Linear-Time Computation of Local Periods

Jean-Pierre Duval[1], Roman Kolpakov[2,⋆], Gregory Kucherov[3], Thierry Lecroq[4], and Arnaud Lefebvre[4]

[1] LIFAR, Université de Rouen, France
Jean-Pierre.Duval@univ-rouen.fr
[2] Department of Computer Science, University of Liverpool, UK
R.Kolpakov@csc.liv.ac.uk
[3] INRIA/LORIA, Nancy, France
Gregory.Kucherov@loria.fr
[4] ABISS, Université de Rouen, France
{Thierry.Lecroq,Arnaud.Lefebvre}@univ-rouen.fr

Abstract. We present a linear-time algorithm for computing *all* local periods of a given word. This subsumes (but is substantially more powerful than) the computation of the (global) period of the word and on the other hand, the computation of a critical factorization, implied by the Critical Factorization Theorem.

1 Introduction

Periodicities in words have been classically studied in word combinatorics and are at the core of many fundamental results [18,2,19]. Besides, notions and techniques related to periodic structures in words find their applications in different areas: data compression [24], molecular biology [12], as well as for designing more efficient string search algorithms [11,3,5].

In this paper, we concentrate, from the algorithmic perspective, on the important notion of *local periods*, that characterize a local periodic structure at each location of the word [9,8]. In informal terms, the local period at a given position is the size of the smallest *square* centered at this position. An importance of local periods is evidenced by the fundamental Critical Factorization Theorem [18,2,19] that asserts that there exists a position in the word (and a corresponding factorization), for which the local period is equal to the global period of the word.

Designing efficient algorithms for computing different periodic structures in words has been for a long time an active area of research. It is well-known that the (global) period of a word can be computed in linear time, using the Knuth-Morris-Pratt string matching method [16,4].On the other hand, in [3] it has been shown that a critical factorization can be constructed in linear time, by computing the smallest and largest suffixes under the lexicographical ordering. In the same work, the factorization has then been used to design a new string matching algorithm.

⋆ On leave from the French-Russian Institute for Informatics and Applied Mathematics, Moscow University, Russia

© Springer-Verlag Berlin Heidelberg 2003

In this paper, we show how to compute *all* local periods in a word in time $O(n)$ assuming an alphabet of constant size. This is substantially more powerful than linear-time computations of a critical factorization and of the global period: indeed, once all local periods have been computed, the global period is simply the maximum of all local periods, and each such maximal value corresponds to a distinct critical factorization.

Note that a great deal of work has been done on finding periodicities occurring in a word (see [13] for a survey). However, none of them allows to compute all local periods in linear time. The reason is that most of those algorithms are intrinsically super-linear, which can be explained by the fact that they tend, explicitly or implicitly, to enumerate all squares in the word, the number of which can be super-linear. The closest result is the one of [17] which claims a linear-time algorithm for finding, for each position i of the string, the smallest square starting at i. The approach is based on a sophisticated analysis of the suffix tree. The absence of a complete proof prevents the comprehension of the algorithm in full details; however, to the best of our understanding, this approach cannot be applied to finding local periods.

Here we design a linear-time algorithm for finding all local periods, based on several different string matching techniques. Some of those techniques (s-factorization, Main-Lorentz extension functions) have already been successfully used for several repetition finding problems [7,21,20,13,14,15]. In particular, in [13], it has been shown that all *maximal repetitions* can be found in linear time, providing an exhaustive information about the periodic structure of the word. However, here again, a direct application of this approach to finding local periods leads to a super-linear algorithm. We then propose a non-trivial modification of this approach, that allows to find a subclass of local periods in linear time. Another tool we use is the *simplified Boyer-Moore shift function*, which allows us to complete the computation of local periods, staying within the linear time bound.

2 Local Periods: Preliminaries

Consider a word $w = a_1...a_n$ over a finite alphabet. $|w|$ denotes the length of w, and w^R stands for the reverse of w, that is $a_n a_{n-1} \dots a_1$. $w[i..j]$, for $1 \leq i, j \leq n$, denotes the subword $a_i...a_j$ provided that $i \leq j$, and the empty word otherwise. A position i in w is an integer number between 0 and n, associated with the factorization $w = uv$, where $|u| = i$.

A *square* s is a word of the form tt (i.e. a word of even length with two equal halves). t is called the *root* of s, and $|t|$ is called its *period.*

Definition 1. *Let $w = uv$, and $|u| = i$. We say that a non-empty square tt* is centered at position i of w *(or* matches w at central position i*) iff the following conditions hold:*

(i) t is a suffix of u, or u is a suffix of t,
(ii) t is a prefix of v, or v is a prefix of t.

In the case when t is a suffix of u and t is a prefix of v, we have a square occurring inside w. We call it an *internal square.* If v is a proper prefix of t (respectively, u is a proper suffix of t), the square is called *right external* (respectively, *left external*).

Definition 2. *The smallest square centered at a position i of w is called the* minimal local *square (hereafter simply* minimal*, for shortness). The* local period *at position i of w, denoted $LP_w(i)$, is the period of the minimal square centered at this position.*

Note that for each position i of w, $LP_w(i)$ is well-defined, and $1 \leq LP_w(i) \leq |w|$.

Any word w has the *(global) period* $p(w)$, which is the minimal integer p such that $w[i] = w[i+p]$ whenever $1 \leq i, i+p \leq |w|$. Equivalently, $p(w)$ is the smallest positive integer p such that words $w[1..n-p]$ and $w[p+1..n]$ are equal. The critical factorization theorem [18,2,19] is a fundamental result relating local and global periods:

Theorem 1 (Critical Factorization Theorem). *For each word w, there exists a position i (and the corresponding factorization $w = uv$, $|u| = i$) such that $LP_w(i) = p(w)$. Moreover, such a position exists among any $p(w)$ consecutive positions of w.*

Apart from its combinatorial consequences, an interesting feature of the critical factorization is that it can be computed very efficiently, in a time linear in the word length [3]. This can be done, for example, using the suffix tree construction [4]. On the other hand, it is well-known that the (global) period of a word can be computed in linear time, using, for example, the Knuth-Morris-Pratt technique [4].

In this paper, we show how to compute *all* local periods in a word in linear time. This computation is much more powerful than that of a critical factorization or the global period : once all local periods are computed, the global period is equal to the maximum among them, and each such maximal local period corresponds to a critical factorization of the word.

The method we propose consists of two parts. We first show, in Section 3, how to compute all *internal* minimal squares. Then, in Section 4 we show how to compute left and right external minimal squares, in particular for those positions for which no internal square has been found. Both computations will be shown to be linear-time, and therefore computing all local periods can be done within linear time too.

3 Computing Internal Minimal Squares

Finding internal minimal squares amounts to computing, for each position of the word, the smallest square centered at this position and occurring entirely inside the word, provided that such a square exists. Thus, throughout this section we will be considering only squares occurring inside the word and therefore, for the sake of brevity, omit the adjective "internal".

The problem of finding squares and, more generally, finding repetitions occurring in a given word has been studied for a long time in the string matching area, we refer to [13] for a survey. A natural idea is then to apply one of those methods in order to compute *all* squares and then select, for each central position, the smallest one. A direct application of this approach, however, cannot result in a linear-time algorithm, for the reason that the overall number of squares in a word can be as big as $\Theta(n \log n)$ (see [6]). Therefore, manipulating the set of all squares explicitly is prohibitive for our purpose. In [13], *maximal repetitions* have been studied, which are maximally extended runs of consecutive squares. Importantly, the set of maximal repetitions encodes the whole set of squares, while being only of linear size.

Our approach here is to use the technique of computing maximal repetitions in order to retrieve squares which are minimal for some position. To present the algorithm in full details, we first need to describe the techniques used in [20,13] for computing maximal repetitions.

3.1 *s*-Factorization, Main-Lorentz Extension Functions, and Computing Repetitions

In this section we recall basic ideas, methods and tools underlying our approach.

The *s*-factorization [7] is a special decomposition of the word. It is closely related to the Lempel-Ziv factorization (implicitly) defined by the well-known Lempel-Ziv compression method. The idea of defining the *s*-factorization is to proceed from left to right and to find, at each step, the longest factor which has another copy on the left. Alternatively, the Lempel-Ziv factorization considers the shortest factor which does not appear to the left (i.e. extends by one letter the longest factor previously occurred). We refer to [12] for a discussion on these two variants of factorization. A salient property of both factorizations is that they can be computed in linear time [22] in the case of constant alphabet.

In their original definition, both of these factorizations allow an overlap between a factor and its left copy. However, we can restrict this and require the copy to be non-overlapping with the factor. This yields a *factorization without copy overlap* (see [15]). Computing the *s*-factorization (or Lempel-Ziv factorization) without copy overlap can still be done in linear time.

In this work we will use the *s*-factorization without copy overlap:

Definition 3. *The s-factorization of w without copy overlap is the factorization $w = f_1 f_2 \dots f_m$, where f_i's are defined inductively as follows:*

(i) $f_1 = w[1]$,

(ii) *assume we have computed $f_1 f_2 \dots f_{i-1}$ ($i \geq 2$), and let $w[b_i]$ be the letter immediately following $f_1 f_2 \dots f_{i-1}$ (i.e. $b_i = |f_1 f_2 \dots f_{i-1}| + 1$). If $w[b_i]$ does not occur in $f_1 f_2 \dots f_{i-1}$, then $f_i = w[b_i]$, otherwise f_i is the longest subword starting at position b_i, which has another occurrence in $f_1 f_2 \dots f_{i-1}$.*

Note however that the choice of the factorization definition is guided by the simplicity of algorithm design and presentation clarity, and is not unique.

Our second tool is Main-Lorentz extension functions [21]. In its basic form, the underlying problem is the following. Assume we are given two words w_1, w_2 and we want to compute, for each position i of w_2, the longest prefix of w_1 which occurs at position i in w_2. This computation can be done in time $O(|w_1| + |w_2|)$ [21]. Note that w_1 and w_2 can be the same word, and that if we invert w_1 and w_2, we come up with the symmetric computation of longest suffixes of $w_2[1..i]$ which are suffixes of w_1.

We now recall how Main-Lorentz extension functions are used for finding repetitions. The key idea is illustrated by the following problem. Assume we have two words $w_1 = w_1[1..m]$ and $w_2 = w_2[1..n]$ and consider their concatenation $w = w_1w_2$. Assume we want to find all squares of w which cross the boundary between w_1 and w_2, i.e. squares which start at some position $\leq m$ and end at some position $> m$ in w (start and end positions of a square are the positions of respectively its first and last letter). First, we divide all such squares into two categories – those centered at a position $< m$ and those centered at a position $\geq m$ – and by symmetry, we concentrate on the squares centered at a position $\geq m$ only. We then compute the following extension functions :

- $\mathit{pref}(i)$, $2 \leq i \leq n+1$ defined by $\mathit{pref}(i) = \max\{j | w_2[1..j] = w_2[i..i+j-1]\}$ for $2 \leq i \leq n$, and $\mathit{pref}(n+1) = 0$,
- $\mathit{suf}(i)$, $1 \leq i \leq n$ defined by $\mathit{suf}(i) = \max\{j | w_1[m-j+1..m] = w[m+i-j+1..m+i]\}$.

Then there exists a square with period p iff

$$\mathit{suf}(p) + \mathit{pref}(p+1) \geq p \tag{1}$$

[20]. This gives a key of the algorithm: we first compute values $\mathit{pref}(p)$ and $\mathit{suf}(p)$ for all possible p, which takes time $O(m+n)$. Then we simply check for each p inequality (1) – each time it is verified, we witness new squares of period p. More precisely, whenever the inequality is verified we have identified, in general, a series (*run*) of squares centered at each position from the interval $[m - \mathit{suf}(p) + p..m + \mathit{pref}(p+1)]$. This run is a *maximal repetition* in w (see [13]). Formally, this maximal repetition may contain squares centered at positions $< m$ (if $\mathit{suf}(p) > p$), and squares starting at positions $> m$ (if $\mathit{pref}(p+1) > p-1$). Therefore, if we want only squares centered at positions $\geq m$ and starting at positions $\leq m$ (as it will be our case in the next Section), we have to restrict the interval of centers to $[\max\{m - \mathit{suf}(p) + p, m\}.. \min\{m + \mathit{pref}(p+1), m+p\}]$. Clearly, verifying inequality (1) takes constant time and the whole computation can be done in $O(n)$.

To find, in linear time, all squares in a word (and not only those which cross a given position), we have to combine the factorization and extension function techniques. In general terms, the idea is the following : we compute the s-factorization and process factors one-by-one from left to right. For each factor f_r, we consider separately those squares which occur completely inside f_r, and those ending in f_r and crossing the boundary with f_{r-1}. The squares of the first type are computed using the fact that f_r has a copy on the left – we can then retrieve those squares from this copy in time $O(|f_r|)$. The squares of the second type are computed using the extension function technique sketched above, together with an additional lemma asserting that those squares cannot extend to the left of f_r by more than $|f_r| + 2|f_{r-1}|$ letters [20]. Therefore, finding all these squares, in form of runs, takes time $O(|f_{r-1}| + |f_r|)$. The whole word can then be processed in time $O(n)$. The reader is referred to [20,13] for full details.

This general approach, initiated in [7,20] has been applied successfully to various repetition finding problems [13,14,15]. In this work we show that it can be also applied to obtain a linear-time algorithm for computing internal local periods. This gives yet another illustration of the power of the approach.

3.2 Finding Internal Minimal Squares

We are ready now to present a linear-time algorithm of computing all internal minimal squares in a given word w.

First, we compute, in linear time, the s-factorization of w without copy overlap and we keep, for each factor f_r, a reference to its non-overlapping left copy. The algorithm processes all factors from left to right and computes, for each factor f_r, all minimal squares ending in this factor. For each minimal square found, centered at position i, the corresponding value $LP_w(i)$ is set. After the whole word has been processed, positions i for which values $LP_w(i)$ have not been assigned are those positions for which no internal square centered at i exists. For those positions, minimal squares are external, and they will be computed at the second stage, presented in Section 4.

Let $f_r = w[m+1..m+l]$ be the current factor, and let $w[p+1..p+l]$ be its left copy (note that $p+l \leq m$). If for some position $m+i$, $1 \leq i < l$, the minimal square centered at $m+i$ occurs entirely inside the factor, that is $LP_w(m+i) \leq \min\{i, l-i\}$, then $LP_w(m+i) = LP_w(p+i)$. Note that $LP_w(p+i)$ has been computed before, as the minimal square centered at $p+i$ ends before the beginning of f_r. Based on this, we retrieve, in time $O(|f_r|)$, all values $LP_w(m+i)$ which correspond to squares occurring entirely inside f_r. It remains to find those values $LP_w(m+i)$ which correspond to minimal squares that end in f_r and extend to the left beyond the border between f_r and f_{r-1}.

To do this, we use the technique of computing squares described in the previous section. The idea is to compute all candidate squares and test which of them are minimal. However, this should be done carefully: as mentioned earlier, this can break down the linear time bound, because of a possible super-linear number of all squares. The main trick is to keep squares in runs and to show that there is only a linear number of individual squares which need to be tested for minimality. As in [20], we divide all squares under consideration into those which are centered inside f_r and those centered to the left of f_r. Two cases are symmetrical and therefore we concentrate on those squares centered at positions $m..m+l-1$. In addition, we are interested in squares starting at positions $\leq m$ and ending inside f_r. We compute all such squares *in the increasing order of pe-*

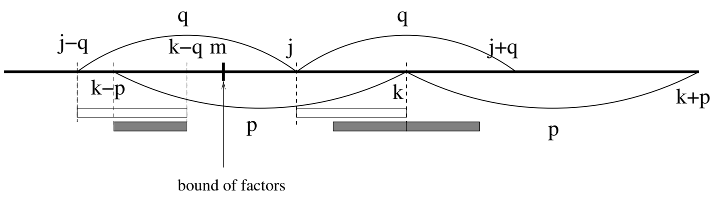

Fig. 1. Case where neither of inequations (2),(3) holds (subcase $k > j$)

riods. For each $p = 1..l-1$ we compute the run of all squares of period p centered at positions belonging to the interval $[m..m+l-1]$, starting at a position $\leq m$, and ending inside f_r, as explained in Section 3.1. Assume we have computed a run of such squares of period p, and assume that $q < p$ is the maximal period for which squares have been previously found. If $p \geq 2q$, then we check each square of the run whether it is minimal or not by checking the value $LP_w(i)$. If this square is not minimal, then its center i has been already assigned a value $LP_w(i)$. Indeed, if a smaller square centered at i exists, it has necessarily been already computed by the algorithm (recall that squares are computed in the increasing order of periods), and therefore a positive value $LP_w(i)$ has been set before. If no value $LP_w(i)$ has yet been assigned, then we have found the minimal square centered at i. Since there is $\leq p$ of considered squares of period p (their centers belong to the interval $[m..m+p-1]$), checking all of them takes $\leq 2(p-q)$ individual checks (as $q \leq p/2$ and $p-q \geq p/2$).

Now assume $p < 2q$. Consider a square $s_q = w[j-q+1..j+q]$ of period q and center j, which has been previously found by the algorithm (square of period q in Figure 1). We now prove that we need to check for minimality only those squares s_p of period p which have their center k verifying one of the following inequalities :

$$|k-j| \leq p-q, \text{ or} \tag{2}$$

$$k \geq j+q \tag{3}$$

In words, k is located either within distance $p-q$ from j, or beyond the end of square s_q.

Show that one of inequations (2),(3) must hold. By contradiction, assume that neither of them holds. Consider the case $k > j$, case $k < j$ is symmetric. The situation with $k > j$ is shown in Figure 1. Now observe that word $w[j+1..k]$ has a copy $w[j-q+1..k-q]$ (shown with empty strips in Figure 1) and that its length is $(k-j)$. Furthermore, since $k-j > p-q$ (as inequation (2) does not hold), this copy overlaps by $p-q$ letters with the left root of s_p. Consider this overlap $w[k-p+1..k-q]$ (shadowed strip in Figure 1). It has a copy $w[k+1..k+(p-q)]$ and another copy $w[k-(p-q)+1..k]$ (see Figure 1). We thus have a smaller square centered at k, which proves that square s_p cannot be minimal.

Therefore, we need to check for minimality only those squares s_p which verify, with respect to s_q, one of inequations (2),(3). Note that there are at most $2(p-q)$

squares s_p verifying (2), and at most $p - q$ squares s_p verifying (3), the latter because s_p must start before the current factor, i.e. $k \leq m+p$. We conclude that there are $\leq 3(p-q)$ squares of period p to check for minimality, among all squares found for period p. Summing up the number of all individual checks results in a telescoping sum, and we obtain that processing all squares centered in the current factor can be done in time $O(|f_r|)$. A similar argument applies to the squares centered on the left of f_r. Note that after processing f_r, all minimal squares ending in f_r have been computed. To sum up, we need to check for minimality only $O(|f_{r-1}| + |f_r|)$ squares, among those crossing the border between f_r and f_{r-1}, each check taking a constant time. We also need $O(|f_r|)$ time to compute minimal squares occurring inside f_r. Processing f_r takes then time $O(|f_{r-1}| + |f_r|)$ overall, and processing the whole word takes time $O(n)$.

Theorem 2. *In a word of length n, all internal minimal squares can be computed in time $O(n)$.*

4 Computing External Minimal Squares

In this section, we show how to compute minimal external squares for those positions which don't have internal squares centered at them. The algorithm is based on the *simplified Boyer-Moore shift function*, used in classical string matching algorithms [1,16].

Definition 4. *For a word w of length n the simplified Boyer-Moore shift function is defined as follows [1,16]:*

$$d_w(i) = \min\{\ell \mid \ell \geq 1 \text{ and } (\text{for all } j,\, i < j \leq n,\, \ell \geq j \text{ or } w[j] = w[j-\ell])\}\,.$$

In words, $d_w(i)$ is the smallest shift between suffix v and its copy in w. If v has no other occurrence in w, then we look for the longest suffix of v occurring in prefix of w. The function d_w can be computed in $O(n)$ time and space [1].

We will show that, given a word w of length n, all minimal external squares can be computed in time $O(n)$. Consider a word w and assume that the function d_w has been computed. Consider a factorization $w = uv$ and assume that there is no internal square centered at position $|u|$. We first consider the case when $|u| \geq |v|$, and show how to compute the minimal right external square centered at $|u|$.

Lemma 1. *Let $w = uv$ with $|u| \geq |v|$. If there is no internal square centered at $i = |u|$, then the minimal right external square has period $d_w(i)$.*

Proof. First note that $d_w(i) > |v|$ must hold, as otherwise there is a copy of v overlapping (or touching) the suffix occurrence of v, which implies that there is an internal square of period $d_w(i)$ centered at i, which contradicts our assumption. We now consider two cases. If $d_w(i) \leq i$,then there is an occurrence of v inside u and therefore $u = u_0vu_1$ for some u_0, u_1. It follows that there is a right external square centered at i with the root vu_1. This is the minimal such square, as the definition of d_w guarantees that u_1 is the shortest possible.

If $d_w(i) > i$,then $v = v_0v_1$ and $u = v_1u_0$ with $|v_1u_0v_0| = d_w(i)$. $v_1u_0v_0$ forms the root of a right and left external square centered at i. Again, the existence of a smaller right external square would contradict the minimality requirement in the definition of d_w.

The case $|u| < |v|$ is symmetric and can be treated similarly by considering the inverse of w.

To conclude, all external minimal squares can be computed in time $O(n)$, for those positions which don't have internal squares centered in them. We then obtain an $O(n)$ algorithm for computing all minimal squares: first, using the algorithm of Section 3 we compute all internal minimal squares and then, using Lemma 1,we compute all external minimal squares for those positions for which no internal square has been found at the first stage. This proves the main result.

Theorem 3. *In a word w of length n, all local periods $LP_w(i)$ can be computed in time $O(n)$.*

5 Conclusions

We presented an algorithm that computes all local periods in a word in a time linear in length of the word. This computation provides an exhaustive information about the local periodic structure of the word. According to the Critical Factorization Theorem, the (global) period of the word is simply the maximum among all local periods. Therefore, as a case application, our algorithm allows to find *all* possible critical factorization of the word. The main difficulty to solve was to extract all shortest local squares without having to process all individual squares occurring in the word, which would break down the linear time bound. This made impossible an off-the-shelf use of existing repetition-finding algorithms, and necessitated a non-trivial modification of existing methods. An interesting research direction would be to study the combinatorics of possible sets of local periods, in a similar way as it was done for the structure of all (global) periods [10,23]. The results presented in this paper might provide an initial insight for a such study.

Acknowledgments

GK, TL and AL have been supported by the french *Action Spécifique "Algorithmes et Séquences"* of *CNRS*. JPD, TL and AL have been supported by the NATO grant PST.CLG.977017. Part of this work has been done during the stay of RK at LORIA in summer 2002, supported by INRIA.

References

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

2. Ch. Choffrut and J. Karhumäki. Combinatorics of words. In G. Rozenberg and A. Salomaa, editors, *Handbook on Formal Languages*, volume I, 329–438, Springer Verlag, 1997.
3. M. Crochemore and D. Perrin. Two-way string matching. *J. ACM*, 38:651–675, 1991.
4. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.
5. M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13:405–425, 1995.
6. M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12:244–250, 1981.
7. M. Crochemore. Recherche linéaire d'un carré dans un mot. *Comptes Rendus Acad. Sci. Paris Sér. I Math.*, 296:781–784, 1983.
8. J.-P. Duval, F. Mignosi, and A. Restivo. Recurrence and periodicity in infinite words from local periods. *Theoretical Computer Science*, 262(1):269–284, 2001.
9. J.-P. Duval. Périodes locales et propagation de périodes dans un mot. *Theoretical Computer Science*, 204(1-2):87–98, 1998.
10. Leo J. Guibas and Andrew M. Odlyzko. Periods in strings. *Journal of Combinatorial Theory,* Series A, 30:19–42, 1981.
11. Z. Galil and J. Seiferas. Time-space optimal string matching. *Journal of Computer and System Sciences*, 26(3):280–294, 1983.
12. D. Gusfield. *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology.* Cambridge University Press, 1997.
13. R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. of FOCS'99, New York (USA)*, 596–604, IEEE Comp. Soc., 1999.
14. R. Kolpakov and G. Kucherov. Finding repeats with fixed gap. In *Proc. of the 7th SPIRE, La Coruña, Spain* , 162–168, IEEE, 2000.
15. R. Kolpakov and G. Kucherov. Finding Approximate Repetitions under Hamming Distance. In F.Meyer auf der Heide, editor, *Proc. of the 9th ESA, Aarhus, Denmark*, LNCS 2161, 170–181, 2001.
16. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.
17. S. R. Kosaraju. Computation of squares in string. In M. Crochemore and D. Gusfield, editors, *Proc. of the 5th CPM*, LNCS 807, 146–150, Springer Verlag, 1994.
18. M. Lothaire. *Combinatorics on Words*, volume 17 of *Encyclopedia of Mathematics and Its Applications.* Addison Wesley, 1983.
19. M. Lothaire. *Algebraic Combinatorics on Words.* Cambridge University Press, 2002.
20. M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25:145–153, 1989.
21. M.G. Main and R.J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
22. M. Rodeh, V.R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981.
23. E. Rivals and S. Rahmann. Combinatorics of periods in strings. In J. van Leuween P. Orejas, P. G. Spirakis, editors, *Proc. of the 28th ICALP*, LNCS 2076, 615–626, Springer Verlag, 2001.
24. J.A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Rockville, MD, 1988.