

# Computing String Covers in Sublinear Time

Jakub Radoszewski<sup>1,\*</sup>[0000-0002-0067-6401] and Wiktor  
Zuba<sup>2,\*\*</sup>[0000-0002-1988-3507]

<sup>1</sup> University of Warsaw, Warsaw, Poland [jrad@mimuw.edu.pl](mailto:jrad@mimuw.edu.pl)

<sup>2</sup> CWI, Amsterdam, The Netherlands [wiktor.zuba@cwi.nl](mailto:wiktor.zuba@cwi.nl)

**Abstract.** Let  $T$  be a string of length  $n$  over an integer alphabet of size  $\sigma$ . In the word RAM model,  $T$  can be represented in  $\mathcal{O}(n/\log_\sigma n)$  space. We show that a representation of all covers of  $T$  can be computed in the optimal  $\mathcal{O}(n/\log_\sigma n)$  time; in particular, the shortest cover can be computed within this time. We also design an  $\mathcal{O}(n(\log \sigma + \log \log n)/\log n)$ -sized data structure that computes in  $\mathcal{O}(1)$  time any element of the so-called (shortest) cover array of  $T$ , that is, the length of the shortest cover of any given prefix of  $T$ . As a by-product, we describe the structure of cover arrays of Fibonacci strings. On the negative side, we show that the shortest cover of a length- $n$  string cannot be computed using  $o(n/\log n)$  operations in the PILLAR model of Charalampopoulos, Kociumaka, and Wellnitz (FOCS 2020).

**Keywords:** Cover · Quasiperiod · Cover array · Packed string matching · PILLAR model

## 1 Introduction

A string  $C$  is called a *cover* (or a *quasiperiod*) of a string  $T$  if each position in  $T$  lies within an occurrence of  $C$  in  $T$ . A cover is called *proper* if it is shorter than the covered string. A string that does not have proper covers is called *superprimitive* (see [6]). The shortest cover of a string of length  $n$  can be computed in  $\mathcal{O}(n)$  time [1]. Furthermore, all covers of a length- $n$  string can be computed in  $\mathcal{O}(n)$  time [33]. A cover of a string is a prefix of the string, so a string of length  $n$  indeed has at most  $n$  covers.

The lengths of all covers of a string of length  $n$  can be represented using  $\mathcal{O}(\log n)$  disjoint arithmetic progressions [16]. For a string  $T$ , we denote such a representation as  $\text{Covers}(T)$ . A similar representation is well known to exist for the set of all borders of a string (see, e.g., [14]).

We consider the standard word RAM model with machine word composed of  $\omega \geq \log_2 n$  bits. In this model, a string of length  $n$  over an alphabet of size  $\sigma$  can be represented using  $\mathcal{O}(n/\log_\sigma n)$  machine words, that is,  $\mathcal{O}(n \log \sigma)$  bits, in a so-called packed representation; see [4]. In Section 3 we show the following result that improves upon [1,6,33] in the case that the string is over a small alphabet.

---

\* Supported by the Polish National Science Center, grant no. 2022/46/E/ST6/00463.

\*\* Supported by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No 101034253.

**Theorem 1.** *A representation  $\text{Covers}(T)$  of all the covers of a string  $T$  of length  $n$  over an alphabet of size  $\sigma$  given in a packed form, consisting of  $\mathcal{O}(\log n)$  arithmetic progressions, can be computed in  $\mathcal{O}(n/\log_\sigma n)$  time.*

The representation  $\text{Covers}(T)$  can be transformed in  $\mathcal{O}(n/\log n)$  time to a Boolean array of size  $n$ , represented in a packed form, that stores for every  $\ell \in [1..n]$  a Boolean value that determines if a length- $\ell$  prefix of  $T$  is a cover of  $T$ .

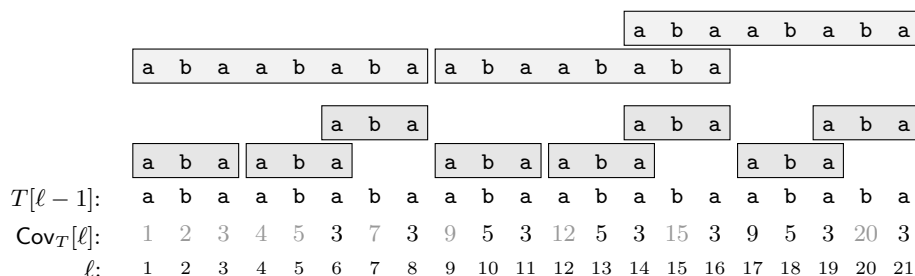


Fig. 1: Both proper covers (aba, abaababa) and the cover array of a Fibonacci string  $T$ . Values  $\text{Cov}_T[\ell] = \ell$  corresponding to superprimitive prefixes  $T[0.. \ell]$  are shown in gray.

The (shortest) cover array of a string  $T$ ,  $\text{Cov}_T[1..|T|]$ , stores for every position  $\ell$  of  $T$  the length of the shortest cover of a length- $\ell$  prefix of  $T$  as  $\text{Cov}_T[\ell]$ ; see Fig. 1. The cover array is the output of Breslauer’s on-line algorithm computing shortest covers [6]; see also [15]. We give a sublinear-sized representation of this array.

**Theorem 2.** *Let  $T$  be a text of length  $n$  over an integer alphabet of size  $\sigma$ . There exists a data structure using space  $\mathcal{O}(n(\log \sigma + \log \log n)/\log n)$  that, given  $\ell \in [1..n]$ , returns  $\text{Cov}_T[\ell]$  in  $\mathcal{O}(1)$  time.*

Our results extend the list of basic stringology problems for which representing the input in a packed form allows to obtain an  $o(n)$ -time solution; see [2,4,9,12,26,34].

As a by-product, we give a characterization of the cover arrays of Fibonacci strings (Theorem 5).

We also consider covers in the PILLAR model. This model was introduced in [10] with the aim of unifying approximate pattern matching algorithms across different settings. In this model, we consider a collection  $\mathcal{X}$  of strings and assume that certain primitive PILLAR operations can be performed efficiently. The set of primitive operations consists of computing the length of the longest common prefix (LCP) or suffix (LCP<sub>R</sub>) of substrings of strings in  $\mathcal{X}$ , so-called internal pattern matching (IPM) queries that ask for the set of occurrences of one substring

in another substring that is at most twice as long, represented as an arithmetic progression, as well as simple operations allowing to access letters of strings. (For a formal definition, see Section 5.)

The strength of the PILLAR model lies in the fact that efficient implementations of its primitives are known in many different settings:

- In the *standard setting*, in which all strings in the collection  $\mathcal{X}$  are substrings of a given string of length  $n$  over an integer alphabet of size  $\sigma$ , each PILLAR operation on its substrings can be performed in  $\mathcal{O}(1)$  time after  $\mathcal{O}(n)$  preprocessing [5,19,31] and even after just  $\mathcal{O}(n/\log_\sigma n)$  preprocessing [26,30].
- In the *dynamic setting*, the collection  $\mathcal{X}$  can updated dynamically under edit operations (insertions, deletions, substitutions) with each edit operation and each PILLAR operation performed in  $\mathcal{O}(\log^{\mathcal{O}(1)} N)$  time, where  $N$  is the total size of  $\mathcal{X}$  [10,27].
- In the *fully compressed setting*, given a collection  $\mathcal{X}$  of straight-line programs (SLPs) of total size  $n$  generating strings of total length  $N$ , each PILLAR operation can be performed in  $\mathcal{O}(\log^2 N \log \log N)$  time after  $\mathcal{O}(n \log N)$ -time preprocessing [10].
- An efficient implementation of the PILLAR operations is also known in the *quantum setting* [23,25].

Thus if a problem can be solved fast in the PILLAR model, it immediately implies its efficient solutions in all the above mentioned settings. For example, the fact that an  $\mathcal{O}(\log n)$ -sized representation of all the periods (equivalently, borders) of a length- $n$  string can be computed in  $\mathcal{O}(\log n)$  time in the PILLAR model [30,31, Period Query] implies that a representation of the periods of a dynamic string can be updated in  $\mathcal{O}(\log^{\mathcal{O}(1)} N)$  time per operation and that a representation of all periods of a fully compressed string of length  $N$  generated by an SLP of size  $n$  can be computed in  $\mathcal{O}(n \log^{\mathcal{O}(1)} N)$  time. In the case of covers, some efficient algorithms were designed for each of the above mentioned non-standard settings separately:

- In the *internal setting*, which is a special case of the standard setting, after  $\mathcal{O}(n \log n)$  preprocessing of a length- $n$  string  $T$ , one can compute a representation of all covers of any substring of  $T$  in  $\mathcal{O}(\log n \log \log n)$  time and the shortest cover of any substring in  $\mathcal{O}(\log n)$  time [16,3].
- In a restricted dynamic setting in which each edit operation is reverted immediately after it is performed, the shortest cover can be updated in  $\mathcal{O}(\log n)$  time [32]. No algorithm is known for computing covers in the fully dynamic setting.
- In the fully compressed setting, a representation of all covers of a length- $N$  string specified by an SLP of size  $n$  with derivation tree of height  $h$  can be computed in  $\mathcal{O}(nh(n + \log^2 N))$  time [24]; with the technique of balancing SLPs [22], the time complexity becomes  $\mathcal{O}(n \log N(n + \log^2 N))$ .

The  $\mathcal{O}(n)$ -time algorithms for computing covers of a length- $n$  string [1,6,33] perform only single-letter comparisons and thus work also in the PILLAR model.

If there was a (much) more efficient algorithm computing the shortest cover of a string in the PILLAR model, one would immediately improve or generalize all the above results, including our Theorem 1. We show that, contrary to the case of periods, no such efficient algorithm for covers exists. A proof of Theorem 3 is given in Section 5.

**Theorem 3.** *There is no algorithm in the PILLAR model that solves any of the following problems for a length- $n$  binary string in  $o(n/\log n)$  time:*

- check if  $T$  is superprimitive;
- check if a given prefix of  $T$  is a cover of  $T$ .

Consequently, computing the shortest cover or a representation of all covers of a string requires  $\Omega(n/\log n)$  time in the PILLAR model.

## 2 Preliminaries

We assume that letters of a string  $T$  are numbered from 0 to  $|T| - 1$ , i.e.,  $T = T[0] \cdots T[|T| - 1]$ . By  $T[i..j] = T[i..j+1)$  we denote a substring  $T[i] \cdots T[j]$ . If  $T$  is given in a packed form, then packed representation of its substring  $T[i..j]$  can be computed in  $\mathcal{O}((j-i+1)/\log_\sigma n)$  time using standard word RAM operations. A substring  $T[i..j]$  is called a prefix if  $i = 0$  and a suffix if  $j = |T| - 1$ . A string  $B$  that occurs in  $T$  as a prefix and as a suffix is called a border of  $T$ .

A positive integer  $p$  is called a period of string  $U$  if  $U[i] = U[i+p]$  holds for all  $i \in [0..|U| - p)$ . A string  $U$  is called *periodic* if the smallest period  $p$  of  $U$  satisfies  $2p \leq |U|$ . Otherwise,  $U$  is called *aperiodic*. We also use the following Periodicity Lemma.

**Lemma 1 (Fine and Wilf, [20]).** *If a string  $U$  has periods  $p$  and  $q$  and  $p + q \leq |U|$ , then  $\gcd(p, q)$  is a period of  $U$ .*

For a string  $X$  and non-negative integer  $k$ , by  $X^k$  we denote a concatenation of  $k$  copies of  $X$ . A non-empty string  $U$  is *primitive* if  $U = X^k$  implies that  $k = 1$ . A string of the form  $X^2$  is called a square. If  $X$  is primitive, the square  $X^2$  is said to be  *primitively rooted*.

**Lemma 2 (Three Squares Lemma, [18]).** *If a string  $U$  has primitively rooted square prefixes  $X^2, Y^2, Z^2$  such that  $|X| < |Y| < |Z|$ , then  $|Z| > |X| + |Y|$ .*

## 3 Sublinear-Time Covers

Let  $c = \lfloor \frac{1}{6} \log_\sigma n \rfloor$ . To show Theorem 1 we divide the set  $\text{Covers}(T)$  into two subsets:

- $\text{SCovers}(T) = \text{Covers}(T) \cap [1..c]$  of short cover lengths, and
- $\text{LCovers}(T) = \text{Covers}(T) \cap [c+1..n] = \text{Covers}(T) \setminus \text{SCovers}(T)$  of long cover lengths

and compute their representations separately. If  $c = 0$ , there are only long covers.

**Lemma 3.** *The representation of  $\text{SCovers}(T)$  can be computed in  $\mathcal{O}(n/\log_\sigma n)$  time.*

*Proof.* Let  $\mathcal{F}$  be the set of all the factors of  $T$  of length  $3c$  that start at positions that are multiples of  $c$ . If the length of  $T$  is not a multiple of  $c$ , when computing  $\mathcal{F}$  we extend  $T$  with at most  $c$  arbitrary letters. Let us notice that every string in  $\mathcal{F}$  fits in a machine word, so those strings can be treated as integers.

To compute  $\mathcal{F}$ , we first construct a Boolean array of all the possible length- $3c$  strings, and then iterate through the length- $3c$  substrings of  $T$  starting at positions that are multiples of  $c$ , addressing the array directly through those integer representations. The size of the array as well as  $|\mathcal{F}|$  is bounded by  $\sigma^{3c} \leq \sigma^{\frac{1}{2} \log_\sigma n} = \sqrt{n}$ . Iterating through all the considered substrings takes  $\mathcal{O}(n/c)$  time. Hence, this computation takes  $\mathcal{O}(n/\log_\sigma n)$  time in total.

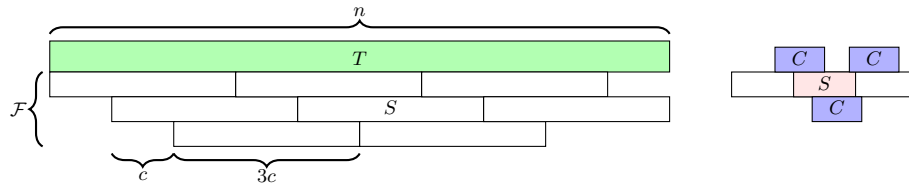


Fig. 2: Algorithm for checking a single candidate for a short cover. For each string in  $\mathcal{F}$  we check if the occurrences of  $C$  therein cover its middle part.

The next claim resembles to some extent a property of seeds; cf. [29, Lemma 2.2]. See Fig. 2 for an illustration.

*Claim.* For  $i \in [1..c]$ ,  $C = T[0..i]$  is a cover of  $T$  if and only if  $C$  is a border of  $T$  and the occurrences of  $C$  in each string  $S$  in  $\mathcal{F}$  cover the middle length- $c$  part of  $S$ .

*Proof.* ( $\Rightarrow$ ) Assume that  $C$  is a cover of  $T$ . As  $C$  has to cover the first and the last position of  $T$ ,  $C$  is a border of  $T$ . For every integer multiple  $ic$  of  $c$  such that  $(i+1)c < |T|$ , occurrences of  $C$  in  $T$  have to cover  $U = T[ic..(i+1)c]$ . These occurrences need to be contained in  $S = T[(i-1)c.. \min((i+2)c, |T| - 1)]$ . We have  $S \in \mathcal{F}$  (possibly after appending  $3c - |S|$  letters) and  $C$  covers the middle length- $c$  part of  $S$ .

( $\Leftarrow$ ) Assume that  $C$  is a border of  $T$  and the occurrences of  $C$  in each string  $S$  in  $\mathcal{F}$  cover the middle length- $c$  part of  $S$ . Since  $C$  is a border of  $T$ , its occurrences cover substrings  $T[0..|C|)$  and  $T[|T| - |C|..|T|)$ . Moreover, for every integer multiple  $ic$  of  $c$  such that  $(i+1)c < |T|$ , occurrences of  $C$  in  $T$  cover  $T[ic..(i+1)c]$ . Hence, occurrences of  $C$  in  $T$  cover all positions of  $T$ .  $\square$

Now to compute  $\text{SCovers}(T)$  we iterate through all the  $c$  lengths of candidates for a short cover independently.

For a single candidate  $C = T[0..i)$ , it is enough to check if  $C$  is a suffix of  $T$ , which can be done in  $\mathcal{O}(1)$  time, and then for each substring  $S \in \mathcal{F}$  check if the occurrences of  $C$  in  $S$  cover its middle part. The latter can be done naively in  $\mathcal{O}(|C| + |S|)$  time, which sums up to  $\mathcal{O}(c \cdot |\mathcal{F}|) = \mathcal{O}(\sqrt{n} \cdot \log_\sigma n)$  time for each  $i$ , and  $\mathcal{O}(\sqrt{n} \cdot \log_\sigma^2 n) = o(n/\log_\sigma n)$  time in total for all the candidates.

The result is reported in the form of the  $\mathcal{O}(c) = \mathcal{O}(\log n)$  lengths of short covers (arithmetic progressions of length 1).  $\square$

In the computation of long covers, we use Internal Pattern Matching. In particular, we use IPM queries which, given two substrings  $X, Y$  of  $T$  such that  $|X| \leq |Y| \leq 2|X|$ , return the set of occurrences of  $X$  in  $Y$  represented as an arithmetic progression. Moreover, we use Period Queries that return the set of all periods (equivalently, borders) of any substring of  $T$ . We need to apply a period query only to  $T$  itself. Such a query returns, for every  $d$  being an integer power of two, the set of lengths of all borders of  $T$  of length between  $d$  and  $2d$  represented as an arithmetic progression. If an arithmetic progression has length greater than two, then the difference  $p$  of the progression is the common shortest period of all the borders represented by this progression.

**Theorem 4 ([30]).** *Assume that a text  $T$  of length  $n$  over integer alphabet of size  $\sigma$  is given in a packed form. After  $\mathcal{O}(n/\log_\sigma n)$  time preprocessing, one can answer an IPM query for substrings  $X, Y$  of  $T$  in  $\mathcal{O}(|Y|/|X|)$  time and a Period Query for  $T$  in  $\mathcal{O}(\log n)$  time.*

**Lemma 4.** *The representation of  $\text{LCovers}(T)$  can be computed in  $\mathcal{O}(n/\log_\sigma n)$  time.*

*Proof.* As we already noticed, a cover of  $T$  is in particular its border. We ask a Period Query of Theorem 4 to compute a representation of the set of all borders of  $T$ . We disregard arithmetic progressions such that all their elements are smaller than  $c$ , as they can only correspond to the case of short covers that was already considered. Moreover, we trim the at most one remaining arithmetic progression that contains elements smaller than  $c$  so that it contains only border lengths greater than  $c$ .

For each arithmetic progression there exists a cut-off value  $t$  such that all borders of length at most  $t$  represented by the progression are covers of  $T$ , while the longer ones are not covers of  $T$ . (This is because a shorter border from the progression is a cover of a longer border from the progression.) It is sufficient to compute this cut-off value for each progression.

We consider the progressions separately. Let us consider a progression  $\Gamma$  of border lengths in  $[d..2d]$ . For the two shortest borders  $B_1, B_2$  represented by the progression  $\Gamma$  (or fewer if progression  $\Gamma$  contains at most one element), we use IPM queries for  $B_i$  and substrings of  $T$  of length  $2|B_i| - 1$  starting at positions  $\equiv 0 \pmod{|B_i|}$  to find a representation of the set of occurrences of  $B_i$  in  $T$  as

$\mathcal{O}(n/d)$  arithmetic progressions. This representation allows us to easily check in  $\mathcal{O}(n/d)$  time if  $B_i$  is a cover of  $T$ .

If any of  $B_1$  and  $B_2$  exists and is not a cover of  $T$ , we can safely ignore all the remaining borders in progression  $\Gamma$ . Otherwise, if at least three borders are represented by  $\Gamma$ , we know that the difference  $p$  of the progression is the common smallest period of all borders represented by  $\Gamma$ . In  $\mathcal{O}(n/d)$  time we partition the already computed occurrences of  $B_1$  in  $T$  into maximal arithmetic progressions of consecutive occurrences with difference  $p$ . Let  $\Delta$  be the minimum length of such an arithmetic progression of occurrences of  $B_1$ . Then exactly the  $\Delta$  shortest borders of the progression  $\Gamma$  are covers of  $T$ , or all borders if the progression contains less than  $\Delta$  elements.

Let us argue for the correctness of the algorithm. Let  $B_1, B_2, \dots, B_r$  be all borders represented by progression  $\Gamma$  ordered by increasing lengths. We have  $|B_{i+1}| = |B_i| + p$  for all  $i \in [1..r)$ . It suffices to note that (1)  $B_{\Delta'}$  for  $\Delta' = \min(\Delta, r)$  is a cover of  $T$  and (2)  $B_{\Delta+1}$ , if it exists, is not a cover of  $T$ .

As for (1), as all arithmetic progressions with difference  $p$  of occurrences of  $B_1$  in  $T$  have length at least  $\Delta'$ , they imply occurrences of  $B_{\Delta'}$  that cover the same set of positions of  $T$  as the occurrences of  $B_1$ , i.e., all positions.

As for (2), assume that  $B_{\Delta+1}$  exists (with  $\Delta \geq 2$ ) and let  $i, i+p, \dots, i+(\Delta-1)p$  be a maximal arithmetic progression of occurrences of  $B_1$  in  $T$ . The previous arithmetic progression has its last element smaller than  $i - (|B_1| - p)$ , as otherwise, by Lemma 1,  $B_1$  would have a period smaller than  $p$ . Similarly, the next arithmetic progression starts at a position greater than  $i + (\Delta-1)p + (|B_1| - p)$ . An occurrence of  $B_{\Delta+1}$  in  $T$  implies an arithmetic progression of occurrences of  $B_1$  with difference  $p$  and  $\Delta+1$  elements starting at the same position. Thus none of the occurrences of  $B_1$  at positions  $i, i+p, \dots, i+(\Delta-1)p$  extend to an occurrence of  $B_{\Delta+1}$ , the last position of  $T$  covered by an occurrence of  $B_{\Delta+1}$  from a previous arithmetic progression is smaller than  $i+p$ , and the first position of  $T$  covered by an occurrence of  $B_{\Delta+1}$  from a next arithmetic progression is greater than  $i+(\Delta-1)p$ . Hence, position  $i+p$  of  $T$  is not covered by occurrences of  $B_{\Delta+1}$ . This proves (2) and concludes correctness of the algorithm.

Overall, an arithmetic progression of border lengths in  $[d..2d]$  is processed in  $\mathcal{O}(n/d)$  time. It suffices to consider  $d$  such that  $2d \geq c$ , i.e.,  $d = 2^i$  for  $i \geq (\log_2 c) - 1$ . The time complexity is thus proportional to:

$$\sum_{i=\lceil \log_2 c \rceil - 1}^{\lfloor \log_2 n \rfloor} \frac{n}{2^i} \leq \frac{n}{2^{\lceil \log_2 c \rceil - 1}} \sum_{i=0}^{\infty} \frac{1}{2^i} = \mathcal{O}(n/c) = \mathcal{O}(n/\log_{\sigma} n),$$

as desired. □

Theorem 1 follows directly from Lemmas 3 and 4.

*Remark 1.* The algorithm for computing long covers works in  $\mathcal{O}(n/\log_{\sigma} n)$  time in the PILLAR model. For a constant  $\sigma$ , the complexity matches our lower bound of Theorem 3. However, the computation of short covers in Lemma 3 works in  $\Theta(n)$  time in the PILLAR model.

## 4 Sublinear Data Structure for Cover Array

### 4.1 Why Representing the Cover Array in Sublinear Space can be a Challenge

The cover array may require  $\Theta(n \log n)$  bits to represent in a straightforward manner. In particular, the array may contain  $\Theta(n)$  different values; this is true even if we disregard trivial positions  $i$  such that  $\text{Cov}_T[i] = i$  and positions  $i$  such that  $T[0..i]$  is periodic, as shown in the following Example 1.

*Example 1.* Let  $T = \mathbf{a}^{2m}\mathbf{b}\mathbf{a}^{3m}\mathbf{b}\mathbf{a}^{2m}$  for positive integer  $m$  be a string of length  $\Theta(m)$ . Then all prefixes of  $T$  of length at least  $2m + 1$  are aperiodic and the last  $m + 1$  positions of the array  $\text{Cov}_T$  contain the following lengths of proper covers:  $3m + 1, 3m + 2, \dots, 4m + 1$ .

Example 1 might still not be fully convincing that a sublinear-sized representation of the cover array is not obvious. Indeed, the the cover array of the string family from Example 1 has an especially simple structure (a prefix consisting only of ones, a substring with an arithmetic sequence with difference 1 corresponding to superprimitive prefixes, and a suffix with arithmetic sequence with difference 1). Below in Corollary 1 we give a different example, that in a Fibonacci string all but a logarithmic number of prefixes have a proper cover and (except for a short prefix) no *two* consecutive positions of the cover array form an arithmetic sequence of difference 1. The cover array of a Fibonacci string contains a logarithmic number of different values.

Let us recall that the Fibonacci strings are defined as follows:  $\text{Fib}_0 = \mathbf{b}$ ,  $\text{Fib}_1 = \mathbf{a}$ , and  $\text{Fib}_m = \text{Fib}_{m-1}\text{Fib}_{m-2}$  for  $m > 1$ . All covers of whole Fibonacci strings (as well as other types of quasiperiodicity) were characterized in [13] (see also [37] for similar results on Tribonacci strings). Moreover, a complete characterization of the lengths of shortest covers of cyclic shifts of Fibonacci strings was shown [17]. However, apparently, the structure of the cover array of Fibonacci strings was not studied before. The theorem below shows the recursive structure of the array; see Fig. 1 for a concrete example.

Let  $\text{Fib}$  be the infinite Fibonacci string (the limit of strings  $\text{Fib}_m$ ). For any  $m > 0$ ,  $\text{Fib}_m$  is a prefix of  $\text{Fib}$ , and hence also  $\text{Cov}_{\text{Fib}_m}$  is a prefix of  $\text{Cov}_{\text{Fib}}$ . Thus it is enough to characterize the values of  $\text{Cov}_{\text{Fib}}$ . Let  $F_k = |\text{Fib}_k|$ .

**Theorem 5.** *In the corner cases  $\text{Cov}_{\text{Fib}}[\ell]$  is equal to*

- $\ell$  if  $\ell \leq 2$ ,
- 3 if  $\ell = F_k$  for odd  $k \geq 3$ ,
- 5 if  $\ell = F_k$  for even  $k \geq 4$ ,
- $\ell$  if  $\ell = F_k - 1$  or  $\ell = 2F_k - 1$  for  $k \geq 4$ .

*Otherwise,  $\text{Cov}_{\text{Fib}}[\ell] = \text{Cov}_{\text{Fib}}[\ell - F_{k-1}]$ , where  $F_k < \ell < F_{k+1}$ .*

*Proof.* It is well-known that for any  $m \geq 1$ ,  $\text{LCP}(\text{Fib}_{m+1}, \text{Fib}_{m-1}\text{Fib}_m) = F_{m+1} - 2$ ; see e.g. [28]. In particular,  $F_{m-1}$  is a period of  $\text{Fib}_{m+1}[0..F_{m+1} - 2) =$



$(\text{Fib}_{m-1}\text{Fib}_m)[0..F_{m+1}-2] = (\text{Fib}_{m-1}\text{Fib}_{m-1}\text{Fib}_{m-2})[0..F_{m+1}-2]$ , but not a period of  $\text{Fib}_{m+1}[0..F_{m+1}-1]$ . Equivalently,  $\text{Fib}[0..\ell - F_{m-1}]$  is a border of  $\text{Fib}[0..\ell]$  if and only if  $F_{m-1} < \ell \leq F_{m+1} - 2$ .

The value of  $\text{Cov}_{\text{Fib}}[F_k]$  as well as of  $\text{Cov}_{\text{Fib}}[\ell]$  for  $\ell \leq 2$  follows from [13]. From the same paper we know that  $\text{Fib}_{k-2}$  is the longest proper border of  $\text{Fib}_k$ . Moreover,  $\text{Fib}[0..F_{k-2}-1]$  is the longest border of  $\text{Fib}[0..F_k-1]$ . Indeed, an existence of a longer border (of length different than  $F_{k-1}-1$ ) would result in  $\text{Fib}[0..F_k-2]$  having period 1 by the periodicity lemma (as it already has periods  $F_{k-1}$  and  $F_{k-2}$ ; see [28]).

We will prove by induction that  $\text{Cov}_{\text{Fib}}[F_k - 1] = F_k - 1$  for  $k \geq 4$ . The base case holds. By the above, the only candidate for the length of a proper cover of  $\text{Cov}_{\text{Fib}}[F_k - 1]$  is  $\text{Cov}_{\text{Fib}}[F_{k-2} - 1]$ , which equals  $F_{k-2} - 1$  by induction. Prefix  $\text{Fib}[0..F_{k-2}-1]$  has an occurrence at positions 0 and  $F_{k-2}$  in this  $\text{Fib}$ , but the position  $F_{k-2} - 1$  remains uncovered; existence of yet another occurrence that contains this position would result in a long overlap of occurrences which, in turn, would result in the string  $\text{Fib}[0..F_{k-2}-1]$  being periodic, which is not the case. Hence,  $\text{Cov}_{\text{Fib}}[F_k - 1] = F_k - 1$ .

Next we prove by induction that  $\text{Cov}_{\text{Fib}}[2F_k - 1] = 2F_k - 1$  for  $k \geq 4$ . Similarly,  $\text{Fib}[0..F_k-1]$  is the longest border of  $\text{Fib}[0..2F_k-1]$ . By the inductive hypothesis,  $\text{Cov}_{\text{Fib}}[F_k - 1] = F_k - 1$  is the only candidate for the length of a proper cover of  $\text{Fib}[0..2F_k - 1]$ . By exactly the same argument as in the previous case, the position  $F_k - 1$  in  $\text{Fib}$  is not covered by this candidate. Thus  $\text{Cov}_{\text{Fib}}[2F_k - 1] = 2F_k - 1$ .

We have  $\text{Cov}_{\text{Fib}}[6] = 3$ . Now, for  $k \geq 5$ , let  $\ell \in [F_k + 1..2F_{k-1} - 2] \cup [2F_{k-1}..F_{k+1} - 2]$ . As noted,  $\text{Fib}[0..\ell - F_{k-1}]$  is a border of  $\text{Fib}[0..\ell]$  (since  $\ell \leq F_{k+1} - 2$ ). Additionally, if  $\ell \leq 2F_{k-1} - 2$ , string  $\text{Fib}[0..\ell - F_{k-1}]$  also appears in  $\text{Fib}$  at position  $F_{k-2}$  (by the LCP equality from the beginning of the proof). Those two or three occurrences cover all the positions of  $\text{Fib}[0..\ell]$ , hence a cover of  $\text{Fib}[0..\ell - F_{k-1}]$  is also a cover of  $\text{Fib}[0..\ell]$ . At the same time a shortest cover of  $\text{Fib}[0..\ell]$  has to be a cover of a border that is a cover, hence  $\text{Cov}_{\text{Fib}}[\ell] = \text{Cov}_{\text{Fib}}[\ell - F_{k-1}]$ .  $\square$

**Corollary 1.** *For any  $m \geq 1$ , the array  $\text{Cov}_{\text{Fib}_m}$  contains  $\Theta(m)$  different values. Only  $\Theta(m)$  prefixes of  $\text{Fib}_m$  are superprimitive. Moreover, for all  $\ell \in [5..F_m]$ , we have  $\text{Cov}_{\text{Fib}_m}[\ell] + 1 \neq \text{Cov}_{\text{Fib}_m}[\ell + 1]$ .*

*Proof.* The first two statements follow readily from Theorem 5. As for the third statement, among the distinct values in  $\text{Cov}_{\text{Fib}}$  from position 5 onwards, the only pairs of consecutive numbers are (3, 4) and (4, 5). (This is because for large enough  $k$ , values  $F_{k+1} - 1$  and  $2F_k - 1$  differ by more than 1.) Therefore, if  $\ell \geq 5$  would be the smallest position such that  $\text{Cov}_{\text{Fib}}[\ell + 1] = \text{Cov}_{\text{Fib}}[\ell] + 1$ , then  $\ell$  or  $\ell + 1$  would be equal to  $F_k$  for some  $k \geq 5$ . By the recursion in Theorem 5, if  $\ell + 1 = F_k$ , then  $\text{Cov}_{\text{Fib}}[\ell] = \ell > 5$  and  $\text{Cov}_{\text{Fib}}[\ell + 1] \in \{3, 5\}$ , so two consecutive values are not possible. If  $\ell = F_k$  and  $\text{Cov}_{\text{Fib}}[\ell] = 3$ , then  $\text{Cov}_{\text{Fib}}[\ell + 1] = 9$  by easy induction, so again two consecutive values on consecutive positions are not possible.  $\square$

The recursive characterization of Theorem 5 allows to compute any element of the cover array of  $\text{Fib}_m$  in  $\mathcal{O}(\log n)$  time, where  $n = F_m$ , without additional space. By Corollary 1, the cover array of  $\text{Fib}_m$  has only  $\mathcal{O}(\log n)$  different values, which allows one to store the cover array of  $\text{Fib}_m$  in a packed form in  $\mathcal{O}(n \log \log n / \log n)$  space so that its elements can be retrieved in  $\mathcal{O}(1)$  time. In the next subsection we show that an equally space-efficient representation exists for every string over a constant-sized alphabet.

## 4.2 Proof of Theorem 2

We use the following known corollary of the periodicity lemma.

**Lemma 5 ([7,35]).** *If  $|X| < |Y| < 2|X|$  are two strings and  $X$  has at least three occurrences in  $Y$  as a substring, then  $X$  is periodic.*

Before we describe the data structure, let us give some intuition.

Assume that  $\text{Cov}_T[\ell] = c$  with  $c < \ell$ . That is, string  $C = T[0..c)$  is a proper shortest cover of a prefix  $T[0.. \ell)$ . If the second occurrence of  $C$  in  $T$  is at position  $j > 0$ , then  $U^2 = T[0..2j)$  is a square. Further,  $j > c/2$ , as otherwise  $C$  would be periodic. Hence,  $C$  is a prefix of  $T[0..2j)$ . This concludes that the square  $T[0..2j)$  is primitively rooted, as otherwise  $C$  would be periodic. By Lemma 2, there are only  $\mathcal{O}(\log n)$  primitively rooted square prefixes of  $T$ . Thus, if  $C$  is a proper shortest cover of a prefix of  $T$ , we can assign to  $C$  one of  $\mathcal{O}(\log n)$  primitively rooted square prefixes of  $T$ .

Let  $P = T[0..p)$  be the shortest aperiodic prefix of  $T$  such that  $p \geq j$ . As  $T[0..c) = C$  is aperiodic,  $p$  is well-defined and  $C$  has a prefix  $P$ . Thus, if  $C$  is a proper shortest cover of a prefix of  $T$ , this allows to assign to  $C$  one of  $\mathcal{O}(\log n)$  aperiodic prefixes of  $T$ .

For  $k = \ell - c$ , we have  $T[k..k+p) = P$ . We observe that there can be no further occurrence of  $P$  in  $T$  at a position in  $(k.. \ell - p)$  (that is, no further occurrence of  $P$  in  $T[0..c)$ ). Indeed, such an occurrence would be a substring of  $C$ , so it would imply an occurrence of  $P$  in  $T$  at a position in  $[1..j)$ . By Lemma 5, this would contradict the fact that  $P$  is aperiodic. In summary, if  $C$  is a proper shortest cover of a prefix  $T[0.. \ell)$ , then  $C$  can be uniquely identified by the rightmost occurrence in  $T[0.. \ell)$  of the aperiodic prefix  $P$  of  $T$  that is assigned to  $C$ . Moreover, the occurrence is at one of the positions in  $(\ell - 2p.. \ell)$ , as  $2p \geq 2j > c$ .

**Data structure:** Let  $j_1, \dots, j_t$  be the half lengths of all primitively rooted square prefixes of  $T$ . By Lemma 2, we have  $t = \mathcal{O}(\log n)$ . The data structure stores  $t$  lengths of aperiodic prefixes of  $T$ ,  $p_1, \dots, p_t$ . For every  $i \in [1..t]$ ,  $p_i$  is the length of the shortest aperiodic prefix of  $T[0..2j_i)$  of length at least  $j_i$ . (It is known that such a prefix exists, as  $T[0..2j_i - 1)$  is aperiodic by Lemma 1.)

For each  $\ell \in [1..n]$ , we store a bit  $sp[\ell]$  that equals 1 if and only if  $T[0.. \ell)$  is superprimitive. If  $sp[\ell] = 0$ , a number  $pref[\ell] \in [1..t]$  is stored that determines the aperiodic prefix  $P = T[0..p_{pref[\ell]})$  of  $T$  that corresponds to the shortest cover  $C$  of  $T[0.. \ell)$ , as discussed above. Precisely, if  $j_i$  is the position of the

second occurrence of  $C$  in  $T$ , then  $\text{pref}[\ell] = i$ . Finally, a data structure for IPM queries in  $T$  is stored.

Overall, provided that the arrays  $sp$  and  $\text{pref}$  are stored in a packed form, the space complexity is  $\mathcal{O}(\log n + n \log \log n / \log n + n / \log n + n / \log_\sigma n) = \mathcal{O}(n \log \log n / \log n + n / \log_\sigma n)$ , as required.

**Queries:** To compute  $\text{Cov}_T[\ell]$ , we first check if  $sp[\ell] = 1$  and, if that is the case, return  $\ell$ . Otherwise, we ask an IPM query to compute the rightmost occurrence of  $P = T[0..p_{\text{pref}[\ell]})$  in  $T[\ell - 2 \cdot p_{\text{pref}[\ell]} + 1.. \ell)$ . As  $P$  is aperiodic, there are at most two such occurrences. We select as  $k$  the starting position of the rightmost occurrence. The shortest cover of  $T[0.. \ell)$  is  $T[k.. \ell)$  (i.e.,  $\text{Cov}_T[\ell] = \ell - k$ ).

By [30], the query time complexity is  $\mathcal{O}(1)$ . This concludes the proof of Theorem 2.

## 5 Lower Bound on the Complexity in the PILLAR Model

### 5.1 The PILLAR model

Let us start first formally introduce the primitives of the PILLAR model [11]. The argument strings are fragments of strings in a given collection  $\mathcal{X}$ :

- $\text{Extract}(S, \ell, r)$ : Retrieve string  $S[\ell..r)$ .
- $\text{LCP}(X, Y)$ ,  $\text{LCP}_R(X, Y)$ : Compute the length of the longest common prefix/suffix of  $X$  and  $Y$ .
- $\text{IPM}(X, Y)$ : Assuming that  $|Y| \leq 2|X|$ , compute the starting positions of all exact occurrences of  $X$  in  $Y$ , expressed as an arithmetic progression.
- $\text{Access}(S, i)$ : Retrieve the letter  $S[i]$ ;
- $\text{Length}(S)$ : Compute the length  $|S|$  of the string  $S$ .

The runtime of algorithms in this model can be expressed in terms of the number of primitive PILLAR operations (and additional operations not performed on the strings themselves).

### 5.2 Lower Bound

We focus on checking if a string over an alphabet  $\{\mathbf{a}, \mathbf{b}\}$  is covered by its border  $\mathbf{aba}$ . Strings covered by  $\mathbf{aba}$  are formed of concatenations of strings of a form  $(\mathbf{ab})^k \mathbf{a}$  for  $k \geq 1$ ; equivalently, strings that have  $\mathbf{aba}$  as a border and do not contain a substring  $\mathbf{bb}$  or  $\mathbf{aaa}$ .

For infinitely many positive integers  $n$ , we show a strategy for an adversary to answer  $Cn / \log n$  PILLAR queries on a length- $n$  binary string, for a certain constant  $C > 0$ , after which the adversary still has the choice of fixing the string in two ways: in one  $T$  has a cover  $\mathbf{aba}$ , and in the other  $T$  is superprimitive (i.e., it has no proper cover).

We define a morphism  $\phi : \{0, 1\} \mapsto \{\mathbf{a}, \mathbf{b}\}$ :

- $\phi(0) = \mathbf{abababa} \mathbf{aba} \mathbf{ababa} = (\mathbf{ab})^3 \mathbf{a} (\mathbf{ab}) \mathbf{a} (\mathbf{ab})^2 \mathbf{a}$

$$- \phi(1) = \text{abababa ababa aba} = (\text{ab})^3 \text{a} (\text{ab})^2 \text{a} (\text{ab}) \text{a}$$

Both  $\phi(0)$  and  $\phi(1)$  have length 15 and have a cover **aba**. Thus  $\phi(S)$ , for any string  $S$  over alphabet  $\{0, 1\}$ , has a cover **aba**.

Let us recall that a de Bruijn sequence of order  $k$  over an alphabet  $\Sigma$  is a string of length  $|\Sigma|^k + k - 1$  over the alphabet  $\Sigma$  such that its every substring of length  $k$  is distinct. It is well-known that such sequences exist for every finite alphabet  $\Sigma$  and integer  $k \geq 1$  [8].

Let  $B_k$  be a de Bruijn sequence of order  $k$  over the binary alphabet  $\{0, 1\}$ . We apply the morphism  $\phi$  on  $B$  to obtain a string  $T_k$  over alphabet  $\{\mathbf{a}, \mathbf{b}\}$  of length  $15 \cdot (2^k + k - 1)$ . Due to the property of de Bruijn sequences, each substring of  $T$  of length at least  $15(k + 1) - 1$  is distinct. Indeed, every “aligned” substring of length  $15k$  starting at a position divisible by 15 in  $T_k$  is distinct, and every substring of  $T_k$  of length  $15(k + 1) - 1$  contains an “aligned” substring of length  $15k$ .

Due to this property, an answer to an LCP or  $\text{LCP}_R$  query on  $T_k$  for two different positions is always bounded from above by  $15(k + 1) - 2$ . Similarly for the IPM queries; if we query for a substring of length at least  $15(k + 1) - 1$ , then we do not gain any interesting information (the only occurrence of the substring is the one used to ask the query). On the other hand, by asking an IPM query for a shorter substring we only gain information about a part of  $T_k$  of length at most  $30(k + 1) - 2$ .

Formally, the strategy of the adversary for a text  $T$  of length  $n = |T_k|$  is as follows. Queries **Extract**, **Access**, **Length**, **LCP**,  $\text{LCP}_R$  are answered as in  $T_k$ . An  $\text{IPM}(X, Y)$  query for  $|X| < 15(k + 1) - 1$  is also answered as in  $T_k$ . Finally, to answer an  $\text{IPM}(X, Y)$  query for  $|X| \geq 15(k + 1) - 1$ , we refer to the fragments  $T[i_x \dots j_x] = X$  and  $T[i_y \dots j_y] = Y$  and return an occurrence of  $X$  in  $Y$  at position  $i_y - i_x$  if  $[i_x \dots j_x] \subseteq [i_y \dots j_y]$  and no occurrence otherwise.

We say that a position  $i \in [0 \dots n)$  of  $T$  has been touched if the algorithm has performed (1) an **Access** query on  $T[i]$ , or (2) an  $\text{LCP}(T[i_x \dots j_x], T[i_y \dots j_y])$  query such that  $i \in [i_x \dots i_x + \ell) \cup [i_y \dots i_y + \ell)$  where  $\ell$  is the result of the **LCP** query, or (3) similarly an  $\text{LCP}_R$  query such that  $i$  belongs to the computed  $\text{LCP}_R$  of one of the two queried substrings of  $T$ , or (4) an  $\text{IPM}(T[i_x \dots j_x], T[i_y \dots j_y])$  query for  $j_x - i_x + 1 < 15(k + 1) - 1$  such that  $i \in [i_x \dots j_x] \cup [i_y \dots j_y]$ . In total, after  $q$  **PILLAR** operations, fewer than  $45q(k + 1) \leq 90kq$  positions of  $T$  have been touched. Thus after  $q = \lfloor 2^k / (6k) \rfloor$  operations, there still exists a position in  $T$  that has not been touched. Assume  $i$  is such a position. Then the adversary can make the choice to set  $T[i]$  as  $T_k[i]$  or as the letter different from  $T_k[i]$ ; all the remaining untouched positions are set as in  $T_k$ . If  $T[i] = T_k[i]$ ,  $T = T_k$  has a cover **aba**. If  $T[i] \neq T_k[i]$ ,  $T$  contains exactly one substring  $\mathbf{a}^s$  for some  $s \in [3 \dots 5]$ , or exactly one substring  $\mathbf{b}^t$ , for some  $t \in [2 \dots 3]$ . It is easy to see that in this case  $T$  is superprimitive.

There exists a constant  $C > 0$  (for example,  $C = 1/180$ ) such that the selected value of  $q$  satisfies  $q \geq Cn / \log n$ . Theorem 3 is proved.

## 6 Open Problems

It remains open if the data structure of Theorem 2 can be constructed in sublinear time or if its space complexity can be decreased to  $\mathcal{O}(n/\log n)$  for  $\sigma = \mathcal{O}(1)$ .

Future work also includes designing sublinear-time algorithms for other notions of quasiperiodicity for which  $\mathcal{O}(n)$ -time algorithms are already known, for a length- $n$  string over an integer alphabet; this includes, for example, seeds [29,36], enhanced covers [21], and partial covers [36].

## References

1. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings. *Inf. Process. Lett.* **39**(1), 17–20 (1991). [https://doi.org/10.1016/0020-0190\(91\)90056-N](https://doi.org/10.1016/0020-0190(91)90056-N)
2. Bannai, H., Ellert, J.: Lyndon arrays in sublinear time. In: Gørtz, I.L., Farach-Colton, M., Puglisi, S.J., Herman, G. (eds.) 31st Annual European Symposium on Algorithms, ESA 2023, September 4–6, 2023, Amsterdam, The Netherlands. LIPIcs, vol. 274, pp. 14:1–14:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICSA.2023.14>
3. Belazzougui, D., Kosolobov, D., Puglisi, S.J., Raman, R.: Weighted ancestors in suffix trees revisited. In: Gawrychowski, P., Starikovskaya, T. (eds.) 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5–7, 2021, Wrocław, Poland. LIPIcs, vol. 191, pp. 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPICSA.CPM.2021.8>
4. Ben-Kiki, O., Bille, P., Breslauer, D., Gasieniec, L., Grossi, R., Weimann, O.: Towards optimal packed string matching. *Theor. Comput. Sci.* **525**, 111–129 (2014). <https://doi.org/10.1016/J.TCS.2013.06.013>
5. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Panario, D., Viola, A. (eds.) LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10–14, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1776, pp. 88–94. Springer (2000). [https://doi.org/10.1007/10719839\\_9](https://doi.org/10.1007/10719839_9)
6. Breslauer, D.: An on-line string superprimitivity test. *Inf. Process. Lett.* **44**(6), 345–347 (1992). [https://doi.org/10.1016/0020-0190\(92\)90111-8](https://doi.org/10.1016/0020-0190(92)90111-8)
7. Breslauer, D., Galil, Z.: Finding all periods and initial palindromes of a string in parallel. *Algorithmica* **14**(4), 355–366 (1995). <https://doi.org/10.1007/BF01294132>
8. de Bruijn, N.G.: A combinatorial problem. *Indagationes Math.* **8**, 461–467 (1946), <http://www.dwc.knaw.nl/DL/publications/PU00018235.pdf>
9. Charalampopoulos, P., Kociumaka, T., Pissis, S.P., Radoszewski, J.: Faster algorithms for longest common substring. In: Mutzel, P., Pagh, R., Herman, G. (eds.) 29th Annual European Symposium on Algorithms, ESA 2021, September 6–8, 2021, Lisbon, Portugal (Virtual Conference). LIPIcs, vol. 204, pp. 30:1–30:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPICSA.2021.30>
10. Charalampopoulos, P., Kociumaka, T., Wellnitz, P.: Faster approximate pattern matching: A unified approach. In: Irani, S. (ed.) 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020,

- Durham, NC, USA, November 16-19, 2020. pp. 978–989. IEEE (2020). <https://doi.org/10.1109/FOCS46700.2020.00095>
11. Charalampopoulos, P., Kociumaka, T., Wellnitz, P.: Faster approximate pattern matching: A unified approach. In: 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020. pp. 978–989. IEEE (2020). <https://doi.org/10.1109/FOCS46700.2020.00095>, full version: arXiv:2004.08350v2
  12. Charalampopoulos, P., Pissis, S.P., Radoszewski, J.: Longest palindromic substring in sublinear time. In: Bannai, H., Holub, J. (eds.) 33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic. LIPIcs, vol. 223, pp. 20:1–20:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). <https://doi.org/10.4230/LIPICs.CPM.2022.20>
  13. Christou, M., Crochemore, M., Iliopoulos, C.S.: Quasiperiodicities in Fibonacci strings. *Ars Comb.* **129**, 211–225 (2016), <https://arxiv.org/abs/1201.6162>
  14. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., Tyczyński, W., Waleń, T.: The maximum number of squares in a tree. In: Kärkkäinen, J., Stoye, J. (eds.) Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7354, pp. 27–40. Springer (2012). [https://doi.org/10.1007/978-3-642-31265-6\\_3](https://doi.org/10.1007/978-3-642-31265-6_3)
  15. Crochemore, M., Iliopoulos, C.S., Pissis, S.P., Tischler, G.: Cover array string reconstruction. In: Amir, A., Parida, L. (eds.) Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6129, pp. 251–259. Springer (2010). [https://doi.org/10.1007/978-3-642-13509-5\\_23](https://doi.org/10.1007/978-3-642-13509-5_23)
  16. Crochemore, M., Iliopoulos, C.S., Radoszewski, J., Rytter, W., Straszynski, J., Waleń, T., Zuba, W.: Internal quasiperiod queries. In: Boucher, C., Thankachan, S.V. (eds.) String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12303, pp. 60–75. Springer (2020). [https://doi.org/10.1007/978-3-030-59212-7\\_5](https://doi.org/10.1007/978-3-030-59212-7_5)
  17. Crochemore, M., Iliopoulos, C.S., Radoszewski, J., Rytter, W., Straszynski, J., Waleń, T., Zuba, W.: Shortest covers of all cyclic shifts of a string. *Theor. Comput. Sci.* **866**, 70–81 (2021). <https://doi.org/10.1016/J.TCS.2021.03.011>
  18. Crochemore, M., Rytter, W.: Squares, cubes, and time-space efficient string searching. *Algorithmica* **13**(5), 405–425 (1995). <https://doi.org/10.1007/BF01190846>
  19. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997. pp. 137–143. IEEE Computer Society (1997). <https://doi.org/10.1109/SFCS.1997.646102>
  20. Fine, N.J., Wilf, H.S.: Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society **16**(1), 109–114 (1965). <https://doi.org/10.2307/2034009>
  21. Flouri, T., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Puglisi, S.J., Smyth, W.F., Tyczyński, W.: Enhanced string covering. *Theor. Comput. Sci.* **506**, 102–114 (2013). <https://doi.org/10.1016/J.TCS.2013.08.013>
  22. Ganardi, M., Jeż, A., Lohrey, M.: Balancing straight-line programs. *J. ACM* **68**(4), 27:1–27:40 (2021). <https://doi.org/10.1145/3457389>
  23. Hariharan, R., Vinay, V.: String matching in  $\tilde{O}(\sqrt{n} + \sqrt{m})$  quantum time. *J. Discrete Algorithms* **1**(1), 103–110 (2003). [https://doi.org/10.1016/S1570-8667\(03\)00010-8](https://doi.org/10.1016/S1570-8667(03)00010-8)

24. I, T., Matsubara, W., Shimohira, K., Inenaga, S., Bannai, H., Takeda, M., Narisawa, K., Shinohara, A.: Detecting regularities on grammar-compressed strings. *Inf. Comput.* **240**, 74–89 (2015). <https://doi.org/10.1016/J.IC.2014.09.009>
25. Jin, C., Nogler, J.: Quantum speed-ups for string synchronizing sets, longest common substring, and  $k$ -mismatch matching. In: Bansal, N., Nagarajan, V. (eds.) *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*. pp. 5090–5121. SIAM (2023). <https://doi.org/10.1137/1.9781611977554.CH186>
26. Kempa, D., Kociumaka, T.: String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In: Charikar, M., Cohen, E. (eds.) *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*. pp. 756–767. ACM (2019). <https://doi.org/10.1145/3313276.3316368>
27. Kempa, D., Kociumaka, T.: Dynamic suffix array with polylogarithmic queries and updates. In: Leonardi, S., Gupta, A. (eds.) *STOC 2022: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*. pp. 1657–1670. ACM (2022). <https://doi.org/10.1145/3519935.3520061>
28. Knuth, D.E., Jr., J.H.M., Pratt, V.R.: Fast pattern matching in strings. *SIAM J. Comput.* **6**(2), 323–350 (1977). <https://doi.org/10.1137/0206024>
29. Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: A linear-time algorithm for seeds computation. *ACM Trans. Algorithms* **16**(2), 27:1–27:23 (2020). <https://doi.org/10.1145/3386369>
30. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal pattern matching queries in a text and applications. *CoRR* **abs/1311.6235** (2013), <http://arxiv.org/abs/1311.6235>
31. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal pattern matching queries in a text and applications. In: Indyk, P. (ed.) *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*. pp. 532–551. SIAM (2015). <https://doi.org/10.1137/1.9781611973730.36>
32. Mitani, K., Mieno, T., Seto, K., Horiyama, T.: Shortest cover after edit. *CoRR* **abs/2402.17428** (2024). <https://doi.org/10.48550/ARXIV.2402.17428>
33. Moore, D.W.G., Smyth, W.F.: A correction to "An optimal algorithm to compute all the covers of a string". *Inf. Process. Lett.* **54**(2), 101–103 (1995). [https://doi.org/10.1016/0020-0190\(94\)00235-Q](https://doi.org/10.1016/0020-0190(94)00235-Q)
34. Munro, J.I., Navarro, G., Nekrich, Y.: Text indexing and searching in sublinear time. In: Gørtz, I.L., Weimann, O. (eds.) *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark. LIPIcs*, vol. 161, pp. 24:1–24:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/LIPICS.CPM.2020.24>
35. Plandowski, W., Rytter, W.: Application of lempel-ziv encodings to the solution of words equations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings. Lecture Notes in Computer Science*, vol. 1443, pp. 731–742. Springer (1998). <https://doi.org/10.1007/BFB0055097>
36. Radoszewski, J.: Linear time construction of cover suffix tree and applications. In: Gørtz, I.L., Farach-Colton, M., Puglisi, S.J., Herman, G. (eds.) *31st Annual European Symposium on Algorithms, ESA 2023, September 4-6, 2023, Amsterdam, The Netherlands. LIPIcs*, vol. 274, pp. 89:1–89:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2023). <https://doi.org/10.4230/LIPICS.ESA.2023.89>

37. Singh, M.: Quasiperiodicity in Tribonacci Word (Mar 2020), <https://hal.science/hal-02141636>, working paper or preprint