Pattern Matching in Strings

Maxime Crochemore, Institut Gaspard Monge, Université de Marne-la-Vallée
Christophe Hancart, Laboratoire d'Informatique de Rouen, Université de Rouen

1 Introduction

The present chapter describes a few standard algorithms used for processing texts. They apply, for example, to the manipulation of texts (text editors), to the storage of textual data (text compression), and to data retrieval systems. The algorithms of the chapter are interesting in different respects. First, they are basic components used in the implementations of practical software. Second, they introduce programming methods that serve as paradigms in other fields of computer science (system or software design). Third, they play an important role in theoretical computer science by providing challenging problems.

Although data are stored variously, text remains the main form of exchanging information. This is particularly evident in literature or linguistics where data are composed of huge corpora and dictionaries. This applies as well to computer science where a large amount of data are stored in linear files. And this is also the case in molecular biology where biological molecules can often be approximated as sequences of nucleotides or amino-acids. Moreover, the quantity of available data in these fields tends to double every eighteen months. This is the reason why algorithms should be efficient even if the speed of computers increases regularly.

The manipulation of texts involves several problems among which are: pattern matching, approximate pattern matching, comparing strings, and text compression. The first problem is partially treated in the present chapter, in that we consider only one-dimensional objects. Extensions of the methods to higher dimensional objects and solutions to the second problem appear in the chapter headed "Generalized Pattern Matching". The third problem includes the comparison of molecular sequences, and is developed in the corresponding chapter. Finally, an entire chapter is devoted to text compression.

Pattern matching is the problem of locating a collection of objects (the pattern) inside raw text. In this chapter, texts and elements of patterns are strings, which are finite sequences of symbols over a finite alphabet. Methods for searching patterns described by general regular expressions derive from standard parsing techniques (see the chapter on formal grammars and languages). We focus our attention to the case where the pattern represents a finite set of strings. Although the latter case is a specialization of the former case, it can be solved with more efficient algorithms.

Solutions to pattern matching in strings divide in two families. In the first one, the pattern is fixed. This situation occurs for example in text editors for the "search" and "substitute" commands, and in telecommunications for checking tokens. In the second family of solutions, the text is considered as fixed while the pattern is variable. This applies to dictionaries and to full-text data bases, for example.

The efficiency of algorithms is evaluated by their worst-case running times and the amount of memory space they require.

The alphabet, the finite set of symbols, is denoted by Σ, and the whole set of strings over Σ by Σ^*. The length of a string u is denoted by $|u|$; it is the length of the underlying finite sequence of symbols. The concatenation of two strings u and v is denoted by uv. A string v is said to be a factor (or a segment) of a string u if u can be written in the from $u'vu''$ where $u', u'' \in \Sigma^*$; if $i = |u'|$ and $j = |u'v| - 1$, we say that the factor v starts at position i and ends at position j in u; the factor v is also denoted by $u[i\,..\,j]$. The symbol at position i in u, that is the $i+1$-th symbol of u, is denoted by $u[i]$.

2 Matching Fixed Patterns

We consider in this section the two cases where the pattern represents a fixed string or a fixed dictionary (a finite set of strings). Algorithms search for and locate all the occurrences of the pattern in any text.

In the string-matching problem, the first case, it is convenient to consider that the text is examined through a window. The window delimits a factor of the text and has usually the length of the pattern. It slides along the text from left to right. During the search, it is periodically shifted according to rules that are specific to each algorithm. When the window is at a certain position on the text, the algorithm checks whether the pattern occurs there or not, by comparing some symbols in the window with the corresponding aligned symbols of the pattern; if there is a whole match, the position is reported. During this scan operation, the algorithm acquires from the text information which are often used to determine the length of the next shift of the window. Some part of the gathered information can also be memorized in order to save time during the next scan operation.

In the dictionary-matching problem, the second case, methods are based on the use of automata, or related data structures.

2.1 The Brute Force Algorithm

The simplest implementation of the sliding window mechanism is the brute force algorithm. The strategy consists here in sliding uniformly the window one position to the right after each scan operation. As far as scans are correctly implemented, this obviously leads to a correct algorithm.

We give below the pseudocode of the corresponding procedure. The inputs are a nonempty string x, its length m (thus $m \geqslant 1$), a string y, and its length n. The variable p in the procedure corresponds to the current left position of the window on the text. It is understood that the string-to-string comparison in line 2 has to be processed symbol per symbol according to a given order.

```
BRUTE-FORCE-MATCHER(x, m, y, n)
1  for p from 0 up to n − m
2     loop if y[p..p + m − 1] = x
3              then report p
```

The time complexity of the brute force algorithm is $O(m \times n)$ in the worst case (for instance when $a^{m-1}b$ is searched in a^n for any two symbol $a, b \in \Sigma$ satisfying $a \neq b$ if we assume that the rightmost symbol in the window is compared last). But its behavior is linear in n when searching in random texts.

2.2 The Karp-Rabin Algorithm

Hashing provides a simple method for avoiding a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position p of the window on the text whether the pattern occurs here or not, it seems to be more efficient to check only if the factor of the text delimited by the window, namely $y[p\,..\,p+m-1]$, "looks like" x. In order to check the resemblance between the two strings, a hash function is used. But, to be helpful for the string-matching problem, the hash function should be highly discriminating for strings. According to the running times of the algorithms, the function should also have the following properties:

- to be efficiently computable;
- to provide an easy computation of the value associated with the next factor from the value associated with the current factor.

The last point is met when symbols of alphabet Σ are assimilated with integers and when the hash function, say h, is defined for each string $u \in \Sigma^*$ by

$$h(u) = \left(\sum_{i=0}^{|u|-1} u[i] \times d^{|u|-1-i} \right) \bmod q,$$

where q and d are two constants. Then, for each string $v \in \Sigma^*$, for each symbols $a', a'' \in \Sigma$, $h(va'')$ is computed from $h(a'v)$ by the formula

$$h(va'') = ((h(a'v) - a' \times d^{|v|}) \times d + a'') \bmod q.$$

During the search for pattern x, it is enough to compare the value $h(x)$ with the hash value associated with each factor of length m of text y. If the two values are equal, that is, in case of collision, it is still necessary to check whether the factor is equal to x or not by symbol comparisons.

The underlying string-matching algorithm, which is denoted as the Karp-Rabin algorithm, is implemented below as the procedure Karp-Rabin-Matcher. In the procedure, the values $d^{m-1} \bmod q$, $h(x)$, and $h(y[0\,..\,m-2])$ are first precomputed, and stored respectively in the variables r, s, and t (lines 1–7). The value of t is then recomputed at each step of the search phase (lines 8–12). It is assumed that the value of symbols ranges from 0 to $c-1$; the quantity $(c-1) \times q$ is added in line 8 to provide correct computations on positive integers.

Karp-Rabin-Matcher(x, m, y, n)
1 $r \leftarrow 1$
2 $s \leftarrow x[0] \bmod q$
3 $t \leftarrow 0$
4 **for** i from 1 up to $m-1$
5 **loop** $r \leftarrow (r \times d) \bmod q$
6 $s \leftarrow (s \times d + x[i]) \bmod q$
7 $t \leftarrow (t \times d + y[i-1]) \bmod q$
8 **for** p from 0 up to $n-m$
9 **loop** $t \leftarrow (t \times d + y[p+m-1]) \bmod q$
10 **if** $t = s$ and $y[p\,..\,p+m-1] = x$
11 **then** report p
12 $t \leftarrow ((c-1) \times q + t - y[p] \times r) \bmod q$

Convenient values for d are powers of 2; in this case, all the products by d can be computed as shifts on integers. The value of q is generally a large prime (such that the quantities $(q-1) \times d + c - 1$ and $c \times q - 1$ do not cause overflows), but it can also be the value of the implicit modulus supported by integer operations. An illustration of the behavior of the algorithm is given in Figure 1.

p	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$y[p]$	`n`	`o`	␣	`d`	`e`	`f`	`e`	`n`	`s`	`e`	␣	`f`	`o`	`r`	␣	`s`	`e`	`n`	`s`	`e`
$h(y[p\,..\,p+4])$	8	8	6	28	9	18	28	26	22	12	17	24	16	0	1	9	—	—	—	—

Figure 1 An illustration of the behavior of the Karp-Rabin algorithm when searching for the pattern x = `sense` in the text y = `no␣defense␣for␣sense`. Here, symbols are assimilated with their ASCII codes (hence $c = 256$), and the values of q and d are set respectively to 31 and 2. This is valid for example when the maximal integer is $2^{16} - 1$. The value of $h(x)$ is $(115 \times 16 + 101 \times 8 + 110 \times 4 + 115 \times 2 + 101) \bmod 31 = 9$. Since only $h(y[4\,..\,8])$ and $h(y[15\,..\,19])$ among the defined values of $h(y[p\,..\,p+4])$ are equal to $h(x)$, two string-to-string comparisons against x are performed.

The worst case complexity of the above string-matching algorithm is quadratic, as it is for the brute force algorithm, but its expected running time is $O(m+n)$ if parameters q and d are adequate.

2.3 The Knuth-Morris-Pratt Algorithm

This section presents the first discovered linear-time string-matching algorithm. Its design follows a tight analysis of a version of the brute force algorithm in which the string-to-string comparison proceeds from left to right. The brute force algorithm wastes the information gathered during the scan of the text. On the contrary, the Knuth-Morris-Pratt algorithm stores the information with two purposes. First, it is used to improve the length of shifts. Second, there is no backward scan of the text.

Consider a given position p of the window on the text. Assume that a mismatch occurs between symbols $y[p+i]$ and $x[i]$ for some i, $0 \leqslant i < m$ (an illustration is given in Figure 2). Thus, we have $y[p\,..\,p+i-1] = x[0\,..\,i-1]$ and $y[p+i] \neq x[i]$. With regard to the information given by $x[0\,..\,i-1]$, interesting shifts are necessarily connected with the borders of $x[0\,..\,i-1]$. (A border of a string u is a factor of u that is both a prefix and a suffix of u). Among the borders of $x[0\,..\,i-1]$, the longest proper border followed by a symbol different from $x[i]$ is the best possible candidate, subject to the existence of such of a border. (A factor v of a string u is said to be a proper factor of u if u and v are not identical, that is, if $|v| < |u|$.) This introduces the function ψ defined for each $i \in \{0, 1, \ldots, m-1\}$ by

$$\psi[i] = \max\{k \mid (0 \leqslant k < i,\ x[i-k\,..\,i-1] = x[0\,..\,k-1],\ x[k] \neq x[i]) \text{ or } (k = -1)\}.$$

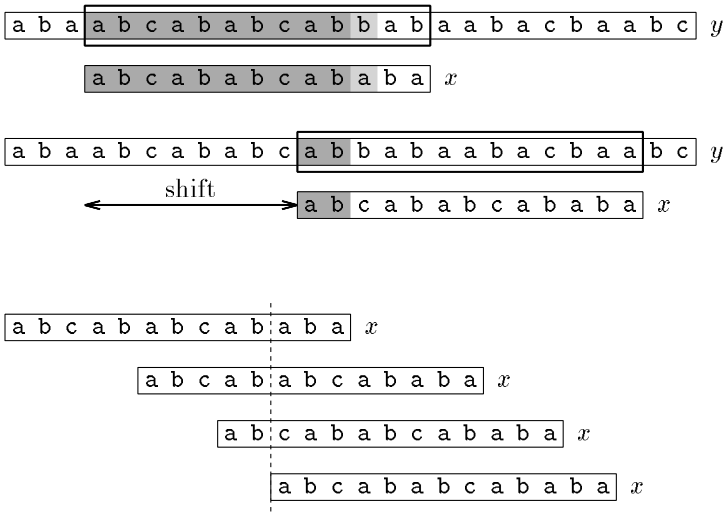
Then, after a shift of length $i - \psi[i]$, the symbol comparisons can resume with $y[p+i]$ against $x[\psi[i]]$ in the case where $\psi[i] \geqslant 0$, and $y[p+i+1]$ against $x[0]$ otherwise. Doing so, we miss no occurrence of x in y, and avoid a backtrack on the text. The previous statement is still valid when no mismatch occurs, that is when $i = m$, if we consider for a moment the string $x\$$ instead of x, where $\$$ is a symbol of alphabet Σ occurring nowhere in x. This amounts to completing the definition of function ψ by setting

$$\psi[m] = \max\{k \mid 0 \leqslant k < m,\ x[m-k\,..\,m-1] = x[0\,..\,k-1]\}.$$

The Knuth-Morris-Pratt string-matching algorithm is given in pseudocode below as the procedure Knuth-Morris-Pratt-Matcher. The values of function ψ are first computed by the function Better-Prefix-Function given after. The value of the variable j is equal to $p + i$ in the remainder of the code (the search phase of the algorithm strictly speaking); this simplifies the code, and points out the sequential processing of the text. Observe that the preprocessing phase applies a similar method to the pattern itself, as if $y = x[1\,..\,m-1]$.

(a)

(b)



(c)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$x[i]$	a	b	c	a	b	a	b	c	a	b	a	b	a	—
$\psi[i]$	-1	0	0	-1	0	2	0	0	-1	0	2	0	2	1

Figure 2 An illustration of the shift in the Knuth-Morris-Pratt algorithm when searching for the pattern x = abcababcababa. **(a)** The window on the text y is at position 3. A mismatch occurs at position 10 on x. The matching symbols are shown darkly shaded, and the current analyzed symbols lightly shaded. Avoiding both a backtrack on the text and an immediate mismatch leads to shift the window 8 positions to the right. The string-to-string comparison resumes at position 2 on the pattern. **(b)** The current shift is the consequence of an analysis of the list of the proper borders of $x[0\,..\,9]$ and of the symbol which follow them in x. The prefixes of x that are borders of $x[0\,..\,9]$ = abcabacbab are right-aligned along the discontinuous vertical line. String $x[0\,..\,4]$ = abcab is a border of $x[0\,..\,9]$, but is followed by symbol a which is identical to $x[10]$. String $x[0\,..\,1]$ is the expected border, since it is followed by symbol c. **(c)** The values of the function ψ for pattern x.

```
KNUTH-MORRIS-PRATT-MATCHER(x, m, y, n)
ψ ← BETTER-PREFIX-FUNCTION(x, m)
i ← 0
for j from 0 up to n − 1
  loop while i ⩾ 0 and y[j] ≠ x[i]
         loop i ← ψ[i]
       i ← i + 1
       if i = m
         then report j + 1 − m
              i ← ψ[m]
```

```
BETTER-PREFIX-FUNCTION(x, m)
ψ[0] ← −1
i ← 0
for j from 1 up to m − 1
  loop if x[j] = x[i]
         then ψ[j] ← ψ[i]
         else ψ[j] ← i
              loop   i ← ψ[i]
                while i ⩾ 0 and x[j] ≠ x[i]
       i ← i + 1
ψ[m] ← i
return ψ
```

The algorithm has a worst-case running time in $O(m+n)$, and requires $O(m)$ extra-space to store function ψ. The linear running time results from the fact that the number of symbol comparisons performed during the preprocessing phase and the search phase is less than $2m$ and $2n$ respectively. All the previous bounds are independent of the size of the alphabet.

2.4 The Boyer-Moore Algorithm

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it, or the entire algorithm, is often implemented in text editors for the "search" and "substitute" commands.

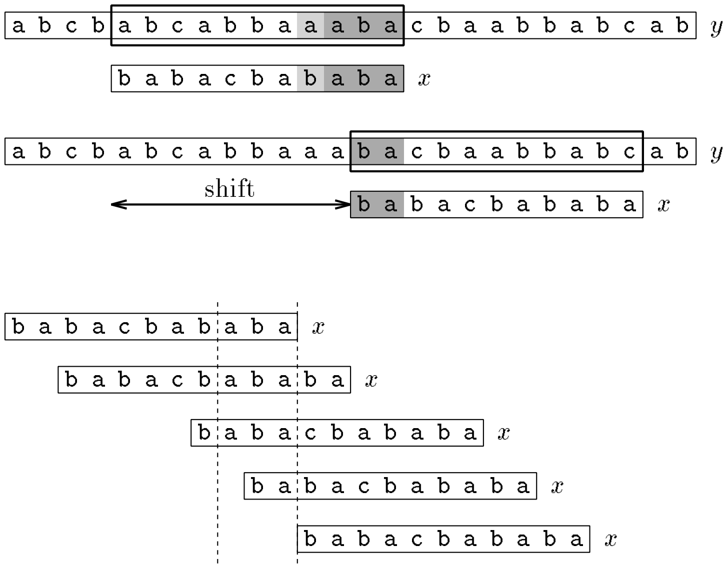
The scan operation proceeds from right to left in the window on the text, instead of left to right as in the Knuth-Morris-Pratt algorithm. In case of a mismatch, the algorithm uses two functions to shift the window. These two shift functions are called the better-factor shift function and the bad-symbol shift function. In the two next paragraphs, we explain the goal of the two functions and we give procedures to precompute their values.

We first explain the aim of the better-factor shift function. Let p be the current (left) position of the window on the text. Assume that a mismatch occurs between symbols $y[p+i]$ and $x[i]$ for some i, $0 \leqslant i < m$ (an illustration is given in Figure 3). Then, we have $y[p+i] \neq x[i]$ and $y[p+i+1\,..\,p+m-1] = x[i+1\,..\,m-1]$. The better-factor shift consists in aligning the factor $y[p+i+1\,..\,p+m-1]$ with its rightmost occurrence $x[k+1\,..\,m-1-i+k]$ in x preceded by a symbol $x[k]$ different from $x[i]$ to avoid an immediate mismatch. If no such factor exists, the shift consists in aligning the longest suffix of $y[p+i+1\,..\,p+m-1]$ with a matching prefix of x. The better-factor shift function β is defined by

$$\begin{aligned}\beta[i] = \min\{i-k \mid\ & (0 \leqslant k < i,\ x[k+1\,..\,m-1-i+k] = x[i+1\,..\,m-1],\ x[k] \neq x[i]) \\ & \text{or } (i-m \leqslant k < 0,\ x = x[i-k\,..\,m-1]x[m-i+k\,..\,m-1])\}\end{aligned}$$

(a)

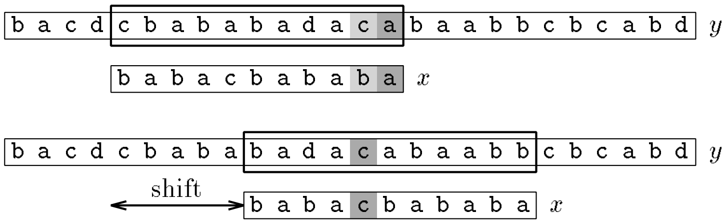
(b)



(c)

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	b	a	b	a	c	b	a	b	a	b	a
$\beta[i]$	7	7	7	7	7	7	2	9	4	11	1

Figure 3 An illustration of the better-factor shift in the Boyer-Moore algorithm when searching for the pattern x = `babacbababa`. **(a)** The window on the text is at position 4. The string-to-string comparison, which proceeds from right to left, stops with a mismatch at position 7 on x. The window is shifted 9 positions to the right to avoid an immediate mismatch. **(b)** Indeed, the string $x[8\,..\,10]$ = `aba` is repeated three times in x, but is preceded each time by symbol $x[7]$ = `b`. The expected matching factor in x is then the prefix `ba` of x. The factors of x identical with `aba` and the prefixes of x ending with a suffix of `aba` are right-aligned along the rightmost discontinuous vertical line. **(c)** The values of the shift function β for pattern x.



a	a	b	c	d
$\omega[a]$	2	1	6	11

(a) **(b)**

Figure 4 An illustration of the bad-symbol shift in the Boyer-Moore algorithm when searching for the pattern $x =$ `babacbababa`. **(a)** The window on the text is at position 4. The string-to-string comparison stops with a mismatch at position 9 on x. Considering only this position and the unexpected symbol occurring at this position, namely symbol $x[9] =$ `c`, leads to shift the window 5 positions to the right. Notice that if the unexpected symbol were `a` or `d`, the applied shift would have been 1 and 10 respectively. **(b)** The values of the table ω for pattern x when alphabet Σ is reduced to $\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$.

for each $i \in \{0, 1, \ldots, m-1\}$. The value $\beta[i]$ is then exactly the length of the shift induced by the better-factor shift. The values of function β are computed by the function given below as the function Better-Factor-Function. An auxiliary table, namely f, is used; it is an analogue of the function ψ used in the Knuth-Morris-Pratt algorithm, but defined this time for the reverse pattern; it is indexed from 0 to $m-1$. The running time of the function Better-Factor-Function is $O(m)$.

```
BETTER-FACTOR-FUNCTION(x, m)
for j from 0 up to m − 1
  loop β[j] ← 0
i ← m
for j from m − 1 down to 0
  loop f[j] ← i + 1
       while i < m and x[j] ≠ x[i]
         loop if β[i] = 0
                then β[i] ← i − j
              i ← f[i] − 1
       i ← i − 1
for j from 0 up to m − 1
  loop if β[j] = 0
         then β[j] ← i + 1
       if j = i
         then i ← f[i] − 1
return β
```

We now come to the aim of the bad-symbol shift function (Figure 4 shows an illustration). Consider again the text symbol $y[p+i]$ that causes a mismatch. Assume first that this symbol occurs in $x[0\,..\,m-2]$. Then, let k be the position of the rightmost occurrence of $y[p+i]$ in $x[0\,..\,m-2]$. The window can be shifted $i-k$ positions to the right if $k < i$, and only one position otherwise, without missing an occurrence of x in y. Assume now that symbol $y[p+i]$ does not occur in x. Then, no occurrence of x in y can overlap the position $p+i$ on the text, and thus, the window can be shifted $i+1$ positions to the right. Let ω be the table indexed on alphabet Σ, and defined for each symbol $a \in \Sigma$ by

$$\omega[a] = \min\{m\} \cup \{m-1-j \mid 0 \leqslant j < m-1,\, x[j] = a\}.$$

According to the above discussion, the bad-symbol shift for the unexpected text symbol a aligned with the symbol at position i on the pattern is the value

$$\gamma[a, i] = \max\{\omega[a] + i - m + 1,\ 1\},$$

which defines the bad-symbol shift function γ on $\Sigma \times \{0, 1, \ldots, m-1\}$. We give now the code of the function LAST-OCCURRENCE-FUNCTION that computes table ω. Its running time is $O(m + \operatorname{card} \Sigma)$.

```
LAST-OCCURRENCE-FUNCTION(x, m)
for each a ∈ Σ
   loop ω[a] ← m
for j from 0 up to m − 2
   loop ω[x[j]] ← m − 1 − j
return ω
```

The shift applied in the Boyer-Moore algorithm in case of a mismatch is the maximum between the better-factor shift and the bad-symbol shift. In case of a whole match, the shift applied to the window is m minus the length of the longest proper border of x, that is also the value $\beta[0]$ (this value is indeed what is called "the period" of the pattern). The code of the entire algorithm is given below.

```
BOYER-MOORE-MATCHER(x, m, y, n)
β ← BETTER-FACTOR-FUNCTION(x, m)
ω ← LAST-OCCURRENCE-FUNCTION(x, m)
p ← 0
while p ⩽ n − m
   loop i ← m − 1
        while i ⩾ 0 and y[p + i] = x[i]
           loop i ← i − 1
        if i ⩾ 0
           then p ← p + max{β[i], ω[y[p + i]] + i − m + 1}
           else report p
                p ← p + β[0]
```

The worst-case running time of the algorithm is quadratic. It is surprising however that, when used to search only for the first occurrence of the pattern, the algorithm runs in linear time. Slight modifications of the strategy yield linear-time algorithms. When searching for $a^{m-1}b$ in a^n with $a, b \in \Sigma$ and $a \neq b$, the algorithm considers only $\lfloor n/m \rfloor$ symbols of the text. This bound is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed. Indeed, the algorithm is expected to be extremely fast on large alphabets (relative to the length of the pattern).

2.5 Practical String-Matching Algorithms

The bad-symbol shift function introduced in the Boyer-Moore algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern (as it is often the case with the ASCII table and ordinary searches made under a text editor), it becomes very useful. Using only the corresponding table produces some efficient algorithms for practical searches. We describe one of these algorithms below.

Consider a position p of the window on the text, and assume that the symbols $y[p+m-1]$ and $x[m-1]$ are identical. If $x[m-1]$ does not occur in the prefix $x[0\,..\,m-2]$ of x, the window can be shifted m positions to the right after the string-to-string comparison between $y[p\,..\,p+m-2]$ and $x[0\,..\,m-2]$ is performed. Otherwise, let k be the position of the rightmost occurrence of $x[m-1]$ in $x[0\,..\,m-2]$; the window can be shifted $m-1-k$ positions to the right. This shows that $\omega[y[p+m-1]]$ is also a valid shift in the case where $y[p+m-1] = x[m-1]$. The underlying algorithm is the Horspool algorithm.

The pseudocode of the Horspool algorithm is given below. To prevent two references to the rightmost symbol in the window at each scan and shift operation, table ω is slightly modified: $\omega[x[m-1]]$ contains the sentinel value 0, after its previous value is saved in variable t. The value of the variable j is the value of the expression $p+m-1$ in the discussion above.

```
Horspool-Matcher(x, m, y, n)
ω ←Last-Occurrence-Function(x, m)
t ← ω[x[m − 1]]
ω[x[m − 1]] ← 0
j ← m − 1
while j < n
   loop s ← ω[y[j]]
          if s ≠ 0
             then j ← j + s
             else  if y[j − m + 1 . . j − 1] = x[0 . . m − 2]
                       then report j − m + 1
                    j ← j + t
```

Just like the brute force algorithm, the Horspool algorithm has a quadratic worst-case time complexity. But its behavior in practice is at least as good as the behavior of the Boyer-Moore algorithm is. An example showing the behavior of both algorithms is given in Figure 5.

2.6 The Aho-Corasick Algorithm

The UNIX operating system provides standard text-file facilities. Among them is the series of `grep` commands that locate patterns in files. We describe in this section the Aho-Corasick algorithm underlying an implementation of the `fgrep` command of UNIX. It searches files for a finite and fixed set of strings (the dictionary), and can for instance output lines containing at least one of the strings.

If we are interested in searching for all occurrences of all strings of a dictionary, a first solution consists in repeating some string-matching algorithm for each string. Considering a dictionary X containing k strings and a text y, the search runs in that case in time $O(m+n\times k)$, where m is the sum of the length of the strings in X, and n the length of y. But this solution is not efficient, since text y has to be read k times. The solution described in this section provides both a sequential read of the text and a total running time which is $O(m+n)$ on a fixed alphabet. The algorithm can be viewed as a direct extension of weaker version of the Knuth-Morris-Pratt algorithm.

The search is done with the help of an automaton that stores the situations encountered during the process. At a given position on the text, the current state is identified with the set of pattern prefixes ending here. The state represents all the factors of the pattern that can possibly lead to occurrences. Among the factors, the longest contains all the information necessary to continue the search. So, the search is realized with an automaton, denoted by

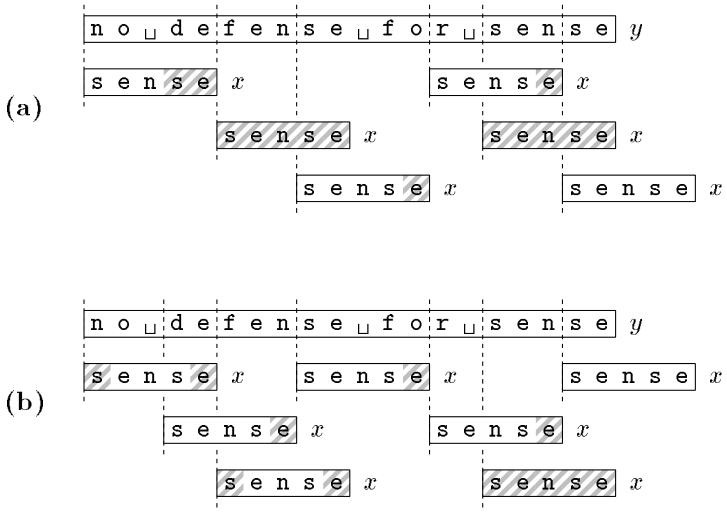


Figure 5 An illustration of the behavior of two fast string-matching algorithms when searching for the pattern $x = \texttt{sense}$ in the text $y = \texttt{no␣defense␣for␣sense}$. The successive positions of the window on the text are suggested by the alignments of x with the corresponding factors of y. The symbols of x considered during each scan operation are shown hachured. **(a)** Behavior of the Boyer-Moore algorithm. The first and second shifts result from the better-shift function, the third and fourth from the bad-symbol function, and the fifth from a shift of the length of x minus the length of its longest proper border (the period of x). **(b)** Behavior of the Horspool algorithm. We assume here that the four leftmost symbols in the window are compared with the symbols of $x[0\,..\,3]$ from left to right.

$\mathcal{D}(X)$, of which states are in one-to-one correspondence with the prefixes of X. Implementing completely the transition function of $\mathcal{D}(X)$ would required a size $O(m \times \operatorname{card}\Sigma)$. Instead of that, the Aho-Corasick algorithm requires only $O(m)$ space. To get this space complexity, a part of the transition function is made explicit in the data, and the other part is computed with the help of a failure function. For the first part, we assume that for any input (p, a), the function denoted by TARGET returns some state q if the triple (p, a, q) is an edge in the data, and the value NIL otherwise. The second part uses the failure function *fail*, which is an analogue of the function ψ used in the Knuth-Morris-Pratt algorithm. But this time, the function is defined on the set of states, and for each state p different from the initial state,

fail$[p]$ = the state identified with the longest proper suffix of the prefix identified with p that is also a prefix of a string of X.

The aim the failure function is to defer the computation of a transition from the current state, say p, to the computation of the transition from the state *fail*$[p]$ with the same input symbol, say a, when no edge from p labeled by symbol a is in the data; the initial state, which is identified with the empty string, is the default state for the statement. We give below the pseudocode of the function NEXT-STATE that computes the transitions in the representation. The initial state is denoted by i.

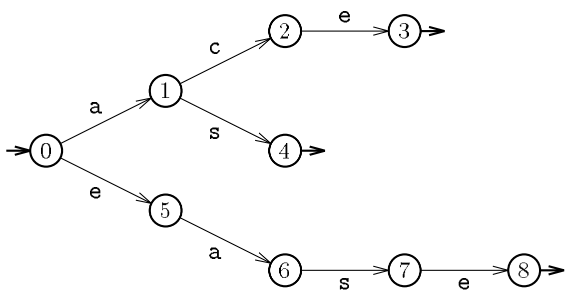


Figure 6 The trie-like automaton of the pattern $X = \{\texttt{ace}, \texttt{as}, \texttt{ease}\}$. The initial state is distinguished by a thick ingoing arrow, each terminal state by a thick outgoing arrow. The states are numbered from 0 to 8, according to the order in which they are created by the construction statement described in the present section. State 0 is identified with the empty string, state 1 with `a`, state 2 with `ac`, state 3 with `ace`, and so on. The automaton accepts the language X.

```
NEXT-STATE(p, a, i)
while p ≠ NIL and TARGET(p, a) = NIL
   loop p ← fail[p]
if p ≠ NIL
   then q ← TARGET(p, a)
   else q ← i
return q
```

The preprocessing phase of the Aho-Corasick algorithm builds the explicit part of $\mathcal{D}(X)$ including function *fail*. It is divided itself into two phases.

The first phase of the preprocessing phase consists in building a sub-automaton of $\mathcal{D}(X)$. It is the trie of X (the digital tree in which branches spell the strings of X and edges are labeled by symbols) having as initial state the root of the trie and as terminal states the nodes corresponding to strings of X (an example is given in Figure 6). It differs from $\mathcal{D}(X)$ in two points:

- it contains only the forward edges;
- it accepts only the set X.

(An edge (p, a, q) in the automaton is said to be forward if the prefix identified with q is in the form ua where u is the prefix corresponding to p.) The function given below as the function TRIE-LIKE-AUTOMATON computes the automaton corresponding to the trie of X by returning its initial state. The terminal mark of each state r is managed through the attribute *terminal*$[r]$; the mark is either TRUE or FALSE depending on whether state r is terminal or not. We assume that the function NEW-STATE creates and returns a new state, and that the procedure MAKE-EDGE adds a given new edge to the data.

```
Trie-Like-Automaton(X)
i ←New-State
terminal[i] ← false
for string x from first to last string of X
   loop p ← i
         for symbol a from first to last symbol of x
            loop q ←Target(p, a)
                  if q = nil
                     then q ←New-State
                           terminal[q] ← false
                           Make-Edge(p, a, q)
                  p ← q
         terminal[p] ← true
return i
```

The second step of the preprocessing phase consists mainly in precomputing the failure function. This is done by a breadth-first traversal of the trie-like automaton. The corresponding pseudocode is given below as the procedure Make-Failure-Function.

```
Make-Failure-Function(i)
fail[i] ← nil
θ ←Empty-Queue
Enqueue(θ, i)
while not Queue-Is-Empty(θ)
   loop p ←Dequeue(θ)
         for each symbol a such that Target(p, a) ≠ nil
            loop q ←Target(p, a)
                  fail[q] ←Next-State(fail[p], a, i)
                  if terminal[fail[q]]
                     then terminal[q] ← true
                  Enqueue(θ, q)
```

During the computation, some states can be made terminal. This occurs when the state is identified with a prefix that ends with a string of X (an illustration is given in Figure 7).

The complete dictionary-matching algorithm, implemented in the pseudocode below as the procedure Aho-Corasick-Matcher, starts with the two steps of the preprocessing; the search follows, which simulates automaton $\mathcal{D}(X)$. It is understood that the empty string does not belong to X.

```
Aho-Corasick-Matcher(X, y)
i ←Trie-Like-Automaton(X)
Make-Failure-Function(i)
p ← i
for symbol a from first to last symbol of y
   loop p ←Next-State(p, a, i)
         if terminal[p]
            then report an occurrence
```

The total number of tests "Target$(p, a) =$ nil" performed by function Next-State during its calls by procedure Make-Failure-Function and during its calls by the search phase of

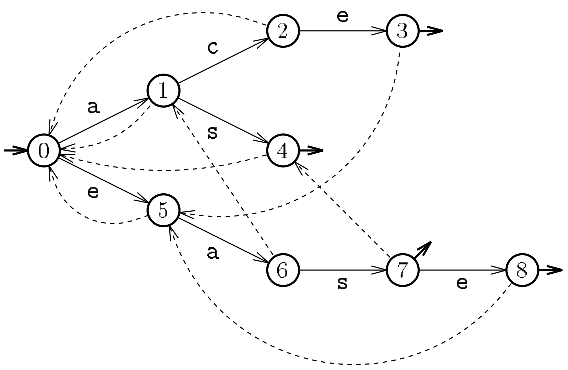


Figure 7 The explicit part of the automaton $\mathcal{D}(X)$ of the pattern $X = \{\texttt{ace}, \texttt{as}, \texttt{ease}\}$. Compared to the trie-like automaton of X displayed in Figure 6, state 7 has been made terminal; this is because the corresponding prefix, namely `eas`, ends with the string `as` that is in X. The failure function *fail* is depicted with discontinuous non-labeled directed edges.

the algorithm are bounded by $2m$ and $2n$ respectively, similarly as the bounds of comparisons in the Knuth-Morris-Pratt algorithm. Using a total order on the alphabet, the running time of function Target is both $O(\log k)$ and $O(\log \operatorname{card} \Sigma)$, since the maximum number of edges outgoing a state in the data representing automaton $\mathcal{D}(X)$ is bounded both by k and by $\operatorname{card} \Sigma$. Thus, the entire algorithm runs in time $O(m+n)$ on a fixed alphabet, and in time $O((m+n) \times \log \min\{k, \operatorname{card} \Sigma\})$ in the general case. The algorithm requires $O(m)$ extra-space to store the data and to implement the queue used during the breadth-first traversal executed in procedure Make-Failure-Function.

Let us discuss the question of reporting occurrences of pattern X (line 7 of procedure Aho-Corasick-Matcher). The simplest way of doing it is to report the ending positions of occurrences. This remains to output the value of the position of the current symbol in the text. A second possibility is to report the whole set of strings in X ending at the current position. To do so, the attribute *terminal* has to be transformed. First, for a state r, *terminal*$[r]$ is the set of the string of X that are suffixes of the string corresponding to r. Second, to avoid a quadratic behavior, sets are manipulated by their identifiers only.

3 Indexing Texts

This section deals with the pattern-matching problem applied to fixed texts. Solutions consist in building an index on the text that speeds up further searches. The indexes that we consider here are data structures that contain all the suffixes and therefore all the factors of the text. Two types of structures are presented: suffix trees and suffix automata. They are both compact representations of suffixes in the sense that their sizes are linear in the length of the text, although the sum of lengths of suffixes of a string is quadratic. Moreover, their constructions take linear time on fixed alphabets. On an arbitrary finite alphabet Σ, assumed to be ordered, a $\log \operatorname{card} \Sigma$ factor has to be added to almost all running times given in the following. This corresponds to the branching operation involved in the respective data structures.

Indexes are powerful tools that have many applications. Here is a non-exhaustive list of them, assuming an index on the text y.

- Membership: testing if a string x occurs in y.
- Occurrence number: producing the number of occurrences of a string x in y.

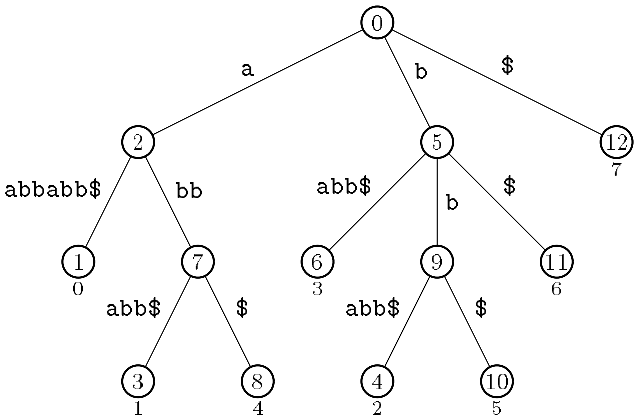


Figure 8 The suffix tree $\mathcal{T}(y)$ of the string y = `aabbabb$`. The nodes are numbered from 0 to 12, according to the order in which they are created by the construction algorithm described in the present section. Each of the eight external nodes of the trie is marked by the position of the occurrence of the corresponding suffix in y. Hence, the branch $(0, 5, 9, 4)$, running from the root to an external node, spells the string `bbabb$`, which is the suffix of y starting at position 2.

- List of positions: analogue of the string-matching problem of Section 2.
- Longest repeated factor: locating the longest factor of y occurring at least twice in y.
- Longest common factor: finding a longest string that occurs both in a string x and in y.

Solutions to some of these problems are first considered with suffix trees, then with suffix automata.

3.1 Suffix Trees

The suffix tree $\mathcal{T}(y)$ of a nonempty string y of length n is a data structure containing all the suffixes of y. In order to simplify the statement, it is assumed that y ends with a special symbol of the alphabet occurring nowhere else in y (this special symbol is denoted by `$` in the examples). The suffix tree of y is a trie which satisfies the following properties:

- the branches from the root to the external nodes spell the nonempty suffixes of y, and each external node is marked by the position of the occurrence of the corresponding suffix in y;
- the internal nodes have at least two successors, except if y is a one-length string;
- the edges outgoing an internal node are labeled by factors starting with different symbols;
- any string that labels an edge is represented by the couple of integers corresponding to its position in y and its length.

(An example of suffix tree is displayed in Figure 8.) The special symbol at the end of y avoids marking nodes, and implies that $\mathcal{T}(y)$ has exactly n external nodes. The other properties then imply that the total size of $\mathcal{T}(y)$ is $O(n)$, which makes it possible to design a linear-time construction of the data structure. The algorithm described in the following and implemented by the procedure Suffix-Tree given further has this time complexity.

The construction algorithm works as follows. It inserts the nonempty suffixes $y[i \mathinner{..} n-1]$, $0 \leqslant i < n$, of y in the data structure from the longest to the shortest suffix. In order to explain how this is performed, we introduce the two notations

$h_i =$ the longest prefix of $y[i \mathinner{..} n-1]$ that is a prefix of some stricly longest suffix of y,

and

$t_i =$ the string w such that $y[i \mathinner{..} n-1]$ is identical with $h_i w$,

defined for each $i \in \{1, \ldots, n-1\}$. The strategy to insert the suffixes is precisely based on these definitions. Initially, the data structure contains only the string y. Then, the insertion of the string $y[i \mathinner{..} n-1]$, $1 \leqslant i < n$, proceeds in two steps:

- first, the "head" in the data structure, that is, the node h corresponding to string h_i, is located, possibly breaking an edge;
- second, a node called the "tail", say t, is created, added as successor of node h, and the edge from h to t is labeled with string t_i.

The second step of the insertion is clearly performed in constant time. Thus, finding the head is critical for the overall performance of the construction algorithm. A brute-force method to find the head consists in spelling the current suffix $y[i \mathinner{..} n-1]$ from the root of the trie, giving an $O(|h_i|)$ time complexity for the insertion at step i, and an $O(n^2)$ running time to build the suffix tree $\mathcal{T}(y)$. Adding "short-circuit" links leads to an overall $O(n)$ time complexity, although there is no guarantee that the insertion at any step i is realized in constant time.

Observe that in any suffix tree, if the string corresponding to a given internal node p in the data structure is in the form au with $a \in \Sigma$ and $u \in \Sigma^*$, then there exists an unique internal node corresponding to the string u. From this remark are defined the suffix links by

$link[p] =$ the node q corresponding to the string u
when p corresponds to the string au for some symbol $a \in \Sigma$

for each internal node p that is different from the root. The links are useful when computing h_i from h_{i-1} because of the property: if h_{i-1} is in the form aw for some symbol $a \in \Sigma$ and some string $w \in \Sigma^*$, then w is a prefix of h_i.

We explain in three following paragraphs how the suffix links help to find the successive heads efficiently. We consider a step i in the algorithm assuming that $i \geqslant 1$. We denote by g the node that corresponds to the string h_{i-1}. The aim is both to insert $y[i \mathinner{..} n-1]$ and to find the node h corresponding to the string h_i. We first study the most general case of the insertion of the suffix $y[i \mathinner{..} n-1]$. Particular cases are studied after.

We assume in the present case that the predecessor of g in the data structure, say g', is both defined and different from the root. Then h_{i-1} is in the form auv where $a \in \Sigma$, $u, v \in \Sigma^*$, au corresponds to the node g', and v labels the edge from g' to g. Since the string uv is a prefix of h_i, it can be fully spelled from the root. Moreover, the spelling operation of uv from the root can be short-circuited by spelling only the string v from the node $link[g']$. The node q reached at the end of the spelling operation (possibly breaking the last partially taken down edge) is then exactly the node $link[g]$. It remains to spell the string t_{i-1} from q for completely inserting the string $y[i \mathinner{..} n-1]$. The spelling stops on the expected node h (possibly breaking again an edge) which becomes the new head in the data structure. The suffix of t_{i-1} that has not been spelled so far, is exactly the string t_i. (An example for the whole previous statement is given in Figure 9.)

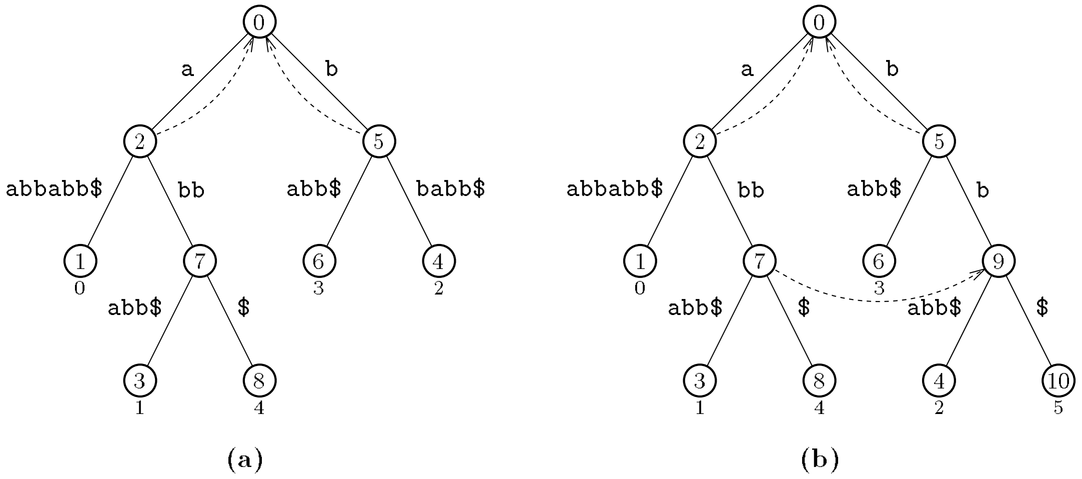


Figure 9 During the construction of the suffix tree $\mathcal{T}(y)$ of the string $y =$ `aabbabb$`, the step 5, that is, the insertion of the suffix `bb$`. The defined suffix link are depicted with discontinuous non-labeled directed edges. **(a)** Initially, the head in the data structure is node 7, and its suffix link is not yet defined. The predecessor of node 7, node 2, is different from the root, and the factor of y that is spelled from the root to node 7, namely $h_4 =$ `abb`, is in the form auv, where $a \in \Sigma$, $u \in \Sigma^*$, and v is the string of Σ^* labeling the edge from node 2 to node 7. Here, $a =$ `a`, u is the empty string, and $v =$ `bb`. Then, the string $uv =$ `bb` is spelled from the node linked with node 2, that is, from node 0; the spelling operation stops on the edge from node 5 to node 4; this edge is broken, which creates node 9. Node 9 is linked to node 7. The string $t_4 =$ `$` is spelled from node 9; the spelling operation stops on node 9, which becomes the new head in the data structure. **(b)** Node 10 is created, added as successor of node 9, and the edge from node 9 to node 10 is labeled by the string `$`, remainder of the last spelling operation.

The second case is when g is a (direct) successor of the root. The string h_{i-1} is then in the form au where $a \in \Sigma$ and $u \in \Sigma^*$. Similarly to the above case, the string u can be fully spelled from the root. The spelling of u gives a node q, which is then linked with g. Afterwards, the string t_{i-1} is spelled from q.

The last case is when g is the root itself. The string t_{i-1} minus its first symbol has to be spelled from the root. Which ends the study of all the possible cases that can arise.

The important aspect of the algorithm is the use of two different implementations for the two spelling operations pointed out above. The first one, given in the pseudocode below as the function Fast-Find, deals with the situation where we know in advance that a given factor $y[j \mathinner{.\,.} j+k-1]$ of y can be fully spelled from a given node p of the trie. It is then sufficient to scan only the first symbols of the labels of the encountered nodes, which justifies the name of the function. The second implementation of the spelling operation spells a factor $y[j \mathinner{.\,.} j+k-1]$ of y from a given node p too, but, this time, the spelling is performed symbol by symbol. The corresponding function is implemented after as the function Slow-Find. Before giving the pseudocode of the functions, we precise the notations used in the following.

- For any input (y, p, j), the function Successor-By-One-Symbol returns the node q such that q is a successor of the node p and the first symbol of the label of the edge from p to q is $y[j]$; if such a node q does not exist, it returns nil.

- For any input (p, q), the function Label returns the two integers that represents the label of the edge from the node p to the node q.

- The function New-Node creates and returns a new node.

- For any input (p, j, k, q, ℓ), the function New-Breaking-Node creates and returns the node r breaking the edge $(p, y[j\,..\,j+k-1], q)$ at the position ℓ in the label $y[j\,..\,j+k-1]$. (Which gives the two edges $(p, y[j\,..\,j+\ell-1], r)$ and $(r, y[j+\ell\,..\,j+k-1], q)$.)

Function Fast-Find returns a couple of nodes such that the second one is the node reached by the spelling, and the first one is its predecessor.

```
Fast-Find(y, p, j, k)
p' ← NIL
while k > 0
  loop p' ← p
       q ←Successor-By-One-Symbol(y, p, j)
       (r, s) ←Label(p, q)
       if s ⩽ k
         then p ← q
              j ← j + s
              k ← k − s
         else p ←New-Breaking-Node(p, r, s, q, k)
              k ← 0
return (p', p)
```

Compared to function Fast-Find, function Slow-Find considers an extra-input that is the predecessor of node p (denoted by p'). It considers in addition two extra-outputs that are the position and the length of the factor that remains to be spelled.

```
Slow-Find(y, p', p, j, k)
b ← FALSE
loop   q ←Successor-By-One-Symbol(y, p, j)
       if q = NIL
         then b ← TRUE
         else (r, s) ←Label(p, q)
              ℓ ← 1
              while ℓ < s and y[j + ℓ] = y[r + ℓ]
                loop ℓ ← ℓ + 1
              j ← j + ℓ
              k ← k − ℓ
              p' ← p
              if ℓ = s
                then p ← q
                else p ←New-Breaking-Node(p, r, s, q, ℓ)
                     b ← TRUE
  while b = FALSE
return (p', p, j, k)
```

The complete construction algorithm is implemented as the function Suffix-Tree given below. The function returns the root of the constructed suffix-tree. Memorizing systematically the predecessors h' and q' of the nodes h and q avoids considering doubly linked tries. The name of the attribute which marks the positions of the external nodes is made explicit.

```
Suffix-Tree(y, n)
p ←New-Node
h′ ← nil
h ← p
r ← −1
s ← n + 1
for i from 0 up to n − 1
   loop if h′ = nil
           then (h′, h, r, s) ←Slow-Find(y, nil, p, r + 1, s − 1)
           else  (j, k) ←Label(h′, h)
                 if h′ = p
                    then (q′, q) ←Fast-Find(y, p, j + 1, k − 1)
                    else  (q′, q) ←Fast-Find(y, link[h′], j, k)
                 link[h] ← q
                 (h′, h, r, s) ←Slow-Find(y, q′, q, r, s)
        t ←New-Node
        Make-Edge(h, (r, s), t)
        position[t] ← i
return p
```

The algorithm runs in time $O(n)$ (more precisely $O(n \times \log \operatorname{card} \Sigma)$ if we take into account the branching in the data structure). Indeed, the instruction at line 4 in function Fast-Find is performed less than $2n$ times, and the number of symbol comparisons done at line 7 in function Slow-Find is less than n.

Once the suffix tree of y is build, some operations can be performed rapidly. We describe four applications in the following. Let x be a string of length m.

Testing whether x occurs in y or not can be solved in time $O(m)$ by spelling x from the root of the trie symbol by symbol. If the operation succeeds, x occurs in y. Otherwise, we get the longest prefix of x occurring in y.

Producing the number of occurrences of x in y starts identically by spelling x. Assume that x occurs actually in y. Let p be the node at the extremity of the last taken down edge, or be the root itself if x is empty. The expected number, say k, is then exactly the number of external nodes of the sub-trie of root p. This number can be computed by traversing the sub-trie. Since each internal node of the sub-trie has at least two successors, the total size of the sub-trie is $O(k)$, and the traversal of the sub-trie is performed in time $O(k)$ (independently of Σ). The method can be improved by precomputing in time $O(n)$ (independently of Σ) all the values associated with each internal node; the whole operation is then performed in time $O(m)$, whatever is the number of occurrences of x.

The method for reporting the list of positions of x in y proceeds in the same way. The running time needed by the operation is $O(m)$ to locate x in the trie, plus $O(k)$ to report each of the positions associated with the k external nodes.

Finding the longest repeated factor of y remains to compute the "deepest" internal node of the trie, that is, the internal node corresponding to a longest possible factor in y. This is performed in time $O(n)$.

3.2 Suffix Automata

The suffix automaton $\mathcal{S}(y)$ of a string y is the minimal deterministic automaton recognizing $\mathrm{Suff}(y)$, that is, the set of suffixes of y. This automaton is minimal among all the deterministic

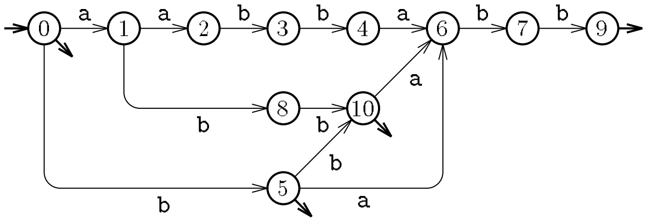


Figure 10 The suffix automaton $\mathcal{S}(y)$ of the string y = `aabbabb`. The states are numbered from 0 to 10, according to the order in which they are created by the construction algorithm described in the present section. The initial state is state 0, terminal states are states 0, 5, 9, and 10. This automaton is the minimal deterministic automaton accepting the language of the suffixes of y.

automata recognizing the same language, which implies that it is not necessarily complete. An example is given in Figure 10.

The main point about suffix automata is that their size is asymptotically linear in the length of the string. More precisely, given a string y of length n, the number of states of $\mathcal{S}(y)$ is equal to $n+1$ when $n \leqslant 2$, and is bounded by $n+1$ and $2n-1$ otherwise; as to the number of edges, it is equal to $n+1$ when $n \leqslant 1$, it is 2 or 3 when $n = 2$, and it bounded by n and $3n-4$ otherwise.

The construction of the suffix automaton of a string y of length n can be performed in time $O(n)$, or, more precisely, in time $O(n \times \log \operatorname{card} \Sigma)$ on an arbitrary alphabet Σ. It makes use of a failure function *fail* defines on the states of $\mathcal{S}(y)$. The set of states of $\mathcal{S}(y)$ identifies with the quotient sets

$$u^{-1}\operatorname{Suff}(y) = \{v \in \Sigma^* \mid uv \in \operatorname{Suff}(y)\}$$

for the strings u in the whole set of factors of y. One may observe that two sets in the form $u^{-1}\operatorname{Suff}(y)$ are either disjoint or comparable. This allows to set

$\mathit{fail}[p]$ = the smallest quotient set stricly containing the quotient set identified with p,

for each state p of the automaton different from the initial state of the automaton. The function given below as the function Suffix-Automaton builds the suffix automaton of y, and returns the initial state, say i, of the automaton. The construction is on-line, which means that at each step of the construction, just after processing a prefix y' of y, the suffix automaton $\mathcal{S}(y')$ is build. Denoting by t the state without outgoing edge in the automaton $\mathcal{S}(y')$, terminal states of $\mathcal{S}(y')$ are implicitly known by the "suffix path" of t, that is, the list of the states

$$t, \mathit{fail}[t], \mathit{fail}[\mathit{fail}[t]], \ldots, i.$$

The algorithm uses the function *length* defined for each state p of $\mathcal{S}(y)$ by

$\mathit{length}[p]$ = the length of the longest string spelled from i to p.

```
SUFFIX-AUTOMATON(y)
i ←NEW-STATE
terminal[i] ← FALSE
length[i] ← 0
fail[i] ← NIL
t ← i
for symbol a from first to last symbol of y
   loop t ←SUFFIX-AUTOMATON-EXTENSION(i, t, a)
p ← t
loop    terminal[p] ← TRUE
        p ← fail[p]
   while p ≠ NIL
return i
```

The on-line construction is based on the function SUFFIX-AUTOMATON-EXTENSION that is implemented below. The latter function processes the next symbol, say a, of the string y. If y' is the prefix of y preceding a, it transforms the suffix automaton $\mathcal{S}(y')$ already build into the suffix automaton $\mathcal{S}(y'a)$.

```
SUFFIX-AUTOMATON-EXTENSION(i, t, a)
t' ← t
t ←NEW-STATE
terminal[t] ← FALSE
length[t] ← length[t'] + 1
p ← t'
loop    MAKE-EDGE(p, a, t)
        p ← fail[p]
   while p ≠ NIL and TARGET(p, a) = NIL
if p = NIL
   then fail[t] ← i
   else q ←TARGET(p, a)
        if length[q] = length[p] + 1
           then fail[t] ← q
           else r ←NEW-STATE
                terminal[r] ← FALSE
                for each letter b such that TARGET(q, b) ≠ NIL
                   loop MAKE-EDGE(r, b,TARGET(q, b))
                length[r] ← length[p] + 1
                fail[r] ← fail[q]
                fail[q] ← r
                fail[t] ← r
                loop    CANCEL-EDGE(p, a,TARGET(p, a))
                        MAKE-EDGE(p, a, r)
                        p ← fail[p]
                   while p ≠ NIL and TARGET(p, a) = q
return t
```

We illustrate the behavior of function SUFFIX-AUTOMATON-EXTENSION in Figure 11.

With the suffix automaton $\mathcal{S}(y)$ of y, several operations can be solved efficiently. We describe three of them. Let x be a string of length m.

(a)

(b)

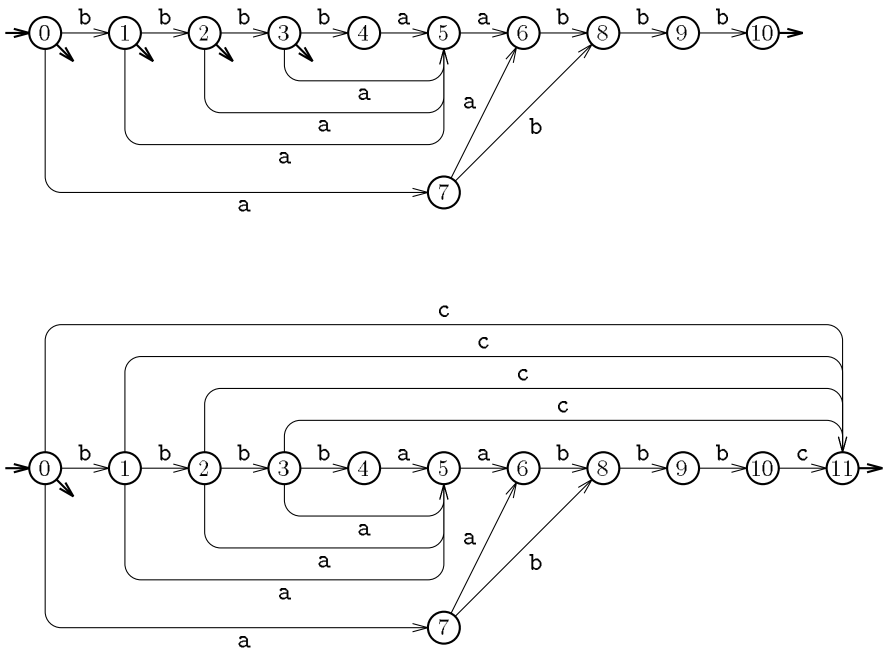
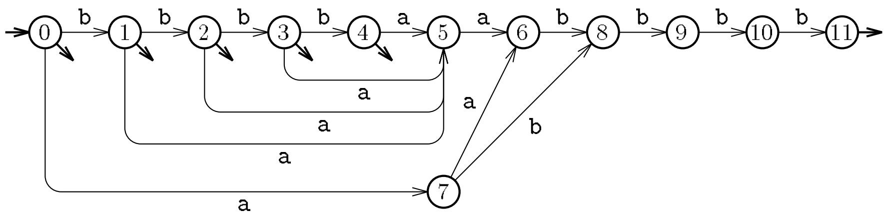
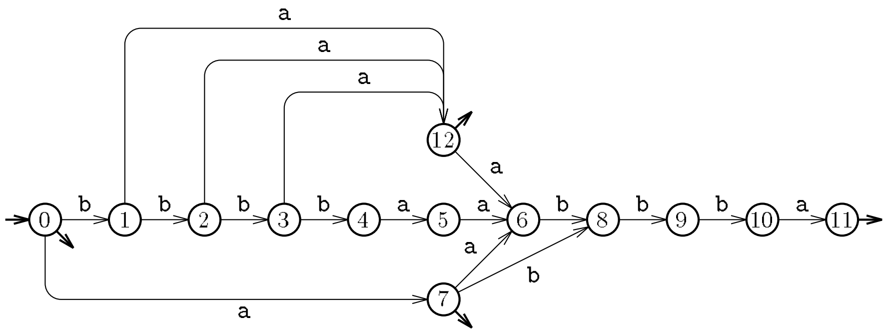


Figure 11 An illustration of the behavior of function SUFFIX-AUTOMATON-EXTENSION. The function transforms the suffix automaton $\mathcal{S}(y')$ of a string y' in the suffix automaton $\mathcal{S}(y'a)$ for any given symbol a (the terminal states being implicitly known). Let us consider that $y' = \texttt{bbbbaabbb}$, and let us examine three possible cases according to a, namely $a = \texttt{c}$, $a = \texttt{b}$, and $a = \texttt{a}$. **(a)** The automaton $\mathcal{S}(\texttt{bbbbaabbb})$. The state denoted by t' is state 10, and the suffix path of t' is the list of the states 10, 3, 2, 1, and 0. During the execution of the first loop of the function, state p runs through a part of the suffix path of t'. At the same time, edges labeled by a are created from p the newly created state $t = 11$, unless such an edge already exists in which case the loop stops. **(b)** If $a = \texttt{c}$, the execution stops with an undefined value for p. The edges labeled by `c` start at terminal states, and the failure of t is the initial state. **(c)** If $a = \texttt{b}$, the loop stops on state $p = 3$, because an edge labeled by `b` is defined on it. The condition at line 12 of function SUFFIX-AUTOMATON-EXTENSION is satisfied, which means that the edge labeled by a from p is not a short-circuit. In this case, the state ending the previous edge is the failure of t. **(d)** Finally, when $a = \texttt{a}$, the loop stops on state $p = 3$ for the same reason, but the edge labeled by a from p is a short-circuit. The string `bbba` is a suffix of the (newly considered) string `bbbbaabbba`, but `bbbba` is not. Since these two strings reach state 5, this state is duplicated into a new state $r = 12$ that becomes terminal. Suffixes `bba` and `ba` are re-directed to this new state. The failure of t is r.

(c)



(d)



Membership test solves in time $O(m)$ by spelling x from the initial state of the automaton. If the entire string is spelled, x occurs in y. Otherwise we get the longest prefix of x occurring in y.

Computing the number k of occurrences of x in y (assuming that x is a factor of y) starts similarly. Let p be the state reached after the spelling of x from the initial state. Then k is exactly the number of terminal states accessible from p. The number k associated with each state p can be precomputing in time $O(n)$ (independently of the alphabet) by a depth-first traversal of the graph underlying the automaton. The query for x is then performed in time $O(m)$, whatever is k.

The base of an algorithm for computing a longest factor common to x and y is implemented in the procedure Ending-Factors-Matcher given below. This procedure reports at each position in y the length of the longest factor of x ending here. It can obviously be used for string matching. It works as the procedure Aho-Corasick-Matcher in the use of the failure function. The running time of the search phase of the procedure is $O(m)$.

```
ENDING-FACTORS-MATCHER(y, x)
i ←SUFFIX-AUTOMATON(y)
ℓ ← 0
p ← i
for symbol a from first to last symbol of x
   loop if TARGET(p, a) ≠ NIL
            then ℓ ← ℓ + 1
                 p ←TARGET(p, a)
            else loop   p ← fail[p]
                   while p ≠ NIL and TARGET(p, a) ≠ NIL
                 if p = NIL
                   then ℓ ← 0
                        p ← i
                   else ℓ ← length[p] + 1
                        p ←TARGET(p, a)
         report ℓ
```

Retaining a largest value of the variable ℓ in the procedure (instead of reporting all values) solves the longest common factor problem.

4 Research Issues and Summary

String searching by hashing was introduced by Harrison (1971), and later fully analyzed in [Karp and Rabin, 1987].

The first linear-time string-matching algorithm is due to Knuth, Morris, and Pratt ([Knuth, Morris, and Pratt, 1977]). It can be proved that, during the search, the delay, that is, the number of times a symbol of the text is compared to symbols of the pattern, is less than $\lfloor \log_\Phi(m+1) \rfloor$, where Φ is the golden ratio $(1+\sqrt{5})/2$. [Simon, 1993] gives a similar algorithm but with a delay bounded by the size of the alphabet (of the pattern). [Hancart, 1993] proves that the delay of Simon's algorithm is less than $1+\lfloor \log_2 m \rfloor$. This paper also proves that this is optimal among algorithms processing the text with a one-symbol buffer. The bound becomes $O(\log \min\{1+\lfloor \log_2 m \rfloor, \mathrm{card}\,\Sigma\})$ using an ordering on the alphabet Σ, which is not a restriction in practice.

[Galil, 1981] gives a general criterion to transform string-matching algorithms that work sequentially on the text into real-time algorithms.

The Boyer-Moore algorithm was designed in [Boyer and Moore, 1977]. The version given in this chapter follows [Knuth, Morris, and Pratt, 1977]. This paper contains the first proof on the linearity of the algorithm when restricted to the search of the first occurrence of the pattern. [Cole, 1994] proves that the maximum number of symbol comparisons is bounded by $3n$ for non periodic patterns, and that this bound is tight.

[Knuth, Morris, and Pratt, 1977] considers a variant of the Boyer-Moore algorithm in which all previous matches inside the current window are memorized. Each window configuration becomes the state of what is called the Boyer-Moore automaton. It is still unknown whether the maximum number of states of the automaton is polynomial or not.

Several variants of the Boyer-Moore algorithm avoid the quadratic behavior when searching for all occurrences of the pattern. Among the most efficient in terms of the number of symbol comparisons are the algorithm of Apostolico and Giancarlo (1986), Turbo-BM algorithm by

Crochemore *et alii* (1992) (the two previous algorithms are analyzed in [Lecroq, 1995]), and the algorithm of Colussi ([Colussi, 1994]).

The Horspool algorithm is from [Horspool, 1980]. The paper contains practical aspects of string matching that are developed in [Hume and Sunday, 1993].

The optimal bound on the expected time complexity of string matching is $O(\frac{\log m}{m} n)$ (see [Knuth, Morris, and Pratt, 1977] and the paper of Yao (1980)).

String matching can be solved by linear-time algorithms requiring only a constant amount of memory in addition to the pattern and the (window on the) text. This can be proved by different techniques presented in [Crochemore and Rytter, 1994]. The most recent solution is by Gąsieniec, Plandowski, and Rytter (1995).

[Cole *et alii*, 1995] shows that, in the worst case, any string-matching algorithm working with symbol comparisons makes at least $n + \frac{9}{4m}(n - m)$ comparisons during its search phase. Some string-matching algorithms make less than $2n$ comparisons. The presently-known upper bound on the problem is $n + \frac{8}{3(m+1)}(n - m)$, but with a quadratic-time preprocessing phase (see [Cole *et alii*, 1995]). With a linear-time preprocessing phase, the current upper bounds are $\frac{4}{3}n - \frac{1}{3}m$ and $n + \frac{4 \log m + 2}{m}(n - m)$ (see respectively [Galil and Giancarlo, 1992] and [Breslauer and Galil, 1993]). Except in a few cases (patterns of length 3 for example), lower and upper bounds do not meet. So, the problem of the exact complexity of string matching is open.

The Aho-Corasick algorithm is from [Aho and Corasick, 1975]. Commentz-Walter (1979) has designed an extension of the Boyer-Moore algorithm that solves the dictionary-matching problem. It is fully described in [Aho, 1990].

The suffix-tree construction of Section 3 is from [McCreight, 1976]. An on-line version is by Ukkonen (1992). A previous algorithm by Weiner (1973) relates suffix trees to a data structure close to suffix automata.

The construction of suffix automata, also described as direct acyclic word graphs and often denoted by the acronym DAWG, is from [Blumer *et alii*, 1985] and from [Crochemore, 1986].

An alternative data structure that implements efficiently indexes is the notion of suffix arrays introduced in [Manber and Myers, 1993].

5 Defining Terms

Border: A string v is a border of a string u if v is both a prefix and a suffix of u. String v is said to be the border of u if it is the longest proper border of u.

Factor: A string v is a factor of a string u if $u = u'vu''$ for some strings u' and u''.

Occurrence: A string v occurs in a string u if v is a factor of u.

Pattern: A finite number of strings that are searched for in texts.

Prefix: A string v is a prefix of a string u if $u = vu''$ for some string u''.

Proper: Qualifies a factor of a string that is not equal to the string itself.

Segment: Equivalent to factor.

Suffix: A string v is a suffix of a string u if $u = u'v$ for some string u'.

Suffix tree: Trie containing all the suffixes of a string.

Suffix automaton: Smallest automaton accepting the suffixes of a string.

Text: A stream of symbols that is searched for occurrences of patterns.

Trie: Digital tree, tree in which edges are labeled by symbols or strings.

Window: Factor of the text that is aligned with the pattern.

6 References

Aho, A.V. 1990. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science*, ed. J. van Leeuwen, vol. A, chap. 5, p. 255–300. Elsevier, Amsterdam.

Aho, A.V. and Corasick, M.J. 1975. Efficient string matching: an aid to bibliographic search. *Comm. ACM.* 18:333–340.

Baase, S. 1988. *Computer algorithms – Introduction to design and analysis.* Addison-Wesley.

Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., Chen, M.T., and Seiferas, J. 1985. The smallest automaton recognizing the subwords of a text. *Theoret. Comput. Sci.* 40:31–55.

Boyer, R.S. and Moore, J.S. 1977. A fast string searching algorithm. *Comm. ACM.* 20:762–772.

Breslauer, D. and Galil, Z. 1993. Efficient comparison based string matching. *J. Complexity.* 9:339–365.

Cole, R. 1994. Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. *SIAM J. Comput.* 23:1075–1091.

Cole, R., Hariharan, R., Zwick, U., and Paterson, M.S. 1995. Tighter lower bounds on the exact complexity of string matching. *SIAM J. Comput.* 24:30–45.

Colussi, L. 1994. Fastest pattern matching in strings. *J. Algorithms.* 16:163–189.

Cormen, T.H., Leiserson, C.E., and Rivest, R.L. 1990. *Introduction to algorithms.* MIT Press.

Crochemore, M. 1986. Transducers and repetitions. *Theoret. Comput. Sci.* 45:63–86.

Crochemore, M. and Rytter, W. 1994. *Text Algorithms.* Oxford University Press.

Galil, Z. 1981. String matching in real time. *J. ACM.* 28:134–149.

Galil, Z. and Giancarlo, R. 1992. On the exact complexity of string matching: upper bounds. *SIAM J. Comput.* 21:407–437.

Gonnet, G.H. and Baeza-Yates, R.A. 1991. *Handbook of algorithms and data structures.* Addison-Wesley.

Hancart, C. 1993. On Simon's string searching algorithm. *Inf. Process. Lett.* 47:95–99.

Horspool, R.N. 1980. Practical fast searching in strings. *Software – Practice and Experience.* 10:501–506.

Hume, A. and Sunday, D.M. 1991. Fast string searching. *Software – Practice and Experience.* 21:1221–1248.

Karp, R.M. and Rabin, M.O. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* 31:249–260.

Knuth, D.E., Morris Jr, J.H., and Pratt, V.R. 1977. Fast pattern matching in strings. *SIAM J. Comput.* 6:323–350.

Lecroq, T. 1995. Experimental results on string-matching algorithms. *Software – Practice and Experience.* 25:727–765.

McCreight, E.M. 1976. A space-economical suffix tree construction algorithm. *J. Algorithms.* 23:262–272.

Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.* 22:935–948.

Sedgewick, R. 1988. *Algorithms.* Addison-Wesley.

Simon, I. 1993. String matching algorithms and automata. In *First American Workshop on String Processing*, ed. R. Baeza-Yates and N. Ziviani, p. 151–157. Universidade Federal de Minas Gerais.

Stephen, G.A. 1994. *String searching algorithms.* World Scientific Press.

7 Further Information

Problems and algorithms presented in the chapter are just a sample of questions related to pattern matching. They share the formal methods used to design efficient algorithms. A wider panorama of algorithms on texts may be found in a few books such as [Crochemore and Rytter, 1994] and [Stephen, 1994].

Research papers in pattern matching are disseminated in a few journals, among which are: *Communications of the ACM*, *Journal of the ACM*, *Theoretical Computer Science*, *Journal of Algorithms*, *SIAM Journal on Computing*, *Algorithmica.*

Two main annual conferences present the latest advances of this field of research:

- *Combinatorial Pattern Matching*, which started in 1990 in Paris (France), and was held since in London (England), Tucson (Arizona), Padova (Italy), Asilomar (California), Helsinki (Finland), Laguna Beach (California).
- *Workshop on String Processing*, which started in 1993 in Belo Horizonte (Brazil), and was held since in Valparaiso (Chile), and Recife (Brazil).

But general conferences in computer science often have sessions devoted to pattern matching.

Several books on the design and analysis of general algorithms contain a chapter devoted to algorithms on texts. Here is a sample of these books: [Baase, 1988], [Cormen, Leiserson, and Rivest, 1990], [Gonnet and Baeza-Yates, 1991], [Sedgewick, 1988].