On-line Minimum Closed Covers

Costas S. Iliopoulos and Manal Mohamed

Department of Informatics, King's College London,
London WC2R 2LS, United Kingdom
c.iliopoulos@kcl.ac.uk

Abstract. The Minimum Closed Covers problem asks us to compute a minimum size of a closed cover of given string. In this paper we present an on-line $O(n)$-time algorithm to calculate the size of a minimum closed cover for each prefix of a given string w of length n. We also show a method to recover a minimum closed cover of each prefix of w in greedy manner from right to left.

Keywords: String, Cover, Closed String, Algorithm.

1 Introduction

The computation of various kinds of "regularities" in a given string has been of interest for more than a century. Such interest has been initiated with the computation of periodicities [23] and later extended, in response to applications arising in data compression and molecular biology, to other concepts of regularities. Apostolico *et al* introduced the idea of *cover* (*quasiperiod*); that is, a substring u is called a cover of given string w if concatenations of u generates w. Clearly the notion of a cover is one way to grasp the repetitive structures of w [2]. Several algorithms to compute the covers of a given string were published in the 1990s [5, 22, 14], culminating in an algorithm [19] that in $O(n)$ time computes a *cover array* specifying all the covers of every prefix of w.

The idea of covers was further extended to consider a set of substrings of w that covers w rather than a single substring. In particular, aiming for a set of minimum size, the *k-covers* problem calculates for a particular integer k, a set of substrings all of length k that covers w [13]. Also, the *λ-covers* problem finds a minimum set of λ substrings of equal length that covers w with the minimum error, under a variety of distance models [12]. More recently, the *palindromic covers* was proposed in which a minimum set of palindromic substrings that covers w is calculated [24]. In this paper, we consider yet another set of substrings that cover w, namely the set of *closed* substrings. We chose to investigate the idea of a cover using set of closed substrings due to the well-studied relation between palindromic and closed strings[6, 4]. We aim to design an algorithm that calculates a minimum set of closed substrings of w that covers a given string w.

More precisely, a string w is called a closed string if it is empty or it has a factor (different from the string itself) occurring exactly twice in, as a prefix and as a

L. Iliadis et al. (Eds.): AIAI 2014 Workshops, IFIP AICT 437, pp. 106–115, 2014.

suffix, that is, with no internal occurrences. Initially, the notion of closed string has been proposed and characterized by Fici [10] as a way to classify trapezoidal strings. In a more recent work, Badkobeh, Fici and Lipták [4] showed that any string of length n contains at least $n+1$ closed factors. They also investigated and provided a combinatorial characterization of the set of strings with the smallest number of closed factors which they called *C-poor* strings.

The paper is organized as follows: in Section 2, we state the preliminaries used throughout the paper. In Section 3, we define the longest closed cover problem and detail our algorithm.

2 Preliminaries

Throughout the paper w denotes a *string* of length n defined on a finite alphabet Σ. We use $w[i]$, for $i = 1, 2, \ldots, n$, to denote the i-th letter of w, and $w[i..j]$ as a notation for the *factor* (*substring*) $w[i]w[i+1]\cdots w[j]$ of w. The *length* of w, the non-negative integer n, is denoted by $|w|$. The empty string has length zero and is denoted by ϵ. A *power* is a string that is a concatenation of copies of another string.

A *prefix* (resp. a *suffix*) of a string w is any string u such that $w = uz$ (resp. $w = zu$) for some string z. A factor of w is a prefix of a suffix (or equivalently, a suffix of a prefix) of w. A *border* of w is any string $u \neq w$ that is a prefix and a suffix of w. From the definitions, we have that ϵ is a prefix, a suffix and a factor of any string. An *occurrence* of a factor u in a string w is a pair of positions (i, j) such that $1 \leq i, j \leq n$ and $u = w[i..j]$.

We recall the definition of closed string given in [10]

Definition 1. *We say that w is closed if and only if it is empty or has a factor $v \neq w$ occurring exactly twice in w, as a prefix and as a suffix of w.*

The string *aba* is closed, since its factor *a* appears only as a prefix and as a suffix. On the contrary, a string *abaa* is not closed. Note that for any letter $a \in \Sigma$ and for any $n > 0$, the string a^n is closed, as a^{n-1} is a factor that occurs only as a prefix and a suffix in it. More generally, any string that is a power is closed.

The concept of closed string is equivalent to that of *periodic-like* string [7]. A string w is periodic-like if its longest repeated prefix does not appear in w followed by different letters. The concept of closed string is also related to *complete return* to a factor u in w, as considered in [11]. A complete return to u in w is any factor of w having exactly two occurrences of u, one as a prefix and one as a suffix. Therefore, w is a closed string if and only if w is a complete return to its longest repeated prefix.

A factor of w that is closed is called a *closed factor* of w. For any string w of length n, the number of closed factors of w is at least $n + 1$. In particular, a string w of length n is called *C-poor* if it has exactly $n + 1$ closed factors, i.e., if it contains the smallest number of closed factors a string of length n can contain. For example, the string *abca* of length 4 is a C-poor as it has 5 closed factors

namely ϵ, a, b, c and $abca$, whereas the string $ababa$ of length 5 is not C-poor as it has 8 closed factors: $\epsilon, a, b, aba, abab, bab, baba$, and $ababa$. Note that there are C-poor strings that are not closed, e.g. ab, and closed strings that are not C-poor, e.g. $abab$.

A set of subintervals $\{[b_1, e_1], \cdots, [b_h, e_h]\}$ is called a *cover* of interval $[1, n]$ if $\bigcup_{i=1}^{h} [b_i, e_i] = [1, n]$. The size of the cover is the number h of subintervals in it. A cover $\{[b_1, e_1], ..., [b_h, e_h]\}$ of $[1, n]$ is said to be a *closed cover* of string w of length n, if $w[b_i..e_i]$ is a closed factor of w for all $1 \leq i \leq h$. A *minimum closed cover* of w is a closed cover of w with the smallest possible size; note that a minimum closed cover is not unique.

2.1 Tools

Suffix Array: The *Suffix Array* of the string w is a data structure used for indexing its content. It is comprised of two arrays:

- The array SA which stores the list of positions of w associated with its suffixes in increasing lexicographic order. Thus SA is indexes by the ranks of suffixes in their sorted list.
- The second array, LCP, which stores the longest common prefixes between consecutive suffixes in the sorted list. That is

$$\text{LCP}[r] = |lcp(\text{SA}[r-1], \text{SA}[r])|,$$

  where $lcp(i, j)$ is the longest common prefix of the two suffixes of w starting at positions i and j, i.e. $w[i..n]$ and $w[j..n]$.

The computation of the suffix array can be done in time $O(n \log n)$ in the comparison model [21]. For an integer alphabet, the suffix array can be built in time $O(n)$ [16–18].

Longest Previous Factor: The *Longest Previous Factor* array has been introduced in [8] and gives for each position i in a given string w the length of the longest factor that occurs both at position i and to the left of i in w. The Longest Previous Factor is defined by $\text{LPF}[1] = 0$ and for $1 < i \leq n$, by

$\text{LPF}[i] = \max \{k \mid w[i..i+k-1] = w[j..j+k-1], \text{for some } 1 \leq j < i\}$.

For example, for string $\underline{ab}b\underline{aba}$, the Longest Previous factor corresponding to position 4 is 2 because ab is the longest factor at position 4 that appears before (at position 1). In the paper we make use of the following known result on LPF from [9]:

Lemma 1. *The Longest Previous Factor array of a string of length n on an integer alphabet can be built from the read-write Suffix Array and Longest Common Prefix array in time $O(n)$ (independently of the alphabet size) with a constant amount of extra memory space.*

The Longest Previous Factor algorithm can compute for no extra cost an array FPOS that stores the positions of the longest previous factor if such factor exists, that is

FPOS[i] = 0 if LPF[i] = 0 and

FPOS[i] = max $\{j \mid w[i..i+\text{LPF}[i]-1] = w[j..j+\text{LPF}[i]-1]$, for $1 \leq j < i\}$.

Range Min in Weighted Tree: For any two nodes u and v in the same path of a weighted rooted tree, let $min(u, v)$ be a query that returns a node in the path with minimum weight. The following lemma from [1] states recent result used in this paper:

Lemma 2. *Under a RAM model, a dynamic tree can be maintained in linear space so that a min query and an operation of adding a leaf to the tree are both supported in the worst-case $O(1)$ time.*

3 Minimum Closed Covers

In this section, we present an algorithm that computes Minimum Closed Covers of all prefixes for a given string w of length n. More precisely, we compute array C such that for each position $1 \leq i \leq n$, C[i] stores the size of a minimum closed cover of $w[1..i]$. If $C[i] = h$ then, by definition, there is a closed cover $\{[b_1..e_1], \cdots, [b_h..e_h]\}$ that covers $w[1..i]$ such that $b_1 = 1$, $e_h = i$ and h is minimum.

Obviously there might be several closed covers that are minimum but we are interested on being able to retrieve one of them. In order to achieve this goal, array C_ℓ is maintained such that $C_\ell[i]$ stores the length of the rightmost closed factor ending at position i which is associated with the rightmost subinterval of a minimum closed cover (of size $C[i]$) of $w[1..i]$. A minimum closed cover of w, and indeed of any prefix of w, can be easily computed in a greedy manner from right to left using C_ℓ.

Before presenting our algorithm, we will introduce several known results concerning closed strings; for proof and more details please refer to [4]:

Lemma 3. *For any strings u and v, one has $|CC(u)|+|CC(v)| \leq |CC(uv)|+1$, where $|CC(w)|$ is the size of a set of closed factors of w.*

Corollary 1. *A string w is C-poor if and only if every closed factor of w is a complete return of a single letter.*

By definition empty strings and one-letter strings are closed. Corollary 1 suggests an additional class of closed strings namely complete return of a single letter; that is for any letter α, string $w = \alpha u \alpha$ is closed if α has no occurrence in string u. Additionally, Lemma 3 suggests that if the size of a string increases by one (let say, by concatenating a single letter) then the total number of closed factors of the new strings should increase. In this paper, we are interested with the minimum number of closed factors that cover a given string. Our main observation suggests that if the size of string increases by one letter then the minimum size of closed cover cannot increase by more than one.

Observation 1. *Let* $\{[b_1..e_1], \cdots, [b_k..e_k]\}$ *be a minimum closed cover* $w[1..i-1]$ *and* $\{[b'_1..e'_1], \cdots, [b'_h..'e_h]\}$ *is a minimum closed covers of* $w[1..i1]$ *then* $h \leq k+1$.

Considering position i, given $C[j]$ for $1 \leq j < i$, while taking $C[i-1]$ into consideration, there are three different possibilities:

1. The size of a minimum closed cover of $w[1..i]$ increases by one. In this case, the one-letter closed factor $w[i]$ becomes the rightmost closed factor of a minimum closed cover.
2. The size of a minimum closed cover of $w[1..i]$ stays the same. This may be achieved if the rightmost closed factor can be extended to the right by letter $w[i]$ and stays closed. If $w[b..i-1]$ is the rightmost closed factor of a minimum closed cover of $w[1..i-1]$ and $w[b..i]$ is closed then $w[b..i]$ is the rightmost closed factor of a minimum closed cover of $w[1..i]$.
3. The size of a minimum closed cover of $w[1..i]$ decreases. This can only be achieved if there exists a position k such that $w[k..i]$ is a closed factor of $w[1..i]$ and $C[k-1] < C[i-1]$. Here $C[i] = C[k-1]+1$ and $w[k..i]$ will be the rightmost closed factor of a minimum closed cover of $w[1..i]$.

Recall that $C[i]$ and $C_\ell[i]$ are the size of a minimum closed cover of $w[1..i]$ and the length of its rightmost closed subinterval, respectively. That is, if $C_\ell[i] = \ell$ then $w[i-\ell+1..i]$ is the rightmost closed factor associated with interval $[i-\ell+1..i]$; the rightmost subinterval of a minimum closed cover of $w[1..i]$. For each position i we will store the length of the border of the rightmost closed factor such that $B_\ell[i] = |border(w[i-\ell+1..i])|$. In this way, array B_ℓ will enable us to check in $O(1)$ time whether the rightmost closed factor can be extended to the right by a single letter and stay closed. We will also maintain an array P that keeps for each position i, the nearest position j to the left of i containing letter $w[i]$ such that $w[j..i]$ is closed as it is a complete return of a single letter (Corollary 1).

Now we are ready to present our Minimum Closed Covers algorithm. The algorithm initializes $C[1] = 1$ and iterates from left to right calculating for each position i the value of $C[i]$ as follows:

$$C[i] = \begin{cases} C[i-1]. & \text{If } \ w[i - C_\ell[i-1] + B_\ell[i-1]] = w[i]. \\ \min \begin{cases} C[P[i]-1]+1, \\ C[i-1]+1, \\ C[k-1]+1. \end{cases} & \text{Otherwise.} \end{cases}$$

At each iteration i, the algorithms first checks whether the rightmost closed factor can be extended to the right by letter $w[i]$ and stays closed. This can be easily achieved by comparing positions $w[i - C_\ell[i-1] + B_\ell[i-1]]$ and $w[i]$. If the two positions are equal then the size of a minimum closed cover does not increase, as we are still able to cover $w[1..i]$ with a minimum cover of size $C[i-1]$, and since $C[i-1]$ is minimum, then $C[i]$ is also minimum. If the rightmost closed

factor cannot be extended, then the algorithm checks all possible closed factors of $w[1..i]$, ending at position i, and chooses the one that minimizes the size of a closed cover. Obviously, there are two closed factors of $w[1..i]$ ending at position i that can be easily checked: factor $w[i]$; a a one-letter closed factor, and factor $w[P[i]..i]$; a complete return of a single letter. These two factors are not the only ones that need to be checked and now we will turn our attention to explain how all other closed factors can be identified and checked efficiently.

Recall that the Longest Previous Factor gives for each position i in w the length of the longest factor that occurs at i and to the left of i in w. We show that the position of the longest previous factor can be calculated at no extra cost. Here, we update the definition of LPF as follows: $LPF[1] = (0, 0)$ and

$$LPF[i] = (\ell, j) \text{ such that } \ell = \max\{k \mid w[i..i+k-1] = w[j..j+k-1], 1 \le j < i\}.$$

Clearly, if $LPF[i] = (\ell, j)$, then $w[j..i+\ell-1]$ is a closed factor with a border (of length ℓ) starting at positions j and i. For the purpose of our algorithm, we want to calculate for each position the maximum closed factor ending at a certain position (not starting). By maximum, we mean non left-expendable. To do so, the longest previous factor array is computed for string $\overleftarrow{w[1..n]} = w[n]w[n-1]..w[1]$; we will call the new array $\overleftarrow{LPF}$. By doing so, we calculate for each position the length of the maximum closed factor ending at each position $n-i+1$ and to the right of $n-i+1$ in w. Thus, if $\overleftarrow{LPF}[i] = (\ell, j)$, then there is a closed factor in w with a left border (of length ℓ), ending at position $n-i+1$, and a right border ending at position $n-j+1$. The relation between closed factors and array $\overleftarrow{LPF}$ has been highlighted in [3].

Example 1. Let $w = aabaaaaabaaaabcdbcd$ be a given string of length 19.

position i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$w[i]$	a	a	b	a	a	a	a	a	b	a	a	a	a	b	c	d	b	c	d
$LPF[i](\underline{\ell}, j)$	0	1	0	2	4	3	7	6	5	5	4	3	②	1	0	0	3	2	1
$LPF[i](\ell, \underline{j})$	0	1	0	1	4	5	1	2	3	5	6	7	⑧	9	0	0	14	15	16

For $i = 13$, $LPF[13] = (2, 8)$ implies that there is a closed factor $\underline{ab}aaaa\underline{ab}$, with a left border starting at position 8, and a right border starting at position 13 in w. The length of the border of this closed factor is 2.
The inverse of string w, $\overleftarrow{w} = dcbdcbaaaabaaaaabaa$.

position i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\overleftarrow{w}[i]$	d	c	b	d	c	b	a	a	a	a	b	a	a	a	a	a	b	a	a
$\overleftarrow{LPF}[i](\underline{\ell}, j)$	0	0	0	3	2	1	0	3	2	1	⑤	4	7	6	5	4	3	2	1
$\overleftarrow{LPF}[i](\ell, \underline{j})$	0	0	0	1	2	3	0	**7**	8	9	⑥	**7**	**7**	8	9	10	11	12	13

For $i = 11$, $\overleftarrow{LPF}[11] = (5, 6)$ implies that there is a closed factor $\underline{baaaa}\ \underline{baaaa}$ in $\overleftarrow{w}$, that corresponds to a closed factor ($\underline{aaaab}\ \underline{aaaab}$) in w, with a left border ending at position $9 = 19 - 11 + 1$, and a right border ending at position $14 = 19 - 6 + 1$. This is a maximum closed factor as it cannot be extended to the left.

Note that position 7 appears three times in $\overleftarrow{LPF}$, which implies that there are three different closed factors with a right border ending at position $13 = 19-7+1$ in w. These three closed factors are different and each is associated with distinct maximum closed factor with respect to the end positions of the left borders:

- left border ending at position $12 = 19 - 8 + 1$ and of length $= 3$: $\underline{a\overline{aaa}}$,
- left border ending at position $8 = 19 - 12 + 1$ and of length $= 4$: $\underline{\underline{aaaa}b\underline{aaaa}}$,
- left border ending at position $7 = 19-13+1$ and of length $= 7$: $aabaaa\underline{aabaaaa}$.

From the above example, one should be able to recognize that the number of distinct maximum closed factors ending at a certain position i in w is not always equal to 1. However, the total number of distinct maximum closed factors over all positions i in w is at most n; the length of w.

Now, we can go back and finalize the Minimum Closed Covers algorithm. Recall that in order to calculate $C[i]$, the algorithm checks whether the rightmost factor can be extending with letter $w[i]$ and remain closed. If this not possible, the algorithm finds the rightmost closed factor ending at position i that minimizes $C[i]$.

Lemma 4. *A factor $w[k..i]$ is a closed factor of w ending at position i if and only if $k = i$ or $k = P[i]$ or*

$$k \in \bigcup [j_i - \ell_{j_i} + 1..j_i - \ell_{j_{i+1}}],$$

where $\{(j_1, \ell_{j_1}), (j_2, \ell_{j_2}), \cdots, (j_h, \ell_{j_h})\}$ is the ordered set of all maximal closed factors of w ending at position i, and with left borders ending at positions j_i and of length ℓ_{j_i} ($j_{h+1} = 0$).

Proof. If $\{(j_1, \ell_{j_1}), (j_2, \ell_{j_2}), \cdots, (j_h, \ell_{j_h})\}$ is the set of closed factors ending at position i in ascending order, i.e. $j_1 < j_2 < ... < j_h$. Then, it should be clear that $\ell_{j_1} > \ell_{j_2} > \cdots > \ell_{j_h}$.

Consider for example positions $j_1 < j_2$, both $w[j_1 - \ell_{j_1} + 1..i]$ and $w[j_2 - \ell_{j_2} + 1..i]$ are closed factors of w ending at position i with borders of length ℓ_{j_1} and ℓ_{j_2}, respectively. By definition, $w[i - \ell_{j_1} + 1i]$ and $w[i - \ell_{j_2} + 1i]$ are the longest suffixes associated with these closed factors. Each suffix appears in the associated factor twice as a suffix and a prefix. If $\ell_{j_1} < \ell_{j_2}$, then $w[i - \ell_{j_1} + 1i]$ is a suffix of $w[i - \ell_{j_2} + 1i]$ and should have internal occurrence in $w[j_1 - \ell_{j_1} + 1..i]$, a contradiction.

Each closed factor $w[j_l - \ell_{j_l} + 1..i]$ is maximum i.e. it cannot be extended to the left by any letter. This is due to the method of construction from LPF. However, in order to calculate all possible positions in w that could start a closed factor ending at position i, non-maximal closed factors need to be considered. This can be done by considering all suffixes of $w[j_l - \ell_{j_l} + 1..i]$ starting at positions $j_l - \ell_{j_l} + 1..j_l - \ell_{j_{l+1}}$. Note that any smaller suffix is not closed due to an internal occurrence. □

Each position k defined by Lemma 4 is a candidate for a start position of a closed factor ending at position i. The closed factor that minimizes $C[i]$ is the one that starts at position k such that $C[k-1]$ is smallest.

Theorem 2. *Given string w of length n, the Minimum Closed Covers algorithm computes $C[i]$ for $1 \leq i \leq n$ in $O(n)$-time in an online manner.*

Proof. Although for each position i, the number of candidate positions at which a rightmost closed factor (ending at position i) is not constant, the number of intervals that need to be checked is bounded by n. The sum of these intervals over all possible i is linear. According to Lemma 2, calculating for each interval, the position k, with the minimum $C[k]$, can be done in $O(1)$ time. □

Example 2. For the same string $w = aabaaaaabaaaabcdbcd$.

position i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$w[i]$	a	a	b	a	a	a	a	a	b	a	a	a	a	b	c	d	b	c	d
$C[i]$	1	1	2	2	1	2	2	2	1	1	1	1	1	2	3	4	2	2	2
$C_\ell[i]$	1	2	1	3	5	1	2	3	9	10	11	12	13	9	1	1	4	5	6

Consider position $i = 14$, given that $\overline{aabaaa\underline{aabaaaa}}$ is the rightmost closed factor of a minimum closed cover of $w[1..13]$. Firstly, the Minimum Closed Covers algorithm checks whether this closed factor is extendable to the right by comparing positions $w[14]$ and $w[8]$. Since $w[14] \neq w[8]$, the closed factor cannot be extended to the right and the algorithm proceeds to compute the optimal value for $C[14]$ as follows:

$$C[14] = \min \begin{cases} C[P[14]-1]+1 = C[8]+1 = 2+1 = 3, \\ C[13]+1 = 1+1 = 2, \\ \min\{C[k-1]+1; \quad \text{for } k \in [5..9]\} = C[5]+1 = 1+1 = 2. \end{cases}$$

In *Example1*, we show that there is one maximum closed factor ending at position 14 that starts at position 5 and with a border of length 5. Although, all five suffixes (starting at positions $\in [5..9]$) of this maximum closed factor are also closed and need to be considered, computing position $k \in [5..9]$ with the minimum $C[i-1]$ can be done constant time using a single min query ($k = 6$). Therefore, $C[14] = 2$ is the size of a minimum closed cover of $w[1..14]$; note that there are two minimum closed covers: $\{[1..13], [14..14]\}$ and $\{[1..5], [6..14]\}$, both of size 2.

The size of a minimum closed cover of w calculated by our proposed algorithm is $C[19] = 2$. It consists of two closed factors that can be retrieved using array C_ℓ. In this case, $\{[1..13][14..19]\}$ is a minimum closed cover with two closed factors: *aabaaaaabaaaa* and *bcdbcd*.

4 Conclusion

We have considered The Minimum Closed Covers problem and proposed an on-line algorithm to calculate such cover in linear time. Our work is an extension to various string cover related problems. Recently, an algorithm to compute the minimum palindromic covers was proposed. Although there are relatively strong relations between palindromic string and closed string, our algorithm is significantly different. Various types of covers are still to be investigated mainly for its theocratical interest. In particular, we are interested in the calculation of the minimum Abelian covers of a string.

References

1. Alstrup, S., Holm, J.: Improved Algorithms for Finding Level Ancestors in Dynamic Trees. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 73–84. Springer, Heidelberg (2000)
2. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal Superprimitivity Testing for Strings. Inform. Processing Letter 39, 17–20 (1991)
3. Badkobeh, G., Bannai, H., Goto, K.: I, Tomohiro, Iliopoulos, C. S., Inenaga, S., Puglisi, S.J., Sugimoto, S.: Closed Factorization (to appear)
4. Badkobeh, G., Fici, G., Lipták, Z.: A Note on Words With the Smallest Number of Closed Factors. CoRR abs/1305.6395 (2013)
5. Breslauer, D.: An On-Line String Superprimitivity Test. Information Processing Letters 44, 345–347 (1992)
6. Bucci, M., de Luca, A., De Luca, A.: Rich and Periodic-Like Words. In: Diekert, V., Nowotka, D. (eds.) DLT 2009. LNCS, vol. 5583, pp. 145–155. Springer, Heidelberg (2009)
7. Carpi, A., de Luca, A.: Periodic-Like Words, Periodicity and Boxes. Acta Informatica 37, 597–618 (2001)
8. Crochemore, M., Ilie, L.: Computing Longest Previous Factor in Linear Time and Applications. Inf. Process. Lett. 106(2), 75–80 (2008)
9. Crochemore, M., Ilie, L., Iliopoulos, C.S., Kubica, M., Rytter, W., Walen, T.: Computing the Longest Previous Factor. Eur. J. Comb. 34(1), 15–26 (2013)
10. Fici, G.: A Classification of Trapezoidal Words. In: Ambroz, P., Holub, S., Masakova, Z. (eds.) 8th International Conference on Words, WORDS 2011. Electronic Proceedings in Theoretical Computer Science, vol. 63, pp. 129–137 (2011)
11. Glen, A., Justin, J., Widmer, S., Zamboni, L.Q.: Palindromic Richness. European J. Combin. 30, 510–531 (2009)
12. Guo, Q., Zhang, H., Iliopoulos, C.S.: Computing the λ-Covers of a String. Inf. Sci. 177(19), 3957–3967 (2007)
13. Iliopoulos, C.S., Mohamed, M., Smyth, W.F.: New Complexity Results for the k-Covers Problem. International Journal of Information Sciences 181, 2571–2575 (2011)
14. Iliopoulos, C.S., Mouchard, L.: An $O(n \log n)$ Algorithm for Computing All Maximal Quasiperiodicities in Strings. Theoratical Computer Science 119, 247–265 (1993)
15. Iliopoulos, C.S., Smyth, W.F.: On-Line Algorithms for k-Covering. In: Proc. Ninth Australasian Workshop on Combinatorial Algorithms, pp. 107–116 (1998)

16. Kärkkäinen, J., Sanders, P.: Simpler Linear Work Suffix Array Construction. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 943–955. Springer, Heidelberg (2003)
17. Kim, D.-K., Sim, J.S., Park, H.-J., Park, K.: Linear-Time Construction of Suffix Arrays. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 186–199. Springer, Heidelberg (2003)
18. Ko, P., Aluru, S.: Space Efficient Linear Time Construction of Suffix Arrays. Journal of Discrete Algorithms 3(24), 143–156 (2005)
19. Li, Y., Smyth, W.F.: Computing the Cover Array in Linear Time. Algorithmica 32, 95–106 (2002)
20. Main, M.G., Lorentz, R.J.: An $O(n \log n)$ Algorithm for Finding All Repetitions in a String. J. Algs. 5, 422–432 (1984)
21. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line Search. SIAM J. on Computing, 935–948 (1993)
22. Moore, D., Smyth, W.F.: An Optimal Algorithm to Compute All the Covers of a String. Information Processing Letters 50, 239–246 (1994)
23. Thue, A.: Über Unendliche Zeichenreihen. Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana 7, 1–22 (1906)
24. Tomohiro, I., Sugimoto, S., Inenaga, S., Bannai, H., Takeda, M.: Computing palindromic factorizations and palindromic covers on-line. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P. (eds.) CPM 2014. LNCS, vol. 8486, pp. 150–161. Springer, Heidelberg (2014)