Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Experimental evaluation of algorithms for computing quasiperiods

Patryk Czajka [1], Jakub Radoszewski *,[1]

Institute of Informatics, University of Warsaw, Warsaw, Poland

ARTICLE INFO

Article history:
Received 20 September 2019
Received in revised form 11 March 2020
Accepted 8 November 2020
Available online xxxx

Keywords:
quasiperiodicity
Cover
Seed

ABSTRACT

Quasiperiodicity is a generalization of periodicity that was introduced in the early 1990s. Since then, dozens of algorithms for computing various types of quasiperiodicity were proposed. Our work is a step towards answering the general question: "Which algorithm for computing quasiperiods to choose?". The central notions of quasiperiodicity are covers and seeds. We implement algorithms for computing covers and seeds in the original and in new simplified versions and compare their efficiency on various types of data. We also discuss other known types of quasiperiodicity, distinguish partial covers as currently the most promising for large real-world data, and check their effectiveness using real-world data.

1. Introduction

Quasiperiodicity was introduced by Apostolico and Ehrenfeucht [8] as an extension of standard periodicity. The most basic type of a quasiperiod is a *cover*. We say that a string C is a cover of a string S if every position of S lies within an occurrence of C. A generalization of the notion of cover is the notion of *seed*. A seed is a cover of a superstring of S. In other words, we allow the seed to cover positions of S with overhanging occurrences. For example, the shortest cover of string *aabaaabaabaa* is *aabaa* and the shortest seeds of this string are *aaba* and *abaa*.

An $\mathcal{O}(n)$-time algorithm that computes the shortest cover of a string of length n was given by Apostolico et al. [9]. An $\mathcal{O}(n)$-time on-line algorithm for the same problem was proposed by Breslauer [13]. Moore and Smyth [54,55] proposed a linear-time algorithm computing all the covers of a string; its simpler implementation can be inferred from a recent work of Crochemore et al. [25]. Finally, Li and Smyth [53] developed a linear-time algorithm for computing the length of the longest cover of every prefix of a string. The output of their algorithm can be used to compute all the covers of any prefix of the string. All the aforementioned algorithms use only basic arrays related to string periodicity (the border array, the Pref array) and, therefore, their complexities do not depend on the alphabet size.

Seeds were introduced and first studied by Iliopoulos et al. [42] who proposed an $\mathcal{O}(n \log n)$-time algorithm computing a representation of all the seeds in a string of length n. A different algorithm with the same time complexity computing the shortest seed of a string was proposed by Christou et al. [20]; they also showed how to fill in a gap in the algorithm

* Corresponding author.
E-mail addresses: patryk.r.czajka@gmail.com (P. Czajka), jrad@mimuw.edu.pl (J. Radoszewski).

[1] Supported by the "Algorithms for text processing with errors and uncertainties" project no. POIR.04.04.00-00-24BA/16 carried out within the HOMING program of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

https://doi.org/10.1016/j.tcs.2020.11.033
0304-3975/

of Iliopoulos et al. [42]. Both algorithms work for strings over arbitrary ordered alphabets; the algorithm of Iliopoulos et al. [42] uses Crochemore's partitioning [22] whereas the algorithm of Christou et al. [20] uses suffix trees. Finally, an $\mathcal{O}(n)$-time algorithm for computing all the seeds of a string of length n over an integer alphabet (i.e., alphabet $\{1, \ldots, n^{\mathcal{O}(1)}\}$) was presented by Kociumaka et al. [50]; it was simplified in [49].

Due to applications in molecular biology, both covers and seeds have been extended into the case of covering a string with multiple quasiperiods [46]. This way, the notions of k-covers [21,41], λ-covers [34], and λ-seeds [32] were introduced. In applications such as molecular biology and computer-assisted music analysis, finding exact repetitions is not always sufficient; the same problem occurs for quasiperiodic repetitions. This led to the introduction of various notions of approximate covers and seeds [3–5,17,36,38,58], enhanced covers [29] and approximate enhanced covers [36], partial covers and seeds [52,51], and approximate λ-covers [33].

Other results in this area include computation of covers and seeds in the parallel [12,14,45] and streaming models [31]. Another line of research is finding maximal substrings of a string having a proper cover; see [8,16,44]. (In these contributions, the definition of maximality is rather subtle.) Intermediate variants of quasiperiodicity between covers and seeds called left and right seeds were also considered [18,20,29], as well as recovering a string from the set of covers of its prefixes [27,30]. Combinatorial properties of covers were presented in [2,19,43]. Covers were also considered in indeterminate [1,6,25,40] and weighted strings [11,39] and in 2-dimensional texts [26,56]. A survey by Apostolico and Breslauer [7] describes in detail the algorithms [8,13,14,45], whereas a survey by Iliopoulos and Mouchard [43] describes the algorithms [8,9,42]. Early results on quasiperiodicity were also surveyed by Smyth [59].

Our contributions

- We provide implementations of known efficient algorithms for computing quasiperiods (covers, seeds) and approximate quasiperiods (partial covers) and compare their efficiency using extensive computer experiments.
- We show how to use a data structure called Joinable Segment Trees to obtain an alternative $\mathcal{O}(n \log n)$-time version of the linear-time algorithm for computing seeds that turns out to perform superior on both artificial data and data from a known corpus. The same data structure is used for efficient computation of partial covers.
- We perform a detailed review of the literature on quasiperiodicity and approximate quasiperiodicity, identify approaches which are currently most promising for computations on large data, and additionally point out a flaw in one of the known algorithms (computation of λ-covers).

We implemented the algorithms of Apostolico et al. [9], Breslauer [13], Moore and Smyth [54,55] (its simplified version) and a folklore $\mathcal{O}(n \log n)$-time algorithm for computing covers, algorithms of Iliopoulos et al. [42] and Kociumaka et al. [49] as well as its simpler $\mathcal{O}(n \log n)$-time version for computing seeds, and a version of the algorithm of Kociumaka et al. [52] for computing partial covers. The C++ programs that were used for comparisons are available at https://github.com/heurezjusz/Quasiperiods.

Our experiments were conducted on Intel® Core™ i7-7700HQ processor with 16GB RAM. We performed experiments on random data, random data that are guaranteed to have an exact cover or seed and on data from the Pizza&Chili corpus.[2]

Structure of the paper In Section 3 we discuss in detail known linear-time algorithms for computing covers [9,13,54,55,25] and a folklore $\mathcal{O}(n \log n)$-time algorithm and compare their efficiency on the most basic problem of computing the shortest cover. For this, we use random data as well as semi-random data that is guaranteed to have non-trivial covers. In Section 4 we make the same type of comparison of algorithms for computing seeds: the $\mathcal{O}(n \log n)$-time algorithm [42], the simplified version [49] of the $\mathcal{O}(n)$-time algorithm [50] and a simpler $\mathcal{O}(n \log n)$-time version of [49]. Here the algorithms are much more involved, so their descriptions are necessarily rather sketchy. Finally, in Section 5 we discuss approximate variants of string quasiperiodicity, distinguish partial covers as currently the most promising on large real-world data, and test the $\mathcal{O}(n \log n)$-time algorithm for partial covers [52] both on artificial data and data from the Pizza&Chili corpus. For the $\mathcal{O}(n \log n)$-time implementations of the algorithms from [49,52], we use a compact segment tree data structure with an especially simple implementation that we describe in Appendix C.

2. Preliminaries

The letters of a string T are numbered 1 through $|T|$. By $T[i]$ we denote the i-th letter of T and by $T[i, j]$ we denote $T[i] \ldots T[j]$ which we call a substring of T. A substring of T other than T itself is called *proper*. For strings T and X, we denote $\mathsf{Occ}_T(X) = \{i : T[i, i + |X| - 1] = X\}$.

We say that an integer p is a *period* of a string T if $T[i] = T[i + p]$ for all $i = 1, \ldots, |T| - p$. By $\mathrm{per}(T)$ we denote the smallest period of T. The string T is called *periodic* if $2\,\mathrm{per}(T) \le |T|$. We say that a string B is a *border* of T if B is both a prefix and a suffix of T. The border array $\mathcal{B}$ of T stores, as $\mathcal{B}[i]$, the length of the longest proper border of $T[1, i]$. It can be computed in linear time using the Knuth–Morris–Pratt (KMP) algorithm [48].

[2] http://pizzachili.dcc.uchile.cl.

For a set of integers $A = \{a_1, \ldots, a_k\}$, $a_1 < a_2 < \cdots < a_k$, by $\mathrm{maxgap}(A)$ we denote the maximum distance between consecutive elements of A: $\mathrm{maxgap}(A) = \max\{a_i - a_{i-1} : i = 2, \ldots, k\}$. If $|A| \leq 1$, we assume that $\mathrm{maxgap}(A) = \infty$.

Definition 2.1. A string C is a *cover* of a string T if

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

A string S is a *seed* of T if $|S| \leq |T|$ and S is a cover of some string containing T as a substring.

3. Computing covers

In this section we compare algorithms for computing covers. They are all relatively simple comparing to the algorithms for computing seeds, so we can afford to explain them with more details. We start by two algorithms which only compute the shortest cover of a string, a folklore one and the algorithm of Apostolico et al. [9], proceed to the algorithm of Breslauer [13] which computes the shortest cover of every prefix of the string and finish by the algorithm by Moore and Smyth [54,55] which computes all covers of a string. We decided not to consider the algorithm of Li and Smyth [53] that computes the longest cover array of a string since it is much more involved. Yet another linear-time algorithm for computing the shortest cover can be inferred as a special case of the linear-time algorithm for computing so-called enhanced covers [29]; see Section 5. We also did not consider this algorithm in the following comparison.

The first two algorithms are based on a relation between covers and borders.

Observation 3.1. *Let T be a string.*

(a) A cover of T is a border of T.
(b) If T has a cover C and a border B such that $|C| \leq |B|$, then C is also a cover of B.
(c) The shortest cover of T is not periodic.

A string is called *superprimitive* if it is equal to its shortest cover, and *quasiperiodic* otherwise.

3.1. Computing shortest cover

We describe two algorithms that compute only the shortest cover of a string, a folklore $\mathcal{O}(n \log n)$-time algorithm and the $\mathcal{O}(n)$-time algorithm of Apostolico et al. [9]. Let $b_1 < \cdots < b_k$ be the sequence of lengths of all borders of T. By Observation 3.1(a), these are all candidates for a cover of T. Both algorithms are based on a known fact that the sequence $b_1, \ldots, b_k$ can be partitioned into at most $\log n$ subsequences, each of which is an arithmetic sequence (for a proof, see e.g. [24, Fact 5]). Moreover, if any of these arithmetic sequences has at least 3 elements and difference p, then all the borders in this subsequence excluding possibly the first one are periodic with period p.

The algorithms use the fact that testing if a string S is a cover of a string T can be done in linear time by computing the set $\mathrm{Occ}_T(S)$ using, say, the KMP algorithm and then applying the operation maxgap. The border array can also be used to compute all the borders of a string in linear time.

In the first algorithm we notice that if $b_i \leq 2b_{i-1}$, then the border of length b_i is periodic. Hence, it cannot be the shortest cover by Observation 3.1(c). The number of the remaining indices i, for which $b_i > 2b_{i-1}$, does not exceed $\log_2 n$. Hence, we can check each of them as a candidate for the cover length. Let us call this algorithm **Folk**.

The algorithm of Apostolico et al. [9], further denoted as **AFI**, is recursive. It starts by computing the longest border B of T. If $|B| \leq \frac{2}{3}n$, it recursively finds the shortest cover of this border and checks whether it is a cover of T. If it is, the shortest cover of the border is returned as an answer, and otherwise the result is the whole T. Correctness of this step follows by Observation 3.1(b). Otherwise, B is periodic (and T is periodic as well) and instead of B it takes the longest shorter border B' of T that is not periodic with the same period. We have $|B'| = p + n \bmod p$, where $p = n - |B|$, so $|B'| \leq \frac{2}{3}n$. The recurrence on the time complexity, $T(n) = T(\frac{2}{3}n) + \mathcal{O}(n)$, yields $T(n) = \mathcal{O}(n)$.

3.2. Computing shortest cover on-line

The algorithm of Breslauer [13], further denoted as **Bres**, is arguably the most popular algorithm for computing covers. It computes the shortest cover of every prefix of the string in an on-line manner.

The algorithm makes use of three arrays $\mathcal{B}$, $\mathcal{C}$, $\mathcal{R}$. Here $\mathcal{B}$ is the border array and $\mathcal{C}[i]$ is the length of the shortest cover of $T[1, i]$. The array $\mathcal{R}$ is defined as follows. Assume that the prefix $T[1, k]$ has been processed so far. Then for every $i = 1, \ldots, k$, $\mathcal{R}[i]$ is not defined if $T[1, i]$ is not superprimitive, and otherwise $\mathcal{R}[i]$ stores the length of the longest prefix of $T[1, k]$ such that $T[1, i]$ is its cover. In the end, $\mathcal{C}[n]$ contains the length of the shortest cover of T. See the pseudocode below.

In the if-statement of the algorithm we check if the shortest cover of the border $T[1, \mathcal{B}[k]]$ covers $T[1, k]$. If so, then it is the shortest cover of $T[1, k]$ by Observation 3.1(b). Otherwise, $\mathcal{B}[k] = 0$ or $\mathcal{C}[\mathcal{B}[k]]$ does not cover $T[1, k]$, so $T[1, k]$ is superprimitive.

Algorithm 1: Breslauer's algorithm.

```
for k := 1 to n do
    if B[k] > 0 and R[C[B[k]]] ≥ k − C[B[k]] then
        C[k] := C[B[k]]
        R[C[k]] := k
    else
        C[k] := R[k] := k
```

3.3. Computing all covers

The algorithm of Moore and Smyth [54,55] computes all the covers of a string T. We describe its slightly simplified version that can be inferred from the work of Crochemore et al. [25].

The algorithm creates a list L of all positions $\{1, \ldots, n\}$ with an additional element $n+1$. It performs n steps. In step i it removes from L all positions j which are not starting positions of occurrences of $T[1, i]$. Hence, after step i the list L represents $\mathsf{Occ}_T(T[1, i])$. The algorithm stores the maxgap of L. If after the ith phase the maxgap of remaining elements is smaller than or equal to i, then $T[1, i]$ is a cover of T. The maxgap of L can only increase, since the elements 1 and $n+1$ are never removed. When an element is removed from L, the maxgap is maximized with the difference of its next and previous elements in L. Elements can be removed efficiently with the aid of an additional array that stores the position of each element from $\{1, \ldots, n+1\}$ in L.

We only need to specify how to identify the elements of the list that are not occurrences of $T[1, i]$. The original algorithm [55] used a data structure called a *border tree*. A border tree is a rooted tree that contains a node for each position $i \in \{0, \ldots, n\}$ in T. The parent of i is the node $B[i]$. We chose a different, more direct solution [25] that uses the *prefix array* Pref which holds, as $\mathsf{Pref}[j]$, the length ℓ of the longest prefix of T such that $T[1, \ell] = T[j, j+\ell-1]$. The prefix array can be computed in $\mathcal{O}(n)$ time by a simple left-to-right scan [28]. Then position j should be removed from the list L at step number $\mathsf{Pref}[j] + 1$.

We denote the resulting algorithm as **MS**.

3.4. Experimental evaluation

As it was mentioned in the beginning of this section, the algorithms serve different purposes. We compare their efficiency on the fundamental task of computing the shortest cover. Hence, this comparison is necessarily limited as it does not convey their different capabilities.

For a string S, an *equality relation* states that a given pair of substrings, $S[i, i+\ell]$ and $S[j, j+\ell]$, match. Such relation is specified by a triple (i, j, ℓ). In [30] a linear-time algorithm is presented that, given a collection of equality relations, constructs a string of a specified length over the largest possible alphabet that satisfies these relations. We used a simpler, $\mathcal{O}(n \log n)$-time version of this algorithm that was also described in [30] to generate quasiperiodic strings with a given set of occurrences of a cover.

General conclusions Graphs that present the running time of the algorithms can be found in Appendix A.1. The best running times on the tests was achieved by the algorithms **Folk** and **AFI**, both achieving similar times. Performance of on-line algorithm **Bres** depended on string properties. The **MS** algorithm, capable of computing all covers of a string, was the slowest.

*Two versions of **MS** algorithm* Because the algorithm computes a lot of auxiliary values, we compared its straightforward implementation with a second version that uses static arrays instead of dynamic `std::vector`. It is visible that static arrays take noticeable time for initialization, but it pays off in tests with longer strings.

Periodicity of the string For this and the following paragraphs, see the graphs in Appendix A.2. The algorithm **Bres** slows down for small periods. One can notice that the slowdown seems to be linear in border length. The cause might be that in case of finding a cover of some prefix $T[1, i]$ shorter than i, which happens quite often for short periods, the algorithm performs more write operations. The smaller the period is, the sooner the algorithm finds the cover which repeats with every period. This might create the effect visible on the plot.

The running times of algorithms **Folk** and **AFI** clearly depend on the period. An interesting split point is when the period reaches $\frac{n}{2}$. The cause is probably that the **AFI** algorithm performs linear-time checks on prefixes of the input string, while the $\mathcal{O}(n \log n)$-time algorithm checks cover candidates on the whole string. The difference shows up when the input string has a short border.

Also there is a noticeable slowdown of the **MS** algorithm for short periods (i.e. 2, 3). The reason is that for such strings the Pref array achieves the largest values, and computation time of Pref depends on its maximum value.

Alphabet size The running time does not seem to depend on the alphabet size. For small alphabets, all approaches have slightly worse execution times, but the reason is that strings consisting of less different letters are more likely to have some kind of regularity, which, as the experiments have shown, increases the execution times.

Shortest cover length There is no clear dependency on the cover length. Fluctuations of performance suggest that the running time probably depends on other properties of the string (like periodicity) caused by the existence of a cover.

4. Computing seeds

We consider three algorithms for computing seeds in a string of length n: an $\mathcal{O}(n \log n)$-time algorithm by Iliopoulos et al. [42], an $\mathcal{O}(n)$-time algorithm of Kociumaka et al. [49] and its simpler $\mathcal{O}(n \log n)$-time version.

Each of the three algorithms computes a linear representation of all seeds of the input string T. The representation is a set of packages. A single package is specified with three integers i, j_1, j_2 and represents strings $T[i, j_1]$, $T[i, j_1 + 1]$, …, $T[i, j_2]$. For example, for $T = aabcab$, the package $(2, 4, 5)$ represents two strings: *abc* and *abca*. Kociumaka et al. [49] prove that all seeds of T can be represented as a linear number of such packages.

Consider an edge of the suffix tree of T, connecting two nodes u (parent) and v (child), denoted as e_{uv}. Let us denote by $|w|$ the depth of node w, which also is the length of the substring represented by this node. Note that all substrings represented by the edge e_{uv} (excluding u and including v) can be represented by a single package $(i_{suf}, i_{suf} + |u| + 1, i_{suf} + |v|)$, where i_{suf} is the starting position of the suffix of T represented by any leaf in the subtree of the node v.

The algorithm of Iliopoulos et al. [42] also uses packages on the reversed string T. An efficient way to convert between reversed and usual package representations is not known.

The main difference in the algorithms lies in how the maxgap values are computed for sets that represent substrings of the string T with the same occurrences.

4.1. $\mathcal{O}(n \log n)$-time algorithm of Iliopoulos et al. [42]

Let us call this algorithm **IMP**. The algorithm uses Crochemore's partitioning; see [22] (see also [23, Section 9.1]). The partitioning performs n steps; on the i-th step it maintains $\mathsf{Occ}_T(S)$ as a sorted list for all i-length substrings S of T. The total time complexity of partitioning is $\mathcal{O}(n \log n)$.

Iliopoulos et al. show that every substring of T of length greater than or equal to $\mathsf{per}(T)$ is a seed of T. Their algorithm computes $\mathsf{per}(T)$ and reports $\mathcal{O}(\mathsf{per}(T))$ packages with seeds of length at least $\mathsf{per}(T)$. Then it performs $\mathsf{per}(T)$ steps of Crochemore's partitioning to find all shorter seeds.

Iliopoulos et al. use the notion of a *candidate set*. A candidate set consists of substrings W_{i_0}, W_{i_1}, …, W_{i_l} of T such that $|W_{i_j}| = i_j = i_0 + j$ and $\mathsf{Occ}_T(W_{i_j}) = \mathsf{Occ}_T(W_{i_0}) = \{pos_1, pos_2, \ldots, pos_k\}$ for all $j = 0, \ldots, l$. Note that a candidate set can be represented as a single package $(pos_1, pos_1 + i_0 - 1, pos_1 + i_l - 1)$. The authors show how to determine the set of seeds from a candidate set in $\mathcal{O}(1)$ time, using pos_1, pos_k, i_0, i_l, $\mathsf{maxgap}(\mathsf{Occ}_T(W_{i_0}))$ and a number of auxiliary arrays that are precomputed in $\mathcal{O}(n)$ time using the KMP algorithm. Due to the auxiliary arrays the algorithm needs to be run twice – once on reversed T, what implies using reversed packages on the output.

The candidate sets can be found by Crochemore's partitioning. Let us assume that a single $\mathsf{Occ}_T(S)$ list was created on step i_0 and was partitioned into smaller lists in step i_l. It represents a candidate set of substrings W_{i_0}, W_{i_1}, …, W_{i_l}. Note that in Crochemore's partitioning, after elements from some list have been removed, the list does not have to be empty. In this case the remaining elements represent $\mathsf{Occ}_T(W)$ for some string W. This is the second way the Occ_T lists are created in Crochemore's partitioning.

The candidate set can be computed from $\mathsf{Occ}_T(W_{i_l})$ list just before it will be partitioned into smaller lists. The positions pos_1 and pos_k can be found trivially, because the list is sorted. We can extend the list by the moment of its creation to find i_0. The problem occurs when we want to extend the list by its maxgap, which will be updated in $\mathcal{O}(1)$ time when some element is removed. The problem is easy when one can assume that the smallest and the largest element are never removed from the list, as it was the case in the algorithm of Moore and Smyth [54,55] that was described in Section 3.3. Unfortunately, it does not have to be the case during the partitioning.

For this issue, Christou et al. [20] proved that, in case of seeds computation, one can skip removing extreme elements from the list without losing correctness. Thus maxgap can be easily maintained, simply by not updating the maxgap value when an extreme element is removed.

4.2. $\mathcal{O}(n \log n)$-time version of the algorithm of Kociumaka et al. [49]

The paper operates on the notions of *a left candidate*, *a right candidate* and *a quasiseed*. A string S is a *left candidate* if for some i it is a suffix and a seed of $T[1, i]$. A string S is a *right candidate* if for some i it is a prefix and a seed of $T[i, n]$. Finally, a string S is a *quasiseed* if $\mathsf{maxgap}(\mathsf{Occ}_T(S)) \le |S|$. Kociumaka et al. prove that a string S is a seed of T if and only if it fulfills all three above definitions.

The paper shows that sets of left and right candidates can be found in $\mathcal{O}(n)$ time and their package representations can be computed within this complexity. It also presents a method of finding an intersection of sets of substrings represented as packages in linear time. The algorithm does not use reversed packages. Also no seeds in the result are duplicated.

The original method for computing quasiseeds in $\mathcal{O}(n)$ time is described in Section 4.3. Here we describe a simpler method working in $\mathcal{O}(n \log n)$ time that we used in the first implementation, called **KKRRW-s**.

Let us denote as e_{uv} an edge of the suffix tree of T connecting the nodes u (parent) and v (child). All substrings of T represented by this edge have the same Occ_T – they occur at the beginning of all suffixes of T represented by leaves in the subtree of v. The package containing all quasiseeds of T represented by e_{uv} is $(i_{suf}, i_{suf} + \max(|u| + 1, \mathsf{maxgap}(\mathsf{Occ}_T(v))), i_{suf} + |v|)$, where i_{suf} represents a starting position of any suffix of T in the subtree of v. Obviously if $\mathsf{maxgap}(\mathsf{Occ}_T(v)) > |v|$, the edge e_{uv} does not contain any quasiseeds.

To find all quasiseeds of T, one needs to compute $\mathsf{maxgap}(\mathsf{Occ}_T(v))$ for every node v. In our implementation we used Joinable Segment Trees. Implementation details are described in Appendix C. A similar data structure was introduced in [47].

4.3. $\mathcal{O}(n)$-time algorithm of Kociumaka et al. [49]

We denote this algorithm as **KKRRW**. The difference between the algorithms **KKRRW-s** and **KKRRW** lies in maxgap computation.

Kociumaka et al. prove a *Gap Lemma* which states that:

Lemma 4.1. *If a string T has at least $\frac{2}{3}n - k + 1$ substrings of length k, then T has no seed of length ℓ such that $2k - 2 \le \ell \le \frac{n}{6}$.*

Note that "at least $\frac{2}{3}n - k + 1$ substrings of length k" can be described also as "at least $\frac{2}{3}n$ substrings of length k including all suffixes of T of length smaller than k"; the number of length-k substrings of T increased by $k - 1$ is further denoted as $\beta_k(T)$. One can easily show that $\beta_i(T) \le \beta_{i+1}(T)$ for all i.

The algorithm computes the maximum x such that $\beta_{4x-3}(T) < \frac{2}{3}n$ (it can be done in $\mathcal{O}(n)$ time using the suffix tree of T). Then it computes all seeds of T divided into three groups according to the Gap Lemma: *long seeds*, of length $\ell > \frac{n}{6}$, in $\mathcal{O}(n)$ time; *medium seeds*, of length ℓ such that $x < \ell < 2k - 2 = 2(4x - 3) - 2 < 8x$, in $\mathcal{O}(n)$ time; and short seeds, of length $\ell < x$, recursively on a string of length $\frac{2}{3}n$.

Long seeds These are the seeds of length $\ell > \frac{n}{6}$. In the Gap Lemma the upper limit on ℓ does not depend on k. We can use this fact to count all long seeds in $\mathcal{O}(n)$ time. Left and right candidates are computed in the same way as in the $\mathcal{O}(n \log n)$-time version of the algorithm. In the computation of quasiseeds, we compute $\max(\mathsf{maxgap}(\mathsf{Occ}_T(v)), \frac{n}{6} + 1)$ for every node v of the suffix tree of T and then deduce the packages describing quasiseeds for every edge e_{uv} based on the values in v. The intersection of these three package representations can be computed in $\mathcal{O}(n)$ time.

Thanks to the fact that we want to compute only quasiseeds of length greater than $\frac{n}{6}$, the algorithm computing $\mathsf{maxgap}(\mathsf{Occ}_T(v))$ values for every node v is much simpler. We can split the list of positions $1, \ldots, n$ into 6 blocks of equal lengths and for each block remember the first occurrence and the last occurrence of v in T. Values $\mathsf{maxgap}(\mathsf{Occ}_T(v))$ that are greater than $\frac{n}{6}$ can be computed correctly from the 12 values remembered in the 6 blocks. We can create and update such a block structure in $\mathcal{O}(1)$ time, thus the total work necessary to compute long seeds is $\mathcal{O}(n)$.

Medium seeds These are the seeds of length ℓ such that $x < \ell < 2k - 2 = 2(4x - 3) - 2 < 8x$. In this case, a fact that a string S is a seed of T if and only if it is a seed of every substring of T of length $2|S| - 1$ is used. Kociumaka et al. show that it is sufficient to compute seeds of substrings of T of length $16x$ with step $8x$, that is $T[1, 16x]$, $T[8x + 1, 24x]$, $T[16x + 1, 32x]$ and so on. The total length of the selected substrings is $2n$. Note that we are looking for seeds of length at least x, so we can use a similar method as in the computation of long seeds, but with 16 blocks. Intersection of solutions found for all such substrings can be computed in $\mathcal{O}(n)$ time, thus the total work necessary to compute the medium seeds is $\mathcal{O}(n)$.

Short seeds Finally, these are the seeds of length $\ell < x$. Using the fact that was mentioned in the previous paragraph, we will find recursively seeds of all substrings of T of length $2x - 1$ and compute an intersection of the results.

The algorithm first marks all positions covered by first occurrences of all substrings of T of length $2x - 1$ (the starting positions of these occurrences can be read from the suffix tree of T). Then it recursively computes the seeds of strings created by maximal intervals of the marked positions. Note that every marked position p either is the middle letter of the first occurrence of a string $T[p - 2x + 1, p + 2x - 1]$ of length $4x - 3$ or is contained in the prefix $T[1, 2x]$. Because $\beta_{4x-3} < \frac{2}{3}n$, the total number of marked positions is at most $\frac{2}{3}n$.

Total work of the algorithm is limited by $c \cdot n + c \cdot \frac{2}{3}n + c \cdot \frac{2^2}{3^2}n + \ldots = c \cdot 3n = \mathcal{O}(n)$, where c is a constant.

4.4. Experimental evaluation

We used Ukkonen's construction [60] of the suffix tree. The children of a node were indexed in a hash map by the letters of the alphabet.

The experimental evaluation (see Appendix B.1) shows that the $\mathcal{O}(n \log n)$-time **KKRRW-s** algorithm worked faster on experimental data than its original $\mathcal{O}(n)$ version **KKRRW**. Performance of the **IMP** algorithm depended on the type of the string. Low performance of the **KKRRW** algorithm is probably a result of a large constant.[3]

On random data, the **KKRRW-s** algorithm had the best performance. The other two algorithms achieved similar running times. On the other hand, on periodic strings the **IMP** algorithm becomes slightly better. The reason behind this is probably that the performance of the **IMP** algorithm strongly depends on the period and the shortest period of a random string is not likely to be smaller than n.

We checked dependencies of algorithm's performance on multiple properties of a string. The most interesting results are listed below.

String period The plot clearly shows a linear dependency of the **IMP** algorithm on period length. For periods longer than $\frac{n}{2}$ it starts to behave like for random strings. The dependency of **KKRRW** algorithm is really interesting – it shows for which periods it starts to make recursive calls. We can see that the plot suddenly drops for a period about $\frac{2}{3}n$, which is the limit for the first recursive call. The rule for subsequent recursive calls is more complicated, thus it is harder to predict the positions of smaller peaks.

Alphabet size (See Appendix B.2.) We expected that the running times of all algorithms would not depend on the alphabet size. The outcome shows that the **IMP** and **KKRRW** algorithms performs slower for a smaller alphabet. For **IMP** the reason is the Crochemore's partitioning mechanism. For a smaller alphabet, when a list is split into shorter lists, it produces on average a smaller number of lists that are longer, which results in more splits overall. **KKRRW** slows down for a similar reason as for algorithms computing the shortest cover – strings consisting of a smaller number of letters are more likely to have some regularity, which slows down this solution.

Experiments do not show any dependency of shortest seed length on algorithms' performance. The huge fluctuations on the plot are a result of other string properties, which differed between tests.

5. Approximate and generalized quasiperiodicity

5.1. Comparison of known approaches

While quasiperiodicity was introduced as a generalization of periodicity [8], the notions of covers and seeds can still be too restrictive in discovering repeating patterns in strings. To address this issue, approximate and generalized notions of quasiperiodicity were introduced. All the algorithms mentioned below work for strings over any ordered alphabet.

If one allows just a portion of positions of the string to be covered, the notion of a *partial cover* is obtained. More precisely, we seek for all substrings of the string that cover at least a given number α of positions. E.g., *aba* is a partial cover of *abababbaba* which covers 8 positions of the string. Partial covers can be computed fast, in $\mathcal{O}(n \log n)$ time [52]. If a string has a partial cover that covers α positions, then a maximum set of non-overlapping occurrences of this partial cover covers at least $\alpha/2$ positions. Hence, finding a substring with a maximum number of non-overlapping occurrences [15] gives a good approximation of a partial cover. Another notion, of an *enhanced cover*, requires the partial cover to be a border of the string. Enhanced covers can be computed in $\mathcal{O}(n)$ time [29], but at the same time they are more restrictive. A generalization of partial covers are *partial seeds*, in which letters covered by overhanging occurrences are also counted. They can also be computed in $\mathcal{O}(n \log n)$ time [51]. The gain in comparison with partial covers takes place only near the ends of a string.

One can also consider the case that a given string S does not have an exact cover, but it is at a small distance from a string T that has a proper exact cover. This way, we obtain the notion of an *approximate cover*, as the shortest proper cover of a string T that has the minimum Hamming distance from the given string S. E.g., *aba* is an approximate cover of *abababbaba* because it is an exact cover of *ababaababa* that is at Hamming distance 1 from the string. The problem of computing an approximate cover of a string is NP-hard [4]. Several relaxations of this problem were considered. If one knows the set of positions where the approximate cover occurs in the string T, then it can be computed in linear time [4,5]. Unfortunately, this does not help if we know just the string S. Assuming that the approximate cover has at least one exact occurrence in the given string S, it can be computed in polynomial time. However, the fastest known algorithm works in $\mathcal{O}(n^4)$ time [3], which is unrealistic for large data.

There is also a different definition of an *approximate cover*, in which one allows to cover the string with a family of strings that are all similar to one string that is considered the approximate cover. For the similarity measure, Hamming and edit distances were considered. This notion was introduced in [58] where it was shown that this version of approximate cover problem is NP-complete under weighted edit distance. In the variant that allows overhanging occurrences, one obtains a notion of *approximate seeds* that was considered in [17]; they also show that computing approximate seeds is NP-

[3] As an example, computing so-called medium seeds in the **KKRRW** algorithm requires computing 16 values for every node of a suffix tree. Overall, we have estimated that the linear-time algorithm has a chance of overtaking its $\mathcal{O}(n \log n)$ version on strings of length around 2^{50}. Unfortunately, available equipment has not allowed us to check this hypothesis.

complete under weighted edit distance. Both papers also considered the relaxation of the problem in which the approximate cover has at least one exact occurrence in the given string S and computed, for every substring of S, the smallest distance threshold under which it is an approximate cover or seed. In this case, $\mathcal{O}(n^3)$-time algorithms for the Hamming distance and $\mathcal{O}(n^4)$-time algorithms for the weighted edit distance were developed for approximate covers and seeds in [58] and [17], respectively. Approximate covers and seeds under Hamming distance were also considered in [38] and [36], respectively. They developed automata-based $\mathcal{O}(n^3k)$-time solutions for variants of these problems where the maximal Hamming distance is bounded by k. Experimental evaluation of the algorithms was performed in [35]. Finally, [37] extended these definitions in a natural way to *approximate enhanced covers* and proposed an equally fast algorithm for solving this problem. (If, in addition, the approximate enhanced cover is required to be a border of the given string, it is shown that the problem can be solved in $\mathcal{O}(n^2)$ time.)

A different line of research is obtained if one is looking for, instead of one cover, a family of λ strings, all of the same length k, whose occurrences cover the given string S. Depending on which of the parameters is specified, we obtain the problem of λ-covers or k-covers; the other parameter is to be minimized. E.g., *abababbaba* has a cover (ab, ba) with $\lambda = k = 2$. The k-covers problem is NP-complete [21,41]. In [34] the authors claim that the λ-covers problem can be solved in $\mathcal{O}(n^2)$ time for a constant λ and a constant-sized alphabet. The analogously defined λ-seeds problem was considered in [32], where an $\mathcal{O}(n^2)$-time algorithm is proposed, and a problem of approximate λ-covers was considered in [33], where an $\mathcal{O}(n^4)$-time algorithm is proposed; in both cases λ and the size of the alphabet were assumed to be constant. In [34] it is stated that the algorithm actually computes all the sets of λ strings of equal length that cover the given string, under additional constraints that $1 < k < n/\lambda$ and that no proper subset of the λ strings covers the string S. However, the example below shows that the number of λ-covers under such definition can still be $\Omega(n^\lambda)$, so they cannot be computed in $\mathcal{O}(n^2)$ time.

Example 5.1. Let the alphabet be $\{a, b_1, \ldots, b_{\lambda-1}\}$, for a constant λ, and consider the string

$$S = (a^m b_1)^2 (a^m b_2)^2 \ldots (a^m b_{\lambda-1})^2 a^m.$$

Then for every non-negative integers $i_1, j_1, \ldots, i_{\lambda-1}, j_{\lambda-1}$ such that

$$0 < i_1 + j_1 = \ldots = i_{\lambda-1} + j_{\lambda-1} < m,$$

$(a^{i_1} b_1 a^{j_1}, \ldots, a^{i_{\lambda-1}} b_{\lambda-1} a^{j_{\lambda-1}}, a^{i_1+j_1+1})$ forms a λ-cover of S consisting of strings of length $i_1 + j_1 + 1$. The number of such λ-covers is $\Omega(m^\lambda) = \Omega(|S|^\lambda)$. Indeed, for every $\frac{m}{2} \le i_1 + j_1 < m$, there are at least $\frac{m}{2}$ choices of the value of each of the parameters $i_1, \ldots, i_{\lambda-1}$.

In conclusion, we have decided to implement and test the algorithm for partial covers, as a good example of a known algorithm for computing approximate quasiperiodicity with fast running time.

5.2. Partial covers

In this problem we are to find all shortest substrings of a string T whose occurrences cover at least α positions of T.

The paper [52] provides the definition of a *Cover Suffix Tree* (CST) of T (previously a similar data structure called Minimal Augmented Suffix Tree, MAST in short, was known [10]), which is a suffix tree of T where implicit nodes representing primitively rooted squares of T are converted to explicit ones. A non-empty string U is *primitive*, if for any string V and integer k, $V^k = U$ implies $V = U$. *A primitively rooted* square is a string U^2, where U is *primitive*. By known combinatorial results on squares in strings, this way only a linear number of nodes will be added.

The paper shows how to construct a CST of T in $\mathcal{O}(n \log n)$ time with its nodes annotated by two values: $cv(v)$ and $\Delta(v)$; $cv(v)$ is the number of positions of T covered by occurrences of the string represented by node v and $\Delta(v)$ is the number of maximal fragments of T that are fully covered by occurrences of v. More formally, $\Delta(v) = 1 + |\{i \in \mathsf{Occ}_T(v) : j - i > |v|, j \text{ is a successor of } i \text{ in } \mathsf{Occ}_T(v)\}|$.

Consider an edge e_{uv} of CST, connecting two explicit nodes u (parent) and v (child). All implicit nodes on e_{uv} have the same value Δ, which equals $\Delta(v)$. This means that for an implicit node w, $\Delta(w) = \Delta(v)$ and $cv(w) = cv(v) - (|v| - |w|)\Delta(v)$. Because cv values of all nodes on edge e_{uv} form an arithmetic sequence, it is easy to determine whether on e_{uv} there is a node w such that $cv(w) \ge \alpha$ and find all such nodes with minimal depth.

This problem can be extended to *All Partial Covers* problem in which for each $\alpha \in \{1, \ldots, n\}$ we are to find any shortest substring of T whose occurrences cover at least α positions of T. Consider an edge e_{uv} containing k implicit nodes. We can represent it as a segment on $\mathbb{N}^2$ plane connecting points $(|v|, cv(v))$ and $(|v| - k, cv(v) - k\Delta(v))$. The upper envelope of such segments can be found in $\mathcal{O}(m \log m)$ time, where m is the number of segments. Then point (x, y) of such envelope represents that y positions can be covered by a substring of length x. One can find examples of such substrings by labeling segments by edges id. This solution of *All Partial Covers* problem works in $\mathcal{O}(n \log n)$ time.

One of subproblems that appear in computing partial covers is an extended Disjoint Set Union problem, in which we require that a Union operation on sets A and B returns a *change list*. A change list consists of pairs $(x, next[x])$,

where $x \in (A \cup B)$ and $next[x]$ is a successor of x in this set, such that x is included in a change list if and only if the value $next[x]$ changed as a result of the union. For example, the change list of the union of sets $\{1, 2, 3, 5\}$ and $\{4, 8\}$ is $[(3, 4), (4, 5), (5, 8)]$. The paper [52] solves this problem in $\mathcal{O}(n \log n)$ time using mergeable AVL trees. An $\mathcal{O}(n \log n)$-time solution of this problem with JSTs, that we used in our implementation due to simplicity of the data structure, is described in Appendix C.4.

5.3. Experimental evaluation

The experiments on Pizza&Chili corpus have shown that in most cases the string which covers the maximum number of positions of the text is a letter which occurs most of the times. The original algorithm computes the shortest partial cover for a given α, so such a letter overwrote all other partial covers. We slightly changed the algorithm to report all partial covers (not only the shortest one) covering at least α positions, but the results were similar to ones achieved for random strings. The only shortest partial covers longer than one letter were detected in XML source: "`journal`" covering 7.5% of the text and "`></article><articlemdate="200`" covering 7.6% of the text. However, both of them are border-free and this corresponds to a non-overlapping partial cover.

6. Conclusions

The main purpose of this work was a practical and, to some extent, theoretical comparison of algorithms for computing various types of quasiperiods in strings. From our study we conclude that the area of approximate quasiperiodicity seems to require further study in order to develop approaches that perform well on large data that come from applications like computational biology. We hope that this work will also be beneficial as a reference material on the state of the art in this area.

Declaration of competing interest

The authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest (such as honoraria; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

Acknowledgements

The authors thank Tomasz Kociumaka and Juliusz Straszyński for helpful discussions.

Appendices A and B. Supplementary material

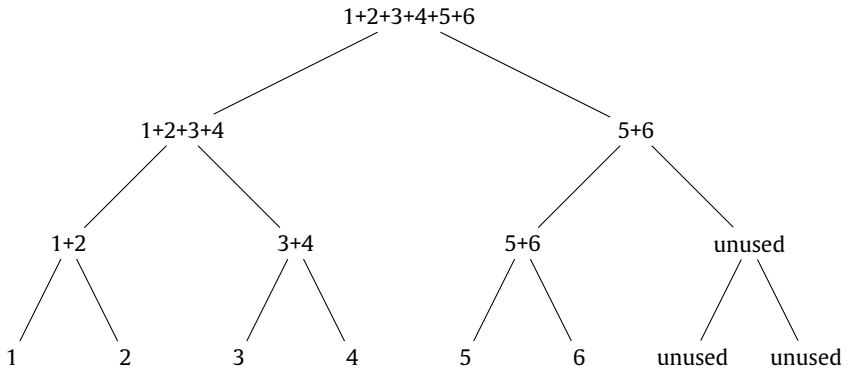
Supplementary material related to this article can be found online at https://doi.org/10.1016/j.tcs.2020.11.033.

Appendix C. Joinable segment trees

A *segment tree* (see, e.g., [57]), also called a static range tree, is a data structure used for maintaining operations on sequences of a fixed length N in $\mathcal{O}(\log N)$ time. As a toy example, it allows to change a given element of the sequence and report the sum of a fragment of a sequence.

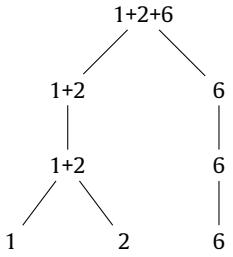
To create a segment tree, we choose the smallest power of 2, denoted as B (the base), which is greater than or equal to N. Then we build a full binary tree with B leaves. Consecutive leaves represent consecutive elements of the sequence. Last $B - N$ leaves remain unused. A node of the tree which is not a leaf, v, accumulates information about a sequence fragment represented by leaves in its subtree. We will call it *the fragment of* v. In the example that was mentioned above, v stores the sum of elements of its fragment. The values stored in nodes should be easy to update after changing the value in a leaf. In the toy example, we could replace the value in every node on the path from the changed leaf to the root (in this order) using the sum of values stored in the children of this node.

The figure below shows an example segment tree for $N = 6$.



C.1. Definition

By a *joinable segment tree* (JST) we mean a segment tree with memory optimization, that is, we will not store nodes which were not initialized with any value, and add them dynamically when they are necessary. Below there is an example of a JST with $N = 6$ in which only the values in the leaves 1, 2 and 6 were initialized.

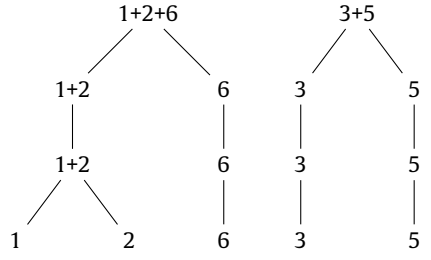


If we have two JSTs, T_A and T_B, containing disjoint sets of initialized leaves, we can join them into a JST T containing leaves from both T_A and T_B using the method shown below:

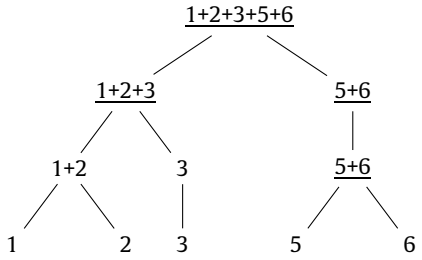
```
def join(v_A, v_B): // nodes of T_A and T_B
    if v_A is None:
        return v_B
    if v_B is None:
        return v_A

    l' = join(left children of v_A and v_B)
    r' = join(right children of v_A and v_B)

    assign (l', r') as left and right children of v_A
    update(v_A)
    return v_A
```

The function `update` recomputes the value in v_A based on values in its children.

For example, joining a JST with leaves 1, 2, 6 with a JST with leaves 3, 5:



gives the following result (updated nodes are underlined):



C.2. Complexity

The total time complexity of creating and joining N JSTs, i-th of which containing only the i-th leaf initialized, is $\mathcal{O}(N \log N)$ and does not depend on the order of joins.

Indeed, let us denote the length of the fragment of node v by $F(v)$. In a full binary tree, the subtree of v would contain exactly $2F(v) - 2$ edges. If during a `join` operation node v was updated, it means that at least one edge was added in its subtree. This means that during all `join` operations node v will be updated at most $2F(v) - 2$ times. In total all nodes with equal $F(v)$ value will be updated $\frac{B}{F(v)}(2F(v) - 2) = \mathcal{O}(B) = \mathcal{O}(N)$ times. There are $\log B$ different values of $F(v)$, thus the total time complexity of all `join` operations is $\mathcal{O}(N \log N)$.

C.3. Maxgap problem

In this section we will describe how to compute $\mathsf{maxgap}(\mathsf{Occ}_T(v))$ for every node v of the suffix tree of T.

For every node v of the suffix tree (from the leaves to the root) we compute a JST representing $\mathsf{Occ}_T(v)$. Leaf i in such JST is initialized if and only if $i \in \mathsf{Occ}_T(v)$. In every node v of the JST we store the minimum and the maximum element of $\mathsf{Occ}_T(s)$ contained in the subtree of v and maxgap of all the elements from the subtree of v. The function `update` will look as following:

```
def update(v):
    if one of children of v is None:
        copy values from the existing child
    else:
        v.minval = v.left.minval
        v.maxval = v.right.maxval
        v.maxgap = max(v.left.maxgap, v.right.maxgap, v.right.minval - v.left.maxval)
```

To compute $\mathsf{maxgap}(\mathsf{Occ}_T(v))$ we need to join all JSTs of the children of v.

C.4. Extended disjoint set union

Here we describe how to extend the Union operation in Disjoint Set Union with computing the *change list*, that is needed for computation of partial covers. The Find operation is performed by a classical Find and Union structure. Extension for Union operation is performed on Joinable Segment Trees.

At the beginning we create n sets with labels $1, 2, \ldots, n$. The i-th set is a JST with the i-th leaf initialized with value i. A node of JST stores the minimum and the maximum value in its subtree. While joining trees A and B we will mark values from tree A with color c_A, and values from tree B with color c_B. Then, in the `update` operation we compare the maximum value of the left child (l_{max}) with the minimum value of the right child (r_{min}). If the colors of l_{max} and r_{min} are different, we add the (l_{max}, r_{min}) pair to the resulting change list.

As we need colors only for the `update` operation, we do not need to mark values that are never used by it. So, if some node of the tree A is relinked as a child of the tree $A \cup B$, and we do not call `update` on it, both of its values are marked with c_A. An analogous rule holds for marking values in nodes of the tree B with c_B. Hence, `update` copies colors along with the values.

All changes of the successor will be added to the change list, because every two consecutive elements i, j from $A \cup B$ will be compared in an `update` operation on the lowest common ancestor of the leaves i and j.

References

[1] Ali Alatabbi, M. Sohel Rahman, William F. Smyth, Computing covers using prefix tables, Discrete Appl. Math. 212 (2016) 2–9, https://doi.org/10.1016/j.dam.2015.05.019.

[2] Amihood Amir, Costas S. Iliopoulos, Jakub Radoszewski, Two strings at Hamming distance 1 cannot be both quasiperiodic, Inf. Process. Lett. 128 (2017) 54–57, https://doi.org/10.1016/j.ipl.2017.08.005.

[3] Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, Benny Porat, Can we recover the cover?, Algorithmica 81 (7) (2019) 2857–2875, https://doi.org/10.1007/s00453-019-00559-8.

[4] Amihood Amir, Avivit Levy, Ronit Lubin, Ely Porat, Approximate cover of strings, Theor. Comput. Sci. 793 (2019) 59–69, https://doi.org/10.1016/j.tcs.2019.05.020.
[5] Amihood Amir, Avivit Levy, Ely Porat, Quasi-periodicity under mismatch errors, in: Gonzalo Navarro, David Sankoff, Binhai Zhu (Eds.), Annual Symposium on Combinatorial Pattern Matching, CPM 2018, in: LIPIcs, vol. 105, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 4, https://doi.org/10.4230/LIPIcs.CPM.2018.4.
[6] Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos, Inuka Jayasekera, Gad M. Landau, Conservative string covering of indeterminate strings, in: Jan Holub, Jan Žďárek (Eds.), Prague Stringology Conference 2008, Czech Technical University, Prague, 2008, pp. 108–115, http://www.stringology.org/event/2008/p10.html.
[7] Alberto Apostolico, Dany Breslauer, Of periods, quasiperiods, repetitions and covers, in: Jan Mycielski, Grzegorz Rozenberg, Arto Salomaa (Eds.), Structures in Logic and Computer Science, A Selection of Essays in Honor of Andrzej Ehrenfeucht, in: Lecture Notes in Computer Science, vol. 1261, Springer, 1997, pp. 236–248, https://doi.org/10.1007/3-540-63246-8_14.
[8] Alberto Apostolico, Andrzej Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Theor. Comput. Sci. 119 (2) (1993) 247–265, https://doi.org/10.1016/0304-3975(93)90159-Q.
[9] Alberto Apostolico, Martin Farach, Costas S. Iliopoulos, Optimal superprimitivity testing for strings, Inf. Process. Lett. 39 (1) (1991) 17–20, https://doi.org/10.1016/0020-0190(91)90056-N.
[10] Alberto Apostolico, Franco P. Preparata, Data structures and algorithms for the string statistics problem, Algorithmica 15 (5) (1996) 481–494, https://doi.org/10.1007/BF01955046.
[11] Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, Jakub Radoszewski, Indexing weighted sequences: neat and efficient, Inf. Comput. 270 (2020), https://doi.org/10.1016/j.ic.2019.104462.
[12] Amir M. Ben-Amram, Omer Berkman, Costas S. Iliopoulos, Kunsoo Park, The subtree max gap problem with application to parallel string covering, in: Daniel Dominic Sleator (Ed.), Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, ACM/SIAM, 1994, pp. 501–510, http://dl.acm.org/citation.cfm?id=314464.314633.
[13] Dany Breslauer, An on-line string superprimitivity test, Inf. Process. Lett. 44 (6) (1992) 345–347, https://doi.org/10.1016/0020-0190(92)90111-8.
[14] Dany Breslauer, Testing string superprimitivity in parallel, Inf. Process. Lett. 49 (5) (1994) 235–241, https://doi.org/10.1016/0020-0190(94)90060-4.
[15] Gerth Stølting Brodal, Rune B. Lyngsø, Anna Östlin, Christian N.S. Pedersen, Solving the string statistics problem in time $O(n \log n)$, in: Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, Ricardo Conejo (Eds.), Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, in: Lecture Notes in Computer Science, vol. 2380, Springer, 2002, pp. 728–739, https://doi.org/10.1007/3-540-45465-9_62.
[16] Gerth Stølting Brodal, Christian N.S. Pedersen, Finding maximal quasiperiodicities in strings, in: Raffaele Giancarlo, David Sankoff (Eds.), Combinatorial Pattern Matching, CPM 2000, in: Lecture Notes in Computer Science, vol. 1848, Springer, 2000, pp. 397–411, https://doi.org/10.1007/3-540-45123-4_33.
[17] Manolis Christodoulakis, Costas S. Iliopoulos, Kunsoo Park, Jeong Seop Sim, Approximate seeds of strings, J. Autom. Lang. Comb. 10 (5/6) (2005) 609–626.
[18] Michalis Christou, Maxime Crochemore, Ondrej Guth, Costas S. Iliopoulos, Solon P. Pissis, On left and right seeds of a string, J. Discret. Algorithms 17 (2012) 31–44, https://doi.org/10.1016/j.jda.2012.10.004.
[19] Michalis Christou, Maxime Crochemore, Costas S. Iliopoulos, Quasiperiodicities in Fibonacci strings, Ars Comb. 129 (2016) 211–225.
[20] Michalis Christou, Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, Tomasz Waleń, Efficient seed computation revisited, Theor. Comput. Sci. 483 (2013) 171–181, https://doi.org/10.1016/j.tcs.2011.12.078.
[21] Richard Cole, Costas S. Iliopoulos, Manal Mohamed, William F. Smyth, Lu Yang, The complexity of the minimum k-cover problem, J. Autom. Lang. Comb. 10 (5/6) (2005) 641–653.
[22] Maxime Crochemore, An optimal algorithm for computing the repetitions in a word, Inf. Process. Lett. 12 (5) (1981) 244–250, https://doi.org/10.1016/0020-0190(81)90024-7.
[23] Maxime Crochemore, Christophe Hancart, Thierry Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[24] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Wojciech Tyczyński, Tomasz Waleń, The maximum number of squares in a tree, in: Juha Kärkkäinen, Jens Stoye (Eds.), Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, in: Lecture Notes in Computer Science, vol. 7354, Springer, 2012, pp. 27–40, https://doi.org/10.1007/978-3-642-31265-6_3.
[25] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Covering problems for partial words and for indeterminate strings, Theor. Comput. Sci. 698 (2017) 25–39, https://doi.org/10.1016/j.tcs.2017.05.026.
[26] Maxime Crochemore, Costas S. Iliopoulos, Maureen Korda, Two-dimensional prefix string matching and covering on square matrices, Algorithmica 20 (4) (1998) 353–373, https://doi.org/10.1007/PL00009200.
[27] Maxime Crochemore, Costas S. Iliopoulos, Solon P. Pissis, German Tischler, Cover array string reconstruction, in: Amihood Amir, Laxmi Parida (Eds.), Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, in: Lecture Notes in Computer Science, vol. 6129, Springer, 2010, pp. 251–259, https://doi.org/10.1007/978-3-642-13509-5_23.
[28] Maxime Crochemore, Wojciech Rytter, Jewels of Stringology, World Scientific, 2003.
[29] Tomás Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, William F. Smyth, Wojciech Tyczyński, Enhanced string covering, Theor. Comput. Sci. 506 (2013) 102–114, https://doi.org/10.1016/j.tcs.2013.08.013.
[30] Paweł Gawrychowski, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Universal reconstruction of a string, Theor. Comput. Sci. 812 (2020) 174–186, https://doi.org/10.1016/j.tcs.2019.10.027, In memoriam Danny Breslauer (1968-2017).
[31] Paweł Gawrychowski, Jakub Radoszewski, Tatiana A. Starikovskaya, Quasi-periodicity in streams, in: Nadia Pisanti, Solon P. Pissis (Eds.), 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, in: LIPIcs, vol. 128, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, 22, https://doi.org/10.4230/LIPIcs.CPM.2019.22.
[32] Qing Guo, Hui Zhang, Costas S. Iliopoulos, Computing the λ-seeds of a string, in: Siu-Wing Cheng, Chung Keung Poon (Eds.), Algorithmic Aspects in Information and Management, AAIM 2006, in: Lecture Notes in Computer Science, vol. 4041, Springer, 2006, pp. 303–313, https://doi.org/10.1007/11775096_28.
[33] Qing Guo, Hui Zhang, Costas S. Iliopoulos, Computing the minimum approximate λ-cover of a string, in: Fabio Crestani, Paolo Ferragina, Mark Sanderson (Eds.), String Processing and Information Retrieval, 13th International Conference, SPIRE 2006, in: Lecture Notes in Computer Science, vol. 4209, Springer, 2006, pp. 49–60, https://doi.org/10.1007/11880561_5.
[34] Qing Guo, Hui Zhang, Costas S. Iliopoulos, Computing the λ-covers of a string, Inf. Sci. 177 (19) (2007) 3957–3967, https://doi.org/10.1016/j.ins.2007.02.020.
[35] Ondřej Guth, Searching Regularities in Strings using Finite Automata, PhD thesis, Czech Technical University in Prague, 2014, https://fit.cvut.cz/sites/default/files/PhDThesis_Guth.pdf.
[36] Ondřej Guth, Bořivoj Melichar, Using finite automata approach for searching approximate seeds of strings, in: Xu Huang, Sio-Iong Ao, Oscar Castillo (Eds.), Intelligent Automation and Computer Engineering, Springer Netherlands, Dordrecht, 2010, pp. 347–360, https://doi.org/10.1007/978-90-481-3517-2_27.
[37] Ondřej Guth, On approximate enhanced covers under Hamming distance, in: Stringology Algorithms, Discrete Appl. Math. 274 (2020) 67–80, https://doi.org/10.1016/j.dam.2019.01.015.

[38] Ondřej Guth, Bořivoj Melichar, Miroslav Balík, Searching all approximate covers and their distance using finite automata, in: Peter Vojtás (Ed.), Proceedings of the Conference on Theory and Practice of Information Technologies, ITAT 2008, in: CEUR Workshop Proceedings, vol. 414, 2008, CEUR-WS.org, http://ceur-ws.org/Vol-414/paper4.pdf.
[39] Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, Athanasios K. Tsakalidis, The weighted suffix tree: an efficient data structure for handling molecular weighted sequences and its applications, Fundam. Inform. 71 (2–3) (2006) 259–277, http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-07.
[40] Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina Perdikuri, William F. Smyth, Athanasios K. Tsakalidis, String regularities with don't cares, Nord. J. Comput. 10 (1) (2003) 40–51.
[41] Costas S. Iliopoulos, Manal Mohamed, William F. Smyth, New complexity results for the k-covers problem, Inf. Sci. 181 (12) (2011) 2571–2575, https://doi.org/10.1016/j.ins.2011.02.009.
[42] Costas S. Iliopoulos, Dennis W.G. Moore, Kunsoo Park, Covering a string, Algorithmica 16 (3) (1996) 288–297, https://doi.org/10.1007/BF01955677.
[43] Costas S. Iliopoulos, Laurent Mouchard, Quasiperiodicity and string covering, Theor. Comput. Sci. 218 (1) (1999) 205–216, https://doi.org/10.1016/S0304-3975(98)00260-6.
[44] Costas S. Iliopoulos, Laurent Mouchard, Quasiperiodicity: from detection to normal forms, J. Autom. Lang. Comb. 4 (3) (1999) 213–228.
[45] Costas S. Iliopoulos, Kunsoo Park, An optimal $O(\log\log n)$-time algorithm for parallel superprimitivity testing, J. Korean Inf. Sci. Soc. 21 (8) (1994) 1400–1404.
[46] Costas S. Iliopoulos, William F. Smyth, An on-line algorithm of computing a minimum set of k-covers of a string, in: Australasian Workshop on Combinatorial Algorithms, AWOCA 1998, 1998, pp. 97–106.
[47] Adam Karczmarz, A simple mergeable dictionary, in: Rasmus Pagh (Ed.), 15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, in: LIPIcs, vol. 53, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 7, https://doi.org/10.4230/LIPIcs.SWAT.2016.7.
[48] Donald E. Knuth, James H. Morris Jr., Vaughan R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350, https://doi.org/10.1137/0206024.
[49] Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, A linear time algorithm for seeds computation, ACM Trans. Algorithms 16 (2) (2020) 27:1–27:23, https://doi.org/10.1145/3386369.
[50] Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, A linear time algorithm for seeds computation, in: Yuval Rabani (Ed.), 23rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, SIAM, 2012, pp. 1095–1112, https://doi.org/10.1137/1.9781611973099.
[51] Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Efficient algorithms for shortest partial seeds in words, Theor. Comput. Sci. 710 (2018) 139–147, https://doi.org/10.1016/j.tcs.2016.11.035.
[52] Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Fast algorithm for partial covers in words, Algorithmica 73 (1) (2015) 217–233, https://doi.org/10.1007/s00453-014-9915-3.
[53] Yin Li, William F. Smyth, Computing the cover array in linear time, Algorithmica 32 (1) (2002) 95–106, https://doi.org/10.1007/s00453-001-0062-2.
[54] Dennis W.G. Moore, William F. Smyth, An optimal algorithm to compute all the covers of a string, Inf. Process. Lett. 50 (5) (1994) 239–246, https://doi.org/10.1016/0020-0190(94)00045-X.
[55] Dennis W.G. Moore, William F. Smyth, A correction to "An optimal algorithm to compute all the covers of a string", Inf. Process. Lett. 54 (2) (1995) 101–103, https://doi.org/10.1016/0020-0190(94)00235-Q.
[56] Alexandru Popa, Andrei Tanasescu, An output-sensitive algorithm for the minimization of 2-dimensional string covers, in: T.V. Gopal, Junzo Watada (Eds.), Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019, in: Lecture Notes in Computer Science, vol. 11436, Springer, 2019, pp. 536–549, https://doi.org/10.1007/978-3-030-14812-6_33.
[57] Mikhail Rubinchik, Arseny M. Shur, Counting palindromes in substrings, in: Gabriele Fici, Marinella Sciortino, Rossano Venturini (Eds.), String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, in: Lecture Notes in Computer Science, vol. 10508, Springer, 2017, pp. 290–303, https://doi.org/10.1007/978-3-319-67428-5_25.
[58] Jeong Seop Sim, Kunsoo Park, Sung-Ryul Kim, Jee-Soo Lee, Finding approximate covers of strings, J. Korean Inf. Sci. Soc. 29 (1) (2002) 16–21, http://www.koreascience.or.kr/article/ArticleFullRecord.jsp?cn=JBGHG6_2002_v29n1_16.
[59] William F. Smyth, Repetitive perhaps, but certainly not boring, Theor. Comput. Sci. 249 (2) (2000) 343–355, https://doi.org/10.1016/S0304-3975(00)00067-0.
[60] Esko Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260, https://doi.org/10.1007/BF01206331.