Efficient seed computation revisited

M. Christou [a], M. Crochemore [a,b], C.S. Iliopoulos [a,c], M. Kubica [d], S.P. Pissis [a], J. Radoszewski [d,*],
W. Rytter [d,e], B. Szreder [d], T. Waleń [d,f]

[a] *King's College London, London WC2R 2LS, UK*

[b] *Université Paris-Est, France*

[c] *Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia*

[d] *Department of Mathematics, Informatics and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warsaw, Poland*

[e] *Department of Math. and Informatics, Copernicus University, ul. Chopina 12/18, 87-100 Toruń, Poland*

[f] *Laboratory of Bioinformatics and Protein Engineering, International Institute of Molecular and Cell Biology in Warsaw, Poland*

ARTICLE INFO

ABSTRACT

The notion of the cover is a generalization of a period of a string, and there are linear time algorithms for finding the shortest cover. The seed is a more complicated generalization of periodicity, it is a cover of a superstring of a given string, and the shortest seed problem is of much higher algorithmic difficulty. The problem is not well understood, no linear time algorithm is known. In the paper we give linear time algorithms for some of its versions—computing shortest left-seed array, longest left-seed array and checking for seeds of a given length. The algorithm for the last problem is used to compute the seed array of a string (i.e., the shortest seeds for all the prefixes of the string) in $O(n^2)$ time. We describe also a simpler alternative algorithm computing efficiently the shortest seeds. As a by-product we obtain an $O(n \log(n/m))$ time algorithm checking if the shortest seed has length at least m and finding the corresponding seed. We also correct some important details missing in the previously known shortest-seed algorithm Iliopoulos et al. (1996) [14].

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

The notion of periodicity in strings is widely used in many fields, such as combinatorics on words, pattern matching, data compression and automata theory (see [18,19]). It is of paramount importance in several applications, not to talk about its theoretical aspects. The concept of quasiperiodicity is a generalization of the notion of periodicity, and was defined by Apostolico and Ehrenfeucht in [3]. In a periodic repetition the occurrences of the period do not overlap. In contrast, the quasiperiods of a quasiperiodic string may overlap. Quasiperiodic properties of strings enable detecting repetitive structure when it cannot be discovered using the classical periodicity. This gives possible applications of quasiperiodicity in DNA sequence analysis and text compression.

Let us first recall some basic terminology related to strings (words) and periodicities. By Σ^* we denote the set of all strings over a finite alphabet Σ including the empty word. The positions in $u \in \Sigma^*$ are numbered from 1 to $|u|$. By Σ^n we denote the set of strings of length n over the alphabet Σ. For $u = u_1 u_2 \ldots u_n$, let us denote by $u[i \mathinner{..} j]$ a factor of u equal to $u_i \ldots u_j$ (in particular $u[i] = u[i \mathinner{..} i]$). Strings $u[1 \mathinner{..} i]$ are called *prefixes* of u, and strings $u[i \mathinner{..} n]$ are called *suffixes* of u. Strings that are both prefixes and suffixes of u are called *borders* of u. By border(u) we denote the length of the longest border of u

---

* Corresponding author. Tel.: +48 22 55 44 484; fax: +48 22 55 44 400.

E-mail addresses: michalis.christou@dcs.kcl.ac.uk (M. Christou), maxime.crochemore@kcl.ac.uk (M. Crochemore), csi@dcs.kcl.ac.uk (C.S. Iliopoulos), kubica@mimuw.edu.pl (M. Kubica), solon.pissis@dcs.kcl.ac.uk (S.P. Pissis), jrad@mimuw.edu.pl (J. Radoszewski), rytter@mimuw.edu.pl (W. Rytter), szreder@mimuw.edu.pl (B. Szreder), walen@mimuw.edu.pl (T. Waleń).
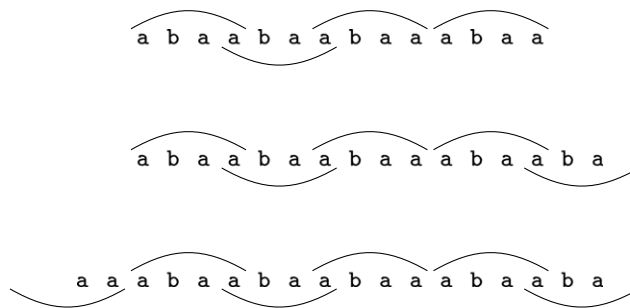
Fig. 1. The string abaa is the shortest cover of abaabaabaaabaa, the shortest left seed of abaabaabaaabaaba, and one of the shortest seeds of the string aaabaabaabaaabaaba (the other shortest seed is aaba).

Table 1
An example string together with its periodic and quasiperiodic arrays. Note that the left-seed array and the seed array are non-decreasing, see also Observation 2.2.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$u[i]$	a	b	a	a	b	a	a	a	b	b	a	a	b	a	a	b
$P[i]$	1	2	2	3	3	3	3	7	7	10	10	11	11	11	11	11
$P'[i]$	11	11	11	11	11	11	7	7	7	3	3	3	3	3	2	1
$B[i]$	0	0	1	1	2	3	4	1	2	0	1	1	2	3	4	5
$C[i]$	1	2	3	4	5	3	4	8	9	10	11	12	13	14	15	16
$C^M[i]$	0	0	0	0	0	3	4	0	0	0	0	0	0	0	0	0
$\mathsf{LSeed}[i]$	1	2	2	3	3	3	3	4	4	10	10	11	11	11	11	11
$\mathsf{LSeed}^M[i]$	0	0	2	3	4	5	6	7	8	0	10	11	12	13	14	15
$\mathsf{Seed}[i]$	1	2	2	3	3	3	3	4	4	8	8	8	8	8	8	11

that is shorter than u. We say that a positive integer p is the (shortest) *period* of a string $u = u_1 \ldots u_n$ (notation: $p = \mathrm{per}(u)$) if p is the smallest positive number, such that $u_i = u_{i+p}$, for $i = 1, \ldots, n - p$. It is a known fact [9,10] that, for any string u, $\mathrm{per}(u) + \mathrm{border}(u) = |u|$.

We say that a string s *covers* the string u if every occurrence of a letter of u is contained in some occurrence of s as a factor of u. Then s is called a *cover* of u. We say that a string s is: a *seed* of u if s is a factor of u and u is a factor of some string w covered by s; a *left seed* of u if s is both a prefix and a seed of u; a *right seed* of u if s is both a suffix and a seed of u (equivalently, the reverse of s is a left seed of the reverse of u), see Fig. 1. Thus, a left seed and a right seed is a generalization of a cover and a seed is a generalization of both of these notions. Seeds were first defined and studied by Iliopoulos et al. [14], who gave an $O(n \log n)$ time algorithm computing all the seeds of a given string u of length n, in particular, the shortest seed of u.

By $\mathrm{cover}(u)$, $\mathrm{seed}(u)$, $\mathrm{lseed}(u)$ and $\mathrm{rseed}(u)$ we denote the length of the shortest: cover, seed, left seed and right seed of u, respectively. By $\mathrm{covermax}(u)$ and $\mathrm{lseedmax}(u)$ we denote the length of the longest cover and the longest left seed of u that is shorter than u, or 0 if no such cover or left seed exists.

For a string $u \in \Sigma^n$, we define its: *period array* $P[1 \ldots n]$, *border array* $B[1 \ldots n]$, *suffix period array* $P'[1 \ldots n]$, *cover array* $C[1 \ldots n]$, *longest cover array* $C^M[1 \ldots n]$, *seed array* $\mathsf{Seed}[1 \ldots n]$, *left-seed array* $\mathsf{LSeed}[1 \ldots n]$, and *longest left-seed array* $\mathsf{LSeed}^M[1 \ldots n]$ as follows (see also Table 1):

$$
\begin{array}{llllll}
P[i] & = & \mathrm{per}(u[1 \ldots i]), & B[i] & = & \mathrm{border}(u[1 \ldots i]), \\
P'[i] & = & \mathrm{per}(u[i \ldots n]), & C[i] & = & \mathrm{cover}(u[1 \ldots i]), \\
C^M[i] & = & \mathrm{covermax}(u[1 \ldots i]), & \mathsf{Seed}[i] & = & \mathrm{seed}(u[1 \ldots i]), \\
\mathsf{LSeed}[i] & = & \mathrm{lseed}(u[1 \ldots i]), & \mathsf{LSeed}^M[i] & = & \mathrm{lseedmax}(u[1 \ldots i]).
\end{array}
$$

The border array, suffix border array and period array can be computed in $O(n)$ time [9,10]. A linear time algorithm finding the shortest cover of a string was proposed by Apostolico et al. [4], and a linear time algorithm computing all the covers of a string was proposed by Moore and Smyth [20]. Breslauer [6] gave an on-line $O(n)$ time algorithm computing the cover array $C[1 \ldots n]$ of a string of length n, and Li and Smyth [17] provided a linear time algorithm for computing the longest cover array $C^M[1 \ldots n]$ of a string. Note that the array C^M enables computing all covers of all prefixes of the string, the same property holds for the border array B. Unfortunately, the LSeed^M array does not share this property.

Example 1.1. Table 1 shows the above defined arrays for $u = $ abaabaaabbaabaab. For example, for the prefix $u[1 \ldots 13]$ the period equals 11, the border is ab, the cover is abaabaaabbaab, the left seed is abaabaaabba, the longest left seed is abaabaaabbaa, and the seed is baabaaab.

Several new and efficient algorithms related to seeds in strings are presented in this paper. Linear time algorithms computing left-seed array and longest left-seed array are given in Section 3. In Section 4 we show a linear time algorithm finding seed-of-a-given-length and apply it to computing the seed array of a string in $O(n^2)$ time. In Section 5 we present
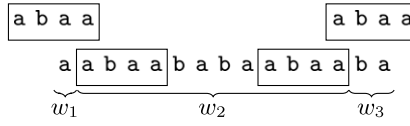
Fig. 2. The string $s =$ abaa is a border seed of $u =$ aabaababaabaaba.

an efficient algorithm testing, for a given string u, which strings from the given set S are quasiperiods of different kinds in u (seeds, left/right seeds, covers). Finally, in Section 6 we describe an alternative simple $O(n \log n)$ time computation of the shortest seed, from which we obtain an $O(n \log (n/m))$ time algorithm checking if the shortest seed has length at least m (described in Section 7). This is a full version of the paper [8].

2. Basic properties of seeds and covers

In this section we list several useful properties of covers and seeds. We start with the following simple observation. We omit its proof since the statements are obvious consequences of definitions of notions of quasiperiodicity.

Observation 2.1. *Let $u \in \Sigma^*$.*

(a) *A cover of a cover of u is also a cover of u.*
(b) *A cover of a left (right) seed of u is also a left (right) seed of u.*
(c) *A cover of a seed of u is also a seed of u.*
(d) *If s and s' are two covers of u, $|s'| < |s|$, then s' is a cover of s.*

The following observation implies, in particular, that for any string $u \in \Sigma^n$, its seed array $\mathsf{Seed}[1 . . n]$ and its left seed array $\mathsf{LSeed}[1 . . n]$ are non-decreasing.

Observation 2.2.

(a) *If u is a prefix of v then $lseed(u) \le lseed(v)$.*
(b) *If u is a factor of v then $seed(u) \le seed(v)$.*

Proof. Part (a) is obvious. As for part (b), let s be the shortest seed of the string v. If $|s| \ge |u|$ then the conclusion holds. In the opposite case, note that u is a factor of some string covered by s. Now consider two cases. If s is a factor of u then s is a seed of u. Otherwise, by theorem 1 from [14], there exists a seed s' of u which is either a factor of s or a cyclic rotation of s or a cyclic rotation of a factor of s. Since $|s'| \le |s|$, the conclusion of the observation holds. $\square$

For a set X of positive integers, let us define the *maxgap* of X as:

$$maxgap(X) = \max\{b - a : a, b \text{ are consecutive numbers in } X\} \quad \text{or} \quad 0 \text{ if } |X| \le 1.$$

For example $maxgap(\{1, 3, 8, 13, 17\}) = 5$.

For a factor v of u, let us define $Occ(v, u)$ as the set of starting positions of all occurrences of v in u. By $first(v, u)$ and $last(v, u)$ we denote $\min Occ(v, u)$ and $\max Occ(v, u)$ respectively. If the string u is fixed, we use a shorter notation $Occ(v)$, $first(v)$, and $last(v)$. For the sake of simplicity, we will abuse notation, and denote $maxgap(v) = maxgap(Occ(v))$.

Assume s is a factor of u. Let us decompose the string u into $w_1 w_2 w_3$, where w_2 is the longest factor of u for which s is a border, i.e.,

$$w_2 = u[first(s) . . (last(s) + |s| - 1)].$$

Then we say that s is a *border seed* of u if s is a seed of $w_1 \cdot s \cdot w_3$, see Fig. 2. It can be observed that s is a border seed of u if and only if:

$$|s| \ge \max(\mathsf{P}[first(s) + |s| - 1], \mathsf{P}'[last(s)])$$

where $\mathsf{P}[1 . . n]$ and $\mathsf{P}'[1 . . n]$ are the period array and the suffix period array of u. This inequality is proved as Fact 3.2 in the next section.

The notions of maxgaps and border seeds provide a useful characterization of seeds which follows from preceding definitions.

Observation 2.3. *Let s be a factor of $u \in \Sigma^*$. The string s is a seed of u if and only if $|s| \ge maxgap(s)$ and s is a border seed of u.*

Example 2.4. The string $s =$ abaa is one of the shortest seeds of the string $u =$ aaabaabaabaaabaaba. Indeed, we have $Occ(s, u) = \{3, 6, 9, 13\}$, hence $maxgap(s) = 4 \le |s|$. The string u can be decomposed as: $u =$ aa $\cdot w_2 \cdot$ ba, where w_2 admits a border s, and by the maxgap condition, w_2 is covered by s. We see that s is a seed of aa $\cdot s \cdot$ ba $=$ aaabaaba, hence s is a border seed of u. Thus s satisfies the conditions from Observation 2.3.

3. Computing left-seed arrays

In this section we show two $O(n)$ time algorithms for computing the left-seed array and an $O(n)$ time algorithm for computing the longest left-seed array of a given string $u \in \Sigma^n$. We start by a simple characterization of the length of the
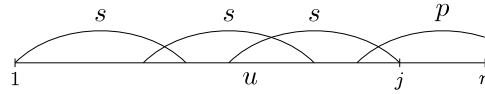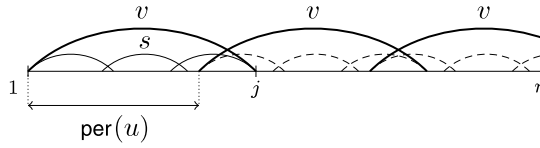
Fig. 3. Illustration of part ($\Rightarrow$) of Lemma 3.1.

Fig. 4. Illustration of part ($\Leftarrow$) of Lemma 3.1.

shortest left seed of the whole string u—see Lemma 3.3. In its proof we utilize the following auxiliary lemma which shows a correspondence between the shortest left seed of u and shortest covers of all prefixes of u.

Lemma 3.1. *Let s be a prefix of u, and let j be the length of the longest prefix of u covered by s. Then s is a left seed of u if and only if $j \geq per(u)$.*

In particular, the shortest left seed s of u is the shortest cover of the corresponding prefix $u[1..j]$.

Proof. ($\Rightarrow$) If s is a left seed of u then there exists a prefix p of s of length at least $n - j$ which is a suffix of u (see Fig. 3). We use here the fact that $u[1..j]$ is the *longest* prefix of u covered by s. Hence, p is a border of u, and consequently border$(u) \geq |p| \geq n - j$. Using the formula border$(u) + per(u) = |u|$, we obtain the desired inequality $j \geq per(u)$.

($\Leftarrow$) The inequality $j \geq per(u)$ implies that $v = u[1..j]$ repeats every $per(u)$ positions, so v is a left seed of u (see Fig. 4). Hence, by Observation 2.1b, the string s, which is a cover of v, is also a left seed of u.

Finally, the "in particular" part is a consequence of Observation 2.1, parts b and d.   □

As a corollary of Lemma 3.1, we obtain the aforementioned crucial property of border seeds.

Fact 3.2. *Let s be a factor of $u \in \Sigma^n$ and let $P[1..n]$ and $P'[1..n]$ be its period array and suffix period array. The string s is a border seed of u if and only if*

$$|s| \geq \max(P[first(s) + |s| - 1], \ P'[last(s)]).$$

Proof. The string s is a border seed of u if and only if s is a right seed of $u[1..first(s) + |s| - 1]$ and a left seed of $u[last(s)..n]$. Hence, by Lemma 3.1, s is a border seed of u if and only if

$$|s| \geq P[first(s) + |s| - 1] \wedge |s| \geq P'[last(s)]$$

which is equivalent to the conclusion of the fact.   □

Lemma 3.3. *Let $u \in \Sigma^n$ and let $C[1..n]$ be its cover array. Then:*

$$\mathsf{lseed}(u) = \min\{C[j] \ : \ j \geq per(u)\}. \tag{1}$$

Proof. By Lemma 3.1, the length of the shortest left seed of u can be found among the values $C[per(u)], \ldots, C[n]$. And conversely, for each of the values $C[j]$ for $per(u) \leq j \leq n$, there exists a left seed of u of length $C[j]$. Thus $\mathsf{lseed}(u)$ equals the minimum of these values, which yields the formula (1).   □

Clearly, the formula (1) provides an $O(n)$ time algorithm for computing the shortest left seed of the whole string u. We show that, employing some algorithmic techniques, one can use this formula to compute shortest left seeds for all prefixes of u, i.e., computing the left-seed array of u, also in $O(n)$ time.

Theorem 3.4. *For $u \in \Sigma^n$, its left-seed array can be computed in $O(n)$ time.*

Proof. Applying (1) to all prefixes of u, we obtain:

$$\mathsf{LSeed}[i] = \min\{C[j] \ : \ P[i] \leq j \leq i\}. \tag{2}$$

Recall that both the period array $P[1..n]$ and the cover array $C[1..n]$ of u can be computed in $O(n)$ time [2,6,9,10].

The minimum in the formula (2) could be computed by data structures for Range-Minimum-Queries [11,13], however in this particular case we can apply a much simpler algorithm. Note that $P[i - 1] \leq P[i]$, therefore the intervals of the form $[P[i], i]$ behave like a sliding window, i.e., both their endpoints are non-decreasing. We use a bidirectional queue Q which stores left-minimal elements in the current interval $[P[i], i]$ with regard to the value $C[j]$. In other words, elements of Q are increasing and if during the step i the queue Q contains an element j then $j \in [P[i], i]$ and $C[j] < C[j']$ for all $j < j' \leq i$. We obtain an $O(n)$ time algorithm ComputeLeftSeedArray.

ALGORITHM ComputeLeftSeedArray(u)

1: P[$1 . . n$] := period array of u;
2: C[$1 . . n$] := cover array of u;
3: Q := *emptyBidirectionalQueue*;
4: **for** i := 1 **to** n **do**
5:    **while** (**not** *empty*(Q)) **and** (*front*(Q) < P[i]) **do** *popFront*(Q);
6:    **while** (**not** *empty*(Q)) **and** (C[*back*(Q)] ≥ C[i]) **do** *popBack*(Q);
7:    *pushBack*(Q, i);
8:    LSeed[i] := C[*front*(Q)];
9:    { Q stores left-minimal elements of the interval [P[i], i] }
10: **return** LSeed[$1 . . n$];

The operations on Q are defined as follows: *front*(Q) and *back*(Q) denote the first and the last element of the queue, *popFront*(Q) and *popBack*(Q) remove a single element from the front and from the back of the queue, *pushBack*(Q, i) adds the element i at the end of the queue, *emptyBidirectionalQueue* creates an empty queue and *empty*(Q) tests if the queue is empty. □

Now we proceed to an alternative algorithm computing the left-seed array, which also utilizes the criterion from Lemma 3.1. We start with an auxiliary algorithm ComputeR-Array. It computes an array R[$1 . . n$] which stores, as R[i], the length of the longest prefix of u for which $u[1 . . i]$ is the shortest cover, 0 if none.

ALGORITHM ComputeR-Array(u)

1: C[$1 . . n$] := cover array of u;
2: **for** i := 1 **to** n **do** R[i] := 0;
3: **for** i := 1 **to** n **do** R[C[i]] := i;
4: **return** R[$1 . . n$];

The algorithm Alternative-ComputeLeftSeedArray computes the array LSeed from left to right using the following observation.

Observation 3.5. *Let* $u \in \Sigma^n$. *For any* $1 \le j \le i \le n$, *if* R[j] ≥ P[i] *then* $u[1 . . j]$ *is a left seed of* $u[1 . . i]$.

Proof. If R[j] ≤ i then the observation is a conclusion of Lemma 3.1. Otherwise, $u[1 . . j]$ is a left seed of $u[1 . . i]$ by definition. □

In the algorithm below, the current value of LSeed is stored in the variable j. By Observation 2.2a, for each i we have LSeed[$i - 1$] ≤ LSeed[i]. Hence, when computing LSeed[i] we can start with j = LSeed[$i - 1$] and keep incrementing this value until the inequality from Observation 3.5 is satisfied.

ALGORITHM Alternative-ComputeLeftSeedArray(u)

1: P[$1 . . n$] := period array of u;
2: R[$1 . . n$] := ComputeR-Array(u);
3: LSeed[0] := 0; j := 0;
4: **for** i := 1 **to** n **do**
5:    { An invariant of the loop: j = LSeed[$i - 1$]. }
6:    **while** R[j] < P[i] **do** j := $j + 1$;
7:    LSeed[i] := j;
8: **return** LSeed[$1 . . n$];

Theorem 3.6. *Algorithm Alternative-ComputeLeftSeedArray runs in linear time.*

Proof. Recall that the arrays P[$1 . . n$] and C[$1 . . n$] can be computed in linear time [2,6,9,10]. The array R[$1 . . n$] is obviously also computed in linear time.

It suffices to prove that the total number of steps of the while-loop in the algorithm Alternative-ComputeLeftSeedArray is linear in terms of n. In each step of the loop, the value of j increases by one; this variable never decreases and it cannot exceed n. Hence, the while-loop performs at most n steps and the whole algorithm runs in $O(n)$ time. □

Concluding this section, we describe a linear-time algorithm computing the longest left-seed array, LSeed^M[$1 . . n$], of the string $u \in \Sigma^n$. The following lemma gives a simple characterization of the length of the longest left seed of the whole string u.

Lemma 3.7. *Let* $u \in \Sigma^n$. *If* $per(u) < n$ *then* lseedmax(u) = $n - 1$, *otherwise* lseedmax(u) = 0.

Proof. First consider the case $per(u) = n$. We show that lseed(u) = n, consequently lseedmax(u) equals 0. Assume to the contrary that lseed(u) < n. Then, a non-empty prefix of the minimal left seed of u, say w, is a suffix of u (consider the occurrence of the left seed that covers $u[n]$). Hence, w is a border of u, consequently $n - |w|$ is a period of u, a contradiction.

Assume now that $\text{per}(u) < n$. Then u is a prefix of the string $u[1 \mathinner{.\,.} \text{per}(u)] \cdot u[1 \mathinner{.\,.} n-1]$ which is covered by $u[1 \mathinner{.\,.} n-1]$. Therefore $u[1 \mathinner{.\,.} n-1]$ is a left seed of u, $\text{lseedmax}(u) \geq n-1$, consequently $\text{lseedmax}(u) = n-1$. $\quad\square$

Using Lemma 3.7 we obtain:

$$\text{LSeed}^M[i] = \begin{cases} i-1 & \text{if } P[i] < i, \\ 0 & \text{otherwise.} \end{cases}$$

This implies the following result.

Theorem 3.8. *The longest left-seed array of $u \in \Sigma^n$ can be computed in $O(n)$ time.*

4. Computing seeds of given length and seed array

In this section we show an $O(n^2)$ time algorithm computing the seed array $\text{Seed}[1 \mathinner{.\,.} n]$ of a given string $u \in \Sigma^n$, note that a trivial approach – computing the shortest seed for every prefix of u – yields $O(n^2 \log n)$ time complexity. In our solution we utilize a subroutine: testing whether u has a seed of a given length k. The following Theorem 4.1 shows that this test can be performed in $O(n)$ time. Before we prove the theorem, let us recall the notion of a suffix array.

The suffix array of the string u basically consists of two tables SUF and LCP. The SUF array stores the list of positions in u sorted according to the increasing lexicographic order of suffixes starting at these positions, i.e.,:

$$u[\text{SUF}[1] \mathinner{.\,.} n] \;<\; u[\text{SUF}[2] \mathinner{.\,.} n] \;<\; \cdots \;<\; u[\text{SUF}[n] \mathinner{.\,.} n].$$

Thus, indices of SUF are ranks of the respective suffixes in increasing lexicographic order. The LCP array is also indexed by the ranks of the suffixes, and stores the lengths of the longest common prefixes of consecutive suffixes in SUF. We set $\text{LCP}[1] = -1$ and, for $1 < i \leq n$, we define $\text{LCP}[i]$ as the longest common prefix of the suffixes $u[\text{SUF}[i-1] \mathinner{.\,.} n]$ and $u[\text{SUF}[i] \mathinner{.\,.} n]$. The tables comprising the suffix array can be constructed in $O(n)$ time [9,15,16].

Theorem 4.1. *It can be checked whether a given string $u \in \Sigma^n$ has a seed of a given length k in $O(n)$ time.*

Proof. Assume we have already computed in $O(n)$ time the arrays SUF and LCP. In the algorithm we start by dividing all factors of u of length k into groups corresponding to equal strings. Every such group can be described as a maximal interval $[i \mathinner{.\,.} j]$ in the suffix array SUF, such that each of the values $\text{LCP}[i+1], \text{LCP}[i+2], \ldots, \text{LCP}[j]$ is at least k. The collection of such intervals can be constructed in $O(n)$ time by a single traversal of the LCP and SUF arrays (lines 1–11 of Algorithm SeedsOfAGivenLength). Moreover, using the Bucket Sort, we can transform this representation into a collection of lists, each of which describes the set $Occ(v, u)$ for some factor v of u, $v \in \Sigma^k$ (lines 12–13 of the algorithm). This can be done in linear time, provided that we use the same set of buckets in each sorting and initialize them just once.

Now we process each of the lists separately, checking the conditions from Observation 2.3: in lines 16–20 of the algorithm we check the "maxgap" condition, and in line 21 the "border seed" condition, employing Fact 3.2.

Thus, having computed the arrays SUF and LCP, and the period arrays $P[1 \mathinner{.\,.} n]$ and $P'[1 \mathinner{.\,.} n]$ of u, we can find all seeds of u of length k in $O(n)$ total time. $\quad\square$

ALGORITHM SeedsOfAGivenLength(u, k)

```
 1: P[1 .. n] := period array of u;  P'[1 .. n] := suffix period array of u;
 2: SUF[1 .. n] := suffix array of u; LCP[1 .. n] := lcp array of u;
 3: Lists := emptyList;
 4: j := 1;
 5: while j ≤ n do
 6:     List := {SUF[j]};
 7:     while j < n and LCP[j + 1] ≥ k do
 8:         j := j + 1; List := append(List, SUF[j]);
 9:     if length(List) > 1 or SUF[j] ≤ n − k + 1 then
10:         Lists := append(Lists, List);
11:     j := j + 1;
12: for all List in Lists do
13:     BucketSort(List); { using the same set of buckets }
14: for all List in Lists do
15:     first := prev := n;  last := 1;  covers := true;
16:     for all i in List do
17:         first := min(first, i);  last := max(last, i);
18:         if i > prev + k then
19:             covers := false;
20:         prev := i;
21:     if covers and (k ≥ max(P[first + k − 1], P'[last])) then
22:         print "u[first .. (first + k − 1)] is a seed of u";
```

We compute the elements of the seed array $\text{Seed}[1 . . n]$ from left to right, i.e., in the order of increasing lengths of prefixes of u. Note that $\text{Seed}[i + 1] \geq \text{Seed}[i]$ for any $1 \leq i \leq n - 1$, this is due to Observation 2.2b. If $\text{Seed}[i + 1] > \text{Seed}[i]$ then we increase the current length of the seed by one letter at a time, in total at most $n - 1$ such operations are performed. Each time we query for the existence of a seed of a given length using the algorithm from Theorem 4.1. Thus we obtain $O(n^2)$ time complexity.

Theorem 4.2. *The seed array of a string $u \in \Sigma^n$ can be computed in $O(n^2)$ time.*

5. Quasiperiodicity testing

Let $u \in \Sigma^n$ and let $S = \{s_1, \ldots, s_k\}$ be a set of strings. In this section we show an algorithm which checks, for each s_i, if s_i is a seed of u, in $O(|u| + \sum_{i=1}^{k} |s_i| + \alpha)$ total time, where α is the total number of occurrences of the strings in S within u. The algorithm checks if s_i is also a left/right seed or a cover of u.

We start by computing, for each s_i, its occurrence set $Occ(s_i, u)$. This can be done using any efficient pattern matching algorithm for multiple patterns, e.g., Aho–Corasick algorithm [1] or suffix trees or arrays [9], in the aforementioned $O(|u| + \sum_{i=1}^{k} |s_i| + \alpha)$ time complexity. Since we use the multi-pattern matching algorithm only as a black box, we omit the details of how these algorithms work.

For each of the strings s_i, we compute $\text{maxgap}(s_i)$, $first(s_i)$, and $last(s_i)$, this can be done in $O(k + \alpha)$ time by simply processing the Occ sets. Finally, we need to compute the period arrays $P[1 . . n]$ and $P'[1 . . n]$ for the string u, which can be done in $O(n)$ time.

Now it all reduces to testing the criterion from Observation 2.3 using the border seed inequality from Fact 3.2: s_i is a seed of u if and only if:

$$|s_i| \geq \text{maxgap}(s_i) \wedge |s_i| \geq \max(\mathsf{P}[first(s_i) + |s_i| - 1], \; \mathsf{P}'[last(s_i)]).$$

These inequalities can be checked in constant time for a given s_i after all the required data structures have been precomputed.

Finally, assume that s_i is a seed of u. Then s_i is a left seed of u if only $first(s_i) = 1$, a right seed of u if $last(s_i) = n - |s_i| + 1$, and s_i is a cover of u if it is both a left seed and a right seed of u. We conclude with the following corollary.

Theorem 5.1. *For a given string u and a set of strings $S = \{s_1, \ldots, s_k\}$, it can be checked, for each s_i, if it is a seed, left seed, right seed, or a cover of u in $O(|u| + \sum |s_i| + \alpha)$ total time, where $\alpha = \sum |Occ(s_i, u)|$.*

6. Alternative algorithm for shortest seeds

In this section we present a new approach to shortest seeds computation based on very simple independent processing of disjoint chains in the suffix tree. It simplifies the computation of shortest seeds considerably.

Our algorithm utilizes a slightly modified version of Observation 2.3, formulated below as Lemma 6.3, which allows us to relax the definition of maxgaps. We discuss an algorithmically easier version of maxgaps, called *prefix maxgaps*, and show that it can substitute maxgap values when looking for the shortest seed.

We start by analyzing the "border seed" condition. We introduce a somewhat more abstract representation of sets of factors of u, called *prefix families*, and show how to find the shortest border seeds of u in such families. Afterwards the key algorithm for computing prefix maxgaps is presented. Finally, both techniques are combined to compute the shortest seed.

Let us fix the input string $u \in \Sigma^n$. For $v \in \Sigma^*$, by $PREF(v)$ we denote the set of all prefixes of v and by $PREF(v, k)$ we denote the set of prefixes of v of length at least k (*limited prefix subset*).

Let $\mathcal{F}$ be a family of limited prefix subsets $PREF(v, k)$ of some factors of u, such that $Occ(v, u) = Occ(v[1 . . k], u)$ for each v, i.e., all strings from one limited prefix subset share the same set of occurrences within u. Then we call $\mathcal{F}$ a *prefix family*. Every element $PREF(v, k) \in \mathcal{F}$ can be represented in a canonical form by a tuple of integers: $(first(v), last(v), k, |v|)$. Such a representation requires only constant space per element. By $\text{bseed}(u, \mathcal{F})$ we denote the shortest border seed of u contained in some element of $\mathcal{F}$.

Example 6.1. Let $u = \texttt{aabaababaabaaba}$ be the example string from Fig. 2. Let:

$$\mathcal{F} = \{PREF(\texttt{abaab}, 4), \; PREF(\texttt{babaa}, 4)\} = \{(2, 10, 4, 5), \; (6, 6, 4, 5)\}.$$

Note that $\bigcup \mathcal{F} = \{\texttt{abaa}, \texttt{abaab}, \texttt{baba}, \texttt{babaa}\}$. We have $\text{bseed}(u, \mathcal{F}) = \texttt{abaa}$.

The proof of the following fact is present implicitly in [14] (type-A and type-B seeds).

Theorem 6.2. *Let $u \in \Sigma^n$ and let $\mathcal{F}$ be a prefix family given in a canonical form. Then $\text{bseed}(u, \mathcal{F})$ can be computed in linear time, that is, in $O(n + |\mathcal{F}|)$ time.*

Alternative proof of Theorem 6.2. There is an alternative algorithm for computing bseed(u, $\mathcal{F}$), based on a special version of Find-Union data structure. Let B[$1 . . n$] be the border-array of u. Denote by *FirstGE*($\mathcal{I}$, c) (*first-greater-equal*) a query:

$$FirstGE(\mathcal{I}, c) \; = \; \min\{i \; : \; i \in \mathcal{I}, \; B[i] \geq c\}^1$$

where $\mathcal{I}$ is a subinterval of [$1 . . n$]. Below we show that a sequence of m such queries, sorted according to non-decreasing values of c, can be answered in $O(n + m)$ time. Using this subroutine, the following algorithm ComputeBorderSeed applies the border seed condition from Fact 3.2 to every element of $\mathcal{F}$.

More precisely, if $s \in PREF(v, k) \in \mathcal{F}$ is a border seed of u, then

$$\max(k, \; \mathsf{P}'[last(v)]) \; \leq \; |s| \; \leq \; |v| \quad \wedge \quad |s| \; \geq \; \mathsf{P}[first(v) + |s| - 1].$$

Note that we have used the assumption that $Occ(v, u) = Occ(s, u)$ to substitute *first*(s) by *first*(v) and *last*(s) by *last*(v). The latter of the above conditions is equivalent to B[$first(v) + |s| - 1$] $\geq first(v) - 1$. Hence, we are interested in the smallest value of $|s|$ which satisfies this inequality and the former of the above conditions, which is implemented as the *FirstGE* query below.

ALGORITHM ComputeBorderSeed(u, $\mathcal{F}$)

1: $bseed := +\infty$;
2: **for all** ($first(v)$, $last(v)$, k, $|v|$) **in** $\mathcal{F}$, in non-decreasing order of $first(v)$, **do**
3:     $k := \max(k, \; \mathsf{P}'[last(v)])$;
4:     $\mathcal{I} := [first(v) + k - 1, \; first(v) + |v| - 1]$;
5:     $pos := FirstGE(\mathcal{I}, \; first(v) - 1)$;
6:     $bseed := \min(bseed, \; pos - first(v) + 1)$;
7: **return** $bseed$;

The implementation of *FirstGE* queries uses a restricted version of the find/union data structure. The universe of the elements is [$1 . . n + 1$], which corresponds to the set of all the positions in u extended to the right by a sentinel (B[$n + 1$] = ∞). Initially the universe is partitioned into single-element sets. Throughout the algorithm every set is a segment and stores the index of its greatest (rightmost) element.

When processing a query *FirstGE*($\mathcal{I}$, c), the universe must be partitioned into maximal segments of elements for which the B value is less than c, followed by a single element for which this value is at least c. If we have such a partition of the set of positions in u, then answering the aforementioned *FirstGE* query for $\mathcal{I} = [a . . b]$ is fairly straightforward: it suffices to return the rightmost element in the set containing the element a, provided that this element does not exceed b. Updating the partition when moving from *FirstGE*($\mathcal{I}$, c) to *FirstGE*($\mathcal{I}'$, c') works as follows: we process all elements i for which B[i] $\in [c, c')$ and union the sets containing i and $i + 1$. Note that all the elements i satisfying the condition B[i] $\in [c, c')$ can be easily identified (one by one) if an inverse of the border array is precomputed.

In total we perform at most n union operations and $2n + m$ find operations, where $m = |\mathcal{F}|$. The sets that we union are only adjacent subintervals. Thus the *structure* of union operations forms a static tree (here it is a path graph) and therefore $O(n + m)$ find/union operations can be performed in $O(n + m)$ time [12]. □

6.1. Computation of the shortest seeds via prefix maxgaps

Recall that for a string u, the suffix tree $T(u)$ of u is a compacted TRIE representing all the factors of u. Each factor of u corresponds to an explicit of implicit node of $T(u)$. By *Nodes*(u) we denote the set of factors of u corresponding to explicit nodes of $T(u)$, for simplicity we identify the nodes with the strings they represent. Each leaf of the suffix tree is labeled with the index of the suffix of u that it corresponds to. Recall that $T(u)$ can be constructed in $O(n)$ time [9,10].

For $v \in Nodes(u)$, the set $Occ(v, u)$ corresponds to a leaf list of the node v (i.e., the set of labels of leaves in the subtree rooted at v), denoted as $LL(v)$. Note that $first(v) = \min LL(v)$ and $last(v) = \max LL(v)$, and such values can be computed for all $v \in Nodes(u)$ in $O(n)$ time. For $v \in Nodes(u)$, we define the *prefix maxgap* of v as:

$$\Delta(v) \; = \; \max\{\mathsf{maxgap}(w) \; : \; w \in PREF(v)\}.$$

Equivalently, $\Delta(v)$ is the maximum of maxgap values on the path from v to the root of $T(u)$. We introduce an auxiliary problem:

Prefix Maxgap Problem:
given a string $u \in \Sigma^n$, compute $\Delta(v)$ for all $v \in Nodes(u)$.

The following lemma (an alternative formulation of Observation 2.3) shows that prefix maxgaps can be used instead of maxgaps in searching for seeds. This is important since computation of prefix maxgaps $\Delta(v)$ is simple, in comparison with

---

1 We assume that $\min \emptyset = +\infty$.

maxgap(v)—this is due to the fact that the $\Delta(v)$ values on each path down the suffix tree $T(u)$ are non-decreasing. Efficient computation of maxgap(v) requires using augmented height-balanced trees [7] or other rather sophisticated techniques [5]. The shortest-seed algorithm in [14] also computes prefix maxgaps instead of maxgaps, however this observation is missing in [14].

Lemma 6.3. *Let s be a factor of* $u \in \Sigma^*$ *and let w be the shortest element of Nodes(u) such that* $s \in PREF(w)$. *The string s is a seed of u if and only if* $|s| \geq \Delta(w)$ *and s is a border seed of u.*

Proof. If $s \in Nodes(u)$ then $s = w$. Otherwise, s corresponds to an implicit node in an edge in the suffix tree, and w is the lower end of the edge. Note that in both cases we have $\Delta(w) \geq$ maxgap(s). By Observation 2.3, this implies part ($\Leftarrow$) of the conclusion of the lemma. As for the part ($\Rightarrow$), it suffices to show that if s is a seed of u then $|s| \geq \Delta(w)$.

Assume, to the contrary, that $|s| < \Delta(w)$. Let $v \in PREF(w) \cap Nodes(u)$ be the string for which maxgap$(v) = \Delta(w)$, and let a, b be consecutive elements of the set $Occ(v, u)$ for which $a +$ maxgap$(v) = b$.

Let us note that no occurrence of s starts at any of the positions $a + 1, \ldots, b - 1$. Moreover, none of the suffixes of the form $u[i \mathinner{.\,.} n]$, for $a + 1 \leq i \leq b - 1$, is a prefix of s. Indeed, v is a prefix of s of length at most $n - b + 1$, and such an occurrence of s (or its prefix) would imply an extra occurrence of v. Note that at most $|s| \leq b - a - 1$ first positions in the interval $[a, b]$ can be covered by an occurrence of s in u (at position a or earlier) or by a suffix of s which is a prefix of u. Hence, position $b - 1$ is not covered by s at all, a contradiction. $\quad\square$

By Lemma 6.3, to complete the shortest seed algorithm it suffices to solve the Prefix Maxgap Problem (this is further clarified in the ComputeShortestSeed algorithm below). For this, we consider yet another problem. By $SORT(X)$ let us denote the sorted sequence of elements of the set $X \subseteq \{1, 2, \ldots, n\}$.

Chain Prefix Maxgap Problem
    Input: a family of disjoint sets $X_1, X_2, \ldots, X_k \subseteq \{1, 2, \ldots, n\}$
              together with $SORT(X_1 \cup X_2 \cup \cdots \cup X_k)$.
              The size of the input is $m = \sum |X_i|$.
    Output: the numbers $\Delta_i = \max_{j \leq i}$ maxgap$(X_j \cup X_{j+1} \cup \ldots \cup X_k)$.

Example 6.4. Let $n = 15$ and let: $X_1 = \{1, 5, 11\}$, $X_2 = \{2, 14\}$, and $X_3 = \{3, 7, 8\}$, hence $m = 7$. Then we have maxgap$(X_3) = 4$, maxgap$(X_2 \cup X_3) = 6$, and maxgap$(X_1 \cup X_2 \cup X_3) = 3$. The output of the Chain Prefix Maxgap Problem is $\Delta_1 = 3$, $\Delta_2 = 6$, and $\Delta_3 = 6$. Note that the resulting sequence is non-decreasing.

Theorem 6.5. *The Chain Prefix Maxgap Problem can be solved in $O(m)$ time using an auxiliary array of size n.*

Proof. Initially we have the list $L = SORT(X_1 \cup X_2 \cup \ldots \cup X_k)$ and compute Δ_1 in a naive way. Then we keep removing the elements of $X_1, X_2, \ldots$ from L and updating the Δ_j values, using the fact that they are non-decreasing.

Let *pred* and *suc* denote the predecessor and successor of an element of L. The elements of L store a Boolean flag *marked*, initially set to false. In the algorithm we use an auxiliary array $pos[1 \mathinner{.\,.} n]$ such that $pos[i]$ is a pointer to the element of value i in L, if there is no such element then the value of $pos[i]$ can be arbitrary. Obviously the algorithm takes $O(m)$ time. $\quad\square$

ALGORITHM ChainPrefixMaxgap(L)

1: $\Delta_1 :=$ maxgap(L); { naive computation }
2: **for** $j := 2$ **to** k **do**
3:     $\Delta_j := \Delta_{j-1}$;
4:     **for all** i **in** X_{j-1} **do** *marked*$(pos[i]) :=$ **true**;
5:     **for all** i **in** X_{j-1} **do**
6:         $p := pred(pos[i])$; $q := suc(pos[i])$;
7:         **if** $(p \neq$ **nil**$)$ **and** $(q \neq$ **nil**$)$ **and** $($**not** *marked*$(p))$
             and $($**not** *marked*$(q))$ **then**
8:           $\Delta_j := \max(\Delta_j,$ *value*$(q) -$ *value*$(p))$;
9:         *delete*$(L, pos[i])$;

Theorem 6.6. *The Prefix Maxgap Problem can be reduced to a collection of Chain Prefix Maxgap Problems of total size $O(n \log n)$.*

Proof. We solve a more abstract version of the Prefix Maxgap Problem. We are given an arbitrary tree T with n leaves annotated with distinct integers from the interval $[1, n]$, and we need to compute the values $\Delta(v)$ for all $v \in Nodes(T)$, defined as follows: maxgap$(v) =$ maxgap$(LL(v))$, where $LL(v)$ is the leaf list of v, and $\Delta(v)$ is the maximum of the values maxgap on the path from v to the root of T. We start by sorting $LL(root(T))$, which can be done in $O(n)$ time. Throughout the algorithm we store a global auxiliary array $pos[1 \mathinner{.\,.} n]$, required in the ChainPrefixMaxgap algorithm.

Let us find a *heaviest path P* in T, i.e., a path from the root down to a leaf, such that all *hanging* subtrees are of size at most $|T|/2$ each. The values of $\Delta(v)$ for $v \in P$ can all be computed in $O(n)$ time, using a reduction to the Chain Prefix Maxgap Problem (see Fig. 5).

Then we perform the computation recursively for the hanging subtrees, previously sorting $LL(T')$ for each hanging subtree T'. Such sorting operations can be performed in $O(n)$ total time for all hanging subtrees.
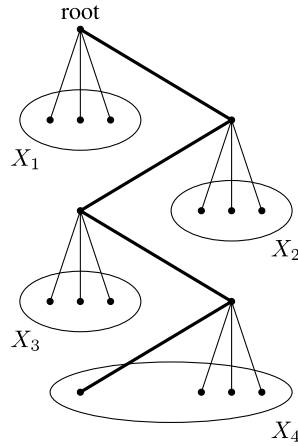
Fig. 5. A tree with an example heaviest path P (in bold). The values $\Delta(v)$ for $v \in P$ can be computed using a reduction to the Chain Prefix Maxgap Problem with the sets $X_1, \ldots, X_4$.

At each level of recursion we need $O(n)$ time, and the depth of recursion is logarithmic in terms of n. Hence, the total size of the invoked Chain Prefix Maxgap Problems is $O(n \log n)$. $\square$

Now we proceed to the shortest seed computation. In the algorithm we consider all factors of u, dividing them into groups corresponding to elements of $Nodes(u)$. Let $w \in Nodes(u)$ and let v be its parent. Let $s \in PREF(w)$ be a string containing v as a proper prefix, i.e., $s \in PREF(w, |v| + 1)$. By Lemma 6.3, the string s is a seed of u if and only if $|s| \geq \Delta(w)$ and s is a border seed of u.

Using the previously described reductions (Theorems 6.2–6.6), we obtain the following algorithm:

ALGORITHM ComputeShortestSeed(u)

1: Construct the suffix tree $T(u)$ for the input string u;
2: Solve the Prefix Maxgap Problem for $T(u)$ using the ChainPrefixMaxgap
3:    algorithm — in $O(n \log n)$ total time (Theorems 6.5 and 6.6);
4: $\mathcal{F} := \{ PREF(w, \max(|v| + 1, \Delta(w))) : (v, w) \text{ is an edge in } T(u) \}$;
5: **return** bseed($u, \mathcal{F}$); { Theorem 6.2 }

Observe that the *workhorse* of the algorithm is the chain version of the Prefix Maxgap Problem, which has a fairly simple linear time solution. The main problem is of a structural nature, we have a collection of very simple problems each computable in linear time but the total size is not linear. This identifies the bottleneck of the algorithm from the complexity point of view.

7. Long seeds

Note that the most time-expensive part of the ComputeShortestSeed algorithm is the computation of prefix maxgaps, all the remaining operations are performed in $O(n)$ time. Using this observation we can show a more efficient algorithm computing the shortest seed provided that its length m is sufficiently large. For example if $m = \Theta(n)$ then we obtain an $O(n)$ time algorithm for the shortest seed.

This is formalized in the following Theorem 7.2. In its proof we utilize the following property of quasiperiodic repetitions relating them to periodic properties of factors.

Observation 7.1. *If s is the shortest cover or the shortest left seed or the shortest seed of a string u then $per(s) > |s|/2$.*

Proof. If $per(s) \leq |s|/2$ for any string s, then s has a border of length at least $|s|/2$. Hence, this border is also a cover of s. Thus, by Observation 2.1 parts a–c, the string s can be neither the shortest cover, nor the shortest left seed nor seed of any string. $\square$

Theorem 7.2. *One can check if the shortest seed of a given string u has length at least m in $O(n \log (n/m))$ time, where $n = |u|$. If so, a corresponding seed can be reported within the same time complexity.*

Proof. We show how to modify the ComputeShortestSeed algorithm. Denote by s the shortest seed of u, and let $|s| = k$.

By Observation 7.1, the longest overlap between consecutive occurrences of s in u is at most $\frac{k}{2}$, therefore the number of occurrences of s in u is at most $\frac{2n}{k}$. Hence, searching for the shortest seed of length at least m, it suffices to consider nodes v of the suffix tree $T(u)$ for which: $|v| \geq m$ and $|LL(v)| \leq \frac{2n}{m}$, where $LL(v)$ is the leaf list of the node v.

Thus, we are only interested in prefix maxgaps for nodes in several subtrees of $T(u)$, each of which contains $O(n/m)$ nodes. Thanks to the small size of each subtree, the algorithm ComputeShortestSeed finds all such prefix maxgaps in $O(n \log (n/m))$ time.

Note that using this algorithm for each node we obtain a prefix maxgap only in the selected subtree containing it (not necessarily in the whole tree). Let $\Delta'(w)$ be such a value for the node w. Then we have $\mathrm{maxgap}(w) \leq \Delta'(w) \leq \Delta(w)$ and these inequalities suffice to use Lemma 6.3 with $\Delta'(w)$ instead of $\Delta(w)$. Indeed, the ($\Leftarrow$) part of the lemma follows, again, from Observation 2.3, since $\Delta'(w) \geq \mathrm{maxgap}(s)$, and the ($\Rightarrow$) part is a weaker version of the ($\Rightarrow$) part of the original Lemma 6.3, since $\Delta'(w) \leq \Delta(w)$.　□

8. Conclusions

We have presented several efficient algorithms related to quasiperiodicities in strings. We have introduced new notions of left and right seeds, intermediate between seeds and covers, and presented linear time algorithms computing the left seed array and the maximal left seed array of a string. We have also given several algorithms related to seeds, in particular, an improved shortest-seed algorithm and the first approach to seed array computation faster than the naive algorithm.

Acknowledgement

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.
[2] A. Apostolico, D. Breslauer, Of periods, quasiperiods, repetitions and covers, in: Structures in Logic and Computer Science, 1997, pp. 236–248.
[3] A. Apostolico, A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Theor. Comput. Sci. 119 (2) (1993) 247–265.
[4] A. Apostolico, M. Farach, C.S. Iliopoulos, Optimal superprimitivity testing for strings, Inf. Process. Lett. 39 (1) (1991) 17–20.
[5] O. Berkman, C.S. Iliopoulos, K. Park, The subtree max gap problem with application to parallel string covering, Inf. Comput. 123 (1) (1995) 127–137.
[6] D. Breslauer, An on-line string superprimitivity test, Inf. Process. Lett. 44 (6) (1992) 345–347.
[7] G.S. Brodal, C.N.S. Pedersen, Finding maximal quasiperiodicities in strings, in: R. Giancarlo, D. Sankoff (Eds.), CPM, in: Lecture Notes in Computer Science, vol. 1848, Springer, 2000, pp. 397–411.
[8] M. Christou, M. Crochemore, C.S. Iliopoulos, M. Kubica, S.P. Pissis, J. Radoszewski, W. Rytter, B. Szreder, T. Walen, Efficient seeds computation revisited, in: R. Giancarlo, G. Manzini (Eds.), CPM, in: Lecture Notes in Computer Science, vol. 6661, Springer, 2011, pp. 350–363.
[9] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[10] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, 2003.
[11] J. Fischer, V. Heun, A new succinct representation of RMQ-information and improvements in the enhanced suffix array, in: B. Chen, M. Paterson, G. Zhang (Eds.), ESCAPE, in: Lecture Notes in Computer Science, vol. 4614, Springer, 2007, pp. 459–470.
[12] H.N. Gabow, R.E. Tarjan, A linear-time algorithm for a special case of disjoint set union. In: Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC), pp. 246–251, 1983.
[13] D. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.
[14] C.S. Iliopoulos, D.W.G. Moore, K. Park, Covering a string, Algorithmica 16 (3) (1996) 288–297.
[15] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: J.C.M. Baeten, J.K. Lenstra, J. Parrow, G.J. Woeginger (Eds.), ICALP, in: Lecture Notes in Computer Science, vol. 2719, Springer, 2003, pp. 943–955.
[16] P. Ko, S. Aluru, Space efficient linear time construction of suffix arrays, in: R.A. Baeza-Yates, E. Chávez, M. Crochemore (Eds.), CPM, in: Lecture Notes in Computer Science, vol. 2676, Springer, 2003, pp. 200–210.
[17] Y. Li, W.F. Smyth, Computing the cover array in linear time, Algorithmica 32 (1) (2002) 95–106.
[18] M. Lothaire (Ed.), Algebraic Combinatorics on Words, Cambridge University Press, 2001.
[19] M. Lothaire (Ed.), Applied Combinatorics on Words, Cambridge University Press, 2005.
[20] D. Moore, W.F. Smyth, Computing the covers of a string in linear time, in: SODA, 1994, pp. 511–515.