1

Text Searching and Indexing

Maxime Crochemore and Thierry Lecroq

Université de Marne-la-Vallée and Université de Rouen

In Z. Ésik, C. Martín-Vide, and V. Mitrana, editors, *Recent Advances in Formal Languages and Applications*, chapter 2, pages 43–80. Springer-Verlag, 2006.

Although data is stored in various ways, text remains the main form of exchanging information. This is particularly evident in literature or linguistics where data is composed of huge corpora and dictionaries. This applies as well to computer science, where a large amount of data is 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 this fields tends to double every 18 months. This is the reason why algorithms should be efficient even if the speed of computers increases at a steady pace.

Pattern matching is the problem of locating a specific pattern inside raw data. The pattern is usually a collection of strings described in some formal language. In this chapter we present several algorithms for solving the problem when the pattern is composed of a single string.

In several applications, texts need to be structured before being searched. Even if no further information is known about their syntactic structure, it is possible and indeed extremely efficient to build a data structure that support searches. In this chapter we present suffix arrays, suffix trees, suffix automata and compact suffix automata.

1.1 Pattern matching

String-matching consists in finding all the occurrences of a pattern x of length m in a text y of length n $(m, n > 0)$. Both strings x and y are built on a finite alphabet V.

Applications require two kinds of solution depending on which string, the pattern or the text, is given first. Algorithms based on the use of automata or

combinatorial properties of strings are commonly implemented to preprocess the pattern in time $O(m)$ and solve the first kind of problem in time $O(n)$. The notion of indexes realized by trees or automata is used in the second kind of solutions to preprocess the text in time $O(n)$. The search of the pattern can then be done in time $O(m)$. This section only investigates algorithms of the first kind.

String-matching is a very important subject in the wider domain of text processing. String-matching algorithms are basic components used in implementations of practical software existing under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (system or software design). Finally, they also play an important role in theoretical computer science by providing challenging problems.

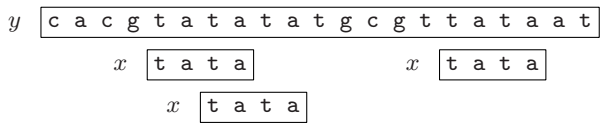


Fig. 1.1. There are three occurrences of $x = \texttt{tata}$ in $y = \texttt{cacgtatatatgcgttataat}$.

Figure 1.1 shows the occurrences of the pattern $x = \texttt{tata}$ in the text $y = \texttt{cacgtatatatgcgttataat}$. The basic operations allowed for comparing symbols are equality and inequality ($=$ and $\neq$).

String-matching algorithms of the present section work as follows. They scan the text through a window which size is generally equal to m. They first align the left ends of the window and the text, then compare the symbols of the window with the symbols of the pattern — this specific work is called an *attempt* — and after a whole match of the pattern or after a mismatch they *shift* the window to the right. They repeat the same procedure again until the right end of the window goes beyond the right end of the text. This mechanism is usually called the *sliding window mechanism*. A string-matching algorithm is thus a succession of scan and shift. Figure 1.2 illustrates this notion.

We associate each attempt with the positions j and $j + m - 1$ in the text when the window is positioned on $y[j \,..\, j + m - 1]$: we say that the attempt is at the left position j and at the right position $j + m - 1$.

The naive algorithm consists in checking, at all positions in the text between 0 and $n - m$, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right. It memorizes no information (see Figure 1.3). It requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase the symbol comparisons can be done in any order. The time complexity of this searching phase is $O(m \times n)$ (the bound is met when searching for $\texttt{a}^{m-1}\texttt{b}$ in $\texttt{a}^n$ for instance). The expected number

```
STRING-MATCHING(x, m, y, n)
 1  put window at the beginning of y
 2  while window on y do
        ▷ scan
 3      if window = x then
 4          report it
        ▷ shift
 5      shift window to the right and
 6      memorize some information for use during next scans and shifts
```

Fig. 1.2. Scan and shift mechanism for string-matching.

of symbol comparisons is $2n$ on a two-symbol alphabet, with equiprobability and independence conditions.

```
NAIVE-SEARCH(x, m, y, n)
 1  j ← 0
 2  while j ≤ n − m do
 3      i ← 0
 4      while i < m and x[i] = y[i + j] do
 5          i ← i + 1
 6      if i = m then
 7          OUTPUT(x occurs in y at position j)
 8      j ← j + 1
```

Fig. 1.3. The naive string-matching algorithm.

1.1.1 Complexities of the problem

The following theorems state some known results on the problem.

Theorem 1 ([GS83]). *The search can be done optimally in time $O(n)$ and space $O(1)$.*

Theorem 2 ([Yao79]). *The search can be done in optimal expected time $O(\frac{\log m}{m} \times n)$.*

Theorem 3 ([CHPZ95]). *The maximal number of comparisons done during the search is $\geq n + \frac{9}{4m}(n-m)$, and can be made $\leq n + \frac{8}{3(m+1)}(n-m)$.*

We now give lower and upper bounds on symbol comparisons with different strategies depending on the access to the text:
Access to the whole text:

- upper: $2n - 1$ [MP70];
- lower: $\frac{4}{3}n$ [GG91].

Search with a sliding window of size m:

- lower: $n + \frac{9}{4m}(n - m)$ [CHPZ95];
- upper: $n + \frac{8}{3(m+1)}(n - m)$ [CHPZ95].

Search with a sliding window of size 1:

- lower and upper: $(2 - \frac{1}{m})n$ [Han93, BCT93];

The delay is defined as the maximum number of comparisons on each text symbol:

- lower: $\min\{1 + \log_2 m, card(V)\}$ [Han93];
- upper: $\min\{\log_\Phi(m + 1), card(V)\}$ [Sim89],
  $\min\{1 + \log_2 m, card(V)\}$ [Han93] and
  $\log \min\{1 + \log_2 m, card(V)\}$ [Han96].

1.1.2 Methods

Actually, searching for the occurrences of a pattern x in a text y consists in identifying all the prefixes of y that are elements of the language V^*x (see Figure 1.4).

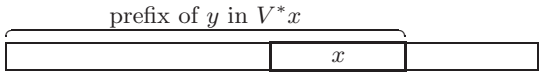


Fig. 1.4. An occurrence of the pattern x in the text y corresponds to a prefix of y in V^*x.

To solve this problem there exist several methods of different types :

- **Sequential searches**: methods in this category adopt a window of size exactly one symbol. They are well adapted to applications in telecommunication. They are based on efficient implementations of automata [KMP77, Sim89, Han93, BCT93].
- **Time-space optimal searches**: these methods are mainly of theoretical interest and are based on combinatorial properties of strings [GS83, CP91, Cro92, GPR95, CGR99].
- **Practically-fast searches**: these methods are typically used in text editors or data retrieval software. They are based on combinatorics on words and theory of automata and often use heuristics [BM77, Gal79, AG86, CCG+94].

1.1.3 Morris and Pratt algorithm

Periods and borders

For a non-empty string u, an integer p such that $0 < p \leq |u|$ is a **period** of u if any of these equivalent conditions is satisfied:

1. $u[i] = u[i+p]$, for $0 \leq i < |u| - p$;
2. u is a prefix of some y^k, $k > 0$, $|y| = p$;
3. $u = yw = wz$, for some strings y, z, w with $|y| = |z| = p$. The string w is called a **border** of u: it occurs both as a prefix and a suffix of u.

The period of u, denoted by $period(u)$, is its smallest period (it can be $|u|$). **The border** of u, denoted by $border(u)$, is its longest border (it can be empty).

Example 1. $u =$ `abacabacaba`

periods	borders of u
4	`abacaba`
8	`aba`
10	`a`
11	empty string

The searching algorithm

The notions of period and border naturally lead to a simple on-line search algorithm where the length of the shift is given by the period of the matched prefix of the pattern. Furthermore the algorithm implements a memorization of the border of the matched prefix of the pattern.

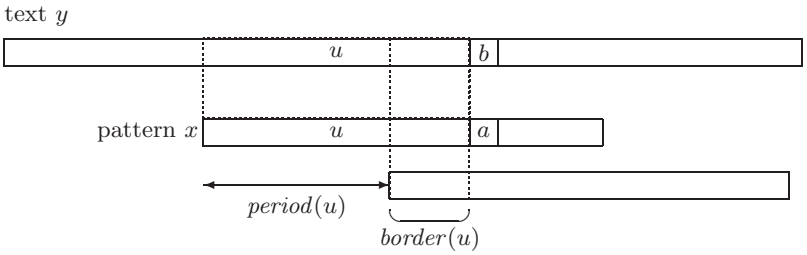


Fig. 1.5. A typical situation during a sequential search.

A typical situation during a sequential search is the following: a prefix u of the pattern has been matched, a mismatch occurs between a symbol a in the pattern and a symbol b in the text ($a \neq b$). Then a shift of length $period(u) = |u| - |border(u)|$ can be applied (see Figure 1.5). The comparisons

are then resumed between symbols $x[|border(u)|]$ of x and b in y (no backtrack is necessary on the text y). The corresponding algorithm MP, due to Morris and Pratt [MP70], is shown Figure 1.6. It uses a table *MPnext* defined by $MPnext[i] = |border(x[0\,..\,i-1])|$ for $i = 0, \ldots, m$.

```
MP(x, m, y, n)
 1  i ← 0
 2  j ← 0
 3  while j < n do
 4     while i = m or (i ≥ 0 and x[i] ≠ y[j]) do
 5        i ← MPnext[i]
 6     i ← i + 1
 7     j ← j + 1
 8     if i = m then
 9        OUTPUT(x occurs in y at position j − i)
```

Fig. 1.6. The Morris and Pratt string-matching algorithm.

Computing borders of prefixes

The table *MPnext* is defined by $MPnext[i] = |border(x[0\,..\,i-1])|$ for $i = 0, \ldots, m$. It can be computed by using the following remark: a border of a border of u is a border of u. A border of u is either $border(u)$ or a border of it. It can be linearly computed by the algorithm presented in Figure 1.7. This algorithm uses an index j that runs through decreasing lengths of borders. The computation of the table *MPnext* proceeds as the searching algorithm, as if $y = x[1\,..\,m-1]$.

```
COMPUTE-MP-NEXT(x, m)
 1  MPnext[0] ← −1
 2  for i ← 0 to m − 1 do
 3     j ← MPnext[i]
 4     while j ≥ 0 and x[i] ≠ x[j] do
 5        j ← MPnext[j]
 6     MPnext[i + 1] ← j + 1
 7  return MPnext
```

Fig. 1.7. A linear time algorithm for computing the table *MPnext* for a string x of length m.

1.1.4 Knuth-Morris-Pratt algorithm

Interrupted periods and strict borders

For a fixed string x and a non-empty prefix u of x, w is a strict border of u if both:

- w is a border of u;
- wb is a prefix of x, but ub is not.

An integer p is an interrupted period of u if $p = |u| - |w|$ for some strict border $|w|$ of u.

Example 2. Prefix `abacabacaba` of `abacabacabacc`

interrupted periods	strict borders of `abacabacaba`
10	`a`
11	empty string

The searching algorithm

The Morris-Pratt algorithm can be further improved. Consider a typical situation (Figure 1.5) where a prefix u of x has been matched and a mismatch occurs between the symbol a in x and the symbol b in y. Then the shift in the Morris-Pratt algorithm consists in aligning the prefix occurrence of the border of u in x with its suffix occurrence in y. But if this prefix occurrence in x is followed by the symbol a then another mismatched will occur with the symbol b in y. An alternative solution consists in precomputing for each prefix $x[0..i-1]$ of x the longest border followed by a symbol different from $x[i]$ for $i = 1, \ldots, m$. Those borders are called **strict borders** and then the length of the shifts are given by **interrupted periods**. It changes only the preprocessing of the string-matching algorithm KMP which is due to Knuth, Morris and Pratt [KMP77].

Computing strict borders of prefixes

The preprocessing of the algorithm KMP consists in computing the table *KMPnext*. *KMPnext*[0] is set to -1. Then for $i = 1, \ldots, m-1$, $k = MPnext[i]$

$$KMPnext[i] = \begin{cases} k & \text{if } x[i] \neq x[k] \text{ or if } i = m, \\ KMPnext[k] & \text{if } x[i] = x[k]. \end{cases}$$

The table *KMPnext* can be computed with the algorithm presented Figure 1.8.

```
COMPUTE-KMP-NEXT(x, m)
 1  KMPnext[0] ← −1
 2  j ← 0
 3  for i ← 1 to m − 1 do
 4     if x[i] = x[j] then
 5        KMPnext[i] ← KMPnext[j]
 6     else KMPnext[i] ← j
 7        j ← KMPnext[j]
 8        while j ≥ 0 and x[i] ≠ x[j] do
 9           j ← KMPnext[j]
10     j ← j + 1
11  KMPnext[m] ← j
12  return KMPnext
```

Fig. 1.8. Preprocessing of the Knuth-Morris-Pratt algorithm.

1.1.5 Complexities of MP and KMP algorithms

Let us consider the algorithm given in Figure 1.6. Every positive comparisons increase the value of j. The value of j runs from 0 to $n-1$. Thus there are at most n positive comparisons. Negative comparisons increase the value of $j-i$ because such comparisons imply a shift. The value of $j-i$ runs from 0 to $n-1$. Thus there are at most n negative comparisons. Altogether, the algorithm makes no more than $2n$ symbol comparisons. This gives the following theorem.

Theorem 4. *On a text of length n, MP and KMP string-searching algorithms run in time $O(n)$. They make less than $2n$ symbol comparisons.*

The **delay** is defined as the maximum number of comparisons on one text symbol.

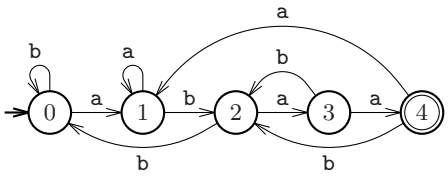
Theorem 5. *With a pattern of length m, the delay for* MP *algorithm is no more than m. The delay for* KMP *algorithm is no more than $\log_{\Phi}(m+1)$, where Φ is the golden ratio, $(1+\sqrt{5})/2$.*

Theorem 5 shows the advantage of KMP algorithm over MP algorithm. Its proof relies on combinatorial properties of strings. The next section sketches a further improvement.

1.1.6 Searching with an automaton

The MP and KMP algorithms simulate a finite automaton. It is possible to build and use the string-matching automaton $SMA(x)$ which is the smallest deterministic automaton accepting the language V^*x.

Example 3. SMA(`abaa`)



Search for abaa in $y =$ babbaabaabaabba:

y	b	a	b	b	a	a	b	a	a	b	a	a	b	b	a
state	0	0	1	2	0	1	1	2	3	4	2	3	4	2	0	1

Two occurrences of x occur in y at (right) positions 8 and 11. This is given by the fact that at these positions the search reaches the only terminal state of the string-matching automaton (state 4).

Searching algorithm

The searching algorithm consists in a simple parsing of the text y with the string-matching automaton $SMA(x)$ (see Figure 1.9).

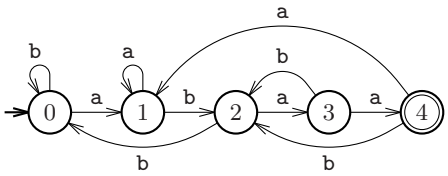
```
SEARCH-WITH-AN-AUTOMATON(x, y)
 1  (Q, V, initial, {terminal}, δ) is the automaton SMA(x)
 2  q ← initial
 3  while not end of y do
 4     a ← next symbol of y
 5     q ← δ(q, a)
 6     if q = terminal then
 7        report an occurrence of x in y
```

Fig. 1.9. Searching with an automaton.

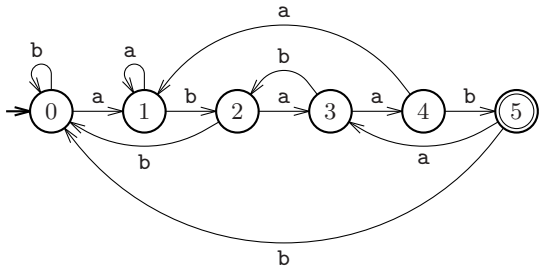
Construction of $SMA(x)$

The on-line construction of the smallest deterministic automaton accepting the language V^*x actually consists in unwinding appropriate arcs. The following example presents one step of the construction.

Example 4. From SMA(abaa)



to $SMA(\texttt{abaab})$



Updating $SMA(\texttt{abaa})$ to $SMA(\texttt{abaab})$ consists in storing the target state of the transition from the terminal state 4 in $SMA(\texttt{abaa})$ labeled by `b` (the new symbol): this state is 2. Then a new terminal state 5 is added and the transition from 4 by `b` is redirected to 5 and transitions for all the symbols of the alphabet from 5 go as all the transitions from 2: 5 by `a` leads to 3 since 2 by `a` leads to 3 (same for 5 by `b` leading to 0 since 2 by `b` leads to 0).

The complete construction can be achieved by the algorithm given in Figure 1.10.

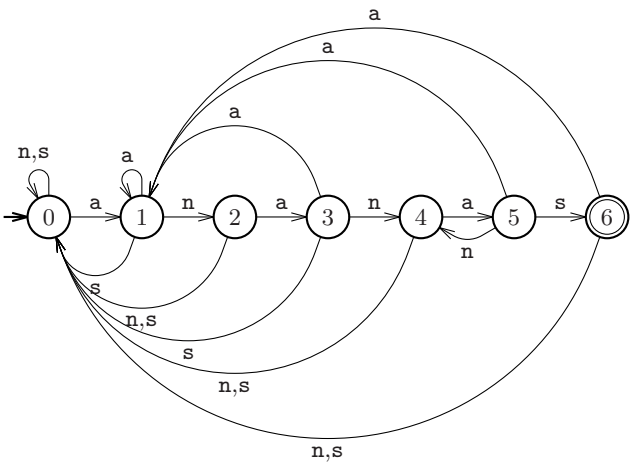
```
AUTOMATON-SMA(x)
 1  let initial be a new state
 2  Q ← {initial}
 3  for each a ∈ V do
 4     δ(initial, a) ← initial
 5  terminal ← initial
 6  while not end of x do
 7     b ← next symbol of x
 8     r ← δ(terminal, b)
 9     add new state s to Q
10     δ(terminal, b) ← s
11     for each a ∈ V do
12        δ(s, a) ← δ(r, a)
13     terminal ← s
14  return (Q, V, initial, {terminal}, δ)
```

Fig. 1.10. The construction of the automaton $SMA(x)$.

Significant arcs

We now characterize the number of significant arcs in the string-matching automaton $SMA(x)$.

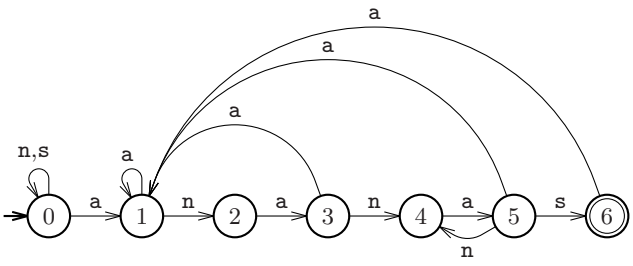
Example 5. Complete automaton *SMA*(ananas):



In such an automaton we distinguish two kinds of arcs:

- **Forward arcs**: arcs that spell the pattern;
- **Backward arcs**: other arcs which do not reach the initial state.

Example 6. SMA(ananas) represented with only forward and backward arcs:



Backward arcs in *SMA*(x)

The different states of *SMA*(x) are identified with prefixes of x. A backward arc is of the form (u, b, vb) with u, v prefixes of x and $b \in V$ a symbol where vb is the longest suffix of ub that is a prefix of x, and $u \neq v$. Note that ub is not a prefix of x. Let $p(u, b) = |u| - |v|$ (a period of u).

Let (u, b, vb) and $(u', b', v'b')$ be two backward arcs. If $p(u, b) = p(u', b') = p$, then $vb = v'b'$. Otherwise, if, for instance, vb is a proper prefix of $v'b'$, vb occurs at position p like v' does, ub is a prefix of x, which is a contradiction. Thus $v = v'$, $b = b'$, and then $u = u'$. Each period p, $1 \leq p \leq |x|$, corresponds to at most one backward arc, thus there are at most $|x|$ such arcs. This gives the following lemma.

Lemma 1. *The automaton SMA(x) contains at most $|x|$ backward arcs.*

The bound of the previous lemma is tight: $SMA(ab^{m-1})$ has m backward arcs ($a \neq b$) and thus constitutes a worst case for the number of backward arcs.

A fairly immediate consequence is that the implementation of $SMA(x)$ and its construction can be done in $O(|x|)$ time and space, independently of the alphabet size.

Complexity of searching with SMA

The complexities of the search with the string-matching automaton depend upon the implementation chosen for the automaton.

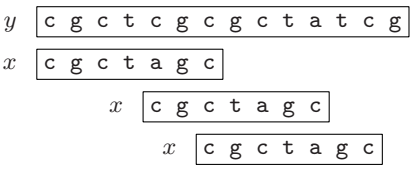
With a complete SMA implemented by transition matrix, the preprocessing on the pattern x can be done in time $O(m \times card(V))$ using a space in $O(m \times card(V))$. Then the search on the text y can be done in time $O(n)$ using a space in $O(m \times card(V))$. The delay is then constant.

With a SMA implemented by lists of forward and backward arcs. The preprocessing on the pattern x can be done in time $O(m)$ using a space in $O(m)$. Then the search on the text y can be done in time $O(n)$ using a space in $O(m)$. The delay becomes $\min\{card(V), \log_2 m\}$ comparisons. This constitutes an improvement on KMP algorithm.

1.1.7 Boyer-Moore algorithm

The Boyer-Moore string-matching algorithm [BM77] performs the scanning operations from right to left inside the window on the text.

Example 7. $x =$ `cgctagc` and $y =$ `cgctcgcgctatcg`



It uses two rules:

- the matching shift: good-suffix rule;
- the occurrence heuristics: bad-character rule;

to compute the length of the shift after each attempt. Extra rules can be used if some memorization are done from one attempt to the next.

The matching shift

A typical situation during the searching phase of the Boyer-Moore algorithm is depicted in Figure 1.11. During an attempt where the window is positioned on $y[j..j+m-1]$, a suffix $u = x[i+1..m-1]$ of x has been matched (from right to left) in y. A mismatch has occurred between symbol $x[i] = a$ in x and $y[j] = b$ in y.

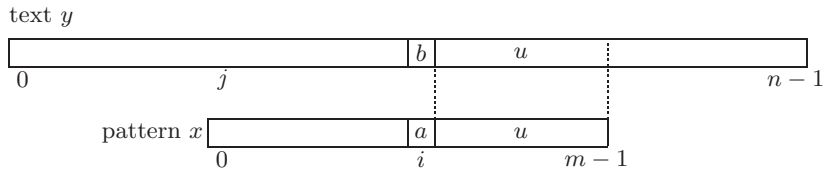


Fig. 1.11. A typical situation during the Boyer-Moore algorithm.

Then a valid shift consists in aligning the occurrence of u in y with a reoccurrence of u in x preceded by a symbol $c \neq a$ (see Figure 1.12). If no such reoccurrence exists, the shift consists in aligning the longest suffix of u in y which is a prefix of x (see Figure 1.13).

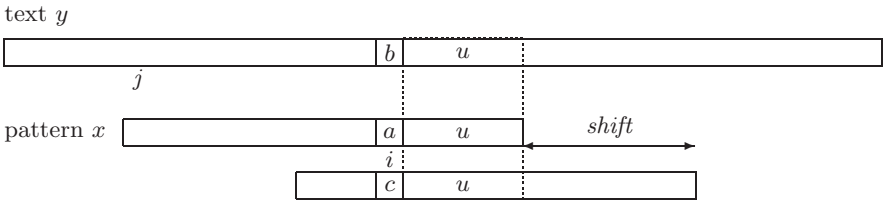


Fig. 1.12. The matching shift: a reoccurrence of u exists in x with $c \neq a$.

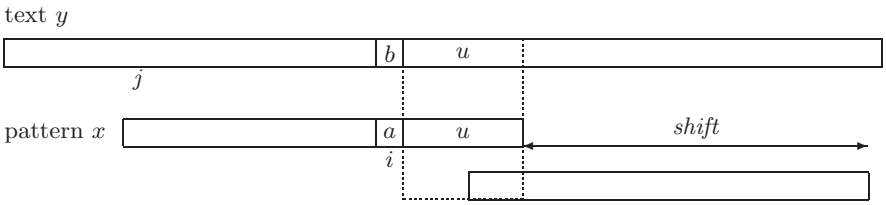


Fig. 1.13. The matching shift: no reoccurrence of u exists in x.

The first case for the matching shift which consists in the computation of the rightmost reoccurrences of each suffix u of x can be done in $O(m)$ time

and space. The second case which basically corresponds to the computation of the period of x can also be performed in $O(m)$ time and space.

A table D implements the good-suffix rule: for $i = 0, \ldots, m-1$,

$$D[i] = \min\{|z| > 0 \mid (x \text{ suffix of } x[i\,..\,m-1]z) \text{ or } (bx[i\,..\,m-1]z \text{ suffix of } x \text{ and } bx[i\,..\,m-1] \text{ not suffix of } x, \text{ for } b \in V)\}$$

and $D[m] = 1$.

The occurrence heuristics

During an attempt, of the searching phase of the Boyer-Moore algorithm, where the window is positioned on $y[j..j+m-1]$, a suffix u of x has been matched (from right to left) in y. A mismatch has occurred between symbol $x[i] = a$ in x and $y[i+j] = b$ in y. The occurrence shift consists in aligning the symbol b in y with its rightmost occurrence in x (possibly leading to a negative shift) (see Figure 1.14).

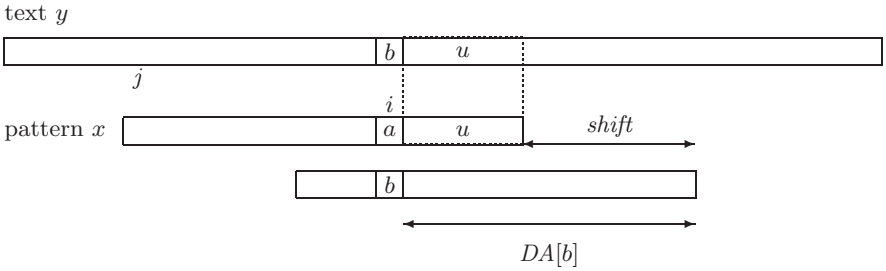


Fig. 1.14. The occurrence shift.

A table DA implements the bad-character rule: $DA[a] = \min(\{|z| > 0 \mid az \text{ suffix of } x\} \cup \{m\})$ for any symbol $a \in V$.

Then the length of the shift to apply is given by $DA[b] - |u| = DA[b] - m + i$.

BM algorithm

The Boyer-Moore string-matching algorithm performs no memorization of previous matches. It applies the maximum between the two shifts. It is presented in Figure 1.15.

Suffix displacement

For $0 \leq i \leq m-1$ we denote by $suf[i]$ the length of the longest suffix of x ending at position i in x. Let us denote by $lcsuf(u, v)$ the longest common suffix of two words u and v.

The computation of the table suf is done by the algorithm SUFFIXES presented in Figure 1.16. Figure 1.17 depicts the variables and the invariants

```
BM(x, m, y, n)
 1  j ← 0
 2  while j ≤ n − m do
 3     i ← m − 1
 4     while i ≥ 0 and x[i] = y[i + j] do
 5        i ← i − 1
 6     if i = −1 then
 7        OUTPUT(j)
 8     j ← j + max{D[i + 1], DA[y[i + j]] − m + i + 1}
```

Fig. 1.15. The Boyer-Moore algorithm.

```
SUFFIXES(x, m)
 1  suf[m − 1] ← m
 2  g ← m − 1
 3  for i ← m − 2 downto 0 do
 4     if i > g and suf[i + m − 1 − f] < i − g then
 5        suf[i] ← suf[i + m − 1 − f]
 6     else g ← min{g, i}
 7        f ← i
 8        while g ≥ 0 and x[g] = x[g + m − 1 − f] do
 9           g ← g − 1
10        suf[i] ← f − g
11  return suf
```

Fig. 1.16. Algorithm SUFFIXES.

of the main loop of algorithm SUFFIXES. The values of *suf* are computed for each position i in x in decreasing order. The algorithm uses two variables f and g which satisfy:

- $g = \min\{j - suf[j] \mid i < j < m - 1\}$;
- f is such that $i < f < m - 1$ and $f - suf[f] = g$.

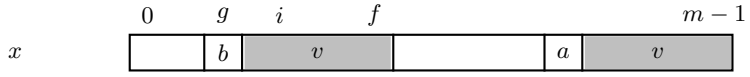


Fig. 1.17. Variables i, f, g of algorithm SUFFIXES. The main loop has invariants: $v = x[g + 1 \,..\, f] = lcsuf(x, x[0 \,..\, f])$ and $a \neq b$ $(a, b \in V)$ and $i < f$. The picture corresponds to the case where $g < i$.

We are now able to give, in Figure 1.18, the algorithm COMPUTE-D that computes the table D using the table *suf*. The invariants of the second loop of algorithm COMPUTE-D are presented in Fig. 1.19.

```
COMPUTE-D(x, m)
 1  j ← 0
 2  for i ← m − 1 downto −1 do
 3     if i = −1 or suf[i] = i + 1 then
 4        while j < m − 1 − i do
 5           D[j] ← m − 1 − i
 6           j ← j + 1
 7  for i ← 0 to m − 2 do
 8     D[m − 1 − suf[i]] ← m − 1 − i
 9  return D
```

Fig. 1.18. Computation of the matching shift.

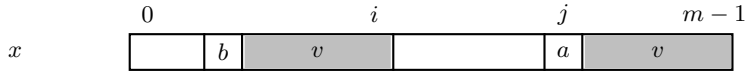


Fig. 1.19. Variables i and j of algorithm COMPUTE-D. Situation where $suf[i] < i + 1$. The loop of lines 7-8 has the following invariants: $v = lcsuf(x, x[0\,..\,i])$ and $a \neq b$ $(a, b \in V)$ and $suf[i] = |v|$. Thus $D[j] \leq m - 1 - i$ with $j = m - 1 - suf[i]$.

The algorithm of Figure 1.20 computes the table DA.

```
COMPUTE-DA(x, m)
 1  for each a ∈ V do
 2     DA[a] ← m
 3  for i ← 0 to m − 2 do
 4     DA[x[i]] ← m − i − 1
 5  return DA
```

Fig. 1.20. Computation of the occurrence shift.

Complexity of BM algorithm

Preprocessing phase: The match shift can be computed in $O(m)$ time while the occurrence shift can be computed in $O(m + card(V))$ time.

Searching phase: When one wants to find all the occurrences of the pattern int the text, the worst case running time of the Boyer-Moore string-matching algorithm is $O(n \times m)$. The minimum number of symbol comparisons is n/m and the maximum number of symbol comparisons $n \times m$.

Extra space: The extra space needed for the two shift functions is $O(m + card(V))$ and it can be reduced to $O(m)$.

Symbol comparisons in variants of BM

In [KMP77], it is proved that for finding the first occurrence of a pattern x of length m in a text y of length n, the BM algorithm performs no more than $7n$ comparisons between symbols of the text and symbols of the pattern. The bound is lowered to $4n$ comparisons in [GO80]. R. Cole [Col94] gives a tight bound of $3n-m$ comparisons for non-periodic patterns (*i.e.* $period(x) > m/2$)

For finding all the occurrences of a pattern x of length m in a text y of length n, linear variants of the BM algorithm have been designed. The Galil algorithm [Gal79] implements a prefix memorization technique when an occurrence of the pattern is located in the text. It gives a linear number of comparisons between symbols of the pattern and symbols of the text and requires a constant extra space.

The Turbo-BM algorithm [CCG+94] implements a last-suffix memorization technique which leads to a maximal of $2n$ comparisons. It also requires a constant extra space. Actually it stores the last match in the text when a matching shift is applied (the memorized factor is called *memory*) (see Figure 1.21). This enables it to perform jumps, in subsequent attempts, on these memorized factors of the text, saving thus some symbol comparisons. It can also perform, in some cases, larger shifts by using turbo-shifts. Its preprocessing is the same as the BM algorithm. The searching phase need an $O(1)$ extra space to store *memory* as a pair (length, right position).

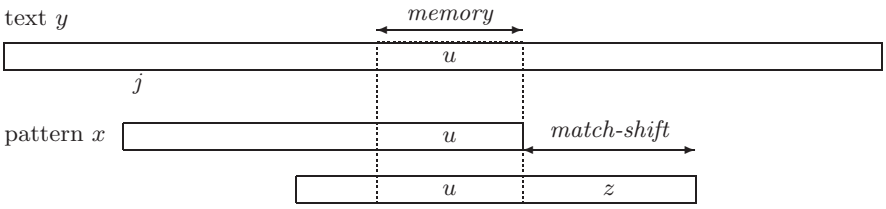


Fig. 1.21. When a match shift is applied the Turbo-BM algorithm memorizes the factor u of y.

The Apostolico-Giancarlo [AG86] implements an all-suffix memorization technique that gives a maximal number of comparisons equal to $1.5n$ [CL97]. It requires an $O(m)$ extra space. The Apostolico and Giancarlo remembers the length of the longest suffix of the pattern ending at the right position of the window at the end of each attempt (see Figure 1.22). These information are stored in a table *skip*. Let us assume that during an attempt at a position less than j the algorithm has matched a suffix of x of length k at position $j+i$ with $0 < i < m$ then $skip[j+i]$ is equal to k. Let $suf[i]$, for $0 \leq i < m$ be equal to the length of the longest suffix of x ending at the position i in x (see Section 1.1.7). During the attempt at position j, if the algorithm compares

successfully the factor of the text $y[j+i+1..j+m-1]$ only in the case where $k = suf[i]$, a "jump" has to be done over the text factor $y[j+i-k+1..j+i]$ in order to resume the comparisons between the symbols $y[j+i-k]$ and $x[i-k]$. In all the other cases, no more comparisons have to be done to conclude the attempt and a shift can be performed.

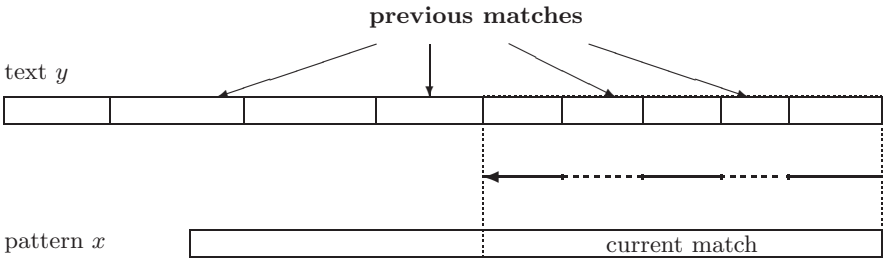


Fig. 1.22. The Apostolico-Giancarlo algorithm stores all the matches that occur between suffixes of x and subwords of y.

1.2 Searching a list of strings — Suffix Arrays

In this section we consider two main questions that are related by the technique used to solve them. The first question on word list searching is treated in the first subsection, and the second one, indexing a text, is treated in Subsection 1.2.3.

1.2.1 Searching a list of words

Input a list L of n strings of V^* stored in increasing lexicographic order in a table: $L_0 \leq L_1 \leq \cdots \leq L_{n-1}$ and a string $x \in V^*$ of length m.
Problem find either i, $-1 < i < n$, with $x = L_i$ if x occurs in L, or d, $-1 \leq d \leq n$, that satisfy $L_d < x < L_{d+1}$ otherwise.

Example 8. **List** L

$$
\begin{aligned}
L_0 &= \texttt{a a a b a a}\\
L_1 &= \texttt{a a a b b}\\
L_2 &= \texttt{a a b b b b}\\
L_3 &= \texttt{a b}\\
L_4 &= \texttt{b a a a}\\
L_5 &= \texttt{b b}
\end{aligned}
$$

The search for `aaabb` outputs 1 as does the search for `aaba`.

1.2.2 Searching algorithm

A standard way of solving the problem is to use a binary search because the list of strings is sorted. Its presentation below makes use of the function *lcp* that computes the longest common prefix (LCP) of two strings.

```
SIMPLE-SEARCH(L, n, x, m)
 1  d ← −1
 2  f ← n
 3  while d + 1 < f do
        ▷ Invariant: L_d < x < L_f
 4    i ← ⌊(d + f)/2⌋
 5    ℓ ← |lcp(x, L_i)|
 6    if ℓ = m and ℓ = |L_i| then
 7      return i
 8    else if (ℓ = |L_i|) or (ℓ ≠ m and L_i[ℓ] < x[ℓ]) then
 9      d ← i
10    else f ← i
11  return d
```

The running time of the binary search is $O(m \times \log n)$ if we assume that the LCP computation of two string takes a linear time, doing it by pairwise symbol comparisons. The worst case is met with the list $L = (\mathtt{a}^{m-1}\mathtt{b}, \mathtt{a}^{m-1}\mathtt{c}, \mathtt{a}^{m-1}\mathtt{d}, \dots)$ and the string $x = \mathtt{a}^m$.

Indeed, it is possible to reduce the running time of the binary search to $O(m + \log n)$ by storing the LCPs of some pairs of strings of the list. These pairs are of the form (L_d, L_f) where (d, f) is a pair of possible values of d and f in the binary search algorithm. Since there are $2n + 1$ such pairs, the extra space required by the new algorithm SEARCH is $O(n)$.

The design of the algorithm is based on properties arising in three cases (plus symmetric cases) that are described below. The algorithm maintains three variables defined as: $ld = |lcp(x, L_d)|$, $lf = |lcp(x, L_f)|$, $i = \lfloor (d + f)/2 \rfloor$. In addition, the main invariant of the loop of the algorithm is $L_d < x < L_f$.

Case one

If $ld \leq |lcp(L_i, L_f)| < lf$, then $L_i < x < L_f$ and $|lcp(x, L_i)| = |lcp(L_i, L_f)|$.

Case two

If $ld \leq lf < |lcp(L_i, L_f)|$, then $L_d < x < L_i$ and $|lcp(x, L_i)| = |lcp(x, L_f)|$.

Case three

If $ld \leq lf = |lcp(L_i, L_f)|$, then we have to compare x and L_i to discover if they match or which one is the smallest. But this comparison symbol by symbol is to start at position *lf* because the strings have a common prefix of length *lf*.

The resulting algorithm including the symmetric cases where $lf \leq ld$ is given in Figure 1.23 and it satisfies the next proposition because LCP can be implemented to run in constant time after preprocessing the list (in time linear in the input size).

```
SEARCH(L, n, Lcp, x, m)
 1  (d, ld) ← (−1, 0)
 2  (f, lf) ← (n, 0)
 3  while d + 1 < f do
        ▷ Invariant : L_d < x < L_f
 4     i ← ⌊(d + f)/2⌋
 5     if ld ≤ Lcp(i, f) < lf then
 6        (d, ld) ← (i, Lcp(i, f))
 7     else if ld ≤ lf < Lcp(i, f) then
 8        f ← i
 9     else if lf ≤ Lcp(d, i) < ld then
10        (f, lf) ← (i, Lcp(d, i))
11     else if lf < ld < Lcp(d, i) then
12        d ← i
13     else ℓ ← max{ld, lf}
14        ℓ ← ℓ + |lcp(x[ℓ .. m − 1], L_i[ℓ .. |L_i| − 1])|
15        if ℓ = m and ℓ = |L_i| then
16           return i
17        else if (ℓ = |L_i|) or (ℓ ≠ m and L_i[ℓ] < x[ℓ]) then
18           (d, ld) ← (i, ℓ)
19        else (f, lf) ← (i, ℓ)
20 return d
```

Fig. 1.23. Search for x in L in time $O(m + \log n)$.

Proposition 1. *Algorithm* SEARCH *finds a string x of length m in a sorted list of n strings in time $O(m + \log n)$.*

It makes no more than $m+\lceil \log(n+1) \rceil$ comparisons of symbols and requires $O(n)$ extra space.

A straightforward extension of the algorithm SEARCH used for suffix arrays in the rest of the section computes the pair (d, f), $-1 \leq d < f \leq n$, that satisfies: $d < i < f$ if and only if x prefix of L_i.

Preprocessing the list is a classical matter.

Sorting can be done by repetitive applications of bin sorting and takes time $O(||L||)$, where $||L|| = \sum_{i=0}^{n-1} |L_i|$.

Computing LCPs of strings consecutive in the sorted list takes the same time by mere symbol comparisons. Computing other LCPs is based on next lemma and takes time $O(n)$.

Lemma 2. *Let $L_0 \leq L_1 \leq \ldots \leq L_{n-1}$. Let d, i and f, $-1 < d < i < f < n$. Then $|lcp(L_d, L_f)| = \min\{|lcp(L_d, L_i)|, |lcp(L_i, L_f)|\}$.*

So, the complete preprocessing time is $O(||L||)$.

1.2.3 Suffix array

A suffix array is a structure for indexing texts. It is used for the implementation of indexes supporting operations of searching for patterns, their number of occurrences, or their list of positions. Contrary to suffix trees or suffix automata whose efficiency relies on the design of a data structure, suffix arrays are grounded on efficient algorithms, one of them being the search algorithm of the previous section.

The suffix array of a text $y \in V^*$ of length n is composed of the elements described for the list of strings, applied to the list of suffixes of the text. So, it consists of both the permutation of positions on the text that gives the sorted list of suffixes and the corresponding array of lengths of their LCPs. They are denoted by p and LCP and defined by:

$$y[p[0]\,..\,n-1] < y[p[1]\,..\,n-1] < \ldots < y[p[n-1]\,..\,n-1]$$

and

$$LCP[i] = |lcp(y[p[i-1]\,..\,n-1], y[p[i]\,..\,n-1])|.$$

Example 9. $y =$ `aabaabaabba`

i	$p[i]$	$LCP[i]$	
0	10	0	`a`
1	0	1	`a a b a a b a a b b a`
2	3	6	`a a b a a b b a`
3	6	3	`a a b b a`
4	1	1	`a b a a b a a b b a`
5	4	5	`a b a a b b a`
6	7	2	`a b b a`
7	9	0	`b a`
8	2	2	`b a a b a a b b a`
9	5	4	`b a a b b a`
10	8	1	`b b a`

There are several algorithms for computing a suffix array efficiently, two of them running in linear time are presented here as a sample. Note that the solutions of Section 1.2.1 would lead to algorithms running in time $O(n^2)$ because $||\mathrm{Suf}(y)|| = O(n^2)$. But they would not exploit the dependencies between the suffixes of the text.

We consider that the alphabet is a bounded segment of integers, as it can be considered in most real applications. The schema for sorting the suffixes is as follows.

1. bucket sort positions i according to $First_3(y[i\,..\,n-1])$, ($First_3(x)$ is either the first three symbols of x if $|x| \geq 3$ or x if $|x| < 3$ for a string $x \in V^*$) for $i = 3q$ or $i = 3q+1$;
   let $t[i]$ be the rank of i in the sorted list.
2. recursively sort the suffixes of the 2/3-shorter word
   $t[0]t[3]\cdots t[3q]\cdots t[1]t[4]\cdots t[3q+1]\cdots$
   let $s[i]$ be the rank of suffix i in the sorted list ($i = 3q$ or $i = 3q+1$)
3. sort suffixes $y[j\,..\,n-1]$, for j of the form $3q+2$, by bucket sorting pairs $(y[j], s[j+1])$.
4. merge lists obtained at steps 2 and 3
   Note: comparing suffixes i (first list) and j (second list) remains to compare:
   $(x[i], s[i+1])$ and $(x[j], s[j+1])$ if $i = 3q$
   $(x[i]x[i+1], s[i+2])$ and $(x[j]x[j+1], s[j+2])$ if $i = 3q+1$

The recursivity of the algorithm yields the recurrence relation $T(n) = T(2n/3) + O(n)$ for its running time, which gives $T(n) = O(n)$.

Example 10. $y =$ `aabaabaabba`

i	0	1	2	3	4	5	6	7	8	9	10
$y[i]$	a	a	b	a	a	b	a	a	b	b	a

Rank t	
0	a
1	a a b
2	a b a
3	a b b
4	b a

Rank s	i	Suf(11142230)
0	10	0
1	0	1 1 1 4 2 2 3 0
2	3	1 1 4 2 2 3 0
3	6	1 4 2 2 3 0
4	1	2 2 3 0
5	4	2 3 0
6	7	3 0
7	9	4 2 2 3 0

Rank j	$(y[j], s[j+1])$
0	2 (b, 2)
1	5 (b, 3)
2	8 (b, 7)

i	0	1	2	3	4	5	6	7	8	9	10
$y[i]$	a	a	b	a	a	b	a	a	b	b	a
$r[i]$	1	4	8	2	5	9	3	6	10	7	0
$p[i]$	10	0	3	6	1	4	7	9	2	5	8

Table r is defined by: $r[j]$ = rank of suffix at position j in the sorted list of all suffixes. It is the inverse of p.

There is a second linear-time algorithm for computing LCPs (see Figure 1.24) of consecutive suffixes in the sorted list (other LCPs are computed as in Section 1.2.1). Its running time analysis is straightforward. The next example illustrates the following lemma that is the clue of algorithm Lcp.

Example 11. $y =$ `aabaabaabba`

i	0	1	2	3	4	5	6	7	8	9	10	11
$y[i]$	a	a	b	a	a	b	a	a	b	b	a	
$p[i]$	10	0	3	6	1	4	7	9	2	5	8	
$LCP[i]$	0	1	6	3	1	5	2	0	2	4	1	0

j	$r[j]$	
0	1	a a b a a b a a b b a
3	2	a a b a a b b a

j	$r[j]$	
1	4	a b a a b a a b b a
4	5	a b a a b b a

Lemma 3. *Let $j \in (1, 2, \ldots, n-1)$ with $r[j] > 0$. Then $LCP[r[j-1]] - 1 \leq LCP[r[j]]$.*

```
LCP(y, n, p, r)
 1  ℓ ← 0
 2  for j ← 0 to n − 1 do
 3     ℓ ← max{0, ℓ − 1}
 4     if r[j] > 0 then
 5        i ← p[r[j] − 1]
 6        while y[i + ℓ] = y[j + ℓ] do
 7           ℓ ← ℓ + 1
 8        LCP[r[j]] ← ℓ
 9  LCP[0] ← 0
10  LCP[n] ← 0
11  return LCP
```

Fig. 1.24. Computation of the LCPs

The next statement summarizes the elements of the present section.

Proposition 2. *Computing the suffix array of a text of length n can be done in time $O(n)$ with $O(n)$ memory space.*

1.3 Indexes

Indexes are data structures that are used to solve the pattern matching problem in static texts. An index for a text y of is a structure that contains all the factors of y. It must enable to deal with the following basic operations:

String-matching: computing the existence of a pattern x of length m in the text y;
All occurrences: computing the list of positions of occurrences of a pattern x of length m in y;
Repetitions: computing a longest subword of y occurring at least k times;
Marker: computing a shortest subword of y occurring exactly once.

Other possible applications includes:

- finding all the repetitions in texts;
- finding regularities in texts;
- approximate matchings.

1.3.1 Implementation of indexes

Indexes are implemented by suffix arrays or by suffix trees or suffix automata in $O(n)$ space. Such structures represent all the subwords of y since every subword of y is a prefix of a suffix of y. Table 1.1 summarizes the complexities of different operations on indexes with these structures.

	suffix array	suffix tree or suffix automaton
Construction	$O(n)$	$O(n \times \log card(V))$
String-matching	$O(m + \log n)$	$O(m \times \log card(V))$
All occurrences	$O(m + \log n + \|output\|)$	$O(m \times \log card(V)) + \|output\|)$
Repetitions	$O(n)$	$O(n)$
Marker	$O(n)$	$O(n)$

Table 1.1. Complexities of different operations on indexes.

1.3.2 Efficient constructions

The notion of position tree is due to Weiner [Wei73], who presents an algorithm for computing its compact version. An off-line computation of suffix trees is given by McCreight [McC76]. Ukkonen [Ukk92] gives an on-line algorithm and Farach [Far97] designs an alphabet independent algorithm for the suffix tree construction. Other implementations of suffix trees are given in [AN93, Kär95, Irv96, MRR01, GGV04].

The suffix automaton is also kwown as the DAWG for Directed Acyclic Word Graph. Its linearity was discovered by Blumer *et al.* [BBE$^+$83]. The minimality of the structure as an automaton is due to Crochemore [Cro84] who shown how to construct the factor automaton with the same complexity.

PAT arrays were designed by Gonnet [Gon87]. Suffix arrays were first designed by Manber and Myers [MM93], for recent results see [KS03, KSPP03, KA03].

SB-trees are used to store this structures in external memory [FG99].

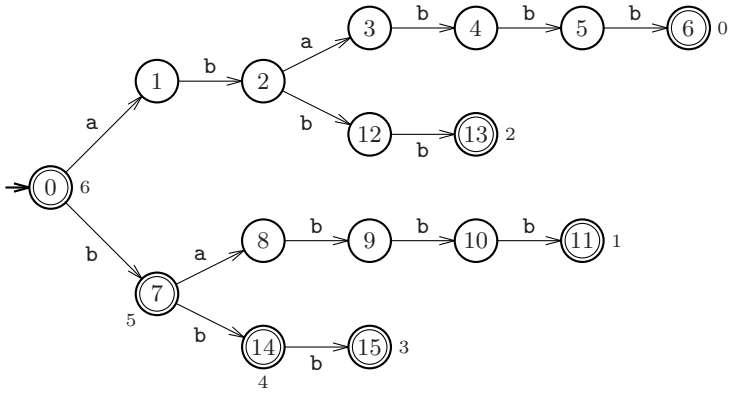
Crochemore and Vérin [CV97] first introduced compact suffix automata. An on-line algorithm for its construction is given in [IHS$^+$01].

1.3.3 Trie of suffixes

The trie of suffixes of y, denoted by $\mathcal{T}(y)$ is a digital tree which branches are labeled by suffixes of y. Actually it is a tree-like deterministic automaton accepting the language Suf(y).

Nodes of $\mathcal{T}(y)$ are identified with subwords of y. **Terminal nodes** of $\mathcal{T}(y)$ are identified with suffixes of y. An output is defined, for each terminal node, which is the starting position of the suffix in y.

Example 12. Suffix trie of `ababbb`

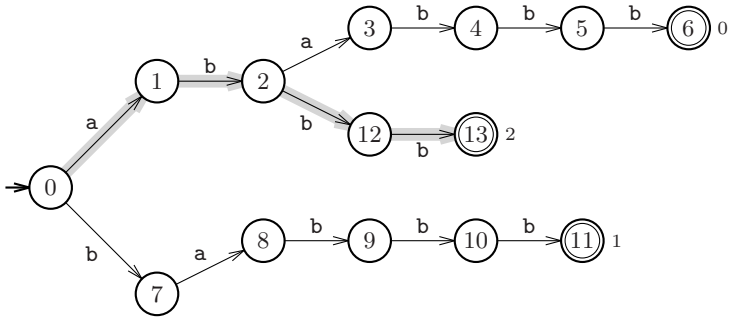


Starting with an empty tree, the trie $\mathcal{T}(y)$ is build by successively inserting the suffixes of y from the longest one (y itself) to the shortest one (the empty word).

Forks

Let us examine the insertion of $u = y[i\,..\,n-1]$ in the structure accepting longer suffixes $(y, y[1\,..\,n-1], \ldots, y[i-1\,..\,n-1])$. The **head** of u is the longest prefix $y[i\,..\,k-1]$ of u occurring before i. The **tail** of u is the the rest $y[k\,..\,n-1]$ of suffix u.

Example 13. With $y =$ `ababbb`, the head of `abbb` is `ab` and its tail is `bb`.



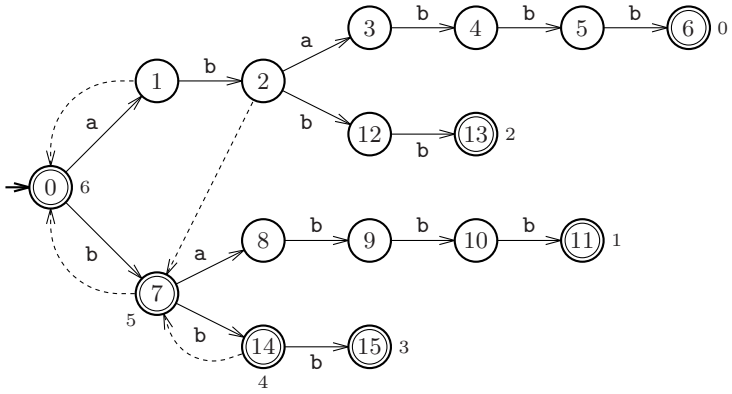
A **fork** is any node that has out-degree 2 at least, or that has both out-degree 1 and is terminal. The head of a prefix of y is a fork. The initial node is a fork if and only if y is non empty.

The insertion of a suffix $u = y[i\,..\,n-1]$ consists first in finding the fork corresponding to the head of u and then in inserting the tail of u from this fork.

Suffix link

A **function** s_y, called *suffix link* is defined as follows: if node p is identified with subword av, $a \in V, v \in V^*$ then $s_y(p) = q$ where node q is identified with v.

Example 14. Suffix links are represented by dotted arrows.

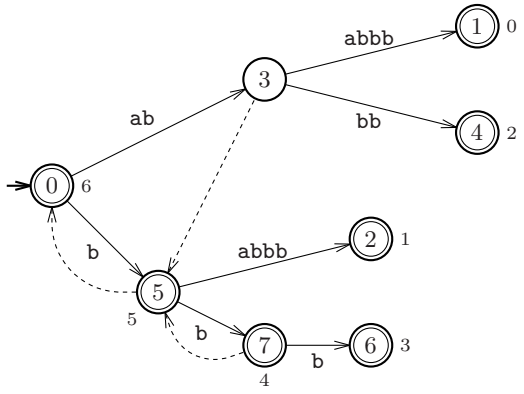


The suffix links create shortcuts that are used to accelerate heads computations. It is useful for forks only. If node p is a fork, so is $s_y(p)$. If the head of $y[i-1\,..\,n-1]$ is of the form au ($a \in V, u \in V^*$) then u is a prefix of the head of $y[i\,..\,n-1]$. Then, using suffix links, the insertion of the suffix $y[i\,..\,n-1]$ consists first in finding the fork corresponding to the head of $y[i\,..\,n-1]$ (starting from suffix link of the fork associated with au) and then in inserting the tail of $y[i\,..\,n-1]$ from this fork.

1.3.4 Suffix Tree

The suffix tree of y, denoted by $\mathcal{S}(y)$, is a compact trie accepting the language $\mathrm{Suf}(y)$. It is obtained from the suffix trie of y by deleting all nodes having out-degree 1 that are not terminal. Edges are then labeled by subwords of y instead of symbols.

Example 15. $\mathcal{S}(\texttt{ababbb})$

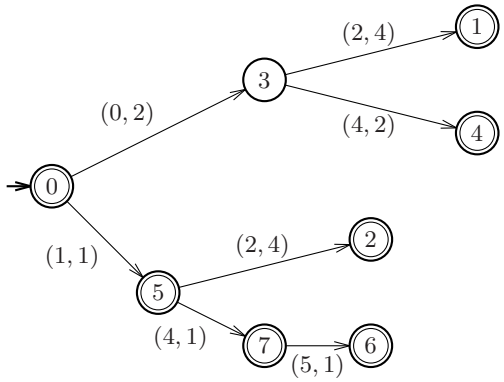


The number of nodes of $\mathcal{S}(y)$ is not more than $2n$ (if $n > 0$) since all internal nodes either have two children at least or are terminal and there are at most n terminal nodes.

Labels of edges

The edge labels are represented by pairs (j, ℓ) representing subwords $y[j\,..\,j+\ell-1]$ of y.

Example 16. $\mathcal{S}$(ababbb)



i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

This technique requires to have y residing in main memory. Thus the size of $\mathcal{S}(y)$ is $O(n)$.

Scheme of suffix tree construction

The algorithm for building the suffix tree of y is given in Figure 1.25. It uses algorithms Fast-Find and Slow-Find that are described next. Starting with an empty tree, $\mathcal{S}(y)$ is built by successively inserting the suffixes of y from

the longest one (y itself) to the shortest one (the empty word). Using suffix links, the insertion of the suffix $y[i\,..\,n-1]$ consists first in finding the fork corresponding to the head of $y[i\,..\,n-1]$ (starting from suffix link of the fork associated with the head of $y[i-1\,..\,n-1]$) and then in inserting the tail of $y[i\,..\,n-1]$ from this fork.

```
SUFFIX-TREE(y, n)
 1  T ← NEW-AUTOMATON()
 2  for i ← 0 to n − 1 do
 3     find fork of head of y[i..n − 1] using
 4     FAST-FIND from node s_y(fork), and then SLOW-FIND
 5     k ← position of tail of y[i..n − 1]
 6     if k < n then
 7        q ← NEW-STATE()
 8        Adj[fork] ← Adj[fork] ∪ {((k, n − k), q)}
 9        output[q] ← i
10     else output[fork] ← i
11  return T
```

Fig. 1.25. Scheme of the construction of the suffix tree of string y of length n.

This scheme requires an adjacency-list representation of labeled arcs.

Let us examine more closely the insertion of the suffix $y[i\,..\,n-1]$ in the tree. The search for the node associated with the head of $y[i\,..\,n-1]$ proceeds in two steps:

1. Assume that the head of $y[i-1\,..\,n-1]$ is $auv = y[i-1\,..\,k-1]$ ($a \in V, u, v \in V^*$) and let *fork* be the associated node. If the suffix link of *fork* is defined, it leads to node s, then the second step starts from this node. Otherwise, the suffix link from *fork* is found by rescanning as follows. Let r be the parent node of *fork* and let (j, ℓ) be the label of edge $(r, fork)$. For the ease of description, assume that $auv = au(y[k-\ell\,..\,k])$ (it may happened that $auv = y[k-\ell\,..\,k]$). There is a suffix link from node r to node p associated with v. The crucial observation here is that $y[k-\ell\,..\,k]$ is the prefix of the label of some branch starting at node p. Then the algorithm rescans $y[k-\ell\,..\,k]$ in the tree: let q be the child of p along that branch and let (h, m) be the label of the edge (p, q). If $m < \ell$ then a recursive scan of $y[k-\ell+m\,..\,k]$ starts from node q. If $m > \ell$ then the edge (p, q) is broken to insert a new node s; labels are updating correspondingly. If $m = \ell$, s is simply set to q. This search is performed by the algorithm FAST-FIND given in Figure 1.26. The suffix link of *fork* is then set to s.
2. A downward search starts from node s to find the fork associated with the head of $y[i\,..\,n-1]$. This search is dictated by the symbols of the tail of $y[i\,..\,n-1]$, one by one from left to right. If necessary a new internal node is created at the end of this scanning (see Figure 1.27).

```
FAST-FIND(r, j, k)
 1 if j ≥ k then
 2    return r
 3 else q ← TARGET(r, y[j])
 4    (j', ℓ) ← label(r, q)
 5    if j + ℓ ≤ k then
 6       return FAST-FIND(q, j + ℓ, k)
 7    else Adj[r] ← Adj[r] \ {((j', ℓ), q)}
 8       p ← NEW-STATE()
 9       Adj[r] ← Adj[r] ∪ {((j, k − j), p)}
10       Adj[p] ← Adj[p] ∪ {((j' + k − j, ℓ − k + j), q)}
11       return p
```

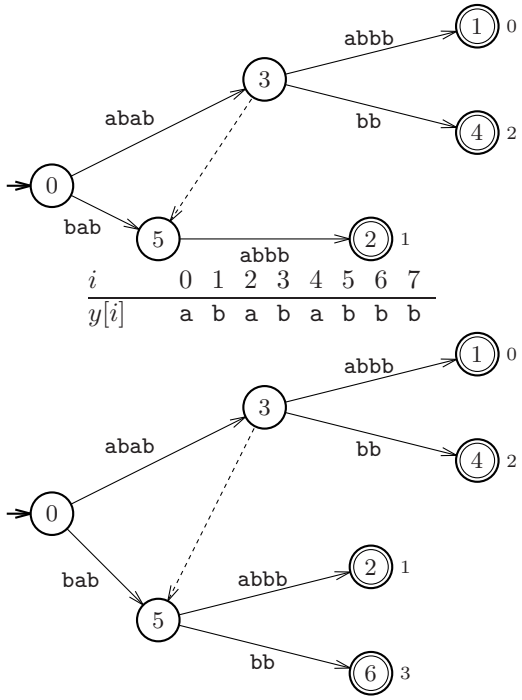
Fig. 1.26. Search for $y[j\,..\,k]$ from node r.

```
SLOW-FIND(p, k)
 1 while k < n and TARGET(p, y[k]) ≠ NIL do
 2    q ← TARGET(p, y[k])
 3    (j, ℓ) ← label(p, q)
 4    i ← j
 5    do
 6       i ← i + 1
 7       k ← k + 1
 8    while i < j + ℓ and k < n and y[i] = y[k]
 9    if i < j + ℓ then
10       Adj[p] ← Adj[p] \ {((j, ℓ), q)}
11       r ← NEW-STATE()
12       Adj[p] ← Adj[p] ∪ {((j, i − j), r)}
13       Adj[r] ← Adj[r] ∪ {((j + i − j, ℓ − i + j), q)}
14       return (r, k)
15    p ← q
16 return (p, k)
```

Fig. 1.27. Search of the longest prefix of $y[k\,..\,n-1]$ from node p. A new node is created when the target lies in the middle of an arc.

The insertion of the tail from the fork associated to the head of $y[i\,..\,n-1]$ is done by adding a new edge labeled by the tail leading to a new node. It is done in constant time.

Example 17. $\mathcal{S}$(`abababbb`)
End of insertion of suffix `babbb`



The head of babbb is bab so its tail is bb.

Complete algorithm

We are now able to give the complete algorithm for building the suffix tree of a text y of length n (see Figure 1.28). A table s implements the suffix links.

Complexity

The execution of SUFFIX-TREE(y) takes $O(|y| \times \log card(V))$ time in the comparison model. Indeed the main iteration increments i, which never decreases, iterations in FAST-FIND increment j, which never decreases, iterations in SLOW-FIND increment k, which never decreases and basic operations run in constant time or in time $O(\log card(V))$ time in the comparison model.

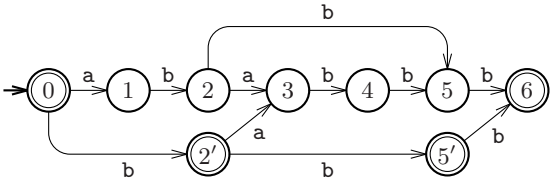
1.3.5 Suffix Automaton

The minimal deterministic automaton accepting Suf(y) is denoted by $\mathcal{A}(y)$. It can be seen as the minimization of the trie $\mathcal{T}(y)$ of suffixes of y.

Example 18. $\mathcal{A}$(ababbb)

```
SUFFIX-TREE(y, n)
 1  T ← NEW-AUTOMATON()
 2  s[initial[T]] ← initial[T]
 3  (fork, k) ← (initial[T], 0)
 4  for i ← 0 to n − 1 do
 5     k ← max{k, i}
 6     if s[fork] = NIL then
 7        r ← parent of fork
 8        (j, ℓ) ← label(r, fork)
 9        if r = initial[T] then
10           ℓ ← ℓ − 1
11        s[fork] ← FAST-FIND(s[r], k − ℓ, k)
12     (fork, k) ← SLOW-FIND(s[fork], k)
13     if k < n then
14        q ← NEW-STATE()
15        Adj[fork] ← Adj[fork] ∪ {((k, n − k), q)}
16        output[q] ← i
17     else output[fork] ← i
18  return T
```

Fig. 1.28. The complete construction of the suffix tree of y of length n.



The states of $\mathcal{A}(y)$ are classes of factors (subwords) of y. Two subwords u and v of y are in the same equivalence class if they have the same right context in y. Formally $u \equiv_y v$ iff $u^{-1}\mathrm{Suf}(y) = v^{-1}\mathrm{Suf}(y)$.

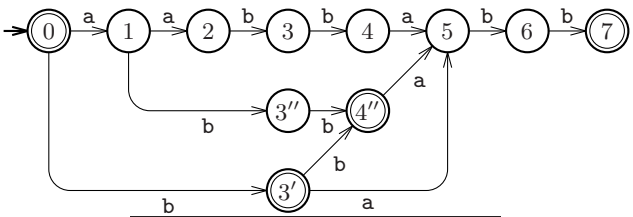
The suffix automaton $\mathcal{A}(y)$ has a linear size:

- it has between $n+1$ and $2n-1$ states;
- it has between n and $3n-4$ arcs.

Suffix link

A function f_y, also called *suffix link* is defined as follows:
let $p = \textsc{Target}(initial[\mathcal{A}(y)], v)$, $v \in V^+$, $f_y(p) = \textsc{Target}(initial[\mathcal{A}(y)], u)$, where u is the longest suffix of v occurring in a different right context ($u \not\equiv_y v$).

Example 19. $\mathcal{A}(\texttt{aabbabb})$



i	1	2	3	$3'$	$3''$	4	$4''$	5	6	7
$f_y(i)$	0	1	$3''$	0	$3'$	$4''$	$3'$	1	$3''$	$4''$

Suffix path

For a state p of $\mathcal{A}(y)$, the suffix path of p denoted by $SP_y(p)$ is defined as follows:

$$SP_y(p) = \langle p, f_y(p), f_y^2(p), \ldots \rangle.$$

Solid arc

For a state p of $\mathcal{A}(y)$, we denote by $L_y(p)$ the length of the longest string u in the class of p. It also corresponds to the length of the longest path from the initial state to state p (this path is labeled by u). An arc (p, a, q) of $\mathcal{A}(y)$ is solid iff $L_y(q) = L_y(p) + 1$.

Construction

Starting with a single state, the automaton $\mathcal{A}(y)$ is build by successively inserting the symbols of y from $y[0]$ to $y[n-1]$. The algorithm is presented in Figure 1.29. Tables f and L implements functions f_y and L_y respectively. Let us assume that $\mathcal{A}(w)$ is correctly build for a prefix w of y and let *last* be the state of $\mathcal{A}(w)$ corresponding to the class of w. The algorithm EXTENSION(a) builds $\mathcal{A}(wa)$ from $\mathcal{A}(w)$ (see Figure 1.30). This algorithm creates a new state *new*. Then in the first while loop, transitions (p, a, new) are created for the first states p of $SP_y(last)$ that do not already have a defined transition for the symbol a. Let q be the first state of $SP_y(last)$ for which a transition is defined for the symbol a, if such a state exists. When the first while loop of EXTENSION(a) ends three cases can arise:

1. p is not defined;
2. (p, a, q) is a solid arc;
3. (p, a, q) is not a solid arc.

Case 1: This situation arises when a does not occur in w. We have then $f_y(new) = initial[\mathcal{A}(w)]$.

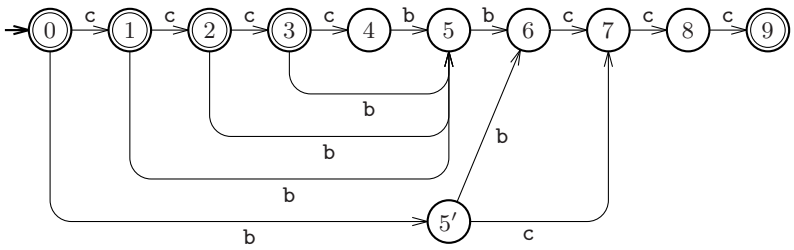
Case 2: Let u be the longest string recognized in state p ($|u| = L_y(p)$). Then ua is the longest suffix of wa that is a subword of w. Thus $f_y(new) = q$.

Case 3: Let u be the longest string recognized in state p ($|u| = L_y(p)$). Then ua is the longest suffix of wa that is a subword of w. Since the arc (p, a, q) is not solid, ua is not the longest string recognized in state q. Then state q is splitted into two states: the old state q and a new state *clone*. The state *clone* has the same transitions than q. The strings v (of the form $v'a$) shorter than ua that were recognized in state q are now recognized in state *clone*.

```
SUFFIX-AUTOMATON(y, n)
 1  T ← NEW-AUTOMATON()
 2  L[initial[T]] ← 0
 3  f[initial[T]] ← NIL
 4  last ← initial[T]
 5  for j ← 0 to n − 1 do
        ▷ Extension of T with the symbol y[j]
 6      last ← EXTENSION(y[j])
 7  p ← last
 8  do
 9      terminal[p] ← TRUE
10      p ← f[p]
11  while p ≠ NIL
12  return T
```

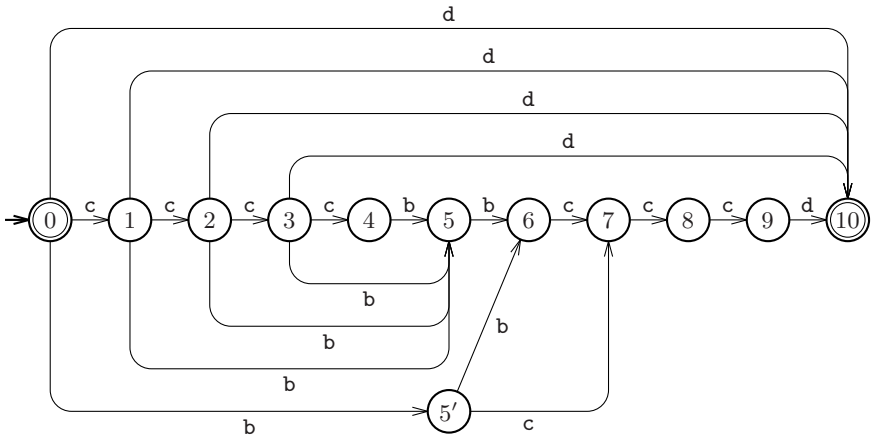
Fig. 1.29. Construction of $\mathcal{A}(y)$.

Example 20. One step: from $\mathcal{A}$(cccbbccc) to $\mathcal{A}$(ccccbbcccd)

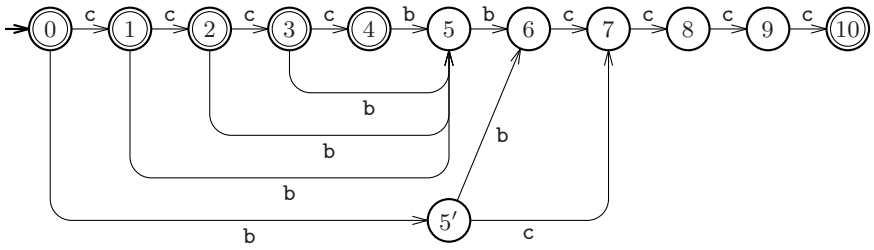


```
EXTENSION(a)
 1  new ← NEW-STATE()
 2  L[new] ← L[last] + 1
 3  p ← last
 4  do
 5     Adj[p] ← Adj[p] ∪ {(a, new)}
 6     p ← f[p]
 7  while p ≠ NIL and TARGET(p, a) = NIL
 8  if p = NIL then
 9     f[new] ← initial[T]
10  else q ← TARGET(p, a)
11     if (p, a, q) is solid, i.e. L[p] + 1 = L[q] then
12        f[new] ← q
13     else clone ← NEW-STATE()
14        L[clone] ← L[p] + 1
15        for each pair (b, q′) ∈ Succ[q] do
16           Adj[clone] ← Adj[clone] ∪ {(b, q′)}
17        f[new] ← clone
18        f[clone] ← f[q]
19        f[q] ← clone
20        do
21           Adj[p] ← Adj[p] \ {(a, q)}
22           Adj[p] ← Adj[p] ∪ {(a, clone)}
23           p ← f[p]
24        while p ≠ NIL and TARGET(p, a) = q
25  return new
```

Fig. 1.30. Construction of $\mathcal{A}(wa)$ from $\mathcal{A}(w)$ for w a prefix of y.

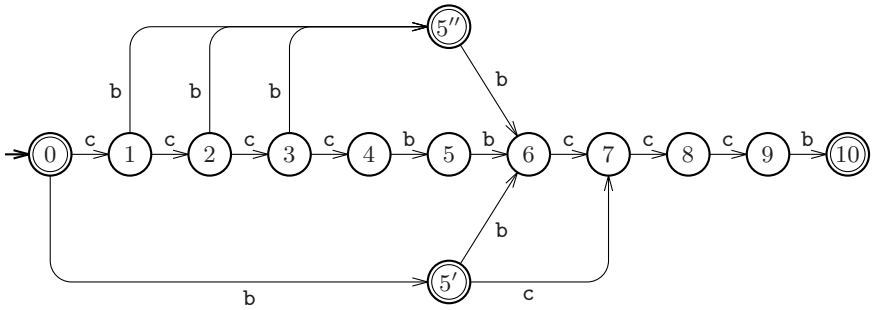


New arcs are created from states of the suffix path $\langle 9, 3, 2, 1, 0\rangle$.
From $\mathcal{A}(\texttt{ccccbbccc})$ to $\mathcal{A}(\texttt{ccccbbcccc})$



$f[9] = 3$ and $(3, \mathtt{c}, 4)$ is a solid arc (not a shortcut) then, $f[10] =$ TARGET$(3, \mathtt{c}) = 4$.

From $\mathcal{A}(\mathtt{ccccbbccc})$ to $\mathcal{A}(\mathtt{ccccbbcccb})$

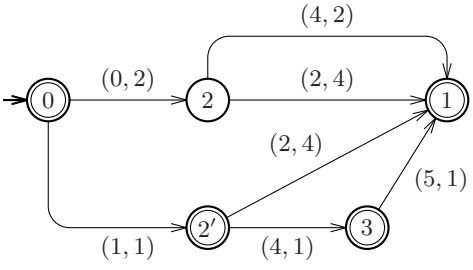


$f[9] = 3$, $(3, \mathtt{b}, 5)$is a non-solid arc , `cccb` is a suffix but `ccccb` is not: state 5 is cloned into $5'' = f[10] = f[5]$, $f[5''] = 5'$. Arcs $(3, \mathtt{b}, 5)$, $(2, \mathtt{b}, 5)$ et $(1, \mathtt{b}, 5)$ are redirected onto $5''$.

1.3.6 Compact Suffix Automaton

The suffix tree results from a compaction of the suffix trie while the minimal suffix automaton results from a minimization of the suffix trie. Minimizing the suffix tree or compacting the minimal suffix automaton results in the same structure called the compact suffix automaton.

Example 21. Compact suffix automaton of `ababbb`



i	0	1	2	3	4	5
$y[i]$	a	b	a	b	b	b

The size of the compact suffix automaton of a string y is linear in the length of y.

Direct construction of the compact suffix automaton

The direct construction of the compact suffix automaton is similar to both the suffix tree construction or the suffix automaton construction [CV97]. It consists of the sequential addition of suffixes in the structure from the longest one (y) to the shortest one (λ).

It uses the following features:

- "slow-find" and "fast-find" procedures;
- suffix links;
- solid and non-solid arcs;
- state splitting;
- re-directions of arcs.

The compact suffix automaton can be built in $O(n \log card(V))$ time using $O(n)$ space. In practice it can save up to 50% space on the suffix automaton [HC02].

References

[AG86] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.

[AN93] A. Andersson and S. Nilsson. Improved behavior of tries by adaptive branching. *Inf. Process. Lett.*, 46(6):295–300, 1993.

[BBE+83] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel. Linear size finite automata for the set of all subwords of a word: an outline of results. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 21:12–20, 1983.

[BCT93] D. Breslauer, L. Colussi, and L. Toniolo. Tight comparison bounds for the string prefix-matching problem. *Inf. Process. Lett.*, 47(1):51–57, 1993.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[CCG+94] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[CGR99] M. Crochemore, L. Gąsieniec, and W. Rytter. Constant-space string-matching in sublinear average time. *Theor. Comput. Sci.*, 218(1):197–203, 1999.

[CHPZ95] R. Cole, R. Hariharan, M. Paterson, and U. Zwick. Tighter lower bounds on the exact complexity of string matching. *SIAM J. Comput.*, 24(1):30–45, 1995.

[CL97] M. Crochemore and T. Lecroq. Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. *Inf. Process. Lett.*, 63(4):195–203, 1997.

[Col94] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. *SIAM J. Comput.*, 23(5):1075–1091, 1994.

[CP91] M. Crochemore and D. Perrin. Two-way string-matching. *J. Assoc. Comput. Mach.*, 38(3):651–675, 1991.

[Cro84] M. Crochemore. Linear searching for a square in a word. *Bull. Eur. Assoc. Theor. Comput. Sci.*, 24:66–72, 1984.

[Cro92] M. Crochemore. String-matching on ordered alphabets. *Theor. Comput. Sci.*, 92(1):33–47, 1992.

[CV97] M. Crochemore and R. Vérin. Direct construction of compact directed acyclic word graphs. In A. Apostolico and J. Hein, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, number 1264 in Lecture Notes in Computer Science, pages 116–129, Aarhus, Denmark, 1997. Springer-Verlag, Berlin.

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science*, pages 137–143, Miami Beach, FL, 1997.

[FG99] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. Assoc. Comput. Mach.*, 46(2):236–280, 1999.

[Gal79] Z. Galil. On improving the worst case running time of the Boyer-Moore string searching algorithm. *Commun. ACM*, 22(9):505–508, 1979.

[GG91] Z. Galil and R. Giancarlo. On the exact complexity of string matching: lower bounds. *SIAM J. Comput.*, 20(6):1008–1020, 1991.

[GGV04] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix array and applications. In J. I. Munro, editor, *Proceedings of the 15th ACM-SIAM Annual Symposium on Discrete Algorithms*, pages 636–645, New Orleans, LO, 2004.

[GO80] L. J. Guibas and A. M. Odlyzko. A new proof of the linearity of the Boyer-Moore string searching algorithm. *SIAM J. Comput.*, 9(4):672–682, 1980.

[Gon87] G. H. Gonnet. The PAT text searching system. Report, Department of Computer Science, University of Waterloo, Ontario, 1987.

[GPR95] L. Gąsieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: sequential sampling. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, number 937 in Lecture Notes in Computer Science, pages 78–89, Espoo, Finland, 1995. Springer-Verlag, Berlin.

[GS83] Z. Galil and J. Seiferas. Time-space optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983.

[Han93] C. Hancart. On Simon's string searching algorithm. *Inf. Process. Lett.*, 47(2):95–99, 1993.

[Han96] C. Hancart, 1996. Personal communication.

[HC02] J. Holub and M. Crochemore. On the implementation of compact dawg's. In J.-M. Champarnaud and D. Morel, editors, *Proceedings of the 7th Conference on Implementation and Application of Automata*, volume 2608 of *Lecture Notes in Computer Science*, pages 289–294, Tours, France, 2002. Springer-Verlag, Berlin.

[IHS+01] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. In A. Amir and G. M. Landau, editors, *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 169–180, Jerusalem, Israel, 2001. Springer-Verlag, Berlin.

[Irv96] R. W. Irving. Suffix binary search trees. Technical report, University of Glasgow, Computing Science Department, 1996.

[KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In R. A. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210, Morelia, Michocán, Mexico, 2003. Springer-Verlag, Berlin.

[Kär95] J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, number 937 in Lecture Notes in Computer Science, pages 191–204, Espoo, Finland, 1995. Springer-Verlag, Berlin.

[KMP77] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[KS03] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, number 2719 in Lecture Notes in Computer Science, pages 943–955, Eindhoven, The Netherlands, 2003. Springer-Verlag, Berlin.

[KSPP03] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In R. A. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 186–199, Morelia, Michocán, Mexico, 2003. Springer-Verlag, Berlin.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. Algorithms*, 23(2):262–272, 1976.

[MM93] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[MP70] J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.

[MRR01] J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001.

[Sim89] I. Simon. Sequence comparison: some theory and some practice. In M. Gross and D. Perrin, editors, *Electronic, Dictionaries and Automata in Computational Linguistics*, number 377 in Lecture Notes in Computer Science, pages 79–92. Springer-Verlag, Berlin, 1989.

[Ukk92] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–212, 1992.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[Yao79] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.