Theoretical Computer Science 92 (1992) 119–144
Elsevier

A variation on the Boyer–Moore algorithm

Thierry Lecroq
C.E.R.I.L, 25 Cours Blaise Pascal, 91000 Evry, France

1. Introduction

String-matching consists in finding all the occurrences of a word w in a text t. Several algorithms have been found for solving this problem. They are presented by Aho in a recent book [1]. Among these algorithms, the Boyer–Moore approach [5, 11] seems to lead to the fastest algorithms for the search phase. Even if the original version of the Boyer–Moore algorithm has a quadratic worst case, its behavior in practice seems to be sublinear. Furthermore, other authors [9, 2] have improved this worst-case time complexity for the search phase so that it becomes linear in the length of the text. The best bound for the number of letter comparisons is due to Apostolico and Giancarlo [2] and is $2n-m+1$, where n is the length of the text and m the length of the word. Another particularity of the Boyer–Moore algorithm is that the study of its complexity is not obvious; see [10, 7].

Basically, the Boyer–Moore algorithm tries to find for a given position in the text the longest suffix of the word which ends at that position. A new approach can possess the ability for a given position in the text to compute the length of the longest prefix of the word which ends at that position. When we know this length, we are able to compute a better shift than the Boyer–Moore approach. In the first version we make a new attempt at matching, forgetting all the previous prefixes matched. This leads to a very simple algorithm but it has a quadratic worst-case running time.

In an improved version we memorize the position where the previous longest prefix found ends and we make a new attempt at matching only the number of characters corresponding to the complement of this prefix. We are then able to compute a shift without reading again backwards more than half the characters of the prefix found in the previous attempt. This leads to a linear-time algorithm which scans the text characters at most three times each.

0304-3975/92/$05.00 © 1992—Elsevier Science Publishers B.V. All rights reserved

The computation of this longest prefix ending at a given position in the text is made possible by the use of the smallest suffix automaton of the reverse of the word w.

This strategy leads to a Boyer–Moore automaton whose number of states is bounded by the cube of the length of the word w.

This paper is organized as follows. Section 2 presents the new approach. Section 3 gives some recalls about the smallest suffix automaton of a word. Section 4 presents the computation of the longest prefix of the word ending at a given position. Section 5 gives the complexity of our first version. Section 6 presents a linear-time method. Section 7 gives the proof of the linearity of the improved method. Section 8 introduces Boyer–Moore automaton and presents the automaton based on our strategy. Section 9 compares the Boyer–Moore algorithm with ours on the basis of some examples.

2. A new approach

Notation

In this paper A is a set of letters, it is called the *alphabet*. A^* is the set of the words over A. A *word* $w \in A^*$ of length m is denoted by $w[1]w[2]...w[m]$ and $|w| = m$. $w[i]$ is the ith letter of w. ε is the empty word and $|\varepsilon| = 0$.

The Boyer–Moore algorithm attempts to match the word against the text starting from the right end of the word and progressing to the left. It consists in finding the longest suffix of the word ending at a given position in the text. When a mismatch occurs or when the whole word has been matched successfully, the Boyer–Moore algorithm computes a shift by which the word is moved to the right. And then a new attempt at matching can be made. The Boyer–Moore algorithm needs two shift functions to perform its shifts [5, 11, 12].

Our approach consists, for a given position i in the text, in being able to compute the length of the longest suffix u of the portion of the text t ending at that position i which is a prefix of the word w. We denote by $p(i)$ this value for a position i in the text (see Fig. 1).

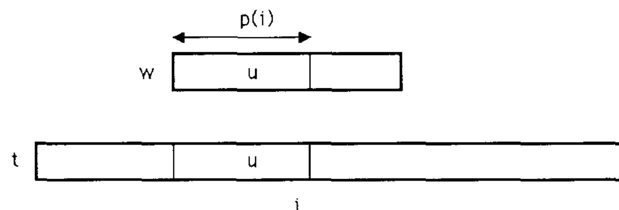

Fig. 1. $p(i)$ = the length of the longest prefix of w ending at i.

If $t = t[1] \ldots t[i] \ldots t[n]$ and $w = w[1] \ldots w[m]$ with $t[i]$'s, $w[j]$'s$\in A^*$, then for $m \leqslant i \leqslant n$

$$p(i) = \max\{j \mid 1 \leqslant j \leqslant m \text{ and (for } 1 \leqslant k \leqslant j,\ w[k] = t[i-j+k])\} \cup \{0\}.$$

We also need, for a given position i in the text, to compute the length of the longest suffix of the portion of the text ending at that position i which is a proper prefix of the word w. We denote by $p'(i)$ this value.

For $m \leqslant i \leqslant n$

$$p'(i) = \max\{j \mid 1 \leqslant j < m \text{ and (for } 1 \leqslant k \leqslant j,\ w[k] = t[i-j+k])\} \cup \{0\}.$$

Then the new algorithm of string-matching can be easily written (see Fig. 2). The procedure P computes the value of the functions p and p' for a given position i in the text.

Example 2.1. If $t = abbabbabbabbaabb$ and $w = bbabbaa$ Algorithm 1 runs as follows:

```
w  bbabbaa
t  abbabbabbabbaabb
i=7, p(i)=p'(i)=6
w   bbabbaa
t  abbabbabbabbaabb
i=8, p(i)=p'(i)=4
w      bbabbaa
t  abbabbabbabbaabb
i=11, p(i)=p'(i)=4
w         bbabbaa
t  abbabbabbabbaabb
i=14, p(i)=7, p'(i)=0
w               bbabbaa
t  abbabbabbabbaabb
```

```
Input :    A text t and a word w.
Output :   All the locations of the occurrences of w in t.
Method :
    Begin
        n := |t|; m := |w|; i := m;
        while i ≤ n do {
            (j, k) := P(i);
            if j = m then {
                output(one occurrence of w found at i-m+1);
                j := k; }
            i := i+m-j; }
    End.
```

Fig. 2. Algorithm 1, the first version.

The underlined characters are the characters read at each attempt.

The procedure P can be easily computed by using the smallest suffix automaton of the reverse of the word w.

3. The smallest suffix automaton

The smallest suffix automaton recognizing all the suffixes of a word w is a deterministic finite automaton denoted by the 5-uple

$$\mathscr{A} = \{A, S, s, F, \delta\},$$

where

- A is the alphabet,
- S is the set of states,
- $s \in S$ is the initial state,
- $F \subseteq S$ is the set of the final states, and
- $\delta : S \times A \rightarrow S$ is the transition function.

The language accepted by $\mathscr{A}$ is: $L(\mathscr{A}) = \{x \in A^* \mid \exists u \in A^* \text{ and } ux = w\}$. Its construction is linear in time and space in the length of the word w [4, 8].

Example 3.1. The smallest suffix automaton for $w = aabbabb$ is:

$$A = \{a, b\},\ S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\},\ s = 0,\ F = \{0, 5, 9, 10\};$$

for δ see Fig. 3.

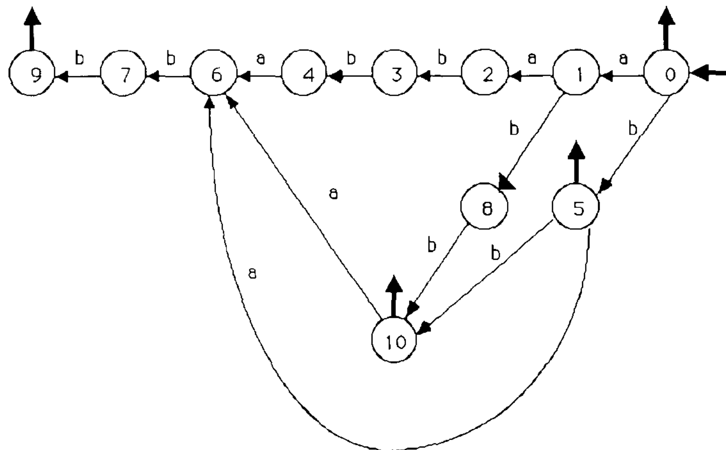

Fig. 3. The smallest suffix automaton for $w = aabbabb$.

The automaton is represented from right to left because our method scans the portions of the text with the help of the automaton from right to left.

4. The computation of procedure P

In order to compute the value of $p(i)$ and $p'(i)$ we scan the text, from right to left starting at position i, with the help of the automaton of the suffixes of the reverse of w. Each time a final state is met we have recognized a prefix of the word w. Then

- $p(i)$ is equal to the length of the path taken from the initial state to the last final state met, and
- if $p(i) < m$, then $p'(i)$ is equal to $p(i)$, otherwise $p'(i)$ is equal to the length of the path taken from the initial state to the last but one final state met.

See Algorithm 2 in Fig. 4 for the details.

Remark. As the value of $p'(i)$ is needed only when $p(i) = m$; in practice, for a more efficient implementation we will omit the last test.

In this paper we make the following assumption: all the transitions of the automaton can be computed in constant time, which is a reasonable assumption for a finite alphabet.

The **while** loop of Algorithm 2 runs at most m times and all the other instructions are in time $O(1)$; thus, the time complexity of this algorithm is obviously $O(m)$.

5. Time complexity of Algorithm 1

Theorem 5.1. *Algorithm* 1 *has a worst-case time complexity in* $O(mn)$.

Proof. The number of times the **while** loop of Algorithm 1 runs depends on the variable i. At each step this variable is affected by the value $i+m-j$, as when j is equal

```
Input :    A text t, a word w, an index i of the text and the
           smallest suffix automaton for wR : A = { A, S, s, F, δ }.
Output :   (value of p(i), value of p'(i))
Method :
     Begin
           m := |w|; state := s; p := 0; j := i;
           while i-j < m and δ(state, t[j]) is defined do {
                 state := δ(state, t[j]); j := j-1;
                 if state ∈ F then {
                       p' := p; p := i-j; }}
           if p = m then return(p, p')
           else return(p, p);
     End.
```

Fig. 4. Algorithm 2, computation of $P(i)$.

to m, j is affected by the value $k<j\leqslant m$. As a consequence, i is always incremented. The variable i can take all the values from m to n. All the instructions of the **while** loop are in $O(1)$ except the procedure P which runs in time $O(m)$ and all the other instructions of Algorithm 1 are in $O(1)$. As a consequence, the worst-case time complexity of Algorithm 1 is $O(mn)$ (with the assumption on the transitions of the automaton). □

6. A linear-time method

If w is a word of length m, then we have the following definition.

Definition. An integer p such that $1\leqslant p\leqslant m$ is a *period* of the word w if for $1\leqslant i\leqslant m-p$ $w[i]=w[i+p]$. A word x is a *border* of the word w if x is a prefix and a suffix of w: $\exists u, v\in A^*$ and $w=xu=vx$.

When we have found the longest prefix u of the word w ending at location i in the text t, then we consider the portion v of the text composed by the $|w|-|u|$ characters on the right of u in t (see Fig. 5). In several cases, even if we have been able to read all the characters of v without being stopped in the automaton, it is not necessary to read backwards the characters of u. And in all cases we do not have to scan backwards more than the right half of u.

In order to improve our first string-matching algorithm we introduce a function T which has the following definition:

$$\begin{aligned}
&T(k,g,\text{state})=(p,c,h,\text{state}') \text{ for } m\leqslant k\leqslant n,\ 0\leqslant g\leqslant m,\ \text{state}\in F,\\
&\quad p=\max\{j \mid 1\leqslant j\leqslant g \text{ and } \delta(\text{state}, t[k]\ldots t[k-j+1])\in F\}\cup\{0\},\\
&\quad c=\max\{j \mid 1\leqslant j\leqslant g \text{ and } \delta(\text{state}, t[k]\ldots t[k-j+1] \text{ is defined}\}\cup\{0\},\\
&\quad h=\begin{cases}\text{sh}(\text{state}, t[k]\ldots t[k-c+1]) & \text{if } c>0,\\ 0 & \text{otherwise,}\end{cases}\\
&\quad \text{state}'=\begin{cases}\delta(\text{state}, t[k]\ldots t[k-c+1]) & \text{if } c>0,\\ \text{state} & \text{otherwise.}\end{cases}
\end{aligned}$$

The shift function sh is defined as follows:

$$\text{sh}(q,a)=\{|x| \mid x\in A^*,\ uvxa \text{ prefix of } w^R,\ u,\ v\in A^*,\ |u| \text{ minimal and } q=\delta(s,v)\}$$

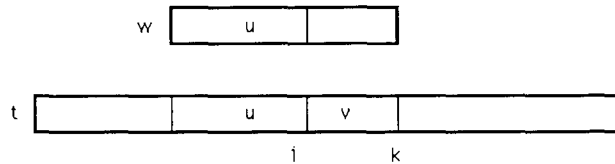

Fig. 5.

for $q \in S$ and $a \in A$ and is defined where δ is defined. Its construction can be computed during the construction of the automaton without changing its time complexity.

Assume then that we are at position i in the text and that we have found the longest prefix u of the word w which ends at i:

$$|t| = n, \qquad |w| = m, \qquad |u| = m - g, \qquad |v| = g.$$

Assume further that we know the length of the smallest period of u: per(u). Then we compute $T(i+g, g, s)$ which corresponds to scanning the g characters of v from right to left starting with the initial state in the automaton:

$$T(i+g, g, s) = (p, c, h, \text{state})$$

Then several cases arise:
- We have not been able to read all the g characters of v.
- We have found an occurrence of w.
- The shift of v in w is a multiple of per(u).
- per(u) is large or per(u) is small.

Case 1: $c \neq g$; it means that all the characters of v have not been read (see Fig. 6).

Lemma 6.1. *If $c \neq g$, we know the longest prefix (of length p) of the word w which ends at $i+g$ and we can make a new attempt with $T(i+g+m-p, m-p, s)$ without missing any occurrence of w.*

Proof. $c = \max\{j \mid 1 \leqslant j \leqslant g$ and $\delta(\text{state}, t[i] \ldots t[i-j+1]$ is defined$\}$ or 0, and $c \neq g$, so $c < g \leqslant m$. No prefix of w longer than c can occur ending at $i+g$ in the text:

$$p = \max\{j \mid 1 \leqslant j \leqslant g \text{ and } \delta(\text{state}, t[i] \ldots t[i-j+1]) \in F\} \cup \{0\},$$

where p is the length of the longest prefix of w ending at $i+g$ in the text. □

Example 6.2.

```
w      bbabbaa
t   cccbbabcbb . . .
u      bbab
w            bbabbaa
```

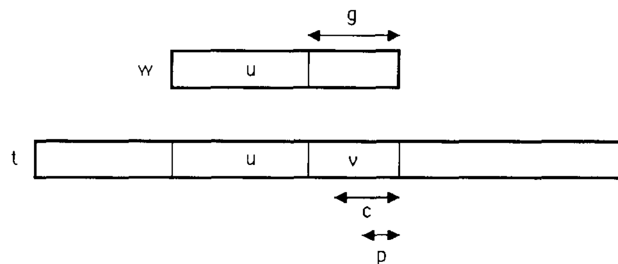

Fig. 6. Case 1, $c \neq g$.

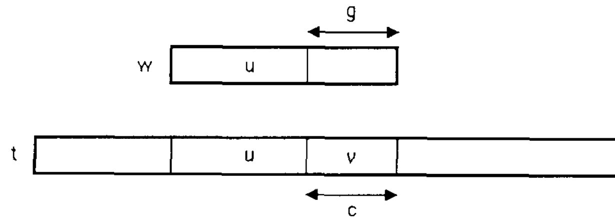

Fig. 7. Case 2, $c=g$, $h=0$.

Case 2: $c=g$ and $h=0$ (see Fig. 7).

Lemma 6.3. *If $c=g$ and $h=0$, we have found an occurrence of the word in the text ending at $i+g$.*

Proof. $c=g$, so $\delta(s, t[i+g]...t[i+1])$ is defined and $h=\text{sh}(s, t[i+g]...t[i+1])=0$, which means that $t[i+g]...t[i+1]$ is a prefix of w^R, so $t[i+1]...t[i+g]$ is a suffix of length g of w. As u is a prefix of length $m-g$ of w, $ut[i+1]...t[i+g]$ is exactly w. □

Example 6.4.

w	*bbabbab*
t	*cccbbabbab*...
u	*bbab*

Case 3: $c=g$, $h>0$ and h is a multiple of per(u) (see Fig. 8).

Lemma 6.5. *If $c=g$, $h>0$ and h is a multiple of per(u), we know a new longest prefix of length $m-h$ of w ending at $i+g$ and we can make a new attempt with $T(i+g+h, h, s)$ without missing any occurrence of w.*

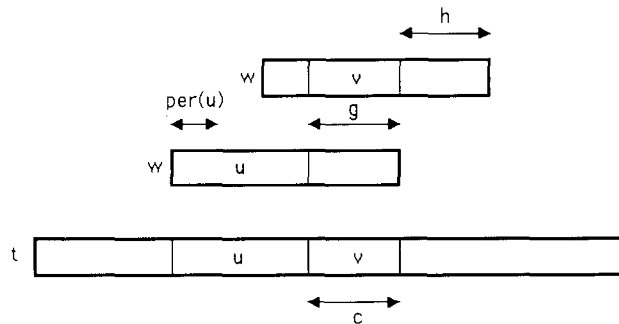

Fig. 8. Case 3, $c=g$, $h>0$ and h is a multiple of per(u).

Proof. $u = u_1^r u_2$ with $|u_1| = \text{per}(u)$ and $|u_2| < |u_1|$ and u_2 prefix of u_1. $w = uu'$, $v = t[i+1] \ldots t[i+g]$ then $w = v'vv''$ with $|v''| = h$. $|u'| = |v|$, so $|u'| < |vv''|$ since $|v''| = h > 0$. So v' is a prefix of u and as $|u'| = |v|$ and h is a multiple of $\text{per}(u)$, then $v' = u_1^{r'} u_2$ with $r' < r$. So $w = u_1^{r'} u_2 vv''$. If $t = xuvx'$, then $t = xu_1^r u_2 vx'$ and $t = x'' u_1^r u_2 vx'$ with $u_1^r u_2 v$ as the longest prefix of w ending at $i+g$ in t, and $|u_1^r u_2 v| = m - h$. □

Example 6.6.

```
w         abcabcabcabxxxxxxx
t   ccccccabcabcabcabcabcab.........
u         abcabcabcab
w                abcabcabcabxxxxxxx
```

Case 4: $c = g$, $h > 0$, h is not a multiple of $\text{per}(u)$ and $\text{per}(u)$ is large (greater than half the length of u: $\text{per}(u) > |u|/2 = (m-g)/2$), then we make a new call with $T(i, |u|\text{-per}(u), \text{state}) = (p', c', h', \text{state}')$ and then two cases arise.
Case 4.1: $p' = 0$ (see Fig. 9 and Example 6.8).
Case 4.2: $p' > 0$ (see Fig. 10 and Example 6.9).

Lemma 6.7. *If $p' = 0$, we know the longest prefix (of length p) of the word w which ends at $i+g$ and we can make a new attempt with $T(i+g+m-p, m-p, s)$ without missing any occurrence of w. If $p' > 0$, we know the longest prefix (of length $g+p'$) of the word w which ends at $i+g$ and we can make a new attempt with $T(i+m-p', m-g-p', s)$ without missing any occurrence of w.*

Proof. It is obvious that w cannot recur between $i - |u|$ and $i - \text{per}(u)$ by the minimality of $\text{per}(u)$, then it enables us to ignore this portion of the text. We just have to find the longest prefix of w ending at $i+g$ and starting from $i - \text{per}(u)$. This is done by the two calls to the function T: $T(i+g, g, s)$ and $T(i, m - \text{per}(u), \text{state})$.

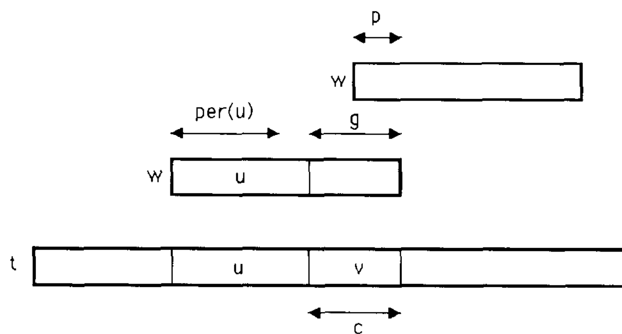

Fig. 9. Case 4.1, $c = g$, h is a multiple of $\text{per}(u)$, $\text{per}(u)$ is large and $p' = 0$.

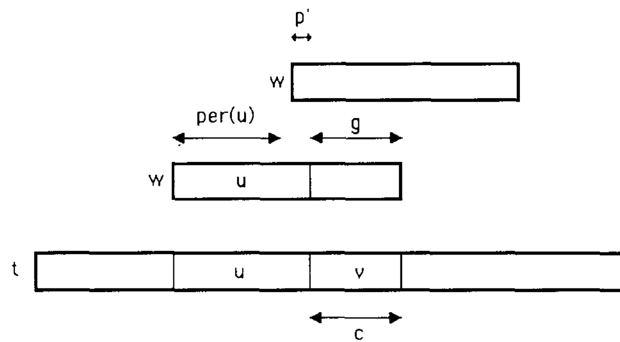

Fig. 10. Case 4.2, $c=g$, h is a multiple of), per(u) is large and $p'>0$.

After that if $p'=0$, it means that there is no prefix of w starting in the portion of the text $t[i-m+\text{per}(u)+1]\ldots t[i]$, so the longest prefix of the word w ending at $i+g$ in the text is the prefix of length p found by the first call to the function T.

If $p'>0$, then $p'=\max\{j \mid 0\leqslant j\leqslant m-\text{per}(u)$ and $\delta(\text{state}, t[i]\ldots t[i-j+1])\in F\}$ and the length of the path taken from s to state is of course equal to g, so the length of the longest prefix of the word w ending at $i+g$ in the text is $p'+g$. □

Example 6.8.

```
w      abbababbcac
t   cccabbababbbca.........
u      abbababb
w                 abbababbcac
```

Example 6.9.

```
w    aaabaaaba
t   ccaaabaaaab......
u    aaabaaa
w          aaabaaaba
```

Case 5: $c=g$, $h>0$, h is not a multiple of per(u) and per(u) is small (less than or equal to half the length of u: per$(u)\leqslant|u|/2$). Then we make a new call with $T(i, \text{per}(u), \text{state})=(p', c', h', \text{state}')$. Then two cases arise.
Case 5.1: $c'=\text{per}(u)$ (see Fig. 11).

Lemma 6.10. *If $c'=\text{per}(u)$, we know the longest prefix (of length $m-h-h'$) of the word w which ends at $i+g$ and we can make a new attempt with $P(i+g+h+h', h+h', s)$.*

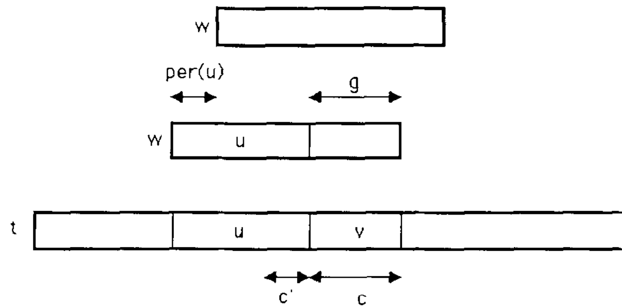

Fig. 11. Case 5.1, $c=g$, h is not a multiple of per(u), per(u) is small and $c'=$per(u).

Proof. $u=u_1^r\,u_2$ with $|u_1|=\text{per}(u)$ and $|u_2|<|u_1|$ and u_2 a prefix of u_1:

$$v=t[i+1]\dots t[i+g],$$

$$t=xuvx',$$

$$u_1=u_2u_3,$$

$$u=(u_2u_3)^r u_2,$$

$$t=x(u_2u_3)^r u_2 v x',$$

$$w=v'u_3u_2vv'' \quad \text{with} \quad |v''|=h+h',$$

$$w=uu',$$

$$|u'|=|v|,\ |v''|\geqslant 0.$$

Then $v'u_3u_2$ a prefix of $u=(u_2u_3)^r u_2$ and $|u_2u_3|=\text{per}(u)$. By the minimality of per(u) the factor u_3u_2 of v' must exactly match a factor u_3u_2 of u, so $v'=(u_2u_3)^{r'}u_2$ with $r'<r$. $v'u_3u_2v$ is the longest prefix of w ending at $i+g$ and its length is equal to $m-h-h'$. □

Example 6.11.

w	bbabbabbabaa
t	ccbbabbabbabba . . . .
u	bbabbabbab
w	bbabbabbabaa

Case 5.2: $c'<\text{per}(u)$, then this is similar to the case 4. The new attempt depends on the value of p'.

Then to write the algorithm we just have to know how to make the new attempt in case 2 and how to compute the length of the smallest period of u at each step.

Lemma 6.12. *If $c=g$ and $h=0$, the length of the shift is the length of the smallest period of the word w which is denoted by $per(w)$, so the next call will be $T(i+g+\text{per}(w), \text{per}(w), s)$.*

The proof is obvious.

In order to compute the length of the smallest period of the longest prefix of the word w we have already found we use the function f of Morris and Pratt defined as follows. For $0<i\leqslant m$

$$f(i)=\text{length of the longest border of } w[1]\ldots w[i] \text{ and } f(0)=0.$$

Figure 12 describes the whole algorithm.

Theorem 6.13. *Algorithm 3 finds all the occurrences of the word w in the text t.*

Proof. All the situations are described in cases 1–5. Lemmas 6.1, 6.3, 6.5, 6.7, 6.10 and 6.12 give the proof that Algorithm 3 finds all the occurrences of w in t. □

```
            Input :   A text t and a word w.
            Output :  All the locations of the occurrences of w in t.
            Method :
                Begin
                    n := |t|; m := |w|; pu := 0; i := m; g := m;
                    while i ≤ n do {
                        (p, c, h, state) := T(i, g, s);
CASE 1                  if c < g then {
                            g := m-p; pu := p-f(p); }
CASE 2                  else if h = 0 then {
                            output(one occurrence of w found at i-m);
                            g := per(w);
                            pu := m-per(w)-f(m-per(w)); }
CASE 3                  else if h is a multiple of pu then {
                            g := h; pu := m-h-f(m-h); }
CASE 4                  else if pu > (m-g)/2 then {
                            (p', c', h', state') := T(i-g, m-g-pu, state);
CASE 4.1                    if p' = 0 then {
                                g := m-p; pu := p-f(p); }
CASE 4.2                    else {
                                g := m-g-p'; pu := g+p' - f(g+p'); }}
CASE 5                  else {
                            (p', c', h', state') := T(i-g, pu, state);
CASE 5.1                    if c' = pu then {
                                g := h+h'; pu := m-h-h'-f(m-h-h'); }
CASE 5.2                    else
                                if p' = 0 then {
                                    g := m-p; pu := p-f(p); }
                                else {
                                    g := m-g-p'; pu := g+p'-f(g+p'); }}
                        i := i+g; }
                End.
```

Fig. 12. Algorithm 3.

```
Input :   A text t, a word w, an index k, a length g and a state
          state of the text, the smallest suffix automaton for wR :
          A = { A, S, s, F, δ } and the shift function sh.
Output :  (p, c, h, state')
Method :
   Begin
        p := 0; h := 0; j := k;
        while k-j < g and δ(state, t[j]) is defined do {
              h := h + sh(state, t[j]);
              state := δ(state, t[j]);
              j := j-1;
              if state ∈ F then p := k-j; }
        return(p, k-j, h, state);
   End.
```

Fig. 13. Algorithm 4, computation of $T(k, g, \text{state})$.

The procedure T can be computed as shown in Fig. 13.

7. Time complexity of Algorithm 3

Assume that we have recognized the longest prefix u_0 ending at a position I in the text. Let us denote by $u_1, u_2, \ldots, u_i, \ldots$ the longest prefixes recognized in the next attempts. Then the v_i's will be the words composed by the $m - |u_i|$ characters on the right of the u_i's in the text.

We define the shifts d_i's by $d_i = |v_{i+1}|$.

Let us denote by k_i the positions of the first character of the u_i's in the text (see Fig. 14). The k_i's describe a strictly increasing sequence: $k_i < k_{i+1}$ $\forall i > 0$. We assume that the characters of u_0 have been read only once before the attempt to scan v_0.

From the description of Algorithm 3 it is obvious that it is not possible to read backwards more than the right half of u_i while scanning v_i.

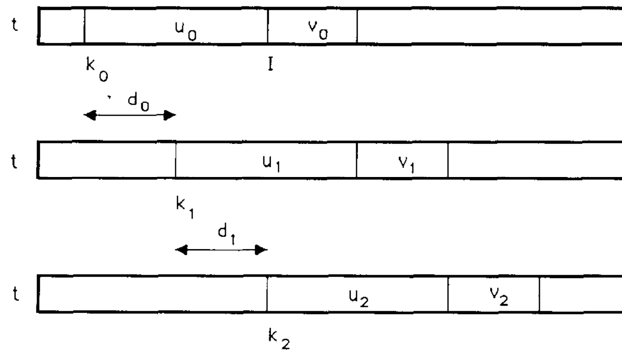

Fig. 14. Three consecutive attempts.

Proposition 7.1. *The locations of the middles of the u_i's describe a strictly increasing sequence.*

Proof. This is obvious if the u_i's do not overlap. Let us examine the case where they all overlap.

$$x_i = |u_i|,$$

$$y_i = |v_i|,$$

$$m_i = k_i + |u_i|/2 = k_i + x_i/2 \quad \text{(position of the middle of } u_i\text{)},$$

$$x_{i+1} = a_i x_i + y_i \quad \text{with} \quad 0 < a_i < 1,$$

$$y_{i+1} = (1 - a_i) x_i,$$

$$k_{i+1} = k_i + y_{i+1}$$
$$= k_i + (1 - a_i) x_i,$$

$$m_{i+1} = k_{i+1} + x_{i+1}/2$$
$$= k_i + (2 - a_i) x_i/2 + y_i/2,$$

$$a_i < 1 \Rightarrow m_i < m_{i+1}. \quad \square$$

Lemma 7.2. *During the next attempts we read again at most twice the characters on the right half of u_0 and never the characters on the left half of u_0.*

Proof. The fact that the left half of u_0 is never read again follows from Proposition 7.1. It remains to consider the right half of u_0. The proof is divided into two parts. Part 1 for the case where the smallest period of u_0 is small and part 2 for the case where the smallest period of u_0 is large.
Part 1: The smallest period of u_0 is small: $u_0 = x_1^r x_2$ with $r > 1$, $|x_1| = \text{per}(u_0)$, $|x_2| < |x_1|$ and x_2 prefix of x_1:

$$x_1 = x_2 x_3,$$

$$u_0 = (x_2 x_3)^r x_2,$$

$$w = u_0 u'.$$

If $|v_0| \leqslant \text{per}(u_0)$ (see Fig. 15) and if we are able to read v_0 without being stopped in the automaton and if we decide to read backwards the characters on the left of v_0 in the text, it means that $w = v' v_0 v''$ and $|v''|$ is not a multiple of $\text{per}(u_0)$; thus, $|v''| > 0$. $|v_0| = |u'|$, then $|v_0 v''| > |u'|$. $|v'| < |u_0|$ and v' and u_0 are prefixes of w, then $v' = x_1^p x_4$ with $p \leqslant r$, $|x_4| < |x_1|$, x_4 a prefix of x_1, and $x_4 \neq x_2$. $w = (x_2 x_3)^p x_4 v_0 v''$.

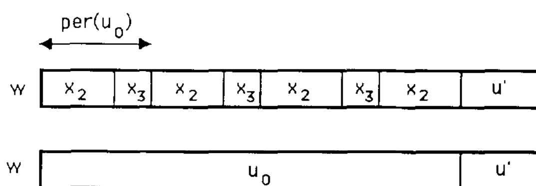

Fig. 15. $u_0=(x_2x_3)^r x_2$ and $|v_0|\leqslant \mathrm{per}(u_0)$.

If we are able to read backwards $\mathrm{per}(u_0)$ characters on the left of v_0 in the text without being stopped in the automaton, we have read x_3x_2 (the $\mathrm{per}(u_0)$ last characters of u_0) which cannot be equal to the factor x_3x_4 just on the left of v_0 in w (by the minimality of $|x_2x_3|=\mathrm{per}(u_0)$ and $x_4\neq x_2$). Then there exists another occurrence of v_0 further on left in w.

We know that the prefix x_3x_2 of $x_3x_2v_0$ must exactly match with the factor x_3x_2 of u_0 and, by the fact that $|v_0|\leqslant \mathrm{per}(u_0)$, we know that v_0 is a prefix of x_3x_2. Then the length of the shift will be exactly $\mathrm{per}(u_0)$ and the smallest period of u_1 is equal to the smallest period of u_0:

$$d_0=\mathrm{per}(u_0) \quad \text{and} \quad \mathrm{per}(u_1)=\mathrm{per}(u_0).$$

We can have the same argument with u_1 and v_1 since $\mathrm{per}(u_1)=\mathrm{per}(u_0)$ and $|v_1|=\mathrm{per}(u_0)\leqslant \mathrm{per}(u_1)$.

So, if we are able to read backwards $\mathrm{per}(u_0)$ characters on the left of v_1 in the text without being stopped in the automaton, then the length of the shift will be equal to $\mathrm{per}(u_0)$ and $|v_2|=\mathrm{per}(u_0)$ and $\mathrm{per}(u_2)=\mathrm{per}(u_1)=\mathrm{per}(u_0)$.

So, if we read backwards $\mathrm{per}(u_0)$ characters on the left of v_2 in the text (which is the maximum), then we read the characters of v_1 and none of u_0 (see Fig. 16).

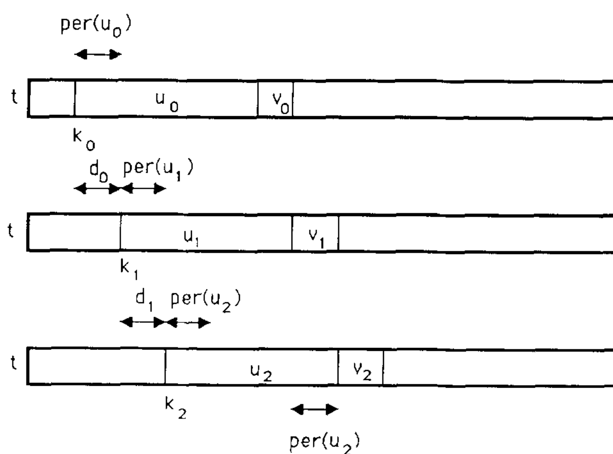

Fig. 16. $\mathrm{per}(u_0)=\mathrm{per}(u_1)=\mathrm{per}(u_2)$.

Then the characters of the second half of u_0 have been read at most three times.

Now we will examine the situations where we are stopped in the automaton while scanning characters on the left of v_2, v_1 and v_0.

If during the rescanning of the characters on the left of v_2 we are stopped in the automaton, then the length of the shift will be at least equal to $(r-1)\text{per}(u_0)$ (since w cannot reappear before):

$$d_2=(r-1)\text{per}(u_0)+d \quad \text{with} \quad d \geqslant 0,$$

$$k_1=k_0+d_0,$$

$$k_1=k_0+\text{per}(u_0),$$

$$k_2=k_1+d_1$$

$$=k_0+2\text{per}(u_0),$$

$$k_3=k_2+d_2$$

$$=k_0+(r+1)\text{per}(u_0)+d,$$

$$k_3+|u_3|/2>k_0+|u_0|$$

$$=k_0+r\,\text{per}(u_0)+|x_2|,$$

as $|x_2|<\text{per}(u_0)$. Then, as it is impossible to read backwards more than half the characters of u_3, we cannot read backwards the characters of u_0 (see Fig. 17).

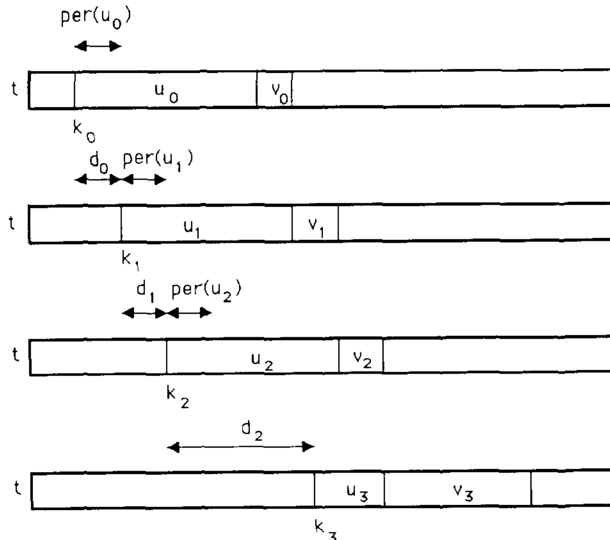

Fig. 17. $d_2=(r-1)\text{per}(u_0)+d$.

If during the rescanning of the characters on the left of v_1 we are stopped in the automaton, then the length of the shift will be at least equal to $(r-1)\,\mathrm{per}(u_0)$ (since w cannot reappear before):

$$d_1 = (r-1)\,\mathrm{per}(u_0) + d \quad \text{with } d \geqslant 0,$$

$$k_1 = k_0 + d_0,$$

$$k_1 = k_0 + \mathrm{per}(u_0),$$

$$k_2 = k_1 + d_1$$

$$= k_0 + r\,\mathrm{per}(u_0) + d,$$

$$|u_2| = |u_1| - d_1 + |v_1|$$

$$= |x_2| + |v_0| - d + \mathrm{per}(u_0),$$

$$k_2 + |u_2|/2 = k_0 + r\,\mathrm{per}(u_0) + d + |x_2|/2 + |v_0|/2 - 1/2 + \mathrm{per}(u_0)/2$$

$$= k_0 + (r + 1/2)\,\mathrm{per}(u_0) + |x_2|/2 + |v_0|/2 + 1/2$$

$$> k_0 + |u_0|$$

$$= k_0 + r\,\mathrm{per}(u_0) + |x_2|,$$

as $|x_2| < \mathrm{per}(u_0)$. Then, as it is impossible to read backwards more than half the characters of u_2, we cannot read backwards the characters of u_0 (see Fig. 18).

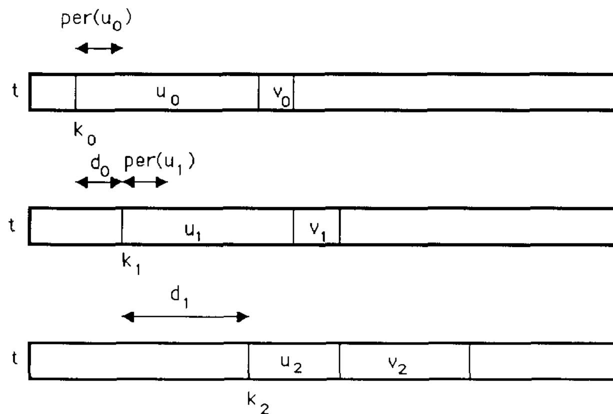

Fig. 18. $d_1 = (r-1)\,\mathrm{per}(u_0) + d$.

If during the rescanning of the characters on the left of v_0 we are stopped in the automaton, then the length of the shift will be at least equal to $r\,\mathrm{per}(u_0)$ (since w cannot reappear before):

$$d_0 = r\,\mathrm{per}(u_0) + d \quad \text{with } d \geqslant 0,$$

$$k_1 = k_0 + r\,\mathrm{per}(u_0) + 1,$$

$$|u_1| = |x_2| + |v_0| - d.$$

Then the value of the next shift is at least equal to 1:

$$d_1 > 0,$$

$$\begin{aligned} k_2 &= k_1 + d_1 \\ &= k_0 + r\,\mathrm{per}(u_0) + d + d_1, \end{aligned}$$

$$\begin{aligned} |u_2| &= |u_1| - d_1 + |v_1| \\ &= |x_2| + |v_0| - d_1 + r\,\mathrm{per}(u_0), \end{aligned}$$

$$\begin{aligned} k_2 + |u_2|/2 &= k_0 + r\mathrm{per}(u_0) + d + d_1 + |x_2|/2 + |v_0|/2 - d_1/2 + (r/2)\mathrm{per}(u_0) \\ &= k_0 + (3r/2)\mathrm{per}(u_0) + |x_2|/2 + |v_0|/2 + d + d_1/2, \\ &> k_0 + |u_0| \\ &= k_0 + r\,\mathrm{per}(u_0) + |x_2|, \end{aligned}$$

as $r > 1$ and $|x_2| < \mathrm{per}(u_0)$. Then, as it is impossible to read backwards more than half the characters of u_2, we cannot read backwards the characters of u_0 (see Fig. 19).

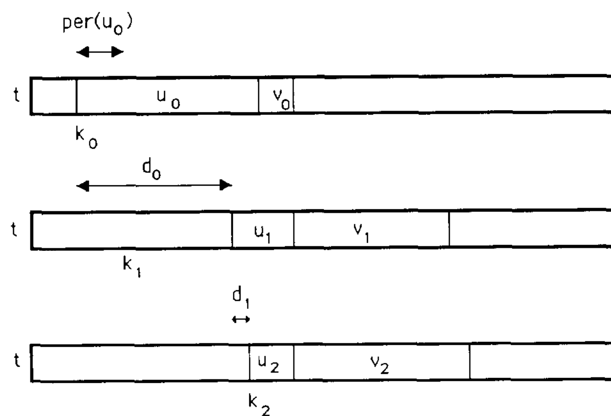

Fig. 19. $d_0 = r\,\mathrm{per}(u_0) + d$.

If $|v_0| > \mathrm{per}(u_0)$,

$$w = (x_2 x_3)^r x_2 u' \quad \text{with} \quad |u'| = |v_0| > \mathrm{per}(u_0).$$

If we are able to read v_0 without being stopped in the automaton and if we decide to read backwards the characters on the left of v_0 in the text, it means that $w = v' v_0 v''$ and $|v''|$ is not a multiple of $\mathrm{per}(u_0)$; thus, $|v''| > 0$.

Then assume without loss of generality that we have $x_3 = x'_3 x''_3$ such that $v_0 = (x''_3 x_2 x'_3)^s x''_3 x_2 v'_0$ with v'_0 a prefix of u'. $w = v'(x''_3 x_2 x'_3)^s x''_3 x_2 v'_0 v''$.

If we are able to read $\mathrm{per}(u_0)$ characters on the left of v_0 in the text without being stopped in the automaton, then we know that $x_3 x_2 v_0$ is a factor of w. The factor $x_3 x_2$ of $x_3 x_2 v_0$ must exactly match a factor $x_3 x_2$ of u_0 (by the minimality of $\mathrm{per}(u_0) = |x_3 x_2|$ and by the fact that $|v''| > 0$). Since $|v_0| > \mathrm{per}(u_0)$, then $x_3 x_2$ is a prefix of v_0: $v_0 = x_3 x_2 v''_0$.

Then we have $v_0 = (x''_3 x_2 x'_3)^s x''_3 x_2 v'_0$ (see Fig. 20) and $v_0 = x_3 x_2 v''_0 = x'_3 x''_3 x_2 v''_0$ (see Fig. 21).

So, we would have $x''_3 x_2 x'_3 = x'_3 x''_3 x_2$. As $|x''_3 x_2 x'_3| = |x'_3 x''_3 x_2| = \mathrm{per}(u_0)$, it is impossible to have a proper cyclic shift of a portion of the length $\mathrm{per}(u_0)$ of u_0. Thus, if

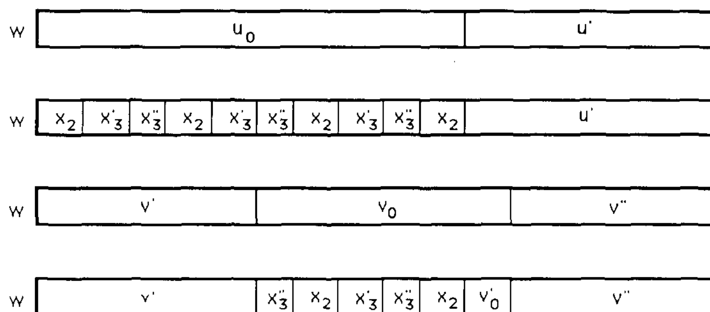

Fig. 20. $v_0 = x''_3 x_2 x'_3 x''_3 v'_0$.

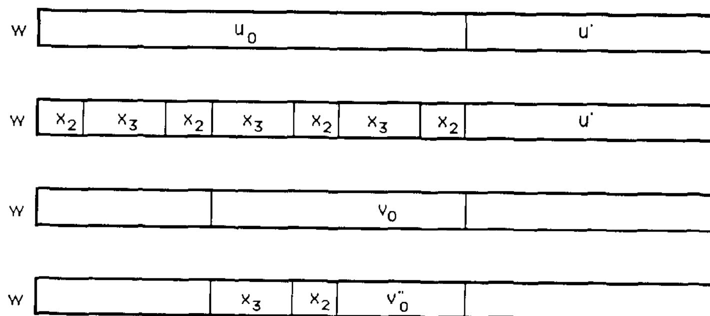

Fig. 21. $v_0 = x_3 x_2 v''_0$.

$|v_0| > \mathrm{per}(u_0)$, it is impossible to read backwards $\mathrm{per}(u_0)$ characters of u_0 without being stopped in the automaton. Then the length of the shift will be at least equal to $r\,\mathrm{per}(u_0)$:

$$d_0 = r\,\mathrm{per}(u_0) + d \quad \text{with} \quad d \geqslant 0,$$

$$k_1 = k_0 + r\,\mathrm{per}(u_0) + d,$$

$$|u_1| = |x_2| + |v_0| - d.$$

Then the value of the next shift is at least equal to 1:

$$d_1 > 0,$$

$$\begin{aligned} k_2 &= k_1 + d_1 \\ &= k_0 + r\,\mathrm{per}(u_0) + d + d_1, \end{aligned}$$

$$\begin{aligned} |u_2| &= |u_1| - d_1 + |v_1| \\ &= |x_2| + |v_0| - d_1 + r\,\mathrm{per}(u_0), \end{aligned}$$

$$\begin{aligned} k_2 + |u_2|/2 &= k_0 + r\,\mathrm{per}(u_0) + d + d_1 + |x_2|/2 + |v_0|/2 - d_1/2 \\ &\qquad + (r/2)\,\mathrm{per}(u_0) \\ &= k_0 + (3r/2)\,\mathrm{per}(u_0) + |x_2|/2 + |v_0|/2 + d + d_1/2 \\ &> k_0 + |u_0| \\ &= k_0 + r\,\mathrm{per}(u_0) + |x_2|, \end{aligned}$$

as $r > 1$ and $|x_2| < \mathrm{per}(u_0)$. Then, as it is impossible to read backwards more than half the characters of u_2, we cannot read backwards the characters of u_0.

Part 2: The smallest period of u_0 is large: $u_0 = xy$ with $|x| = \mathrm{per}(u_0)$, $|y| < |x|$ and y a prefix of x.

If $|v_0| \leqslant \mathrm{per}(u_0)$, the length of the shift is at least equal to $\mathrm{per}(u_0)$:

$$d_0 = \mathrm{per}(u_0) + d \quad \text{with} \quad d \geqslant 0,$$

$$k_1 = k_0 + \mathrm{per}(u_0) + 1,$$

$$|u_1| = |y| + |v_0| - 1.$$

Then the value of the next shift is at least equal to 1:

$$d_1 > 0,$$

$$\begin{aligned} k_2 &= k_1 + d_1 \\ &= k_0 + \mathrm{per}(u_0) + d + d_1, \end{aligned}$$

$$\begin{aligned} |u_2| &= |u_1| - d_1 + |v_1| \\ &= |y| + |v_0| - d_1 + \mathrm{per}(u_0), \end{aligned}$$

$$\begin{aligned}k_2+|u_2|/2&=k_0+\mathrm{per}(u_0)+d+d_1+|y|/2+|v_0|/2-d_1/2+\mathrm{per}(u_0)/2\\&=k_0+3\mathrm{per}(u_0)/2+|x_2|/2+|v_0|/2+d+d_1/2\\&>k_0+|u_0|\\&=k_0+r\,\mathrm{per}(u_0)+|y|,\end{aligned}$$

as $|y|<\mathrm{per}(u_0)$. Then, as it is impossible to read backwards more than half the characters of u_2, we cannot read backwards the characters of u_0 (see Fig. 22).

If $|v_0|>\mathrm{per}(u_0)$, $w=xyu'$. If we scan the characters of v_0 from right to left without being stopped in the automaton, then v_0 is a factor of w: $w=v'v_0v''$.

If we decide to read backwards the characters of u_0 (on the left of v_0 in the text), it means that $|v''|$ is not a multiple of $\mathrm{per}(u_0)$, so $|v''|>0$:

$$x=yy',$$

$$u_0=yy'y,$$

$$w=yy'yu'.$$

If we are able to read $\mathrm{per}(u_0)$ characters on the left of v_0 in the text without being stopped in the automaton, then we know that $y'yv_0$ is a factor of w. The factor $y'y$ of $y'yv_0$ must exactly match the factor $y'y$ of u_0 (by the minimality of $\mathrm{per}(u_0)=|y'y|$). But this means that $|v''|=0$, which is a contradiction.

So, if $|v_0|>\mathrm{per}(u_0)$, it is impossible to read backwards $\mathrm{per}(u_0)$ characters of u_0 without being stopped in the automaton. Then the length of the shift will be at least equal to $\mathrm{per}(u_0)$ as when $|v_0|\leqslant\mathrm{per}(u_0)$ and we know that in this case it is impossible to read backwards the characters of u_0.

This ends the proof of Lemma 7.2. □

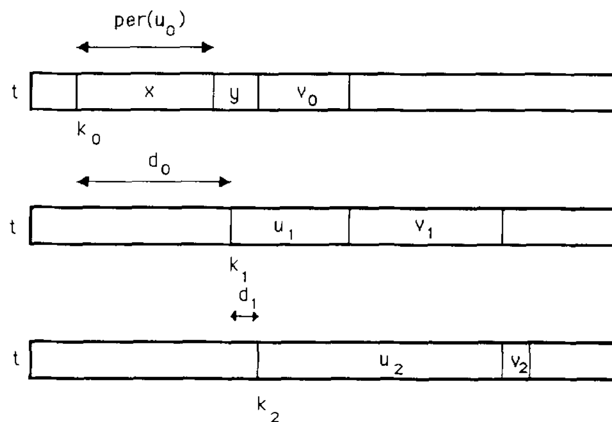

Fig. 22. $u_0=xy$ and $|v_0|\leqslant|x|$.

Theorem 7.3. *Algorithm* 3 *reads at most three times the characters of the text.*

Proof. The result follows directly from Lemma 7.2. □

8. The Boyer–Moore automaton

The Boyer–Moore automaton was introduced in [11]. It is a way to keep track of the characters already matched for the last m current characters of the text. This leads to a deterministic finite automaton (A, Q, q_0, d) associated with a shift function s, where
- A is the alphabet,
- Q is the set of states; a state $q \in Q$ is a word of length m and for $1 \leqslant i \leqslant m$ $q[i] = w[i]$ or $q[i] = \$$, where $\$ \notin A$,
- $q_0 \in A$ is the initial state (for $1 \leqslant i \leqslant m$ $q_0[i] = \$$),
- $d: Q \times A \rightarrow Q$ is the transition function,
- $s: Q \times A \rightarrow \{0, \ldots, 2m\}$.

Actually, the states carry the information about the characters already matched. When $q[i] = \$$, it means that we miss the information.

During the scan if we are at position i in the text and in state q in the automaton, then the next position in the text will be $i + s[q, t[i]] - 1$ and the next state will be $d(q, t[i])$. This corresponds to the more simple strategy which consists in trying to match the rightmost unknown character.

Example 8.1. $w = aba$ and $A = \{a, b\}$ (see Fig. 23).

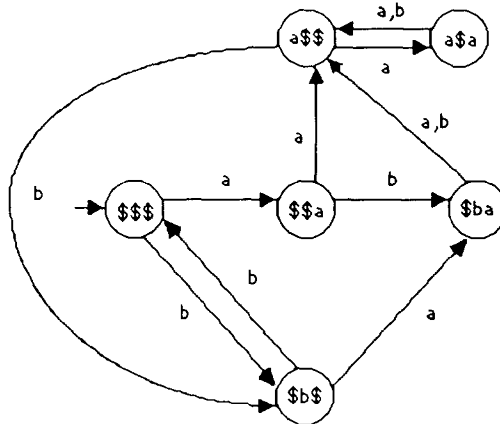

Fig. 23. The Boyer–Moore automaton for $w = aba$.

The shift function s is given as follows:

	a	b
\$\$\$	0	2
\$\$$a$	4	0
\$$b$\$	3	6
a\$\$	0	2
\$$ba$	6	6
a\$$a$	5	5

Actually, no upper bound different from the straightforward 2^m is known for the number of states of the Boyer–Moore automaton. We only know a polynomial upper bound for a family of words; see [3, 6].

Our strategy leads to an automaton with a number of states in $O(m^3)$ since the states are all of the form: a known prefix (possibly empty), an unknown portion, another known portion (possibly empty) and another unknown part. And we try to match the unknown character just on the left of the rightmost known portion or the rightmost character of the text if the rightmost known portion of the text is empty:

$$q = w[i_0]\ldots w[i_1]u[j_0]\ldots u[j_1]v[k_0]\ldots v[k_1]x[q_0]\ldots x[q_1],$$

where

$$i_0 = 0 \text{ or } (i_0 = 1 \text{ and } i_0 \leqslant i_1 \leqslant m),$$

$$j_0 = i_1 \text{ or } (j_0 = i_1 + 1, j_0 \leqslant j_1 \leqslant m \text{ and } u[j] = \$ \text{ for } j_0 \leqslant j \leqslant j_1),$$

$$k_0 = j_1 \text{ or } (k_0 = j_1 + 1, k_0 \leqslant k_1 \leqslant m \text{ and } v[k] = w[k] \text{ for } k_0 \leqslant k \leqslant k_1),$$

$$q_0 = k_1 \text{ or } (q_0 = k_1 + 1, q_0 \leqslant q_1 \leqslant m \text{ and } x[q] = \$ \text{ for } q_0 \leqslant q \leqslant q_1),$$

$$\forall q \in Q \quad q = w[1]\ldots w[i]\$\ldots\$w[j]\ldots w[j+k]\$\ldots\$.$$

There are obviously m^3 words of this form.

9. Comparison with the Boyer–Moore algorithm on few examples

Intuitively, one can expect a better behavior of Algorithm 3 (A3) than of Boyer–Moore algorithm (BM). The latter tends to match small suffixes v of the word against the text. These suffixes are likely to reappear very close in their left context in the word, which leads to small shifts (see Fig. 24).

Algorithm 3 matches small prefixes u of the word but it enables it to perform better shifts than those of Boyer–Moore algorithm (see Fig. 25).

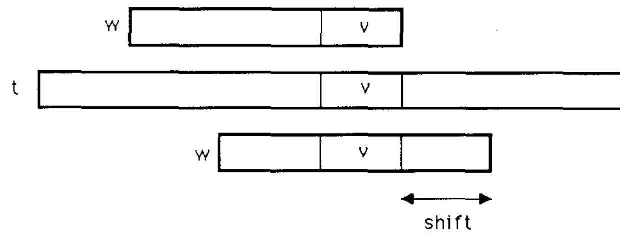

Fig. 24. Shift of the Boyer–Moore algorithm.

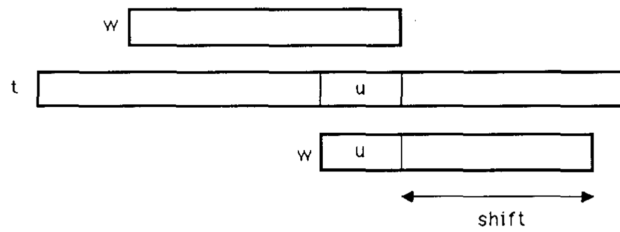

Fig. 25. Shift of Algorithm 3.

Example 9.1.

```
        t  ..........cbca........
        w        abbcabca
BM
        t  ..........cbca........
        w           abbcabca
A3
        t  ..........cbca........
        w               abbcabca
```

Example 9.2.

```
        t  ..........cdabe........
        w       abdabecabe
BM
        t  ..........cdabe........
        w           abdabecabe
A3
        t  ..........cdabe........
        w                 abdabecabe
```

Example 9.3.

t (*dc*)*

t *dcdcdcdcdcdcdcdc* . . . . .

w *acbc*

BM

t *dcdcdcdcdcdcdcdc* . . . . .

w *acbc*

A3

t *dcdcdcdcdcdcdcdc* . . . . .

w   *acbc*

Example 9.4.

t (*aaab*)*

t *aaabaaabaaabaaabaaab* . . . . .

w *aabaa*

BM and A3

t *aaabaaabaaabaaabaaab* . . . . .

w *aabaa*

w   *aabaa*

10. Conclusion

We have presented a new linear string-matching algorithm using the smallest suffix automaton. This algorithm scans each text character at most three times. Its preprocessing is linear in time and space in the length of the word to be found. Intuitively, its behavior seems to be better than that of the Boyer–Moore algorithm, which is known to be one of the fastest string-matching algorithms in practice. The strategy used in this algorithm leads to a Boyer–Moore automaton whose number of states is bounded by the cube of the length of the word.

Acknowledgment

I thank Wojciech Rytter for an interesting discussion on this subject.

References

[1] A.V. Aho, Algorithms for finding patterns in strings, in J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity* (Elsevier, Amsterdam, 1990) 255–300.

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

[3] R.S. Baeza-Yates and G.H. Gonnet, Boyer–Moore automata, Research Report, University of Waterloo, 1989.
[4] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M.T. Chen and J. Seiferas, The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.* **40** (1985) 31–55.
[5] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Comm. ACM* **20** (1977) 762–772.
[6] C. Choffrut, An optimal algorithm for building the Boyer–Moore automaton, *EATCS Bull.* **40** (1990) 217–225.
[7] R. Cole, Tight bounds on the complexity of the Boyer–Moore pattern matching algorithm, to appear.
[8] M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.* **45** (1986) 63–86.
[9] Z. Galil, On improving the worst-case running time of the Boyer–Moore string-matching algorithm, *Comm. ACM* **22** (1979) 505–508.
[10] L.J. Guibas and A.M. Odlyzko, A new proof of the linearity of the Boyer–Moore string searching algorithm, *SIAM J. Comput.* **9** (1980) 672–682.
[11] D.E. Knuth, J.H. Morris Jr. and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977) 323–350.
[12] W. Rytter, A correct preprocessing algorithm for Boyer–Moore string-searching, *SIAM J. Comput.* **9** (1980) 509–512.