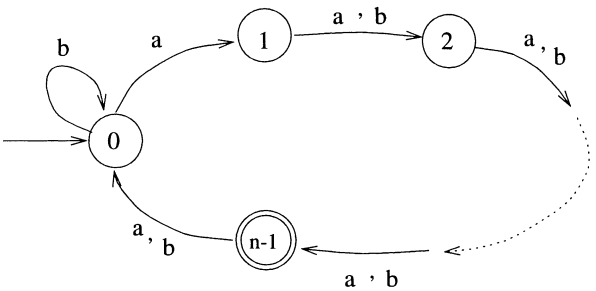


Fig. 17. An n-state DFA A_n: The language $(L(A_n))^*$ requires $2^{n-1} + 2^{n-2}$ states

We prove (I) by induction on the size of the state set. (Note that each state is a subset of Q_n except q'_0.)

Consider all q such that $q \in Q'$ and $|q| = 1$. We have $\{0\} = \delta'_n(q'_0, b)$ and $\{i\} = \delta'_n(i-1, a)$ for each $0 < i < n-1$.

Assume that all q such that $|q| < k$ are reachable. Consider q where $|q| = k$. Let $q = \{i_1, i_2, \ldots, i_k\}$ such that $0 \leq i_1 < i_2 \ldots < i_k < n-1$ if $n-1 \notin q$, $i_1 = n-1$ and $0 = i_2 < \ldots i_k < n-1$ otherwise. There are four cases:

(i) $i_1 = n-1$ and $i_2 = 0$. Then $q = \delta'_n(\{n-2, i_3 - 1, \ldots, i_k - 1\}, a)$ where the latter state contains $k-1$ states.
(ii) $i_1 = 0$ and $i_2 = 1$. Then $q = \delta'_n(q', a)$ where $q' = \{n-1, 0, i_3-1, \ldots i_k-1\}$ which is considered in case (i).
(iii) $i_1 = 0$ and $i_2 = 1+t$ for $t > 0$. Then $q = \delta'_n(q', b^t)$ where $q' = \{0, 1, i_3 - t, \ldots, i_k - t\}$. The latter state is considered in case (ii).
(iv) $i_1 = t > 0$. Then $q = \delta'_n(q', a^t)$ where $q' = \{0, i_2 - t, \ldots, i_k - t\}$ is considered in either case (ii) or case (iii).

To prove (II), let $i \in p - q$ for some $p, q \in Q'_n$ and $p \neq q$. Then $\delta'_n(p, a^{n-1-i}) \in F'_n$ but $\delta'_n(q, a^{n-1-i}) \notin F'_n$. □

Note that a DFA accepting the star of a 1-state DFA language may need up to two states. For example, $\emptyset$ is accepted by a 1-state DFA and any complete DFA accepting $\emptyset^* = \{\varepsilon\}$ has at least two states.

It is clear that any DFA accepting the reversal of an n-state DFA language does not need more than 2^n states. But can this upper bound be reached? A result on alternating finite automata ([23], Theorem 5.3) gives a positive answer to the above question if n is of the form 2^k for some integer $k \geq 0$. Leiss has solved this problem in [73] for all $n > 0$. A modification of Leiss's solution is shown in Figure 18. If we reverse all the transitions of this automaton, we will get a good example for showing that, in the worst case, a DFA equivalent to an n-state NFA may need exactly 2^n states.

5.1.3 An open problem

For the state complexity of catenation, we have proved the general result $(m2^n - 2^{n-1})$ using automata with a three-letter input alphabet. We have also given the complexity for the one-letter alphabet case. We do not know whether the result obtained for the three-letter alphabet still holds if the size of the alphabet is two.

5.2 Time and space complexity issues

Almost all problems of interest are decidable for regular languages, i.e., there exist algorithms to solve them. However, for the purpose of implementation, it is necessary to know how hard these problems are and what the time and space complexities of the algorithms are. In the following, we list some basic problems, mostly decision problems, for regular languages together with their

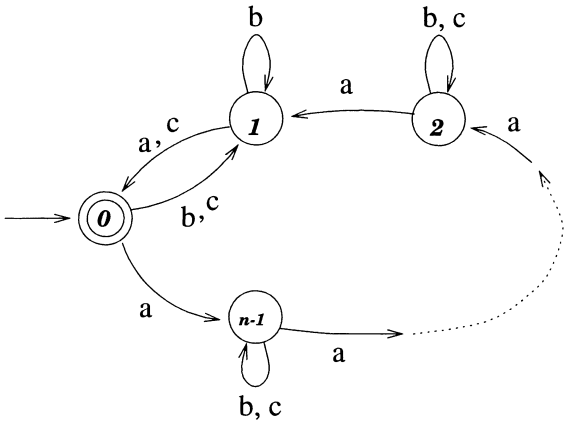


Fig. 18. An n-state DFA such that $L(A)^R$ requires 2^n states

complexity. We give a brief explanation and references for each problem. The reader may refer to [46, 57, 4] for terminology in complexity theory.

One may observe that many of the following problems are NP-complete or even PSPACE-complete, which are categorized as computationally intractable in complexity theory. However, finite automata and regular expressions used in many applications are fairly small in size. In such cases, even exponential algorithms can be practically feasible.

(1) **DFA Membership Problem:**
Given an arbitrary DFA A with the input alphabet Σ and an arbitrary word $x \in \Sigma^*$, is $x \in L(A)$?
Complexity: *DLOGSPACE-complete* [64].

(2) **NFA Membership Problem:**
Given an arbitrary NFA A with the input alphabet Σ and an arbitrary word $x \in \Sigma^*$, is $x \in L(A)$?
Complexity: *NLOGSPACE-complete* [66].

(3) **AFA Membership Problem:**
Given an arbitrary AFA A with the input alphabet Σ and a word $x \in \Sigma^*$, is $x \in L(A)$?
Complexity: *P-complete* [64].

(4) **Regular Expression Membership Problem:**
Given a regular expression e over Σ and a word $x \in \Sigma^*$, is $x \in L(e)$?
Complexity: *NLOGSPACE-complete* [64].

(5) **DFA Emptiness Problem:**
Given an arbitrary DFA A, is $L(A) = \emptyset$?
Complexity: *NLOGSPACE-complete* [66].

(6) **NFA Emptiness Problem:**
Given an arbitrary NFA A, is $L(A) = \emptyset$?
Complexity: *NLOGSPACE-complete* [66].

(7) **AFA Emptiness Problem:**
Given an arbitrary AFA A, is $L(A) = \emptyset$?
Complexity: *PSPACE-complete* [64].
(8) **DFA Equivalence Problem:**
Given two arbitrary DFA A_1 and A_2, is $L(A_1) = L(A_2)$?
Complexity: *NLOGSPACE-complete* [26].
(9) **NFA Equivalence Problem:**
Given two arbitrary NFA A_1 and A_2, is $L(A_1) = L(A_2)$?
Complexity: *PSPACE-complete* [46].
(10) **AFA Equivalence Problem:**
Given two arbitrary AFA A_1 and A_2, is $L(A_1) = L(A_2)$?
Complexity: *PSPACE-complete* [64].
(11) **Regular Expression Equivalence Problem:**
Given two regular expressions e_1 and e_2, is $L(e_1) = L(e_2)$?
Complexity: *PSPACE-complete* [59]. (Note that if one of the regular expressions denotes a language of polynomial density, then the complexity is *NP-complete.*)

The following problems can also be converted into decision problems. However, we prefer to keep them in their natural form:

(i) **DFA Minimization Problem:**
Given a DFA with n states, convert it to an equivalent minimum-state DFA.
Complexity: $O(n \log n)$ [56].
(ii) **NFA Minimization Problem:**
Given an NFA, convert it to an equivalent minimum-state NFA.
Complexity: *PSPACE-complete* [59, 119].
(iii) **DFA to Minimal NFA Problem:**
Given a DFA, convert it to an equivalent minimum-state NFA.
Complexity: *PSPACE-complete* [65].

The following problems remain open:

(a) Is membership for regular expressions over a one-letter alphabet *NLOGSPACE-hard*?
(b) Is membership for extended regular expressions *P-hard*?

Acknowledgements I wish to express my deep gratitute to Kai Salomaa for his significant contributions to this chapter. He has read the chapter many times and made numerous suggestions. I consider him truly the second author of the chapter. However, he has been insisting not to put his name as a co-author. I wish to thank J. Brzozowski for his great help in tracking the status of the six open problems he raised in 1979. Special thanks due K. Culik II, A. Salomaa, J. Shallit, A. Szilard, and R. Webber for their careful reading and valuable suggestions. I wish to express my gratitute to G. Rozenberg and A.

Salomaa for their great idea and efforts in organizing the handbook. Finally, I wish to thank the Natural Sciences and Engineering Research Council of Canada for their support.

References

1. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*, Vol. 1, Prentice-Hall, Englewood Cliffs, N.J., (1972).
2. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, Reading, (1986).
3. J. C. M. Baeten and W. P. Weijland, *Process Algebra*, Cambridge University Press, Cambridge, (1990).
4. J. L. Balcázar, J. Díaz, and J. Gabarró, *Structured Complexity* I, II, EATCS Monagraphs on Theoretical Computer Science, vol. 11 and 22, Springer-Verlag, Berlin 1988 and 1990.
5. Y. Bar-Hillel, M. Perles, and E. Shamir, "On formal properties of simple phrase structure grammars", *Z. Phonetik. Sprachwiss. Kommunikationsforsch.* 14 (1961) 143–172.
6. J.-C. Birget, "State-Complexity of Finite-State Devices, State Compressibility and Incompressibility", *Mathematical Systems Theory* 26 (1993) 237–269.
7. G. Berry and R. Sethi, "From Regular Expressions to Deterministic Automata", *Theoretical Computer Science* 48 (1986) 117–126.
8. J. Berstel, *Transductions and Context-Free Languages*, Teubner, Stuttgart, (1979).
9. J. Berstel and M. Morcrette, "Compact representation of patterns by finite automata", *Pixim 89: L'Image Numérique à Paris*, André Gagalowicz, ed., Hermes, Paris, (1989), pp.387–395.
10. J. Berstel and C. Reutenauer, *Rational Series and Their Languages*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin (1988).
11. W. Brauer, *Automatentheorie*, Teubner, Stuttgart, (1984).
12. W. Brauer, "On Minimizing Finite Automata", *EATCS Bulletin* 35 (1988) 113–116.
13. A. Brüggemann-Klein, "Regular expressions into finite automata", *Theoretical Computer Science* 120 (1993) 197–213.
14. A. Brüggemann-Klein and D. Wood, "Deterministic Regular Languages", Proceedings of STACS'92, *Lecture Notes in Computer Science* 577, A. Finkel and M. Jantzen (eds.), Springer-Verlag, Berlin (1992) 173–184.
15. J. A. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events", *Mathematical Theory of Automata*, vol. 12 of MRI Symposia Series, Polytechnic Press, NY, (1962), 529–561.
16. J. A. Brzozowski, "Derivatives of Regular Expressions", *Journal of the ACM* 11:4 (1964) 481–494.
17. J. A. Brzozowski, "Developments in the theory of regular languages", *Information Processing 80*, S. H. Lavington edited, Proceedings of IFIP Congress 80, North-Holland, Amsterdam (1980) 29–40.
18. J. A. Brzozowski, "Open problems about regular languages", *Formal Language Theory – Perspectives and open problems*, R. V. Book (ed.), Academic Press, New York, (1980), pp.23–47.

19. J. A. Brzozowski and E. Leiss, "On Equations for Regular Languages, Finite Automata, and Sequential Networks", *Theoretical Computer Science* 10 (1980) 19–35.
20. J. A. Brzozowski and I. Simon, "Characterization of locally testable events", *Discrete Mathematics* 4 (1973) 243–271.
21. J. A. Brzozowski and M. Yoeli, *Digital Networks*, Prentice-Hall, Englewood Cliffs, N. J., (1976).
22. A. K. Chandra and L. J. Stockmeyer, "Alternation", FOCS 17 (1976) 98–108.
23. A. K. Chandra, D. C. Kozen, L. J. Stockmeyer, "Alternation", *Journal of the ACM* 28 (1981) 114–133.
24. J. H. Chang, O. H. Ibarra and B. Ravikumar, "Some observations concerning alternating Turing machines using small space", *Inform. Process. Lett.* 25 (1987) 1–9.
25. C.-H. Chang and R. Paige, "From Regular Expressions to DFA's Using Compressed NFA's", *Proceedings of the Third Symposium on Combinatorial Pattern Matching* (1992) 90–110.
26. S. Cho and D. Huynh, "The parallel complexity of finite state automata problems", *Technical Report* UTDCS-22–88, University of Texas at Dallas, (1988).
27. D. I. A. Cohen, *Introduction to Computer Theory*, Wiley, New York, (1991).
28. K. Culik II and S. Dube, "Rational and Affine Expressions for Image Description", *Discrete Applied Mathematics* 41 (1993) 85–120.
29. K. Culik II and S. Dube, "Affine Automata and Related Techniques for Generation of Complex Images", *Theoretical Computer Science* 116 (1993) 373–398.
30. K. Culik II, F. E. Fich and A. Salomaa, "A Homomorphic Characterization of Regular Languages", *Discrete Applied Mathematics* 4 (1982)149–152.
31. K. Culik II and T. Harju, "Splicing semigroups of dominoes and DNA", *Discrete Applied Mathematics* 31 (1991) 261–277.
32. K. Culik II and J. Karhumäki, "The equivalence problem for single-valued two-way transducers (on NPDTOL languages) is decidable", *SIAM Journal on Computing*, vol. 16, no. 2 (1987) 221–230.
33. K. Culik II and J. Kari, "Image Compression Using Weighted Finite Automata", *Computer and Graphics*, vol. 17, 3, (1993) 305–313.
34. J. Dassow, G. Păun, A. Salomaa, "On Thinness and Slenderness of L Languages", *EATCS Bulletin* 49 (1993) 152–158.
35. F. Dejean and M. P. Schützenberger, "On a question of Eggan", *Information and Control* 9 (1966) 23–25.
36. A. de Luca and S. Varricchio, "On noncounting regular classes", *Theoretical Computer Science* 100 (1992) 67–104.
37. V. Diekert and G. Rozenberg (ed.), *The Book of Traces*, World Scientific, Singapore, (1995).
38. D. Drusinsky and D. Harel, "On the power of bounded concurrency I: Finite automata", *Journal of the ACM* 41 (1994) 517–539.
39. L. C. Eggan, "Transition graphs and the star height of regular events", *Michigan Math. J.* 10 (1963) 385–397.
40. A. Ehrenfeucht, R. Parikh, and G. Rozenberg, "Pumping Lemmas for Regular Sets", *SIAM Journal on Computing* vol. 10, no. 3 (1981) 536–541.
41. S. Eilenberg, *Automata, Languages, and Machines*, Vol. A, Academic Press, New York, (1974).
42. S. Eilenberg, *Automata, Languages, and Machines*, Vol. B, Academic Press, New York, (1974)
43. C. C. Elgot and J. D. Rutledge, "Operations on finite automata", *Proc. AIEE Second Ann. Symp. on Switching Theory and Logical Design*, Detroit, (1961).

44. A. Fellah, *Alternating Finite Automata and Related Problems*, PhD Dissertation, Dept. of Math. and Computer Sci., Kent State University, (1991).
45. A. Fellah, H. Jürgensen, S. Yu, "Constructions for alternating finite automata", *Intern. J. Computer Math.* 35 (1990) 117–132.
46. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, (1979.)
47. S. Ginsburg, *Algebraic and automata-theoretic properties of formal languages*, North-Holland, Amsterdam, (1975).
48. V. M. Glushkov, "The abstract theory of automata", *Russian Mathematics Surveys* 16 (1961) 1–53.
49. D. Gries, "Describing an Algorithm by Hopcroft", *Acta Informatica* 2 (1973) 97–109.
50. L. Guo, K. Salomaa, and S. Yu, "Synchronization Expressions and Languages", *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing* (1994) 257–264.
51. M. A. Harrison, *Introduction to Formal Language Theory*, Addison-Wesley, Reading, (1978).
52. K. Hashiguchi, "Algorithms for Determining Relative Star Height and Star Height", *Information and Computation* 78 (1988) 124–169.
53. T. Head, "Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors", *Bull. Math. Biol.* 49 (1987) 737–759.
54. F. C. Hennie, *Finite-State Models for Logical Machines*, Wiley, New York, (1968).
55. T. Hirst and D. Harel, "On the power of bounded concurrency II: Pushdown automata", *Journal of the ACM* 41 (1994), 540–554.
56. J. E. Hopcroft, "An $n \log n$ algorithm for minimizing states in a finite automaton", in *Theory of Machines and Computations*, Z. Kohavi (ed.), Academic Press, New York, (1971).
57. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, (1979), 189–196.
58. J. M. Howie, *Automata and Languages*, Oxford University Press, Oxford, (1991).
59. H. B. Hunt, D. J. Rosenkrantz, and T. G. Szymanski, "On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages", *Journal of Computer and System Sciences* 12 (1976) 222–268.
60. O. Ibarra, "The unsolvability of the equivalence problem for epsilon-free NGSM's with unary input (output) alphabet and applications", *SIAM Journal on Computing* 4 (1978) 524–532.
61. K. Inoue, I. Takanami, and H. Tanaguchi, "Two-Dimensional alternating Turing machines", *Proc. 14th Ann. ACM Symp. On Theory of Computing* (1982) 37–46.
62. K. Inoue, I. Takanami, and H. Tanaguchi, " A note on alternating on-line Turing machines", *Information Processing Letters* 15:4 (1982) 164–168.
63. J. Jaffe, "A necessary and sufficient pumping lemma for regular languages", *SIGACT News* (1978) 48–49.
64. T. Jiang and B. Ravikumar, "A note on the space complexity of some decision problems for finite automata", *Information Processing Letters* 40 (1991) 25–31.
65. T. Jiang and B. Ravikumar, "Minimal NFA Problems are Hard", *SIAM Journal on Computing* 22 (1993), 1117–1141. *Proceedings of 18th ICALP*, Lecture Notes in Computer Science 510, Springer-Verlag, Berlin (1991) 629–640.
66. N. Jones, "Space-bounded reducibility among combinatorial problems", *Journal of Computer and System Sciences* 11 (1975) 68–85.

67. T. Kameda and P. Weiner, "On the state minimization of nondeterministic finite automata", *IEEE Trans. on Computers* C-19 (1970) 617–627.
68. D. Kelley, *Automata and Formal Languages – An Introduction*, Prentice-Hall, Englewood Cliffs, N. J., (1995).
69. S. C. Kleene, "Representation of events in nerve nets and finite automata", *Automata Studies*, Princeton Univ. Press, Princeton, N. J., (1996), pp.2–42.
70. D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings", *SIAM Journal on Computing*, vol.6, no.2 (1977) 323–350.
71. D. Kozen, "On parallelism in Turing machines", *Proceedings of 17th FOCS* (1976) 89–97.
72. R. E. Ladner, R. J. Lipton and L. J. Stockmeyer, "Alternating pushdown automata", *Proc. 19th IEEE Symp. on Foundations of Computer Science*, Ann Arbor, MI, (1978) 92–106.
73. E. Leiss, "Succinct representation of regular languages by boolean automata", *Theoretical Computer Science* 13 (1981) 323–330.
74. E. Leiss, "On generalized language equations", *Theoretical Computer Science* 14 (1981) 63–77.
75. E. Leiss, "Succinct representation of regular languages by boolean automata II", *Theoretical Computer Science* 38 (1985) 133–136.
76. E. Leiss, "Language equations over a one-letter alphabet with union, concatenation and star: a complete solution", *Theoretical Computer Science* 131 (1994) 311–330.
77. E. Leiss, "Unrestricted complementation in language equations over a one-letter alphabet", *Theoretical Computer Science* 132 (1994) 71–84.
78. H. R. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, N. J., (1981).
79. P. A. Lindsay, "Alternation and w-type Turing acceptors", *Theoretical Computer Science* 43 (1986) 107–115.
80. P. Linz, *An Introduction to Formal Languages and Automata*, D. C. Heath and Company, Lexington, (1990).
81. O. B. Lupanow, "Über den Vergleich zweier Typen endlicher Quellen", *Probleme der Kybernetik* 6 (1966) 328–335, and *Problemy Kibernetiki* 6 (1962) (Russian original).
82. A. Mateescu, "Scattered deletion and commutativity", *Theoretical Computer Science* 125 (1994) 361–371.
83. W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity", *Bull. Math. Biophysics* 5 (1943) 115–133.
84. R. McNaughton, *Counter-Free Automata*, MIT Press, Cambridge, (1971).
85. R. McNaughton and H. Yamada, "Regular Expressions and State Graphs for Automata", *Trans. IRS* EC-9 (1960) 39–47. Also in *Sequential Machines – Selected Papers*, E. F. Moore ed., Addison-Wesley, Reading, (1964), 157–174.
86. G. H. Mealy, "A method for synthesizing sequential circuits", *Bell System Technical J.* 34: 5 (1955), 1045–1079.
87. A. R. Meyer and M. J. Fischer. "Economy of description by automata, grammars, and formal systems", *FOCS* 12 (1971) 188–191.
88. E. F. Moore, "Gedanken experiments on sequential machines", *Automata Studies*, Princeton Univ. Press, Princeton, N.J., (1966), pp.129–153.
89. F. R. Moore, "On the Bounds for State-Set Size in the Proofs of Equivalence Between Deterministic, Nondeterministic, and Two-Way Finite Automata", *IEEE Trans. Computers* 20 (1971), 1211–1214.
90. J. Myhill, "Finite automata and the representation of events", WADD TR-57-624, Wright Patterson AFB, Ohio, (1957), 112–137.

91. A. Nerode, "Linear automata transformation", *Proceedings of AMS* 9 (1958) 541–544.
92. O. Nierstrasz, "Regular Types for Active Objects", *OOPSLA'93*, 1–15.
93. M. Nivat, "Transductions des langages de Chomsky", *Ann. Inst. Fourier, Grenoble* 18 (1968) 339–456.
94. W. J. Paul, E. J. Prauss and R. Reischuck, "On Alternation", *Acta Inform.* 14 (1980) 243–255.
95. G. Păun and A. Salomaa, "Decision problems concerning the thinness of DOL languages", *EATCS Bulletin* 46 (1992) 171–181.
96. G. Păun and A. Salomaa, "Closure properties of slender languages", *Theoretical Computer Science* 120 (1993) 293–301.
97. G. Păun and A. Salomaa, "Thin and slender languages", *Discrete Applied Mathematics* 61 (1995) 257–270.
98. D. Perrin, (Chapter 1) Finite Automata, *Handbook of Theoretical Computer Science*, Vol. B, J. van Leeuwen (ed.), MIT Press, (1990).
99. M. O. Rabin and D. Scott, "Finite automata and their decision problems", *IBM J. Res.* 3: 2 (1959) 115–125.
100. G. N. Raney, "Sequential functions", *Journal of the ACM* 5 (1958) 177.
101. B. Ravikumar, "Some applications of a technique of Sakoda and Sipser", *SIGACT News*, 21:4 (1990) 73–77.
102. B. Ravikumar and O. H. Ibarra, "Relating the type of ambiguity of finite automata to the succinctness of their representation", *SIAM Journal on Computing* vol. 18, no. 6 (1989), 1263–1282.
103. W. L. Ruzzo, "Tree-size bounded alternation", *Journal of Computer and System Sciences* 21 (1980) 218–235.
104. A. Salomaa, *On the Reducibility of Events Represented in Automata*, Annales Academiae Scientiarum Fennicae, Series A, I. Mathematica 353, (1964).
105. A. Salomaa, *Theorems on the Representation of events in Moore-Automata*, Turun Yliopiston Julkaisuja Annales Universitatis Turkuensis, Series A, 69, (1964).
106. A. Salomaa, *Theory of Automata*, Pergamon Press, Oxford, (1969).
107. A. Salomaa, *Jewels of Formal Language Theory*, Computer Science Press, Rockville, Maryland, (1981).
108. A. Salomaa, *Computation and Automata*, Cambridge University Press, Cambridge, (1985).
109. A. Salomaa and M. Soittola, *Automata-Theoretic Aspects of Formal Power Series*, Springer-Verlag, New York, (1978).
110. K. Salomaa and S. Yu, "Loop-Free Alternating Finite Automata", *Technical Report* 482, Department of Computer Science, Univ. of Western Ontatio, (1996).
111. K. Salomaa, S. Yu, Q. Zhuang, "The state complexities of some basic operations on regular languages", *Theoretical Computer Science* 125 (1994) 315–328.
112. M. P. Schützenberger, "Finite Counting Automata", *Information and Control* 5 (1962) 91–107.
113. M. P. Schützenberger, "On finite monoids having only trivial subgroups", *Information and Control* 8 (1965) 190–194.
114. M. P. Schützenberger, "Sur les relations rationelles", in *Proc. 2nd GI Conference, Automata Theory and Formal languages*, H. Braklage (ed.), *Lecture Notes in Computer Science* 33, Springer-Verlag, Berlin (1975) 209–213.
115. J. Shallit, "Numeration systems, linear recurrences, and regular sets", *Information and Computation* 113 (1994) 331–347.
116. J. Shallit and J. Stolfi, "Two methods for generating fractals", *Computers & Graphics* 13 (1989) 185–191.

117. P. W. Shor, "A Counterexample to the triangle conjecture", *J. Combinatorial Theory*, Series A (1985) 110–112.
118. J. L. A. van de Snepscheut, *What Computing Is All About*, Springer-Verlag, New York, (1993).
119. L. Stockmeyer and A. Meyer, "Word problems requiring exponential time (preliminary report)", *Proceedings of the 5th ACM Symposium on Theory of Computing*, (1973) 1–9.
120. A. Szilard, S. Yu, K. Zhang, and J. Shallit, "Characterizing Regular Languages with Polynomial Densities", *Proceedings of the 17th International Symposium on Mathematical Foundations of Computer Science*, *Lecture Notes in Computer Science* 629 Springer-Verlag, Berlin (1992) 494–503.
121. K. Thompson, "Regular Expression Search Algorithm", *Communications of the ACM* 11:6 (1968) 410–422.
122. B. W. Watson, *Taxonomies and Toolkits of Regular Language Algorithms*, PhD Dissertation, Department of Mathematics and Computing Science, Eindhoven University of Technology, (1995).
123. D. Wood, *Theory of Computation*, Wiley, New York, (1987).
124. S. Yu and Q. Zhuang, "On the State Complexity of Intersection of Regular Languages", *ACM SIGACT News*, vol. 22, no. 3, (1991) 52–54.
125. Y. Zalcstein, "Locally testable languages", *Journal of Computer and System Sciences* 6 (1972) 151–167.