Table of Contents

Automata for Matching Patterns

Maxime Crochemore[1] and Christophe Hancart[2]

[1] Institut Gaspard Monge, Université de Marne-la-Vallée, 93166 Noisy-le-Grand Cedex, France
[2] Laboratoire d'Informatique de Rouen, Université de Rouen, Faculté des Sciences et Techniques, 76821 Mont-Saint-Aignan Cedex, France

1. Pattern matching and automata

This chapter describes several methods of word pattern matching that are based on the use of automata.

Pattern matching (in words) is the problem of locating occurrences of a pattern in a text file. The file is just a string of symbols, but the pattern can be specified in various ways. Here, we only consider patterns described by regular expressions or weaker mechanisms.

Solutions to the problem are basic parts of many text processing tools, such as editors, parsers, and information retrieval systems. They are also widely used in the analysis of biological sequences. The algorithms that solve the problem classically decompose in two steps: a preprocessing phase and a search phase. When the text file is considered to be dynamic (as in editing applications), the preprocessing is applied to the pattern (see Sections 4, 5, and 6). This leads *a posteriori* to a good solution regarding the efficiency of the algorithms of this chapter. When the text file is static (if it is a dictionary, for example) the preprocessing applied to the text builds an index that can later support efficiently several series of queries (see Section 7).

We present solutions in which the search phase is based on automata as opposed to solutions based on combinatorial properties of words. Thus, the algorithms perform on-line searches with a buffer on the text that does not need to store more than one letter at a time. The solutions are adequate for processing sequential-access files or streams of symbols.

The main algorithms of this chapter solve special instances of the determinization or minimization problems of automata. Basically, given an automaton that recognizes the language X on the alphabet A, algorithms build a deterministic, and sometimes minimal, automaton for the language A^*X, which is applied afterwards to search efficiently for words of X.

The time complexity of algorithms is given as a function of the input, and is typically linear in the length of the input. This takes into account the set of letters actually occurring in the input. But the running time may depend on the output as well. So, a careful statement of each problem is necessary, to avoid for example quadratic-size outputs that would obviously imply quadratic-time algorithms.

The complexity of algorithms is analyzed in a model of a machine in which the basic operation on letters is comparison in the form less-equal-greater. The implicit ordering on the alphabet is exploited in several algorithms. The assumption on the model makes it possible to process words over a potentially unbounded alphabet. Some algorithms for the simplest pattern-matching problem (searching for only one word) operates in a weaker model (comparison in the form equal-unequal). We also mention how the running times of most algorithms are affected when branchings in automata are performed by looking up a transition table (see Section 3). This is valid if the alphabet is known in advance and if the letters can be assimilated to indices on a table. Otherwise, a straightforward simulation implies that the running times are multiplied by $O(\log \mathrm{card}(A))$, while in the comparison model the running times of some algorithms are independent of the alphabet.

The regular-expression-matching problem (Section 4) is when the pattern is a general regular expression. The standard solution is certainly by Thompson (1968). The mechanism is one of the basic features of the UNIX operating system and of its tools.

When the language described by the pattern reduces to a finite set of words (Section 5), called a dictionary, the pattern-matching algorithm runs in linear time (on a fixed alphabet) instead of quadratic time for the general solution. Moreover, when the pattern is only one word (Section 6), the same running time holds, independently of the alphabet.

The suffix automata presented in Section 7 serve as indexes. They provide a solution to the pattern-matching instance where the searched text has to be preprocessed. The main point of the section is the linear-time construction of suffix automata (on a fixed alphabet), which results partially from their linear size.

The efficiency of pattern-matching algorithms based on automata strongly relies on particular representations of these automata. This is why a review of several techniques is given in Section 3. The regular-expression-matching problem, the dictionary-matching problem, and the string-matching problem are treated respectively in Sections 4, 5, and 6. Section 7 deals with suffix automata and their applications.

2. Notations

This section is devoted to a review of the material used in this chapter: alphabet, words, languages, regular expressions, finite automata, algorithms for matching patterns.

2.1 Alphabet and words

Let A be a finite set, called the *alphabet*. Its elements are called *letters*, and, for convenience, we denote them by a, b, c, and so on. Furthermore, we assume that there is an ordering on the alphabet.

A *word* is a finite-length sequence of letters. The *length* of a word u is denoted by $|u|$, and its j-th letter by u_j. The set of all words is denoted by A^*, the empty word by ε, and A^+ stands for $A^* \setminus \{\varepsilon\}$.

The *product* of two words u and v, denoted by $u \cdot v$ or uv, is the word obtained by writing sequentially the letters of u then the letters of v. Given a word u, the product of k words identical with u is denoted by u^k, setting $u^0 = \varepsilon$. Denoted respectively by uw^{-1} and $v^{-1}u$ are the words v and w when $u = vw$.

A word v is said to be a *factor* of a word u if $u = u'vu''$ for some words u' and u''; it is a *proper* factor of u if $v \neq u$, a *prefix* of u if $u' = \varepsilon$, and a *suffix* of u if $u'' = \varepsilon$.

2.2 Languages

A *language* is any subset of A^*. The *product* of two languages U and V, denoted by $U \cdot V$ or UV, is the language $\{uv \mid (u, v) \in U \times V\}$. Denoted by U^k is the set of words obtained by making products of k words of U. The *star* of U, denoted by U^*, is the language $\bigcup_{k \geqslant 0} U^k$. By convention, the order of decreasing precedence for language operations in expressions denoting languages is star or power, product, union. By misuse, a language reduced to only one word u may be denoted by u itself if no confusion arises (with further notations).

The sets of prefixes, of factors, and of suffixes of a language U are denoted respectively by $\mathit{Pref}(U)$, $\mathit{Fact}(U)$, and $\mathit{Suff}(U)$. If U is finite, $|U|$ stands for $\sum_{u \in U} |u|$ (therefore, note that $\mathrm{card}(A) = |A|$).

The *right context* of a word u according to a language W is the language $\{u^{-1}w \mid w \in W\}$. The equivalence generated over A^* by the relations

$$u^{-1}W = v^{-1}W, \quad u, v \in A^*$$

is denoted by $\equiv_W$; it is the *right syntactic congruence* associated with the language W.

2.3 Regular expressions

Regular expressions and the languages they describe, the *regular languages*, are defined inductively as follows:

- $\mathbf{0}$, $\mathbf{1}$, and a are regular expressions and describes respectively $\varnothing$ (the empty set), $\{\varepsilon\}$, and $\{a\}$, for each $a \in A$;
- if $\boldsymbol{u}$ and $\boldsymbol{v}$ are regular expressions describing respectively the regular languages U and V, then $\boldsymbol{u} + \boldsymbol{v}$, $\boldsymbol{u} \cdot \boldsymbol{v}$, and $\boldsymbol{u}^*$ are regular expressions describing respectively the regular languages $U \cup V$, $U \cdot V$, and U^*.

By convention, the order of decreasing precedence for operations in regular expressions is star (*), product ($\cdot$), sum ($+$). The dot $\cdot$ is often omitted. Parenthesizing can be used to change the precedence order of operators.

The language described by a regular expression $\boldsymbol{u}$ is denoted by $Lang(\boldsymbol{u})$. The *length* $|\boldsymbol{u}|$ of a regular expression $\boldsymbol{u}$ is the length of $\boldsymbol{u}$ reckoned on the alphabet $A \cup \{\mathbf{0}, \mathbf{1}, +, ^*\}$ (parentheses and product operator $\cdot$ are not reckoned).

2.4 Finite automata

A *(finite) automaton (with one initial state)* is given by a finite set Q, whose elements are called *states*, an *initial* state i, a subset $T \subseteq Q$ of *terminal* states, and a set $E \subseteq Q \times A \times Q$ of *edges*.

An edge (p, a, q) of the automaton (Q, i, T, E) is an *outgoing* edge for state p and an *ingoing* edge for state q; state p is the *source* of this edge, letter a its *label*, and state q its *target*. The number of edges outgoing a state p is called the *(outgoing) degree* of p. We say that there is a *path* labeled by u from state p to state q if there is a finite sequence $(r_{j-1}, a_j, r_j)_{1 \leqslant j \leqslant n}$ of edges such that $n = |u|$, $a_j = u_j$ for each $j \in \{1, \ldots, n\}$, $r_0 = p$ and $r_n = q$. One agrees to define a unique path labeled by ε from each state p to itself.

A word u is *recognized* by the automaton $\mathcal{A} = (Q, i, T, E)$ if there exists a path labeled by u from i to some state in T. The set of all words recognized by $\mathcal{A}$ is denoted by $Lang(\mathcal{A})$. A language X is *recognizable* if there exists an automaton $\mathcal{A}$ such that $X = Lang(\mathcal{A})$.

As an example, the automaton depicted in Figure 2.1 recognizes the language $\{\mathsf{a}, \mathsf{b}\}^*\mathsf{abaaab}$. Its initial state is 0, and its only terminal state is 6.

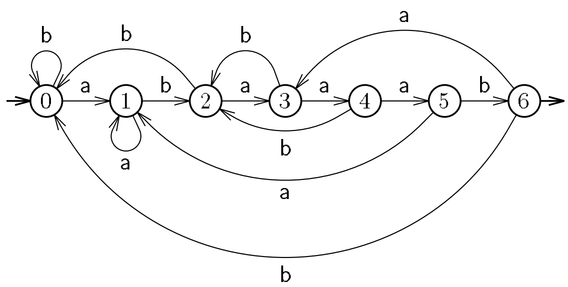


Fig. 2.1. An automaton recognizing the language $\{\mathsf{a}, \mathsf{b}\}^*\mathsf{abaaab}$.

The automaton (Q, i, T, E) is *deterministic* if for each $(p, a) \in Q \times A$ there is at most one state q such that $(p, a, q) \in E$. It is *complete* if for each $(p, a) \in Q \times A$ there is at least one state q such that $(p, a, q) \in E$. It is *normalized* if $\operatorname{card}(T) = 1$, the initial state has no ingoing edge, and the terminal state has no outgoing edge. It is *minimal* if it is deterministic and if each deterministic automaton recognizing the same language maps onto it; it has the minimal number of states. The minimal automaton recognizing the language U is denoted by $\mathcal{M}(U)$. It can be defined with the help of right contexts by:

$$\Big(\{u^{-1}U \mid u \in A^*\},\ \{U\},\ \{u^{-1}U \mid u \in U\},$$
$$\{(u^{-1}U, a, (ua)^{-1}U) \mid u \in A^*, a \in A\}\Big)$$

In case $\mathcal{A} = (Q, i, T, E)$ is a deterministic automaton, it is convenient to consider the *transition function* $\delta: Q \times A \to Q$ of $\mathcal{A}$ defined for each $(p, a) \in Q \times A$ such that there is an outgoing edge labeled by a for p by

$$\delta(p, a) = q \iff (p, a, q) \in E$$

(notice that δ is a partial function). Equivalently, the quadruple (Q, i, T, δ) denotes the automaton $\mathcal{A}$. In a natural way, the transition function extends to a function mapping from $Q \times A^*$ to Q and also denoted by δ setting

$$\delta(p,u) = \begin{cases} p, & \text{if } u = \varepsilon, \\ \delta(\delta(p,a),v), & \text{if } \delta(p,a) \text{ is defined and } u = av \\ & \quad \text{for some } (a,v) \in A \times A^*, \\ \text{undefined}, & \text{otherwise,} \end{cases}$$

for each $(p,u) \in Q \times A^*$.

In algorithms that manipulate automata, we constantly use the function STATE-CREATION described in Figure 2.2 (+ stands for the union of sets). This avoids going into details of the implementation of automata that is precisely the subject of Section 3.

```
STATE-CREATION
chose a state q out of Q
Q ← Q + {q}
return q
```

Fig. 2.2. Creation of a new state and adjunction to the set of states Q.

2.5 Algorithms for matching patterns

The pattern matching problem is to search and locate occurrences of patterns in words (or textual data, less formally speaking). A *pattern* represents a language and is described either by a word, by a finite set of words, or more generally, by a regular expression. We do not consider patterns described by other mechanisms.

Let y be the searched word. An *occurrence* in y of a pattern represented by the language X is a triple (u,x,v) where $u,v \in A^*$, $x \in X$, and such that $y = uxv$. The *position* of the occurrence (u,x,v) of x in y is the length $|u|$; it is sometimes more convenient to consider the *end-position* of the same occurrence, which is defined as the length $|ux|$. Observe that searching y for words in a language X is equivalent to search for prefixes of y that belong to the language A^*X; the language of most automata considered in this chapter is of this form.

According to a specific matching problem, the input of an algorithm is a language X described by a word, by a finite set of words, or by a regular expression, and a word y. The output can have several forms. To implement an algorithm that tests whether the pattern occurs in the word or not, the output is just the boolean value TRUE or FALSE respectively. In an on-line search, what is desired is the word, say z, on the alphabet $\{0,1\}$ that encodes the existence of end-positions of the pattern; the length of z is $|y|+1$, and its $j+1$-th letter is 1 exactly when an occurrence of the pattern ends at position j in y. The output can also be the set, say P, of positions (or end-positions) in y of the pattern. To avoid presenting several variants of algorithms, we introduce the statement

occurrenceif e

where e is an appropriate predicate. It can be translated by

```
if e
  then return TRUE
```

in the first case,

```
if e
  then z ← z · 1
  else z ← z · 0
```

in the second case, and

```
if e
  then P ← P + {the current position in y}
```

```
MATCHER(X, y)
    Preprocessing phase
    built an automaton (Q, i, T, E) recognizing A*X
    Search phase
    let δ be the transition function of (Q, i, T, E)
    p ← i
    occurrenceif p ∈ T
    for letter a from first to last letter of y
      loop p ← δ(p, a)
           occurrenceif p ∈ T
```

Fig. 2.3. Given a regular language X and a word y, locate all occurrences of words in X that are factors of y.

in the third case. In the first case, the "**return** FALSE" statement has to be included correspondingly at the end of the algorithm; and in the other cases, word z and set P should be initialized at the beginning of the algorithm and returned at the end of the algorithm. From now on, the standard algorithm for matching patterns in words can be written as in Figure 2.3.

The asymptotic time and space complexities of algorithm MATCHER depend on the representation of the automaton, and more specifically, on the representation of the transition function δ (see Section 3). More generally, the complexities of algorithms, functions or procedures developed in this chapter are expressions of the size of the input. They include the size of the language, the length of the searched word, and the size of the alphabet. We assume that the "**occurrenceif** e" statement is performed in constant time. Nevertheless, an *ad hoc* output often underlies the complexity result.

3. Representations of deterministic automata

Several pattern matching algorithms rely on a particular representation of the deterministic automaton underlying the method. Implementing a deterministic automaton (Q, i, T, E) remains to implement the transition function δ of the automaton, which is the general problem of realizing partial functions. Five methods are described in this section: transition matrix, adjacency lists, transition list, failure function, and table-compression.

The choice of the representation of the automaton influence the time needed to compute a transition, *i.e.* the time to evaluate $\delta(p, a)$, for any state p and any letter a. This time is called the *delay*, in that it is also the time spent on letter a before moving to the next letter of the input word. Basically: on the one hand, the time to evaluate $\delta(p, a)$ is constant in a model where branchings are allowed and a transition matrix implements δ; on the other hand, if comparison of letters is the only operation allowed on them, the time to evaluate $\delta(p, a)$ is $O(\log \mathrm{card}(A))$, assuming that any two letters can be compared in one unit of time (using binary operations $=$, $\neq$, $<$ or $>$). In the following, we give the memory space and the delay associated to each type of representation. There is an obvious trade-off between these two quantities.

In the chapter, having a representation R of the transition function δ, the automaton is indifferently denoted by (Q, i, T, E), (Q, i, T, δ), and (Q, i, T, R).

3.1 Transition matrix

The simplest method to implement the transition function δ is to store its values in a $Q \times A$-matrix. This is a method of choice for a complete deterministic automaton on a small alphabet and when letters can be assimilated to indices on an array. The space required is $O(\mathrm{card}(Q) \times \mathrm{card}(A))$ and the delay is $O(1)$.

When the automaton is not complete, the representation still works except that the searching procedure can stop on an undefined transition. The matrix can even be initialized in time proportional to the number of edges of the automaton with the help of a sparse matrix representation technique. The above complexities are still valid in this situation.

This kind of representation implicitly assumes that the working alphabet is fixed and known by advance. This contrasts with the representations of Sections 3.2 and 3.4 for which the basic operation on the alphabet is comparing letters.

3.2 Adjacency lists

A traditional way of implementing graphs is to use adjacency lists. This applies to automata as well. Doing so, the set of couples $(a, \delta(p, a))$, whenever $\delta(p, a)$ is defined, is associated with each state $p \in Q$. The space required to represent the $\mathrm{card}(Q)$ adjacency lists of the automaton is $O(\mathrm{card}(Q) + \mathrm{card}(E))$. Contrary to the previous method, this one works even if the only possible elementary operation on letters is comparison. Denoting by d the maximum degree of states of the automaton, the delay is $O(\log d)$, which is also $O(\log \min\{\mathrm{card}(Q), \mathrm{card}(A)\})$, using an efficient implementation of sets based for instance on balanced trees.

The space complexity may be further reduced by considering a *default (target) state* associated to each adjacency list (the most frequently occurring target of a given adjacency list is an obvious choice as default for this adjacency list). The delay can even be improved at the same time because adjacency lists become smaller.

When implementing the automaton, each adjacency list is stored in an array G indexed by Q. If the deterministic automaton is complete and if the initial state i is the uniform default state (i as default state fits in perfectly with pattern matching applications), the computation of a transition from any state p by any letter a, that is, the computation of $\delta(p, a)$, is done by the function of Figure 3.1.

```
AdjacencyLists-Transition(p, a)
1  p ← state of the couple of label a in G[p]
2  if p = NIL
3     then p ← i
4  return p
```

Fig. 3.1. Computation of the transition from a state p by a letter a when an array G of adjacency lists represents the transition function.

3.3 Transition list

The transition list method consists in implementing the list of triples (p, a, q) of edges of the automaton. The space required by the implementation is only $O(\mathrm{card}(E))$. Doing so, it is assumed that the list is stored in a hashing table to provide fast computations of transitions. The corresponding hashing function is defined on couples (p, a) from $Q \times A$. Then, given a couple (p, a), the access to the transition (p, a, q), if it appears in the list, is performed in constant time on the average under usual hypotheses on the technique.

3.4 Failure function

The main idea of the failure function method is to reduce the space needed by δ, by deferring, in most possible cases, the computation of the transition from the current state to the computation of the transition from an other given state with the same input letter. It serves to implement deterministic automata in the comparison model. Its main advantage is that, in general, it provides a linear-space representation, and, simultaneously, gives a linear-time cost for a series of transitions, though the time to compute one transition is not necessarily constant.

We only consider the case where the deterministic automata is complete and where i is the default state (extensions of the following statement are not needed in the chapter).

Let γ be a function from $Q \times A$ to Q, and let f be a function from Q into itself. We say that the couple (γ, f) represents the transition function δ if γ is a subfunction of δ and if

$$\delta(p, a) = \begin{cases} \gamma(p, a), & \text{if } \gamma(p, a) \text{ is defined,} \\ \delta(f(p), a), & \text{if } \gamma(p, a) \text{ is undefined and } f(p) \text{ is defined,} \\ i, & \text{otherwise,} \end{cases}$$

for each $(p, a) \in Q \times A$. In this situation, the state $f(p)$ is a stand-in of state p. The functions γ and f are respectively said to be a *subtransition function* and a *failure function*, according to δ. However, this representation is correct if we assume that f defines an order on Q.

Assuming a representation of γ by adjacency lists, the space needed to represent δ by the couple (γ, f) is $O(\mathrm{card}(Q) + \mathrm{card}(E'))$, where

$$E' = \{(p, a, q) \mid (p, a, q) \in E \text{ and } \gamma(p, a) \text{ is defined}\}),$$

which is of course $O(\mathrm{card}(Q) + \mathrm{card}(E))$ since $E' \subseteq E$. (Notice that γ is the transition function of the automaton (Q, i, T, E').) If d is the maximum degree of states of the automaton (Q, i, T, E'), the delay is typically $O(\mathrm{card}(Q) \times \log d)$, that is also $O(\mathrm{card}(Q) \times \log \mathrm{card}(A))$.

When implementing the automaton, the values of the failure function f are stored in an array F indexed by Q. The computation of a transition is done by the function of Figure 3.2. The function always stops if we assume that f defines an order on Q.

FailureFunction-Transition(p, a)
while $p \neq$ NIL and $\gamma(p, a) =$ NIL
  loop $p \leftarrow F[p]$
if $p \neq$ NIL
  then $p \leftarrow \gamma(p, a)$
  else $p \leftarrow i$
return p

Fig. 3.2. Computation of the transition from a state p by a letter a when a subtransition γ and an array F corresponding to a failure function represent the transition function.

3.5 Table-compression

The latest method is a mix of the previous ones that provides fast computations of transitions via arrays and a compact representation of edges via failure function.

Four arrays, denoted here by *fail*, *base*, *target*, and *check*, are used. The *fail* and *base* arrays are indexed by Q, and, for each $(p, a) \in Q \times A$, $base[p] + a$ is an index on *target* and *check* arrays, assimilating letters to integers.

The computation of the transition from some state p with some input letter a proceeds as follows: let $k = base[p] + a$; then, if $check[k] = p$, $target[k]$ is the target of the edge of source p and label a; otherwise, this statement is repeated recursively with state $fail[p]$ and letter a. (Notice that it is correct if *fail* defines an order on Q, as in Section 3.4, and if the targets from the smallest element are all defined.) The corresponding function is given in Figure 3.3.

TableCompression-Transition(p, a)
while $check[base[p] + a] \neq p$
  loop $p \leftarrow fail[p]$
return $target[base[p] + a]$

Fig. 3.3. Computation of the transition from a state p by a letter a in the table-compression method with suitable arrays *fail*, *base*, *target*, and *check*.

In the worst case, the space needed is $O(\mathrm{card}(Q) \times \mathrm{card}(A))$ and the delay is $O(\mathrm{card}(Q))$. However the method can reduce the space to $O(\mathrm{card}(Q) + \mathrm{card}(A))$ with an $O(1)$ delay in best possible situation.

4. Matching regular expressions

4.1 Outline

Problem 4.1. (Regular-expression-matching problem.) Given a regular expression $\boldsymbol{x}$, preprocess it in order to locate all occurrences of words of $Lang(\boldsymbol{x})$ that occur in any given word y.

A well-known solution to the above problem is composed of two phases. First, transform the regular expression $\boldsymbol{x}$ into a nondeterministic automaton that recognizes the language described by $\boldsymbol{x}$, following a construction due to Thompson. Second, simulate the obtained automaton with input word y in such a way that it recognizes each prefix of y that belongs to $A^*Lang(\boldsymbol{x})$.

Main Theorem 4.1. *The regular expression-matching problem for $\boldsymbol{x}$ and y can be achieved in the following terms:*

- *a preprocessing phase on $\boldsymbol{x}$ building an automaton of size $O(|\boldsymbol{x}|)$, performed in time $O(|\boldsymbol{x}|)$ and $O(|\boldsymbol{x}|)$ extra-space;*
- *a search phase executing the automaton on y performed in time $O(|\boldsymbol{x}|\,|y|)$ and $O(|\boldsymbol{x}|)$ space, the time spent on each letter of y being $O(|\boldsymbol{x}|)$.*

The construction of the automaton is given in Section 4.2. In Section 4.3, we first show how to solve the membership test, namely, "does y belongs to $Lang(\boldsymbol{x})$?"; we then present the solution to the search phase of the regular-expression-matching problem as a mere transformation of the previous test. Finally, in Section 4.4, we discuss a possible use of a deterministic automaton to solve the problem.

In the whole section, we assume that the regular expression contains no redundant parentheses, because otherwise the parsing of the expression would not be necessarily asymptotically linear in the length of the expression.

4.2 Regular-expression-matching automata

In order to solve the problem in space linear in the length of the regular expression, we consider special nondeterministic automata.

We say that an automaton is *extended* if it is defined on the extended alphabet $A \cup \{\varepsilon\}$. Observe that a transition from a state to another in an extended automaton may either result of the reading of a letter from the input word, or not (ε-transition).

Theorem 4.2. *Let $\boldsymbol{x}$ be a regular expression. There exists a normalized extended automaton recognizing $Lang(\boldsymbol{x})$ satisfying the following conditions:*

(i) *the number of states is bounded by $2|\boldsymbol{x}|$;*
(ii) *the number of edges labeled by letters of A is bounded by $|\boldsymbol{x}|$, and the number of edges labeled by ε is bounded by $4|\boldsymbol{x}|$;*
(iii) *for each state the number of ingoing or outgoing edges is at most 2, and it is exactly 2 only when the edges are labeled by ε.*

Proof. The proof is by induction on the length of regular expressions.

The regular expressions of length equal to 1 are $\mathbf{0}$, $\mathbf{1}$, and a, for each $a \in A$. They are respectively recognized by normalized extended automata in the form

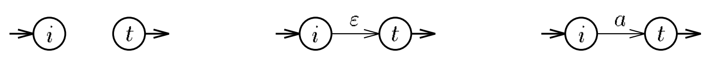
$$\Big(\{i,t\},\ i,\ \{t\},\ \varnothing\Big),$$

$$\Big(\{i,t\},\ i,\ \{t\},\ \{(i,\varepsilon,t)\}\Big),$$

and

$$\Big(\{i,t\},\ i,\ \{t\},\ \{(i,a,t)\}\Big),$$

where i and t are two distinct states. The automata are depicted in Figure 4.1.



(I) (II) (III)

Fig. 4.1. Normalized extended automata recognizing the regular expressions $\mathbf{0}$ (I), $\mathbf{1}$ (II), and a (III) for some $a \in A$.

Now, let $(Q', i', \{t'\}, E')$ and $(Q'', i'', \{t''\}, E'')$ be normalized extended automata recognizing respectively the regular expressions $\boldsymbol{u}$ and $\boldsymbol{v}$, assuming that $Q' \cap Q'' = \varnothing$. Then the regular expressions $\boldsymbol{u}+\boldsymbol{v}$, $\boldsymbol{u}\cdot\boldsymbol{v}$, and $\boldsymbol{u}^*$ are respectively recognized by normalized extended automata in the form

$$\Big(Q' \cup Q'' \cup \{i,t\},\ \ i,\ \ \{t\},\ \ E' \cup E'' \cup \big\{(i,\varepsilon,i'),\,(i,\varepsilon,i''),\,(t',\varepsilon,t),\,(t'',\varepsilon,t)\big\}\Big)$$

where i and t are two distinct states chosen out of $Q' \cup Q''$,

$$\Big(Q' \cup Q'',\ \ i',\ \ \{t''\},\ \ E' \cup E'' \cup \big\{(t',\varepsilon,i'')\big\}\Big),$$

and

$$\Big(Q' \cup \{i,t\},\ \ i,\ \ \{t\},\ \ E' \cup E'' \cup \big\{(i,\varepsilon,i'),\,(i,\varepsilon,t),\,(t',\varepsilon,i'),\,(t',\varepsilon,t)\big\}\Big)$$

where i and t are two distinct states chosen out of Q'. The automata are depicted in Figure 4.2.

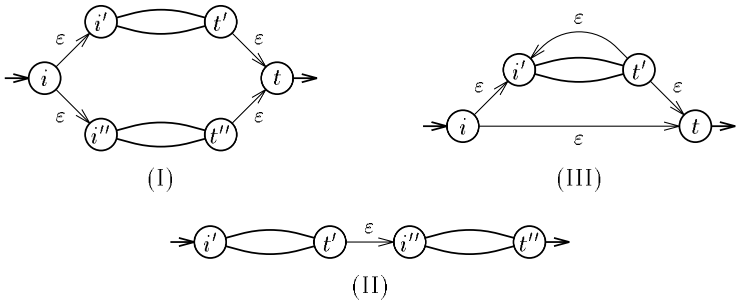


Fig. 4.2. Normalized extended automata recognizing the regular expressions $\boldsymbol{u}+\boldsymbol{v}$ (I), $\boldsymbol{u}\cdot\boldsymbol{v}$ (II), and $\boldsymbol{u}^*$ (III), obtained from normalized extended automata $(Q', i', \{t'\}, E')$ and $(Q'', i'', \{t''\}, E'')$ recognizing respectively the regular expressions $\boldsymbol{u}$ and $\boldsymbol{v}$.

The above construction clearly proves the existence of a normalized extended automaton recognizing the language described by any given regular expression. It remains to check that the automaton satisfies conditions (i) to (iii). Condition (i) holds since exactly two nodes are created for each letter of a regular expression accounting for its length. Condition (ii) is easy to establish, using similar arguments. And the last condition follows from construction. □

The previous result proves one half of the theorem of Kleene (the second half of the proof may be found in any standard textbook on automata or formal language theory).

Theorem 4.3 (Kleene, 1956). *A language is recognizable if and only if it is regular.*

We denote by $\mathcal{E}(\boldsymbol{x})$ the normalized extended automaton constructed in the proof of Theorem 4.2 from the regular expression $\boldsymbol{x}$, and we call it the *regular-expression-matching automaton* of $\boldsymbol{x}$.

To evaluate the time complexity of the above construction, it is necessary to give some hints about the data structures involved in the representation of regular-expression-matching automata. Due to the conditions stated in Theorem 4.2, a special representation of regular-expression-matching automata is possible providing an efficient implementation of the construction function. States are simply indices on an array that store edges; each cell of the array has to store at most two edges whose ingoing state is its index. Indices of the initial state and of the terminal state are stored separately. This shows that the space required to store $\mathcal{E}(\boldsymbol{x})$ is linear in its number of states, which is linear in the length of $\boldsymbol{x}$ according to Theorem 4.2.

Hence, each of the operations induced by $+$, $\cdot$ or * can be implemented to work in constant time. This proves that the time spent on each letter of $\boldsymbol{x}$ is constant. In addition, the function which builds the regular-expression-matching automaton corresponding to a given regular expression is driven by a parser of regular expressions. Then, if parenthesizing in $\boldsymbol{x}$ is not redundant, the time and the space needed for the construction of $\mathcal{E}(\boldsymbol{x})$ is linear in the length of $\boldsymbol{x}$.

We have established the following result:

Theorem 4.4. *Let $\boldsymbol{x}$ be a regular expression. The space needed to represent $\mathcal{E}(\boldsymbol{x})$ is $O(|\boldsymbol{x}|)$. The computation of the automaton is performed in time and space $O(|\boldsymbol{x}|)$.*

4.3 Searching with regular-expression-matching automata

The search for end-positions of words in $Lang(\boldsymbol{x})$ is performed with a simulation of a deterministic automaton recognizing $A^*Lang(\boldsymbol{x})$. Indeed, the determinization is avoided because it may lead to an automaton with a number of states which is exponential in the length of the regular expression (see Section 4.4). But the determinization via the subset construction is just simulated: at a given time, the automaton is not in a given state, but in a set of states. This subset is recomputed whenever necessary in the execution of the search.

As for the determinization of automata with ε-transitions, the searching procedure needs the notion of *closure* of a set of states: if S is a set of states, its closure is the set of states q such that there exists a path labeled by ε from a state of S to q. From the closure of a set of states, it is possible to compute effectively the transitions induced by any input letter.

The simulation of a regular-expression-matching automaton consists in repeating the two operations closure and computation of transitions on a set of states. These two operations are respectively performed by functions Closure and Transitions of Figures 4.3 and 4.4. With careful implementation, based on standard manipulation of sets and queues, the time and the space required to compute a closure or the transitions from a closure are linear in the size of involved sets of states.

```
Closure(E, S)
  R ← S
  ϑ ← EmptyQueue
  for each state p in S
    loop Enqueue(ϑ, p)
  while not QueueIsEmpty(ϑ)
    loop p ← Dequeue(ϑ)
         for each state q such that (p, ε, q) is in E
           loop if q is not in R
                  then R ← R + {q}
                       Enqueue(ϑ, q)
  return R
```

Fig. 4.3. Computation of the closure of a set S of states, with respect to a set E of edges.

```
Transitions(E, S, a)
  R ← ∅
  for each state p in S
    loop for each state q such that (p, a, q) is in E
           loop R ← R + {q}
  return R
```

Fig. 4.4. Computation of the transitions by a letter a from states of a set S, with respect to a set E of edges.

A basic use of an automaton consists in testing whether it recognizes some given word. Testing whether y is in the language described by $\boldsymbol{x}$ is implemented by the algorithm of Figure 4.5. The next proposition states the complexity of such a test.

Proposition 4.1. *Given a regular expression $\boldsymbol{x}$, testing whether a word y belongs to $Lang(\boldsymbol{x})$ can be performed in time $O(|\boldsymbol{x}|\,|y|)$ and space $O(|\boldsymbol{x}|)$.*

Proof. The proof is given by algorithm Tester of Figure 4.5 for which we analyze the complexity.

According to Theorem 4.4, the regular-expression-matching automaton $(Q, i, \{t\}, E)$ of $\boldsymbol{x}$ can be built in time and space $O(|\boldsymbol{x}|)$.

Each computation of functions Closure and of Transitions requires time and space $O(\mathrm{card}(Q))$, which is $O(|\boldsymbol{x}|)$ from Theorem 4.2. This is repeated $|y|$ times. This gives $O(|\boldsymbol{x}|\,|y|)$ time. □

```
RegularExpressionTester(x, y)
built the regular-expression-matching automaton (Q, i, {t}, E) of x
C ← Closure(E, {i})
for letter a from first to last letter of y
  loop C ← Closure(E, Transitions(E, C, a))
return t ∈ C
```

Fig. 4.5. Algorithm for testing whether a word y belongs to $\mathit{Lang}(\boldsymbol{x})$, $\boldsymbol{x}$ being a regular expression.

We now come back to our main problem. It is slightly different than the previous one, because the answer to the test has to be reported for each factor of y, and not only on y itself. But no transformation of $\mathcal{E}(\boldsymbol{x})$ is necessary. A mere transformation of the search phase of the algorithm is sufficient: at each iteration of the closure computation, the initial state is integrated to the current set of states. Doing so, each factor of y is tested. Moreover, the "**occurrenceif** $t \in C$" instruction is done at each stage. The entire algorithm is given in Figure 4.6. The following theorem established the complexity of the search phase of the algorithm.

```
RegularExpressionMatcher(x, y)
built the regular-expression-matching automaton (Q, i, {t}, E) of x
C ← Closure(E, {i})
occurrenceif t ∈ C
for letter a from first to last letter of y
  loop C ← Closure(E, Transitions(E, C, a) + {i})
       occurrenceif t ∈ C
```

Fig. 4.6. Algorithm for computing prefixes of a word y that belong to $A^*\mathit{Lang}(\boldsymbol{x})$, $\boldsymbol{x}$ being a regular expression.

Theorem 4.5. *Let $\boldsymbol{x}$ be a regular expression and y be a word. Finding all end-positions of factors of y that are recognized by $\mathcal{E}(\boldsymbol{x})$ can be performed in time $O(|\boldsymbol{x}|\,|y|)$ and space $O(|\boldsymbol{x}|)$. The time spent on each letter of y is $O(|\boldsymbol{x}|)$.*

Proof. See the proof of Proposition 4.1. The second part of the statement comes from that fact: the time spent on each letter of y is linear in the time required by the computations of functions Closure and Transitions. ☐

4.4 Time-space trade-off

The regular-expression-matching problem for a regular expression $\boldsymbol{x}$ and a word y admits a solution based on deterministic automata. It proceeds as follow: build the automaton $\mathcal{E}(\boldsymbol{x})$; built an equivalent deterministic automaton; search with the deterministic automaton. The drawback of this approach is that the deterministic automaton can have a number of states exponential in the length of $\boldsymbol{x}$. This is the situation, for example, when

$$\boldsymbol{x} = \mathsf{a}\,\overbrace{(\mathsf{a}+\mathsf{b})\cdots(\mathsf{a}+\mathsf{b})}^{m-1 \text{ times}}$$

for some $m \geqslant 1$; here, the minimal deterministic automaton recognizing $A^*\mathit{Lang}(\boldsymbol{x})$ has exactly 2^m states since the recognition process has to memorize the last m letters read from the input word y. However, all states of the deterministic automaton for $A^*\mathit{Lang}(\boldsymbol{x})$ are not necessarily met during the search phase. So, a lazy construction of the deterministic automaton during the search is a possible compromise for practical purposes.

5. Matching finite sets of words

5.1 Outline

Problem 5.1. (Dictionary-matching problem.) Given a finite set of words X, the dictionary, preprocess it in order to locate words of X that occur in any given word y.

The classical solution to this problem is due to Aho and Corasick. It essentially consists in a linear-space implementation of a complete deterministic automaton recognizing the language A^*X. The implementation uses both adjacency lists and an appropriate failure function.

Main Theorem 5.1 (Aho and Corasick, 1975). *The dictionary-matching problem for X and y can be achieved in the following terms:*

- *a preprocessing phase on X building an implementation of size $O(|X|)$ of an automaton recognizing A^*X, performed in time $O(|X| \times \log \mathrm{card}(A))$ and $O(\mathrm{card}(X))$ extra-space;*
- *a search phase executing the automaton on y performed in time $O(|y| \times \log \mathrm{card}(A))$ and constant extra-space, the delay being $O(|X| \times \log \mathrm{card}(A))$.*

If we allow more extra-space, the asymptotic time complexities can be reduced. This is achieved, for instance, by using techniques of Section 3 for representing deterministic automata with a sparse matrix, and assuming that $O(|X| \times \mathrm{card}(A))$ space is available. The time complexities of the preprocessing and search phases are respectively reduced to $O(|X|)$ and $O(|y|)$, and the delay to $O(|X|)$. Nevertheless, notice that the times complexities given in the above theorem are still linear in $|X|$ or $|y|$ if we consider fixed alphabets.

The method behind Theorem 5.1 is based on a specific automaton recognizing A^*X: its states are the prefixes of words in X (their number is finite as X is). The automaton is not minimal in the general case. It is presented in Section 5.2, and its implementation with a failure function is given in Section 5.3. Section 5.4 is devoted to the search for X with the automaton.

5.2 Dictionary-matching automata

We give a complete deterministic automaton that recognizes A^*X. In order to formalize this automaton, we introduce for each language U the mapping $h_U : A^* \to \mathit{Pref}(U)$ defined for each word v by

$$h_U(v) = \text{the longest suffix of } v \text{ that belongs to } \mathit{Pref}(U).$$

(In the whole Section 5, U refers to an ordinary language, and X refers to a finite language.)

Proposition 5.1. *Let X be a finite language. Then the automaton*

$$\Big(\mathit{Pref}(X),\ \ \varepsilon,\ \ \mathit{Pref}(X) \cap A^*X,\ \ \big\{(p, a, h_X(pa)) \mid p \in \mathit{Pref}(X), a \in A\big\}\Big)$$

recognizes the language A^*X. This automaton is deterministic and complete.

In the following, we denote by $\mathcal{D}(X)$ the automaton of Proposition 5.1 applied to X, and we call it the *dictionary-matching automaton* of X.

The proof of Proposition 5.1 relies on the following result.

Lemma 5.1. *Let $U \subseteq A^*$. Then*

(i) $v \in A^*U$ *iff* $h_U(v) \in A^*U$, *for each* $v \in A^*$.

Furthermore, h_U satisfies the relations:

(ii) $h_U(\varepsilon) = \varepsilon$;
(iii) $h_U(va) = h_U(h_U(v)a)$, *for each* $(v, a) \in A^* \times A$.

Proof. If $v \in A^*U$, then v is in the form wu where $w \in A^*$ and $u \in U$; by definition of h_U, u is necessarily a suffix of $h_U(v)$; therefore $h_U(v) \in A^*U$. Conversely, if $h_U(v) \in A^*U$, we have also $v \in A^*U$, because $h_U(v)$ is a suffix of v. Which proves (i).

Property (ii) clearly holds.

It remains to prove (iii). Both words $h_U(va)$ and $h_U(v)a$ are suffixes of va, and therefore one of them is a suffix of the other. Then we distinguish two cases according to which word is a suffix of the other.

First case: $h_U(v)a$ is a proper suffix of $h_U(va)$ (hence $h_U(va) \neq \varepsilon$). Consider the word w defined by $w = h_U(va)a^{-1}$. Thus we have: $h_U(v)$ is a proper suffix of w, w is a suffix of v, and since $h_U(va) \in$

$Pref(U)$, $w \in Pref(U)$. Whence w is a suffix of v that belongs to $Pref(U)$, but strictly longest than $h_U(v)$. This contradicts the maximality of $|h_U(v)|$. So this case is impossible.

Second case: $h_U(va)$ is a suffix of $h_U(v)a$. Then, $h_U(va)$ is a suffix of $h_U(h_U(v)a)$. Now, since $h_U(v)a$ is a suffix of va, $h_U(h_U(v)a)$ is a suffix of $h_U(va)$. Both properties implies $h_U(va) = h_U(h_U(v)a)$, and the expected result follows. □

Proof of Proposition 5.1. Let $v \in A^*$. It follows from properties (ii) and (iii) of Lemma 5.1 that

$$\big(h_X(v_1v_2\cdots v_{j-1}),\ \ v_i,\ \ h_X(v_1v_2\cdots v_j)\big)_{1\leqslant j\leqslant |v|}$$

is a path labeled by v from the initial state ε to the state $h_X(v)$.

If $v \in A^*X$, we get $h_X(v) \in A^*X$ from (i) of Lemma 5.1; which shows that $h_X(v)$ is a terminal state, and finally that v is recognized by the automaton.

Conversely, if v is recognized by the automaton, we have $h_X(v) \in A^*X$ by definition of the automaton. This implies that $v \in A^*X$ from (i) of Lemma 5.1 again. □

We show how to implement the automaton $\mathcal{D}(X)$ in the next section.

5.3 Linear dictionary-matching automata

The automaton $\mathcal{D}(X)$ is implemented with a failure function. The aim is to get a representation that does not depend on the size of the alphabet.

For each language U, let $f_U\colon Pref(U) \to Pref(U)$ be the function defined for each nonempty word u in $Pref(U)$ by

$$f_U(u) = \text{the longest proper suffix of } u \text{ that belongs to } Pref(U).$$

Lemma 5.2. *Let $U \subseteq A^*$. For each $(u,a) \in Pref(U) \times A$, we have:*

$$h_U(ua) = \begin{cases} ua, & \text{if } ua \in Pref(U), \\ h_U(f_U(u)a), & \text{if } u \neq \varepsilon \text{ and } ua \notin Pref(U), \\ \varepsilon, & \text{otherwise.} \end{cases}$$

Proof. The identity clearly holds when $ua \in Pref(U)$ or when $ua \notin Pref(U)$ but $u = \varepsilon$.

It remains to examine the case where $ua \notin Pref(U)$ and $u \neq \varepsilon$. Here, $f_U(u)a$ is a proper suffix of ua. What is more, $h_U(f_U(u)a)$ is the longest suffix of ua that belongs to $Pref(U)$. Indeed, if we assume the existence of a suffix v of ua satisfying $v \in Pref(U)$ and $|v| \geqslant |f_U(v)a|$, we get that va^{-1} is a proper suffix of u belonging to $Pref(U)$; then $va^{-1} = f_U(u)$ because of the maximality of $|f_U(u)|$. Which achieves the proof. □

We introduce for each language U the function $\gamma_U\colon Pref(U) \times A \to Pref(U)$ associating with each $(u,a) \in Pref(U) \times A$ such that $ua \in Pref(U)$ the word ua. Thus, with conventions of Section 3.4, we have:

Proposition 5.2. *For each finite language X, the couple (γ_X, f_X) represents the transition function of $\mathcal{D}(X)$; function γ_X is a subtransition function and function f_x a failure function, according to the transition function of $\mathcal{D}(X)$.*

Proof. Follows from Lemma 5.2. □

Now, let us observe that function γ_X is exactly the transition function of the deterministic automaton

$$\Big(Pref(X),\ \ \varepsilon,\ \ X,\ \ \big\{(p,a,pa) \mid p \in Pref(X), a \in A, pa \in Pref(X)\big\}\Big).$$

This automaton recognizes the language X, and is classically called the *trie* of X, as a reference to "information re*trie*val". It is built by function Trie of Figure 5.1.

Proposition 5.3. *Function* Trie *applied to any finite language X builds the trie of X. If the edges of the automaton are implemented via adjacency lists, the size of the trie is $O(|X|)$, and the construction is performed in time $O(|X| \times \log d)$ within constant extra-space, d being the maximum degree of states.*

```
TRIE(X)
    let δ be the transition function of (Q, i, T, E)
    (Q, T, E) ← (∅, ∅, ∅)
    i ← STATE-CREATION
    for word x from first to last word of X
      loop t ← i
           for letter a from first to last letter of x
             loop q ← δ(t, a)
                  if q = NIL
                    then q ← STATE-CREATION
                         E ← E + {(t, a, q)}
                  t ← q
           T ← T + {t}
    return (Q, i, T, E)
```

Fig. 5.1. Construction of the trie of a finite set of words X.

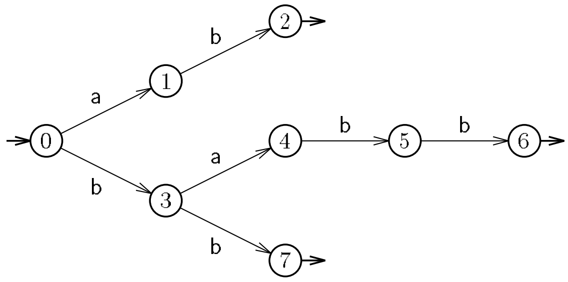


Fig. 5.2. The trie of $\{\mathsf{ab}, \mathsf{babb}, \mathsf{bb}\}$.

When $X = \{\mathsf{ab}, \mathsf{babb}, \mathsf{bb}\}$, the trie of X is as depicted in Figure 5.2. This example shall be considered twice in the following.

To achieve the goal of implementing $\mathcal{D}(X)$ in linear size, we use Proposition 5.2. Then, it remains to give methods for computing f_X and for marking the set of terminal states. This can be done by a breadth first search on the graph underlying the trie starting at the initial state, as shown by the two following lemmas.

Lemma 5.3. *Let $U \subseteq A^*$. For each $(u, a) \in \mathit{Pref}(U) \times A$, we have:*

$$f_U(ua) = \begin{cases} h_U(f_U(u)a), & \text{if } u \neq \varepsilon, \\ \varepsilon, & \text{otherwise.} \end{cases}$$

Proof. Similar to the proof of Lemma 5.2. □

Lemma 5.4. *Let $U \subseteq A^*$. For each $u \in \mathit{Pref}(U)$, we have:*

$$u \in A^*U \iff (u \in U) \text{ or } (u \neq \varepsilon \text{ and } f_U(u) \in A^*U).$$

Proof. It is clearly sufficient to prove that

$$u \in (A^*U) \backslash U \implies f_U(u) \in A^*U.$$

So, let $u \in (A^*U) \backslash U$. The word u is in the form vw where $v \in A^*$ and w is a proper suffix of u belonging to U. Then, by definition of f_U, w is a suffix of $f_U(u)$. Therefore $f_U(u) \in A^*U$. Which ends the proof. □

The complete function constructing the representation of $\mathcal{D}(X)$ with the subtransition γ_X and the failure function f_X is given in Figure 5.3. Let us recall that the transition function δ of $\mathcal{D}(X)$ is assumed to be computed by function FAILUREFUNCTION-TRANSITION of Section 3.4. The next theorem states the correctness of the construction and its time and space complexities. We call this representation of $\mathcal{D}(X)$ the *linear dictionary-matching automaton* of X. The term "linear" (in $|X|$ is understood) is suitable if we work with a fixed alphabet, since degrees are upper-bounded by $\mathrm{card}(A)$.

LinearDictionaryMatchingAutomaton(X)
    let γ *be the transition function of* (Q, i, T, E')
    let δ *be the transition function of* $(Q, i, T, (\gamma, F))$
1 $(Q, i, T, E') \leftarrow$ Trie(X)
2 $F[i] \leftarrow$ nil
3 $\vartheta \leftarrow$ EmptyQueue
4 Enqueue(ϑ, i)
5 **while** not QueueIsEmpty(ϑ)
6   **loop** $p \leftarrow$ Dequeue(ϑ)
7   **for** each letter a such that $\gamma(p, a) \neq$ nil
8   **loop** $q \leftarrow \gamma(p, a)$
9          $F[q] \leftarrow \delta(F[p], a)$
10   **if** $F[q]$ is in T
11   **then** $T \leftarrow T + \{q\}$
12   Enqueue(ϑ, q)
13 **return** $(Q, i, T, (\gamma, F))$

Fig. 5.3. Construction of the linear dictionary-matching automaton of a finite set of words X.

Theorem 5.2. *The linear dictionary-matching automaton of any finite language X is built by function* LinearDictionaryMatchingAutomaton*. The size of this representation of $\mathcal{D}(X)$ is $O(|X|)$. The construction is performed in time $O(|X| \times \log d)$ within $O(\mathrm{card}(X))$ extra-space, d being the maximum degree of states of the trie of X.*

Proof. The correctness of the function and the order of the size of the representation is consecutive to Propositions 5.1, 5.2, and 5.3, and Lemmas 5.2, 5.3, and 5.4. The extra-space is linear in the size of the queue ϑ, which has always less than $\mathrm{card}(X)$ elements.

In order to prove the announced time complexity, we shall see that the last test of the loop of function FailureFunction-Transition (for computing $\delta(F[p], a)$; see Section 3.4) is executed less than $2|X|$ times. To avoid ambiguity, the state variable p of function FailureFunction-Transition is renamed r.

First. We remark that less tests are executed on the trie than if the words of X were considered separately.

Second. Considering separately each word x of X, and assimilating variables p and r with the prefixes of x they represent, the quantity $2|p| - |r|$ grows of at least one unity between two consecutive tests "$\gamma(r, a) =$ nil". When $|x| \leqslant 1$, no test is performed. But when $|x| \geqslant 2$, this quantity is equal to 2 before the execution of the first test ($|p| = 1$, $|r| = 0$), and is less than $2|x| - 2$ after the execution of the last test ($|p| = |x| - 1$, $|r| \geqslant 0$); which shows that less than $2|x| - 3$ tests are executed in this case.

This proves the expected result on the number of tests.

Now, since each of these tests is performed in time $O(\log d)$, the loop of lines 5–12 of function LinearDictionaryMatchingAutomaton is performed in time $O(|X| \times \log d)$. This is also the time complexity of the whole function, since line 1 is also performed in time $O(|X| \times \log d)$ according to Proposition 5.3. □

Figure 5.4 displays the linear dictionary-matching automaton of X when $X = \{\mathsf{ab}, \mathsf{babb}, \mathsf{bb}\}$. The failure function f_X is depicted with non-labeled discontinuous edges.

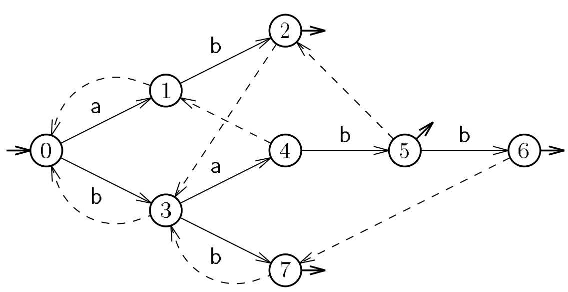


Fig. 5.4. The linear dictionary-matching automaton of $\{\mathsf{ab}, \mathsf{babb}, \mathsf{bb}\}$.

To be complete, we add that f_U can be expressed independently of h_U for any language U.

Lemma 5.5. *Let $U \in A^*$. For each $(u,a) \in \mathit{Pref}(U) \times A$, we have:*

$$f_U(ua) = \begin{cases} f_U(u)a, & \text{if } u \neq \varepsilon \text{ and } ua \in \mathit{Pref}(U), \\ f_U(f_U(u)a), & \text{if } u \neq \varepsilon \text{ and } ua \notin \mathit{Pref}(U), \\ \varepsilon, & \text{if } u = \varepsilon. \end{cases}$$

Proof. This follows from Lemmas 5.2 and 5.3. □

However interesting this result is, it does not lead to another computation of linear dictionary-matching automata than the computation performed by the function of Figure 5.3.

5.4 Searching with linear dictionary-matching automata

We prove in this section that matching a finite set of words can be performed in linear time on fixed alphabets. This is stated in the following theorem.

Theorem 5.3. *Let X be a finite set of words and y be a word. Let ℓ be the maximum length of words of X and d be the maximum degree of states of the trie of X. Using the linear dictionary-matching automaton of X, searching for all occurrences of words of X as factors of y (search phase of algorithm* MATCHER*) is performed in time $O(|y| \times \log d)$, constant extra-space, within a delay of $O(\ell \times \log d)$.*

Proof. The proof is similar to the proof of Theorem 5.2.

Here, instead of the quantity $2|p| - |r|$, we consider the quantity $2|y'| - |p|$ where y' is the already read prefix of y. We obtain that less than $2|y| - 1$ tests "$\gamma(p,a) = \text{NIL}$" are executed. This proves that the total time is $O(|y| \times \log d)$. For the delay, the test "$\gamma(p,a) = \text{NIL}$" cannot be executed strictly more than ℓ times on each input letter a, which gives a time $O(\ell \times \log d)$. □

The search phase can be improved to prevent unnecessary calls to the failure function as far as it is possible.

Assume for instance that during the search state 5 of Figure 5.4 has been reached, and that the next letter of the input word, say c, is not b. The failure function has to be iterated at least twice since neither $\gamma_X(5,c)$ nor $\gamma_X(2,c)$ are defined. It is clear that the test on state 2 is useless, whatever c is. The next attempt is to compute $\gamma_X(3,c)$. Here, state 3 plays its role because c might be equal to a. But now, if $\gamma_X(3,c)$ is undefined, it is needless to iterate again the failure function on state 0, since c is then neither a nor b.

Following a similar reasoning for each states of the linear dictionary-matching automaton of X when $X = \{\mathsf{ab}, \mathsf{babb}, \mathsf{bb}\}$ leads to consider the representation depicted in Figure 5.5.

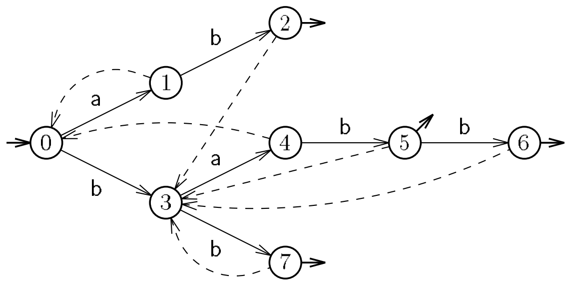


Fig. 5.5. The optimized representation of $\mathcal{D}(\{\mathsf{ab}, \mathsf{babb}, \mathsf{bb}\})$.

More generally, given a finite language X and the failure function f_X, the representation of $\mathcal{D}(X)$ can be optimized by considering another failure function, denoted here by $\hat{f}_X$. Introducing the notation $\mathit{Follow}_U(u)$ to denote the set defined for each language U and for each word u in $\mathit{Pref}(U)$ by

$$\mathit{Follow}_U(u) = \{a \mid a \in A, ua \in \mathit{Pref}(U)\},$$

it is set that

$$\hat{f}_X(p) = \begin{cases} f_X(p), & \text{if } p \neq \varepsilon \text{ and } \mathit{Follow}_X(f_X(p)) \not\subseteq \mathit{Follow}_X(p), \\ \hat{f}_X(f_X(p)), & \text{if } p \neq \varepsilon \text{ and } \mathit{Follow}_X(f_X(p)) \subseteq \mathit{Follow}_X(p), \\ \text{undefined}, & \text{otherwise}, \end{cases}$$

for each $p \in \mathit{Pref}(X)$. The couple $(\gamma_X, \hat{f}_X)$ represents clearly the transition function of $\mathcal{D}(X)$. New failure states can be computed during a second breadth first search, and this can be done directly on array F_X.

However, substituting $\hat{f}_X$ to f_X does not affect the maximum delay of the searching algorithm that still remains $O(l \times \log d)$. To show this point, we give a worst case example. Let $\varphi(m)$ be the language defined for each $m \geqslant 1$ by:

$$\varphi(m) = \{\mathsf{a}^{m-1}\mathsf{b}\} \cup \{\mathsf{a}^{2j-1}\mathsf{ba} \mid 1 \leqslant j < \lceil m/2 \rceil\} \cup \{\mathsf{a}^{2j}\mathsf{bb} \mid 0 \leqslant j < \lfloor m/2 \rfloor\}.$$

If $X = \varphi(m)$ for some $m \geqslant 1$, and if $\mathsf{a}^{m-1}\mathsf{bc}$ is the already read prefix of the input, m accesses to the failure function of the linear dictionary-matching automaton of X are made when reading letter c, whatever function f_X or $\hat{f}_X$ is chosen. (See the example given in Figure 5.6.)

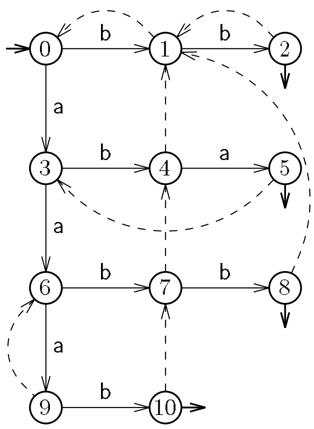


Fig. 5.6. The optimized representation of $\mathcal{D}(\varphi(4))$.

6. Matching words

6.1 Outline

Problem 6.1. (String-matching problem.) Given a word x, preprocess it in order to locate all its occurrences in any given word y.

Let us first observe that this problem can be viewed as a particular case of the dictionary-matching problem (see Section 5). Here, the dictionary has only one element. Moreover, the dictionary-matching automaton $\mathcal{D}(\{x\})$, which recognizes the language A^*x, has the minimum number of states required to recognize A^*x, *i.e.* $|x|+1$ states. Therefore, the minimal automaton recognizing A^*x, denoted by $\mathcal{M}(A^*x)$, can be identified with $\mathcal{D}(\{x\})$. Since the maximum degree of states of the trie of $\{x\}$ is upper-bounded by one, implementing this automaton with the help of the optimized failure function described Section 5.4 leads to the following results.

Theorem 6.1 (Knuth, Morris, and Pratt, 1977). *The string-matching problem for x and y can be performed in time $O(|x|+|y|)$ and space $O(|x|)$, the delay being $\Theta(\log|x|)$ in the worst-case.*

We just have to hark back to the order of the delay for the algorithm of Knuth, Morris and Pratt. It is proved that the number of times the transition function of the trie of $\{x\}$ is performed on any input letter cannot exceed $\lfloor \log_\Phi(|x|+1) \rfloor$ where $\Phi = \big(1+\sqrt{5}\big)/2$ is the golden ratio. This upper bound is a consequence of a combinatorial property of words due to Fine and Wilf (known as the "periodicity lemma"). But it is closed to the worst-case bound, obtained when x is a prefix of the infinite Fibonacci word (see Chapter "Combinatorics of words").

However, as we shall see, implementing $\mathcal{M}(A^*x)$ with adjacency lists solves the string-matching problem with the additional feature of having a real-time search phase on fixed alphabets, *i.e.* with a delay bounded by a constant.

Main Theorem 6.2. *The string-matching problem for x and y can be achieved in the following terms:*

- *a preprocessing phase on x building an implementation of $\mathcal{M}(A^*x)$ of size $O(|x|)$, performed in time $O(|x|)$ and constant extra-space;*
- *a search phase executing the automaton on y performed in time $O(|y|)$ and constant extra-space, the delay being $O(\log \min\{1 + \lfloor \log_2 |x| \rfloor, \mathrm{card}(A)\})$.*

Underlying the above result are indeed optimal bounds on the complexity of string-matching algorithms for which the search phase is on-line with a one-letter buffer. Relaxing the on-line condition leads to another theorem stated below. But its proof is based on combinatorial properties of words unrelated to automata and not considered in this chapter.

Theorem 6.3 (Galil and Seiferas, 1983). *The string-matching problem for x and y previously stored in memory can be performed in time $O(|x|+|y|)$ and constant extra-space.*

In Section 6.2, we describe an on-line construction of $\mathcal{M}(A^*x)$. The linear implementation via adjacency lists is discussed in Section 6.3. We establish in Section 6.4 properties of $\mathcal{M}(A^*x)$ that are used in Section 6.5 to prove the asymptotic bounds of the search phase claimed in Theorem 6.2.

6.2 String-matching automata

We give a method to build the automaton $\mathcal{M}(A^*x)$. The feature of this method is that it is based on an on-line construction and that it does not use the usual procedures of determinization and minimization of automata.

In the remainder of Section 6 we identify $\mathcal{M}(A^*x)$ with $\mathcal{D}(\{x\})$, which is the automaton

$$\Big(\mathit{Pref}(x),\ \ \varepsilon,\ \ \{x\},\ \ \big\{(p, a, h_x(pa)) \mid p \in \mathit{Pref}(x), a \in A\big\}\Big),$$

$h_x(v)$ being the longest suffix of v which is a prefix of x, for each $v \in A^*$. We call this automaton the *string-matching automaton* of x.

An example of string-matching automaton is given in Figure 2.1: the depicted automaton is $\mathcal{M}(A^*\mathsf{abaaab})$ assuming that $A = \{\mathsf{a}, \mathsf{b}\}$.

We introduce the notions of "border" as follows. A word v is said to be a *border* of a word u if v is both a prefix and a suffix of u. The longest proper border of a nonempty word u is said to be *the border* of u and is denoted by $\mathit{Bord}(u)$. As a consequence of definitions, we have:

$$h_x(pa) = \begin{cases} pa, & \text{if } pa \text{ is a prefix of } x, \\ \mathit{Bord}(pa), & \text{otherwise,} \end{cases}$$

for each $(p,a) \in \mathit{Pref}(x) \times A$.

In order to build $\mathcal{M}(A^*x)$, the construction of the set of edges of the string-matching automaton of x is to be settled. The construction is on-line, as suggested by the following lemma.

Lemma 6.1. *Let us denote by E_u the set of edges of $\mathcal{M}(A^*u)$ for any $u \in A^*$. We have:*

$$E_\varepsilon = \big\{(\varepsilon, b, \varepsilon) \mid b \in A\big\}.$$

Furthermore, for each $(u,a) \in A^* \times A$ we have:

$$E_{ua} = E'_{ua} \cup E''_{ua}$$

with

$$E'_{ua} = \Big(E_u \backslash \{(u, a, h_u(ua))\}\Big) \cup \{(u, a, ua)\}$$

and

$$E''_{ua} = \{(ua, b, w) \mid (h_u(ua), b, w) \in E'_{ua}\}.$$

Proof. The property for E_ε clearly holds.

Now, let $u \in A^*$ and $a \in A$, let E'_{ua} and E''_{ua} be as in the lemma, and set $v = h_u(ua)$.

Each edge in E_{ua} outgoing a state no longer than $|u|$ belongs to E'_{ua}. The converse is also true.

It remains to prove that each edge in E_{ua} outgoing state ua belongs to E''_{ua}, and that the converse holds. This is to prove that for each $b \in A$, the targets w and w' of the edges (v, b, w) and (ua, b, w'), both in E_{ua}, are identical.

Since v is a border of ua, w is both a suffix of uab and a prefix of ua. Which implies that w is shorter than w'.

Conversely. We have that $|w'| \leqslant |vb|$. (Assuming the contrary leads to consider that $w'b^{-1}$ is a border of ua contradicting the maximality of v.) Since w' and vb are both suffixes of uab, w' is a suffix of vb. Now w' is also a prefix of ua. This shows that w' is shorter than w, and ends the proof. □

The construction of $\mathcal{M}(A^*ua)$ from $\mathcal{M}(A^*u)$ can be interpreted in a visual point of view as the "unfolding" of the edge $(u, a, h_u(ua))$ of the automaton $\mathcal{M}(A^*u)$. An example is given in Figure 6.1 that depicts four steps related to the construction of $\mathcal{M}(A^*\mathsf{abaaab})$.

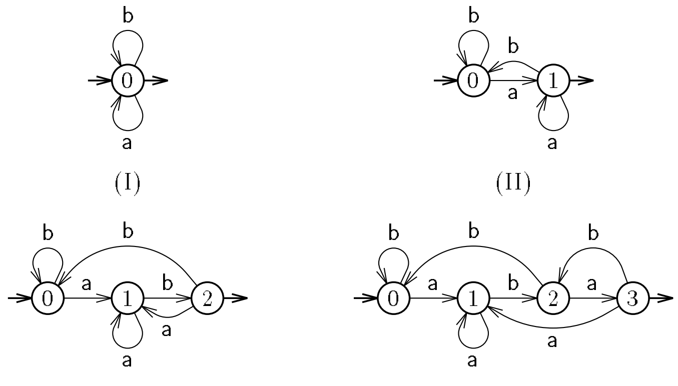


Fig. 6.1. During the construction of the string-matching automaton of abaaab, unfolding of the edge $(\varepsilon, \mathsf{a}, \varepsilon)$ from step "ε" (I) to step "a" (II), of the edge $(\mathsf{a}, \mathsf{b}, \varepsilon)$ from step "a" to step "ab" (III), and of the edge $(\mathsf{ab}, \mathsf{a}, \mathsf{a})$ from step "ab" to step "aba" (IV). It is assumed that $A = \{\mathsf{a}, \mathsf{b}\}$.

A function that builds the string-matching automaton of x following the method suggested by Lemma 6.1 is given in Figure 6.2. This can be used straightforwardly to implement the automaton via its transition matrix. Following the same scheme, we describe in the next section an implementation of the string-matching automaton of x which size is both linear in $|x|$ and independent of the alphabet.

6.3 Linear string-matching automata

We show in this section that implementing string-matching automata via adjacency lists gives representations that are time-linear and space-linear in the length of the pattern. Indeed, the property comes from the fact: with ε as default state in the adjacency lists (see Section 3.2), the total length of these lists is linear. This representation reduces the automaton to its significant part. Another way of saying it, is to consider "significant edges" as follows.

An edge (p, a, q) of a given string-matching automaton is *significant* if $q \neq \varepsilon$, and *null* otherwise; if the edge is significant, it is *forward* if $q = pa$ and *backward* otherwise.

```
StringMatchingAutomaton(x)
    let δ be the transition function of (Q, i, ∅, E)
    (Q, E) ← (∅, ∅)
    i ← State-Creation
    for each letter b in A
        loop E ← E + {(i, b, i)}
    t ← i
    for letter a from first to last letter of x
        loop r ← δ(t, a)
                q ← State-Creation
                E ← E − {(t, a, r)} + {(t, a, q)}
                for each letter b in A
                    loop E ← E + {(q, b, δ(r, b))}
                t ← q
    return (Q, i, {t}, E)
```

Fig. 6.2. Construction of the string-matching automaton of a word x.

Picking up again the case $x = \mathtt{abaaab}$, the string-matching automaton of x has 6 forward edges and 5 backward edges (see the automaton given in Figure 2.1).

Proposition 6.1. *The number of significant edges of the string-matching automaton of any word x is upper-bounded by $2|x|$; more precisely, its number of forward edges is exactly $|x|$, and its number of backward edges is upper-bounded by $|x|$. The bounds are reached for instance when the first letter of x occurs only at the first position in x.*

In order to prove Proposition 6.1, we shall establish the following result.

Lemma 6.2. *Let (p, a, q) and (p', a', q') be two distinct backward edges of the string-matching automaton of some word u. Then $|p| - |q| \neq |p'| - |q'|$.*

Proof. Suppose for a contradiction the existence of two distinct backward edges (p, a, q) and (p', a', q') of $\mathcal{M}(A^*u)$ satisfying $|p| - |q| = |p'| - |q'|$.

In case $p = p'$, we have that $q = q'$. Since the two edges are significant, this implies that $a = a'$. Which is impossible.

Thus, we can assume without loss of generality that $|p| > |p'|$, thus, $|q| > |q'|$. Since qa^{-1} is a border of p ($|p| - |qa^{-1}|$ is a period of p) and since q' is a proper prefix of q, we have

$$a' = p_{|q'|} = p_{|q'|+|p|-|qa^{-1}|} = p_{|q'|+|p'|-|q'|+1} = p_{|p'|+1}.$$

Which contradicts the fact that (p', a', q') is a backward edge. □

Proof of Proposition 6.1. The number of forward edges of the automaton is obviously $|x|$.

Let us prove the upper bound on the number of backward edges. Since the number $|p| - |q|$ associated to the backward edge (p, a, q) ranges from 0 to $|x| - 1$, Lemma 6.2 implies that the total number of backward edges is bounded by $|x|$.

We show that the upper bound on the number of backward edges is optimal. Consider that the first letter of x occurs only at the first position in x. The edge (p, x_1, x_1) is an outgoing edge for each state p of non-zero length of the automaton, and this edge is a backward edge. So, the total number of backward edges is $|x|$ in this case. □

Figure 6.3 displays a string-matching automaton which number of significant edges is maximum for a word of length 7.

From the previous proposition, an implementation of $\mathcal{M}(A^*x)$ via adjacency lists with the initial state ε as uniform default state has a size linear in $|x|$, since the edges represented in the adjacency lists are the significant edges of the automaton. We call this representation of the string-matching automaton of x the *linear string-matching automaton* of x. It is constructed by the function given in Figure 6.4. This function is a mere adaptation of the general function given in Figure 6.2. Recall that the transition function of the linear string-matching automaton of x is assumed to be computed by function AdjacencyLists-Transition of Section 3.2.

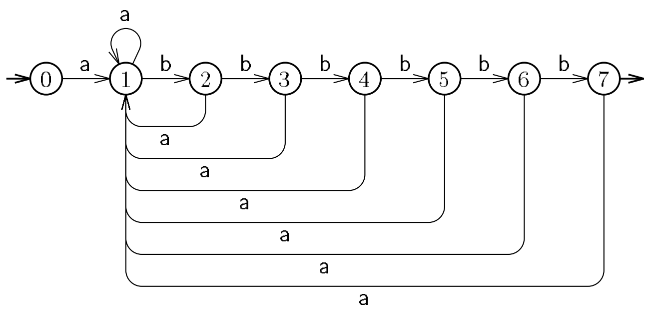


Fig. 6.3. A string-matching automaton with the maximum number of significant edges. The significant edges are the only depicted edges; the target of other edges is 0.

```
LinearStringMatchingAutomaton(x)
     let δ be the transition function of (Q, i, ∅, G)
 1   Q ← ∅
 2   i ← State-Creation
 3   G[i] ← ∅
 4   t ← i
 5   for letter a from first to last letter of x
 6       loop r ← δ(t, a)
 7            q ← State-Creation
 8            if r ≠ i
 9               then G[t] ← G[t] − {(a, r)}
10            G[t] ← G[t] + {(a, q)}
11            G[q] ← G[r]
12            t ← q
13   return (Q, i, {t}, G)
```

Fig. 6.4. Construction of the linear string-matching automaton of a word x.

Theorem 6.4. *Function* LinearStringMatchingAutomaton *builds the linear string-matching automaton of any given word x. The size of this representation of $\mathcal{M}(A^*x)$ is $O(|x|)$. The construction is performed in time $O(|x|)$ and constant extra-space.*

Proof. The correctness of the function is consecutive to Lemma 6.1. The order of the size of the representation follows from Proposition 6.1.

The time required to build the set of all significant edges outgoing a given state is linear in their number (the operations executed on the adjacency list associated to a given state $p \neq x$ are the operations occurring in Figure 6.4 at line 3 if $p = i$ and at line 11 otherwise, then at line 9 if necessary, then finally at line 10; the corresponding operations for state x are at line 3 if $x = \varepsilon$ and at line 11 otherwise). Hence, the total time is $O(|x|)$ from Proposition 6.1. □

We show in Section 6.5 that the linear representation of the string-matching automaton of x described above yields a search for occurrences of x in y that runs in time linear in $|y|$. Before that, we establish combinatorial properties of string-matching automata in the next section.

6.4 Properties of string-matching automata

We establish in this section some upper bounds for the number of significant edges of string-matching automata. These bounds complete the global bound given in Proposition 6.1, by focusing on the number of outgoing significant edges. The two main results, namely Propositions 6.2 and 6.3, are intensively used in Section 6.5.

Given a word u, we denote by $se_u(p)$ the number of significant edges outgoing the state p of the string-matching automaton of u; if p is a prefix of u and q a prefix of p, the notation $se_u(p,q)$ stands for the number of significant edges which sources range in the set of prefixes of u from q to p, *i.e.* the number

$$se_u(q) + se_u(q \cdot p_{|q|+1}) + \cdots + se_u(q \cdot p_{|q|+1} \cdots p_{|p|-1}) + se_u(p).$$

The next two lemmas provide recurrence relations satisfied by the numbers $se_u(p)$. The expressions are stated using the following notation: given a predicate e, the integer denoted by $\chi(e)$ has value 1 when e is true, and value 0 otherwise.

Lemma 6.3. *Let $(u,a) \in A^* \times A$. For each $v \in Pref(ua)$, we have:*

$$se_{ua}(v) = \begin{cases} se_u(Bord(ua)), & \text{if } v = ua, \\ se_u(u) + \chi(Bord(ua) = \varepsilon), & \text{if } v = u, \\ se_u(v), & \text{otherwise.} \end{cases}$$

Proof. This is a straightforward consequence of Lemma 6.1. □

Lemma 6.4. *Let $u \in A^+$. For each $v \in Pref(u)$, we have:*

$$se_u(v) = \begin{cases} se_u(Bord(u)), & \text{if } v = u, \\ se_u(Bord(v)) + \chi(Bord(va) = \varepsilon), & \text{if } va \in Pref(u) \\ & \quad \text{for some } a \in A, \\ 1, & \text{if } v = \varepsilon. \end{cases}$$

Proof. Follows from Lemma 6.3. □

The next lemma is the "cornerstone" of the proof of the logarithmic bound given in Proposition 6.2 stated afterwards.

Lemma 6.5. *Let $u \in A^+$. For each $v \in Pref(u)\backslash\{\varepsilon\}$, we have:*

$$2|Bord(v)| \geqslant |v| \quad \Longrightarrow \quad se_u(Bord(v)) = se_u(Bord^2(v)).$$

Proof. Set $k = 2|Bord(v)| - |v|$, $w = v_1 v_2 \cdots v_k$, and $a = v_{k+1}$. Since wa is a proper border of $Bord(v)a$, the border of $Bord(v)a$ is nonempty. Then we apply Lemma 6.4 to the proper prefix $Bord(v)$ of u. □

Proposition 6.2. *Let $u \in A^*$. For each state p of $\mathcal{M}(A^*u)$, we have:*

$$se_u(p) \leqslant 1 + \lfloor \log_2(|p|+1) \rfloor.$$

Proof. We prove the result by induction on $|p|$. From Lemma 6.4, this is true if $|p| = 0$. Next, suppose $|p| \geqslant 1$.

Let j be the integer such that

$$2^j \leqslant |p| + 1 < 2^{j+1},$$

then let k be the integer such that

$$|Bord^{k+1}(p)| + 1 < 2^j \leqslant |Bord^k(p)| + 1.$$

Let $\ell \in \{0, \ldots, k-1\}$; we have $2|Bord^{\ell+1}(p)| \geqslant 2^{j+1} - 2 \geqslant |p| \geqslant |Bord^{\ell}(p)|$; which implies $se_u(Bord^{\ell+1}(p)) = se_u(Bord^{\ell+2}(p))$ from Lemma 6.5. Hence we get the equality

$$se_u(Bord(p)) = se_u(Bord^{k+1}(p)).$$

From the induction hypothesis applied to the state $Bord^{k+1}(p)$, we get

$$se_u(Bord^{k+1}(p)) \leqslant 1 + \lfloor \log_2(|Bord^{k+1}(p)| + 1) \rfloor.$$

Now Lemma 6.4 implies

$$se_u(p) \leqslant se_u(Bord(p)) + 1.$$

This shows that

$$se_u(p) \leqslant j + 1 = 1 + \lfloor \log_2(|p|+1) \rfloor,$$

and ends the proof. □

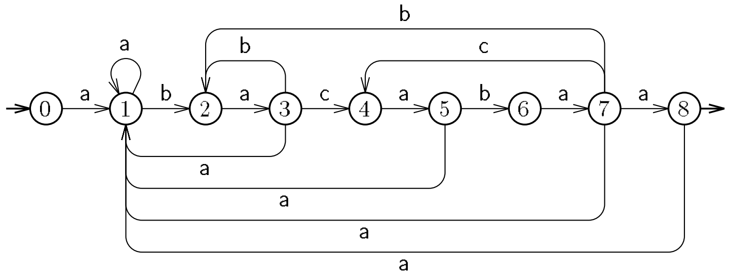


Fig. 6.5. The string-matching automaton of abacabad without its null edges.

By way of illustration, we consider the case where $x =$ abacabad. Given a state p of $\mathcal{M}(A^*x)$ (see Figure 6.5), the $1 + \lfloor \log_2(|p| + 1) \rfloor$ bound for the number of significative edges outgoing p is reached when $|p| = 0$, 1, 3, or 7.

Proposition 6.3. *Let $u \in A^*$. For each backward or null edge (p, a, q) of $\mathcal{M}(A^*u)$, we have:*

$$se_u(p, q) \leqslant 2|p| - 2|q| + 2 - \chi(p = u) - \chi(q = \varepsilon).$$

Proof. The property clearly holds when u is the power of some letter. The remainder of the proof is by induction.

So, let $u \in A^*$, $b \in A$, and let (p, a, q) be a backward or null edge of $\mathcal{M}(A^*ub)$.

If $|p| \leqslant |u|$, (p, a, q) is also an edge of $\mathcal{M}(A^*u)$. By application of Lemma 6.3, we obtain that

$$se_{ub}(p, q) \leqslant se_u(p, q) + \chi(p = u).$$

Otherwise $p = ub$. Let r be the border of ub. We only have to examine the case where r is a proper prefix of u (if $r = u$, then $u \in b^*$). Thus (r, a, q) is an edge of $\mathcal{M}(A^*u)$ and of $\mathcal{M}(A^*ub)$. If it is a forward edge, *i.e.* if $q = ra$, we obtain from Lemma 6.3 that

$$se_{ub}(p, q) = se_u(u, r) + \chi(r = \varepsilon),$$

and if it is a backward edge we obtain that

$$se_{ub}(p, q) = se_u(r, q) + se_u(u, r) + \chi(r = \varepsilon).$$

The result now follows by applying of the induction hypothesis to u. ◻

The previous result is illustrated by the example given in Figure 6.3, *i.e.* when $x =$ abbbbbb. In this case, the $2|p| - 2|q| + 2 - \chi(p = x) - \chi(q = \varepsilon)$ bound is reached for any backward or null edge of $\mathcal{M}(A^*x)$.

Observe that Proposition 6.3 provides another proof of the $2|x|$ bound given in Proposition 6.1 as follows. We consider a null edge outgoing state x (possibly extending the alphabet by one letter). For this edge, with the notation of Proposition 6.3, we have $p = u = x$ and $q = \varepsilon$. Thus, $se_x(x, \varepsilon)$, which is the total number of significant edges of $\mathcal{M}(A^*x)$, is not greater than $2|x| - 2|\varepsilon| + 2 - 1 - 1 = 2|x|$.

6.5 Searching with linear string-matching automata

Our proof of Theorem 6.2 consists in considering linear string-matching automata for matching words. We then consider the model of computation where none ordering on the alphabet is assumed, and give some optimal bounds for string-matching algorithms for which the search phase is on-line with a one-letter buffer.

Consider the search phase of algorithm Matcher using the linear string-matching automaton of a given word x. For each backward or null edge (p, a, q) of the string-matching automaton of x, let us denote by $c_x(p, q)$ the maximum time for executing the series of transitions from q to q via p, *i.e.* for reading the word $x_{|q|+1}x_{|q|+2} \cdots x_{|p|}a$ starting in state q. Let us also denote by $C_x(y)$ the time for executing the search phase when y is the searched word, *i.e.* the time for executing the automaton on y.

Lemma 6.6. *Let $x, y \in A^*$. There exists a finite sequence of backward or null edges of $\mathcal{M}(A^*x)$, say $((p_j, a_j, q_j))_{1 \leqslant j \leqslant k}$, satisfying the three following conditions:*

(i) $q_k = \varepsilon$;

(ii) $\sum_{j=1}^{k} (|p_j| - |q_j| + 1) = |y|$;

(iii) $C_x(y) \leqslant \sum_{j=1}^{k} c_x(p_j, q_j)$.

Proof. The proof is by induction on $|y|$. Since the property trivially holds when $|y| = 0$, we assume that $|y| \geqslant 1$.

Observe first that since an upper bound is expected for $C_x(y)$, we can assume, even if the alphabet has to be extended by one letter, that the last letter of y is not a letter occurring actually in x. Hence, we can assume that the lastly performed transition corresponds to a null edge.

Now, let $(p_\ell)_{0 \leqslant \ell \leqslant |y|}$ be the sequence of successive values of the current state p of algorithm Matcher (in other words $p_\ell = h_x(y_1 y_2 \cdots y_\ell)$). Let m, $0 \leqslant m \leqslant |y| - 1$, be the minimal integer satisfying $p_{m+1} = p_{m'}$ for some $m' \leqslant m$, then let m' be the integer in $\{0, \ldots, m\}$ such that $p_{m'} = p_{m+1}$. Thus, the triple $(p_m, y_{m+1}, p_{m'})$ is a backward or null edge of $\mathcal{M}(A^*x)$, and the $m - m' + 1$ successive letters $y_{m'+1}, y_{m'+2}, \ldots, y_{m+1}$ of y have been read during the computation of the transitions from $p_{m'}$ to p_{m+1} via p_m. Consider the word y' defined by $y' = y_1 y_2 \cdots y_{m'} \cdot y_{m+2} y_{m+3} \cdots y_{|y|}$. Following the definition of m and m' we have that $C_x(y) \leqslant c_x(p_m, p_{m'}) + C_x(y')$. Applying the induction hypothesis to y' gives the existence of a finite sequence e' as depicted in the statement. The expected sequence related to y can then be obtained by adding the edge $((p_m, y_{m+1}, p_{m'}))$ in front of the sequence e'. It clearly satisfies conditions (i) to (iii). This ends the proof. □

Theorem 6.5. *Let x and y be words. Using the linear string-matching automaton of x, searching for all occurrences of x as factors of y (search phase of algorithm* Matcher*) is performed in time $O(|y|)$, constant extra-space, within a delay $O(\log \min\{1 + \lfloor \log_2 |x| \rfloor, \mathrm{card}(A)\})$.*

Proof. Whatever efficient is the implementation of adjacency lists, we may assume that the time for executing the transition from the current state by the current input letter is asymptotically linear in the number of significant edges outgoing the involved state. For each backward or null edge (p, a, q) of $\mathcal{M}(A^*x)$, this assumption implies that

$$c_x(p, q) = O(se_x(p, q));$$

which leads to

$$c_x(p, q) = O(|p| - |q| + 1),$$

by application of Proposition 6.3. We finally apply Lemma 6.6, and get the $O(|y|)$ bound.

We now turn to the proof of the delay. The cardinality of each adjacency list is both upper-bounded by $\mathrm{card}(A)$, and, from Lemma 6.3 and Proposition 6.2, by $1+\lfloor \log_2 |x| \rfloor$. Now, observe that each adjacency list can be arranged in a balanced tree when computing it, without loosing the linear-time complexity of the construction. This provides a logarithmic time for computing a transition. Which proves the asymptotic bound of the delay. □

In the remainder of the section, no ordering on the alphabet is assumed, contrary to what is assumed for the previous statement. The model of computation is the *comparison model* in which algorithms have access to the input words by comparing pairs of letters to test whether they are equal or not. Within this model, given a word x, we denote by $\mathcal{S}(x)$ the family of the string-matching algorithms for which the search phase is on-line with a one-letter buffer.

String-matching algorithms based on the linear string-matching automaton of x can be classified according to the way the adjacency lists are ordered or scanned. For example, the adjacency lists can be ordered by decreasing length of target, or by the frequency of labels as letters of the prefix already read; each adjacency list can also be scanned according to a random processing. (Let us observe that any of these variations preserves the linear time of the search). We denote by $\mathcal{L}(x)$ the subfamily of algorithms in $\mathcal{S}(x)$ which use the linear string-matching automaton of x to search a given word for occurrences of x.

Theorem 6.6. *Given $x \in A^+$, consider an algorithm λ in $\mathcal{L}(x)$, and an input of non-zero length n. In the comparison model, λ performs no more than $2n-1$ letter comparisons, and compares each of the n letter of the input less than* $\min\{1+\lfloor \log_2 |x| \rfloor, \mathrm{card}(A)\}$ *times.*

Proof. This is similar to the proof of Theorem 6.5. The term "-1" of the $2n-1$ bound results from the fact that we can assume that at least one transition by a null edge of $\mathcal{M}(A^*x)$ is simulated (the edge (q_{k-1}, a_k, q_k) of Lemma 6.6). □

The $2n-1$ bound of Theorem 6.6 is also the bound reached by the algorithm of Section 5 when ab is a prefix of the only word of the dictionary and the input is in a^*. However, this worst-case bound can be lowered in $\mathcal{L}(x)$, using the special strategy described in the following statement for computing transitions.

Theorem 6.7. *Given $x \in A^+$, consider an algorithm λ in $\mathcal{L}(x)$ that applies the following strategy: to compute a transition from any state p, scan the edges outgoing p in such a way that the forward edge (if any) is scanned last. Then, in the comparison model, λ executes no more than $\lfloor (2-1/|x|) \times n \rfloor$ letter comparisons on any input of length n.*

Proof. Let (p, a, q) be a backward or null edge of $\mathcal{M}(A^*x)$. The number of comparisons while executing the series of transitions from q to q via p is bounded by $se_x(p,q) - \chi(p \neq x)$. By application of Proposition 6.3, we obtain that at most $(2-1/|x|) \times (|p|-|q|+1)$ comparisons are performed during this series of transitions. Then we apply Lemma 6.6 and obtain the expected bound. □

For a given length m of patterns, the delay $\min\{1+\lfloor \log_2 m \rfloor, \mathrm{card}(A)\}$ (Theorem 6.6) and the coefficient $2-1/m$ (Theorem 6.7) are optimal quantities. This is proved by the next two propositions.

Proposition 6.4. *Consider the comparison model. Then, for each $m \geqslant 1$, for each $n \geqslant m$, there exist $x \in A^m$ and $y \in A^n$ such that any algorithm in $\mathcal{S}(x)$ performs at least* $\min\{1+\lfloor \log_2 m \rfloor, \mathrm{card}(A)\}$ *letter comparisons in the worst-case on some letter of the searched word y.*

Proof. Define recursively the mapping $\xi\colon A^* \to A^*$ by $\xi(ua) = \xi(u) \cdot a \cdot \xi(u)$ for each $(u,a) \in A^* \times A$ and $\xi(\varepsilon) = \varepsilon$. (For instance, we have $\xi(\mathsf{abcd}) = \mathsf{abacabadabacaba}$.)

Set $k = \min\{1+\lfloor \log_2 m \rfloor, \mathrm{card}(A)\}$, choose k pairwise distinct letters, say $a_1, a_2, \ldots, a_k$, and assume that $\xi(a_1 a_2 \cdots a_{k-1})a_k$ is a prefix of x. If $\xi(a_1 a_2 \cdots a_{k-1})$ is the already read prefix of y, the algorithm can suppose that an occurrence of x starts at one of the positions in the form $2^{k-\ell}$, $1 \leqslant \ell \leqslant k$. Hence, the algorithm performs never less than k letter comparisons at position 2^{k-1} in the worst-case. □

Proposition 6.5. *Consider the comparison model. Then, for each $m \geqslant 1$, for each $n \geqslant 0$, there exist $x \in A^m$ and $y \in A^n$ such that any algorithm in $\mathcal{S}(x)$ performs at least $\lfloor (2-1/m) \times n \rfloor$ letter comparisons in the worst-case when searching y.*

Proof. Assume that $x = \mathsf{ab}^{m-1}$ and $y \in \mathit{Pref}((\mathsf{a}\{\mathsf{a},\mathsf{b}\}^{m-1})^*)$. Then let j, $1 \leqslant j \leqslant n$, be the current position on y, and let v be the longest suffix of $y_1 y_2 \cdots y_{j-1}$ that is also a proper prefix of x.

If $v \neq \varepsilon$, the algorithm has to query both if $y_j = \mathsf{a}$ and if $y_j = \mathsf{b}$ in the worst-case, in order to be able to report later an occurrence of x at position either j or $j-|v|$. Otherwise $v = \varepsilon$, and the algorithm can just query if $y_j = \mathsf{a}$. But, according to the definition of y, the second case, namely $v = \varepsilon$, may occur only when $j \equiv 1 \pmod m$. Therefore, the number of letter comparisons performed on y is then never less than $2n - \lceil n/m \rceil = \lfloor (2-1/m) \times n \rfloor$ in the worst-case. □

7. Suffix automata

7.1 Outline

The *suffix automaton* of a word x is defined as the minimal deterministic (non necessarily complete) automaton that recognizes the (finite) set of suffixes of x. It is denoted by $\mathcal{M}(\mathit{Suff}(x))$ according to notations of Section 2.

An example of suffix automaton is displayed in Figure 7.1.

The automaton $\mathcal{M}(\mathit{Suff}(x))$ can be used as an index on x to solve the following problem.

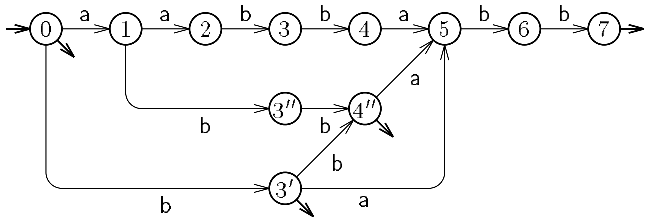


Fig. 7.1. The minimal deterministic automaton recognizing the suffixes of aabbabb.

Problem 7.1. (Index problem.) Given a word x, preprocess it in order to locate all occurrences of any word y in x.

An alternative solution to the problem is to implement data structures techniques based on a representation of the set of suffixes of x by compact tries. This structure is known as the suffix tree of x.

The suffix automaton provides another solution to the string-matching problem (see Section 6) since it can also be used to search a word y for factors of x. This yields a space efficient solution to the search for rotations (Section 7.5) of a given word.

The surprising property of suffix automata is that their size is linear although the number of factors of a word can be quadratic in the length of the word. The construction of suffix automata is also linear on fixed alphabets.

Main Theorem 7.1. *The size of the suffix automaton of a word x is $O(|x|)$. The automaton can be implemented in time $O(|x| \times \log \mathrm{card}(A))$ and $O(|x|)$ extra-space.*

The implementation which is referred to in the theorem is based on adjacency lists. As in Section 5, if we allow more extra-space, the time complexity reduces to $O(|x|)$. This is valid also for Theorems 7.5 and 7.6, Propositions 7.2, 7.3, 7.4, 7.5, and 7.8.

We first review in Section 7.2 properties of suffix automata that are useful to design a construction method. At the same time, we provide exact bounds on the size of the automata. These results have consequences on the running time of the method. Section 7.3 is devoted to the construction of suffix automata itself. The same approach is presented in Section 7.6 for factor automata. Sections 7.4 and 7.5 show how these automata can be used either as indexes or as string-matching automata.

7.2 Sizes and properties

7.2.1 End-positions. Right contexts according to $\mathit{Suff}(x)$ satisfy a few properties stated in the next lemmas and used later in the chapter. The first remark concerns the context of a suffix of a word.

Lemma 7.1. *Let $u, v \in A^*$. If $u \in \mathit{Suff}(v)$, then $v^{-1}\mathit{Suff}(x) \subseteq u^{-1}\mathit{Suff}(x)$.*

Proof. If $v^{-1}\mathit{Suff}(x) = \varnothing$ the inclusion trivially holds. Otherwise, let $z \in v^{-1}\mathit{Suff}(x)$. Then, $vz \in \mathit{Suff}(x)$ and, since $u \in \mathit{Suff}(v)$, $uz \in \mathit{Suff}(x)$. So, $z \in u^{-1}\mathit{Suff}(x)$. □

Right contexts satisfy a kind of converse statement. To formalize it, we introduce the function $\mathit{endpos}_x \colon \mathit{Fact}(x) \to \mathbb{N}$ defined for each word u by

$$\mathit{endpos}_x(u) = \min\{|w| \mid w \text{ is a prefix of } x \text{ and } u \text{ is a suffix of } w\}.$$

The value $\mathit{endpos}_x(u)$ marks the ending position of the first (or leftmost) occurrence of u in x.

Lemma 7.2. *Let $u, v \in \mathit{Fact}(x)$. If $u \equiv_{\mathit{Suff}(x)} v$, we have the equality $\mathit{endpos}_x(u) = \mathit{endpos}_x(v)$, which is equivalent to say that one of the words u and v is a suffix of the other.*

Proof. Let $y, z \in A^*$ be such that $x = yz$ and $u \in \mathit{Suff}(y)$. We assume in addition that $|y| = \mathit{endpos}_x(u)$. Then z is the longest word of $u^{-1}\mathit{Suff}(x)$. The hypothesis implies that z is also the longest word of $v^{-1}\mathit{Suff}(x)$, which shows that $|y| = \mathit{endpos}_x(v)$. In this situation, u and v are both suffixes of y, which proves that one of them is a suffix of the other. □

Another often used property of the syntactic congruence associated with $\mathit{Suff}(x)$ is that it partitions the suffixes of factors into intervals (with respect to the lengths of suffixes).

Lemma 7.3. *Let $u, v, w \in \mathit{Fact}(x)$. Then, if $u \in \mathit{Suff}(v)$, $v \in \mathit{Suff}(w)$, and $u \equiv_{\mathit{Suff}(x)} w$, we have $u \equiv_{\mathit{Suff}(x)} v \equiv_{\mathit{Suff}(x)} w$.*

Proof. By Lemma 7.1, we have the inclusions $w^{-1}\mathit{Suff}(x) \subseteq v^{-1}\mathit{Suff}(x) \subseteq u^{-1}\mathit{Suff}(x)$. But then, the equality $u^{-1}\mathit{Suff}(x) = w^{-1}\mathit{Suff}(x)$ implies the conclusion of the statement. □

A consequence of the next property is that the direct inclusion of right contexts relative to $\mathit{Suff}(x)$ induces a tree structure on them. In the tree, the parent link corresponds to the proper direct inclusion. This link is discussed in Section 7.2.2 where it is called the "suffix function".

Corollary 7.1. *Let $u, v \in A^*$. Then, one of the three following conditions holds:*

(i) $u^{-1}\mathit{Suff}(x) \subseteq v^{-1}\mathit{Suff}(x)$;
(ii) $v^{-1}\mathit{Suff}(x) \subseteq u^{-1}\mathit{Suff}(x)$;
(iii) $u^{-1}\mathit{Suff}(x) \cap v^{-1}\mathit{Suff}(x) = \varnothing$.

Proof. We just have to show that if $u^{-1}\mathit{Suff}(x) \cap v^{-1}\mathit{Suff}(x) \neq \varnothing$, then we have the inclusion $u^{-1}\mathit{Suff}(x) \subseteq v^{-1}\mathit{Suff}(x)$ or the inclusion $v^{-1}\mathit{Suff}(x) \subseteq u^{-1}\mathit{Suff}(x)$. Let $z \in u^{-1}\mathit{Suff}(x) \cap v^{-1}\mathit{Suff}(x)$. Then, uz and vz are suffixes of x. So, u and v are suffixes of xz^{-1}, which implies that one of the words u and v is a suffix of the other. Therefore, the conclusion follows by Lemma 7.1. □

7.2.2 Suffix function. We consider the function $s_x \colon \mathit{Fact}(x) \to \mathit{Fact}(x)$ defined for each nonempty word v in $\mathit{Fact}(x)$ by

$$s_x(v) = \text{the longest } u \in \mathit{Suff}(v) \text{ such that } u \not\equiv_{\mathit{Suff}(x)} v.$$

Regarding Lemma 7.1, this is equivalent to

$$s_x(v) = \text{the longest } u \in \mathit{Suff}(v) \text{ such that } v^{-1}\mathit{Suff}(x) \subset u^{-1}\mathit{Suff}(x).$$

The function s_x is called the *suffix function* relative to x. An obvious consequence of the definition is that $s_x(v)$ is a proper suffix of v. The next lemma shows that the suffix function s_x induces what we call a "suffix link" on states of $\mathcal{M}(\mathit{Suff}(x))$.

Lemma 7.4. *Assuming $x \neq \varepsilon$, let $u, v \in \mathit{Fact}(x) \backslash \{\varepsilon\}$. If $u \equiv_{\mathit{Suff}(x)} v$, then $s_x(u) = s_x(v)$.*

Proof. From Lemma 7.2 we can assume without loss of generality that $u \in \mathit{Suff}(v)$. The word u cannot be a suffix of $s_x(v)$, because Lemma 7.3 would then imply $s_x(v)^{-1}\mathit{Suff}(x) = v^{-1}\mathit{Suff}(x)$, which contradicts the definition of $s_x(v)$. Therefore, $s_x(v)$ is a suffix of u. Since, by definition, it is the longest suffix of v non equivalent to it, it is equal to $s_x(u)$. □

Lemma 7.5. *If $x \neq \varepsilon$, $s_x(x)$ is the longest suffix of x that occurs at least twice in x.*

Proof. The set $x^{-1}\mathit{Suff}(x)$ is equal to $\{\varepsilon\}$. Since x and $s_x(x)$ are not equivalent, the set $s_x(x)^{-1}\mathit{Suff}(x)$ contains some nonempty word z. Therefore, $s_x(x)z$ and $s_x(x)$ are suffixes of x, which proves that $s_x(x)$ occurs at least twice in x. Any suffix w of x, longer than $s_x(x)$, satisfies $w^{-1}\mathit{Suff}(x) = x^{-1}\mathit{Suff}(x) = \{\varepsilon\}$ by definition of $s_x(x)$. Thus, w occurs only as a suffix of x, which ends the proof. □

The next lemma shows that the image of a factor of x by the suffix function is a word of maximum length in its own congruence class. This fact is needed in Section 7.5 where the suffix automaton is used as a matching automaton.

Lemma 7.6. *Assuming $x \neq \varepsilon$, let $u \in \mathit{Fact}(x) \backslash \{\varepsilon\}$. Then, any word equivalent to $s_x(u)$ is a suffix of $s_x(u)$.*

Proof. Let $w = s_x(u)$ and $v \equiv_{\mathit{Suff}(x)} s_x(u)$. The word w is a proper suffix of u. If the conclusion of the statement is false, Lemma 7.2 insures that w is a proper suffix of v. Let $z \in u^{-1}\mathit{Suff}(x)$. Since w is a suffix of u and is equivalent to v, we have $z \in w^{-1}\mathit{Suff}(x) = v^{-1}\mathit{Suff}(x)$. Therefore, u and v are both suffixes of xz^{-1}, which implies that one of them is a suffix of the other. But this contradicts either the definition of w, or the conclusion of Lemma 7.3. This proves that v is necessarily a suffix of w. □

7.2.3 State splitting. In this section we present the properties that yield to the on-line construction of suffix automata described in Section 7.3. This is achieved by deriving relations between the congruences $\equiv_{\mathit{Suff}(w)}$ and $\equiv_{\mathit{Suff}(wa)}$ for any couple $(w,a) \in A^* \times A$. The first property, stated in Lemma 7.8, is that $\equiv_{\mathit{Suff}(wa)}$ is a refinement of $\equiv_{\mathit{Suff}(w)}$. The next lemma shows how right contexts evolves.

Lemma 7.7. *Let $w \in A^*$ and $a \in A$. For each $u \in A^*$, we have:*

$$u^{-1}\mathit{Suff}(wa) = \begin{cases} u^{-1}\mathit{Suff}(w)a \cup \{\varepsilon\}, & \text{if } u \in \mathit{Suff}(wa), \\ u^{-1}\mathit{Suff}(w)a, & \text{otherwise.} \end{cases}$$

Proof. Note first that $\varepsilon \in u^{-1}\mathit{Suff}(wa)$ is equivalent to $u \in \mathit{Suff}(wa)$. So, it remains to prove $u^{-1}\mathit{Suff}(wa)\backslash\{\varepsilon\} = u^{-1}\mathit{Suff}(w)a$.

Let z be a nonempty word in $u^{-1}\mathit{Suff}(wa)$. This means $uz \in \mathit{Suff}(wa)$. The word uz can then be written $uz'a$ with $uz' \in \mathit{Suff}(w)$. Thus, $z' \in u^{-1}\mathit{Suff}(w)$, and $z \in u^{-1}\mathit{Suff}(w)a$.

Conversely. Let z be a (nonempty) word in $u^{-1}\mathit{Suff}(w)a$. It can be written $z'a$ for some $z' \in u^{-1}\mathit{Suff}(w)$. Therefore, $uz' \in \mathit{Suff}(w)$, which implies $uz = uz'a \in \mathit{Suff}(wa)$, that is $z \in u^{-1}\mathit{Suff}(wa)$. □

Lemma 7.8. *Let $w \in A^*$ and $a \in A$. The congruence $\equiv_{\mathit{Suff}(wa)}$ is a refinement of the congruence $\equiv_{\mathit{Suff}(w)}$, that is, for each $u, v \in A^*$, $u \equiv_{\mathit{Suff}(wa)} v$ implies $u \equiv_{\mathit{Suff}(w)} v$.*

Proof. We assume $u \equiv_{\mathit{Suff}(wa)} v$, that is, $u^{-1}\mathit{Suff}(wa) = v^{-1}\mathit{Suff}(wa)$, and prove $u \equiv_{\mathit{Suff}(w)} v$, that is, $u^{-1}\mathit{Suff}(w) = v^{-1}\mathit{Suff}(w)$. We only show that $u^{-1}\mathit{Suff}(w) \subseteq v^{-1}\mathit{Suff}(w)$ because the reverse inclusion follows by symmetry.

If $u^{-1}\mathit{Suff}(w)$ is empty, the inclusion trivially holds. Otherwise, let $z \in u^{-1}\mathit{Suff}(w)$. This is equivalent to $uz \in \mathit{Suff}(w)$, which implies $uza \in \mathit{Suff}(wa)$. The hypothesis gives $vza \in \mathit{Suff}(wa)$, and thus $vz \in \mathit{Suff}(w)$ or $z \in v^{-1}\mathit{Suff}(w)$, which achieves the proof. □

Given a word w, the congruence $\equiv_{\mathit{Suff}(w)}$ partitions A^* into classes. And Lemma 7.8 remains to say that these classes are union of classes according to $\equiv_{\mathit{Suff}(wa)}$, $a \in A$. It turns out that only one or two classes according to $\equiv_{\mathit{Suff}(w)}$ split into two sub-classes to get the partition induced by $\equiv_{\mathit{Suff}(wa)}$. One of the class that splits is the class of words not occurring in w. It contains the word wa itself that gives rise to a new class and a new state of the suffix automaton (see Lemma 7.9). Theorem 7.2 and its corollaries exhibit conditions under which another class also splits and how it splits.

Lemma 7.9. *Let $w \in A^*$ and $a \in A$. Let z be the longest suffix of wa occurring in w. If u is a suffix of wa such that $|u| > |z|$, $u \equiv_{\mathit{Suff}(wa)} wa$.*

Proof. This is a straightforward consequence of Lemma 7.5. □

Theorem 7.2. *Let $w \in A^*$ and $a \in A$. Let z be the longest suffix of wa occurring in w. Let z' be the longest factor of w such that $z' \equiv_{\mathit{Suff}(w)} z$. For each $u, v \in \mathit{Fact}(w)$, we have:*

$$u \equiv_{\mathit{Suff}(w)} v \quad \text{and} \quad u \not\equiv_{\mathit{Suff}(w)} z \quad \Longrightarrow \quad u \equiv_{\mathit{Suff}(wa)} v.$$

Furthermore, for each $u \in A^*$, we have:

$$u \equiv_{\mathit{Suff}(w)} z \quad \Longrightarrow \quad \begin{cases} u \equiv_{\mathit{Suff}(wa)} z, & \text{if } |u| \leqslant |z|, \\ u \equiv_{\mathit{Suff}(wa)} z', & \text{otherwise.} \end{cases}$$

Proof. Let $u, v \in \mathit{Fact}(w)$ be such that $u \equiv_{\mathit{Suff}(w)} v$, that is, $u^{-1}\mathit{Suff}(w) = v^{-1}\mathit{Suff}(w)$. We first assume $u \not\equiv_{\mathit{Suff}(w)} z$ and prove $u \equiv_{\mathit{Suff}(wa)} v$, that is $u^{-1}\mathit{Suff}(wa) = v^{-1}\mathit{Suff}(wa)$.

By Lemma 7.7, we just have to prove that $u \in \mathit{Suff}(wa)$ is equivalent to $v \in \mathit{Suff}(wa)$. Indeed, it is even sufficient to prove that $u \in \mathit{Suff}(wa)$ implies $v \in \mathit{Suff}(wa)$ because the reverse implication comes by symmetry.

Assume then that $u \in \mathit{Suff}(wa)$. Since $u \in \mathit{Fact}(w)$, u is a suffix of z, by definition of z. So, we can consider the largest integer $k \geqslant 0$ such that $|u| \leqslant |{s_w}^k(z)|$. Note that ${s_w}^k(z)$ is a suffix of wa (as z is), and that Lemma 7.3 insures that $u \equiv_{\mathit{Suff}(w)} {s_w}^k(z)$. So, $v \equiv_{\mathit{Suff}(w)} {s_w}^k(z)$ by transitivity.

Since $u \not\equiv_{Suff(w)} z$, we have that $k > 0$. Thus, Lemma 7.6 implies that v is a suffix of ${s_w}^k(z)$, and then that v is a suffix of wa as expected. This proves the first part of the statement.

Consider now a word u such that $u \equiv_{Suff(w)} z$.

If $|u| \leqslant |z|$, to prove $u \equiv_{Suff(wa)} z$, using the above argument, we just have to show that $u \in Suff(wa)$ because $z \in Suff(wa)$. Indeed, this is a simple consequence of Lemma 7.2.

Conversely, assume that $|u| > |z|$. When such a word u exists, $z' \neq z$ and $|z'| > |z|$ (z is a proper suffix of z'). Therefore, by the definition of z, u and z' are not suffixes of wa. Using again the above argument, this shows that $u \equiv_{Suff(wa)} z'$.

This proves the second part of the statement and ends the proof. □

Corollary 7.2. *Let $w \in A^*$ and $a \in A$. Let z be the longest suffix of wa occurring in w. Let z' be the longest word such that $z' \equiv_{Suff(w)} z$. If $z' = z$, then, for each $u, v \in Fact(w)$, $u \equiv_{Suff(w)} v$ implies $u \equiv_{Suff(wa)} v$.*

Proof. The conclusion follows directly from Theorem 7.2 if $u \not\equiv_{Suff(w)} z$. Otherwise, $u \equiv_{Suff(w)} z$, and by the hypothesis on z and Lemma 7.2, we get $|u| \leqslant |z|$. Thus, Theorem 7.2 again gives the same conclusion. □

Corollary 7.3. *Let $w \in A^*$ and $a \in A$. Assume that letter a does not occur in w. Then, for each $u, v \in Fact(w)$, $u \equiv_{Suff(w)} v$ implies $u \equiv_{Suff(wa)} v$.*

Proof. Since a does not occur in w, the word z of Corollary 7.2 is the empty word. This word is the longest word in its own congruence class. So, the hypothesis of Corollary 7.2 holds. Therefore, the same conclusion follows. □

7.2.4 Sizes of suffix automata. We discuss the size of suffix automata both in term of number of states and number of edges. We show that the global size of $\mathcal{M}(Suff(x))$ is $O(|x|)$. The set of states and the set of edges of $\mathcal{M}(Suff(x))$ are respectively denoted by Q and E (without mention of x that is implicit in statements).

Corollary 7.4. *If $|x| = 0$, $\mathrm{card}(Q) = 1$; and if $|x| = 1$, $\mathrm{card}(Q) = 2$. Otherwise $|x| \geqslant 2$; then, $|x| + 1 \leqslant \mathrm{card}(Q) \leqslant 2|x| - 1$ and the upper bound is reached only when x is in the form $ab^{|x|-1}$ for two distinct letters a and b.*

Proof. The minimum number of states is obviously $|x| + 1$, and is reached when x is in the form $a^{|x|}$ for some $a \in A$. Moreover, we have exactly $\mathrm{card}(Q) = |x| + 1$ when $|x| \leqslant 2$.

Assume now that $|x| \geqslant 3$. By Theorem 7.2, each symbol x_k, $3 \leqslant k \leqslant |x|$, increases by at most 2 the number of states of $\mathcal{M}(Suff(x_1 x_2 \cdots x_{k-1}))$. Since the number of states for a word of length 2 is 3, we get that

$$\mathrm{card}(Q) \leqslant 3 + 2(|x| - 2) = 2|x| - 1,$$

as announced in the statement.

The construction of a word x reaching the upper bound for the number of states of $\mathcal{M}(Suff(x))$ is a mere application of Theorem 7.2 considering that each letter x_k, $3 \leqslant k \leqslant |x|$, should effectively increase by 2 the number of states of $\mathcal{M}(Suff(x_1 x_2 \cdots x_{k-1}))$. □

Figure 7.2 displays a suffix automaton whose number of states is maximum for a word of length 7.

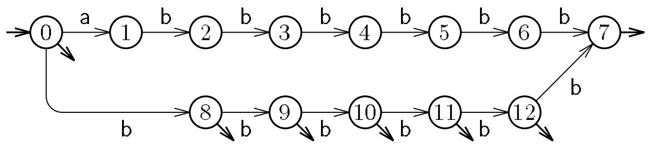


Fig. 7.2. A suffix automaton with the maximum number of states.

Let $length_x \colon Q \to \mathbb{N}$ be the function associating to each state q of $\mathcal{M}(Suff(x))$ the length of the longest word u in the congruence class q. It is also the length of the longest path from the initial state to q. (This path is labeled by u.) Longest paths form a spanning tree on $\mathcal{M}(\mathit{Suff}(x))$ (a consequence of Lemma 7.2). Transitions that belong to that tree are called *solid* edges. Equivalently, for each edge (p, a, q) of $\mathcal{M}(\mathit{Suff}(x))$, we have that:

$$(p, a, q) \text{ is solid} \iff \mathit{length}_x(q) = \mathit{length}_x(p) + 1.$$

This notion is used in the construction of suffix automata to test the condition stated in Theorem 7.2. We use it here to derive exact bounds on the number of edges of suffix automata.

Lemma 7.10. *Assuming* $|x| \geqslant 1$, $\mathrm{card}(E) \leqslant \mathrm{card}(Q) + |x| - 2$.

Proof. Consider the spanning tree of longest paths from the initial state in $\mathcal{M}(\mathit{Suff}(x))$. The tree contains $\mathrm{card}(E) - 1$ edges of $\mathcal{M}(\mathit{Suff}(x))$, which are the solid edges.

To each non-solid edge (p, a, q) we associate the suffix uav of x defined as follows: u is the label of the longest path from the initial state to p, and v is the label of the longest path from q to a terminal state. Note that, doing so, two different non-solid edges are associated with two different suffixes of x. Since suffixes x and ε are labels of paths in the tree, they are not considered in the correspondence. Thus, the number of non-solid edges is at most $|x| - 1$.

Counting together the number of both kinds of edges gives the expected upper bound. □

Corollary 7.5. *If* $|x| = 0$, $\mathrm{card}(E) = 0$; *if* $|x| = 1$, $\mathrm{card}(E) = 1$; *and if* $|x| = 2$, $2 \leqslant \mathrm{card}(E) \leqslant 3$. *Otherwise* $|x| \geqslant 3$; *then* $|x| \leqslant \mathrm{card}(E) \leqslant 3|x| - 4$, *and the upper bound is reached when x is in the form* $ab^{|x|-2}c$, *for three pairwise distinct letters a, b, and c.*

Proof. The lower bound is obvious, and reached when x is in the form $a^{|x|}$ for some $a \in A$. The upper bound can be checked by hand for the cases where $|x| \leqslant 2$.

Assume now that $|x| \geqslant 3$. By Corollary 7.4 and Lemma 7.10 we have $\mathrm{card}(E) \leqslant 2|x| - 1 + |x| - 2 = 3|x| - 3$. The quantity $2|x| - 1$ is the maximum number of states obtained only when x is in the form $ab^{|x|-1}$ for two distinct letters a and b. But the number of edges in $\mathcal{M}(\mathit{Suff}(ab^{|x|-1}))$ is only $2|x| - 1$. So, $\mathrm{card}(E) \leqslant 3|x| - 4$.

The automaton $\mathcal{M}(\mathit{Suff}(ab^{|x|-2}c))$, for three pairwise distinct letters a, b and c, has $2|x| - 2$ states and exactly $3|x| - 4$ edges composed of $2|x| - 3$ solid edges and $|x| - 1$ non-solid edges. □

Figure 7.3 displays a suffix automaton whose number of edges is maximum for a word of length 7.

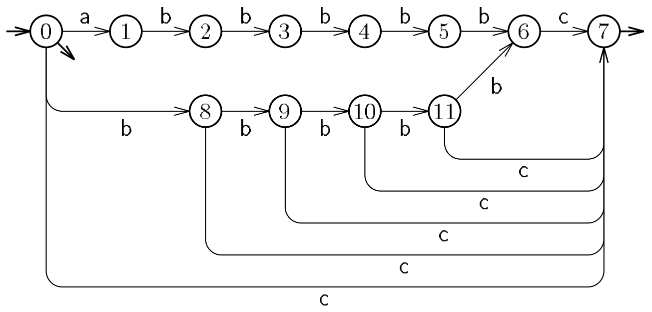


Fig. 7.3. A suffix automaton with the maximum number of edges.

As a conclusion of Section 7.2, we get the following statement, direct consequence of Corollaries 7.4 and 7.5.

Theorem 7.3. *The total size of the suffix automaton of a word is linear in the length of the word.*

7.3 Construction

We describe in Sections 7.3.1, 7.3.2 and 7.3.3 an on-line construction of the suffix automaton $\mathcal{M}(\mathit{Suff}(x))$.

7.3.1 Suffix links and suffix paths. The construction of $\mathcal{M}(Suff(x))$ follows Theorem 7.2 and its corollaries stated in Section 7.2. Conditions that appear in these statements are checked on the automaton with the help of a function defined on its states and called the "suffix link". It is a failure function in the sense of Section 3.4, and is used with this purpose in Section 7.5.

Let $(Q,i,T,E)=\mathcal{M}(Suff(x))$ and δ be the corresponding transition function. Let $p\in Q\backslash\{i\}$. State p is a class of factors of x congruent with respect to $\equiv_{Suff(x)}$. Let u be any word in the class of p ($u\neq\varepsilon$ because $p\neq i$). Then, the *suffix link* of p is the congruence class of $s_x(u)$. By Lemma 7.4 the value $s_x(u)$ is independent of the word u chosen in the class p, which makes the definition coherent. We denote by f_x the function assignating to each state p its congruence class $s_x(u)$.

Suffix links induce by iteration "suffix paths" in $\mathcal{M}(Suff(x))$. Note that if $q = f_x(p)$, then $length_x(q) < length_x(p)$. Therefore, the sequence

$$(p,\ \ f_x(p),\ \ {f_x}^2(p),\ \ \ldots)$$

is finite and ends with the initial state i. It is called the *suffix path* of p.

We denote by $last_x$ the state of $\mathcal{M}(Suff(x))$ that is the class of x itself. State $last_x$ has no outgoing edge (otherwise $\mathcal{M}(Suff(x))$ would recognize words longer than x). The suffix path of $last_x$, *i.e.*

$$(last_x,\ \ f_x(last_x),\ \ {f_x}^2(last_x),\ \ \ldots),$$

plays an important role in the on-line construction. It is used to test efficiently conditions appearing in statements of the previous section.

Proposition 7.1. *Let $u\in Fact(x)\backslash\{\varepsilon\}$ and set $p=\delta(i,u)$. Then, for any integer $k\geqslant 0$ for which ${s_x}^k(u)$ is defined, ${f_x}^k(p)=\delta(i,{s_x}^k(u))$.*

Proof. The proof is by induction on k.

For $k=0$, the equality holds by hypothesis.

Next, let $k\geqslant 1$ such that ${s_x}^k(u)$ is defined and assume that ${f_x}^{k-1}(p)=\delta(i,{s_x}^{k-1}(u))$. By definition of f_x, $f_x({f_x}^{k-1}(p))$ is the congruence class of the word $s_x({s_x}^{k-1}(u))$. Therefore, ${f_x}^k(p)=\delta(i,{s_x}^k(u))$ as expected. □

Corollary 7.6. *Terminal states of $\mathcal{M}(Suff(x))$, the states in T, are exactly the states of the suffix path of state $last_x$.*

Proof. Let p be a state of the suffix path of $last_x$. Then, $p={f_x}^k(last_x)$ for some integer $k\geqslant 0$. By Proposition 7.1, since $last_x=\delta(i,x)$, we have $p=\delta(i,{s_x}^k(x))$. Since ${s_x}^k(x)$ is a suffix of x, $p\in T$.

Conversely, let $p\in T$. So, for some $u\in Suff(x)$, $p=\delta(i,u)$. Since $u\in Suff(x)$, we can consider the largest integer $k\geqslant 0$ such that $|u|\leqslant|{s_x}^k(x)|$. By Lemma 7.3 we get $u\equiv_{Suff(x)}{s_x}^k(x)$. Thus, $p=\delta(i,{s_x}^k(x))$ by definition of $\mathcal{M}(Suff(x))$. Then, Proposition 7.1 applied to x shows that $p={f_x}^k(last_x)$, which proves that p belongs to the suffix path of $last_x$. □

7.3.2 On-line construction. This section presents an on-line construction of suffix automata. At each stage of the construction, just after processing a prefix $x_1x_2\cdots x_\ell$ of x, the suffix automaton $\mathcal{M}(Suff(x_1x_2\cdots x_\ell))$ is built. Terminal states are implicitly known by the suffix path of $last_{x_1x_2\cdots x_\ell}$ (see Corollary 7.6). The state $last_{x_1x_2\cdots x_\ell}$ is explicitly represented by a variable in the function building the automaton.

Two other elements are also used: *Length* and F. The table *Length* represents the function $length_x$ defined on states of the automaton. All edges are solid or non-solid according to the definition of Section 7.2 that relies on function $length_x$. Suffix links of states (different from the initial state) are stored in a table denoted by F that stands for the function f_x. The implementation of $\mathcal{M}(Suff(x))$ with these extra features is discussed in the next section.

The on-line construction in Figure 7.4 is based on procedure SA-Extend given in Figure 7.5. The latter procedure processes the next letter, say x_ℓ, of the word x. It transforms the suffix automaton $\mathcal{M}(Suff(x_1x_2\cdots x_{\ell-1}))$ already built into the suffix automaton $\mathcal{M}(Suff(x_1x_2\cdots x_\ell))$.

We illustrate how procedure SA-Extend processes the current automaton through three examples. Let us consider that $x_1x_2\cdots x_{\ell-1}=\mathsf{ccccbbccc}$, and let us examine three possible cases according to

```
SuffixAutomaton(x)
    let δ be the transition function of (Q, i, T, E)
    (Q, E) ← (∅, ∅)
    i ← State-Creation
    Length[i] ← 0
    F[i] ← nil
    last ← i
    for ℓ from 1 up to |x|
       loop SA-Extend(ℓ)
    T ← ∅
    p ← last
    loop    T ← T + {p}
            p ← F[p]
       while p ≠ nil
    return ((Q, i, T, E), Length, F)
```

Fig. 7.4. On-line construction of the suffix automaton of a word x.

```
SA-Extend(ℓ)
a ← x_ℓ
newlast ← State-Creation
Length[newlast] ← Length[last] + 1
p ← last
loop    E ← E + {(p, a, newlast)}
        p ← F[p]
   while p ≠ nil and δ(p, a) = nil
if p = nil
   then F[newlast] ← i
   else q ← δ(p, a)
        if Length[q] = Length[p] + 1
           then F[newlast] ← q
           else q' ← State-Creation
                for each letter b such that δ(q, b) ≠ nil
                   loop E ← E + {(q', b, δ(q, b))}
                Length[q'] ← Length[p] + 1
                F[newlast] ← q'
                F[q'] ← F[q]
                F[q] ← q'
                loop    E ← E − {(p, a, q)} + {(p, a, q')}
                        p ← F[p]
                   while p ≠ nil and δ(p, a) = q
last ← newlast
```

Fig. 7.5. From $\mathcal{M}(\mathit{Suff}(x_1x_2\cdots x_{\ell-1}))$ to $\mathcal{M}(\mathit{Suff}(x_1x_2\cdots x_\ell))$.

$a = x_\ell$, namely $a = \mathsf{d}$, $a = \mathsf{c}$, and $a = \mathsf{b}$. The suffix automaton of $x_1x_2\cdots x_{\ell-1}$ is depicted in Figure 7.6. Figures 7.7, 7.8, and 7.9 display respectively $\mathcal{M}(\mathit{Suff}(\mathsf{ccccbbcccd}))$, $\mathcal{M}(\mathit{Suff}(\mathsf{ccccbbcccc}))$, and $\mathcal{M}(\mathit{Suff}(\mathsf{ccccbbcccb}))$.

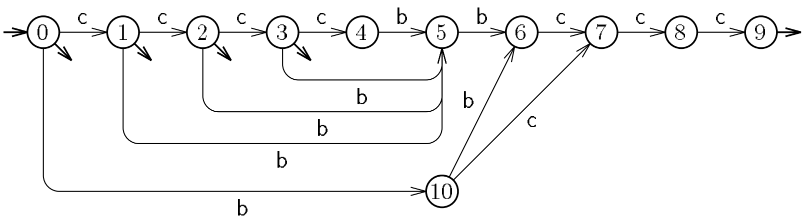


Fig. 7.6. $\mathcal{M}(\mathit{Suff}(\mathsf{ccccbbccc}))$.

During the execution of the first loop of the procedure, state p runs through a part of the suffix path of *last*. At the same time, edges labeled by a are created from p to the newly created state, unless such an edge already exists in which case the loop stops.

If $a = \mathsf{d}$, the execution of the loop stops at the initial state. The edges labeled by d start at terminal states of $\mathcal{M}(\mathit{Suff}(\mathsf{ccccbbccc}))$. This case corresponds to Corollary 7.3. The resulting automaton is given in Figure 7.7.

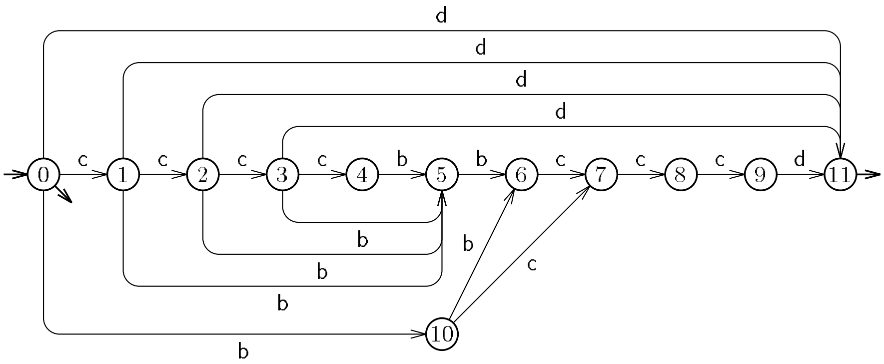


Fig. 7.7. $\mathcal{M}(\mathit{Suff}(\mathsf{ccccbbcccd}))$.

If $a = \mathsf{c}$, the loop stops on state $3 = F[\mathit{last}]$ (of the automaton depicted in Figure 7.6) because an edge labeled by c is defined on it. Moreover, the edge is solid, so, we get the suffix link of the new state. Nothing else should be done according to Corollary 7.2. This gives the automaton of Figure 7.8.

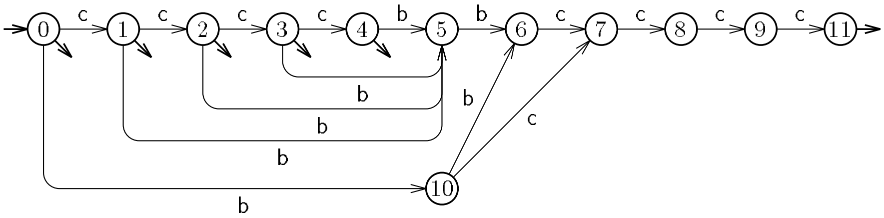


Fig. 7.8. $\mathcal{M}(\mathit{Suff}(\mathsf{ccccbbcccc}))$.

Finally, when $a = \mathsf{b}$, the loop stops on state $3 = F[\mathit{last}]$ for the same reason, but the edge labeled by b from 3 is non-solid. The word cccb is a suffix of the new word $\mathsf{ccccbbcccb}$ but ccccb is not. Since these two words reach state 5, this state is duplicated into a new state that becomes a terminal state. Suffixes ccb and cb are re-directed to this new state, according to Theorem 7.2. We get the automaton of Figure 7.9.

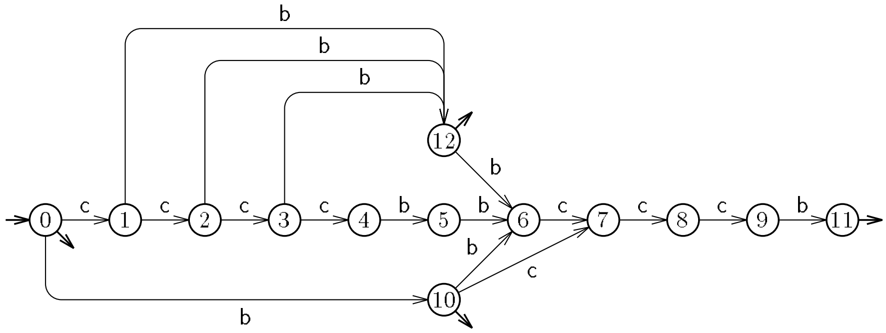


Fig. 7.9. $\mathcal{M}(Suff(\mathsf{ccccbbcccb}))$.

Theorem 7.4. *Function* SUFFIXAUTOMATON *builds the suffix automaton of any given word* x.

Proof. The proof is by induction on the length of x. It heavily relies on the properties stated previously.

If $x = \varepsilon$, the function builds an automaton with only one state that is both initial and terminal. No edge is defined. So, the automaton recognizes the language $\{\varepsilon\}$, which is $Suff(x)$.

Otherwise $x \neq \varepsilon$. Let $w \in A^*$ and $a \in A$ be such that $x = wa$. We assume, after preprocessing w, that the current values of Q and E are respectively the set of states and of edges of $\mathcal{M}(Suff(w))$, that *last* is the state $\delta(i, w)$, that $Length[r] = length_w(r)$ for each $r \in Q$, and that $F[r] = f_w(r)$ for each $r \in Q\backslash\{i\}$. We prove first that procedure SA-EXTEND correctly updates sets Q and E, variable *last*, and tables *Length* and F. Then, we show that terminal states are eventually correctly marked by function SUFFIXAUTOMATON.

The variable p of procedure SA-EXTEND runs through the states of the suffix path of *last* of $\mathcal{M}(Suff(w))$. The first loop creates edges by letter a onto the new created state *newstate* according to Lemma 7.9, and we have the equality $Length[newlast] = length_x(newlast)$.

When the loop stops, three exclusive cases can be distinguished:

(i) p is undefined;
(ii) (p, a, q) is a solid edge;
(iii) (p, a, q) is a non-solid edge.

Case (i). The letter a does not occur in w, so, $f_x(newlast) = i$. Then, we have $F[newlast] = f_x(newlast)$. For any other state r, $f_w(r) = f_x(r)$ by Corollary 7.3. Then, again $F[r] = f_x(r)$ at the end of execution of procedure SA-EXTEND.

Case (ii). Let u be the longest word such that $\delta(i, u) = p$. By induction and by Lemma 7.6, we have $|u| = length_x(p) = Length[p]$. The word ua is the longest suffix of x occurring in w. Then, $f_x(newlast) = q$, and thus $F[newlast] = f_x(newlast)$.

Since the edge (p, a, q) is solid, using the induction again, we obtain $|ua| = Length[q] = length_x(q)$, which shows that words congruent to ua according to $\equiv_{Suff(w)}$ are not longer than ua. Therefore, Corollary 7.2 applies with $z = ua$. And as in case (i), $F[r] = f_x(r)$ for each state different than *newlast*.

Case (iii). Let u be the longest word such that $\delta(i, u) = p$. The word ua is the longest suffix of wa occurring in w. Then, $f_x(newlast) = q$, and thus $F[newlast] = f_x(newlast)$.

Since the edge (p, a, q) is non-solid, ua is not the longest word in its own congruence class according to $\equiv_{Suff(w)}$. Theorem 7.2 applies with $z = ua$, and z' the longest word, label of the path from i to q. The class of ua according to $\equiv_{Suff(w)}$ splits into two classes according to $\equiv_{Suff(x)}$. They are represented by states q and q'.

Words v shorter than ua and such that $v \equiv_{Suff(w)} ua$ are in the form $v'a$ with $v' \in Suff(u)$ (consequence of Lemma 7.2). Before the execution of the last loop, all these words v satisfy $q = \delta(i, v)$. Therefore, after the execution of the loop, they satisfy $q' = \delta(i, v)$, as expected from Theorem 7.2. Words v longer than ua and such that $v \equiv_{Suff(w)} ua$ satisfy $q = \delta(i, v)$ after the execution of the loop, as expected from Theorem 7.2 again. It is easy to check that suffix links are correctly updated.

Finally, in the three cases (i), (ii), and (iii), the value of *last* is correctly updated at the end of procedure SA-EXTEND.

Thus, the induction proves that the sets Q and E, variable *last*, tables *Length* and F are correct after the execution of procedure SA-EXTEND.

That terminal states are correctly marked during the last loop of function SUFFIXAUTOMATON is a consequence of Corollary 7.6. □

7.3.3 Complexity. In order to analyze the complexity of the above construction, we first describe a possible implementation of elements required by the construction. We assume that the automaton is represented by adjacency lists. Doing so, the operations of adding, updating, and accessing a transition (computing $\delta(p,a)$) take $O(\log \mathrm{card}(A))$ time with an efficient implementation of adjacency lists (see Section 3.2). Function f_x is implemented by the array F that gives access to $f_x(p)$ in constant time.

For the implementation of the solid/non-solid quality of edges, we have chosen to use an array, namely *Length*, representing function $length_x$, as suggested by the description of procedure SA-EXTEND. Another possible implementation is to tie a boolean value to edges themselves. Doing so, the first edges created at steps 5 and 20 should be marked as solid. The other edges should be defined as non-solid. This type of implementation do not require the array *Length* that can be eliminated. But the array can be used in applications like the one presented in Section 7.5. Both types of implementation provide a constant-time access to the quality of edges.

Theorem 7.5. *Function* SUFFIXAUTOMATON *can be implemented to work in time $O(|x| \times \log \mathrm{card}(A))$ within $O(|x|)$ space on each given word x.*

Proof. The set of states of $\mathcal{M}(\mathit{Suff}(x))$ and arrays *Length* and F require $O(\mathrm{card}(Q))$ space. The set of adjacency lists require $O(\mathrm{card}(E))$ space. Thus, the implementation takes $O(|x|)$ space by Corollaries 7.4 and 7.5.

Another consequence of these corollaries is that all operations executed once for each state or each edge take $O(|x| \times \log \mathrm{card}(A))$ on the overall. The same result holds for operations executed once for each letter of x. So, it remains to prove that the total running time of the two loops of lines 5–6 and lines 20–21 inside procedure SA-EXTEND is also $O(|x| \times \log \mathrm{card}(A))$.

Assume that procedure SA-EXTEND is going to update $\mathcal{M}(\mathit{Suff}(w))$, w being a prefix of x. Let u be the longest word reaching state p during the test of the loop of lines 5–6. The initial value of u is $s_w(w)$, and its final value satisfies $ua = s_{wa}(wa)$ (if p is defined). Let k be the quantity $|w| - |u|$, which is the position of the suffix occurrence of u in w. Then, each test strictly increases the value of k during a single run of the procedure. Moreover, the final value of k after a run of the procedure is not greater than its initial value at the beginning of the next run. Therefore, tests and instructions of that loop are executed at most $|x|$ times.

We use a similar argument for the loop of lines 20–21 of procedure SA-EXTEND. Let v be the longest word reaching state p during the test of this loop. The initial value of v equals ${s_w}^k(w)$ for some integer $k \geqslant 2$, and its final value satisfies $va = {s_{wa}}^2(wa)$ (if p is defined). Then, the position of v as a suffix of w strictly increases at each test over all runs of the procedure. Again, tests and instructions of that loop are executed at most $|x|$ times.

Therefore, the accumulated running time of the two loops of lines 5–6 and lines 20–21 altogether is $O(|x| \times \log \mathrm{card}(A))$. Which ends the proof. □

7.4 As indexes

The suffix automaton of a word naturally provides an index on its factors. We consider four basic operations on indexes: membership, first position, number of occurrences, and list of positions. The suffix automaton also helps computing efficiently the number of factors in a word, as well as the longest factor occurring at least twice in a word.

7.4.1 Membership.

Problem 7.2. (Membership problem for *Fact*(x).) Given $w \in A^*$, find its longest prefix that belongs to *Fact*(x).

Proposition 7.2. *With $\mathcal{M}(\mathit{Suff}(x))$, computing the longest prefix u of a word w such that $u \in \mathit{Fact}(x)$ can be performed in time $O(|u| \times \log \mathrm{card}(A))$.*

Proof. Just spell the word w in $\mathcal{M}(\mathit{Suff}(x))$ considering the two implementations described in Section 7.3. Stopping the search on the first undefined transition gives the longest prefix u of w for which $\delta(i,u)$ is defined, which means that it is a factor of x. □

7.4.2 First position.

Problem 7.3. (First (respectively last) position of w in x.) Given $w \in \mathit{Fact}(x)$, find its first (respectively last) position in x.

We assume that $w \in \mathit{Fact}(x)$. This test ("does w belong to $\mathit{Fact}(x)$?") can be performed separately as in Section 7.4.1, or can be merged with the solution of the present problem.

The problem of finding the first position $\mathit{fp}_x(w)$ of w in x is equivalent to computing $\mathit{endpos}_x(w)$ because

$$\mathit{fp}_x(w) = \mathit{endpos}_x(w) - |w|.$$

Moreover, this is also equivalent to computing the maximum length of right contexts of w in x,

$$\mathit{lc}_x(w) = \max\{|z| \mid z \in w^{-1}\mathit{Fact}(x)\},$$

because

$$\mathit{fp}_x(w) = |x| - \mathit{lc}_x(w) - |w|.$$

Symmetrically, finding the last position $\mathit{lp}_x(w)$ of w in x remains to computing the smallest length $\mathit{sc}_x(w)$ of its right contexts because

$$\mathit{lp}_x(w) = |x| - \mathit{sc}_x(w) - |w|.$$

To be able to answer efficiently requests on the first or last positions of factors of x, we precompute arrays indexed by states of $\mathcal{M}(\mathit{Suff}(x))$ representing functions lc_x and sc_x. We get the next result.

Proposition 7.3. *The automaton $\mathcal{M}(\mathit{Suff}(x))$ can be preprocessed in time $O(|x|)$ so that the first (or last) position in x of any word $w \in \mathit{Fact}(x)$ can be computed in time $O(|w| \times \log \mathrm{card}(A))$ within $O(|x|)$ space.*

Proof. We consider an array LC defined on states of $\mathcal{M}(\mathit{Suff}(x))$ as follows. Let p be a state and u be such that $p = \delta(i,u)$; then, we define $LC[p] = \mathit{lc}_x(u)$. Note that the value of $LC[p]$ does not depend on the word u because for an equivalent word v, $\mathit{lc}_x(u) = \mathit{lc}_x(v)$ (by Lemma 7.2). The array LC satisfies the induction relation:

$$LC[p] = \begin{cases} 0, & \text{if } p = \mathit{last}_x, \\ 1 + \max\{LC[q] \mid q = \delta(p,a), a \in A\}, & \text{otherwise.} \end{cases}$$

So, the computation of LC can be done during a depth-first traversal of the graph of $\mathcal{M}(\mathit{Suff}(x))$. Since the total size of the graph is $O(|x|)$ (Theorem 7.3), this takes time $O(|x|)$.

To compute $\mathit{fp}_x(w)$, we first locate the state $p = \delta(i,w)$, and then return $|x| - |w| - LC[p]$. This takes the same time as for the membership problem.

To find the last occurrence of w in x we consider the array SC that represents the function sc_x. If $p = \delta(i,u)$, we set $SC[p] = \mathit{sc}_x(u)$, which is a coherent definition. We then use the next relation to compute the array during a depth-first traversal of $\mathcal{M}(\mathit{Suff}(x))$:

$$SC[p] = \begin{cases} 0, & \text{if } p \in T, \\ 1 + \min\{SC[q] \mid q = \delta(p,a), a \in A\}, & \text{otherwise.} \end{cases}$$

After the preprocessing, we get the same complexity as above. This ends the proof. □

7.4.3 Occurrence number.

Problem 7.4. (Number of occurrences of w in x.) Given $w \in \mathit{Fact}(x)$, find how many times w occurs in x.

Proposition 7.4. *The automaton $\mathcal{M}(\mathit{Suff}(x))$ can be preprocessed in time $O(|x|)$ so that the number of occurrences in x of any word $w \in \mathit{Fact}(x)$ can be computed in time $O(|w| \times \log \mathrm{card}(A))$ within $O(|x|)$ space.*

Proof. The number of occurrences of w in x is

$$\mathrm{card}\{z \mid z \in A^* \text{ and } wz \in \mathit{Suff}(x)\}.$$

If $\delta(i, w) = p$, this is also

$$\mathrm{card}\{z \mid z \in A^* \text{ and } \delta(p, z) \in T\}.$$

Let $\mathit{NB}[p]$ be this quantity, for any state of $\mathcal{M}(\mathit{Suff}(x))$.

The array NB satisfies the recurrence relation:

$$\mathit{NB}[p] = \begin{cases} 1 + \sum_{q=\delta(p,a), a \in A} \mathit{NB}[q], & \text{if } p \in T, \\ \sum_{q=\delta(p,a), a \in A} \mathit{NB}[q], & \text{otherwise,} \end{cases}$$

which shows that the array NB can be computed in time proportional to the size of the automaton during a depth-first traversal of the graph. This takes $O(|x|)$ time.

Afterwards, the problem remains to access $\mathit{NB}[p]$ for $p = \delta(i, w)$. (If p is undefined, w does not occur in x.) Computing p takes the time announced in the statement. □

An argument similar to that of the previous proof gives the computation of the number of factors occurring in x, *i.e.* the size of $\mathit{Fact}(x)$. Indeed, $\mathit{Fact}(x)$ is the particular right context associated with the initial state of $\mathcal{M}(\mathit{Suff}(x))$. And to compute its size, we evaluate contexts sizes $\mathit{CS}[p]$ of all states of the automaton using the relation:

$$\mathit{CS}[p] = \begin{cases} 1, & \text{if } p = \mathit{last}_x, \\ 1 + \sum_{q=\delta(p,a), a \in A} \mathit{CS}[q], & \text{otherwise.} \end{cases}$$

This provides a linear-time computation of $\mathrm{card}(\mathit{Fact}(x)) = \mathit{CS}[i]$.

7.4.4 List of positions.

Problem 7.5. (Positions of w in x.) Given $w \in \mathit{Fact}(x)$, produce the list of positions of w in x.

Proposition 7.5. *The automaton $\mathcal{M}(\mathit{Suff}(x))$ can be preprocessed in time $O(|x|)$ so that the list L of positions in x of any $w \in \mathit{Fact}(x)$ can be computed in time $O(|w| \times \log \mathrm{card}(A) + \mathrm{card}(L))$ within $O(|x|)$ space.*

Proof. We just sketch the proof of the statement. The automaton is preprocessed in order to create shortcuts over states on which exactly one edge is defined and that are not terminal states. To do so, we create a graph structure superimposed on the automaton. The nodes of the graph are either terminal states or states whose degree is at least two. Arcs of the graph are labeled by the labels of the corresponding path in the automaton. From a given state, labels of outgoing arcs start with pairwise distinct letters (because the automaton is deterministic).

Once the node q associated with w (or an extension of it) is found in the graph, the list of positions of w in x is computed by traversing the subgraph rooted at q. Consider the tree of the traversal. Its internal nodes have at least two children, and its leaves are associated with distinct positions (some positions can correspond to internal nodes). Therefore, the number of nodes of the tree is less than $2 \times \mathrm{card}(L)$, which proves that the time of the traversal is $O(\mathrm{card}(L))$. The extra running time is used to find q. □

7.4.5 Longest repeated factor. There are two dual problems efficiently solvable with the suffix automaton of x:

– find a longest factor repeated in x;
– find a shortest factor occurring only once in x.

Problem 7.6. (Longest repeated factor in x.) Produce a longest word $u \in \mathit{Fact}(x)$ that occurs twice in x.

If the table *NB* used to compute the number of occurrences of a factor is already computed, the problem is equivalent to find the deepest state p in $\mathcal{M}(\mathit{Suff}(x))$ for which $NB[p] > 1$. The label of the path from the initial state to p is a solution to the problem. In fact, the problem can be solved without any use of the table *NB*. We just consider the deepest state p which satisfies one of the two conditions:

(i) the degree of p is at least two;
(ii) p is a terminal state and the degree of p is at least one.

Doing so, no preprocessing on $\mathcal{M}(\mathit{Suff}(x))$ is even needed, which gives the following result.

Proposition 7.6. *With $\mathcal{M}(\mathit{Suff}(x))$, computing a longest repeated factor of x can be performed in time $O(|x|)$.*

Given a longest repeated factor u of x, ua is a factor of x for some letter a. It is clear that this word is a shortest factor occurring once only in x, *i.e.*, this word is a solution to the dual problem. Hence, the proposition also holds for the second problem.

7.5 As string-matching automata

The suffix automaton $\mathcal{M}(\mathit{Suff}(x))$ of x can be used to solve the string-matching problem, to locate the occurrences of x in a word y. The search procedure behaves like the search phase of algorithm Matcher (see Section 2) that processes y in an on-line manner. The existence of failure links in $\mathcal{M}(\mathit{Suff}(x))$ is essential for this application, which gives them their name. The search procedure is a consequence of a generic procedure, given Figure 7.10, that can be used for other purposes.

7.5.1 Ending factors. Procedure EndingFactors of Figure 7.10 computes the longest factor of x ending at each position in y, or more exactly the length of this factor. More precisely, we define for each $k \in \{0, \ldots, |y|\}$ the number

$$\ell_k = \max\{|w| \mid w \in \mathit{Suff}(y_1 y_2 \cdots y_k) \cap \mathit{Fact}(x)\}.$$

The procedure EndingFactors performs an on-line computation of the sequence $(\ell_k)_{0 \leqslant k \leqslant |y|}$ of lengths of longest ending factors. The output is given as a word on the alphabet $\{0, \ldots, |x|\}$. Function length_x of Section 7.2 (implemented via table *Length*) is used to reset properly the current length just after a suffix link has been traversed.

The core of procedure EndingFactors is the computation of transitions with the failure table F (implementing the suffix link f_x), similarly as in the general method described in Sections 3.4 and 5.4.

Theorem 7.6. *Procedure* EndingFactors *computes the lengths of longest ending factors of x in y in time $O(|y| \times \log \mathrm{card}(A))$. It executes less than $2|y|$ transitions in $\mathcal{M}(\mathit{Suff}(x))$, and requires $O(|x|)$ space.*

Proof. See the proof of Theorem 5.3. □

```
EndingFactors((Q, i, T, E), Length, F, y)
    let δ be the transition function of (Q, i, T, E)
    (ℓ, p) ← (0, i)
    L ← 0
    for letter a from first to last letter of y
      loop if δ(p, a) ≠ NIL
             then (ℓ, p) ← (ℓ + 1, δ(p, a))
             else loop    p ← F[p]
                    while p ≠ NIL and δ(p, a) = NIL
                  if p ≠ NIL
                    then (ℓ, p) ← (Length[p] + 1, δ(p, a))
                    else  (ℓ, p) ← (0, i)
           L ← L · ℓ
    return L
```

Fig. 7.10. Computing lengths of factors of a word x ending at all positions in a word y, with $(Q,i,T,E) = \mathcal{M}(\mathit{Suff}(x))$.

7.5.2 Optimization of suffix links. Indeed, instead of the suffix link f_x, we rather use another link, denoted by $\hat{f}_x$, that optimizes the delay of searches. Its definition is based on transitions defined on states of the automaton, and parallels what is done in Section 5.4.

The "follow set" of a state q of $\mathcal{M}(\mathit{Suff}(x))$ is

$$\mathit{Follow}_x(q) = \{a \mid a \in A, \delta(q,a) \text{ is defined}\}.$$

Then, $\hat{f}_x(q)$ is defined by the relation:

$$\hat{f}_x(q) = \begin{cases} f_x(q), & \text{if } \mathit{Follow}_x(f_x(q)) \not\subseteq \mathit{Follow}_x(q), \\ \hat{f}_x(f_x(q)), & \text{otherwise.} \end{cases}$$

Note that $\hat{f}_x(q)$ can be left undefined with this definition.

A property of Follow_x sets simplifies the computation of $\hat{f}_x$. In the suffix automaton we always have $\mathit{Follow}_x(q) \subseteq \mathit{Follow}_x(f_x(q))$. This is because $f_x(q)$ corresponds to a suffix v of any word u for which $q = \delta(i,u)$. Then, any letter following u in x also follows v (see Lemma 7.1). And this property transfers to follow sets of q and $f_x(q)$ respectively. With this remark, the definition of the failure function $\hat{f}_x$ can be equivalently stated as:

$$\hat{f}_x(q) = \begin{cases} f_x(q), & \text{if the degrees of } q \text{ and of } f_x(q) \text{ are different,} \\ \hat{f}_x(f_x(q)), & \text{otherwise.} \end{cases}$$

Thus, the computation of $\hat{f}_x$ has only to consider degrees of states of the automaton $\mathcal{M}(\mathit{Suff}(x))$, and can be executed in linear time.

Proposition 7.7. *For procedure* EndingFactors *using a table, say $\hat{F}$, implementing the suffix link $\hat{f}_x$ instead of the table F, the delay is $O(\mathrm{card}(A))$.*

Proof. This is a consequence of:

$$\mathit{Follow}_x(q) \subset \mathit{Follow}_x(\hat{f}_x(q)) \subseteq A$$

for any state q for which $\hat{f}_x(q)$ is defined. □

7.5.3 Searching for rotations. The knowledge of the sequence of lengths $(\ell_k)_{0 \leqslant k \leqslant |y|}$ leads to several applications such as searching for x in y, computing $\mathit{lcf}(x,y)$, the maximum length of a factor common to x and y, or computing the *subword distance* between two words:

$$d(x,y) = |x| + |y| - 2 \times \mathit{lcf}(x,y).$$

The computation of positions of x in y relies on the simple observation:

$$\ell_k = |x| \iff x \text{ occurs at position } k - |x| \text{ in } y.$$

The same remark applies as well to design an efficient solution to the next problem. A *rotation* (or a *conjugate*) of a word u is a word in the form wv, $w, v \in A^*$, when $u = vw$.

Problem 7.7. (Searching for rotations.) Given $x \in A^*$, locate all occurrences of rotations of x in any given word y.

A first solution to the problem is to apply the algorithm of Section 5 to the set of rotations of x. However, the space required by this solution can be quadratic in $|x|$ like can be the size of the corresponding trie. A solution based on suffix automata keeps the memory space linear.

Proposition 7.8. *After a preprocessing on x in time $O(|x| \times \log \mathrm{card}(A))$, positions of occurrences of rotations of x occurring in y can be computed in time $O(|y| \times \log \mathrm{card}(A))$ within $O(|x|)$ space.*

Proof. Note that the factors of length $|x|$ of the word xx are all the rotations of x. And that longer factors have a rotation of x as a suffix. (In fact, the word xuA^{-1}, where u is the shortest period of x, satisfies the same property.)

The solution consists in running the procedure EndingFactors with the automaton $\mathcal{M}(\mathit{Suff}(xx))$ after adding this modification: retain position $k - |x|$ each time $\ell_k \geqslant |x|$. Indeed, $\ell_k \geqslant |x|$ if and only if the longest factor w of xx ending at position k is not shorter than x. Thus, the suffix of length $|x|$ of w is a rotation of x. The complexity of the new procedure is the same as that of procedure EndingFactors. □

7.6 Factor automata

The *factor automaton* of a word x is the minimal deterministic automaton recognizing $\mathit{Fact}(x)$. It is denoted by $\mathcal{M}(\mathit{Fact}(x))$. It is clear that the suffix automaton $\mathcal{M}(\mathit{Suff}(x))$ recognizes $\mathit{Fact}(x)$ if all its states are transformed into terminal states. But the automaton so obtained is not always minimal. For example, the factor automaton of aabbabb, shown in Figure 7.11, is smaller than the suffix automaton of the same word (Figure 7.1). In this section we briefly review few elements related to factor automata: their relation to suffix automata, their sizes, and their construction.

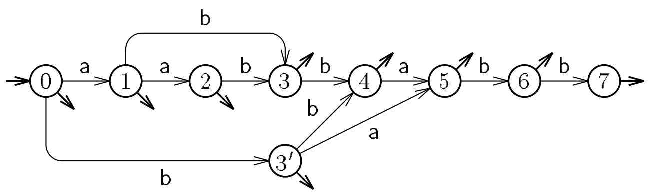


Fig. 7.11. Minimal deterministic automaton recognizing the factors of aabbabb.

7.6.1 Relation to suffix automata. The construction of factor automata by an on-line algorithm is slightly more tricky than the construction of suffix automata. The latter can be simply deduced from a procedure that builds factor automata as follows. To get $\mathcal{M}(\mathit{Suff}(x))$, first build $\mathcal{M}(\mathit{Fact}(x\$))$, extending alphabet A by letter \$, then set as terminal states only those states from which an edge by letter \$ outgoes, and finally remove all edges by letter \$ and the state they reach. The correctness of this procedure is straightforward, but is also a consequence of Theorem 7.7 below.

Conversely, the construction of $\mathcal{M}(\mathit{Fact}(x))$ from $\mathcal{M}(\mathit{Suff}(x))$ requires a minimization procedure. This is related to the non-solid path in $\mathcal{M}(\mathit{Fact}(x))$ considered in the on-line construction, and that is presented here.

Let us denote by ff_x the suffix function corresponding to the right syntactic congruence associated with $\mathit{Fact}(x)$ (and denoted by $\equiv_{\mathit{Fact}(x)}$ in this chapter). Let $z = \mathit{ff}_x(x)$ (the longest suffix of x occurring at least twice in it). Let $(p_j)_{0 \leqslant j \leqslant |z|}$ be the sequence of states of $\mathcal{M}(\mathit{Fact}(x))$ defined by $p_0 = i$, and, for $0 < j \leqslant |z|$, $p_j = \delta(p_{j-1}, z_j)$, where δ is the transition function of $\mathcal{M}(\mathit{Fact}(x))$, and i its initial state. Let k, $0 \leqslant k \leqslant |z|$, be the smallest integer for which (p_k, z_{k+1}, p_{k+1}) is a non-solid edge (setting $k = |z|$ if no such edge exists). Then, the non-solid path of $\mathcal{M}(\mathit{Fact}(x))$ is composed of edges

$$(p_k, z_{k+1}, p_{k+1}), \quad (p_{k+1}, z_{k+2}, p_{k+2}), \quad \ldots, \quad (p_{|z|-1}, z_{|z|}, p_{|z|}).$$

In equivalent terms, the word z is decomposed into uv, where $u = z_1 z_2 \cdots z_k$ and $v = z_{k+1} z_{k+2} \cdots z_{|z|}$. The word u is the longest prefix of z which is the longest word in its own congruence class according to $\equiv_{Fact(x)}$. This implies that all shorter prefixes of z satisfy the same condition while longer prefixes do not. The word v labels the non-solid path of $\mathcal{M}(Fact(x))$. It is the empty word if the non-solid path contains no edge.

With the above notion we can describe an alternative method to derive $\mathcal{M}(Suff(x))$ from $\mathcal{M}(Fact(x))$. It consists of first building $\mathcal{M}(Fact(x))$, and then duplicating states $p_{k+1}, p_{k+2}, \ldots, p_{|z|}$, of the non-solid path of $\mathcal{M}(Fact(x))$ into terminal states while creating edges and suffix links accordingly. This gives also an idea of how, by symmetry, the automaton $\mathcal{M}(Suff(x))$ can be minimized efficiently into $\mathcal{M}(Fact(x))$.

For example, in the automaton $\mathcal{M}(Fact(\mathsf{aabbabb}))$ of Figure 7.11, the non-solid path is composed of the two edges $(1, \mathsf{b}, 3)$ and $(3, \mathsf{b}, 4)$. The automaton $\mathcal{M}(Suff(\mathsf{aabbabb}))$ is obtained by cloning respectively states 3 and 4 into states $3''$ and $4''$ of Figure 7.1.

The duplication of the non-solid path labeled by v as above is implemented by the procedure of Figure 7.12. The input (r, k), which represents the non-solid path, is defined by $k = |xv^{-1}|$ and $r = \delta(i, xv^{-1})$. For example, with the automaton $\mathcal{M}(Fact(\mathsf{aabbabb}))$ of Figure 7.11 the input is $(5, 5)$.

```
FA-TO-SA(r, k)
for letter a from k + 1-st to last letter of x
  loop t ← δ(r, a)
       p ← F[t]
       q ← δ(p, a)
       q' ← STATE-CREATION
       for each letter b such that δ(q, b) ≠ NIL
         loop E ← E + {(q', b, δ(q, b))}
       Length[q'] ← Length[p] + 1
       F[t] ← q'
       F[q'] ← F[q]
       F[q] ← q'
       loop   E ← E − {(p, a, q)} + {(p, a, q')}
              p ← F[q]
         while p ≠ NIL and δ(p, a) = q
       r ← t
```

Fig. 7.12. From $\mathcal{M}(Fact(x))$ to $\mathcal{M}(Suff(x))$. It is assumed that the couple (r, k) is $(\delta(i, xv^{-1}), |xv^{-1}|)$ where v is the label of the non-solid path of $\mathcal{M}(Fact(x))$.

7.6.2 Size of factor automata. Bounds on the size of factor automata are similar to those of suffix automata. We state the results in this section. We set $(Q, i, T, E) = \mathcal{M}(Fact(x))$.

Proposition 7.9. *If* $|x| \leqslant 2$, $\mathrm{card}(Q) = |x| + 1$. *Otherwise* $|x| \geqslant 3$, *and* $|x| + 1 \leqslant \mathrm{card}(Q) \leqslant 2|x| - 2$. *If* $|x| \geqslant 4$, *the upper bound is reached only when* x *is in the form* $ab^{|x|-2}c$ *for three letters* a, b, *and* c, *such that* $a \neq b \neq c$.

Proof. The argument of the proof of Corollary 7.4 works again, except that the last letter of x yields the creation of only one state. Therefore, the upper bound on the number of states is one unit less than that of suffix automata.

By Theorem 7.2, in order to get the maximum number of states, letters $x_3, x_4, \ldots, x_{|x|-1}$ should eventually lead to the creation of two states (letters x_1, x_2, and $x_{|x|}$ cannot). This happens only when $x_1 \neq x_2$, $x_2 = x_3 = \cdots = x_{|x|-1}$. If $x_{|x|} = x_{|x|-1}$ the word x is in the form ab^{m-1}, with $m \geqslant 3$ and $a \neq b$, and its factor automaton has exactly $m + 1$ states. Therefore, we must have $x_{|x|} \neq x_{|x|-1}$ to get the $2|x| - 2$ bound, and this is also sufficient. □

Figure 7.13 displays a factor automaton having the maximum number of states for a word of length 7.

Proposition 7.10. *If* $|x| \leqslant 2$, $|x| \leqslant \mathrm{card}(E) \leqslant 2|x| - 1$. *Otherwise* $|x| \geqslant 3$, *and* $|x| \leqslant \mathrm{card}(E) \leqslant 3|x| - 4$. *If* $|x| \geqslant 4$, *the upper bound is reached only when* x *is in the form* $ab^{|x|-2}c$ *for three pairwise distinct letters* a, b, *and* c.

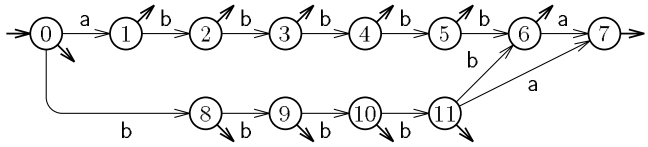


Fig. 7.13. A factor automaton with the maximum number of states.

Proof. Lemma 7.10 is still valid for factor automata, and gives the upper bound $3|x| - 4$ for $|x| \geqslant 3$. The rest can be checked by hand.

To reach the upper bound, regarding Lemma 7.10 again, $\mathcal{M}(Fact(x))$ should have the maximum number of states. Note that if x is in the form $ab^{|x|-2}a$, with $a, b \in A$ and $a \neq b$, $\mathrm{card}(E) = 2|x| - 1$ only. If x is in the form $ab^{|x|-2}c$ for three pairwise letters a, b, and c, $\mathrm{card}(E) = 3|x| - 4$. Therefore, by Proposition 7.9, this is the only possibility to reach the upper bound. □

Figure 7.14 displays a factor automaton having the maximum number of edges for a word of length 7.

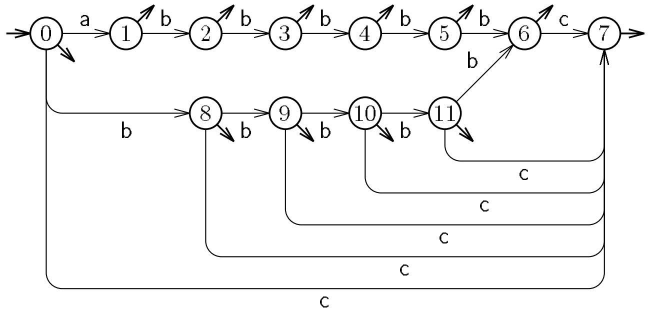


Fig. 7.14. A factor automaton with the maximum number of edges.

7.6.3 On-line construction. The on-line construction of factor automata is similar to the construction of suffix automata (Section 7.3). The main difference is that the non-solid path of the current automaton is stored and updated after the processing of each letter, and that table F implements $f\!f_x$ instead of f_x that is related to $\equiv_{Suff(x)}$. The couple of variables (r, k) represents the path as explained previously. The couple gives access to the waiting list of states that are possibly duplicated afterwards.

The function of Figure 7.15 relies on procedure FA-Extend (given in Figure 7.16) aimed at transforming the current automaton $\mathcal{M}(Fact(w))$ into $\mathcal{M}(Fact(wa))$, $a \in A$. The correctness of the function is based on a crucial property stated in the next theorem. For each nonempty word w, we denote by $\mathcal{R}(\mathcal{M}(Fact(w)))$ the automaton obtained from $\mathcal{M}(Fact(w))$ by removing the state $last_w$ and all edges reaching it.

FactorAutomaton(x)
$(Q, E) \leftarrow (\varnothing, \varnothing)$
$i \leftarrow$ State-Creation
$Length[i] \leftarrow 0$
$F[i] \leftarrow$ nil
$last \leftarrow i$
$(r, k) \leftarrow (i, 0)$
for ℓ from 1 up to x
  loop FA-Extend(ℓ)
return (Q, i, Q, E), $Length$, F

Fig. 7.15. On-line construction of the factor automaton of a word x.

Theorem 7.7. *Let $w \in A^*$, z the longest suffix of w that occurs at least twice in it, and $a \in A$. If $za \notin Fact(w)$, then, disregarding terminal states,*

$$\mathcal{R}(\mathcal{M}(Fact(wa))) = \mathcal{M}(Suff(w)).$$

Before proving the theorem, we prove the following lemma.

Lemma 7.11. *Let $w \in A^*$, z the longest suffix of w that occurs twice in it, and $a \in A$. If $za \notin Fact(w)$, then, for each $s \in A^*$, we have*

$$s \equiv_{Fact(wa)} z \quad \Longrightarrow \quad s \in Suff(z).$$

Proof. The condition on z implies that $(za)^{-1}Fact(wa) = \{\varepsilon\}$. Then, we have $(sa)^{-1}Fact(wa) = \{\varepsilon\}$, which implies that s is a suffix of w. Since z occurs twice in w, this is also the case for s (because some word ta, with $t \neq \varepsilon$, belongs to $z^{-1}Fact(wa) = s^{-1}Fact(wa)$). Therefore, by the definition of z, s is a suffix of z. □

Proof of Theorem 7.7. We prove that, for any words $u, v \in Fact(w)$,

$$u \equiv_{Fact(wa)} v \quad \Longleftrightarrow \quad u \equiv_{Suff(w)} v,$$

which is a re-statement of the conclusion of the theorem.

Assume first $u \equiv_{Fact(wa)} v$. After Lemma 7.2 we can consider, for example, that $u \in Suff(v)$. Then, $v^{-1}Suff(w) \subseteq u^{-1}Suff(w)$, and to prove $u \equiv_{Suff(w)} v$ it remains to show $u^{-1}Suff(w) \subseteq v^{-1}Suff(w)$.

Let $t \in u^{-1}Suff(w)$. We show that $t \in v^{-1}Suff(w)$. Since $ut \in Suff(w)$, we have $uta \in Suff(wa) \subseteq Fact(wa)$, and using the hypothesis, namely $u \equiv_{Fact(wa)} v$, we get that $vta \in Fact(wa)$. If ut occurs only once in w, $(ut)^{-1}Fact(w) = \{\varepsilon\}$. Therefore, $(vt)^{-1}Fact(w) = \{\varepsilon\}$ because $\equiv_{Fact(wa)}$ is a congruence, which proves that $vt \in Suff(w)$, *i.e.* $t \in v^{-1}Suff(w)$. If ut occurs at least twice in w, by definition of z, ut is a suffix of z. So, $z = z'ut$ for some prefix z' of z. Then, $z'vt \equiv_{Fact(wa)} z$, which implies that vt is a suffix of z and consequently of w by Lemma 7.11. Hence again, $t \in v^{-1}Suff(w)$. This ends the first part of the statement.

Conversely, let us consider that $u \equiv_{Suff(w)} v$, and prove $u \equiv_{Fact(wa)} v$. Without loss of generality, we assume $u \in Suff(v)$, so it remains to prove $u^{-1}Fact(wa) \subseteq v^{-1}Fact(wa)$.

Let $t \in u^{-1}Fact(wa)$. If $t = \varepsilon$, $t \in v^{-1}Fact(wa)$ because $v \in Fact(w)$. We then assume that t is a nonempty word. If $ut \in Fact(w)$, for some $t' \in A^*$, $utt' \in Suff(w)$, that is $tt' \in u^{-1}Suff(w)$. The hypothesis $u \equiv_{Suff(w)} v$ implies $tt' \in v^{-1}Suff(w)$, and consequently $t \in v^{-1}Fact(wa)$. If $vt \notin Fact(w)$, t is a suffix of wa. It can be written $t'a$ for some suffix t' of w. So, $t' \in u^{-1}Suff(w)$, and then $t' \in v^{-1}Suff(w)$ which shows that vt' is a suffix of w. Therefore, $t \in v^{-1}Fact(wa)$. This ends both the second part of the statement and the whole proof. □

During the construction of $\mathcal{M}(Fact(x))$, the following property is invariant: let $\mathcal{M}(Fact(w))$ be the current automaton and z be as in Theorem 7.7, then $\delta(i, z) = F[last]$. Consequently, the condition on z that appears in Theorem 7.7 translates into the test "$\delta(F[last], a) \neq$ NIL". If its value is TRUE, the automaton and the suffix z extend, and the procedure FA-EXTEND updates the pair (r, k) if necessary in a natural way. Otherwise, Theorem 7.7 applies, which leads to first transform $\mathcal{M}(Fact(w))$ into $\mathcal{M}(Suff(w))$. After that, the automaton is extended by the letter a. In this situation, the new non-solid path is composed of at most one edge, as a consequence of Lemma 7.6 (see also the proof of Theorem 7.7).

Finally, we state the complexity of function FACTORAUTOMATON: the construction of factor automata by this function takes linear time on a fixed alphabet.

Theorem 7.8. *Function* FACTORAUTOMATON *can be implemented to work on the input word x in time $O(|x| \times \log \mathrm{card}(A))$ within $O(|x|)$ space.*

Proof. If, for a moment, we do not consider the calls to procedure FA-TO-SA, it is rather simple to see that there is a linear number of instructions executed to built $\mathcal{M}(Fact(x))$. Implementing the automaton with adjacency lists, the cost of computing a transition is $O(\log \mathrm{card}(A))$. Which gives the $O(|x| \times \log \mathrm{card}(A))$ time for the considered instructions.

```
FA-EXTEND(ℓ)
a ← x_ℓ
if δ(F[last], a) ≠ NIL
  then newlast ← STATE-CREATION
       E ← E + {(lasta, newlast)}
       Length[newlast] ← Length[last] + 1
       F[newlast] ← δ(F[last], a)
       if k = ℓ and Length[F[last]] + 1 = Length[δ(F[last], a)]
         then (r, k) ← (newlast, ℓ + 1)
  else FA-TO-SA(r, k)
       newlast ← STATE-CREATION
       p ← last
       loop  E ← E + {(p, a, newlast)}
             p ← F[p]
         while p ≠ NIL and δ(p, a) = NIL
       if p = NIL
         then F[newlast] ← i
              (r, k) ← (newlast, ℓ + 1)
         else q ← δ(p, a)
              F[newlast] ← q
              if Length[p] + 1 = Length[q]
                then (r, k) ← (newlast, ℓ + 1)
                else (r, k) ← (last, ℓ)
last ← newlast
```

Fig. 7.16. From $\mathcal{M}(Fact(x_1 \cdots x_{\ell-1}))$ to $\mathcal{M}(Fact(x_1 \cdots x_\ell))$.

The sequence of calls to procedure FA-TO-SA behaves like a construction of $\mathcal{M}(Suff(x))$ (or just of $\mathcal{M}(Suff(x'))$, for some prefix x' of x). Thus, their accumulated running time is as announced by the theorems of Section 7.3, that is, $O(|x| \times \log \mathrm{card}(A))$.

This sketches the proof of the result. □

Bibliographic notes

Only a few books are entirely devoted to pattern matching. One can refer to [19] and [31]. The topic is treated in some books on the design of algorithms such as [3], [7], [23], [15]. An extensive bibliography is also included in [1], and the subject is partially treated in relation to automata in [29].

The notion of a failure function to represent efficiently an automaton is implicit in the work of Morris and Pratt (1970). It is intensively used in [26] and [2]. The table-compression method is explained in [4]. It is the base of some implementations of `lex` (compiler of lexical analyzer) and `yacc` (compiler of compiler) UNIX software tools that involve automata.

The regular-expression-matching problem is considered for the construction of compilers (see [4] for example). The transformation of a regular expression into an automaton is treated in standard textbooks on the subject. The construction described in Section 4 is by Thompson [32]. Many tools under the UNIX operating system use a similar method. For example the `grep` command implements the method with reduced regular expressions. While the command `egrep` operates on complete regular expressions and uses a lazy determinization of the underlying automaton.

The first linear-time solution of the string-matching problem was given by Morris and Pratt, and improved in [26]. The analogue solution to the dictionary-matching problem was designed by Aho and Corasick [2] and is implemented by the `fgrep` UNIX command.

Several authors have considered a variant of the dictionary-matching problem in which the set of words changes during the search. This is called the "dynamic-dictionary-matching problem" (see [5], [6], [25]). A related work based on suffix automata is treated in [27]. A solution to the problem restricted to uniform dictionaries is given in [11] in the comparison model of computation.

The linear size of suffix automata (also called "directed acyclic word graphs" and denoted by the acronym DAWG) and factor automata has been first noticed by Blumer *et al.*, and their efficient construction is in [9] and [16]. An alternative data structure that stores efficiently the factors (subwords) of a word is the suffix tree. It has been first introduced as a position tree by Weiner [35], but the most practical algorithms are by McCreight [28] and Ukkonen [33]. Suffix automata and suffix trees have similar applications to the implementation of indexes (inverted files), to pattern matching, and to data compression.

The solution of the string-matching problem presented in Section 6 is adapted from an original algorithm of Simon [30]. It has been analyzed and improved by Hancart [24]. These algorithms as well as algorithms in [26] solve the string-prefix-matching problem which is the computation of ending prefixes at each position on the searched word. Breslauer *et al.* [12] gave lower bounds to this problem that meet the upper bounds of Theorem 6.7. Galil showed in [20] how to transform the algorithms of Knuth, Morris and Pratt, so that the search phase works in real time (independently of the alphabet). This applies as well to the algorithm of Simon.

There are many other solutions to the string-matching problem if we relax the condition that the buffer on the searched word is reduced to only one letter. The most practically efficient solutions are based on the algorithm of Boyer and Moore [10]. With the use of automata, it has been extended to the dictionary-matching problem by Commentz-Walter [14] (see also [1]) and by Crochemore *et al.* (see [19]). The set of configurations possibly met during the search phase of a variant of the algorithm of Boyer and Moore in which all information is retained leads to what is called the "Boyer-Moore automaton". It is still unknown if the number of states of the automaton is polynomial as conjectured by Knuth (see [26, 13, 8]). Theorem 6.3 is by Galil and Seiferas [21]. Like their solution, several other proofs (in [18, 17, 22]) of the same result are based on combinatorial properties of words (see Chapter "Combinatorics of words").

Other kinds of patterns are considered in the approximate string-matching problem (see [19] for references on the subject). The first kind arises when mismatches are allowed between a given word and factors of the searched word (base of the Hamming distance). A second kind of patterns arises when approximate patterns are defined by transformation rules (substitution, insertion, and deletion) that yield the edit distance (see Chapter "String editing and DNA"). This notion is widely used for matching patterns in DNA sequences (see [34]).

References

1. A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, pages 255–300. Elsevier, Amsterdam, 1990.
2. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.
3. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, 1974.
4. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, techniques and tools.* Addison-Wesley, Reading, MA, 1986.
5. A. Amir and M. Farach. Adaptative dictionary matching. In *Proceedings of the 32th IEEE Annual Symposium on Foundations of Computer Science*, pages 760–766. IEEE Computer Society Press, 1991.
6. A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Fully dynamic dictionary matching. *Journal of Computer and System Sciences*, 49:208–222, 1994.
7. S. Baase. *Computer algorithms – Introduction to design and analysis.* Addison-Wesley, Reading, MA, 1988.
8. R. A. Baeza-Yates, C. Choffrut, and G. H. Gonnet. On Boyer-Moore automata. *Algorithmica*, 12:268–292, 1994.
9. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.
10. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.
11. D. Breslauer. Dictionary-matching on unbounded alphabets: uniform length dictionaries. *Journal of Algorithms*, 18:278–295, 1995.
12. D. Breslauer, L. Colussi, and L. Toniolo. Tight comparison bounds for the string prefix-matching problem. *Information Processing Letters*, 47:51–57, 1993.
13. V. Bruyère. *Automates de Boyer-Moore.* Thèse annexe, Université de Mons-Hainaut, Belgique, 1991.
14. B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th International Conference on Automata, Languages and Programmation*, Lecture Notes in Computer Science, pages 118–132. Springer-Verlag, Berlin, 1979.
15. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press, 1990.
16. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
17. M. Crochemore. String-matching on ordered alphabets. *Theoretical Computer Science*, 92:33–47, 1992.
18. M. Crochemore and D. Perrin. Two-way string-matching. *Journal of the ACM*, 38:651–675, 1991.
19. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.
20. Z. Galil. String matching in real time. *Journal of the ACM*, 28:134–149, 1981.
21. Z. Galil and J. Seiferas. Time-space optimal string matching. *Journal of Computer and System Sciences*, 26:280–294, 1983.
22. L. Gąsieniec, W. Plandowski, and W. Rytter. The zooming method: a recursive approach to time-space efficient string-matching. *Theoretical Computer Science*, 147:19–30, 1995.

23. G. H. Gonnet and R. A. Baeza-Yates. *Handbook of algorithms and data structures.* Addison-Wesley, Reading, MA, 1991.
24. C. Hancart. On Simon's string searching algorithm. *Information Processing Letters*, 47:95–99, 1993.
25. R. M. Idury and A. A. Schäffer. Dynamic dictionary matching with failure function. *Theoretical Computer Science*, 131:295–310, 1994.
26. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.
27. G. Kucherov and M. Rusinowitch. Matching a set of strings with variable length don't cares. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, number 937 in Lecture Notes in Computer Science, pages 230–247. Springer-Verlag, Berlin, 1995.
28. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of Algorithms*, 23:262–272, 1976.
29. D. Perrin. Finite automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 1, pages 1–57. Elsevier, Amsterdam, 1990.
30. I. Simon. String matching algorithms and automata. In *Results and Trends in Theoretical Computer Science*, number 814 in Lecture Notes in Computer Science, pages 386–395. Springer-Verlag, Berlin, 1994.
31. G. A. Stephen. *String searching algorithms.* World Scientific Press, 1994.
32. K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11:419–422, 1968.
33. E. Ukkonen. Constructing suffix trees on-line in linear time. In J. van Leeuwen, editor, *Proceedings of the IFIP 12th World Computer Congress*, pages 484–492, Madrid, 1992. North-Holland.
34. M. S. Waterman. *Introduction to computational biology.* Chapman and Hall, 1995.
35. P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Index