Theoretical Computer Science 92 (1992) 33–47
Elsevier

String-matching on ordered alphabets*

Maxime Crochemore
LITP, Université Paris 7, 2 place Jussieu, 75251 Paris Cedex 05, France

Abstract

Crochemore, M., String-matching on ordered alphabets, Theoretical Computer Science 92 (1992) 33–47.

We present a new string-matching algorithm that exploits an ordering of the alphabet. The algorithm is linear in time and uses a fixed number of memory locations in addition to the text and the pattern. Therefore, it is time–space optimal. Its main characteristic is that it scans the pattern from left to right. No preprocessing of the pattern is needed and the complexity is independent of the size of the pattern. An important consequence is the possibility of computing the periods of a word in linear time and constant space. The algorithm can also be turned into a real-time string-matching algorithm.

Introduction

The string-matching problem is to locate a nonempty string, the pattern x, inside another string, the text t. The symbols of both strings belong to a common alphabet A. Several classical string-matching algorithms solve the problem, and the reader can refer to [1] for a review on the subject.

In this paper we consider a particular class of string-matching algorithms. The class is defined by the way words x and t are processed. The algorithms of the class perform a succession of scans of the pattern against the text interrupted by what can be considered as shifts of the pattern towards the end of the text. We even restrict our attention to those algorithms in which the pattern is scanned from left to right. The classical Knuth–Morris–Pratt algorithm (KMP for short) [10] belongs to this class. Its time complexity is linear in the length of the inputs and it also requires an extra space proportional to the length of the pattern. An improvement on KMP algorithm

* The present work was supported by PRC "Mathématiques Informatique" and by NATO under Grant CRG900293.

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

has recently been designed by Simon [12] through an implementation of the automaton underlying KMP algorithm.

Algorithms in the above class are certainly not the most efficient ones according to the minimum number of letter comparisons they perform. From that point of view the practical efficiency of the Boyer–Moore (BM) string-matching algorithm [3] is well known. In fact, in this paper, we are interested in algorithms requiring only a constant amount of extra memory space at run-time. Both KMP and BM algorithms use a space proportional to the length of the pattern.

There exists two time–space optimal string-matching algorithms, namely, the GS algorithm [9] and the CP algorithm [5]. They require a fixed amount of extra space and their worst-case running time is linear. This is the best that can be achieved and this is why they are time–space optimal. But none of them perform left-to-right scans, so that it has become a challenge to design such a time–space optimal algorithm.

Each of the algorithms GS and CP is based on a combinatorial property of words. The former deals with periodicities in the prefixes of the patterns. The latter uses critical factorizations of words. The algorithms factorize the pattern p into uv according to some property of the periodicities existing inside the pattern. The search phase is thereafter guided by the search for the right part v of the pattern. The common feature of both methods is that, through their respective factorizations of the pattern, they tend to avoid some periodicities of the pattern. This contrasts with both KMP algorithm and Simon's algorithm whose first phase amounts to computing the starting periods of the pattern. The present algorithm adopts the same strategy and computes periods instead of avoiding them.

The string-matching algorithm of the present paper can be considered as a first attempt to exploit the fact that the alphabet A is ordered. Of course this is not a restriction because any alphabet can be ordered. Ordering the alphabet has already proved useful for critical factorizations of words (see [5]). Recently also, Apostolico [2] has designed a square-freeness test on words based on an ordering of the alphabet. This is a strange phenomenon that ordering the alphabet, which apparently seems to have nothing to do with the initial problem, turns out to be very fruitful.

The existence of a time–space-optimal, left-to-right-scan string-matching algorithm is rather surprising. The result is of main interest from a theoretical point of view. It provides a new proof that string-matching can be realized by a multihead finite automaton (see [9] for a discussion on this point). Moreover, the algorithm can be turned into a real-time algorithm through a simple application of the method introduced by Galil in [7]. At this stage of the research, however, the algorithm has no practical purpose. It cannot compete, for instance, with algorithms of [3] or [5].

The paper is organized as follows. Section 1 briefly recalls general things about the string-matching problem. Section 2 is devoted to the relation between the smallest period of a word and its maximal suffix according to the alphabetical ordering. The complete new string-matching algorithm is given in Section 3. The method involved is previously presented in a preliminary simpler version of the algorithm. Finally, in

Section 4, we show how the algorithm produces a time–space optimal computation of the smallest period of a word.

1. Scan-and-shift string-matching algorithms

Classical string-matching algorithms find the occurrences of a pattern p inside a test t by considering increasing positions in t. At each position met during the execution of the algorithm, a scan is done to decide whether the pattern occurs here or not. This kind of algorithms thus execute a series of scans and shifts (see Fig. 1). The scans of the pattern against the text at a given position can be realized in several ways. But a scan from left to right is certainly most natural. KMP algorithm, for instance, implements efficiently this left-to-right-scan method.

The typical current situation for left-to-right-scan algorithms is shown in Fig. 2. Text t is equal to $zybz'$ and pattern x is equal to yau', where z, z', u, u' are words on A and a, b are distinct letters of A. Another analogous situation is when $t=zxbz'$, i.e. when an occurrence of the pattern x has been discovered in t at position $|z|$. In either case, after the scan the pattern is shifted to the right and the process continues a series of scans and shifts.

Left-to-right-scan string-matching algorithms differ in the number of places the pattern is shifted to the right. For instance, this number is 1 in the most straightforward algorithm, but more efficient algorithms perform longer shifts that avoid going backwards along the text. Shifts always take into account the periodicities of the pattern that we define now.

Let x be a word on the alphabet A. It is also written $x[1]x[2]...x[n]$ where $x[i]$ is its ith letter. The length n of x is noted $|x|$. An integer p such that $1 \leqslant p \leqslant n$ is called a *period* of x if $x[i]=x[p+i]$ whenever the two sides of the equality are defined. The smallest period of x is called *the period* of x and denoted by $p(x)$.

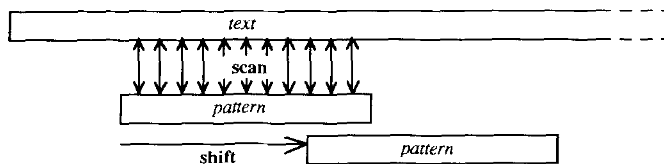

Fig. 1. Scan-and-shift algorithm.

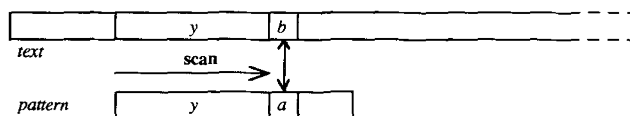

Fig. 2. Left-to-right-scan.

From the situation in Fig. 2, the length of the shift performed during the search phase of the Morris and Pratt's algorithm [11] is $p(y)$. It is $p'(y)$ in KMP algorithm [10], where $p'(y)$ is a period of y not incompatible with the letter b that yields the mismatch between t and x (i.e. not incompatible with any letter different from the letter a that follows y). The best length of the shift is $p(yb)$. Computing all the periods $p(yb)$ amounts to considering the minimal deterministic automaton recognizing the language A^*x. Indeed, MP and KMP algorithms implicitly use a representation of this automaton. Simon [12] has shown how another implementation leads to a faster algorithm. Figure 3 illustrates differences between the three algorithms. Among all the previous algorithms, the naive algorithm is the only one which requires only a fixed number of memory locations in addition to the text and the pattern, but its maximum time complexity is $O(|t| \cdot |x|)$. On the other hand, the three other algorithms work in time $O(|t|)$ but need $|x|$ extra locations and even $|A| \cdot |x|$ for the algorithm that stores all the precomputed periods $p(yb)$. They achieve their linear time complexity mainly because they avoid scanning twice prefixes of x occurring in the text t.

When a shift is to be done, the algorithm of Section 3 tries to compute the period of yb. It does not always find the exact period of yb, but in any case it computes an approximation of it. Moreover, this computation requires only a bounded extra memory space, and the approximation is good enough to produce an overall linear time algorithm.

2. Periods of words and their maximal suffixes

The computation of periods needed to realize shifts in our string-matching algorithm relies on few combinatorial properties of words related to a fixed alphabetical ordering on the set of words A^*. The ordering is denoted by $\leqslant$.

Let v be the suffix of x which is maximal according to the alphabetical ordering. Let u be such that $x = uv$. Considering its smallest period, the word v can be written as $w^e w'$ where $e \geqslant 1$, $|w| = p(v)$ is the period of v and w' is a proper prefix of w. Recall that the pattern x is nonempty so that words u, v, w and w' are well defined. The sequence

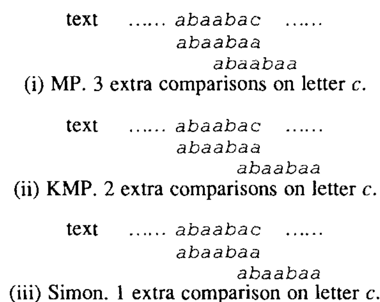

Fig. 3. Shifts after the first comparison on letter c.

(u, w, e, w') is called the *MS-decomposition* of the nonempty word x (MS stands for maximal suffix).

The MS-decomposition of x into uw^ew' as above gives rather precise information on the periods of x. Among interesting properties it is worth noting that w has no smaller period than the trivial one, its length. Such a word is called *border-free.* The inequality $|u| < p(x)$ is a rather intuitive property of the maximal suffix, saying that it must start inside the first period of the word. Proposition 2 provides a more accurate approximation of the smallest period of the word x.

Proposition 1. *Let uw^ew' be the MS-decomposition of a nonempty word x. Then w is border-free which is equivalent to saying that no proper prefix of w is also a suffix of w or that $p(w) = |w|$.*

Proof. Assume that $w = zz' = z''z$ for three nonempty words z, z' and z''. The word $zw^{e-1}w'$ is a suffix of x distinct from v; so, it is greater than v according to the alphabetical ordering. The inequality $zw^{e-1}w' < v$ rewrites $zw^{e-1}w' < zz'w^{e-1}w'$ and implies $w^{e-1}w' < z'w^{e-1}w'$. Moreover, $zw^{e-1}w'$ is not a prefix of v because otherwise the smallest period of v would be $|z''|$, less than $|w|$, contrary to the definition of w. Since then $w^{e-1}w'$ is not a prefix of $z'w^{e-1}w'$, there is a word y and letters a and b such that ya is a prefix of $w^{e-1}w'$, yb is a prefix of $z'w^{e-1}w'$ and $a < b$. Since ya is also a prefix of v, we get $v < z'w^{e-1}w'$, a contradiction with the definition of v.

Thus, w cannot be properly written simultaneously as zz' and $z''z$. This means that $p(w) = |w|$. □

Proposition 2. *Let uw^ew' be the MS-decomposition of a nonempty word x and let $v = w^ew'$ be the maximal suffix of x. Then, the four properties hold:*

(i) *if u is a suffix of w, $p(x) = p(v)$,*
(ii) *$p(x) > |u|$,*
(iii) *if $|u| \geqslant |w|$, $p(x) > |v| = |x| - |u|$,*
(iv) *if u is not a suffix of w and $|u| < |w|$, $p(x) > \min(|v|, |uw^e|)$.*

Proof. (i) When u is a suffix of w, $|w|$ is obviously a period of the whole word x. The smallest period of x cannot be less than the smallest period of its suffix v. Since this period is precisely $|w|$, we conclude that $p(x) = p(v) = |w|$.

(ii) We prove ab absurdo that $p(x) > |u|$. If $p(x) \leqslant |u|$ there is an occurrence of v in x distinct from its occurrence which is a suffix. In other words, x can be written as $u'vv'$ with $|u'| < |u|$ and $|v'| > 0$. But the suffix vv' is then alphabetically greater than v, a contradiction with the definition of v.

(iii) We assume that $|u| \geqslant |w|$. We prove ab absurdo that $p(x) > |v| = |x| - |u|$. If the contrary holds there is an occurrence of u in x distinct from the occurrence which is a prefix of x. This occurrence overlaps v (see Fig. 4). Then, taking into account that $|u| \geqslant |w|$, there is a nonempty word z that is both a prefix of w and a suffix of u. The former property shows that v can be written as zz' (for some word z') and the latter

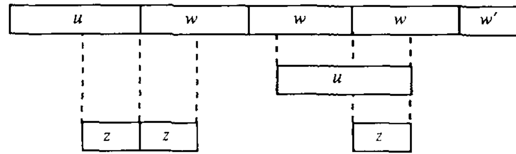

Fig. 4. No suffix of u is a prefix of w.

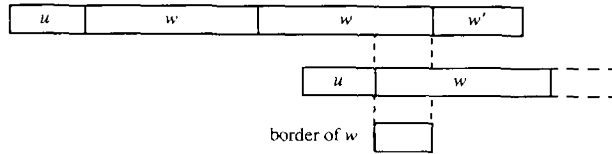

Fig. 5. Impossible because w is border-free.

property shows that zv is a suffix of x. The maximality of v implies $v > zv$, i.e. $zz' > zv$. But this yields $z' > v$, a contradiction with the definition of v.

(iv) Assume that u is not a suffix of w and that $|u| < |w|$. Assume also ab absurdo that $p(x) \leqslant \min(|v|, |uw^e|)$. Let z be the prefix of x of length $p(x)$. The word x itself is then a prefix of zx. From $p(x) \leqslant |v|$ we deduce that the word zu is a common prefix to x and zx. And from $p(x) \leqslant |uw^e|$ we know that u overlaps w^e. If u overlaps the boundary between two w's or the boundary between the last w and w', the same argument as in case (iii) applies and leads to a contradiction. The remaining situation is when u is a factor of w, as shown in Fig. 5. The last occurrence of w in the prefix zuw of zx overlaps an occurrence of w in the prefix uw^e of x. Note that these occurrences of w cannot be equal or adjacent because u is not a suffix of w. This gives a contradiction with the border-freeness of w stated in Proposition 1. □

Note. One may note that in case (iii) u cannot be a suffix of w. Because this would imply $u = w$ which is impossible by (ii).

Examples. Bounds on the smallest period given in Proposition 2 are sharp. We list few examples of patterns that give evidence of this fact. We consider words on the aphabet $\{a, b, c\}$ with the usual ordering $(a < b < c)$.

Let x be *aaaaba*. The maximal suffix of x is $v = ba$ whose smallest period is 2. In the MS-decomposition uw^ew' of x, $u = aaaa$, $w = ba$ and w' is empty. Then $p(x) = 5 = |u| + 1$. As stated in Proposition 2 case (ii) $p(x)$ is greater than $|u|$ but only by one unit.

The word $x = aababa$ meets case (iii) of Proposition 2. Here $u = aa$, $w = ba$ and w' is empty. The smallest period of x, 5, is exactly one unit more than the length of the maximal suffix *baba*.

We exhibit two examples for case (iv) of Proposition 2. The first example is $x = acabca$. We have $u = a$, $w = cab$ and $w' = ca$. Then the quantity $\min(|v|, |uw^e|)$ is $|uw^e| = 4$. The smallest period of x is $p(x) = 5$. The word satisfies $p(x) = |uw^e| + 1 = |v|$. The second example is $x = ababbbab$. For it, $u = aba$, $w = bbba$ and $w' = b$. This is a reverse situation where $\min(|v|, |uw^e|) = |v| = 5$. The smallest period of x is now 6 and we have $p(x) = |v| + 1 < |uw^e|$.

An immediate consequence of Proposition 2 is that condition (i) is certainly true for words having a small period, i.e. having a period not greater than half their length ($p(x) \leqslant |x|/2$). The computation of the smallest period of these words can then be deduced from a computation of their maximal suffixes (and MS-decomposition) together with a test "is u a suffix of w?". This fact has already been used in [5] to approximate smallest periods of patterns. The following corollary is directly used in the proof of the string-matching algorithm of the next section.

Corollary 3. *Let uw^ew' be the MS-decomposition of a nonempty word x and $v = w^ew'$ be the maximal suffix of x.*
If u is a suffix of w, $p(x) = p(v) = |w|$.
Otherwise, $p(x) > \max(|u|, \min(|v|, |uw^e|)) \geqslant |x|/2$.

Proof. When u is not a suffix of w, statements (ii), (iii) and (iv) of Proposition 2 show that the inequality $p(x) > \max(|u|, \min(|v|, |uw^e|))$ holds. It remains to prove that the last quantity is greater than $|x|/2$. The inequality is trivially satisfied if $|u| \geqslant |x|/2$. Otherwise, $|v|$ which is equal to $|x| - |u|$ is greater than $|x|/2$. And $|uw^e|$, equal to $|x| - |w'|$, is also greater than $|x|/2$ because $|w'| < |w|$. This completes the proof. □

3. String-matching algorithm

As output of a string-matching algorithm we consider the set of positions of occurrences of x in t, say

$$P(x, t) = \{pos \in \mathrm{N} \mid 0 \leqslant pos \leqslant |t| \text{ and } t[pos + i] = x[i],\ 1 \leqslant i \leqslant |x|\}.$$

In particular, this set contains the overhanging occurrences of x in t, i.e. the occurrences of x at positions $pos > |t| - |x|$ in t. The set is never empty since it contains at least the position $|t|$.

A first version of the string-matching algorithm is given in Fig. 7. It uses two variables *pos* and *m* that are pointers on text t and pattern x, respectively, as shown in Fig. 6. At each position *pos* inside text t, the algorithm of Fig. 7 executes a scan (line 1), tests whether an occurrence has been found (line 2) and then realizes a shift (lines 4, 5, 8 and 9). The instruction at line 3 is unessential. It takes care of overhanging occurrences of the pattern inside the text, avoiding to go beyond the end of the text. It

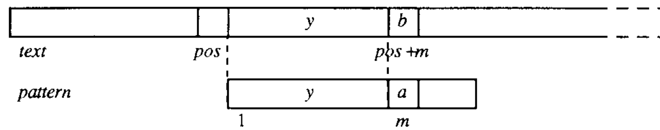

Fig. 6. Variables *pos* and *m*.

```
function POSITIONS(x,t); /* PRELIMINARY VERSION */
begin
  P:=∅;
  pos :=0;  m:=1;
  while (pos ≤ |t|) do
  { /* scan from left to right */
1   while (pos+m≤|t|) and (m≤|x|) and (t[pos+m]=x[m]) do m:=m+1;
    /* test for occurrence */
2   if (pos+m=|t|+1) or (m=|x|+1) then add pos to P;
    /* take care of overhanging occurrences */
3   if (pos+m=|t|+1) then m:=m-1;
    /* shift to the right */
4   (i,j,k,p):=MAXIMAL_SUFFIX(x[1]...x[m-1]t[pos+m]);
5   if (x[1]...x[i] is suffix of
        the prefix of length p of x[i+1]...x[m-1]t[pos+m]) then
8   { pos:=pos+p;  m:=m-p+1; }
    else
9   { pos:=pos+max(i,min(m-i,j))+1;  m:=1; }
  }
  return(P);
end function.
```

Fig. 7. First version of Algorithm POSITIONS.

can be deleted for a search of full occurrences only. The algorithm of Fig. 7 executes shifts of the pattern (lines 8 or 9), as explained in Section 1, after a maximal suffix computation. It applies directly to Corollary 3. To see this point we must explain the meaning of variables i, j, k and p. Let us describe precisely what is the output of Function MAXIMAL_SUFFIX applied to a nonempty word z.

Let the word z be factorized into uv where v is its maximal suffix. Let $uw^e w'$ be the MS-decomposition of z (then $v = w^e w'$). Values of variables i, j, k and p are then given by

$$i = |u|, \qquad j = |uw^e|, \qquad k = |w'| + 1, \qquad p = |w| = p(v).$$

Figure 8 displays the situation. The value of i is the position in z of its maximal suffix v, and j is the position of the rest w'. The smallest period of v is given by p. The 4-tuple (i, j, k, p) is called the *MS-tuple* of z. The MS-decomposition of z and its MS-tuple characterize the same concept. The output of Algorithm MAXIMAL_SUFFIX is precisely the MS-tuple of its input word. The Algorithm itself is given in the Appendix.

Proposition 4. *Algorithm* POSITIONS *in Fig. 7 computes the set of positions of occurrences of pattern x in text t.*

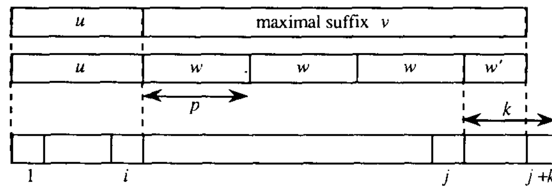

Fig. 8. The MS-tuple (i, j, k, p) of the pattern.

Proof. First note that if the pattern x occurs at position *pos* in t, instruction at line 1 stops with $m=|x|+1$. Thus, the next instruction correctly adds the value of *pos* to the final output P.

It remains to show that no occurrence of the pattern is missed by a too long shift. But this amounts to prove that the length of the shift realized at line 8 or at line 9 is not greater than the period of $x[1] \ldots x[m-1]t[pos+m]$. This is exactly what is stated by Corollary 3. □

Algorithm MAXIMAL_SUFFIX may be implemented to run in time proportional to the length of its input (see the Appendix). But even with this assumption, the algorithm of Fig. 7 does not always run in linear time. Quadratic complexity is due to computations of maximal suffixes. For instance, if the pattern is a^n and the text is also a long repetition of the only letter a, at each position in the text, the maximal suffix of the whole pattern is recomputed leading to an $O(|x| \cdot |t|)$ time complexity. The complete version of the algorithm improves the first version by avoiding entire recomputations of maximal suffixes. This is only possible under certain conditions and it is achieved with the help of another combinatorial property of maximal suffixes. The condition is first introduced on an example and stated in Proposition 5.

Example. Consider the pattern $x=babbbabbbab$. Its maximal suffix is $v=bbbabbbab$. Both words have smallest period 4. With the notation of MS-decompositions, we have $u=ba$, $w=bbba$ and $w'=b$. The exponent of w in v is $e=2$. If the last four letters of x are deleted, corresponding, for instance, to a shift of 4 positions to the right, we are left with the word $x_1=babbbab$. Its MS-decomposition produces $u_1=ba$, $v_1=bbbab$, $w_1=bbba$ and $w'_1=b$. We note that the second decomposition is produced from the first one by pruning one occurrence of w. This generalizes to any word for which the exponent e of the MS-decomposition is greater than 1 (see Fig. 9).

The above result is not necessarily true when $e=1$. Let, for instance, x be $bbabbbabb$. It has period 4 like its maximal suffix $v=bbbabb$. Deletion of the last four letters yields $x_1=bbabb$ which is its own maximal suffix and has period 3. A situation pretty far from the previous one.

Proposition 5. *Let (u, w, e, w') be the MS-decomposition of a nonempty word x ($w^e w'$ is the maximal suffix of $x=uw^e w'$). Assume that $p(x)=|w|$. Then, if $e>1$, $(u, w, e-1, w')$ is*

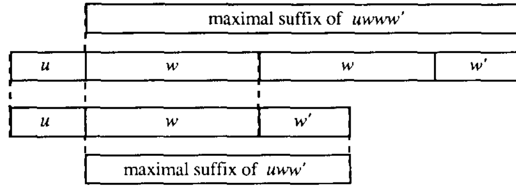

Fig. 9. Highly periodic pattern.

the MS-decomposition of $x'=uw^{e-1}w'$. *In particular, the word* $w^{e-1}w'$ *is the maximal suffix of* x' *and it has period* $p(w^{e-1}w')=|w|$.

Proof. Let $v=w^ew'$ be the maximal suffix of x. Any proper suffix of v of the form $zw^{e-1}w'$ with $z\neq\varepsilon$ is less than v itself by definition. But, since w is border-free (Proposition 1), the longest common prefix of w and z is shorter than z. This leads to $w>z$ and proves that w is greater than all its proper suffixes. This further proves that $w^{e-1}w'$ is its own maximal suffix. And, since hypothesis $p(x)=|w|$ is equivalent to "u is a suffix of w", this also proves that $w^{e-1}w'$ is the maximal suffix of $uw^{e-1}w'$.

Equality $p(w^{e-1}w')=|w|$ is another consequence of the border-freeness of w stated in Proposition 1. Thus, $(u, w, e-1, w')$ is the MS-decomposition of x' as announced. □

One can note that a pattern x that satisfies the hypothesis of Proposition 5 has a rather small period and may be considered as highly periodic. Its smallest period is not greater than half its length. Conversely, if the smallest period of x is not greater than $|x|/3$ then Proposition 5 applies.

We now explain how Proposition 5 is used to improve on the complexity of the first version of Algorithm POSITIONS. When a shift, done according to a period (line 8), leaves a match between the text and the pattern of the form $uw^{e-1}w'$ we can avoid computing the next maximal suffix from scratch. We may better exploit the known decomposition of the match. To do so, we consider Algorithm NEXT_MAXIMAL_SUFFIX that computes the maximal suffix of its input x starting from the maximal suffix of a prefix of x. The algorithm is given in Fig. 10. Except for initializations, it is a copy of Algorithm MAXIMAL_SUFFIX given in the Appendix. The input (i, j, k, p) is assumed to be the MS-tuple associated with the prefix of $x[1]\ldots x[n]$ of length $j+k-1$.

The final version of the string-matching algorithm is shown in Fig. 11. The difference with the first version in Fig. 7 lies in the computation of the MS-tuple (i, j, k, p). In this version, the MS-tuple is still initialized as in Fig. 7 at the beginning of the execution of the algorithm and after each shift, except when the situation of Proposition 5 is met. In this case (shift at line 7) the variable j of the MS-tuple is just decreased by the period p, length of the shift. The test $j-i>p$ at line 6 stands for condition $e\geqslant 2$ in Proposition 5.

```
function NEXT_MAXIMAL_SUFFIX (x[1]...x[n],(i,j,k,p))
  while (j + k ≤ n) do {
    if (x[i+k] = x[j+k]) then
    { if (k = p) then {j:=j+p; k:=1;} else k:=k+1; }
    else if (x[i+k] > x[j+k]) then
    { j:=j+k; k:=1; p:=j-i; }
    else
    { i:=j; j:=i+1; k:=1; p:=1; }
  } end while
  return (i,j,k,p);
end function.
```

Fig. 10. Computing maximal suffixes.

```
function POSITIONS(x,t); /* FINAL VERSION */
begin
  P:=∅;
  pos:=0; m:=1; (i,j,k,p):=(0,1,1,1);
  while (pos ≤ |t|) do
1 { while (pos+m≤|t|) and (m≤|x|) and (t[pos+m]=x[m]) do m:=m+1;
2   if (pos+m=|t|+1) or (m=|x|+1) then P:=P ∪ {pos};
3   if (pos+m=|t|+1) then m:=m-1;
4   (i,j,k,p):=NEXT_MAXIMAL_SUFFIX(x[1]...x[m-1]t[pos+m],(i,j,k,p));
5   if (x[1]...x[i] suffix of
        the prefix of length p of x[i+1]...x[m-1]t[pos+m]) then
6     if (j-i > p) then
7     { pos:=pos+p; m:=m-p+1; j:=j-p; }
      else
8     { pos:=pos+p; m:=m-p+1; (i,j,k,p):=(0,1,1,1); }
    else
9   { pos:=pos+max(i,min(m-i,j))+1; m:=1; (i,j,k,p):=(0,1,1,1);}
  }
  return(P);
end function.
```

Fig. 11. String-matching algorithm.

Proving that the new version of Algorithm POSITION works well extends in a rather straightforward way from Proposition 4 and is left to the reader.

Proposition 6. *The number of letter comparisons executed during a run of Algorithm* POSITIONS *on words x and t is less than* $6\cdot|t|+5$. *This includes comparisons done during calls of Function* NEXT_MAXIMAL_SUFFIX.

Proof. First consider line 5 in Fig. 11. The comparisons executed on the prefix $x[1]\ldots x[i]$ of the pattern can be charged to $t[pos+1]\ldots t[pos+i]$. The instruction at line 5 is eventually followed by instructions at line 7, at line 8 or at line 9. Whichever instruction follows, the value of *pos* increases by more than i. Thus, never again comparisons at line 5 are charged to the factor $t[pos+1]\ldots t[pos+i]$ of the text. The total number of comparisons at line 5 is thus bounded by $|t|$.

We next prove that each other letter comparison executed during a run of Algorithm POSITIONS (including comparisons done during calls of Function NEXT_MAXIMAL_SUFFIX) leads to a strict increment of the value of expression $5 \cdot pos + m + i + j + k$. Since its initial value is 3 and its final value is $5 \cdot |t| + 8$, this proves the claim.

Whatever is the result of the letter comparison inside Function NEXT_MAXIMAL_SUFFIX, the value of $i + j + k$ increases by 1, and even by more than 1 when the last line of the function is executed. It is worth noting that the relation $k \leqslant j - i$ always holds.

Successful comparisons at line 1 trivially increase m and, consequently, expression $5 \cdot pos + m + i + j + k$. At the same line there is at most one unsuccessful comparison. This comparison is eventually followed by instructions at line 7, at line 8 or at line 9. We examine successively the three possibilities.

The effect of line 7 is to replace $5 \cdot pos + m + i + j + k$ by $5 \cdot (pos + p) + (m - p + 1) + i + (j - p) + k$. The expression is increased by $3 \cdot p + 1$ which is greater than 1. Line 8 is executed when $j - i > p$ does not hold. This means indeed that $j - i = p$ ($j - i$ is always a multiple of p). We also know that $i < p$ [see Proposition 2 case (ii)] and one can observe on Function NEXT_MAXIMAL_SUFFIX that $k \leqslant p$. Now, at line 8, m decreases by $p - 1$, i and k decreases by less than p, and $j = i + p$ decreases by less than $2 \cdot p$. Since pos is replaced by $pos + p$ the value of $5 \cdot pos + m + i + j + k$ is increased by more than 1.

Let us examine the execution of line 9. If s is the value of $\max(i, \min(m - i, j)) + 1$ the increment of expression $5 \cdot pos + m + i + j + k$ is $I = 5 \cdot s - (m - 1) - i - (j - 1) - (k - 1) = 5 \cdot s - m - i - j - k + 3$ or $5 \cdot s - 2 \cdot m - i + 2$, because $j + k = m + 1$ after the function call at line 4. The value s is greater than or equal to both i and $m/2$. So $I \geqslant 2$, which proves that again at line 9 expression $5 \cdot pos + m + i + j + k$ increases by more than 1.

The effect of line 6, that can decrease m by one unit, is of no importance in the preceding analysis. □

The complexity of Algorithm POSITIONS is independent of the size of the pattern. For information only, the upper bound on letter comparisons executed by algorithm KMP is $2 \cdot |x| + 2 \cdot |t|$. In the exceptional situation where the pattern is almost as long as the text, the bound becomes $4 \cdot |t|$, a quantity that is to be compared to the $6 \cdot |t| + 5$ of Algorithm POSITIONS.

Theorem 7. *Algorithm* POSITIONS *in Fig.* 11 *applied to a text t and a pattern x runs in time* $O(|t|)$ *and requires constant space.*

Proof. This is an immediate consequence of Proposition 6 because time complexity of Algorithm POSITIONS is proportional to the number of letter comparisons. Variables used in the algorithm require only a constant amount of space. □

4. Time–space optimal computation of periods of words

When p is a period of the word x, the word $x[1] \dots x[n-p]$ is both a proper prefix and suffix of x. It is called a *border* of x. In fact, this defines a one-to-one correspondence between periods and borders of x and the smallest period of x is associated with its longest (proper) border. In other terms, the word x overlaps itself or it occurs inside itself at the overhanging position p.

The main feature of the Knuth, Morris and Pratt's string-matching algorithm is an efficient preprocessing of the borders of all the prefixes of the pattern x. It then provides an algorithm to compute the periods of x. The computation requires between $|x|$ and $2|x|$ letter comparisons and $O(|x|)$ extra memory locations.

Since Algorithm POSITIONS computes all overhanging occurrences of the pattern inside the text, it can be used, in a natural way, to compute all periods of a word. The periods of a word x are the elements of $P(x, x)$, i.e. the positions of x in x itself. Figure 12 gives an algorithm that computes the smallest period of a word. It is a straightforward adaptation of Algorithm POSITIONS and it can easily be extended to compute all periods of its input.

Theorem 8. *The periods of a word x can be computed in time* $O(|x|)$ *with a constant amount of space.*

```
function SMALLEST_PERIOD(x);
begin
  per:=1; m:=1; (i,j,k,p):=(0,1,1,1);
  while (per+m ≤ |x|) do
  { if (x[per+m]=x[m]) then
    { m:=m+1; }
    else
    { (i,j,k,p):=NEXT_MAXIMAL_SUFFIX(x[1]...x[m-1]x[per+m],(i,j,k,p));
      if (x[1]...x[i] suffix of
          the prefix of length p of x[i+1]...x[m-1]x[per+m]) then
        if (j-i > p) then
        { per:=per+p; m:=m-p+1; j:=j-p; }
        else
        { per:=per+p; m:=m-p+1; (i,j,k,p):=(0,1,1,1); }
      else
      { per:=per+max(i,min(m-i,j))+1; m:=1; (i,j,k,p):=(0,1,1,1);}
    }
  }
  return(per);
end function.
```

Fig. 12. Optimal computation of the smallest period.

5. Conclusion

In [9], Galil and Seiferas indicate how their optimal algorithm can be modified to compute overhanging occurrences of the pattern, and thus, the period of a word. Even if their paper contains a flaw related to the adaptation, Galil [8] has shown how it can be adapted. A similar technique applies to the other optimal string-matching algorithm of [5]. Both algorithms scan the word from a precomputed internal position, which makes the computation inherently off-line. The present algorithm is also time–space optimal but, unlike the two above algorithms, it scans the word from left to right. Consequently, the computation of the period is made on-line.

The interest in the string-matching algorithm presented in this paper is mainly theoretical. The algorithm cannot reach the efficiency of the algorithms of [3] (and variants) and [5]. This is due to the left-to-right-scan feature that yields a linear lower bound on the number of letter comparisons (this lower bound is sublinear for BM and CP). Nevertheless with a left-to-right scan the buffer on the text can be reduced to half the length of the pattern. We do not know how much the upper bound $6\cdot|t|$ on the number of letter comparisons can be reduced for optimal string-matching algorithms with left-to-right scan. It is still possible to improve on lengths of shifts in Algorithm POSITIONS, but this does not seem to affect the bound.

The algorithm of Fig. 11 has also the advantage of turning into a real-time algorithm as are both algorithms of [7] and [4]. The proof relies on the transformation introduced by Galil in [7]. He defines a "predictability condition" that is a sufficient condition for an on-line algorithm to be transformable into a real-time algorithm. The predictability condition holds for Algorithm POSITIONS. It also holds for the string-matching algorithm of Simon [12].

Finally, we would like to point out a surprising phenomenon. The combinatorial result of Proposition 2 provides an approximation of the period of a word. The property leads to the string-matching algorithm of Fig. 11 which itself adapts to an exact computation of periods by an optimal and on-line algorithm. This raises the natural question: can the iteration of this feedback effect be helpful in some way, for instance to design a faster string-matching algorithm?

Appendix

The core of the string-matching algorithm of Fig. 7 is an algorithm that computes the alphabetically maximal suffix of words. This algorithm is a straightforward adaptation of an algorithm of [6]. A proof of it can also be found in [5]. The algorithm works in linear time (hint: each letter comparison yields a strict increment of the value of $i+j+k$).

```
function MAXIMAL_SUFFIX (x[1]...x[n])
  i :=0; j:=1; k:=1; p:=1;
  while (j + k ≤ n) do {
    if (x[i+k] = x[j+k]) then
    { if (k = p) then {j:=j+p; k:=1;} else k:=k+1; }
    else if (x[i+k] > x[j+k]) then
    { j:=j+k; k:=1; p:=j-i; }
    else
    { i:=j; j:=i+1; k:=1; p:=1; }
  } end while
  /* the maximal suffix of x[1]...x[n] is x[i+1]...x[n], */
  /* and p is the smallest period of x[i+1]...x[n]       */
  return (i,j,k,p);
end function.
```

Fig. 13. Computation of maximal suffixes.

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, Optimal parallel detection of squares in strings, Reports CSD-TR-932 and CSD-TR-1012, Purdue University, 1990.

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

[4] M. Crochemore, Longest common factor of two words, in: Ehrig, Kowalski, Levi and Montanari, eds., *TAPSOFT '87, Vol. 1* (Springer, Berlin, 1987) 26–36.

[5] M. Crochemore and D. Perrin, Two-way pattern matching, Rapport LITP 89-8, Université Paris 7, 1989; to appear in *J. ACM.*

[6] J.P. Duval, Factorizing Words over an Ordered Alphabet, *J. Algorithms* **4** (1983) 363–381.

[7] Z. Galil, String Matching in real time, *J. ACM* **28** (1) (1981) 134–149.

[8] Z. Galil, personal communication, 1990.

[9] Z. Galil and J. Seiferas, Time–space optimal string matching, *J. Comput. System Sci.* **26** (1983) 280–294.

[10] D.E. Knuth, J.H. Morris Jr and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (2) (1977) 323–350.

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

[12] I. Simon, personal communication, 1989.