Quasi-Periodicity in Streams

Paweł Gawrychowski
University of Wrocław, 50-137 Wrocław, Poland
gawry@cs.uni.wroc.pl

Jakub Radoszewski
Institute of Informatics, University of Warsaw, 02-097 Warsaw, Poland
jrad@mimuw.edu.pl

Tatiana Starikovskaya
DIENS, École normale supérieure, PSL Research University, 75005 Paris, France
tat.starikovskaya@gmail.com

Abstract

In this work, we show two streaming algorithms for computing the length of the shortest cover of a string of length n. We start by showing a two-pass algorithm that uses $\mathcal{O}(\log^2 n)$ space and then show a one-pass streaming algorithm that uses $\mathcal{O}(\sqrt{n \log n})$ space. Both algorithms run in near-linear time. The algorithms are randomized and compute the answer incorrectly with probability inverse-polynomial in n. We also show that there is no sublinear-space streaming algorithm for computing the length of the shortest seed of a string.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Streaming algorithms, quasi-periodicity, covers, seeds

Digital Object Identifier 10.4230/LIPIcs.CPM.2019.22

Funding Jakub Radoszewski is supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

1 Introduction

One of the major tasks in processing data streams is trend analysis. In this work, we focus on a specific representative trend of streaming data, namely, *periodicity*. The motivation for analyzing the periodicity trend is that it can be used for detecting anomalies in streams, for example, in streams of financial data.

The study of periodicity in data streams was initiated by Ergün et al. in [20]. For a stream of length n, they showed a one-pass streaming algorithm to compute the length of the shortest period of the stream in polylogarithmic space assuming that the length of the period is at most $n/2$. On the other hand, they showed that there is no sublinear-space algorithm for computing periods of length larger than $n/2$. Motivated by real-life applications, where data streams are almost never exactly periodic, this work was later extended to allow approximate periods with mismatches and wildcards [18, 19].

We consider a different relaxation of the notion of periods, that of *quasi-periodicity*. Quasi-periodicity has been extensively studied in the RAM model of computation, starting from the early works of Apostolico and Ehrenfeucht [7], and by now is a well-established approach to detecting repetitive structure of a string when the classical definition of periodicity fails. There are two basic definition of quasi-periods that allow detecting different kinds of repetitive structure of a string: *covers* and *seeds*. Informally, a cover of a string T is a substring C of T such that every position of T lies within some occurrence of C. A string S is said to be a seed of T if S is a cover of some string containing T as a substring. The shortest cover and the shortest seed of a string can be computed in linear time; see [8] and [30, 31] (and

a much older $\mathcal{O}(n \log n)$-time algorithm [27]), respectively. Other works include computing all covers [37, 38] and shortest covers of all prefixes of a string [2, 11, 34]; left and right seeds being notions intermediate between covers and seeds [14, 16]; combinatorial studies on covers [3, 15]; computing approximate quasi-periodicities called enhanced covers [1, 22], partial covers [32, 36], partial seeds [33], approximate covers [4–6, 39], approximate seeds [13], and other variations [25, 35]; as well as quasi-periodicities that consist of multiple strings, called λ-covers, k-covers, and λ-seeds [17, 23, 24, 26, 41].

In this work, we commence a study of quasi-periodicity in streams. Recall that in the streaming model of computation, the input arrives as a stream, one character at a time, and we must account for all the space used, including the space used to store the input. We show two streaming algorithms for computing the shortest cover of a string of length n. We start by showing a two-pass algorithm that uses $\mathcal{O}(\log^2 n)$ space (Section 3, Theorem 12) and then show a one-pass streaming algorithm that uses $\mathcal{O}(\sqrt{n \log n})$ space (Section 4, Theorem 18). Both algorithms run in near-linear time. The algorithms are randomized and compute the answer incorrectly with probability inverse-polynomial in n. We also show that there is no sublinear-space one-pass streaming algorithm for computing the shortest seed of a string (Section 6).

2 Preliminaries

We assume that the characters of a string T are numbered 1 through $|T|$. By $T[i]$ we denote the i-th character of T and by $T[i, j]$ we denote $T[i] \dots T[j]$ which we call a substring of T. If $i = 1$, the substring $T[i, j]$ is called a prefix of T, and if $j = |T|$, a suffix of T. For strings T and X, we denote $\mathsf{Occ}_T(X) = \{i : T[i, i + |X| - 1] = X\}$.

2.1 Periods and quasi-periods

▶ **Definition 1** (Periods, borders). *We say that a positive integer p is a* period *of a string T if $T[i] = T[i + p]$ for all $i = 1, \dots, |T| - p$. By $\mathsf{per}(T)$ we denote the smallest period of T. The string T is called* periodic *if $2\,\mathsf{per}(T) \le |T|$. We say that a string B is a border of T if B is both a prefix and a suffix of T.*

▶ **Lemma 2** (Fine and Wilf's periodicity lemma [21]). *If a string Q has two periods p and q and $p + q \le |Q|$, then Q also has a period $\gcd(p, q)$.*

Let $\mathsf{Occ}_T(X) = \{a_1, \dots, a_m\}$ such that $a_1 < \dots < a_m$. We say that $a_j, \dots, a_k$, for $1 \le j \le k \le m$, form a *chain* of occurrences of X in T if for every $i = j + 1, \dots, k$, we have $a_i - a_{i-1} \le \frac{|X|}{2}$. A chain is called *maximal* if $k = m$ or $a_{k+1} - a_k > \frac{|X|}{2}$.

▶ **Corollary 3** (of Lemma 2). *Every chain of occurrences of X in T is an arithmetic progression with difference $\mathsf{per}(X)$.*

Proof. Assume to the contrary that $a_i - a_{i-1} \neq \mathsf{per}(X) = d$ for some chain $a_j, \dots, a_k$ and $j < i \le k$. Then $a_i - a_{i-1}$ is a period of X. By the periodicity lemma, it is a multiple of d. Hence, $a_{i-1} + d \in \mathsf{Occ}_T(X)$, a contradiction. ◀

For a set of integers $A = \{a_1, \dots, a_m\}$, $a_1 < a_2 < \dots < a_m$, by $\mathsf{maxgap}(A)$ we denote the maximum distance between consecutive elements of A: $\mathsf{maxgap}(A) = \max\{a_i - a_{i-1} : i = 2, \dots, m\}$.

▶ **Definition 4** (Covers, seeds). *A string C is a* cover *of a string T if*

$$\mathsf{maxgap}(\mathsf{Occ}_T(C) \cup \{-|C|+1, |T|+1\}) = |C|.$$

We say that a string C is a seed *of T if $|C| \leq |T|$ and C is a cover of some string containing T as a substring.*

▶ **Example 5.** *aabaa* is the shortest cover and *aba* is a shortest seed of *aabaabaa*.

▶ **Observation 6.** *Any cover of a string T is a border of T. Moreover, the shortest cover of T is not periodic.*

2.2 Reminder: Streaming pattern matching

▶ **Definition 7** (Rabin–Karp fingerprint [28, 40]). *The Rabin–Karp fingerprint of a string $X = X[1] \dots X[\ell]$ is defined as $\varphi(X) = (\sum_{i=1}^{\ell} X[i] \cdot r^i) \bmod p$, where p is a prime and r is a random integer in $\mathbb{F}_p$. We assume that together with the Rabin–Karp fingerprint we store $r^\ell \bmod p$ and $r^{-\ell} \bmod p$.*

If we choose p to be large enough, then the collision probability of any two ℓ-length strings X and Y, where $\ell \leq n$, will be at most $1/n^{\mathcal{O}(1)}$ [28, 40]. We will also need the following fact which follows immediately from the definition.

▶ **Fact 8.** *Let X, Y be two strings and $Z = XY$ be their concatenation. From the Rabin–Karp fingerprints of two of the strings X, Y, Z one can compute the Rabin–Karp fingerprint of the third string in $\mathcal{O}(1)$ time using the formula:*

$$\varphi(Z) = (\varphi(X) + r^{|X|} \cdot \varphi(Y)) \bmod p.$$

▶ **Corollary 9.** *Let X be a string and $Z = X^\alpha$ be the α-th power of X for positive integer α, i.e. the concatenation of α copies of X. From the Rabin–Karp fingerprint of X one can compute the Rabin–Karp fingerprint of Z in $\mathcal{O}(\log \alpha)$ time.*

We recall the main idea of the streaming pattern matching algorithm by Porat and Porat [40] in a simplified form by Breslauer and Galil [12]. The algorithm takes as an input a pattern Q and a streaming text T of length n, and outputs all occurrences of Q in T. The algorithm is randomized and can output an incorrect answer with probability inverse-polynomial in n. It uses $\mathcal{O}(\log |Q|)$ space and takes $\mathcal{O}(\log |Q|)$ time per text character.

We assume that the algorithm receives the pattern first, in a form of a stream, and computes the Rabin–Karp fingerprints of Q and of the prefixes $Q[1, 2^j]$ for all j. During the main stage, the algorithm stores $\mathcal{O}(\log |Q|)$ levels of positions of T. Positions in level j are occurrences of $Q[1, 2^j]$ in the suffix of the current text T of length 2^{j+1}. The algorithm stores the Rabin–Karp fingerprints of the prefixes of T up to each of these positions. If there are at least 3 such positions at one level, then, by the periodicity lemma, all the positions form a chain (that is, a single arithmetic progression with difference $\mathsf{per}(Q[1, 2^j])$; see Corollary 3). This allows storing the aforementioned information very compactly, using only $\mathcal{O}(\log |Q|)$ space in total. Finally, the algorithm stores the Rabin–Karp fingerprint of the current text. When a new character $T[i]$ arrives, the algorithm considers the leftmost position ℓ_j in each level j. If $i - \ell_j + 1$ is smaller than 2^{j+1}, the algorithm does nothing. Otherwise if the fingerprints imply that ℓ_j is an occurrence of $Q[1, 2^{j+1}]$, the algorithm promotes it to the next level, and if ℓ_j is not an occurrence of $Q[1, 2^{j+1}]$, the algorithm discards it. When a position reaches the level $j = \lfloor \log |Q| \rfloor + 1$, which corresponds to the whole pattern Q, it is an occurrence of Q and the algorithm outputs it. For more details, see [40].

The complexity of [40] has been later improved by [12] to use $\mathcal{O}(\log |Q|)$ space and $\mathcal{O}(1)$ time per character of the text. Given a prefix of a streaming text T, we can use either of the algorithms to find all occurrences of the prefix in T.

3 Two-pass algorithm for shortest cover

In this section we give a two-pass streaming algorithm for computing the length of the shortest cover of a stream T of given length n. We start with the following simple observation.

▶ **Lemma 10.** *There is a streaming algorithm that checks if T ($|T| = n$) has a cover of length ℓ in $\mathcal{O}(\log \ell)$ space and $\mathcal{O}(1)$ time per character. The algorithm is randomized and has error probability inverse-polynomial in n.*

Proof. We apply the streaming pattern matching algorithm [12] to compute the subsequent elements of the set $A = \mathsf{Occ}_T(T[1, \ell])$. It suffices to check if

$$\mathsf{maxgap}(A \cup \{-\ell + 1, n + 1\}) \leq \ell. \qquad \blacktriangleleft$$

In the first pass of the algorithm, we identify $\mathcal{O}(\log n)$ candidates for the length of the shortest cover. In the second pass, we apply Lemma 10 to verify the candidates.

For a positive integer x, let $I(x) = [x, \frac{3}{2}x)$. Let us consider a sequence $x_1, x_2, \ldots, x_r$ of length $r = \mathcal{O}(\log n)$ such that $\{1, \ldots, n\} \subseteq \bigcup_{i=1}^{r} I(x_i)$. From now on let us focus on a single $x = x_i$. In the first pass, our goal is to find a candidate for the shortest cover of length in the interval $I(x)$. To this end, we run streaming pattern matching for $X = T[1, x]$ in T. We choose the candidate using the following algorithm:

Algorithm 1: Find-Candidate($\boldsymbol{x}$).

1. Find the maximal chain of occurrences of $X = T[1, x]$ in T that starts at position 1. Let d be its length.
2. Let $i \in \mathsf{Occ}_T(X)$ be the last position that starts a maximal chain of occurrences of length d. Choose $n - i + 1$ as the candidate.

If the candidate is outside the interval $I(x)$, we discard it.

▶ **Lemma 11.** *If the shortest cover C of T ($|T| = n$) has length $|C| \in I(x)$, then $|C| = n - i + 1$ where i is the last position that starts a maximal chain of occurrences of X of length d and d is the length of the maximal chain of occurrences of X that contains position 1.*

Proof. Let $a_1, \ldots, a_d$ be the maximal chain of occurrences of X in T that starts with $a_1 = 1$. Corollary 3 asserts that every chain is an arithmetic progression with difference $\mathsf{per}(X)$. Hence, by Observation 6, $|C| \geq a_d + x - 1$ ($|C| > a_d + x - 1$ if $d > 1$), so C contains the whole chain. Clearly, any occurrence of C at position j in T implies a chain of length at least d starting at j. Moreover, it is easy to show that in this case the maximal chain starting at position j has length exactly d. Indeed, assume to the contrary that it has length at least $d + 1$ and j' is its $(d + 1)$-th element. If $j' + x - j \leq |C|$, then this would imply that the chain starting at position 1 also has length at least $d + 1$. Otherwise, C would be periodic with period $\mathsf{per}(X)$, contradicting Observation 6.

Let $i' < i$ be two positions where a maximal chain of length d starts. We will show that i' cannot be the last occurrence of C. Then $i - i' > \frac{x}{2}$, so $n - i' + 1 > n - i + 1 + \frac{x}{2} \geq \frac{3}{2}x$. Hence, $n - i' + 1 \notin I(x)$. This shows that only the last position where a maximal chain of length d starts may be the last occurrence of the shortest cover of length in $I(x)$. $\blacktriangleleft$

▶ **Theorem 12.** *The length of the shortest cover of a string of length n can be computed by a two-pass streaming algorithm which uses $\mathcal{O}(\log^2 n)$ space and $\mathcal{O}(n \log n)$ time in each pass. The algorithm is randomized and has error probability inverse-polynomial in n.*

Proof. In the first pass we run Find-Candidate(x_i) for each $i = 1, \ldots, r$. The algorithm requires only $\mathcal{O}(1)$ time and space in addition to the streaming pattern matching algorithm [12] for pattern $X_i = T[1, x_i]$ in text T. Indeed, at each moment it suffices to store the rightmost inclusion-maximal chain of occurrences of X_i, represented in $\mathcal{O}(1)$ space as an arithmetic sequence. We compute the length d of the first chain. Afterwards, when a chain ends, we can compute the d-th element from its end (if any) and store it until the next such element is encountered.

This produces at most $r = \mathcal{O}(\log n)$ candidates for the length of the shortest cover. In the second pass, we use Lemma 10 to verify them. The complexities of the algorithm follow. The error probability follows from the union bound. ◀

4 One-pass algorithm for shortest cover

In this section we give a one-pass streaming algorithm for computing the length of the shortest cover of a stream T of given length n. It consists of two processes that we run in parallel. The first one finds the length of the shortest cover if it is at most $\sqrt{n \log n}$ and the second if it is greater than $\sqrt{n \log n}$.

4.1 Covers of small length

Our first algorithm computes the length of the shortest cover if it is at most $\sqrt{n \log n}$. It is based on the online algorithm of Breslauer [11] which we briefly recall below.

▶ **Definition 13** (Super-primitivity). *A string is called* super-primitive *if it is equal to its shortest cover.*

The algorithm makes use of three arrays B, C, R, where $B[i]$ is the length of the longest proper border of the prefix $T[1, i]$, $C[i]$ is the length of the shortest cover of $T[1, i]$, and $R[i]$ is not defined if $T[1, i]$ is not super-primitive and otherwise stores the length of the longest prefix of T, up to the latest arrived character $T[k]$, such that $T[1, i]$ is its cover. In the end, $C[n]$ contains the length of the shortest cover of the text.

The algorithm applies the Knuth–Morris–Pratt algorithm [29] that computes B in $\mathcal{O}(n)$ space and time.

Algorithm 2: Compute-Shortest-Cover.

1. **for** $k = 1$ **to** n **do**
   - $B[k]$ = the length of the longest proper border of $T[1, k]$
   - **if** $B[k] > 0$ and $R[C[B[k]]] \geq k - C[B[k]]$ **then**
     #If the shortest cover of $B[k]$ covers $T[1, k]$,
     #then it is the shortest cover of $T[1, k]$.
     - $C[k] = C[B[k]]$; $R[C[k]] = k$
   - **else** #If $B[k] = 0$ or $C[B[k]]$ does not cover $T[1, k]$, then $T[1, k]$ is
     #super-primitive.
     - $C[k] = k$; $R[k] = k$

We now explain our streaming algorithm that computes the length of the shortest cover if it is at most $\sqrt{n \log n}$. Similar to the algorithm of Breslauer, it uses three arrays B', C', and R'. $B'[i]$ is defined to be the largest $0 \leq i' \leq \sqrt{n \log n}$, $i' < i$, such that $T[1, i]$ has a border of length i', $C'[i]$ is the length of the shortest cover of $T[1, i]$ if it is at most $\sqrt{n \log n}$ (and otherwise undefined), and $R'[i]$, for $1 \leq i \leq \sqrt{n \log n}$ is not defined if $T[1, i]$ is not super-primitive and otherwise stores the length of the longest prefix of T, up to the latest arrived character $T[k]$, such that $T[1, i]$ is its cover.

Our algorithm proceeds exactly as the algorithm of Breslauer except that it uses C', R', B' instead of C, R, B. To compute B', we use the following straightforward corollary of [29].

▶ **Corollary 14** (of Knuth, Morris, Pratt [29]). *There is a (deterministic) streaming algorithm that computes B' using $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n)$ time.*

Proof. We take as a basis the Knuth–Morris–Pratt algorithm. We then note that to compute $B'[k]$ it suffices to know only $B'[k-1]$ and the first $\sqrt{n \log n}$ entries of the array B'. The complexities follow. ◀

Let $T[k]$ be the last arrived character of T. The algorithm first computes $B'[k]$. If $B'[k] > 0$, it checks if the shortest cover of $T[1, B'[k]]$ covers $T[1, k]$ (using the condition $R'[C'[B'[k]]] \geq k - C'[B'[k]]$) and, if so, sets $C'[k] = C'[B'[k]]$ and $R'[C'[k]] = k$. Otherwise, if $k \leq \sqrt{n \log n}$, it sets $C'[k] = R'[k] = k$, else it leaves $C'[k]$ undefined. The final answer is $C'[n]$.

The algorithm uses $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n)$ time in total. We now argue that the algorithm is correct. From Observation 6 it follows that any cover of $T[1, k]$ must be its border. That is, the only possible candidates for the shortest cover of $T[1, k]$ that have length $\leq \sqrt{n \log n}$ are the borders of $T[1, k]$ of length $\leq \sqrt{n \log n}$. Correctness of our algorithm for the case $B'[k] = 0$ follows. For the case $B'[k] > 0$ we use the following claim:

▶ **Fact 15** (Breslauer [11]). *If a string W is a cover of a string Z, and another string V, such that $|W| \leq |V|$, is a border of Z, then W is a cover of V. Hence, if a string Z has two covers W and V, such that $|W| \leq |V|$, then W is a cover of V.*

It follows that if the length of the shortest cover of $T[1, k]$ is at most $\sqrt{n \log n}$, then it is the length of the shortest cover of $T[1, B'[k]]$, which concludes the proof.

4.2 Covers of large length

We now give an algorithm that computes the length of the shortest cover of T if it is larger than $\sqrt{n \log n}$. Let $x_1, \ldots, x_r$ and $I(x)$ be defined as in Section 3. For all $x_{i+1} > \sqrt{n \log n}$, we seek for the shortest cover of length in $I(x_i)$ (if any). From now on we focus on a single $x = x_i$. Let $X = T[1, x]$.

Let a be the last element of the maximal chain of occurrences of X in T that starts at position 1 and $x' = a + x - 1$, $X' = T[1, x']$. We obtain the following lemma as a corollary of Lemma 11.

▶ **Lemma 16.** *If the shortest cover C of T has length $|C| \in I(x)$, then* $\max \mathsf{Occ}_T(C) = \max \mathsf{Occ}_T(X')$.

Proof. It suffices to note that the last position that starts a maximal chain of occurrences of X of length d is exactly the last occurrence of X'. Indeed, any occurrence of X' starts a chain of occurrences of X of length at least d (and conversely). If the length of the chain was greater than d, then X' would also occur $\mathsf{per}(X)$ positions later. ◀

In the beginning, we compute the maximal chain of occurrences of X in T that starts at position 1 and compute x' and $\varphi(X')$. We then continue with streaming pattern matching to find all occurrences of X' in T. We note that we can do the preprocessing for the streaming pattern matching algorithm simultaneously with computing X'.

One small technical difficulty here is that x' is known only once we know that the initial chain of occurrences of X does not extend, which can take place at position $x' + \frac{x}{2}$. To overcome this, whenever a new occurrence of X in the chain is found, say at position j, we assume that $x' = j + x - 1$ and start the pattern matching algorithm for $X' = T[1, x']$. If the next occurrence in the same chain is found, x' is overwritten. Note that the pattern matching algorithm for X' always looks for occurrences of prefixes of X' of lengths being powers of two, so it can be easily updated when the chain is extended.

We use the following key observation.

▶ **Observation 17.** *For every maximal chain of occurrences of X' in T, only the last position in it can be an occurrence of the shortest cover C.*

X' can be periodic and there can be many occurrences of X' in T, but by Observation 17, only the last position in every maximal chain can be an occurrence of C. Occurrences that satisfy this condition are henceforth called *relevant*. We maintain a stack of relevant occurrences (the topmost one can be non-relevant). When a new occurrence p of X' arrives, we check if it is in the same maximal chain at the occurrence in the top of the stack. If it is, we first pop from the stack, and then push p to the stack (in other words, we replace the occurrence in the top of the stack with p). Otherwise, we simply push p to the stack.

Suppose that in the end, the stack contains occurrences $p_1, p_2, \ldots, p_k$. Let $l_j = p_{j+1} - p_j$. Note that $l_j > \frac{x}{2}$ for every $j = 1, \ldots, k-1$, as the relevant occurrences are in different maximal chains. Therefore, there are $\mathcal{O}(n/x) = \mathcal{O}(\sqrt{n/\log n})$ occurrences in total. For each p_j, we memorize the fingerprint $\varphi(T[1, p_j - 1])$. By Lemma 16, there is only one possible candidate for the shortest cover in $I(x)$, the suffix C of T that starts at p_k. Then $\varphi(C)$ can be computed from $\varphi(T)$ and $\varphi(T[1, p_k - 1])$ via Fact 8.

If we have a way to test whether p_j is a starting position of an occurrence of C, then we can check if C is a cover of T using the maxgap. We now explain how we test p_j. If any of the differences l_j satisfies $l_j > |C|$, we immediately return NO. If $l_j = |C|$, we can check whether p_j is a starting position of an occurrence of C by computing $\varphi(T[p_j, p_{j+1} - 1])$ and comparing it with $\varphi(C)$. From now on, we focus on j such that $l_j < |C|$. From $|C| < \frac{3}{2}x$, it follows that $|C| - l_j \le x'$ and therefore $T[p_{j+1}, p_{j+1} + |C| - l_j - 1] = T[1, |C| - l_j]$. Consequently, if we know $\varphi(T[1, |C| - l_j])$, we can check whether p_j is a starting position of an occurrence of C in $\mathcal{O}(1)$ time in three steps: first, compute $\varphi(T[p_j, p_{j+1} - 1]) = \varphi(T[p_j, p_j + l_j - 1])$, second, compute $\varphi(T[p_j, p_j + |C| - 1])$ from $\varphi(T[p_j, p_j + l_j - 1])$ and $\varphi(T[1, |C| - l_j])$ via Fact 8, and finally, compare $\varphi(T[p_j, p_j + |C| - 1])$ and $\varphi(C)$. If the fingerprints are equal, p_j is an occurrence of C with high probability.

We compute the fingerprints $\varphi(T[1, |C| - l_j])$ as follows. Assume that we know the fingerprints $\varphi(T[1, n - l_j])$ and recall that C starts at p_k. By Fact 8, we can compute $\varphi(T[1, |C| - l_j])$ for all $j = 1, 2, \ldots, k-1$ from $\varphi(T[1, p_k - 1])$. Namely, we compute $\varphi(T[p_k, n - l_j]) = \varphi(T[1, n - l_j - p_k + 1])$, and

$$n - l_j - p_k + 1 = n - p_k + 1 - l_j = |C| - l_j.$$

We now explain how we compute the fingerprints $\varphi(T[1, n - l_j])$. First, consider all j such that $n - l_j \ge p_{j+1} + x' - 1$. Note that since $l_j < |C| < \frac{3}{2}x$ and the distance between any two consecutive positions is greater than $\frac{x}{2}$, this inequality holds for all $j \le k - 4$:

$$n - l_j > n - \tfrac{3}{2}x \ge p_k + x' - 1 - \tfrac{3}{2}x > p_{j+1} + (k - j - 1)\tfrac{x}{2} + x' - 1 - \tfrac{3}{2}x \ge p_{j+1} + x' - 1.$$

We maintain a balanced BST of all l_j. Say that the streaming pattern matching algorithm reports a new occurrence p of X' (which happens when $T[p + x' - 1]$ arrives). If the previous occurrence p_j was more than $\frac{x}{2}$ positions earlier, we assume for now that $p_{j+1} = p$, compute $l_j = p_{j+1} - p_j$ and insert it to the BST. If the previous position p_j was at most $\frac{x}{2}$ positions earlier, it is popped from the stack, we assume that $p_j = p$ and l_{j-1} is recomputed. Furthermore, when a new position i of the text arrives, we check whether the BST contains $l_j = n - i$ and, if so, memorize the fingerprint of $T[1, i] = T[1, n - l_j]$. The algorithm will indeed compute $\varphi(T[1, n - l_j])$ for all j such that $n - l_j \geq p_{j+1} + x' - 1$.

It remains to compute $\varphi(T[1, n - l_j])$ for all $j \leq k - 1$ such that $n - l_j < p_{j+1} + x' - 1$. As explained above, this inequality can hold only for $j \in \{k - 3, k - 2, k - 1\}$. We note that, additionally, we have $p_{j+1} \leq n - l_j$. Indeed, from $l_j < |C|$, we have

$$n - l_j \geq n - |C| + 1 = p_k \geq p_{j+1}.$$

For all such j we use a different subroutine. The subroutine is initialised at the position $p_j + x' - 1$ and must output $\varphi(T[1, n - l_j])$ at the position $p_{j+1} + x' - 1$ if $p_{j+1} \leq n - l_j < p_{j+1} + x' - 1$. In Section 5 we will show such a subroutine that takes $\mathcal{O}(\log n)$ space and $\mathcal{O}(\log n)$ time per character of the text.

We use the subroutine in the following way. When we find a new occurrence p of X' such that $n < p + x' - 1 + 2 \cdot \frac{3}{2}x$, we initialize a new instance of the subroutine. If later we find out that p is not a relevant occurrence, we kill the process. Note that at any time, there will be a constant number of instances of the subroutine running (because the number of survived processes is equal to the number of occurrences p_j satisfying $n < p_j + x' - 1 + 3x$, and every two occurrences p_j are at least $\frac{x}{2}$ positions apart). Therefore, in total this step takes $\mathcal{O}(\log n)$ space and $\mathcal{O}(\log n)$ time per character.

In conclusion, we can compute $\varphi(T[1, n - l_j])$ for all $j \leq k - 1$ using $\mathcal{O}(\sqrt{n/\log n})$ space and $\mathcal{O}(\log n)$ time per character of the text.

The whole algorithm for a given x can be stated as below.

1. Perform streaming pattern matching for X in T. Compute x' and X'.
   #x' can be updated several times due to "guessing"
2. Start streaming pattern matching for X' in T:
   - Using a stack, compute subsequent relevant occurrences of X'. When occurrence p_{j+1} is found, store $\varphi(T[1, p_{j+1} - 1])$ and insert $l_j = p_{j+1} - p_j$ to a BST.
     #l_j can be updated several times due to "guessing" of p_{j+1}
   - When $T[i]$ is read and the BST contains $l_j = n - i$, memorize the fingerprint of $T[1, i] = T[1, n - l_j]$.
   - For every occurrence p_j that satisfies $n < p_j + x' - 1 + 3x$, start the subroutine of Section 5 to compute $T[1, n - l_j]$.
     #If the occurrence turns out not to be relevant, kill the subroutine.
3. When the end of the text is reached:
   - If $l_j > |C|$ for any $j = 1, \ldots, k - 1$, where $C = T[p_k, n]$, return NO.
   - Compute $\varphi(C)$ from $\varphi(T[1, p_k - 1])$ and $\varphi(T)$.
   - Compute $\varphi(T[p_j, p_j + |C| - 1])$ for all $j = 1, \ldots, k - 1$ using $\varphi(T[p_j, p_{j+1} - 1])$ and $\varphi(T[1, n - l_j])$ (if $l_j < |C|$).
   - Let $P = \{p_j : \varphi(T[p_j, p_j + |C| - 1]) = \varphi(C)\}$. If $\mathsf{maxgap}(P \cup \{-|C| + 1, n + 1\}) = |C|$, return $|C|$. Otherwise, return NO.

The algorithm uses $\mathcal{O}(k + \log n) = \mathcal{O}(\sqrt{n/\log n})$ space and spends $\mathcal{O}(\log n)$ time per text character, apart from the last position of the text where it spends $\mathcal{O}(\sqrt{n/\log n})$ time. Over

all $x = x_i$ for $i = 1, \ldots, r$ such that $\frac{3}{2}x_i > \sqrt{n \log n}$, the algorithm uses $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n \log^2 n)$ time in total. Combining this algorithm with the algorithm of Section 4.1, we arrive at the following result.

▶ **Theorem 18.** *The length of the shortest cover of a string of length n can be computed by a one-pass streaming algorithm which uses $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n \log^2 n)$ time in total. The algorithm is randomized and has error probability inverse-polynomial in n.*

5 Computing the fingerprint of $T[1, n - l_j]$ for $j \in \{k-3, k-2, k-1\}$

We will show how to solve a more general problem.

▶ **Problem 1.** *Let T be a streaming text of a given length n and $y \leq n$ be a positive integer. For some position* start*, $y \leq \mathsf{start} \leq n$, when we have read $T[\mathsf{start}]$, we are given an integer z. Let $\mathsf{len}(q) = z - 2q$. For each q such that $T[q, q+y-1] = T[1, y]$, at the moment when we have read $T[q + y - 1]$, we are to report $\psi(q) = \varphi(T[q, q + \mathsf{len}(q)])$ provided that $\mathsf{start} \leq q + \mathsf{len}(q)$ and $0 \leq \mathsf{len}(q) < y$.*

If we take $\mathsf{start} = p_j + x' - 1$, $y = x'$, $z = n + p_j$, then for $q = p_{j+1}$ we have $q + \mathsf{len}(q) = z - q = n + p_j - p_{j+1} = n - l_j$. We also have $p_j + x' - 1 = \mathsf{start} \leq q + \mathsf{len}(q) = n - l_j$ since $p_j + x' - 1 + l_j = p_{j+1} + x' - 1 \leq n$. Finally, we require that $p_{j+1} \leq n - l_j < p_{j+1} + x' - 1$ which translates to $0 \leq \mathsf{len}(q) < y - 1$. Therefore, using an algorithm for Problem 1 we can compute $\varphi(T[p_{j+1}, n - l_j])$, and therefore, by Fact 8, $\varphi(T[1, n - l_j])$ from $\varphi(T[1, p_{j+1} - 1])$ that is stored by the exact pattern matching algorithm for X' and T.

We now show an algorithm for Problem 1 that takes $\mathcal{O}(\log n)$ space and $\mathcal{O}(\log n)$ time per character of the text. Denote $Y = T[1, y]$ and let

$$\mathsf{Occ}_T(j) = \{q \in \mathsf{Occ}_T(Y[1, 2^j]) \,|\, \mathsf{start} \leq q + \mathsf{len}(q) \text{ and } 2^j \leq \mathsf{len}(q) + 1 < 2^{j+1}\}.$$

For $j = \lceil \log y \rceil$, we additionally assume that $\mathsf{len}(q) < y$. Note that the second condition in the above definition can be written equivalently as

$$\tfrac{z+1}{2} - 2^j < q \leq \tfrac{z+1}{2} - 2^{j-1}.$$

which concludes, in particular, that $\mathsf{Occ}_T(j)$ either contains at most one element or is a single chain of occurrences of $Y[1, 2^j]$.

▶ **Observation 19.** *The set of all $q \in \mathsf{Occ}_T(Y)$, $\mathsf{start} \leq q + \mathsf{len}(q)$, such that $0 \leq \mathsf{len}(q) < y$, is a subset of $\bigcup_{0 \leq j \leq \lfloor \log y \rfloor} \mathsf{Occ}_T(j)$.*

We will show how to compute and store a small amount of additional information for each set $\mathsf{Occ}_T(j)$ that we will use to retrieve the fingerprints $\psi(q)$ for $q \in \mathsf{Occ}_T(j) \cap \mathsf{Occ}_T(Y)$.

We will build our solution upon the streaming pattern matching algorithm for Y in T. Recall that the algorithm stores, for every $j = 0, \ldots, \lfloor \log y \rfloor$, a subset of $\mathsf{Occ}_T(Y[1, 2^j])$ that corresponds to occurrences in the suffix of length 2^{j+1} of $T[1, i]$. Let us denote this subset by $S_j(i)$. The set $S_j(i)$ is stored as either at most two single occurrences or a chain with period $\Delta_j = \mathsf{per}(Y[1, 2^j])$, so that one can check in $\mathcal{O}(1)$ time if $q \in S_j(i)$. Moreover, for every $q \in S_j(i)$, we can recover $\varphi(T[1, q - 1])$; we also store $\varphi(T[1, i])$.

For every position $i \geq \mathsf{start}$, we can compute the position q that satisfies $q + \mathsf{len}(q) = i$; we have $q = z - i$. Let j be such that $2^j \leq \mathsf{len}(q) + 1 < 2^{j+1}$. If $q \notin S_j(i)$, then $q \notin \mathsf{Occ}_T(Y)$ and we can ignore it. Otherwise, at this point we can compute $\psi(q)$ using Fact 8. If we could

store $\psi(q)$ for every $q \in \mathsf{Occ}_T(j)$, then, when the streaming pattern matching algorithm outputs a position $q \in \mathsf{Occ}_T(Y)$, we could in $\mathcal{O}(\log n)$ time find j such that $q \in \mathsf{Occ}_T(j)$ and then output $\psi(q)$, as desired.

Unfortunately, if $|\mathsf{Occ}_T(j)|$ is large, we cannot afford to store $\psi(q)$ for every position $q \in \mathsf{Occ}_T(j)$. Moreover, at the time when q is processed, we actually do not know if q will be an occurrence of Y and only this information makes this value relevant. We will use periodicity to overcome these setbacks and design a small representation of values $\psi(q)$ of all potentially relevant elements $q \in \mathsf{Occ}_T(j)$; see Example 20 for an illustration.

For a given j, let us consider the first moment $i = i'$ (if any) when $q' = z - i' \in \mathsf{Occ}_T(j)$, for any $i \geq \mathsf{start}$. We then have $q' = \max \mathsf{Occ}_T(j)$. If $q \in \mathsf{Occ}_T(j)$, then $q + \mathsf{len}(q) \geq i'$ but $\mathsf{len}(q) < 2^{j+1}$, so $q \in S_j(i')$. In other words, $\mathsf{Occ}_T(j) \subseteq S_j(i')$. Let $\{q_1, \ldots, q_r\}$, with $q_1 < \ldots < q_r$, be the subsequence of consecutive elements of $S_j(i')$ such that q_1 is the smallest element of $S_j(i')$ and $q_r = q'$. There is an index $\ell \in \{1, \ldots, r\}$ such that $\mathsf{Occ}_T(j) = \{q_\ell, \ldots, q_r\}$. If $r - \ell + 1 \leq 2$, we will store $\psi(q_k)$ for every $k = \ell, \ldots, r$ explicitly. Otherwise, $q_\ell, \ldots, q_r$ is a chain of occurrences of $Y[1, 2^j]$ with step Δ_j.

In the beginning, the pattern matching algorithm computes the set $S_j(2^{j+1})$. Let d_j be the length of the longest chain of occurrences of $Y[1, 2^j]$ in $S_j(2^{j+1})$ that starts at position 1. We will store this value in the algorithm.

Let $i_k = z - q_k$. When i increases from i_r to i_ℓ, the algorithm computes $\psi(q_k)$ for subsequent $k = r, \ldots, \ell$. We will store $\psi(q_r)$ as well as $\psi(q_k)$ for the last element q_k that was considered. Let us now consider $i = i_k$ for $k < r$. Let $q_{r+1}, \ldots, q_{t_k}$ be the subsequent elements of the chain that contains q_r in $S_j(i_k)$. If $q_{t_k} + 2^j - 1 \geq i_{k+1}$, then $T[q_{k+1}, i_{k+1}]$ is periodic with period Δ_j. Hence, for every $k' > k$,

$$T[q_{k'}, i_{k'}] = T[1, \Delta_j]^{2(r-k')}\, T[q_r, i_r].$$

Thus $\psi(q_{k'})$ for any $k' > k$ can be computed from $\psi(q_r)$ (which is stored) and $\varphi(T[1, \Delta_j])$ (which can be computed from $\varphi(T[1, q_r - 1])$ and $\varphi(T[1, q_{r-1} - 1])$ using Fact 8) in $\mathcal{O}(\log n)$ time via Corollary 9.

As for the opposite case, let us consider the first $k < r$ for which $q_{t_k} + 2^j - 1 < i_{k+1}$. If it exists, let us denote this value of k by k_0. This means that the chain of occurrences of $Y[1, 2^j]$ that contains q_r definitely ends at position $q_{t_{k_0}}$. Then Y can occur at position q_p for $p \in \{\ell, \ldots, r\}$ only if $t_{k_0} - p + 1 = d_j$. If $p > k_0 + 1$, $\psi(q_p)$ can be restored as shown above. If $p = k_0 + 1$, $\psi(q_p)$ is still stored from the previous step. Otherwise, we will store $\psi(q_p)$ when the position i_p is processed.

If the position k_0 is not found, we also store $\psi(q_\ell)$ since $T[q_\ell, i_\ell]$ might not have period Δ_j.

▶ **Example 20.** Let us consider the setting from Fig. 1 with $j = 5$. Assume that z is such that $\mathsf{Occ}_T(j) = \{q_2, \ldots, q_6\}$ (i.e., $\ell = 2$ and $r = 6$) and $i_k = q_k + \mathsf{len}(q_k)$ for $k = 2, \ldots, 6$. Further assume that $\mathsf{start} \leq i_6$.

When $i = i_6$, only the occurrences $q_1, \ldots, q_6$ are known. At this moment we compute and store $\psi(q_6)$.

When $i = i_5$, only the occurrences $q_1, \ldots, q_7$ are known (i.e., $t_5 = 7$) and $T[q_6, i_6]$ has period Δ_j (since $q_7 + 2^j - 1 \geq i_6$), so $k_0 \neq 5$. At this moment we compute $\psi(q_5)$ and store it until the next step. We also compute and store $\varphi(T[1, \Delta_j])$.

When $i = i_4$, all the occurrences $q_1, \ldots, q_8$ are known (i.e., $t_4 = 8$) and $T[q_5, i_5]$ has period Δ_j (since $q_8 + 2^j - 1 \geq i_5$), so $k_0 \neq 4$. We compute $\psi(q_4)$ and store it until the next step.

When $i = i_3$, there is no new occurrence in the chain (i.e., $t_3 = 8$), so $q_8 + 2^j - 1 < i_4$ and $k_0 = 3$ (note that $T[q_4, i_4]$ still could have period Δ_j, if the character $\mathtt{x}$ of T was equal to $\mathtt{c}$; see Fig. 1). At this moment we know that, among $\psi(q_k)$ for $k = 2, \ldots, 6$, at most one

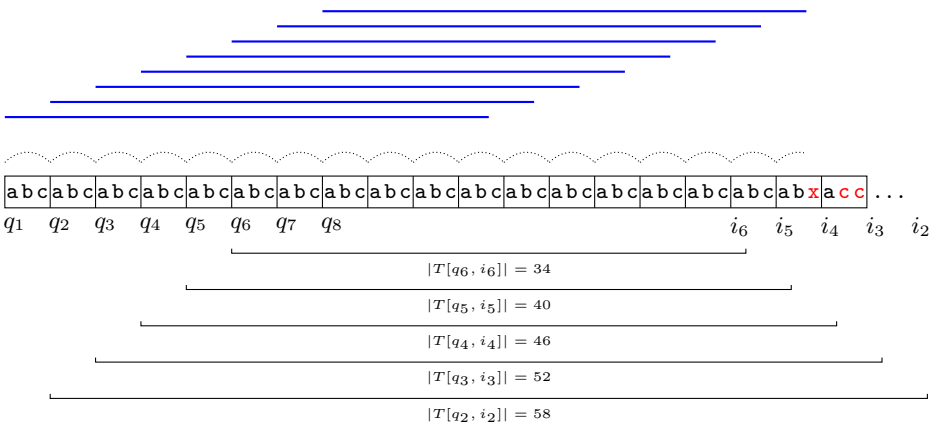

Figure 1 Let $j = 5$ and assume that $Y[1, 32] = (\mathtt{abc})^{10}\mathtt{ab}$ and $y \geq 64$. The positions $q_1, \ldots, q_8$ form a maximal chain of occurrences of $Y[1, 32]$ in T (the occurrences are depicted as blue lines) and the period $Y[1, \Delta_j]$ is shown by dotted arcs.

value can be relevant, for $8 - k + 1 = d_j$. If $k > 4$, this value can be computed from $\psi(q_6)$ and $\varphi(T[1, \Delta_j])$. Otherwise, it is either $\psi(q_4)$ that is still stored or one of the values $\psi(q_k)$ for $k \leq 3$ that will be computed in this or subsequent steps.

E.g., if Y starts with $(\mathtt{abc})^{14}\mathtt{abx}$, then $d_j = 5$ and we only need to store $\psi(q_4)$.

In conclusion, the algorithm for a given $j \in \{0, \ldots, \lfloor \log z \rfloor\}$ works as follows:

1. Compute d_j, the length of the longest chain of occurrences starting from position 1 in $S_j(2^{j+1})$.
2. Starting from position start, look for the first $i \geq \mathsf{start}$ such that $q = z - i \in \mathsf{Occ}_T(j)$. Let $q_1, \ldots, q_r$ be the maximal increasing subsequence of $S_j(i)$ that ends at $q = q_r$. Let q_ℓ be the minimal element in $\mathsf{Occ}_T(j)$ (it can be computed in $\mathcal{O}(1)$ time). Compute $\psi(q_r)$ using Fact 8 and store it. If $r = \ell$, no more data is stored.
3. For each subsequent position i of the text, if $i = i_k$ for some $k \in \{\ell, \ldots, r-1\}$, then:
   - Compute $\psi(q_k)$ using Fact 8 and store it until position i_{k-1}.
   - Assume that $k = r - 1$. If $k = \ell$, store this fingerprint. Otherwise, compute $\varphi(T[1, \Delta_j])$.
   - If $q_{t_k} + 2^j - 1 \geq i_{k+1}$, continue.
   - When the first $k < r$ for which $q_{t_k} + 2^j - 1 < i_{k+1}$ is encountered, $k_0 = k$, store only $\psi(q_p)$ for $p = tk_0 - d_j + 1$. If $p > k_0$, this value can be restored from the previous steps in $\mathcal{O}(\log n)$ time. Otherwise, continue the algorithm until i_p.
   - If there was no such k_0, store $\psi(q_\ell)$.

When $q \in \mathsf{Occ}_T(Y)$ is encountered, $\mathsf{start} \leq q + \mathsf{len}(q)$, and $2^j \leq \mathsf{len}(q) + 1 < 2^{j+1}$, compute k such that $q = q_k$ and:

- If $r - \ell \leq 1$ or k_0 exists, return the stored value of $\psi(q_k)$ (for $k \in \{q, r\}$) in $\mathcal{O}(1)$ time;
- If k_0 does not exist, return the stored value if $k = \ell$ or compute $\psi(q_k)$ from $\psi(q_r)$ and $\varphi(T[1, \Delta_j])$ in $\mathcal{O}(\log n)$ time if $k > \ell$.

In total, for a fixed value of j, we use $\mathcal{O}(1)$ space and spend $\mathcal{O}(\log n)$ time per text character in the worst case. The value q such that $q + \mathsf{len}(q) = i$ is always unique, so at every position of the text, computations for only one index j take place. Overall, we spend $\mathcal{O}(\log n)$ time per text character. This completes the description of the algorithm for Problem 1.

6 Hardness of computation of seeds in a stream

While above we showed an $\mathcal{O}(\sqrt{n \log n})$-space one-pass streaming algorithm for computing the shortest cover, there is no sublinear-space one-pass streaming algorithm for computing the shortest seed. The proof follows the lines of the proof of the space lower bound for computing the shortest period of a stream by Ergün et al. [20].

Consider the communication game between Alice and Bob who hold two strings T_1 and T_2 of length $n/2$ each, where the goal is to compute the shortest seed of $T = T_1T_2$. By a standard reduction, the lower bound on the communication complexity for this problem is a space lower bound for any streaming algorithm computing the shortest seed of a string T of length n. We can show the lower bound of $\Omega(n)$ bits for the communication complexity of the problem by a reduction from the augmented indexing problem: Suppose Alice is given a string $X \in \{0,1\}^{n/2}$, and Bob is given an index $i \in [1, n/2]$ and a string $Y \in \{0,1\}^{i-1}$ such that $Y = X[1, i-1]$. Bob must decide whether $X[i] = 1$. The randomized communication complexity of this problem is $\Omega(n)$ bits [9, 10]. On the other hand, for $i < n/2$ we can reduce it to the problem of computing the shortest seed by taking $T_1 = X$ and $T_2 = \$^{n/2-i}\ Y\ 1$, where \$ is a special character different from $0, 1$. It is easy to see that the shortest seed of $T = T_1T_2$ is equal to $n - i$ iff $X[i] = 1$. Therefore, the communication game of computing the shortest seed requires $\Omega(n)$ bits of communication, and any streaming algorithm for computing the shortest seed of a string of length n requires $\Omega(n)$ bits of space as well.

7 Conclusion and open questions

In this work, we give the first sublinear-space streaming algorithms for computing the length of the shortest cover of a stream. Our two-pass streaming algorithm uses $\mathcal{O}(\log^2 n)$ space and $\mathcal{O}(n \log n)$ time, and our one-pass streaming algorithm uses $\mathcal{O}(\sqrt{n \log n})$ space and $\mathcal{O}(n \log^3 n)$ time. It is an interesting question if similar algorithms can be developed for computing the shortest covers of all prefixes of the stream, and if the complexity of the algorithms can be improved. We also state an open question concerning computation of seeds: Design a streaming algorithm with any number of passes and $\mathcal{O}(n^{1-\varepsilon})$ space, for $\varepsilon > 0$, for computing the shortest seed of a stream.

References

1 Ali Alatabbi, Abu Sayed Md. Sohidull Islam, Mohammad Sohel Rahman, Jamie Simpson, and William F. Smyth. Enhanced Covers of Regular and Indeterminate Strings Using Prefix Tables. *Journal of Automata, Languages and Combinatorics*, 21(3):131–147, 2016. doi:10.25596/jalc-2016-131.

2 Ali Alatabbi, M. Sohel Rahman, and William F. Smyth. Computing covers using prefix tables. *Discrete Applied Mathematics*, 212:2–9, 2016. doi:10.1016/j.dam.2015.05.019.

3 Amihood Amir, Costas S. Iliopoulos, and Jakub Radoszewski. Two strings at Hamming distance 1 cannot be both quasiperiodic. *Information Processing Letters*, 128:54–57, 2017. doi:10.1016/j.ipl.2017.08.005.

4 Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat. Can We Recover the Cover? In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, pages 25:1–25:15, 2017. doi:10.4230/LIPIcs.CPM.2017.25.

5 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate Cover of Strings. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78, pages 26:1–26:14, 2017. doi:10.4230/LIPIcs.CPM.2017.26.

6 Amihood Amir, Avivit Levy, and Ely Porat. Quasi-Periodicity Under Mismatch Errors. In *Proceedings of the Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, pages 4:1–4:15, 2018. doi:10.4230/LIPIcs.CPM.2018.4.

7 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993. doi:10.1016/0304-3975(93)90159-Q.

8 Alberto Apostolico, Martin Farach, and Costas S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi:10.1016/0020-0190(91)90056-N.

9 Ziv Bar-Yossef, T. S. Jayram, Robert Krauthgamer, and Ravi Kumar. The Sketching Complexity of Pattern Matching. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and of the International Workshop on Randomization and Computation, APPROX-RANDOM 2004*, pages 261–272, 2004. doi:10.1007/978-3-540-27821-4_24.

10 Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. Information theory methods in communication complexity. In *Proceedings of the IEEE Annual Conference on Computational Complexity, CCC 2002*, pages 93–102, 2002. doi:10.1109/CCC.2002.1004344.

11 Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.

12 Dany Breslauer and Zvi Galil. Real-Time Streaming String-Matching. *ACM Transasctions on Algorithms*, 10(4):22:1–22:12, 2014. doi:10.1145/2635814.

13 Manolis Christodoulakis, Costas S. Iliopoulos, Kunsoo Park, and Jeong Seop Sim. Approximate seeds of strings. *Journal of Automata, Languages and Combinatorics*, 10:609–626, 2005. doi:10.25596/jalc-2005-609.

14 Michalis Christou, Maxime Crochemore, Ondrej Guth, Costas S. Iliopoulos, and Solon P. Pissis. On left and right seeds of a string. *Journal of Discrete Algorithms*, 17:31–44, 2012. doi:10.1016/j.jda.2012.10.004.

15 Michalis Christou, Maxime Crochemore, and Costas S. Iliopoulos. Quasiperiodicities in Fibonacci strings. *Ars Combinatoria*, 129:211–225, 2016.

16 Michalis Christou, Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, and Tomasz Waleń. Efficient seed computation revisited. *Theoretical Computer Science*, 483:171–181, 2013. doi:10.1016/j.tcs.2011.12.078.

17 Richard Cole, Costas S. Iliopoulos, Manal Mohamed, William F. Smyth, and Lu Yang. The Complexity of the Minimum k-cover Problem. *Journal of Automata, Languages and Combinatorics*, 10(5–6):641–653, 2005. doi:10.25596/jalc-2005-641.

18 Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Streaming Periodicity with Mismatches. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and of the International Workshop on Randomization and Computation, APPROX-RANDOM 2017*, pages 42:1–42:21, 2017. doi:10.4230/LIPIcs.APPROX-RANDOM.2017.42.

19 Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou. Periodicity in Data Streams with Wildcards. In *Proceedings of the International Computer Science Symposium in Russia, CSR 2018*, pages 90–105, 2018. doi:10.1007/978-3-319-90530-3_9.

20 Funda Ergün, Hossein Jowhari, and Mert Sağlam. Periodicity in Streams. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and of the International Workshop on Randomization and Computation, APPROX-RANDOM 2010*, pages 545–559, 2010. doi:10.1007/978-3-642-15369-3_41.

21 Nathan J. Fine and Herbert S. Wilf. Uniqueness Theorems for Periodic Functions. *Proceedings of the American Mathematical Society*, 16:109–114, 1965.

22 Tomáš Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, William F. Smyth, and Wojciech Tyczyński. Enhanced string covering. *Theoretical Computer Science*, 506:102–114, 2013. doi:10.1016/j.tcs.2013.08.013.

23 Qing Guo, Hui Zhang, and Costas S. Iliopoulos. Computing the λ-Seeds of a String. In *Proceedings of Algorithmic Aspects in Information and Management, AAIM 2006*, pages 303–313, 2006. doi:10.1007/11775096_28.

24 Qing Guo, Hui Zhang, and Costas S. Iliopoulos. Computing the λ-covers of a string. *Information Sciences*, 177(19):3957–3967, 2007. doi:10.1016/j.ins.2007.02.020.

25 Ondřej Guth. On approximate enhanced covers under Hamming distance. *Discrete Applied Mathematics*, 2019. doi:10.1016/j.dam.2019.01.015.

26 Costas S. Iliopoulos, Manal Mohamed, and William F. Smyth. New complexity results for the k-covers problem. *Information Sciences*, 181(12):2571–2575, 2011. doi:10.1016/j.ins.2011.02.009.

27 Costas S. Iliopoulos, Dennis W. G. Moore, and Kunsoo Park. Covering a string. *Algorithmica*, 16(3):288–297, September 1996. doi:10.1007/BF01955677.

28 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

29 Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:322–350, 1977. doi:10.1137/0206024.

30 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A Linear Time Algorithm for Seeds Computation. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012*, pages 1095–1112, 2012. URL: http://dl.acm.org/citation.cfm?id=2095116.2095202.

31 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A Linear Time Algorithm for Seeds Computation. *CoRR*, abs/1107.2422v2, 2019. arXiv:1107.2422v2.

32 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Fast Algorithm for Partial Covers in Words. *Algorithmica*, 73(1):217–233, September 2015. doi:10.1007/s00453-014-9915-3.

33 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient algorithms for shortest partial seeds in words. *Theoretical Computer Science*, 710:139–147, 2018. Advances in Algorithms & Combinatorics on Strings (Honoring 60th birthday for Prof. Costas S. Iliopoulos). doi:10.1016/j.tcs.2016.11.035.

34 Yin Li and William F. Smyth. Computing the Cover Array in Linear Time. *Algorithmica*, 32(1):95–106, January 2002. doi:10.1007/s00453-001-0062-2.

35 Neerja Mhaskar and William F. Smyth. Frequency Covers for Strings. *Fundamenta Informaticae*, 163(3):275–289, 2018. doi:10.3233/FI-2018-1744.

36 Neerja Mhaskar and William F. Smyth. String covering with optimal covers. *Journal of Discrete Algorithms*, 51:26–38, 2018. doi:10.1016/j.jda.2018.09.003.

37 Dennis Moore and William F. Smyth. A Correction to "An Optimal Algorithm to Compute All the Covers of a String". *Information Processing Letters*, 54(2):101–103, April 1995. doi:10.1016/0020-0190(94)00235-Q.

38 Dennis W. G. Moore and William F. Smyth. An Optimal Algorithm to Compute all the Covers of a String. *Information Processing Letters*, 50:239–246, 1994. doi:10.1016/0020-0190(94)00045-X.

39 Alexandru Popa and Andrei Tanasescu. Hardness and algorithmic results for the approximate cover problem. *CoRR*, abs/1806.08135, 2018. arXiv:1806.08135.

40 Benny Porat and Ely Porat. Exact And Approximate Pattern Matching In The Streaming Model. In *Proceedings of the Annual Symposium on Foundations of Computer Science, FOCS 2009*, pages 315–323, 2009. doi:10.1109/FOCS.2009.11.

41 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Algorithms for Computing the λ-regularities in Strings. *Fundamenta Informaticae*, 84(1):33–49, 2008. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi84-1-04.