



PHD DISSERTATION

UNIVERSITY OF WARSAW

FACULTY OF MATHEMATICS, INFORMATICS AND MECHANICS

Exact Covers and Pattern Matching with Mismatches

Author:

Juliusz Straszyński

Supervisor:

Dr hab. Jakub Radoszewski

April 1, 2022

Author's declaration:

I hereby declare that this dissertation is my own work.

April 1, 2022

Date

Juliusz Straszyński

Supervisor's declaration:

This dissertation is ready to be reviewed.

April 1, 2022

Date

dr hab. Jakub Radoszewski

Abstract

This thesis consists of several results in the field of algorithms on strings. In particular, we consider variants of string covers and pattern matching with mismatches. From now on, we will use n to denote the length of the input text.

In the first part we consider exact variants of finding string covers, that is, testing whether a text can be generated by concatenations and superpositions of a shorter string.

In “Efficient Computation of 2-covers of a String” we present an algorithm which tests whether a text can be generated by concatenations and superpositions of 2 strings of the same length. The algorithm works in $\mathcal{O}(n \log n \log \log n)$ expected time and $\mathcal{O}(n)$ space.

“Shortest Covers of All Cyclic Shifts of a String” solves the titular problem in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. It also provides a precise description of the set of lengths of the shortest covers of all cyclic shifts of Fibonacci words.

“Internal Quasiperiod Queries” shows how to efficiently answer internal queries about various types of quasiperiod related queries. The paper presents an algorithm which allows to find the shortest cover on a given interval in $\mathcal{O}(\log n \log \log n)$ or all covers in $\mathcal{O}(\log n (\log \log n)^2)$ time and $\mathcal{O}(n \log n)$ space. We also present more efficient solutions in the offline model.

“String Covers of a Tree” considers a natural generalization of cover, where one can cover a tree with paths corresponding to the cover. The algorithm from the paper finds all covers in a directed tree in $\mathcal{O}(n \log n / \log \log n)$ time and in the case of undirected tree in $\mathcal{O}(n^2)$ time and space. In the case of undirected tree we provide an alternative algorithm using $\mathcal{O}(n)$ space and $\mathcal{O}(n^2 \log n)$ time.

The second part of the thesis provides asymptotically efficient solutions to practical problems.

“Efficient Computation of Sequence Mappability” considers a problem where one needs to compute mappability for each substring of length m with up to k mismatches. There, we provide a collection of results, most notably an $\mathcal{O}(n \min\{\log^k n, m^k\})$ time and $\mathcal{O}(n)$ space algorithm for fixed k and that one cannot solve (k, m) -mappability in strongly subquadratic time unless Strong Exponential Time Hypothesis fails.

“Circular Pattern Matching with k Mismatches” is a variant of pattern matching where we allow the pattern to be arbitrarily circularly shifted and up to k characters replaced. We provide a simple $\mathcal{O}(nk)$ time and $\mathcal{O}(m)$ space solution and more elaborate $\mathcal{O}(n + nk^4/m)$ time and $\mathcal{O}(m)$ space algorithm.

The results usually exploit periodicity analysis and apply a mix of classic stringology tools and state-of-the-art data structures.

2012 ACM Subject Classification: Theory of computation → Pattern matching

Keywords: Algorithm, Data Structure, String, Cover, Periodicity, Quasiperiodicity, Hamming Distance, Pattern Matching, Mappability, k -Mismatch

Streszczenie

Praca jest kompilacją serii wyników z dziedziny algorytmów tekstowych. W szczególności rozważamy warianty problemu znajdowania szablonu i wyszukiwania wzorca w tekście z niedopasowaniami. Jako n będziemy oznaczać długość wejściowego tekstu.

W pierwszej części rozpatrujemy dokładne warianty problemu znajdowania szablonów. W oryginalnym problemie szukamy takiego słowa, którego wystąpienia pokrywają dany tekst.

W pracy “Efficient Computation of 2-covers of a String” analizowane jest pojęcie 2-szablonów, czyli par słów o tej samej długości, których wystąpienia pokrywają dany tekst. W pracy przedstawiamy algorytm znajdujący dla każdej długości k przykładowy 2-szablon długości k (jeśli dla niej takowy istnieje) w oczekiwanym czasie $\mathcal{O}(n \log n \log \log n)$ i pamięci $\mathcal{O}(n)$.

“Shortest Covers of All Cyclic Shifts of a String” rozwiązuje problem znalezienia najkrótszych szablonów dla wszystkich cyklicznych przesunięć w czasie $\mathcal{O}(n \log n)$ i pamięci $\mathcal{O}(n)$. Dodatkowo, opracowany jest tam bardzo dokładny opis zbioru długości najkrótszych szablonów wszystkich cyklicznych przesunięć słów Fibonacciego.

W pracy “Internal Quasiperiod Queries” pokazujemy jak efektywnie odpowiadać na zapytania wewnętrzne o najkrótsze i wszystkie szablony. Opracowana struktura danych pozwala na znalezienie najkrótszego szablonu na przedziale w czasie $\mathcal{O}(\log n \log \log n)$ lub wszystkich szablonów w czasie $\mathcal{O}(\log n (\log \log n)^2)$. Struktura danych potrzebuje $\mathcal{O}(n \log n)$ pamięci i $\mathcal{O}(n \log n)$ czasu obliczeń wstępnych. Dodatkowo, przedstawione są efektywne rozwiązania w modelu offline.

W pracy “String Covers of a Tree” analizowane jest naturalne uogólnienie szablonu, w którym pokrywamy nie słowo, a drzewo. Krawędzie są etykietowane literami. Etykietą ścieżki jest słowo będące sekwencją liter na przechodzonych krawędziach. Słowo nazywamy szablonem drzewa, jeśli ścieżki etykietowane tym słowem pokrywają całe drzewo. Dla skierowanego drzewa przedstawiamy algorytm znajdujący wszystkie szablony w czasie $\mathcal{O}(n \log n / \log \log n)$, a w przypadku nieskierowanym w czasie i pamięci $\mathcal{O}(n^2)$. Dodatkowo, w przypadku nieskierowanym przedstawiamy alternatywny algorytm używający tylko $\mathcal{O}(n)$ pamięci, lecz działający w czasie $\mathcal{O}(n^2 \log n)$.

W drugiej części rozprawy przedstawiamy asymptotycznie efektywne rozwiązania do praktycznych problemów.

W pracy “Efficient Computation of Sequence Mappability” rozważany jest problem mapowalności. Musimy w nim policzyć dla każdego indeksu i , będącego początkiem pod słowa długości m , na ilu innych indeksach znajduje się pasujące pod słowo tej samej długości, dopuszczając do k niedopasowań. Przedstawiamy kilka wyników, z których głównym jest algorytm działający w czasie $\mathcal{O}(n \min\{\log^k n, m^k\})$ i pamięci $\mathcal{O}(n)$ dla ustalonego k . Dodatkowo, pokazujemy, że nie można rozwiązać problemu mapowalności w ściśle subkwadratowym czasie, chyba że hipoteza Strong Exponential Time jest nieprawdziwa.

W pracy “Circular Pattern Matching with k Mismatches” analizujemy wariant wyszukiwania wzorca, w którym wzorec może być dowolnie przesuwany cyklicznie i dopuszczamy do k niedopasowań. Przedstawiamy prosty algorytm działający w czasie $\mathcal{O}(nk)$ i pamięci $\mathcal{O}(m)$ i bardziej skomplikowany algorytm $\mathcal{O}(n+nk^4/m)$, działający również w pamięci $\mathcal{O}(m)$.

Wyniki często bazują na analizie okresowości i użyciu zarówno klasycznych narzędzi stringologicznych, jak i najnowocześniejszych, zaawansowanych struktur danych.

Tytuł rozprawy w języku polskim: Dokładne szablony i wyszukiwanie wzorca z dozwolonymi niedopasowaniami

Słowa kluczowe: algorytmy, struktury danych, słowo, okresowość, quasi-okresowość, szablon, wyszukiwanie wzorca w tekście z niedopasowaniami, odległość Hamminga, mapowalność

Contents

1	Introduction	5
1.1	Preface	5
1.2	Covers	6
1.3	Cyclicity	7
1.4	Pattern matching with mismatches	7
1.5	Computational model	8
1.6	String notations	9
2	Covers	10
2.1	Efficient Computation of 2-Covers of a String	10
2.2	Shortest Covers of All Cyclic Shifts of a String	12
2.3	Internal Quasiperiod Queries	13
2.4	String Covers of a Tree	14
3	Pattern matching with mismatches	16
3.1	Efficient Computation of Sequence Mappability	16
3.2	Circular Pattern Matching with k Mismatches	19
4	Final words	21
A	Efficient Computation of 2-Covers of a String	31

B Shortest Covers of All Cyclic Shifts of a String	48
C Internal Quasiperiod Queries	60
D String Covers of a Tree	76
E Efficient Computation of Sequence Mappability	91
F Circular Pattern Matching with k Mismatches	114

Chapter 1

Introduction

1.1 Preface

One of the most fundamental ways of representing data is encoding them into text. Text, or string, is a sequence of characters from some fixed alphabet. Consecutive parts of text are known as substrings or fragments.

Algorithms on strings are the subject of stringology. In my thesis I analyze strings under two perspectives – pattern matching and identifying regularities. While it might not be immediately clear, they are closely associated with each other.

To know whether a string is regular, intuitively sounds useful. But what is more surprising, knowledge that a text is **not** regular can be often exploited as well. Let us show some specific examples.

First, let us define the key concept of this thesis: *periodicity*. A text is called periodic when it is generated by repeated concatenation of a shorter word, a *fragment*, and then possibly trimmed on ends. If a text is periodic, this means that it is very regular, i.e. it can be stored by just its period and a few numbers describing its generation. On the other hand, when we know that a fragment of a text is non-periodic, it can be exploited as well. For example, occurrences of a non-periodic fragment cannot overlap too much, which gives us an upper bound on number of occurrences of such fragment. If a text is of length n and its non-periodic fragment is of length m , then the fragment might occur on at most $\mathcal{O}(n/m)$ positions in the text.

Pattern matching in its most basic form has to answer whether a string is a fragment of a text. Since the first linear algorithm (Morris Jr. and Pratt, 1970), many new variants of the original problem have emerged. One can allow for a fragment to occur in the text inexactly, allowing for small deviations under Hamming or edit distance or some other measure we can think of. Equality measure can go even further, e.g. allowing circular rotations of a pattern. Practical applications and imagination grant us interesting scenarios awaiting to be solved.

Pattern matching and identifying regularities are practically inseparable. For example, (Kociumaka, Radoszewski, et al., 2015) present a solution to the following problem:

Problem 1.1.1: Internal pattern matching

Input:

- T – text of length n

Queries:

- Input:
 - Indices of fragments $T[i_1..i_2], T[j_1..j_2]$
- Output:
 - Representation of all occurrences of $T[i_1..i_2]$ in $T[j_1..j_2]$

The paper presents a data structure which answers those kinds of queries in constant time if fragments are of similar length, i.e. when $\frac{|T[j_1..j_2]|}{|T[i_1..i_2]|}$ is $\mathcal{O}(1)$. The authors' solution heavily relies on analyzing maximal periodic fragments of the text, also known as *runs*. The number of such fragments in a text is at most linear and the sum of their exponents is at most linear as well (Kolpakov and Kucherov, 1999).

In my thesis I have taken interest in two subjects: variants of covers and pattern matching with mismatches. Additionally, cyclic shifts of patterns or texts are often analyzed in my work.

1.2 Covers

In the preface, I have introduced the notion of a period; *cover* is its natural evolution. While periodicity is generated only by concatenation, covers allow superpositions as well. Therefore, a fragment is a cover of a text if all occurrences of the fragment cover the entire text. Different variants of regularity are shown in Figure 1.1.

The first linear time algorithm finding the shortest cover was introduced in (Apostolico et al., 1991). Several years later, in (Moore and Smyth, 1994a,b, 1995) optimal algorithms identifying all covers were given.

Over the years, several variants of covers were considered. The most popular variant is *seed*, which allows for occurrences of the fragment to go beyond

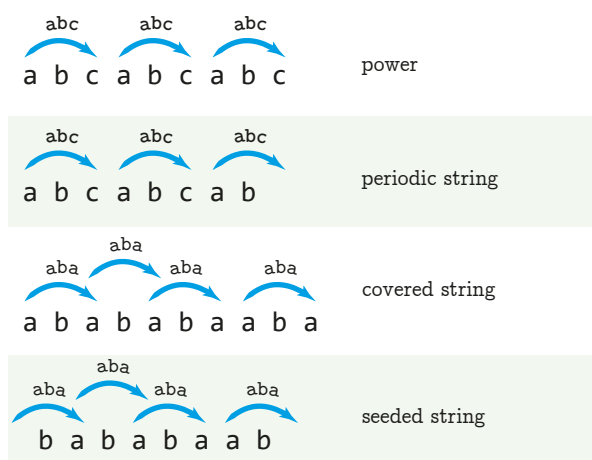


Figure 1.1: Different variants of string regularity.

the boundaries of the text. Therefore, a cover of the text is a seed of a substring of the text. It should **not** be confused with *spaced-seed*, which is a pattern in which some fixed positions are allowed to be wild-cards. Seeds were introduced in (Iliopoulos, Moore, et al., 1996), together with an algorithm that finds their representation in $\mathcal{O}(n \log n)$ time. It was not until many years later when a linear time algorithm was found (Kociumaka, Kubica, et al., 2012, 2020).

Later, many other variants of covers emerged. They were considered:

- in approximate variants:
(Amir, Levy, Lewenstein, et al., 2019; Amir, Levy, Lubin, et al., 2019; Christodoulakis et al., 2005; Flouri et al., 2013; Kedzierski and Radoszewski, 2020; Kociumaka, Pissis, et al., 2015, 2018)
- in different models of computation:
(Ben-Amram et al., 1994; Breslauer, 1994; Gawrychowski, Radoszewski, et al., 2019)
- in non-standard stringology:
(Alatabbi et al., 2016; Barton et al., 2020; Crochemore, Iliopoulos, Kociumaka, et al., 2017; Iliopoulos, Makris, et al., 2006; Iliopoulos, Mohamed, et al., 2003; Kikuchi et al., 2020)
- in 2-dimensional texts:
(Charalampopoulos, Radoszewski, et al., 2021; Crochemore, Iliopoulos, and Korda, 1998; Popa and Tanasescu, 2019)

Lately, new results were focused on approximate variants of covers. In my thesis I have focused on exact variants instead.

1.3 Cyclicity

The original pattern matching assumes that the ends of the pattern are somehow important – they mark the beginning and ending of a word. It turns out that sometimes, in practical uses, we would like to ignore that. Usually, this happens when we can freely rotate the object. Therefore, we should pick an equality measure which would reflect that. For example, if we want to check polygons for equality, a natural textual representation would consist of a sequence of angles and lengths of sides. Other motivations for considering cyclic patterns can be found in (Iliopoulos, Pissis, et al., 2017).

1.4 Pattern matching with mismatches

In practical uses, to succeed is often to find something even remotely similar to the pattern. It remains to define this similarity. In string algorithms there are several prevalent measures of similarity of two texts T_1, T_2 :

- Hamming distance: $\sum_i \llbracket T_1[i] \neq T_2[i] \rrbracket$

- L_1 distance (Manhattan): $\sum_i |T_1[i] - T_2[i]|$,
- edit distance (Levenshtein's): minimal number of replacements, insertions, deletions to change T_1 into T_2

In my thesis I focus on Hamming distance, which is the number of positions on which two strings do not match. This measure is relatively inexpensive computationally and sufficiently describes some phenomena from the real world (e.g. mutations or bit flips during transmission).

Efficient algorithms that find Hamming distance between pattern P of length m and all fragments of length m of text T have been researched for over 30 years. A variant of this problem adds a limit k on the number of mismatches, in which we seek only such fragments which are sufficiently matching to P with at most k mismatches. Over the years, the following exact algorithms were proposed (time complexity is given):

- $\mathcal{O}(n\sqrt{m \log m})$ (Abrahamson, 1987), independently (Kosaraju, 1987),
- $\mathcal{O}(nk)$ (Landau and Vishkin, 1986),
- $\mathcal{O}(n\sqrt{k \log k})$ (Amir, Lewenstein, et al., 2004),
 $\mathcal{O}((n + (nk^3)/m) \log k)$ (Amir, Lewenstein, et al., 2004),
- $\mathcal{O}((n/m)(k^2 \log k) + npolylogn)$ (Clifford, Fontaine, et al., 2016),
- $\mathcal{O}((n/m)(m + k\sqrt{m})polylogn)$ (Gawrychowski and Uznański, 2018),
- $\mathcal{O}((n/m)(m + k^2 \log \log k))$ (Charalampopoulos, Kociumaka, and Wellnitz, 2020),
- $\mathcal{O}((n/m)(m + \min(k\sqrt{m \log m}, k^2)))$ (Chan et al., 2020).

Other variants were also considered in:

- compressed representations of text (Bille et al., 2009; Bringmann et al., 2019; Charalampopoulos, Kociumaka, and Wellnitz, 2020; Gawrychowski and Straszak, 2013; Tiskin, 2014),
- distributed model (Galil and Giancarlo, 1987),
- streaming model (Clifford, Fontaine, et al., 2016; Clifford, Kociumaka, et al., 2019; B. Porat and E. Porat, 2009),
- parameterized model (Hazay et al., 2007),
- order-preserving model (Gawrychowski and Uznański, 2016).

1.5 Computational model

In all my papers, we assume word RAM model. In this model, memory consists of M cells which store machine words, i.e. W -bit numbers. Cells are addressed with consecutive numbers from 0 to $M - 1$. The

model allows for random access in constant time. Whenever memory is mentioned, it is measured by the number of required memory cells.

Strings are stored as an array. Each letter is stored in a separate memory cell.

1.6 String notations

- String is synonymous to word.
- Strings are numbered from 1 to n , where n is the length of a string.
- $T[i]$ denotes the i -th letter of T .
- $a \cdot b$ denotes concatenation of a and b .
- $T[i..j]$ denotes the substring of T from index i to index j inclusive.
- $T[i..j)$ denotes the substring of T from index i to index $j - 1$ inclusive.

Chapter 2

Covers

2.1 Efficient Computation of 2-Covers of a String

Problem 2.1.1: Finding 2-covers

Input:

- T – text of length n

Output:

- A representation of all pairs of substrings of equal length $S_1 = T[i..i + m), S_2 = T[j..j + m)$, such that each position i in T belongs to an occurrence of S_1 and/or S_2 in T .

In the case of standard covers, we assume that the text has been generated using concatenations or superpositions of a single word. In the case of 2-covers we would like to test whether text can be generated in this way by using two words of the same length.

Just from the problem formulation, one can guess that the problem might be harder than the original variant. Intuitively, to find all covers of a text, one must check only $\mathcal{O}(n)$ of similar candidates, i.e. all prefixes. That is not the case in 2-covers, as a text of length n may have as many as $\Theta(n^2)$ different non-trivial 2-covers (a simple example $T = a^m b a^m b a^m$ was shown in (Czajka and Radoszewski, 2021)).

The main algorithm from our paper (Radoszewski and Straszyński, 2020) finds a representation of all 2-covers in $\mathcal{O}(n \log n \log \log n)$ expected time. From this representation one can retrieve all 2-covers in $\mathcal{O}(n \log n \log \log n + |\text{output}|)$ time, where $|\text{output}|$ is the number of all 2-covers. In particular, this representation allows to find an example 2-cover for each length (if it exists) in $\mathcal{O}(n \log n \log \log n)$ total time. Despite seemingly increased difficulty by an order of magnitude we managed to stay in nearly linear world.

The problem was divided into two cases (see fig. 2.1):

- finding prefix-suffix 2-covers, where one word from the 2-cover is a prefix and the other is a suffix of the text,
- finding border 2-covers, where one word from the 2-cover is both a prefix and a suffix.

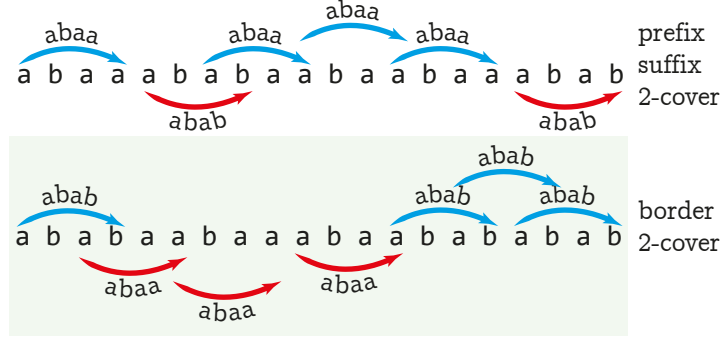


Figure 2.1: Two texts illustrating two types of 2-covers.

The first case is simpler, but still non-trivial. One word of the 2-cover must be a prefix, and the other a suffix. Therefore, a fixed length automatically determines a single pair of candidates for a 2-cover. The algorithm maintains the covered area by storing a set of linear functions. Ultimately this part takes $\mathcal{O}(n \log \log n)$ time.

The second case is harder as there can be as many as $\mathcal{O}(n^2)$ candidates. Therefore, we had to find an efficient representation. The solution is based on an analysis of periodicity and internal pattern matching queries.

In the algorithm we iterate over the prefixes which we consider to be a possible part of some 2-cover. Using internal pattern matching queries, we check for the parts of the text covered by occurrences of this prefix. There is a well known fact in stringology (e.g. (Kociumaka, Radoszewski, et al., 2015)) that whenever we have a text of length n and its fragment of length L , then at least one of these conditions must hold true:

$$\begin{cases} \text{the fragment has } \mathcal{O}(\frac{n}{L}) \text{ occurrences} \\ \text{the fragment is periodic and has } \mathcal{O}(\frac{n}{L}) \text{ arithmetic sequences of occurrences.} \end{cases}$$

Therefore, if we fix the length of the prefix to be L , then the uncovered area consists of $\mathcal{O}(\frac{n}{L})$ gaps of consecutive positions, which must be covered by the second word of 2-cover. From these gaps we deduce a set of restrictions for the second word, equivalent with covering the remaining text. We solve this system of restrictions in $\mathcal{O}(\frac{n}{L \log \log n})$ time, which ultimately leads to $\mathcal{O}(n \log n \log \log n)$ time for the whole algorithm.

All presented algorithms in the paper require a linear amount of space.

Contribution

I am the author of:

- algorithm finding prefix-suffix 2-covers,

- algorithm solving the system of restrictions in border 2-covers (in the paper: *Positioned Cover*).

I have contributed to the ultimate algorithm finding border 2-covers by making combinatorial observations and finding mistakes in previous incorrect solutions. I have also written most of the introduction to the paper and the section on prefix-suffix 2-covers. Additionally, I have made some figures.

2.2 Shortest Covers of All Cyclic Shifts of a String

Problem 2.2.1: Shortest covers of all cyclic shifts of a string

Input:

- T – text of length n ,

Output:

- For each x , the shortest cover of $CyclicShift_x(T)$, where

$$CyclicShift_x\left(T[1..n]\right) := T(n-x..n) \cdot T[1..n-x]$$

where $A \cdot B$ stands for concatenation of A and B .

In our paper (Crochemore, Iliopoulos, Radoszewski, et al., 2020b, 2021) we solve the problem in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space. It is an interesting combinatorial problem. Specifically, it can be solved in a more involved way with our later paper (Crochemore, Iliopoulos, Radoszewski, et al., 2020a) about internal quasiperiod queries in linear time.

A representation of all seeds, being a set of $\mathcal{O}(n)$ disjoint paths on the suffix tree, can be found in $\mathcal{O}(n)$ time (Kociumaka, Kubica, et al., 2020). From (Iliopoulos, Moore, et al., 1996) we know that covers of a cyclic word S are seeds of S^2 . Therefore, one can find the shortest cover of cyclic word in linear time.

The problem is solved by string combinatorics and two observations:

- A cover of a cyclic shift of T must have its square occur in T^3 .
- A cover of a cyclic shift of T must be a seed of T^3 .

The main combinatorial lemma shows that if a fragment of T^3 is a square with its middle at the right place and its half is a seed of T^3 , then it is a necessary and sufficient condition to being a cover of the appropriate cyclic shift of T . The remaining part of the algorithm uses stringological tools and periodicity analysis to test for those two conditions.

Additionally, we show an analysis of the shortest covers of all cyclic shifts of Fibonacci words. This family of words has the following recursive definition:

- $Fib_0 = b$,
- $Fib_1 = a$,
- $Fib_k = Fib_{k-1} \cdot Fib_{k-2}$ for $k \geq 2$.

Let $CyCoSet(T)$ (abbreviating *Cyclic Cover Set*) be the set of lengths of the shortest covers of all cyclic shifts of T . In our paper we show a precise description of CyCoSets of Fibonacci words.

Contribution

I am the author of the first solution $\mathcal{O}(n \log^2 n)$ and the Lemma 3, i.e. the main combinatorial lemma, which was written down by Wiktor Zuba. I am also a co-author of the final $\mathcal{O}(n \log n)$ algorithm. I have produced some of the figures. Apart from that, I took part in discussions of the team and proofreading of the paper.

2.3 Internal Quasiperiod Queries

Problem 2.3.1: Internal quasiperiod queries

Input:

- T – text of length n

Queries:

- Input:
 $1 \leq i \leq j \leq n$ – queried interval
- Output:
The shortest cover of fragment $T[i..j]$, or
Compact representation of all covers of fragment $T[i..j]$.

The data structure described in our paper (Crochemore, Iliopoulos, Radoszewski, et al., 2020a) requires $\mathcal{O}(n \log n)$ time for preprocessing and $\mathcal{O}(n \log n)$ memory. It can answer queries for:

- the shortest cover of an interval in $\mathcal{O}(\log n \log \log n)$ time,
- all covers of an interval in $\mathcal{O}(\log n (\log \log n)^2)$ time.

Additionally, in the offline model, i.e. when we do not have to answer the queries immediately, we can service m queries for:

- the shortest cover of an interval in $\mathcal{O}((n + m) \log n)$ time,
- all covers of an interval in $\mathcal{O}((n + m) \log n \log \log n)$ time,

using linear space.

In our paper we create a static segment tree, in which each of the nodes corresponds to a fragment of the text of length being a power of two, also known as base interval. Each node stores information about seeds of the corresponding fragment. To answer a query for an interval, first we decompose it into $\mathcal{O}(\log n)$ base intervals. Next, we retrieve from each base interval the information about its seeds. Then, we merge this information to find seeds of the entire queried interval. Finally we check for border conditions to filter out seeds which are not covers.

Contribution

I have improved the algorithm for finding the shortest cover of the interval by one logarithmic factor by efficient segment tree traversal. I have also participated in team discussions and proofreading.

2.4 String Covers of a Tree

Problem 2.4.1: All covers of a directed tree

Input:

- Rooted tree with n nodes:
 - each edge is labeled with a letter,
 - each edge is directed from node closer to the root to the node further from the root.

Output:

- All words S such that one can cover all edges by paths corresponding to that word.

The problem is solved in our paper (Radoszewski, Rytter, et al., 2021) in $\mathcal{O}(n \log n / \log \log n)$ time. The solution is an adaptation of algorithm for finding all covers in text in linear time (Moore and Smyth, 1995). First, we notice that if we fix an arbitrary leaf, then some occurrence of the cover must end at that leaf. Therefore, we can narrow our candidates to $\mathcal{O}(n)$ words corresponding to suffixes of word from root to that leaf. Iterating over suffixes from the shortest to the longest, we maintain a data structure which allows to find the maximum of all distances between vertically neighbouring occurrences. If that distance is smaller than or equal to the currently analyzed length, then the entire tree is covered.

Apart from that, we have considered an undirected variant (cf. fig. 2.2):

Problem 2.4.2: All covers of an undirected tree

Input:

- Tree with n nodes:
 - each edge is labeled with a letter.

Output:

- All words S such that one can cover all edges by paths corresponding to that word.

In our paper there are two algorithms, solving the above problem, with the following complexities:

- $\mathcal{O}(n^2)$ time and $\mathcal{O}(n^2)$ space,
- $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space.

The quadratic algorithm efficiently groups the paths by corresponding string-labels. After arbitrarily rooting our tree, each path might be expressed as a sum of two vertical paths. After preprocessing information about all paths, we check each of $\mathcal{O}(n)$ candidates in linear time.

In algorithm $\mathcal{O}(n^2 \log n)$ we test each of the candidates in $\mathcal{O}(n \log n)$ time using centroid decomposition. After fixing a candidate, we find a tree's centroid and we analyze only paths which pass through the centroid. Next, we efficiently process paths with matching label, mark appropriate edges as covered, remove the centroid from the tree and finally we do it recursively for disjoint components. This way, we will find in $\mathcal{O}(n \log n)$ time all edges of the tree which are covered by that candidate.

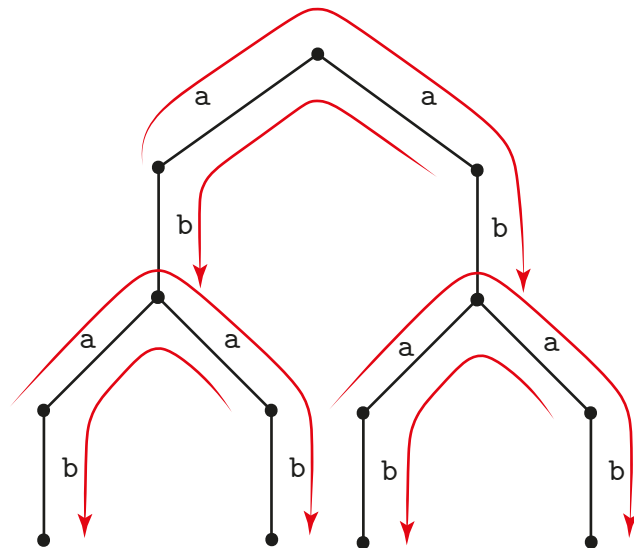


Figure 2.2: Paths corresponding to string aab cover the entire tree.

Contribution

I am the author of the $\mathcal{O}(n^2)$ time algorithm. I am also the author of the ultimate algorithm for testing candidates in $\mathcal{O}(n^2 \log n)$ time. I have also participated in team discussions and proofreading.

Chapter 3

Pattern matching with mismatches

3.1 Efficient Computation of Sequence Mappability

Problem 3.1.1: Sequence (k, m) -mappability

Input:

- T – text of length n ,
- k – maximum considered Hamming distance,
- m – length of analyzed fragments.

Output:

- Compute the function: $f(i) = \sum_{j \neq i} \left[\text{Hamming}(T[i..i+m], T[j..j+m]) \leq k \right]$

Our paper (Alzamel et al., 2018; Charalampopoulos, Iliopoulos, et al., 2022) solves the problem of (k, m) -mappability in $\mathcal{O}(n \cdot \min\{m^k, \log^k n\})$ time and $\mathcal{O}(n)$ space, assuming fixed k . Additionally, there are some auxiliary results:

- proof that for every $k, m = \Theta(\log n)$ one cannot solve (k, m) -mappability in strongly subquadratic time, unless Strong Exponential Time Hypothesis fails.
- $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ space algorithm finding (k, m) -mappability for each k or for each m .

The problem has stemmed from real problem in bioinformatics, processing of mapping genome. The goal of sequencing a genome is to construct a genome of some specimen. In this scenario we have:

- a fully constructed reference genome,

- random segments of the genome to be sequenced, all of fixed length m , without information from where they come from.

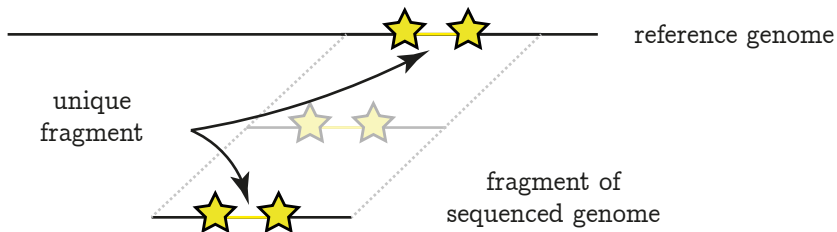


Figure 3.1: Identifying unique fragments of a genome helps to match sequenced genome fragments at the right position.

Firstly, segments might overlap, which helps to pair matching fragments of the genome. Secondly, in genome one can find unique fragments which are not present anywhere else (fig. 3.1). If the sampled segment contains such a unique fragment, then it can be easily matched at an appropriate position. (k, m) -mappability is nothing but simply identifying unique fragments. Mappability for such fragments will be equal to 0. It remains to explain why we need to allow for k mismatches; it is because segment reads might contain errors or mutations.

The process of genome mapping has been subsequently improved by various techniques and tools (Fonseca et al., 2012a). Problem of (k, m) -mappability was introduced for the first time in (Derrien et al., 2012) (see also (Antoniou et al., 2009)) together with some heuristic solutions. Before our paper there has been no solution which computed exact mappability under any asymptotic time guarantees for $k > 1$.

From algorithmics perspective, our paper is valuable because its the first which presents a technique of counting a Hamming-distance-based function for all pairs of elements from some set of strings. The main result is based on the algorithm of finding the longest common substring with k mismatches allowed (Charalampopoulos, Crochemore, et al., 2018) (see also (Thankachan et al., 2016)), which in turn it is based on k -errata trees from (Cole et al., 2004).

The main result was achieved by applying combinatorial observations and techniques:

- inclusion-exclusion principle to count exact number of matching pairs,
- smaller-half principle when merging subtrees to simulate mismatches,

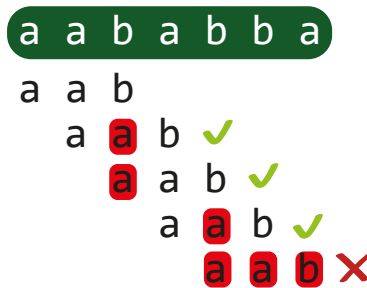


Figure 3.2: $T = aababba$, $m = 3$, $k = 1$. Mappability of the fragment at index $i = 1$ is equal to 3 – the number of matching fragments with at most one mismatch.

- emulating suffix tree using finger-search trees to perform needed operations on an ordered set of substrings,
- standard stringology tools such as suffix array.

The paper has been extended to the journal version (Charalampopoulos, Iliopoulos, et al., 2022). There, we have introduced the following additional problem:

Problem 3.1.2: All matching pairs with at most k mismatches

Input:

- R – set of r strings of length m ,
- k – maximum number of mismatches.

Output:

- All pairs of strings $S_1, S_2 \in R$ such that $\text{Hamming}(S_1, S_2) \leq k$.

In our paper we present an algorithm working in $\mathcal{O}(rm + r \log r)$ space and $\mathcal{O}(rm + r \log^{k+1} r + |\text{output}| \log r)$ time, where $|\text{output}|$ is the number of output pairs for fixed k .

The problem is motivated by finding evolutionary relations between different species or taxons. Some techniques in this field require to compute distance between all pairs of sequences representing analyzed species/taxons. Depending on the specific application and assumed model of evolution, frequently there is no need to compute the entire distance matrix (e.g. (Fonseca et al., 2012b), (Crochemore, Francisco, et al., 2017)). Therefore, in this step we are only interested in such pairs for which the Hamming distance doesn't exceed some fixed threshold.

Before this paper, there have been already developed some efficient algorithms but under average-case model with limits on value k and assumptions on elements of set R (Crochemore, Francisco, et al., 2017; Gog and Venturini, 2016; Manber and Myers, 1993).

Contribution

I am the author of:

- the first version of the algorithm for (k, m) -mappability (the main result),
- $\mathcal{O}(n^2)$ time algorithm finding answers for all k ,
- algorithm finding all pairs with at most k mismatches.

Additionally I have produced parts respective to those algorithms (including the main result). Additionally, I have produced the figure 3.1 from (Charalampopoulos, Iliopoulos, et al., 2022). I have participated in team's discussions and proofreading.

3.2 Circular Pattern Matching with k Mismatches

Problem 3.2.1: Circular pattern matching with k mismatches

Input:

- T – text of length n ,
- P – pattern of length m ,
- k – maximum number of mismatches.

Output:

- All indices i such that for some x

$$\text{Hamming}\left(\text{CyclicShift}_x\left(T[i..i+m]\right), P\right) \leq k$$

where

$$\text{CyclicShift}_x\left(T[1..n]\right) := T[n-x+1..n] \cdot T[1..n-x]$$

(see fig. 3.3).

All algorithms in our paper (Charalampopoulos, Kociumaka, Pissis, Radoszewski, Rytter, et al., 2019, 2021) require $\mathcal{O}(m)$ space. Our work has two main results:

- $\mathcal{O}(nk)$ time algorithm,
- $\mathcal{O}\left(n + \frac{n}{m}k^4\right)$ time algorithm (journal version improves upon the conference version's $\mathcal{O}\left(n + \frac{n}{m}k^5\right)$ time algorithm).

First, we simplify the problem to the scenario in which $n \leq 2m$ using a standard trick in which we divide the text into $\mathcal{O}(n/m)$ overlapping parts (windows). Additionally we need to use perfect hashing on alphabet to construct necessary stringology tools in linear time. Next, we solve the problem on each part independently.

The $\mathcal{O}(nk)$ time algorithm is simple and practical. Let us notice that a cyclic shift of P is a concatenation of its suffix and corresponding prefix. The position in T where prefix of P begins is called an *anchor*. We test every possible $\mathcal{O}(n)$ positions for anchors and for each of them we compute what is the longest matching prefix for each number of allowed mismatches from 1 to k . Similarly, we find the longest matching suffixes for each number of mismatches at the index immediately before the anchor. This procedure is performed using kangaroo-jumps technique (Galil and Giancarlo, 1987; Landau and Vishkin, 1986).

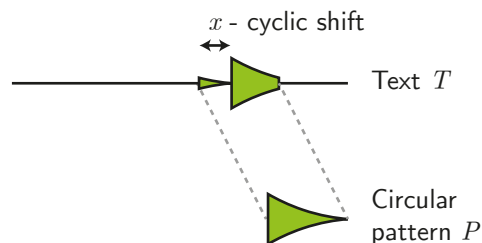


Figure 3.3: Matching a circular pattern.

The other main result – $\mathcal{O}(n + \frac{n}{m}k^4)$ time algorithm is based on (Bringmann et al., 2019). In the algorithm we divide the pattern into $\mathcal{O}(k)$ parts and we consider $\mathcal{O}(k)$ of them such that some of them must have occurred in the text exactly, without any mismatches. Therefore, if our considered part is S , then:

- if S is non-periodic, it occurs only $\mathcal{O}(k)$ times in the considered window and each occurrence corresponds to two possible anchors at maximum
- if S is periodic, then we need to consider $\mathcal{O}(k)$ arithmetic sequences of occurrences. Each of the group we process with technique “few mismatches or almost periodicity” from (Bringmann et al., 2019).

Contribution

I am the author of the $\mathcal{O}(nk)$ time algorithm. Additionally, I have written down the section corresponding to this algorithm in our paper. Also, I have presented and summarized the paper (Bringmann et al., 2019) to our research team. Our solution for the periodic case was strongly inspired by this paper. Finally, I have participated in team discussions and proofreading.

Chapter 4

Final words

Presented solutions use a wide array of stringological tools, from classical string algorithms to sophisticated data structures and string combinatorics.

In my opinion, the most characteristic trick for the most of my papers is periodicity analysis. Study of periodic and non-periodic cases turned out to be quite helpful.

Exploiting non-periodicity

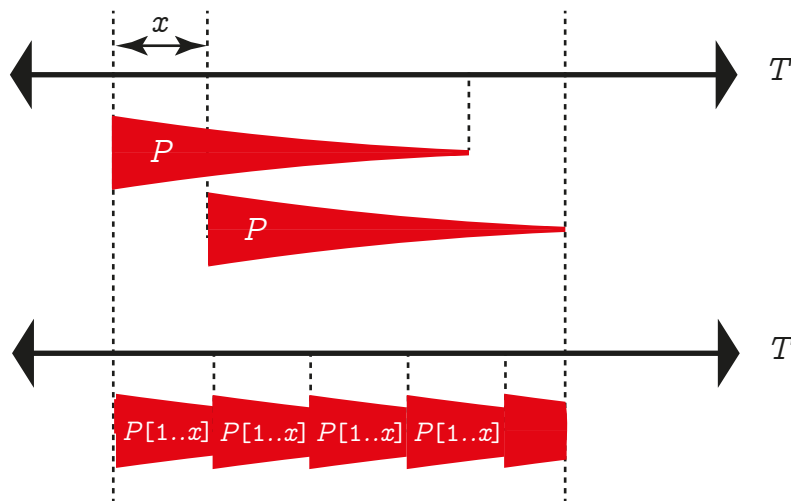


Figure 4.1: If occurrences of fragment P in T are overlapping too much, then P must be periodic. The last part might be trimmed, but it must be a prefix of P .

Non-periodicity is helpful in this regard that occurrences of a non-periodic string cannot overlap too much. If two occurrences of the same fragment overlap on at least half of the length, it means that the fragment has a regular structure (see fig. 4.1).

Sometimes the non-periodic case is the only case that is of interest to us. Let us notice that a cover of a cover of a text is also a cover of the text. Granted that a periodic cover always has a non-periodic cover itself, we can only consider non-periodic candidates when searching for the shortest cover, e.g. in (Crochemore, Iliopoulos, Radoszewski, et al., 2020b).

Other times, we exploit the fact that there can be at most $\mathcal{O}(n/m)$ occurrences of a fragment of length m in text of length n (Charalampopoulos, Kociumaka, Pissis, Radoszewski, Rytter, et al., 2019, 2021).

Exploiting periodicity

While for a non-periodic fragment of length m the number of occurrences in text of length n is $\mathcal{O}(n/m)$, the number of occurrences for periodic fragments can be compressed to $\mathcal{O}(n/m)$ arithmetic sequences. Those arithmetic sequences can be deduced based on runs, maximal fragments of periodicity. Periodic fragments had to be considered in (Charalampopoulos, Kociumaka, Pissis, Radoszewski, Rytter, et al., 2019, 2021; Crochemore, Iliopoulos, Radoszewski, et al., 2020a, 2021; Radoszewski and Straszyński, 2020).

In the case of covers, a short periodic fragment covers (sometimes omitting ends) a long run, which was exploited in (Radoszewski and Straszyński, 2020)

Data structures

Internal pattern matching queries (Kociumaka, Radoszewski, et al., 2015) were heavily used in many of my papers. This data structure is used directly or indirectly in (Charalampopoulos, Kociumaka, Pissis, Radoszewski, Rytter, et al., 2019, 2021; Crochemore, Iliopoulos, Radoszewski, et al., 2020a,b; Radoszewski and Straszyński, 2020).

Suffix tree and its derivatives are fundamental data structures in stringology. What is interesting, in (Alzamel et al., 2018; Charalampopoulos, Iliopoulos, et al., 2022) we traverse a suffix tree without explicitly constructing it. Instead, we find edges on the run by using *finger-search* trees which are a variant of a balanced binary search tree. In (Crochemore, Iliopoulos, Radoszewski, et al., 2020a) we store the information about seeds as paths on the suffix tree. In (Radoszewski, Rytter, et al., 2021) we use the suffix tree directly. Algorithms from papers (Alzamel et al., 2018; Charalampopoulos, Iliopoulos, et al., 2022; Charalampopoulos, Kociumaka, Pissis, Radoszewski, Rytter, et al., 2019, 2021) use internal queries for the longest common prefix based on the suffix array.

To answer queries, in (Alzamel et al., 2018; Charalampopoulos, Iliopoulos, et al., 2022; Radoszewski, Rytter, et al., 2021) we have used divide-and-conquer technique of splitting the queried interval into base intervals.

Possible improvements

Some of the presented results have been already improved. In a recent preprint, my team managed to solve the problem of computing shortest covers of all cyclic shifts of a text from (Crochemore, Iliopoulos,

Radoszewski, et al., 2020b) in linear time. We also know how to compute prefix-suffix 2-covers in linear time; however, this does not improve the overall time complexity of computing 2-covers from (Radoszewski and Straszyński, 2020) because computing border 2-covers is the bottleneck.

It is possible that the paper about circular pattern matching can be improved by using state-of-the-art tools from the recent paper (Charalampopoulos, Kociumaka, and Wellnitz, 2020).

It is not known whether the time complexity of the main algorithm for computing sequence mappability (Charalampopoulos, Iliopoulos, et al., 2022) can be improved. On one hand there is some indirect connection with hardness of indexing with k mismatches (Cohen-Addad et al., 2019). On the other hand in a problem akin to mappability, finding the longest common substring with k mismatches, the complexity of an $\mathcal{O}(n \log^k n)$ time algorithm was very recently improved by a factor of $\sqrt{\log n}$ (Charalampopoulos, Kociumaka, Pissis, and Radoszewski, 2021).

With the aid of a recent optimal data structure for weighted ancestor queries on a suffix tree (Belazzougui et al., 2021), internal shortest cover queries can be answered without changing the technique from our paper (Crochemore, Iliopoulos, Radoszewski, et al., 2020a) in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n \log n)$ preprocessing. It is not clear if the $\mathcal{O}(n \log n)$ space of the data structure can be improved upon. For internal shortest period queries, the best known query complexity is $\mathcal{O}(\log n)$ but the data structure uses only linear space (Kociumaka, Radoszewski, et al., 2015).

Bibliography

- Abrahamson, K. R. (1987). “Generalized String Matching”. *SIAM Journal on Computing*, 16(6), pp. 1039–1051. DOI: 10.1137/0216067 (cit. on p. 8).
- Alatabbi, A., Rahman, M. S., and Smyth, W. F. (2016). “Computing covers using prefix tables”. *Discrete Applied Mathematics*, 212, pp. 2–9. DOI: 10.1016/j.dam.2015.05.019 (cit. on p. 7).
- Alzamel, M., Charalampopoulos, P., Iliopoulos, C. S., Kociumaka, T., Pissis, S. P., Radoszewski, J., and Straszyński, J. (2018). “Efficient Computation of Sequence Mappability”. In: *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*. Ed. by T. Gagie, A. Moffat, G. Navarro, and E. Cuadros-Vargas. Vol. 11147. Lecture Notes in Computer Science. Springer, pp. 12–26. URL: https://doi.org/10.1007/978-3-030-00479-8_2 (cit. on pp. 16, 22).
- Amir, A., Levy, A., Lewenstein, M., Lubin, R., and Porat, B. (2019). “Can We Recover the Cover?” *Algorithmica*, 81(7), pp. 2857–2875. URL: <https://doi.org/10.1007/s00453-019-00559-8> (cit. on p. 7).
- Amir, A., Levy, A., Lubin, R., and Porat, E. (2019). “Approximate cover of strings”. *Theoretical Computer Science*, 793, pp. 59–69. URL: <https://doi.org/10.1016/j.tcs.2019.05.020> (cit. on p. 7).
- Amir, A., Lewenstein, M., and Porat, E. (2004). “Faster algorithms for string matching with k mismatches”. *Journal of Algorithms*, 50(2), pp. 257–275. DOI: 10.1016/S0196-6774(03)00097-X (cit. on p. 8).
- Antoniou, P., Daykin, J. W., Iliopoulos, C. S., Kourie, D., Mouchard, L., and Pissis, S. P. (2009). “Mapping uniquely occurring short sequences derived from high throughput technologies to a reference genome”. In: *9th International Conference on Information Technology and Applications in Biomedicine, ITAB 2009*. IEEE, pp. 1–4. URL: <https://doi.org/10.1109/itab.2009.5394394> (cit. on p. 17).
- Apostolico, A., Farach, M., and Iliopoulos, C. S. (1991). “Optimal Superprimitivity Testing for Strings”. *Information Processing Letters*, 39(1), pp. 17–20. URL: [https://doi.org/10.1016/0020-0190\(91\)90056-N](https://doi.org/10.1016/0020-0190(91)90056-N) (cit. on p. 6).
- Barton, C., Kociumaka, T., Liu, C., Pissis, S. P., and Radoszewski, J. (2020). “Indexing weighted sequences: Neat and efficient”. *Information and Computation*, 270, p. 104462. DOI: 10.1016/j.ic.2019.104462 (cit. on p. 7).
- Belazzougui, D., Kosolobov, D., Puglisi, S. J., and Raman, R. (2021). “Weighted Ancestors in Suffix Trees Revisited”. In: *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*. Ed. by P. Gawrychowski and T. Starikovskaya. Vol. 191. LIPIcs. Schloss Dagstuhl -

- Leibniz-Zentrum für Informatik, 8:1–8:15. URL: <https://doi.org/10.4230/LIPIcs.CPM.2021.8> (cit. on p. 23).
- Ben-Amram, A. M., Berkman, O., Iliopoulos, C. S., and Park, K. (1994). “The Subtree Max Gap Problem with Application to Parallel String Covering”. In: *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. Ed. by D. D. Sleator. ACM/SIAM, pp. 501–510. URL: <http://dl.acm.org/citation.cfm?id=314464.314633> (cit. on p. 7).
- Bille, P., Fagerberg, R., and Gørtz, I. L. (2009). “Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts”. *ACM Transactions on Algorithms*, 6(1), 3:1–3:14. DOI: 10.1145/1644015.1644018 (cit. on p. 8).
- Breslauer, D. (1994). “Testing String Superprimitivity in Parallel”. *Information Processing Letters*, 49(5), pp. 235–241. URL: [https://doi.org/10.1016/0020-0190\(94\)90060-4](https://doi.org/10.1016/0020-0190(94)90060-4) (cit. on p. 7).
- Bringmann, K., Wellnitz, P., and Künnemann, M. (2019). “Few Matches or Almost Periodicity: Faster Pattern Matching with Mismatches in Compressed Texts”. In: *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*. Ed. by T. M. Chan. SIAM, pp. 1126–1145. DOI: 10.1137/1.9781611975482.69 (cit. on pp. 8, 20).
- Chan, T. M., Golan, S., Kociumaka, T., Kopelowitz, T., and Porat, E. (2020). “Approximating text-to-pattern Hamming distances”. In: *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*. Ed. by K. Makarychev, Y. Makarychev, M. Tulsiani, G. Kamath, and J. Chuzhoy. ACM, pp. 643–656. URL: <https://doi.org/10.1145/3357713.3384266> (cit. on p. 8).
- Charalampopoulos, P., Crochemore, M., Iliopoulos, C. S., Kociumaka, T., Pissis, S. P., Radoszewski, J., Rytter, W., and Waleń, T. (2018). “Linear-Time Algorithm for Long LCF with k Mismatches”. In: *29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018*. Ed. by G. Navarro, D. Sankoff, and B. Zhu. Vol. 105. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 23:1–23:16. URL: <https://doi.org/10.4230/LIPIcs.CPM.2018.23> (cit. on p. 17).
- Charalampopoulos, P., Iliopoulos, C. S., Kociumaka, T., Pissis, S. P., Radoszewski, J., and Straszyński, J. (2022). “Efficient Computation of Sequence Mappability”. In: *Algorithmica*. DOI: 10.1007/s00453-022-00934-y (cit. on pp. 16, 18, 22, 23).
- Charalampopoulos, P., Kociumaka, T., Pissis, S. P., and Radoszewski, J. (2021). “Faster Algorithms for Longest Common Substring”. In: *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*. Ed. by P. Mutzel, R. Pagh, and G. Herman. Vol. 204. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:17. URL: <https://doi.org/10.4230/LIPIcs.ESA.2021.30> (cit. on p. 23).
- Charalampopoulos, P., Kociumaka, T., Pissis, S. P., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., and Zuba, W. (2019). “Circular Pattern Matching with k Mismatches”. In: *Fundamentals of Computation Theory - 22nd International Symposium, FCT 2019, Copenhagen, Denmark, August 12-14, 2019, Proceedings*. Ed. by L. A. Gasieniec, J. Jansson, and C. Levcopoulos. Vol. 11651. Lecture Notes in Computer Science. Springer, pp. 213–228. URL: https://doi.org/10.1007/978-3-030-25027-0_15 (cit. on pp. 19, 22).

- Charalampopoulos, P., Kociumaka, T., Pissis, S. P., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., and Zuba, W. (2021). “Circular pattern matching with k mismatches”. *Journal of Computer and System Sciences*, 115, pp. 73–85. URL: <https://doi.org/10.1016/j.jcss.2020.07.003> (cit. on pp. 19, 22).
- Charalampopoulos, P., Kociumaka, T., and Wellnitz, P. (2020). “Faster Approximate Pattern Matching: A Unified Approach”. In: *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*. Ed. by S. Irani. IEEE, pp. 978–989. URL: <https://doi.org/10.1109/FOCS46700.2020.00095> (cit. on pp. 8, 23).
- Charalampopoulos, P., Radoszewski, J., Rytter, W., Waleń, T., and Zuba, W. (2021). “Computing Covers of 2D-Strings”. In: *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*. Ed. by P. Gawrychowski and T. Starikovskaya. Vol. 191. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:20. URL: <https://doi.org/10.4230/LIPIcs.CPM.2021.12> (cit. on p. 7).
- Christodoulakis, M., Iliopoulos, C. S., Park, K., and Sim, J. S. (2005). “Approximate Seeds of Strings”. *Journal of Automata, Languages, and Combinatorics*, 10(5/6), pp. 609–626. URL: <https://doi.org/10.25596/jalcs-2005-609> (cit. on p. 7).
- Clifford, R., Fontaine, A., Porat, E., Sach, B., and Starikovskaya, T. (2016). “The k -mismatch problem revisited”. In: *27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*. Ed. by R. Krauthgamer. SIAM, pp. 2039–2052. DOI: 10.1137/1.9781611974331.ch142 (cit. on p. 8).
- Clifford, R., Kociumaka, T., and Porat, E. (2019). “The streaming k -mismatch problem”. In: *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*. Ed. by T. M. Chan. SIAM, pp. 1106–1125. DOI: 10.1137/1.9781611975482.68 (cit. on p. 8).
- Cohen-Addad, V., Feuilloley, L., and Starikovskaya, T. (2019). “Lower bounds for text indexing with mismatches and differences”. In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*. Ed. by T. M. Chan. SIAM, pp. 1146–1164. URL: <https://doi.org/10.1137/1.9781611975482.70> (cit. on p. 23).
- Cole, R., Gottlieb, L., and Lewenstein, M. (2004). “Dictionary matching and indexing with errors and don’t cares”. In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. Ed. by L. Babai. ACM, pp. 91–100. URL: <https://doi.org/10.1145/1007352.1007374> (cit. on p. 17).
- Crochemore, M., Francisco, A. P., Pissis, S. P., and Vaz, C. (2017). “Towards Distance-Based Phylogenetic Inference in Average-Case Linear-Time”. In: *17th International Workshop on Algorithms in Bioinformatics, WABI 2017*. Ed. by R. Schwartz and K. Reinert. Vol. 88. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 9:1–9:14. URL: <https://doi.org/10.4230/LIPIcs.WABI.2017.9> (cit. on p. 18).
- Crochemore, M., Iliopoulos, C. S., Kociumaka, T., Radoszewski, J., Rytter, W., and Waleń, T. (2017). “Covering problems for partial words and for indeterminate strings”. *Theoretical Computer Science*, 698, pp. 25–39. DOI: 10.1016/j.tcs.2017.05.026 (cit. on p. 7).
- Crochemore, M., Iliopoulos, C. S., and Korda, M. (1998). “Two-Dimensional Prefix String Matching and Covering on Square Matrices”. *Algorithmica*, 20(4), pp. 353–373. DOI: 10.1007/PL00009200 (cit. on p. 7).

- Crochemore, M., Iliopoulos, C. S., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., and Zuba, W. (2020a). “Internal Quasiperiod Queries”. In: *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*. Ed. by C. Boucher and S. V. Thankachan. Vol. 12303. Lecture Notes in Computer Science. Springer, pp. 60–75. URL: https://doi.org/10.1007/978-3-030-59212-7_5 (cit. on pp. 12, 13, 22, 23).
- (2020b). “Shortest Covers of All Cyclic Shifts of a String”. In: *WALCOM: Algorithms and Computation - 14th International Conference, WALCOM 2020, Singapore, March 31 - April 2, 2020, Proceedings*. Ed. by M. S. Rahman, K. Sadakane, and W. Sung. Vol. 12049. Lecture Notes in Computer Science. Springer, pp. 69–80. URL: https://doi.org/10.1007/978-3-030-39881-1_7 (cit. on pp. 12, 22).
- (2021). “Shortest covers of all cyclic shifts of a string”. *Theor. Comput. Sci.*, 866, pp. 70–81. URL: <https://doi.org/10.1016/j.tcs.2021.03.011> (cit. on pp. 12, 22).
- Czajka, P. and Radoszewski, J. (2021). “Experimental evaluation of algorithms for computing quasiperiods”. *Theoretical Computer Science*, 854, pp. 17–29. URL: <https://doi.org/10.1016/j.tcs.2020.11.033> (cit. on p. 10).
- Derrien, T., Estellé, J., Sola, S. M., Knowles, D. G., Raineri, E., Guigó, R., and Ribeca, P. (2012). “Fast Computation and Applications of Genome Mappability”. *PLoS ONE*, 7(1). Ed. by C. A. Ouzounis, e30377. URL: <https://doi.org/10.1371/journal.pone.0030377> (cit. on p. 17).
- Flouri, T., Iliopoulos, C. S., Kociumaka, T., Pissis, S. P., Puglisi, S. J., Smyth, W. F., and Tyczynski, W. (2013). “Enhanced string covering”. *Theoretical Computer Science*, 506, pp. 102–114. URL: <https://doi.org/10.1016/j.tcs.2013.08.013> (cit. on p. 7).
- Fonseca, N. A., Rung, J., Brazma, A., and Marioni, J. C. (2012a). “Tools for mapping high-throughput sequencing data”. *Bioinformatics*, 28(24), pp. 3169–3177. URL: <https://doi.org/10.1093/bioinformatics/bts605> (cit. on p. 17).
- (2012b). “Tools for mapping high-throughput sequencing data”. *Bioinformatics*, 28(24), pp. 3169–3177. URL: <https://doi.org/10.1093/bioinformatics/bts605> (cit. on p. 18).
- Galil, Z. and Giancarlo, R. (1987). “Parallel String Matching with k Mismatches”. *Theoretical Computer Science*, 51, pp. 341–348. DOI: 10.1016/0304-3975(87)90042-9 (cit. on pp. 8, 19).
- Gawrychowski, P., Radoszewski, J., and Starikovskaya, T. A. (2019). “Quasi-Periodicity in Streams”. In: *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019*. Ed. by N. Pisanti and S. P. Pissis. Vol. 128. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:14. DOI: 10.4230/LIPIcs.CPM.2019.22 (cit. on p. 7).
- Gawrychowski, P. and Straszak, D. (2013). “Beating $\mathcal{O}(nm)$ in Approximate LZW-Compressed Pattern Matching”. In: *Algorithms and Computation, ISAAC 2013*. Ed. by L. Cai, S. Cheng, and T. W. Lam. Vol. 8283. LNCS. Springer, pp. 78–88. DOI: 10.1007/978-3-642-45030-3_8 (cit. on p. 8).
- Gawrychowski, P. and Uznański, P. (2016). “Order-preserving pattern matching with k mismatches”. *Theoretical Computer Science*, 638, pp. 136–144. DOI: 10.1016/j.tcs.2015.08.022 (cit. on p. 8).
- (2018). “Towards Unified Approximate Pattern Matching for Hamming and L_1 Distance”. In: *Automata, Languages, and Programming, ICALP 2018*. Ed. by I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella. Vol. 107. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 62:1–62:13. DOI: 10.4230/LIPIcs.ICALP.2018.62 (cit. on p. 8).

- Gog, S. and Venturini, R. (2016). “Fast and Compact Hamming Distance Index”. In: *39th International ACM-SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2016*. Ed. by R. Perego, F. Sebastiani, J. A. Aslam, I. Ruthven, and J. Zobel. ACM, pp. 285–294. URL: <https://doi.org/10.1145/2911451.2911523> (cit. on p. 18).
- Hazay, C., Lewenstein, M., and Sokol, D. (2007). “Approximate parameterized matching”. *ACM Transactions on Algorithms*, 3(3), p. 29. DOI: 10.1145/1273340.1273345 (cit. on p. 8).
- Iliopoulos, C. S., Makris, C., Panagis, Y., Perdikuri, K., Theodoridis, E., and Tsakalidis, A. K. (2006). “The Weighted Suffix Tree: An Efficient Data Structure for Handling Molecular Weighted Sequences and its Applications”. *Fundamenta Informaticae*, 71(2-3), pp. 259–277. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-07> (cit. on p. 7).
- Iliopoulos, C. S., Mohamed, M., Mouchard, L., Perdikuri, K., Smyth, W. F., and Tsakalidis, A. K. (2003). “String Regularities with Don’t Cares”. *Nordic Journal on Computing*, 10(1), pp. 40–51 (cit. on p. 7).
- Iliopoulos, C. S., Moore, D. W. G., and Park, K. (1996). “Covering a String”. *Algorithmica*, 16(3), pp. 288–297. DOI: 10.1007/BF01955677 (cit. on pp. 7, 12).
- Iliopoulos, C. S., Pissis, S. P., and Rahman, M. S. (2017). “Searching and Indexing Circular Patterns”. In: *Algorithms for Next-Generation Sequencing Data, Techniques, Approaches, and Applications*. Ed. by M. Elloumi. Springer, pp. 77–90. DOI: 10.1007/978-3-319-59826-0_3 (cit. on p. 7).
- Kedzierski, A. and Radoszewski, J. (2020). “k-Approximate Quasiperiodicity under Hamming and Edit Distance”. In: *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*. Ed. by I. L. Gørtz and O. Weimann. Vol. 161. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:15. URL: <https://doi.org/10.4230/LIPIcs.CPM.2020.18> (cit. on p. 7).
- Kikuchi, N., Hendrian, D., Yoshinaka, R., and Shinohara, A. (2020). “Computing Covers Under Substring Consistent Equivalence Relations”. In: *String Processing and Information Retrieval*. Ed. by C. Boucher and S. V. Thankachan. Cham: Springer International Publishing, pp. 131–146 (cit. on p. 7).
- Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., and Waleń, T. (2012). “A linear time algorithm for seeds computation”. In: *23rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012*. Ed. by Y. Rabani. SIAM, pp. 1095–1112. DOI: 10.1137/1.9781611973099 (cit. on p. 7).
- (2020). “A Linear-Time Algorithm for Seeds Computation”. *ACM Transactions on Algorithms*, 16(2), Article 27. DOI: 10.1145/3386369 (cit. on pp. 7, 12).
- Kociumaka, T., Pissis, S. P., Radoszewski, J., Rytter, W., and Waleń, T. (2015). “Fast Algorithm for Partial Covers in Words”. *Algorithmica*, 73(1), pp. 217–233. URL: <https://doi.org/10.1007/s00453-014-9915-3> (cit. on p. 7).
- (2018). “Efficient algorithms for shortest partial seeds in words”. *Theoretical Computer Science*, 710, pp. 139–147. DOI: 10.1016/j.tcs.2016.11.035 (cit. on p. 7).
- Kociumaka, T., Radoszewski, J., Rytter, W., and Waleń, T. (2015). “Internal Pattern Matching Queries in a Text and Applications”. In: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*. Ed. by P. Indyk. SIAM, pp. 532–551. URL: <https://doi.org/10.1137/1.9781611973730.36> (cit. on pp. 6, 11, 22, 23).

- Kolpakov, R. M. and Kucherov, G. (1999). “Finding Maximal Repetitions in a Word in Linear Time”. In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, pp. 596–604. URL: <https://doi.org/10.1109/SFFCS.1999.814634> (cit. on p. 6).
- Kosaraju, S. (1987). “Efficient string matching”. Manuscript (cit. on p. 8).
- Landau, G. M. and Vishkin, U. (1986). “Efficient String Matching with k Mismatches”. *Theoretical Computer Science*, 43, pp. 239–249. DOI: 10.1016/0304-3975(86)90178-7 (cit. on pp. 8, 19).
- Manber, U. and Myers, E. W. (1993). “Suffix Arrays: A New Method for On-Line String Searches”. *SIAM Journal on Computing*, 22(5), pp. 935–948. URL: <https://doi.org/10.1137/0222058> (cit. on p. 18).
- Moore, D. W. G. and Smyth, W. F. (1994a). “An Optimal Algorithm to Compute all the Covers of a String”. *Information Processing Letters*, 50(5), pp. 239–246. URL: [https://doi.org/10.1016/0020-0190\(94\)00045-X](https://doi.org/10.1016/0020-0190(94)00045-X) (cit. on p. 6).
- (1994b). “Computing the Covers of a String in Linear Time”. In: *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA*. Ed. by D. D. Sleator. ACM/SIAM, pp. 511–515. URL: <http://dl.acm.org/citation.cfm?id=314464.314636> (cit. on p. 6).
- (1995). “A Correction to ‘An Optimal Algorithm to Compute all the Covers of a String’”. *Information Processing Letters*, 54(2), pp. 101–103. URL: [https://doi.org/10.1016/0020-0190\(94\)00235-Q](https://doi.org/10.1016/0020-0190(94)00235-Q) (cit. on pp. 6, 14).
- Morris Jr., J. H. and Pratt, V. R. (1970). “A linear pattern-matching algorithm”. *Technical report 40* (cit. on p. 5).
- Popa, A. and Tanasescu, A. (2019). “An Output-Sensitive Algorithm for the Minimization of 2-Dimensional String Covers”. In: *Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019*. Ed. by T. V. Gopal and J. Watada. Vol. 11436. Lecture Notes in Computer Science. Springer, pp. 536–549. DOI: 10.1007/978-3-030-14812-6_33 (cit. on p. 7).
- Porat, B. and Porat, E. (2009). “Exact and Approximate Pattern Matching in the Streaming Model”. In: *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009*. IEEE Computer Society, pp. 315–323. DOI: 10.1109/FOCS.2009.11 (cit. on p. 8).
- Radoszewski, J., Rytter, W., Straszynski, J., Waleń, T., and Zuba, W. (2021). “String Covers of a Tree”. In: *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*. Ed. by T. Lecroq and H. Touzet. Vol. 12944. Lecture Notes in Computer Science. Springer, pp. 68–82. URL: https://doi.org/10.1007/978-3-030-86692-1_7 (cit. on pp. 14, 22).
- Radoszewski, J. and Straszynski, J. (2020). “Efficient Computation of 2-Covers of a String”. In: *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*. Ed. by F. Grandoni, G. Herman, and P. Sanders. Vol. 173. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 77:1–77:17. URL: <https://doi.org/10.4230/LIPIcs.ESA.2020.77> (cit. on pp. 10, 22, 23).

- Thankachan, S. V., Apostolico, A., and Aluru, S. (2016). “A Provably Efficient Algorithm for the k -Mismatch Average Common Substring Problem”. *Journal of Computational Biology*, 23(6), pp. 472–482. URL: <https://doi.org/10.1089/cmb.2015.0235> (cit. on p. 17).
- Tiskin, A. (2014). “Threshold Approximate Matching in Grammar-Compressed Strings”. In: *Prague Stringology Conference 2014, PSC 2014*. Ed. by J. Holub and J. Zdárek. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, pp. 124–138. URL: <http://www.stringology.org/event/2014/p12.html> (cit. on p. 8).

Efficient Computation of 2-Covers of a String

Jakub Radoszewski 

Institute of Informatics, University of Warsaw, Poland
Samsung R&D Poland, Warsaw, Poland
jrad@mimuw.edu.pl

Juliusz Straszynski 

Institute of Informatics, University of Warsaw, Poland
jks@mimuw.edu.pl

Abstract

Quasiperiodicity is a generalization of periodicity that has been researched for almost 30 years. The notion of cover is the classic variant of quasiperiodicity. A cover of a text T is a string whose occurrences in T cover all positions of T . There are several algorithms computing covers of a text in linear time. In this paper we consider a natural extension of cover. For a text T , we call a pair of strings a 2-cover if they have the same length and their occurrences cover the text T . We give an algorithm that computes all 2-covers of a string of length n in $\mathcal{O}(n \log n \log \log n + \text{output})$ expected time or $\mathcal{O}(n \log n \log^2 \log n / \log \log \log n + \text{output})$ worst-case time, where output is the size of output.

If (X, Y) is a 2-cover of T , then either X is a prefix and Y is a suffix of T , in which case we call (X, Y) a ps-cover, or one of X, Y is a border (that is, both a prefix and a suffix) of T , and then we call (X, Y) a b-cover. A string of length n has up to n ps-covers; we show an algorithm that computes all of them in $\mathcal{O}(n \log \log n)$ expected time or $\mathcal{O}(n \log^2 \log n / \log \log \log n)$ worst-case time. A string of length n can have $\Theta(n^2)$ non-trivial b-covers; our algorithm can report one b-cover per length (if it exists) or all shortest b-covers in $\mathcal{O}(n \log n \log \log n)$ expected time or $\mathcal{O}(n \log n \log^2 \log n / \log \log \log n)$ worst-case time. All our algorithms use linear space.

The problem in scope can be generalized to $\lambda > 2$ equal-length strings, resulting in the notion of λ -cover. Cole et al. (2005) showed that the λ -cover problem is NP-complete. Our algorithms generalize to λ -covers, with (the first component of) the algorithm's complexity multiplied by $n^{\lambda-2}$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases quasiperiodicity, cover of a string, 2-cover, lambda-cover

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.77

Funding Supported by the “Algorithms for text processing with errors and uncertainties” project carried out within the HOMING program of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund, project no. POIR.04.04.00-00-24BA/16, and by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

Acknowledgements The authors thank Patryk Czajka for helpful discussions on the initial version of the algorithm.

1 Introduction

Identifying repetitive structure of a string is one of the key research areas of text algorithms, with applications to computational biology; see e.g. the books [19, 28]. Processing of a string that has a regular structure can be performed more efficiently, be it for pattern matching or for data compression.

The most elementary notion that grasps repetitiveness is *periodicity*. If a string can be generated by repeated concatenation of its smaller piece, then we say that it is periodic. The field of periodicity has been expanded upon by allowing not only concatenation, but also superpositions, which resulted in the introduction of *quasiperiodicity* by Apostolico and Ehrenfeucht [6].



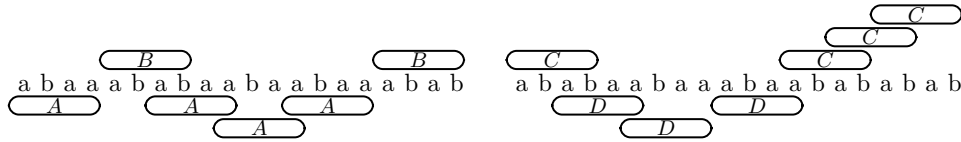
© Jakub Radoszewski and Juliusz Straszynski;
licensed under Creative Commons License CC-BY
28th Annual European Symposium on Algorithms (ESA 2020).

Editors: Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders; Article No. 77; pp. 77:1–77:17

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The basic terms of quasiperiodicity are the notions of *cover* and *seed*. A cover of a text T is a string whose occurrences in T cover all positions of T , while a seed of T is a cover of some superstring of T . An $\mathcal{O}(n)$ -time algorithm for computing the shortest cover of a text of length n was presented by Apostolico et al. [7]. Moore and Smyth showed that all the covers of a string can be computed in $\mathcal{O}(n)$ time [43, 44, 45]. Moreover, $\mathcal{O}(n)$ -time algorithms for computing covers of all prefixes of a string were shown [13, 41]. Seeds were introduced by Iliopoulos et al. [33] who showed an algorithm for finding a representation of all seeds of a string in $\mathcal{O}(n \log n)$ time. The majority of these classic algorithms were developed in the 1990s. It was not until many years later that an $\mathcal{O}(n)$ -time algorithm for computing seeds was found [36, 37]. Various approximate variants of covers and seeds were studied – see e.g. [3, 4, 16, 25, 38, 39] – as well as covers in other models of computation [10, 14, 26], in non-standard stringology [1, 9, 20, 31, 32] and in 2-dimensional texts [21, 47].

We consider *2-covers* which are a natural generalization of covers. A 2-cover of a text T is a pair of equal-length strings whose occurrences in T cover all positions of T ; see Figure 1. A yet more general notion of a λ -cover, that is, a λ -tuple of strings whose occurrences cover the whole text T , was introduced by Iliopoulos et al. in [27, 49]. Unfortunately, the authors’ $\mathcal{O}(n^2)$ time algorithm for finding all λ -covers under fixed λ and constant-size alphabet has been proven to be faulty. In reality, the algorithm has, at worst, exponential runtime [23].



■ **Figure 1** Two examples of a 2-cover of a string: a ps-cover (left) and a b-cover (right). Note that none of these strings has a proper cover.

In this paper, we present an $\mathcal{O}(n \log n \log \log n + \text{output})$ time algorithm for finding all 2-covers of a text of length n . Each string from a 2-cover in the output is represented by giving endpoints of its sample occurrence. Our algorithm can compute a 2-cover of each length or all shortest 2-covers in $\mathcal{O}(n \log n \log \log n)$ time. The complexities show the expected running time of the algorithms; they can be made worst-case at a cost of an additional $\log \log n / \log \log \log n$ factor. The space complexities of the algorithms are $\mathcal{O}(n)$. We assume the standard word-RAM model of computation.

In the case of previously mentioned seeds and covers, the input text is generated by concatenations and superpositions of a single string. However, in our problem, we need to check if the text can be generated by two strings of equal length. This alone suggests that the problem is computationally harder than its original counterpart. Intuitively, to find all covers of a string we need to check only $\mathcal{O}(n)$ candidates, i.e. all prefixes. This is not the case with 2-covers, because a text of length n can have up to $\Theta(n^2)$ different non-trivial 2-covers. (A simple example $T = a^m b a^m b a^m$ of such a text was shown in [23].) The general λ -cover problem was shown to be NP-hard by Cole et al. [17].

There are two types of 2-cover of a text T , as shown in Figure 1: a *ps-cover* (X, Y) that is composed of a prefix X and a suffix Y of T and a *b-cover* (X, Y) in which one of the strings, let us say X , is a border of T . (2-covers (X, Y) in which X is actually a cover of T are considered to be trivial and can be ignored.) Our main result consists of two algorithms, one for each of the types.

The first algorithm finds ps-covers. This is the easier type and for it, we propose an $\mathcal{O}(n \log \log n)$ expected time algorithm. It iterates over all possible candidates (there are $\mathcal{O}(n)$ of them) and maintains a set of *gaps*, that is, parts of the text that are not covered yet. There, we exploit locality of changes in coverage between consecutive lengths by using a predecessor data structure [5, 48]. Secondly, we efficiently express the dynamics of the gaps by storing linear functions.

The remaining, harder algorithm, finds b-covers (X, Y) . In this case there are significantly more candidates to consider (up to $\mathcal{O}(n^2)$ [23]). For each length ℓ we use string periodicity to compute a set of $\mathcal{O}(n/\ell)$ positions in T , called *anchors*, that implies all non-redundant occurrences of any string Y in a b-cover of length ℓ . This set is computed using Internal Pattern Matching [40]. Finally our algorithm forms a set of constraints on Y based on the anchors and finds all strings that satisfy these constraints in $\mathcal{O}(n \log \log n/\ell + \text{output})$ expected time using predecessor queries.

Our algorithms easily generalize to the λ -covers problem, achieving $\mathcal{O}(n^{\lambda-1} \text{polylog } n + \text{output})$ time.

► **Remark 1.** “String cover” is also used to describe a different notion that should not be confused with the one studied in this work. Namely, a string cover C of a set of strings S is a set of factors of strings from S such that every string in S can be written as a concatenation of the strings in C ; see [12, 15, 30, 46].

2 Preliminaries

By $[i..j]$ we denote the integer interval $\{i, \dots, j\}$; we use a round bracket if the interval does not contain one of its ends. For a set S of integers and integer a , by $S \oplus a$ and $S \ominus a$ we denote the sets $\{s + a : s \in S\}$ and $\{s - a : s \in S\}$, respectively, and by $intervals_k(S)$ we denote the set $\{i..i+k : i \in S\}$.

A string T is a sequence of letters from a given alphabet. The length of string T is denoted by $|T|$. We assume that the positions in T are numbered 1 through $|T|$, with letter at position i denoted as $T[i]$. By $T[i..j]$ we denote the string $T[i] \dots T[j]$ that is called a *factor* of T (the same notation is used for open intervals of positions). A factor $T[i..j]$ is called a *prefix* if $i = 1$ and a *suffix* if $j = |T|$.

For a string X , by $Occ_T(X)$ we denote the set of starting positions of occurrences of X in T and by $Cov_T(X)$ the set of positions that are covered by occurrences of X in T , i.e.,

$$Cov_T(X) = \bigcup intervals_{|X|}(Occ_T(X)).$$

We omit the subscript T when it is clear from the context. We say that a set of strings \mathcal{S} is a λ -cover of length ℓ of T if the following conditions hold:

- $|\mathcal{S}| = \lambda$
- $|X| = \ell$ for all $X \in \mathcal{S}$
- $\bigcup_{X \in \mathcal{S}} Cov(X) = [1..|T|]$

Periodicity of strings. We say that string S has *period* p (for $p \in [1..|S|]$) if $S[i] = S[i+p]$ for all $i \in [1..|S| - p]$.

► **Fact 2** (Periodicity lemma; Fine and Wilf [24]). *If string S has periods p and q such that $p + q \leq |S|$, then it has a period $\text{gcd}(p, q)$.*

A string is called *periodic* if it has a period that is at most a half of its length and *aperiodic* otherwise. Moreover, a string is called *4-periodic* if it has a period that is at most a quarter of its length.

77:4 Efficient Computation of 2-Covers of a String

► **Fact 3** (Folklore; see [2]). *If S is periodic and S' is a string of length $|S|$ that differs from S at exactly one position, then S' is aperiodic.*

In particular, if S is periodic with smallest period p and the letter c is different from $S[|S| - p + 1]$, then Sc , i.e., S concatenated with c , is aperiodic.

String B is called a *border* of string S if B is a prefix and a suffix of S . String S has a period p if and only if it has a border of length $n - p$. In particular, this implies the following.

► **Observation 4.** *If string S is not periodic, then $|Occ_T(S)| = \mathcal{O}(|T|/|S|)$.*

A string S is *primitive* if $S = V^k$ for a string V and positive integer k implies that $k = 1$.

► **Fact 5** (Synchronization property; [19, Lemma 1.11]). *A primitive string S has exactly two occurrences in S^2 .*

PREF table. The table *PREF* over a length- n string T stores, as $PREF[i]$, the length of the longest common prefix of T and $T[i..n]$. Let $PREF^R[i]$ denote the length of the longest common suffix of T and $T[1..i]$. Both arrays can be computed in $\mathcal{O}(n)$ time by a classical comparison-based algorithm, as in the Main-Lorentz algorithm [42]; see also the book [22].

Longest Common Extension (LCE) queries. Assume that string T is over an integer alphabet $[1..n^{\mathcal{O}(1)}]$. A longest common prefix (longest common suffix) query on T , given indices $i, j \in [1..n]$, returns the length of the longest common prefix of suffixes $T[i..n]$ and $T[j..n]$ (the length of the longest common suffix of $T[1..i]$ and $T[1..j]$, respectively). Both types of queries are often referred to as LCE queries. It is well-known that after $\mathcal{O}(n)$ -time preprocessing, one can answer LCE queries for T in $\mathcal{O}(1)$ time using the suffix array [34] and range minimum queries [11]. Moreover, we use the inverse suffix array that gives, for each suffix, its position in the sorted list of suffixes.

Assume that $T[i..j]$ is periodic with smallest period p . A position $j' > j$ ($i' < i$) is said to *break the periodicity* of $T[i..j]$ if $j' = \min\{k > j : T[k] \neq T[k - p]\}$ ($i' = \max\{k < i : T[k] \neq T[k + p]\}$, respectively). We set $i' = 0$ and $j' = n + 1$ if the respective position does not exist. One can use LCE queries to compute the positions breaking periodicity of a given factor $T[i..j]$, if they exist, in $\mathcal{O}(1)$ time.

Internal Pattern Matching (IPM) queries. Again assume that T is over an integer alphabet $[0..n^{\mathcal{O}(1)}]$. The IPM problem requires one to preprocess a text T of length n so that one can efficiently compute the occurrences of a factor of T in another factor of T . An $\mathcal{O}(n)$ -sized data structure, with $\mathcal{O}(n)$ expected time construction, that answers IPM queries in $\mathcal{O}(1)$ time when the ratio between the lengths of the two factors is at most 2 was presented in [40]. The set of occurrences is returned as a single arithmetic sequence. Moreover, if the sequence contains at least three elements, then its difference equals the smallest period of the pattern factor. A deterministic version of this data structure can be found in [35]. This data structure can also be used to answer in $\mathcal{O}(1)$ time so-called *two-period queries*, in which we are asked to find the smallest period of a given factor of T if this factor is periodic (an alternative data structure was proposed in [8]).

Predecessor data structures. For a set of integers A , by $pred(x, A)$ and $succ(x, A)$ we denote the predecessor and successor of x in A , that is, $\max\{a \in A : a < x\}$ and $\min\{a \in A : a > x\}$, respectively. (We assume that $\max \emptyset = -\infty$ and $\min \emptyset = \infty$.) We use the following known efficient dynamic predecessor data structures. A collection $A \subseteq [1..n]$ can be maintained

under insertions and deletions and can answer predecessor and successor queries in $\mathcal{O}(\log \log n)$ expected time per operation using a y-fast trie [48] or in $\mathcal{O}(\log^2 \log n / \log \log \log n)$ worst-case time using an exponential search tree [5]. Below by τ_n we denote the time complexity of an operation on a predecessor data structure. Moreover, we use Han's deterministic algorithm [29] to sort n numbers in $\mathcal{O}(n \log \log n)$ time.

3 Computing ps-covers

Let T be a string of length n . Let us start with a simpler but less efficient approach for computing ps-covers. For each length ℓ we would like to check if there is a ps-cover (X, Y) of length ℓ of T . We aim at $\mathcal{O}(n/\ell)$ time complexity after linear-time preprocessing. In the preprocessing phase we compute the data structures for LCE-queries [11, 34] and IPM queries [35, 40] in T . If T has a ps-cover (X, Y) of length ℓ , then $X = T[1.. \ell]$ and $Y = T[n - \ell + 1.. n]$. We apply IPM queries to compute the sets of occurrences $Occ(X)$ and $Occ(Y)$, represented as unions of $\mathcal{O}(n/\ell)$ of arithmetic sequences, in $\mathcal{O}(n/\ell)$ time. This lets us compute the sets $Cov(X)$ and $Cov(Y)$, represented as unions of $\mathcal{O}(n/\ell)$ maximal intervals, sorted left-to-right. Then we need to check if $Cov(X) \cup Cov(Y) = [1.. n]$, which can be done in linear time w.r.t. to the sizes of the representations of these sets by merging the sorted lists of intervals. Thus we have shown the following result.

► **Lemma 6.** *Let T be a string of length n over an integer alphabet. After $\mathcal{O}(n)$ -time and space preprocessing, one can compute a ps-cover of T of a given length ℓ , if it exists, in $\mathcal{O}(n/\ell)$ time.*

Let us note that Lemma 6 applied for all lengths $\ell = 1, \dots, n$ allows us to compute all ps-covers in $\mathcal{O}(n \log n)$ time. However, there is a more efficient approach that does not involve the intricate technique of IPM queries and also works for strings over any alphabet. We will use the lemma when computing λ -covers in Section 5.

Let P_ℓ be the prefix of length ℓ of T and S_ℓ be the suffix of length ℓ of T . For each length ℓ there is only one candidate for a ps-cover, that is, (P_ℓ, S_ℓ) . Furthermore, the set of positions of the text T that are covered by $Cov(P_\ell) \cup Cov(S_\ell)$ does not change much when ℓ is incremented.

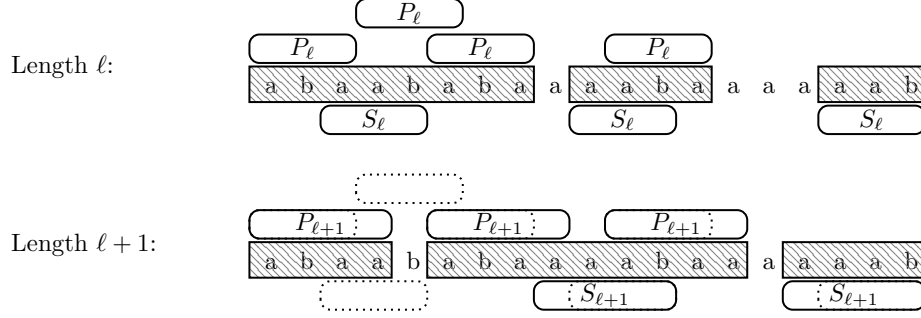
The idea is to iterate over increasing values of ℓ and check whether occurrences of P_ℓ and S_ℓ cover the entire text. We are going to maintain a set of *gaps*, that is, parts of the text that are covered by occurrences of neither the prefix nor the suffix.

First, let us identify an occurrence of a prefix P_ℓ with the index of its leftmost character and an occurrence of a suffix S_ℓ with the index of its *rightmost* character. In this way, when the length ℓ is incremented, some occurrences persist and get their length increased by one and other occurrences disappear. Specifically, occurrences of the prefix extend to the right and, respectively, occurrences of the suffix extend to the left. As a result, some gaps shrink or disappear and some other gaps are created. For an example, see Figure 2. Because of the way how joint occurrences of the prefix and the suffix affect the sizes of gaps, we will refer to these occurrences as the *pressing factors*.

We will iterate over subsequent $\ell = 1, \dots, n$ and observe the set of gaps. If for some ℓ the set of gaps is empty, then (P_ℓ, S_ℓ) is a ps-cover. We track the following data:

- length ℓ
- the set \triangleright_ℓ of left endpoints of occurrences of P_ℓ
- the set \triangleleft_ℓ of right endpoints of occurrences of S_ℓ
- a set of pairwise disjoint gaps and an expiration time (value of ℓ) for each of them.

77:6 Efficient Computation of 2-Covers of a String



■ **Figure 2** Illustration of gap dynamics. After incrementation of ℓ , a gap b was created, a gap a disappeared, and a gap aaa shrunk to a .

The sets will be maintained using predecessor data structures, which allow to perform predecessor/successor queries in τ_n time. Using the aforementioned data, the outline of the algorithm is as follows:

■ **Algorithm 1** Outline of the algorithm for computing ps-covers.

```

pressing_factors := Occurrences of  $P_1$  and  $S_1$  in  $T$ ;
gaps := Gaps between pressing_factors;
for  $\ell := 1$  to  $n$  do
    to_remove := Expired pressing_factors;
    Remove to_remove from pressing_factors;
    foreach expired_factor in to_remove do
        Recalculate elements of gaps around expired_factor;

```

An occurrence $i \in \triangleright_\ell$ ($i \in \triangleleft_\ell$) persists as long as $\ell \leq \text{PREF}[i]$ ($\ell \leq \text{PREF}^R[i]$, respectively). Therefore, for $\ell = \text{PREF}[i] + 1$ ($\ell = \text{PREF}^R[i] + 1$), we consider that occurrence as expired. In conclusion, the PREF arrays allow us to compute expiration times of every prefix and suffix. This allows us to efficiently compute expired pressing factors in amortized $\mathcal{O}(1)$ time by precomputing a list of factors to expire for each moment of time in $\mathcal{O}(n)$ time.

Now let us simulate gap dynamics. Incrementations of ℓ successively get a gap increasingly covered (by occurrences of a prefix and/or suffix) until it expires completely. Assuming that none of the relevant pressing factors disappears, a gap expiration depends on the closest prefix occurrence to the left and the closest suffix occurrence to the right of the gap. If we know that some position p belongs to a gap, we would like to know the following:

- $L_\triangleright = \max\{a : a \in \triangleright_\ell, a < p\}$ and $L_\triangleleft = \max\{a : a \in \triangleleft_\ell, a < p\}$
- $R_\triangleright = \min\{b : b \in \triangleright_\ell, b > p\}$ and $R_\triangleleft = \min\{b : b \in \triangleleft_\ell, b > p\}$.

Unfortunately, this is too much to maintain. One factor that expires might influence many gaps. Let us analyze it further. Let us fix some prefix occurrence, i.e. pressing factor that extends to the right. It might influence expiration time of many gaps to the right. On the other, hand we can safely note this exclusively in the closest gap to the right. This is because the pressing factor won't reach other gaps before closing the immediate gap. When the gap closes, we can propagate the information to neighbouring gaps. Therefore, in a gap we only take into consideration pressing factors whose immediate neighbour is this gap and ignore them otherwise. We can easily check for this and compute all these values in τ_n time. If the gap initially covers the interval $[i..j]$, then it can expire in two ways:

- it can close on one boundary by a single opposing pressing factor, so the gap will close no later than $\ell = \min(R_\triangleleft - i + 1, j - L_\triangleright + 1)$, or

- it can close in the middle of the gap, by both pressing factors simultaneously, at $\ell = \lceil \frac{R_{\triangleleft} - L_{\triangleright} + 1}{2} \rceil$.

The endpoints of a gap at moment ℓ can be computed using the formulas:

$$i = \max(L_{\triangleright} + \ell, L_{\triangleleft} + 1) \text{ and } j = \min(R_{\triangleleft} - \ell, R_{\triangleright} - 1).$$

When a gap is created or its neighbouring pressing factors are altered, we use these formulas to recompute the gap boundaries. The predecessor data structure that stores gaps uses, for each gap, its recently computed left boundary for comparison. It is sufficient since the left-to-right order of gaps is never changed.

Thus we can recompute the expiration moment of a single gap given at least one position belonging to the gap. The remaining issue is to know which gaps need to be updated. Note that each expired factor can affect at most two existing neighbouring gaps and possibly introduce a new one. We can find the neighbouring gaps via predecessor/successor queries. Positions that were not covered will still not be covered after removing the expired factor, so we can pick an arbitrary position from this gap and recalculate its boundaries.

Now, we need to check if some new gap was created in the boundaries of the expired factor. In this case we have some intervals of length ℓ , representing the set $Cov(P_\ell) \cup Cov(S_\ell)$, and we would like to know if removing one interval creates a gap in coverage. Thanks to the fact that all intervals are of the same length, if the expired factor is $[i..j]$, we only need to find the last interval ending at most at j and the first interval starting at least at i . If found intervals do not cover the entirety of $[i..j]$, we have at least one position of the gap and we are able to calculate its boundaries. Otherwise, removing the factor did not change the coverage, so no new gap was created. All of this can be performed using the predecessor data structures in τ_n time.

In conclusion, the entire computation of ps-covers takes $\mathcal{O}(n\tau_n)$ time and $\mathcal{O}(n)$ space. We obtain the following result.

► **Theorem 7.** *Let T be a string of length n over any alphabet that allows $\mathcal{O}(1)$ -time checking of letter equality. One can compute a ps-cover of T of every possible length in $\mathcal{O}(n\tau_n)$ time and $\mathcal{O}(n)$ space.*

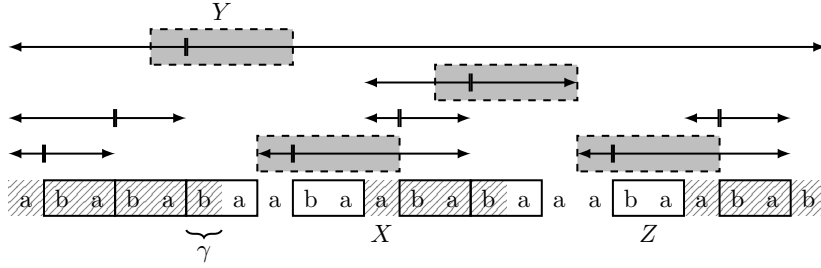
4 Computing b-covers

Let T be a string of length n . Our goal in this section is, given a length ℓ , to check if there is a b-cover (X, Y) of length ℓ of T . We aim at $\mathcal{O}(n\tau_n/\ell)$ time complexity after linear-time preprocessing. In the preprocessing phase we compute the data structures for LCE-queries [11, 34] and IPM queries [35, 40] in T .

Let $X = T[1.. \ell]$. We apply IPM queries to compute the set $Occ(X)$, represented as a union of $\mathcal{O}(n/\ell)$ of arithmetic sequences, and the set $Cov(X)$, represented as a union of $\mathcal{O}(n/\ell)$ maximal intervals, in $\mathcal{O}(n/\ell)$ time. If $n - \ell + 1 \notin Occ(X)$, there is no b-cover of length ℓ , and if $Cov(X) = [1..n]$, we skip this length since we have the trivial case of a 2-cover containing a cover. Henceforth we assume that X is a border of T whose occurrences do not cover the whole string T .

Our goal is to find all strings Y for which (X, Y) is a b-cover of T . We start by building up some intuition. We have $|Y| = \ell$, so in order for Y to cover all positions from the set $Cov(Y)$, it suffices to use $\mathcal{O}(n/\ell)$ occurrences of Y (instead of, potentially, $\Theta(n)$ occurrences). Let P_Y be a set of starting positions of such a set of occurrences. We will compute $t = \mathcal{O}(1)$ sets $\Gamma_1, \dots, \Gamma_t$, each of size $\mathcal{O}(n/\ell)$, that contain information about all (Y, P_Y) , for every Y that can form a b-cover with X . In each set Γ_i we will select an element $\gamma_i \in \Gamma_i$ and consider only length- ℓ factors Y starting at positions $\gamma_i - a$ for $a \in [0.. \ell]$.

77:8 Efficient Computation of 2-Covers of a String



■ **Figure 3** This string has a b-cover $(X = abab, Y = abaa)$. The sets $Cov(X)$ and $Cov(Y)$ are shown in gray. We have $Z = ba$, $\Gamma = Occ_T(Z)$, and γ is the position of the occurrence of Z that ends at the first position that is not covered by $Cov(X)$. The set P_Y of occurrences of Y that is generated by $(\Gamma, \gamma, 1)$ is shown. For the meaning of arrows, see Section 4.3.

In particular, for every such (Y, P_Y) we would like to have $P_Y \subseteq (\Gamma_i \ominus a)$ and $Y = T[\gamma_i - a .. \gamma_i - a + \ell]$ for some $i \in [1..t]$ and $a \in [0.. \ell)$. We then say that (Y, P_Y) is generated by (Γ_i, γ_i, a) . Moreover, for each set Γ_i we will provide an interval $J_i \subseteq [0.. \ell)$ such that for every Y that forms a b-cover with X , the factor Y is generated by (Γ_i, γ_i, a) for just a constant number of $a \in J_i$. This will allow us to report each sought factor Y a constant number of times and filter out repetitions in the end.

In the algorithm we first compute a constant number of factors Z_1, \dots, Z_t of T length $z = \lceil \ell/2 \rceil$ such that if (X, Y) is a b-cover, then Y contains at least one of Z_1, \dots, Z_t as a factor. Let Z be a factor of Y such that $a + 1 \in Occ_Y(Z)$. If $i \in Occ_T(Z)$ and $i - a \in Occ_T(Y)$, then we say that the occurrence i of Z *a-anchors* the occurrence $i - a$ of Y and that the latter is *a-anchored* at the former. If Z_i is aperiodic, by Observation 4, we have $|Occ_{Z_i}(T)| = \mathcal{O}(n/\ell)$ and $|Occ_{Z_i}(Y)| = \mathcal{O}(1)$ for any length- ℓ string Y . In this case we will take $\Gamma_i = Occ_{Z_i}(T)$ and $J_i = [0.. \ell - z]$. If Z_i is periodic with period p , we will only be interested in factors Y that are periodic with the same period. In this case we will take as Γ_i a sufficient subset of $Occ_{Z_i}(T)$ and set $J_i = [0.. p)$. See Figure 3 for an example.

Formally, we reduce computing a b-cover of a given length to a constant number of instances of the following problem.

POSITIONED COVER OF LENGTH ℓ
Input: A factor Z of T , a set of positions $\Gamma \subseteq Occ_T(Z)$, its element $\gamma \in \Gamma$, and an interval $J \subseteq [0.. \ell)$.
Output: Report all $a \in J$ such that $Cov_T(X) \cup (\bigcup intervals_\ell(P_Y)) = [1.. n]$ for (Y, P_Y) that is generated by (Γ, γ, a) , with $|Y| = \ell$.

In Section 4.3 we show how to solve this problem efficiently if $|\Gamma| = \mathcal{O}(n/\ell)$. Clearly:

► **Observation 8.** *If an instance of POSITIONED COVER OF LENGTH ℓ for any Γ, γ, J has a solution (X, Y) for some $a \in J$, then (X, Y) is a b-cover of T .*

Let i be the first position of T that is not covered by occurrences of X . Hence, i has to be covered by the second string Y of the b-cover. Let us denote

$$z = \lceil \ell/2 \rceil, \quad Z_1 = T[i - z + 1 .. i], \quad Z_2 = T[i .. i + z].$$

► **Observation 9.** *If (X, Y) is a b-cover of length ℓ of T , then Z_1 or Z_2 is a factor of Y .*

Proof. Let $T[j..j+\ell)$ be an occurrence of Y that covers the position i . If $j \leq i - z + 1$, then it covers the factor Z_1 . Otherwise, $j + \ell - 1 \geq i + z$ and $j \leq i$, so it covers Z_2 . ◀

We will consider as Z each of the two factors Z_1, Z_2 and denote by i_Z the starting position of the occurrence of Z mentioned in the definition. We can ask a two-period query [8, 35, 40] to check if Z is periodic and, if so, compute its smallest period.

► **Observation 10.** *If Z is aperiodic, then Y is not 4-periodic. If Z is periodic, then either Y is 4-periodic with the same period, or Y is not 4-periodic.*

Proof. Assume that Z is aperiodic. If Y was 4-periodic with period p , i.e., $4p \leq \ell$, then p would also be a period of its factor Z and $2p \leq z$, so Z would be periodic.

Assume now that Z is periodic. Let p be the smallest period of Z ; we have $2p \leq z$. Assume to the contrary that Y is 4-periodic with smallest period p' such that $p' \neq p$. We have $4p' \leq \ell$, so $2p' \leq z$. Then p' is not a multiple of p , since otherwise p would have been a period of Y . By the periodicity lemma (Fact 2), Z has period $\gcd(p, p') < p$, a contradiction. ◀

In the remainder of the reduction we consider two cases depending on if Y is 4-periodic.

4.1 Reduction for non-4-periodic Y

If Z is periodic, then we try two ways of substituting it with a string that is not periodic.

► **Observation 11.** *Assume that Z is periodic with smallest period p , Y is not 4-periodic and an occurrence $T[i..i+\ell)$ of Y contains $T[i_Z..i_Z+z)$. Let $i' < j'$ be the positions that break the periodicity of $T[i_Z..i_Z+z)$. Then $T[i..i+\ell)$ contains at least one of the fragments $T[i'..i'+z)$, $T[j'-z+1..j')$.*

We denote the fragments in the conclusion of the observation by Z' and Z'' , respectively. Let us recall that if Z is periodic, the positions breaking the periodicity can be computed using LCE queries. Hence, Z' and Z'' can be computed in $\mathcal{O}(1)$ time. By Fact 3, if Z' or Z'' exists, it is aperiodic. If Z is periodic, we try replacing it by Z' or Z'' (and redefine the occurrence i_Z).

In total we obtain up to four aperiodic strings Z such that if Y is not 4-periodic, its occurrence contains the occurrence $T[i_Z..i_Z+z)$ for at least one of them. We have $|Occ(Z)| = \mathcal{O}(n/\ell)$ (Observation 4) and all the occurrences can be found in $\mathcal{O}(n/\ell)$ time using IPM queries. The following lemma summarizes the above argument.

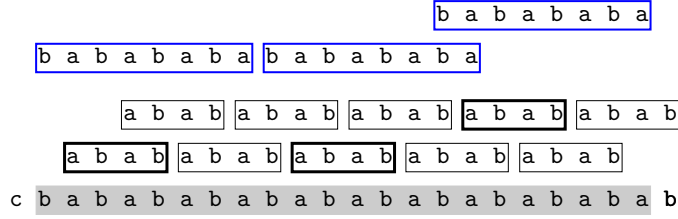
► **Lemma 12.** *If T has a b -cover (X, Y) of length ℓ with non-4-periodic Y , then (Y, P_Y) is generated by (Γ, γ, a) where $\Gamma = Occ(Z)$, $\gamma = i_Z$ and $a \in [0.. \ell - z]$, for one of up to four aperiodic strings Z . We have $|\Gamma| = \mathcal{O}(n/\ell)$ and Γ, γ can be computed in $\mathcal{O}(n/\ell)$ time.*

4.2 Reduction for 4-periodic Y

By Observation 10, in this case Z is necessarily periodic with the same smallest period as Y . If we used the same reduction as in Lemma 12, we could unfortunately have $|\Gamma| = |Occ(Z)| = \Theta(n)$. We deal with this problem by choosing the first occurrence of Z in Y as an anchor and selecting only some of the occurrences of Z in T to the set Γ that are sufficient for Y to cover all positions in $Cov(Y)$; see Figure 4.

► **Lemma 13.** *If T has a b -cover (X, Y) of length ℓ with 4-periodic Y , then (Y, P_Y) is generated by (Γ, γ, a) where $\Gamma \subseteq Occ(Z)$ and $a \in [0.. p)$, for one of up to two periodic strings Z with smallest period p and one of up to two positions γ . We have $|\Gamma| = \mathcal{O}(n/\ell)$ and Γ, γ, p can be computed in $\mathcal{O}(n/\ell)$ time.*

77:10 Efficient Computation of 2-Covers of a String



■ **Figure 4** $Z = \text{abab}$ (black rectangles), $Y = \text{babababa}$ (blue rectangles); gray color shows $\text{Cov}(Y)$. The occurrences of Y that are 1-anchored at marked occurrences of Z are shown and cover $\text{Cov}(Y)$.

Proof. Let $p = \text{per}(Z)$. We apply IPM queries to compute the set $\text{Occ}(Z)$, represented as a union of $\mathcal{O}(n/\ell)$ arithmetic sequences with difference p . Let us further merge these arithmetic sequences into maximal sequences with difference p , that we denote as S_1, \dots, S_k . We note that $Z[1..p]$ is primitive, since otherwise Z would have a smaller period. By the synchronization property (Fact 5) for $Z[1..p]$, we can assume that $\max(S_i) + p < \min(S_{i+1})$ for $i = 2, \dots, k$, so $\sum_{i=1}^k |S_i| = \mathcal{O}(n/p)$. Initially let $\Gamma = \text{Occ}(Z)$. We will show how to prune Γ by leaving $\mathcal{O}(|S_i|p/\ell)$ elements in each of the sequences S_i . This will indeed give $|\Gamma| = \mathcal{O}(n/\ell)$.

The set $\text{Occ}_Y(Z)$ is an arithmetic sequence with difference p and first element $t \in [1..p]$. Let $m = |\text{Occ}_Y(Z)|$; we have $2 \leq m \leq \ell/p$. An occurrence of Y in T implies a subsequence of length m of consecutive elements in one of the sequences S_i . Moreover, any arithmetic sequence $j, j+p, \dots, j+(m+1)p$ of $m+2$ occurrences of Z in T implies an occurrence of Y in T at position $j+p-t+1$. (Note that a difference- p arithmetic sequence of $m+1$ occurrences of Z in T does not have to imply an occurrence of Y , e.g. if $T = \text{abababab}$, $Z = \text{abab}$ and $Y = \text{babababa}$.)

We can now construct the pruned set Γ' as follows. Let us consider $S_i = \{j, j+p, \dots, j+(w-1)p\}$. If $w+1 < m$, then we can ignore S_i . Otherwise we insert to Γ' :

- the elements j and $j+p$;
- all elements $j+m \cdot p \cdot t \in S_i$ for positive integer t ;
- the elements $j+(w-m-1)p$ and $j+(w-m)p$.

This way $\mathcal{O}(w/m) = \mathcal{O}(|S_i|p/\ell)$ elements are inserted to Γ' , so indeed $|\Gamma'| = \mathcal{O}(n/\ell)$.

Finally, let S_b be the arithmetic sequence that contains the position i_Z . Then we have two choices for γ : $\min(S_b)$ or $\min(S_b) + p$. ◀

4.3 Solution to Positioned Cover problem

Let us recall the problem statement.

POSITIONED COVER OF LENGTH ℓ

Input: A factor Z of T , a set of positions $\Gamma \subseteq \text{Occ}_T(Z)$, its element $\gamma \in \Gamma$, and an interval $J \subseteq [0.. \ell)$.

Output: Report all $a \in J$ such that $\text{Cov}_T(X) \cup (\bigcup \text{intervals}_\ell(P_Y)) = [1..n]$ for (Y, P_Y) that is generated by (Γ, γ, a) , with $|Y| = \ell$.

► **Lemma 14.** After $\mathcal{O}(n)$ time and space preprocessing, assuming that $|\Gamma| = \mathcal{O}(n/\ell)$, POSITIONED COVER OF LENGTH ℓ over an integer alphabet can be solved in $\mathcal{O}(n\tau_n/\ell + \text{output})$ time and $\mathcal{O}(n/\ell)$ space.

Proof. Let $A = \text{Cov}(X)$, $A' = [1..n] \setminus A$, and $\mathcal{A} \subseteq [1..n]^2$ be the set of maximal intervals of A . We have $|\mathcal{A}| \leq n/\ell$ and \mathcal{A} can be computed in $\mathcal{O}(n/\ell)$ time. Then the POSITIONED COVER problem can be solved with the following Claim 15 for

$$S = \{(i, \text{lcs}(T[1..i], T[1..\gamma]), \text{lcp}(T[i..n], T[\gamma..n])) : i \in \Gamma\},$$

where lcp and lcs is the length of the longest common prefix and the longest common suffix, respectively. Intuitively, if $(i, x, y) \in S$, then there is an occurrence of a length- ℓ factor Y a -anchored at $i \in \text{Occ}(Z)$ if and only if $a \leq x$ and $|Z| \leq \ell - a \leq y$. See the arrows in Figure 3.

▷ **Claim 15.** In $\mathcal{O}(n\tau_n/\ell + \text{output})$ time and $\mathcal{O}(n/\ell)$ space one can report all $a \in J$ such that

$$A \cup \bigcup \text{intervals}_\ell(S'_a \ominus a) = [1..n], \quad (1)$$

where $S'_a = \{i : (i, x, y) \in S, a \leq x, \ell - a \leq y\}$.

Proof. In the algorithm we store \mathcal{A} in a predecessor data structure $D_{\mathcal{A}}$ sorted by the left endpoints of intervals. We will consider all $a \in J$ in a decreasing order and store the current set S'_a in a predecessor data structure D_S . However, we will only consider values of a for which $S'_a \neq S'_{a+1}$. Let us note that $(i, x, y) \in S$ contributes to $i \in S'_a$ for $a \in [\ell - y..x]$. Hence, if this interval is non-empty, we will insert i to S'_a for $a = x$ and remove it for $a = \ell - y - 1$. We have $|S| = \mathcal{O}(n/\ell)$, so all events of insertion and deletion to D_S can be precomputed and sorted in $\mathcal{O}(n \log \log n/\ell)$ time using Han's algorithm [29].

Assume that D_S is the data structure that stores S'_a for all a in an interval $J_0 \subseteq J$. Let $i \in D_S$ and $i' = \text{succ}(i, D_S)$. We can observe that:

- If $K = [i..i'] \setminus A$ is non-empty and (1) holds for some $a \in J_0$, then $K \subseteq [i - a..i - a + \ell) \cup [i' - a..i' - a + \ell)$.

Hence, if $[i..i')$ is to be covered by the left hand side of (1) for some $a \in J_0$, we have the following set $C(i, i')$ of constraints on a (see Figure 5 in the appendix):

- (a) If $i' - i \leq \ell$ or $[i..i') \subseteq A$, no constraints are imposed. If there are at least two intervals from \mathcal{A} that are fully contained in $[i..i')$, then there is no such a .
- (b) Otherwise, if no interval in \mathcal{A} is a subset of $[i..i')$, then $a \geq i' - j$ or $\ell - a \geq j' - i + 1$, where $j = \text{succ}(i, A')$ and $j' = \text{pred}(i' - 1, A')$.
- (c) Otherwise, if there is an interval $[u..v] \in \mathcal{A}$ such that $[u..v] \subseteq [i..i')$, then $a \geq i' - v - 1$ and $\ell - a \geq u - i$.

The respective cases can be checked and $C(i, i')$ can be constructed in τ_n time using $D_{\mathcal{A}}$. A similar set of conditions can be stated for the left hand side of (1) to contain all elements of $[1.. \min D_S)$ and $[\max D_S..n]$; we denote the resulting constraints by $C(0, \min D_S)$ and $C(\max D_S, n + 1)$, respectively, and insert 0 and $n + 1$ to S'_a .

Let us note that each of the constraints from the set $C(i, i')$ is a conjunction of at most two constraints of the form $a \notin I$ for some interval I . Indeed,

$$(a \geq x) \vee (a \leq y) \Leftrightarrow a \notin (y..x), \quad (a \geq x) \wedge (a \leq y) \Leftrightarrow (a \notin [0..x)) \wedge (a \notin (y.. \ell)).$$

When i is inserted to D_S , we remove the constraints $C(i', i'')$ imposed by the pair $i' = \text{pred}(i, D_S)$ and $i'' = \text{succ}(i, D_S)$ and insert the constraints $C(i', i)$ and $C(i, i'')$. For every constraint $a \notin I$, we will retain the value a_1 of a for which it is inserted and the value a_2 for which it is removed. If $I' = [a_1..a_2)$, the constraint imposes a constraint $a \notin (I \cap I')$ on values of a that satisfy (1).

77:12 Efficient Computation of 2-Covers of a String

Overall we obtain $\mathcal{O}(n/\ell)$ constraints of the form $a \notin I$ for (1) to hold. Our goal is to report all $a \in J$ that satisfy all the constraints, i.e., all a in the complement of the union of the $\mathcal{O}(n/\ell)$ intervals from the constraints. This task can be completed by a classic 1d sweep algorithm if the endpoints of intervals from the constraints are sorted [29].

The data structure $D_{\mathcal{A}}$ takes $\mathcal{O}(n\tau_n/\ell)$ time to construct since $|\mathcal{A}| = \mathcal{O}(n/\ell)$ and admits $\mathcal{O}(n/\ell)$ queries. The data structure D_S admits $\mathcal{O}(n/\ell)$ operations. Additional sorting takes $\mathcal{O}(n\tau_n/\ell)$ time. Finally, all values of a for which (1) is satisfied are reported in $\mathcal{O}(\text{output})$ time. The complexity follows. \blacktriangleleft

This concludes the solution to POSITIONED COVER problem. \blacktriangleleft

A single string Y can be generated by (Γ, γ, a) with $a \in J$ from Lemma 12 a constant number of times because Z is aperiodic, and a constant number of times from Lemma 13 because of the synchronization property. By combining Lemma 14 with Observation 8 and the reductions of Lemmas 12 and 13, we obtain the following result and its corollary.

► **Lemma 16.** *Let T be a string of length n over an integer alphabet. After $\mathcal{O}(n)$ -time and space preprocessing, one can report all b -covers of T of a given length ℓ , each of them $\mathcal{O}(1)$ times, in $\mathcal{O}(n\tau_n/\ell + \text{output})$ time.*

► **Theorem 17.** *Let T be a string of length n over any ordered alphabet. All b -covers of T can be computed in $\mathcal{O}(n\tau_n \log n + \text{output})$ time and $\mathcal{O}(n)$ space.*

Proof. Let us sort and renumber letters in T so that they are in $[1..n]$. This takes $\mathcal{O}(n \log n)$ time. Then we apply Lemma 16 for every possible length ℓ of a b -cover. Apart from the time to report the output, the complexity becomes $\sum_{\ell=1}^n \mathcal{O}(n\tau_n/\ell) = \mathcal{O}(n\tau_n \log n)$.

Finally, we need to make sure that each b -cover is reported only once. We can use the inverse suffix array to sort all factors Y of a given length in the lexicographic order. The sorting is performed globally, across all lengths, using radix sort. We can then iterate over length- ℓ strings Y in the sorted order and remove duplicates using LCE-queries. \blacktriangleleft

5 Computation of 2-covers and λ -covers

We summarize the results of Theorems 7 and 17 and use efficient predecessor data structures [5, 48] to obtain the following result.

► **Theorem 18.** *Let T be a string of length n over any ordered alphabet. All 2-covers of T can be computed in $\mathcal{O}(n \log n \log \log n + \text{output})$ expected time or $\mathcal{O}(n \log n \log^2 \log n / \log \log \log n + \text{output})$ worst-case time and $\mathcal{O}(n)$ space.*

Let us recall that there are up to n ps-covers. Moreover, the algorithm behind Lemma 16 allows one to generate as many b -covers of a given length as one requires. This shows that indeed one can compute a 2-cover of each possible length or all the shortest 2-covers in $\mathcal{O}(n\tau_n \log n)$ time.

Theorem 19 extends Theorem 18 to λ -covers for any $\lambda \geq 2$. As in the case of 2-covers, we are only interested in computing λ -covers of lengths for which T does not have a $(\lambda - 1)$ -cover.

► **Theorem 19.** *Let T be a string of length n over any ordered alphabet. For any $\lambda \geq 2$, all λ -covers of T can be computed in $\mathcal{O}(n^{\lambda-1} \log n \log \log n + \text{output})$ expected time or $\mathcal{O}(n^{\lambda-1} \log n \log^2 \log n / \log \log \log n + \text{output})$ worst-case time and $\mathcal{O}(n)$ space.*

Proof. It suffices to give a proof for $\lambda \geq 3$. Similarly as in the case of 2-covers, we classify λ -covers $\mathcal{S} = (X_1, \dots, X_\lambda)$ into ps- λ -covers, for which X_1 is a prefix and X_2 is a suffix of T , and b- λ -covers, for which X_1 is a border of T . (Formally, in order to compute all λ -covers, in case of ps- λ -covers in the end we need to generate all tuples where the prefix and suffix of T are not the first two respective elements of the tuple, and similarly for b- λ -covers.) The two cases are handled similarly as ps-covers and b-covers, respectively. The number of ps- λ -covers is upper bounded by $n^{\lambda-1}$, whereas the number of b- λ -covers can be $\Theta(n^\lambda)$; see [23].

Let us show how to compute all ps- λ -covers of a given length $\ell \in [1..n]$. First we use IPM queries to compute $Cov(X_1) \cup Cov(X_2)$, represented as a union of $\mathcal{O}(n/\ell)$ maximal intervals, as in Lemma 6. We LCE-queries on suffixes of the suffix array of T to partition positions of T into classes C_1, \dots, C_m such that positions i, j belong to the same class if and only if $T[i..i+\ell] = T[j..j+\ell]$. This could be also done in $\mathcal{O}(n \log n)$ total time using Crochemore's partitioning [18]. For each of the $\binom{m}{\lambda-2}$ choices of $\lambda-2$ classes $C_{i_1}, \dots, C_{i_{\lambda-2}}$, if none of them corresponds to X_1 or X_2 , we compute their union B . The sets B are computed simultaneously for several choices containing $\Theta(n)$ elements in total using radix sort in order to achieve $\mathcal{O}(|C_{i_1}| + \dots + |C_{i_{\lambda-2}}|)$ amortized time per choice. Within the same time complexity we can compute the set $\bigcup intervals_\ell(B)$ represented as a union of maximal intervals. Finally, we merge this set with $Cov(X_1) \cup Cov(X_2)$ and check if their union is $[1..n]$. The time complexity for a given choice of classes is $\mathcal{O}(|C_{i_1}| + \dots + |C_{i_{\lambda-2}}| + n/\ell)$.

Over all choices, the running time is proportional to

$$\sum_{1 \leq i_1 \leq \dots \leq i_{\lambda-2} \leq m} \left(|C_{i_1}| + \dots + |C_{i_{\lambda-2}}| + \frac{n}{\ell} \right) = \binom{m-1}{\lambda-3} \sum_{i=1}^m |C_i| + \binom{m}{\lambda-2} \frac{n}{\ell} \leq \frac{2n^{\lambda-1}}{\ell}. \quad (2)$$

The total cost of computing all classes C_i , over all $\ell \in [1..n]$, is $\mathcal{O}(n^2)$ (or $\mathcal{O}(n \log n)$), and the other preprocessing (LCE and IPM) takes $\mathcal{O}(n)$ time. Thus the overall cost of computing all ps- λ -covers is $\mathcal{O}(n^{\lambda-1} \log n)$.

Computation of b- λ -covers is a similar adjustment to the computation of b-covers of a given length. Recall that X_1 is a length- ℓ border of T . We iterate over all $\binom{m}{\lambda-2}$ choices of $\lambda-2$ classes $C_{i_1}, \dots, C_{i_{\lambda-2}}$ which corresponds to selecting factors $X_2, \dots, X_{\lambda-1}$ from the b- λ -cover. A selection for which $X_i = X_1$ for some $i > 1$ is discarded. The set $\mathcal{C} := Cov(X_1) \cup \dots \cup Cov(X_{\lambda-1})$ can be expressed as a union of $\mathcal{O}(n/\ell)$ maximal intervals in $\mathcal{O}(|C_{i_1}| + \dots + |C_{i_{\lambda-2}}| + n/\ell)$ time, which is $\mathcal{O}(n^{\lambda-1}/\ell)$ overall by (2).

In order to compute X_λ , we we make a reduction to a generalization of POSITIONED COVER OF LENGTH ℓ in which we take \mathcal{C} instead of $Cov_T(X)$. The factors Z_1 and Z_2 are computed as in Observation 9, by setting i to the first position in T that is not covered by \mathcal{C} . This allows us to compute Z depending on if X_λ is 4-periodic, as in Sections 4.1 and 4.2, in $\mathcal{O}(1)$ time. The solution of the general POSITIONED COVER OF LENGTH ℓ is the same as the one given in Lemma 14, but using \mathcal{C} instead of $Cov_T(X)$. The time complexity of the solution is $\mathcal{O}(n\tau_n/\ell)$ plus the time needed to output b- λ -covers. These steps need to be performed for each of the $\binom{m}{\lambda-2} \leq n^{\lambda-2}$ choices of classes, which gives $\mathcal{O}(n^{\lambda-1}\tau_n/\ell)$ for the given length ℓ , and $\mathcal{O}(n^{\lambda-1}\tau_n \log n)$ in total (plus output).

The complexity follows by summing the complexities of computing ps- λ -covers and b- λ -covers and using efficient predecessor data structures [5, 48]. ◀

6 Conclusions and open problems

We presented quasi-linear time algorithms (plus the time to report the output) for computing 2-covers of a string. One could ask if a shortest 2-cover can be computed in linear time. A further problem is to check if the general λ -cover problem parameterized by λ is fixed-parameter tractable.

One could also consider alternative definitions of 2-covers (and λ -covers) in which the factors that are to cover the text need not to be of the same length. Efficient computation of such covers seems to be an interesting open problem. In particular, under this alternative definition, there can be $\Theta(n^4)$ candidates for a 2-cover (every pair of factors).

References

- 1 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.
- 2 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.
- 3 Amihood Amir, Avivit Levy, Moshe Lewenstein, Ronit Lubin, and Benny Porat. Can we recover the cover? *Algorithmica*, 81(7):2857–2875, 2019. doi:10.1007/s00453-019-00559-8.
- 4 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate cover of strings. *Theoretical Computer Science*, 793:59–69, 2019. doi:10.1016/j.tcs.2019.05.020.
- 5 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):13, 2007. doi:10.1145/1236457.1236460.
- 6 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.
- 7 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.
- 8 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The “runs” theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.
- 9 Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *Information and Computation*, 270:104462, 2020. doi:10.1016/j.ic.2019.104462.
- 10 Amir M. Ben-Amram, Omer Berkman, Costas S. Iliopoulos, and Kunsoo Park. The subtree max gap problem with application to parallel string covering. In Daniel Dominic Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 501–510. ACM/SIAM, 1994. URL: <http://dl.acm.org/citation.cfm?id=314464.314633>.
- 11 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.
- 12 Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, Michael T. Hallett, and Harold T. Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, 11(1):49–57, 1995. doi:10.1093/bioinformatics/11.1.49.
- 13 Dany Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992. doi:10.1016/0020-0190(92)90111-8.
- 14 Dany Breslauer. Testing string superprimitivity in parallel. *Information Processing Letters*, 49(5):235–241, 1994. doi:10.1016/0020-0190(94)90060-4.

- 15 Stefan Canzar, Tobias Marschall, Sven Rahmann, and Chris Schwiegelshohn. Solving the minimum string cover problem. In *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012*, pages 75–83. SIAM / Omnipress, 2012. doi:10.1137/1.9781611972924.8.
- 16 Manolis Christodoulakis, Costas S. Iliopoulos, Kunsoo Park, and Jeong Seop Sim. Approximate seeds of strings. *Journal of Automata, Languages, and Combinatorics*, 10(5/6):609–626, 2005. doi:10.25596/jalc-2005-609.
- 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.
- 18 Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981. doi:10.1016/0020-0190(81)90024-7.
- 19 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.
- 20 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Covering problems for partial words and for indeterminate strings. *Theoretical Computer Science*, 698:25–39, 2017. doi:10.1016/j.tcs.2017.05.026.
- 21 Maxime Crochemore, Costas S. Iliopoulos, and Maureen Korda. Two-dimensional prefix string matching and covering on square matrices. *Algorithmica*, 20(4):353–373, 1998. doi:10.1007/PL00009200.
- 22 Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2003. doi:10.1142/4838.
- 23 Patryk Czajka and Jakub Radoszewski. Experimental evaluation of algorithms for computing quasiperiods. *CoRR*, abs/1909.11336, 2019 (accepted to *Theoretical Computer Science*). arXiv:1909.11336.
- 24 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.
- 25 Tomás 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.
- 26 Paweł Gawrychowski, Jakub Radoszewski, and Tatiana A. Starikovskaya. Quasi-periodicity in streams. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019*, volume 128 of *LIPICs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPICs.CPM.2019.22.
- 27 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.
- 28 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 29 Yijie Han. Deterministic sorting in $O(n \log \log n)$ time and linear space. *Journal of Algorithms*, 50(1):96–105, 2004. doi:10.1016/j.jalgor.2003.09.001.
- 30 Danny Hermelin, Dror Rawitz, Romeo Rizzi, and Stéphane Vialette. The minimum substring cover problem. *Information and Computation*, 206(11):1303–1312, 2008. doi:10.1016/j.ic.2008.06.002.
- 31 Costas S. Iliopoulos, Christos Makris, Yannis Panagis, Katerina Perdikuri, Evangelos Theodoridis, and Athanasios K. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundamenta Informaticae*, 71(2-3):259–277, 2006. URL: <http://content.iospress.com/articles/fundamenta-informaticae/fi71-2-3-07>.
- 32 Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina Perdikuri, William F. Smyth, and Athanasios K. Tsakalidis. String regularities with don't cares. *Nordic Journal on Computing*, 10(1):40–51, 2003.

- 33 Costas S. Iliopoulos, Dennis W. G. Moore, and Kunsoo Park. Covering a string. *Algorithmica*, 16(3):288–297, 1996. doi:10.1007/BF01955677.
- 34 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.
- 35 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018. URL: <https://mimuw.edu.pl/~kociumaka/files/phd.pdf>.
- 36 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear-time algorithm for seeds computation. *ACM Transactions on Algorithms*, 16(2):Article 27, 2020. doi:10.1145/3386369.
- 37 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for seeds computation. In Yuval Rabani, editor, *23rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012*, pages 1095–1112. SIAM, 2012. doi:10.1137/1.9781611973099.
- 38 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. doi:10.1016/j.tcs.2016.11.035.
- 39 Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Fast algorithm for partial covers in words. *Algorithmica*, 73(1):217–233, 2015. doi:10.1007/s00453-014-9915-3.
- 40 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.
- 41 Yin Li and William F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002. doi:10.1007/s00453-001-0062-2.
- 42 Michael G. Main and Richard J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984. doi:10.1016/0196-6774(84)90021-X.
- 43 Dennis Moore and W. F. Smyth. Computing the covers of a string in linear time. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '94*, page 511–515, USA, 1994. Society for Industrial and Applied Mathematics.
- 44 Dennis W. G. Moore and William F. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994. doi:10.1016/0020-0190(94)00045-X.
- 45 Dennis W. G. 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, 1995. doi:10.1016/0020-0190(94)00235-Q.
- 46 Jean Néraud. Elementariness of a finite set of words is co-NP-complete. *RAIRO Theoretical Informatics and Applications*, 24:459–470, 1990. doi:10.1051/ita/1990240504591.
- 47 Alexandru Popa and Andrei Tanasescu. An output-sensitive algorithm for the minimization of 2-dimensional string covers. In T. V. Gopal and Junzo Watada, editors, *Theory and Applications of Models of Computation - 15th Annual Conference, TAMC 2019*, volume 11436 of *Lecture Notes in Computer Science*, pages 536–549. Springer, 2019. doi:10.1007/978-3-030-14812-6_33.
- 48 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.
- 49 Hui Zhang, Qing Guo, and Costas S. Iliopoulos. Algorithms for computing the lambda-regularities in strings. *Fundamenta Informaticae*, 84(1):33–49, 2008. URL: <http://content.iiospress.com/articles/fundamenta-informaticae/fi84-1-04>.

A Supplementary Figure

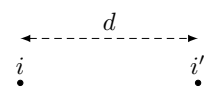
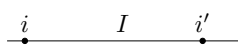
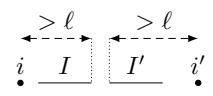
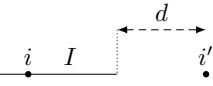
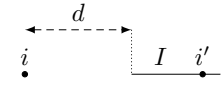
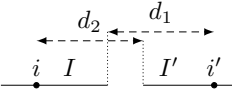
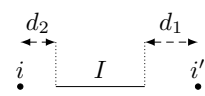
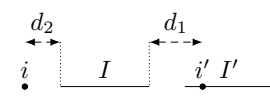
 <p>YES $\Leftrightarrow d \leq \ell$</p>	 <p>YES</p>	 <p>NO</p>
 <p>$a \geq d$</p>	 <p>$\ell - a \geq d$</p>	 <p>$(a \geq d_1) \vee (\ell - a \geq d_2)$</p>
 <p>$(a \geq d_1) \wedge (\ell - a \geq d_2)$</p>	 <p>$(a \geq d_1) \wedge (\ell - a \geq d_2)$</p>	

Figure 5 Sets of constraints $C(i, i')$ generated depending on the interactions with intervals $I, I' \in \mathcal{A}$. The respective rows correspond to items (a)–(c).



Shortest covers of all cyclic shifts of a string

Maxime Crochemore^a, Costas S. Iliopoulos^a, Jakub Radoszewski^{b,*},
Wojciech Rytter^b, Juliusz Straszynski^{b,1}, Tomasz Waleń^{b,1}, Wiktor Zuba^{b,1}

^a Department of Informatics, King's College London, London, UK

^b Institute of Informatics, University of Warsaw, Warsaw, Poland

ARTICLE INFO

Article history:

Received 30 June 2020

Received in revised form 20 February 2021

Accepted 4 March 2021

Available online 17 March 2021

Keywords:

Cover

Quasiperiod

Seed

Fibonacci string

ABSTRACT

A factor C of a string S is called a cover of S , if each position of S is contained in an occurrence of C . Breslauer (1992) [3] proposed a well-known $\mathcal{O}(n)$ -time algorithm that computes the shortest cover of every prefix of a string of length n . We show an $\mathcal{O}(n \log n)$ -time and $\mathcal{O}(n)$ -space algorithm that computes the shortest cover of every cyclic shift of a string of length n and an $\mathcal{O}(n)$ -time algorithm that computes the shortest among these covers. We also provide a combinatorial characterization of shortest covers of cyclic shifts of Fibonacci strings that leads to efficient algorithms for computing these covers.

We further consider the bound on the number of different lengths of shortest covers of cyclic shifts of the same string of length n . We show that this number is $\Theta(\log n)$ for Fibonacci strings.

© 2021 Elsevier B.V. All rights reserved.

1. Introduction

We consider strings as finite sequences of letters drawn from an alphabet $\Sigma = [0, n^{\mathcal{O}(1)}]$, often referred to as an integer alphabet [1]. The notion of periodicity in strings and its many variants have been well-studied in many fields like combinatorics on words, pattern matching, data compression, automata theory, formal language theory, and molecular biology. A typical regularity, the *period* U of a given string S , grasps the repetitiveness of S since S is a prefix of a string constructed by concatenations of U . If $S = AWB$, for some, possibly empty, strings A, W, B , then W is called a *factor* of S and, respectively, S is a superstring of W . A factor C of S is called a *cover* of S , if each position of S is contained in an occurrence of C . A factor C of S is called a *seed* of S , if there exists a superstring of S which is constructed by concatenations and superpositions of C . In other words, C is a seed of S if S is covered by occurrences and left and right overhangs of C . For example, abc is a period of $abcabcabca$, $abca$ is a cover of $abcabcaabca$, and $abca$ is a seed of $bcabcaabc$. The notions “cover” and “seed” are generalizations of periods in the sense that superpositions as well as concatenations are considered to define them, whereas only concatenations are considered for periods.

In computation of covers, two problems have been considered in the literature. The shortest-cover problem (also known as the superprimitivity test) is that of computing the shortest cover of a given string of length n , and the all-covers problem is that of computing all the covers of a given string. Apostolico et al. [2] introduced the notion of covers and gave a linear-time algorithm for the shortest-cover problem. Breslauer [3] proposed an on-line algorithm for computing the shortest cover that works in linear time. In particular, his algorithm computes the shortest cover of every prefix of a string. The

* Corresponding author.

E-mail addresses: maxime.crochemore@kcl.ac.uk (M. Crochemore), c.iliopoulos@kcl.ac.uk (C.S. Iliopoulos), jrad@mimuw.edu.pl (J. Radoszewski), rytter@mimuw.edu.pl (W. Rytter), jks@mimuw.edu.pl (J. Straszynski), walen@mimuw.edu.pl (T. Waleń), w.zuba@mimuw.edu.pl (W. Zuba).

¹ Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

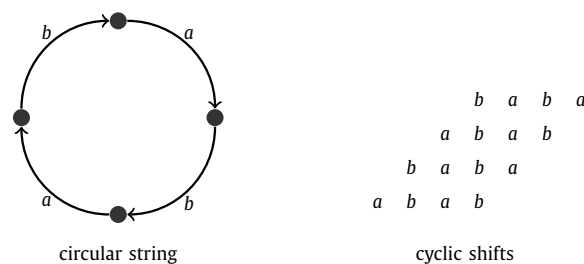


Fig. 1. The string aba is a cover of the string $S = abab$ treated as a single circular string, but is not a cover of any of cyclic shifts of S .

other direction was taken by Moore and Smyth [4,5] and by Li and Smyth [6] who computed all the covers of a string and a representation of all the covers of all prefixes of a string, respectively. A circular string S' corresponding to a given string S is formed by concatenating the first letter of S to the right of its last letter. Covers of circular strings were also considered. It is implicit in [7] that covers of a circular string S are exactly seeds of S^2 . Covers and seeds of Fibonacci strings were studied in [8], whereas covers of circular Fibonacci strings were considered in [9].

All the seeds of a string of length n can be represented in $\mathcal{O}(n)$ space as a collection of a linear number of disjoint paths in the suffix trees of the string and of its reversal. This representation can be computed in $\mathcal{O}(n \log n)$ time [7] and even in $\mathcal{O}(n)$ -time [10]. Recently it was also shown in [11] that all the seeds can also be represented as a linear number of disjoint paths in just the suffix tree of the string. This implies the following fact:

Lemma 1. *The problem of computing the shortest cover of a circular string can be solved in linear time.*

We say that a string Y is a *cyclic shift* of a string X if $X = AB$ and $Y = BA$ for some strings A and B ; in this case we also write $Y = \text{rot}_{|A|}(X)$. It seems that the problem of computing shortest covers of all cyclic shifts of a string is harder than that of computing the shortest cover of a circular string. A straightforward application of any of the aforementioned algorithms for computing covers of a string yields an $\mathcal{O}(n^2)$ -time solution to the problem. One should note that covers of circular strings are a different notion than that of covers of cyclic shifts of a string; see Fig. 1.

The shortest covers of cyclic shifts of a string can behave rather irregularly. For example, the length of the shortest cover of $S = abaabababababababa$ equals 3, whereas the shortest cover of $\text{rot}_1(S)$ has length 18.

We consider the following problem.

SHORTEST COVERS OF ALL CYCLIC SHIFTS OF A STRING
Input: A string S of length n .
Output: The lengths of the shortest covers of all cyclic shifts of S .

Let S be a string of length n and $\text{ShCov}(S)$ denote the shortest cover of S . We introduce an array \mathbf{CC}_S of length n such that $\mathbf{CC}_S[i] = |\text{ShCov}(\text{rot}_i(S))|$. Our main result is computing this array. We also denote

$$\mathbf{CCSet}(S) = \{\mathbf{CC}_S[i] : i = 0, \dots, n - 1\}.$$

Example 1. For the Fibonacci strings $S_1 = abaab$, $S_2 = abaababaabaab$ we have:

$$\mathbf{CC}_{S_1} = [5, 5, 5, 3, 5], \mathbf{CC}_{S_2} = [5, 5, 13, 3, \dots]$$

$$\mathbf{CCSet}(S_1) = \{3, 5\}, \mathbf{CCSet}(S_2) = \{3, 5, 8, 13\}.$$

Our results. We show that the whole array \mathbf{CC}_S and $\min_i \mathbf{CC}_S[i]$ for a string S of length n can be computed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ time, respectively, and $\mathcal{O}(n)$ space. For this we use a characterization of covers of cyclic shifts of a string by seeds and squares, i.e., strings of the form W^2 , and the suffix tree data structure.

We give a simple recursive formula for computing \mathbf{CC}_S for a Fibonacci string S . It implies a linear-time algorithm for computing this array as a whole and can be used to devise time and space efficient algorithms for computing subsequent elements of this array. We also show that for the family of Fibonacci strings we have $|\mathbf{CCSet}(S)| = \Theta(\log |S|)$.

Structure of the paper. In Section 2 we recall basic definitions and illustrate them by showing a linear-time algorithm that solves a similar problem to the one in scope, that is, computing the shortest periods of all cyclic shifts of a string. Then in Section 3 we present characterizations of shortest covers of cyclic shifts of a string, which lead us to the main algorithmic results in Section 4. Shortest covers of cyclic shifts of Fibonacci strings are studied in Section 5. We conclude and mention some open problems in Section 6.

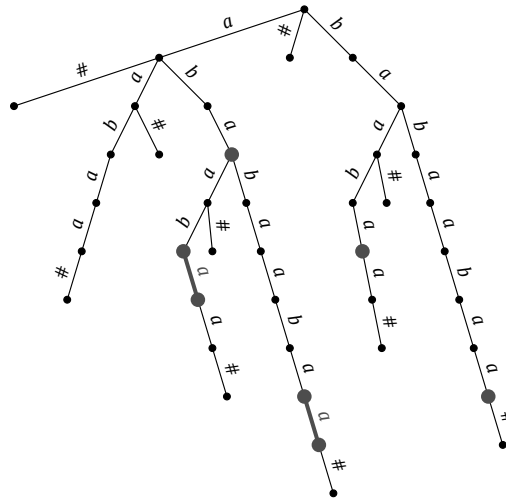


Fig. 2. The string $S = ababaabaa$ has the following seeds: $aba, abaab, baaba, abaaba, ababaaba, babaabaa, ababaabaa$. They can be represented on the (uncompressed) suffix tree of S as shown in the figure. Each seed is a path from root to marked node. In some cases, e.g. $abaab, abaaba$, multiple seeds are represented on a single path.

This is a full version of the paper [12]. In particular, compared to the conference version, it contains a much more precise characterization of shortest covers of cyclic shifts of Fibonacci strings.

2. Preliminaries

We assume that positions of a string S are numbered 0 through $|S| - 1$, $S = S[0] \dots S[|S| - 1]$. By $S[i..j]$ we denote a factor of S equal to $S[i] \dots S[j]$. A factor is called a prefix if $i = 0$ and a suffix if $j = |S| - 1$. A factor that occurs both as a prefix and as a suffix of S is called a border of S . A positive integer p is a period of S if $S[i] = S[i + p]$ for all $i = 0, \dots, |S| - p - 1$.

2.1. Suffix trees

Recall that a *suffix tree* of a string S is a compact tree of all the suffixes of $S\#$, where $\#$ is a special end marker. The root, branching nodes (i.e., nodes with more than one child), and leaves of the tree are *explicit*. All the remaining nodes are *implicit* in the tree. Each leaf is labeled with the starting position of the corresponding suffix. Every factor of S is represented as an explicit or implicit node of the tree. A suffix tree of a string of length n over an integer alphabet can be constructed in $\mathcal{O}(n)$ time [1].

Observation 1. Let S be a string of length n . After $\mathcal{O}(n)$ -time preprocessing, all the occurrences of a factor of S , represented as a node in the suffix tree of S , can be reported in linear time w.r.t. the number of these occurrences.

Proof. It suffices to store a list L of leaves of the suffix tree in a left-to-right order. Then for every explicit node v of the tree, we precompute the endpoints of the sublist of L that corresponds to the occurrences of the string v . This precomputation is done bottom-up in $\mathcal{O}(n)$ time. □

We also use the following lemma.

Lemma 2 ([11]). Given a collection of factors U_1, \dots, U_k of a string S of length n , each represented by an occurrence in S , in $\mathcal{O}(n + k)$ time we can compute the implicit or explicit node in the suffix tree of S that corresponds to each factor U_i . Moreover, all these nodes can be made explicit in $\mathcal{O}(n + k)$ time.

The set (possibly of a quadratic size) of all seeds of a string can be represented as a collection of linearly many disjoint paths in the suffix tree [11]. It can be assumed that each path belongs to a single edge of the suffix tree. The endpoints of the paths can be implicit nodes. For an example, see Fig. 2.

2.2. Runs and squares

A string X is called *primitive* if $X = Y^k$ for positive integer k implies that $k = 1$. A string of the form Z^2 is called a *square*; it is called *primitively rooted* if Z is primitive.

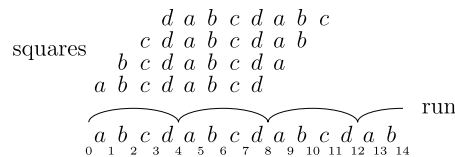


Fig. 3. Primitive squares can be derived from runs, knowing the shortest periods of runs. The leftmost square $(abcd)^2$ has its center at interposition 4.

i	$rot_i(S)$	min. period
0	abababa	2
1	bababaa	7
2	ababaab	5
3	babaaba	5
4	abaabab	5
5	baababa	5
6	aababab	7

Fig. 4. Smallest periods of all cyclic shifts of string $S = abababa$.

A run (also called a maximal repetition) in a string S is a triple (a, b, p) such that $S[a..b]$ has period p , $2p \leq b - a + 1$ and the interval $[a, b]$ cannot be extended to the left nor to the right without violating the above property, that is, $S[a - 1] \neq S[a + p - 1]$ and $S[b - p + 1] \neq S[b + 1]$, provided that the respective positions exist. The exponent of a run (a, b, p) is defined as $\frac{b-a+1}{p}$. A string of length n has $\mathcal{O}(n)$ runs and they can all be computed in $\mathcal{O}(n)$ time [13,14].

From a run (a, b, p) we can produce all triples (a, b, kp) for integer $k \geq 1$ such that $2kp \leq b - a + 1$; we call such triples generalized runs. That is, the period of a generalized run need not be the shortest period. The number of generalized runs is also $\mathcal{O}(n)$ as the sum of exponents of runs is $\mathcal{O}(n)$ [13,14].

We say that a square factor $S[i..i + 2\ell - 1]$ in S is induced by a generalized run (a, b, p) if $\ell = p$ and $[i, i + 2\ell - 1] \subseteq [a, b]$. A square factor is induced by exactly one generalized run and a primitively rooted square factor is induced by exactly one run [15]; see also Fig. 3.

An interposition i in S , for $i = 1, \dots, |S| - 1$, is a delimiter between positions i and $i - 1$. Moreover, interposition 0 precedes the first letter of S and interposition $|S|$ follows the last letter. Thus a string S has $|S| + 1$ interpositions. We use interpositions to describe centers of square factors; see Fig. 3.

As an illustration of these notions, we show below that smallest periods of all cyclic shifts of a string can be computed in $\mathcal{O}(n)$ time. See also Fig. 4.

SMALLEST PERIODS OF ALL CYCLIC SHIFTS OF A STRING

Input: A string S of length n .

Output: The lengths of the smallest periods of all cyclic shifts of S .

Proposition 1. *Smallest periods of all cyclic shifts of a string of length n can be computed in $\mathcal{O}(n)$ time.*

Proof. Let S be a string of length n . We will show how to compute the smallest periods of strings of the form $rot_i(S)$ from the longest squares in the string $W = S^3$. Let $f_{W,k}(i)$ be the half length of the longest square with the center at the interposition i in W and half length smaller than k :

$$f_{W,k}(i) = \max\{j \in [0, k - 1] : W[i - j..i - 1] = W[i..i + j - 1]\}.$$

We can observe that $f_{W,n}(n + i)$ is the longest border of $rot_i(S)$ (see Fig. 5), so the smallest period of $rot_i(S)$ is $n - f_{W,n}(n + i)$.

In [15] it was shown how to compute the shortest square centered at each interposition of a string in linear time from the runs in the string. The solution used a min-variant of a so-called Manhattan Skyline Problem. The array $f_{W,n}$ can be computed using a max-variant of the problem, stated formally below.

MAX-VARIANT OF MANHATTAN SKYLINE PROBLEM

Input: A set \mathcal{I} of $\mathcal{O}(n)$ subintervals of $[0, 3n]$ with natural heights of size $\mathcal{O}(n)$

Output: The table $f[t] = \max\{\text{height}([i, j]) : t \in [i, j], [i, j] \in \mathcal{I}, t \in [0, 3n]\}$.

The solution to the problem is obtained in the same way as it was shown in [15] for the min-variant.

Claim 1 (See [15, Lemma 16]). *The Max-Variant of the Manhattan Skyline Problem can be solved in linear time.*

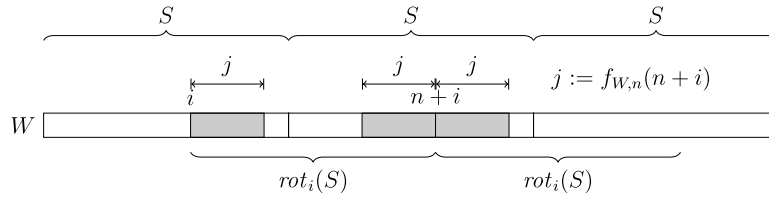


Fig. 5. Relation between $f_{W,n}(n+i)$ and the smallest period of $rot_i(S)$.

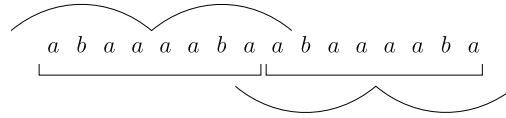


Fig. 6. Illustration of Example 2.

In our case subintervals correspond to generalized runs in S , since each generalized run (a, b, p) induces squares of half length p with centers at interpositions $\{a + p, \dots, b - p + 1\}$. \square

Remark 1. In the above proof, taking $W = S^3$ (and not, say, $W = S^2$) is necessary when $f_{W,n}(n+i)$ is almost n .

3. Covers of cyclic shifts

We denote by $\frac{1}{2}$ -Squares(S) (square halves) the set of factors Z of S such that the square Z^2 is also a factor of S and by $\frac{1}{2}$ -PSquares(S) the subset of $\frac{1}{2}$ -Squares(S) that consists only of primitive strings. We further denote by Seeds(S) the set of factors which are seeds of S . We use these sets for the string S^3 in order to characterize covers of all cyclic shifts of S .

Lemma 3. Let S be a string of length n and C be a string of length up to n . Then C is a cover of $rot_i(S)$ if and only if $C \in Seeds(S^3) \cap \frac{1}{2}$ -Squares(S^3) and C^2 occurs with its center at interposition $j \equiv i \pmod{n}$ in S^3 .

Moreover, if C is the shortest cover of $rot_i(S)$, then $C \in Seeds(S^3) \cap \frac{1}{2}$ -PSquares(S^3).

Proof.

(\Rightarrow) String C is a cover of $(rot_i(S))^4$, and thus a seed of its factor S^3 . Moreover, $S^3[j - |C| \dots j + |C| - 1]$, that is, the factor of S^3 of length $2|C|$ with center at interposition j , is equal to C^2 for $j = i + n$.

(\Leftarrow) The square C^2 occurs in S^3 with its center at interposition $j \equiv i \pmod{n}$. Thus C is a border of $rot_j(S) = rot_i(S)$ as $|C| < n$. C is also a seed of $rot_i(S)$ which is a factor of S^3 , hence it is a cover of $rot_i(S)$.

As for the “moreover” part, it suffices to note that the shortest cover of a string is obviously primitive. \square

Example 2. In the above lemma, one could not take S^2 instead of S^3 . Indeed, for $S = abaaaaba$ we have that $rot_4(S) = aabaaba$ has the shortest cover $aabaa$, but $S^2 = abaaaabaabaaaaba$ does not contain the square $(aaba)^2$; see Fig. 6.

Let $T(S^3)$ be the suffix tree of S^3 in which we distinguish the nodes v corresponding to strings Z^2 for $Z \in Seeds(S^3) \cap \frac{1}{2}$ -PSquares(S^3). These nodes are called *candidate nodes*. Some of these nodes could be implicit nodes in the suffix tree. Then they are made explicit. Denote by $CandAnc(v)$ the set of ancestor nodes of v in $T(S^3)$ which are candidate nodes. Let $|v|$ be the length of the string corresponding to the node v .

We can reformulate Lemma 3 as follows; see also Fig. 7.

Lemma 4. $CC_S[i]$, i.e., the length of the shortest cover of $rot_i(S)$, equals

$$\min_{j,v} \{k : k = |v|/2, i = (j + k) \pmod{n}, j \in Leaves(T(S^3)), v \in CandAnc(j)\}.$$

Clearly if C is a cover of $rot_i(S)$, then C is a cover of S treated as a circular string. As we have already noted in Fig. 1, the converse is not necessarily true. However, we show that every shortest cover of the circular string S is a cover of the corresponding cyclic shift of S .

Lemma 5. A shortest cover of a circular string is always a (shortest) cover of some cyclic shift.

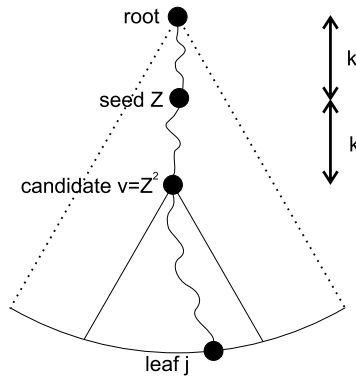


Fig. 7. Illustration of Lemma 4. The situation when $CC_S[i] = k$. We have that $i = (j + k) \bmod n$ and Z^2 is a primitively rooted square of length $2k$; it corresponds to the node v which is possibly inside an edge of the suffix tree.

Proof. We need the following claim.

Claim 2 (See [7]). *String C is a cover of S considered as a circular string iff it is a seed of S^2 , hence also iff it is a seed of S^3 .*

Let C be a cover of a circular string S , such that C^2 does not occur in it. Consider the last position covered by any occurrence of C in the string.

The position must be also covered by another occurrence of C (the next position must be covered and cannot be the first position of some C). Thus by erasing the last position of C we obtain a shorter cover. Hence if C is a shortest cover then C^2 must appear in the circular string S .

By Lemma 3, C is a cover of some cyclic shift of S . \square

By computing the shortest cover of the circular string S using Lemma 1 we obtain the following preliminary result.

Corollary 1. *For a string S of length n , $\min CC_S$ can be computed in $\mathcal{O}(n)$ time.*

4. Main algorithm

First we have to show how to compute efficiently the tree $T(S^3)$. We denote by $OccPSquares(S)$ the set of all occurrences of primitively rooted squares in S . Each occurrence is represented in $\mathcal{O}(1)$ space as a factor of S . A direct consequence of the Three-square-prefix Lemma, see [16], is that a string of length n has no more than $\log n$ prefixes that are primitively rooted squares.

Lemma 6 ([16]). *For a string S of length n , $|OccPSquares(S)| = \mathcal{O}(n \log n)$.*

Lemma 7. *For a string S of length n , $|\frac{1}{2}\text{-PSquares}(S)| = \mathcal{O}(n)$ and this set can be computed in $\mathcal{O}(n)$ time.*

Proof. Let us start with efficient computation of squares.

Claim 3 ([17,15,18–20]). *For a string S of length n , we have $|\frac{1}{2}\text{-Squares}(S)| = \mathcal{O}(n)$ and this set can be computed in $\mathcal{O}(n)$ time.*

By the claim, $|\frac{1}{2}\text{-PSquares}(S)| = \mathcal{O}(n)$ since $\frac{1}{2}\text{-PSquares}(S) \subseteq \frac{1}{2}\text{-Squares}(S)$.

The set $\frac{1}{2}\text{-PSquares}(S)$ can be computed by filtering out the factors from $\frac{1}{2}\text{-Squares}(S)$ that are not primitive. This can be done in $\mathcal{O}(1)$ time per factor after $\mathcal{O}(n)$ -space and time preprocessing using so-called Two-Period queries [14,21]. A more direct approach would be to (effortlessly) adapt the algorithm for computing different square factors from [15] using relations between primitive squares and runs (maximal repetitions); see Fig. 3. \square

Lemma 8. *The tree $T(S^3)$ can be computed in $\mathcal{O}(n)$ time.*

Proof. We use a version of a minimal augmented suffix tree (MAST, in short), a data structure that was initially introduced in [22].

Let us recall that Lemma 2 can be used to augment the suffix tree of S^3 with nodes that correspond to a set of factors of S^3 . We first apply the lemma to the collection of factors $\frac{1}{2}\text{-PSquares}(S^3)$, which can be efficiently computed due to the previous lemma.

Fib₂ = ab

<i>i</i>	cyclic shift	shortest cover	CF ₂ [<i>i</i>]
0	ab	ab	2
1	ba	ba	2

Fib₃ = aba

0	aba	aba	3
1	baa	baa	3
2	aab	aab	3

Fib₄ = abaab

0	abaab	abaab	5
1	baaba	baaba	5
2	aabab	aabab	5
3	ababa	aba	3
4	babaa	babaa	5

Fib₅ = abaababa

<i>i</i>	cyclic shift	shortest cover	CF ₅ [<i>i</i>]
0	abaababa	aba	3
1	baababaa	baababaa	8
2	aababaab	aababaab	8
3	ababaaba	aba	3
4	babaabaa	babaabaa	8
5	abaabaab	abaab	5
6	baabaaba	baaba	5
7	aabaabab	aabaabab	8

Fib₆ = abaababaabaab

<i>i</i>	Fibonacci representation of <i>i</i> + 1	cyclic shift	shortest cover	CF ₆ [<i>i</i>]
0	000001	abaababaabaab	abaab	5
1	000010	baababaabaaba	baaba	5
2	000100	aababaabaabab	aababaabaabab	13
3	000101	ababaabaababa	aba	3
4	001000	babaabaababaa	babaabaababaa	13
5	001001	abaabaababaab	abaab	5
6	001010	baabaababaaba	baaba	5
7	010000	aabaababaabab	aabaababaabab	13
8	010001	abaababaababa	aba	3
9	010010	baababaababaa	baababaa	8
10	010100	aababaababaab	aababaab	8
11	010101	ababaababaaba	aba	3
12	100000	babaababaabaa	babaababaabaa	13

Fig. 8. Shortest covers of cyclic shifts of Fibonacci strings.

Lemma 10. Let $n \geq 2$.

- (a) If S is a seed of Fib_n^3 and $|S| \leq F_{n-1}$, then S is a seed of Fib_{n+1}^3 .
- (b) $rot_k(Fib_n)$ is a seed of Fib_{n+1}^3 for $0 \leq k \leq F_{n-1} - 2$.

Proof. (a) We will show that S is a cover of a factor U of Fib_{n+1}^2 of length at least F_{n+1} . This is sufficient to prove that S is a seed of Fib_{n+1}^3 .

By Observation 2, we have that $lcp(Fib_n^3, Fib_{n+1}^2) = F_{n+2} - 2$; see Fig. 9. Let X be the set of occurrences of S in Fib_n^3 that start at the first $2F_n$ positions. Those occurrences cover the string $U = Fib_n^3[\min(X) .. \max(X) + |S| - 1]$. Let us note that the first occurrence of S in the third Fib_n is at position $2F_n + \min(X)$ (possibly it is an overhang), so the position $2F_n + \min(X) - 1$ must be covered by the occurrence at position $\max(X)$. Therefore

$$\max(X) \geq 2F_n + \min(X) - |S|, \quad \text{hence} \quad |U| = \max(X) + |S| - \min(X) \geq 2F_n > F_{n+1}. \tag{2}$$

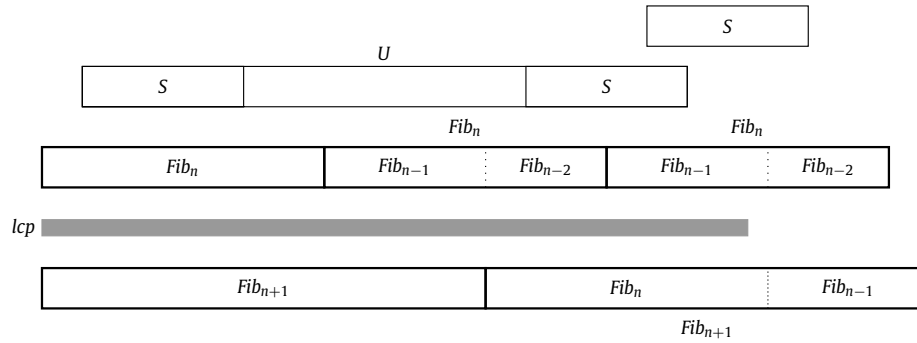


Fig. 9. $\text{lcp}(\text{Fib}_n^3, \text{Fib}_{n+1}^2) = F_{n+2} - 2$. Factor U and the first occurrence of S in Fib_n^3 outside its prefix Fib_n^2 are shown.

If $\max(X) + |S| - 1 < F_{n+2} - 2$, then U is a factor of Fib_{n+1}^2 , which concludes the proof. Otherwise $|S| = F_{n-1}$ and $\max(X) = 2F_n - 1$; in this case it can be readily verified that S is not a seed of Fib_n^3 .

(b) The string $S = \text{rot}_k(\text{Fib}_n)$ occurs at positions k and $F_n + k$ in Fib_n^3 , covering a factor of length $2F_n > F_{n+1}$. It ends at position $2F_n + k - 1 < F_{n+2} - 2$, so it occurs also in Fib_{n+1}^2 . Hence, S is a seed of Fib_{n+1}^3 . \square

Example 3. The upper bound on k in Lemma 10(b) is tight. E.g., $\text{Fib}_3 = aba$ is a seed of Fib_3^3 and of $\text{Fib}_4^3 = abaababaababaab$, whereas $\text{rot}_1(\text{Fib}_3) = baa$ is a seed of Fib_3^3 but not of Fib_4^3 .

For a sequence of numbers X let us denote by X^+ the sequence X with elements F_n changed to F_{n+1} .

Theorem 2. For $n \geq 4$ we have:

- (a) $\mathbf{CF}_{n+1} = C^+ B^+ C^+ A B^+$, where $\mathbf{CF}_n = ABC$, $|B| = F_{n-3}$, $|A| = |C| = F_{n-2}$.
- (b) Let S_n denote the prefix of \mathbf{CF}_n of length $F_{n-1} - 1$. Then

$$\mathbf{CF}_n = S_{n-2} F_n S_{n-3} F_n S_{n-2} F_n S_{n-1} F_n.$$

Proof. (a) The proof goes by induction. Two additional conditions are included in the statement:

- A and B do not contain F_{n-1} (this immediately follows from the fact that they are constructed from blocks with a $+$);
- F_n occurs in \mathbf{CF}_n only four times, at positions $F_{n-3} - 1, F_{n-2} - 1, F_{n-1} - 1, F_n - 1$ (that is, at ends of blocks A, B, C and once in the middle of the block A).

We use the following crucial observation that immediately follows from the characterization of covers of cyclic shifts (Lemma 3) and Lemma 10(a).

Observation 3. If $x = \mathbf{CF}_n[i]$ with $x \neq F_n$ and a string of length $2x$ with its center at interposition i in Fib_n (seen as a circular string) is the same as a string of length $2x$ with its center at interposition j in Fib_{n+1} (also circular), then $\mathbf{CF}_{n+1}[j] = x$.

Let us factorize strings Fib_n and Fib_{n+1} into named blocks, as shown in Fig. 10. We see that:

- Y_1 has the same surrounding of length $F_{n-1} - 2$ as X_3 ($\text{lcp}(\text{Fib}_{n-1}, \text{Fib}_{n-3}\text{Fib}_{n-2}) = F_{n-1} - 2$ by Observation 2).
- Y_2 has the same surrounding of length F_{n-2} as X_2 (Fib_{n-2} ends with a different letter than Fib_{n-3}).
- Y_3 has the same surrounding of length F_{n-1} as X_3 (Fib_{n-2} ends with a different letter than Fib_{n-3}).
- Y_4 has the same surrounding of length at least F_n as X_1 ($\text{Fib}_{n-3}\text{Fib}_{n-2}$ starts with Fib_{n-2}).
- Y_5 has the same surrounding of length $F_n - 2$ as X_2 (again by Observation 2).

For each of the blocks Y_3, Y_4, Y_5 , the surrounding of length at least F_{n-1} is the same as for the corresponding X -block. By the observation, for each of these blocks, the parts of \mathbf{CF}_{n+1} contain the same values $\leq F_{n-1}$ in the same places as their

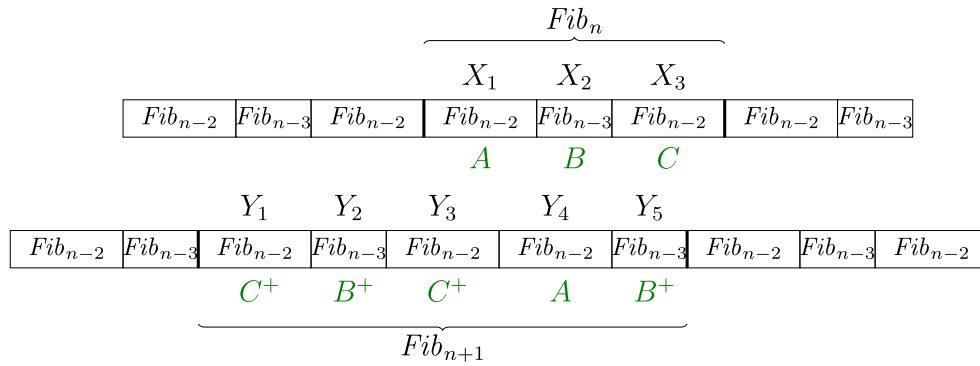


Fig. 10. Factorizations of Fib_n and Fib_{n+1} .

corresponding X -blocks. The same also holds for blocks Y_1 and Y_2 ; in case of Y_1 we use the fact that the last position of C contains F_n , and in case of Y_2 that B does not contain F_{n-1} .

Now it remains to take care of the last six positions of \mathbf{CF}_{n+1} , the ones obtained from the four positions of \mathbf{CF}_n that are equal to F_n . By Lemma 9 each of those positions has their \mathbf{CF}_{n+1} value equal to either F_n or F_{n+1} (Lemma 10(a) works in both ways). For positions $F_{n-2} - 1, F_{n-1} - 1, F_n - 1, F_{n+1} - 1$ one can check that the strings of length $2F_n$ with their centers at the corresponding interpositions are not squares. For example, for position $F_{n-2} - 1$ the string in question equals

$$a\text{Fib}_{n-2}\text{Fib}_{n-3}\text{Fib}'_{n-2} \quad a\text{Fib}_{n-3}\text{Fib}_{n-2}\text{Fib}'_{n-2},$$

where a is the last letter of Fib_{n-2} and Fib'_n is Fib_n with the last letter removed; we have $\text{Fib}_{n-2}\text{Fib}_{n-3} \neq \text{Fib}_{n-3}\text{Fib}_{n-2}$ by Observation 2. Hence for those positions \mathbf{CF}_{n+1} contains values F_{n+1} .

However, for positions $F_n + F_{n-3} - 1$ and $F_n + F_{n-2} - 1$ such strings match and are equal to $\text{Fib}_{n-4}\text{Fib}_{n-3}\text{Fib}_{n-1}$ shifted to the right by one and to $\text{Fib}_{n-3}\text{Fib}_{n-2}\text{Fib}_{n-1}$ shifted to the right by one, respectively. For example,

$$\text{rot}_{F_{n-3}}(\text{Fib}_n) = \text{rot}_{F_{n-3}}(\text{Fib}_{n-3}\text{Fib}_{n-4}\text{Fib}_{n-3}\text{Fib}_{n-3}\text{Fib}_{n-4}) = \text{Fib}_{n-4}\text{Fib}_{n-3}\text{Fib}_{n-1}. \tag{3}$$

By Lemma 10(b) these strings are seeds of Fib_{n+1}^3 , hence \mathbf{CF}_{n+1} contains F_n at these positions.

(b) We prove by induction that $A = S_{n-2}F_nS_{n-3}, B = S_{n-2}F_n, C = S_{n-1}F_n$.

C is obtained from AB from the previous step by changing the last element from F_{n-1} to F_n . This AB is a prefix of \mathbf{CF}_{n-1} of length $F_{n-2} = |S_{n-1}| + 1$.

B is obtained from C from the previous step which is obtained from AB for $n - 2$ again by first changing F_{n-2} into F_{n-1} and then to F_n .

A is obtained from CB from the previous step; we again have the same construction. \square

The recursive characterizations of Theorem 2 easily imply efficient algorithms for computing the \mathbf{CF} array as well as its subsequent elements.

Theorem 3.

- (a) \mathbf{CF}_n can be computed in linear ($\mathcal{O}(F_n)$) time.
- (b) Given n and k , we can compute $\mathbf{CF}_n[k]$ in $\mathcal{O}(n)$ time and $\mathcal{O}(1)$ space.

Proof. The first point follows directly from the previous theorem.

For a proof of (b), we use the following algorithm ComputeCF that implements the formula of Theorem 2(b). \square

Corollary 2. We have $\mathbf{CCSet}(\text{Fib}_n) = \{F_3, \dots, F_n\}$. More precisely, for $n \geq 4$, \mathbf{CF}_n contains 4 occurrences of F_n , F_{n-3} occurrences of 3 and $2F_{n-k}$ occurrences of F_k for $3 < k < n$.

Proof. From Theorem 2(b) we have that $S_n = S_{n-2}F_nS_{n-3}F_nS_{n-2}$. It is enough to prove by induction that S_n and S_{n-1} contain together F_{n-3} occurrences of 3 and $2F_{n-k}$ occurrences of F_k for $3 < k \leq n$. The base cases for $n = 4, 5$ are illustrated by Fig. 8. The content of $S_n + S_{n-1}$ is equal to the contents of $(S_{n-2} + S_{n-3} + S_{n-2}) + S_{n-1} = (S_{n-2} + S_{n-3}) + (S_{n-1} + S_{n-2})$ plus two occurrences of F_n . By induction for $n \geq 6$ we have $F_{n-5} + F_{n-4} = F_{n-3}$ occurrences of 3 and $2F_{n-k-2} + 2F_{n-k-1} = 2F_{n-k}$ occurrences of F_k for $3 < k \leq n - 2$. F_{n-1} occurs $2F_1 = 2$ times (both in S_{n-1}). \square

Algorithm 2: ComputeCF(n, k).

```

while  $n \geq 4$  do
  if  $k + 1 \in \{F_{n-3}, F_{n-2}, F_{n-1}, F_n\}$  then
    return  $F_n$ 
  if  $k \geq F_{n-1}$  then  $n := n - 1$ ;  $k := k - F_n$ 
  else if  $k \geq F_{n-2}$  then  $n := n - 2$ ;  $k := k - F_n$ 
  else if  $k \geq F_{n-3}$  then  $n := n - 3$ ;  $k := k - F_n$ 
  else  $n := n - 2$ 
return  $F_n$ 

```

Let $repr_n(k)$ be a Fibonacci representation of $k + 1$ where the most significant (leftmost) digit represents F_n . E.g., for $n = 10$ we have $49 + 1 = 50 = 34 + 13 + 3 = F_8 + F_6 + F_3$, hence $repr_{10}(49) = 0010100100$. Further let $odd(k) = k \bmod 2$ and $even(k) = 1 - odd(k)$.

Theorem 4. Assume that $repr_n(k)$ ends with $0^x 10^y$, where $x, y \geq 0$ are maximal. Then $CF_n[k] = F_l$, where

$$l = \begin{cases} n & \text{if } x + y = n - 1, x \leq 1 \\ odd(x) + y + 3 & \text{if } x + y = n - 1, x > 1 \\ even(x) + y + 3 & \text{otherwise.} \end{cases}$$

Proof. Let us recall the algorithm of Theorem 3(b) and check what happens to $repr_n(k)$ in all the cases:

- If $repr_n(k) = 0^x 10^y$ and $x \in \{0, 1\}$, then $k + 1$ equals F_n or F_{n-1} , hence $l = n$.
- If $repr_n(k) = 0^x 10^y$ and $x \in \{2, 3\}$, then $l = x \bmod 2 + y + 3 = n$ as well.
- If $repr_n(k) = 0^x 10^*$ and $x \in \{1, 2, 3\}$, then $repr_{n'}(k')$ (representation for the reduced n and k) will be equal to 00^* (leading 0's are erased and 1 is changed into 0).
- If $repr_n(k) = 0^x 10^*$ and $x > 3$ then $repr_{n'}(k')$ will be equal to $0^{x-2} 10^*$ (n decreases by 2).

Hence for $repr_n(k) = 0^x 10^y$ two trailing 0's will be erased unless $x \leq 3$, hence in the important step (in which output is produced) $repr_{n'}(k')$ will be equal to $0^{x \bmod 2 + 2} 10^y$, hence n' will be equal to $x \bmod 2 + 2 + 1 + y = l$.

For $repr_n(k) = *10^x 10^y$ in some step it will be changed into $0^{x+1} 10^y$ (reduction to the previous case). \square

Corollary 3. We can output $CF_n[i..j]$ in $\mathcal{O}(n + (j - i))$ time using $\mathcal{O}(\log(j - i))$ working space.

Proof. In $\mathcal{O}(n)$ time we compute $repr_n(i)$ and store the positions of 1's in a sorted linked list starting from the least significant bit (e.g., for $i = 49$ the list contains (3, 6, 8)).

By Theorem 4 we can compute $CF_n[k]$ in constant time using $repr_n(k)$ (the positions of two least significant 1's are stored at the beginning of the list, and this information is sufficient for our purpose).

We can update the list storing $repr_n(k)$ to obtain the list storing $repr_n(k + 1)$ in constant amortized time (the possible extra $\mathcal{O}(n)$ time for small $(j - i)$ is covered in the complexity).

The above gives an algorithm which works in $\mathcal{O}(n)$ space. However, using a simple trick we can reduce it to $\mathcal{O}(\log(j - i))$. Consider the most significant bit on which $repr_n(i)$ and $repr_n(j)$ differ. It is enough to remember only one higher bit (all higher bits will be constant during the entire runtime). Let x be the number for which $repr_n(x)$ is equal to $repr_n(j)$ with all low bits zeroed (bits less important than the one which differs $repr_n(i)$ and $repr_n(j)$). From x to j we will only have up to $\Theta(\log(j - x))$ 1's (not counting the forgotten high bits). For the sequence from i to $x - 1$ we count the new highest differing bit. This bit is $\Theta(\log(x - i))$ and $\max(\log(x - i), \log(j - x)) = \mathcal{O}(\log(j - i))$. \square

6. Conclusions and open problems

Breslauer [3] proposed a linear-time algorithm for computing the shortest cover of every prefix of a string. We have proposed an $\mathcal{O}(n \log n)$ -time algorithm for computing the shortest cover of every cyclic shift of a string. It remains an open problem if these values can be computed in $\mathcal{O}(n)$ time.

$\mathcal{O}(n)$, $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^2)$ -time algorithms for computing the shortest left seed, right seed, and seed, respectively, of all the prefixes of a string are known; see [25,26]. Here left and right seed are notions that are intermediate between cover and seed. It remains an open problem if the shortest left seed, right seed, and seed can be computed efficiently for all the cyclic shifts of a string.

Based on computer experiments we make the following conjecture.

Conjecture 1. For a string S of length n , $|CCSet(S)| = \mathcal{O}(\log n)$.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

- [1] M. Farach, Optimal suffix tree construction with large alphabets, in: 38th Annual Symposium on Foundations of Computer Science, FOCS 1997, IEEE Computer Society, 1997, pp. 137–143, <https://doi.org/10.1109/SFCS.1997.646102>.
- [2] A. Apostolico, M. Farach, C.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](https://doi.org/10.1016/0020-0190(91)90056-N).
- [3] D. 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](https://doi.org/10.1016/0020-0190(92)90111-8).
- [4] D.W.G. Moore, W.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](https://doi.org/10.1016/0020-0190(94)00045-X).
- [5] D.W.G. Moore, W.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](https://doi.org/10.1016/0020-0190(94)00235-Q).
- [6] Y. Li, W.F. Smyth, Computing the cover array in linear time, *Algorithmica* 32 (1) (2002) 95–106, <https://doi.org/10.1007/s00453-001-0062-2>.
- [7] C.S. Iliopoulos, D.W.G. Moore, K. Park, Covering a string, *Algorithmica* 16 (3) (1996) 288–297, <https://doi.org/10.1007/BF01955677>.
- [8] M. Christou, M. Crochemore, C.S. Iliopoulos, Quasiperiodicities in Fibonacci strings, *Ars Comb.* 129 (2016) 211–225.
- [9] C.S. Iliopoulos, D.W.G. Moore, W.F. Smyth, The covers of a circular Fibonacci string, *J. Comb. Math. Comb. Comput.* 26 (1998) 227–236.
- [10] T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, A linear time algorithm for seeds computation, in: Y. Rabani (Ed.), Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, SIAM, 2012, pp. 1095–1112, <https://doi.org/10.1137/1.9781611973099.86>.
- [11] T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, A linear-time algorithm for seeds computation, *ACM Trans. Algorithms* 16 (2) (2020) 27:1–27:23, <https://doi.org/10.1145/3386369>.
- [12] M. Crochemore, C.S. Iliopoulos, J. Radoszewski, W. Rytter, J. Straszyński, T. Waleń, W. Zuba, Shortest covers of all cyclic shifts of a string, in: WALCOM: Algorithms and Computation - 14th International Conference, WALCOM 2020, in: Lecture Notes in Computer Science, vol. 12049, Springer, 2020, pp. 69–80, https://doi.org/10.1007/978-3-030-39881-1_7.
- [13] R.M. Kolpakov, G. Kucherov, Finding maximal repetitions in a word in linear time, in: 40th Annual Symposium on Foundations of Computer Science, FOCS 1999, IEEE Computer Society, 1999, pp. 596–604, <https://doi.org/10.1109/SFCS.1999.814634>.
- [14] H. Bannai, T. I. S. Inenaga, Y. Nakashima, M. Takeda, K. Tsuruta, The “runs” theorem, *SIAM J. Comput.* 46 (5) (2017) 1501–1514, <https://doi.org/10.1137/15M1011032>.
- [15] M. Crochemore, C.S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, Extracting powers and periods in a word from its runs structure, *Theor. Comput. Sci.* 521 (2014) 29–41, <https://doi.org/10.1016/j.tcs.2013.11.018>.
- [16] M. Crochemore, W. Rytter, Squares, cubes, and time-space efficient string searching, *Algorithmica* 13 (5) (1995) 405–425, <https://doi.org/10.1007/BF01190846>.
- [17] H. Bannai, S. Inenaga, D. Köppl, Computing all distinct squares in linear time for integer alphabets, in: 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, in: LIPIcs, vol. 78, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, pp. 22:1–22:18, <https://doi.org/10.4230/LIPIcs.CPM.2017.22>.
- [18] A. Deza, F. Franek, A. Thierry, How many double squares can a string contain?, *Discrete Appl. Math.* 180 (2015) 52–69, <https://doi.org/10.1016/j.dam.2014.08.016>.
- [19] A.S. Fraenkel, J. Simpson, How many squares can a string contain?, *J. Comb. Theory, Ser. A* 82 (1) (1998) 112–120, <https://doi.org/10.1006/jcta.1997.2843>.
- [20] D. Gusfield, J. Stoye, Linear time algorithms for finding and representing all the tandem repeats in a string, *J. Comput. Syst. Sci.* 69 (4) (2004) 525–546, <https://doi.org/10.1016/j.jcss.2004.03.004>.
- [21] T. Kociumaka, J. Radoszewski, W. Rytter, T. Waleń, Internal pattern matching queries in a text and applications, in: P. Indyk (Ed.), Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM, 2015, pp. 532–551, <https://doi.org/10.1137/1.9781611973730.36>.
- [22] A. Apostolico, F.P. Preparata, Data structures and algorithms for the string statistics problem, *Algorithmica* 15 (5) (1996) 481–494, <https://doi.org/10.1007/BF01955046>.
- [23] P. Séébold, *Propriétés combinatoires des mots infinis engendrés par certains morphismes*, Report no. 85-16, LITP, Paris, 1985.
- [24] C.S. Iliopoulos, D.W.G. Moore, W.F. Smyth, A characterization of the squares in a Fibonacci string, *Theor. Comput. Sci.* 172 (1–2) (1997) 281–291, [https://doi.org/10.1016/S0304-3975\(96\)00141-7](https://doi.org/10.1016/S0304-3975(96)00141-7).
- [25] M. Christou, M. Crochemore, O. Guth, C.S. Iliopoulos, S.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>.
- [26] M. Christou, M. Crochemore, C.S. Iliopoulos, M. Kubica, S.P. Pissis, J. Radoszewski, W. Rytter, B. Szreder, T. Waleń, Efficient seed computation revisited, *Theor. Comput. Sci.* 483 (2013) 171–181, <https://doi.org/10.1016/j.tcs.2011.12.078>.

Internal Quasiperiod Queries

Maxime Crochemore¹[0000-0002-6024-1557], Costas S. Iliopoulos¹[0000-0002-2477-1702], Jakub Radoszewski^{2,*}[0000-0002-0067-6401],
Wojciech Rytter²[0000-0002-9162-6724], Juliusz Straszyński^{2,**}[0000-0003-2207-0053], Tomasz Walen^{2,*}[0000-0002-7369-3309], and
Wiktor Zuba^{2,*}[0000-0002-1988-3507]

¹ Department of Informatics, King’s College London, UK,
{maxime.crochemore,c.ilopoulos}@kcl.ac.uk

² Institute of Informatics, University of Warsaw, Poland
{jrad,rytter,jks,walen,w.zuba}@mimuw.edu.pl

Abstract. Internal pattern matching requires one to answer queries about factors of a given string. Many results are known on answering internal period queries, asking for the periods of a given factor. In this paper we investigate (for the first time) internal queries asking for covers (also known as quasiperiods) of a given factor. We propose a data structure that answers such queries in $\mathcal{O}(\log n \log \log n)$ time for the shortest cover and in $\mathcal{O}(\log n (\log \log n)^2)$ time for a representation of all the covers, after $\mathcal{O}(n \log n)$ time and space preprocessing.

Keywords: cover · quasiperiodicity · internal pattern matching · seed · run (maximal repetition)

1 Introduction

A *cover* (also known as a quasiperiod) is a weak version of a period. It is a factor of a text T whose occurrences cover all positions in T ; see Fig. 1. The notion of cover is well-studied in the off-line model. Linear-time algorithms for computing the shortest cover and all the covers of a string of length n were proposed in [2] and [23,24], respectively. Moreover, linear-time algorithms for computing shortest and longest covers of all prefixes of a string are known; see [6] and [22], respectively. Covers were also studied in parallel [5,7] and streaming [13] models of computation. Definitions of other variants of quasiperiodicity can be found in the survey [12]. In this work we introduce covers to the internal pattern matching model [20].

In the internal pattern matching model, a text T of length n is given in advance and the goal is to answer queries related to factors of the text. One of the basic internal queries in texts are *period queries*, that were introduced in [19] (actually, internal primitivity queries were considered even earlier [9,10]).

* Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

** Partially supported by ERC Consolidator Grant 772346 TUGBOAT and by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

$T : \quad \overbrace{a b a a b a} \overbrace{b a a b a} \overbrace{b a a b a}$

Fig. 1. $\text{MINCOVER}(T) = aba$ is the shortest cover of T and $\text{MINCOVER}(T[2..13]) = baababa$ is the shortest cover of its suffix of length 12.

A period query requires one to compute all the periods of a given factor of T . It is known that they can be expressed as $\mathcal{O}(\log n)$ arithmetic sequences. The fastest known algorithm answering period queries is from [20]. It uses a data structure of $\mathcal{O}(n)$ size that can be constructed in $\mathcal{O}(n)$ expected time and answers period queries in $\mathcal{O}(\log n)$ time (a deterministic construction of this data structure was given in [16]). A special case of period queries are *two-period queries*, which ask for the shortest period of a factor that is known to be periodic. In [20] it was shown that two-period queries can be answered in constant time after $\mathcal{O}(n)$ -time preprocessing. Another algorithm for answering such queries was proposed in [3].

Let us denote by $\text{MINCOVER}(S)$ and $\text{ALLCOVERS}(S)$, respectively, the length of the shortest cover and the lengths of all covers of a string S . Similarly as in the case of periods, it can be shown that the set $\text{ALLCOVERS}(S)$ can be expressed as a union of $\mathcal{O}(\log |S|)$ pairwise disjoint arithmetic sequences. We consider data structures that allow to efficiently answer these queries in the internal model.

INTERNAL QUASIPERIOD QUERIES

Input: A text T of length n

Query: For any factor S of T , compute $\text{MINCOVER}(S)$ or $\text{ALLCOVERS}(S)$ after efficient preprocessing of the text T

Recently [11] we have shown how to compute the shortest cover of each cyclic shift of a string T of length n , that is, the shortest cover of each length- $|T|$ factor of T^2 , in $\mathcal{O}(n \log n)$ total time. This work can be viewed as a generalization of [11] to computing covers of any factor of a string. It also generalizes the earlier works on computing covers of prefixes of a string [6,22].

Our results. We show that MINCOVER and ALLCOVERS queries can be answered in $\mathcal{O}(\log n \log \log n)$ time and $\mathcal{O}(\log n (\log \log n)^2)$ time, respectively, with a data structure that uses $\mathcal{O}(n \log n)$ space and can be constructed in $\mathcal{O}(n \log n)$ time. In particular, the time required to answer an ALLCOVERS query is slower by only a poly $\log \log n$ factor from optimal. Moreover, we show that any m MINCOVER or ALLCOVERS queries can be answered off-line in $\mathcal{O}((n+m) \log n)$ and $\mathcal{O}((n+m) \log n \log \log n)$ time, respectively, and $\mathcal{O}(n+m)$ space. In particular, the former matches the complexity of the best known solution for computing shortest covers of all cyclic shifts of a string [11], despite being far more general. We assume the word RAM model of computation with word size $\Omega(\log n)$.

Our approach. Our main tool are *seeds*, a known generalization of the notion of cover. A seed is defined as a cover of a superstring of the text [14]. A representation of all seeds of a string T , denoted here $\text{SeedSet}(T)$, can be computed in linear

time [17]. We will frequently extract individual seeds from $SeedSet(T)$; each time such an auxiliary query needs $\mathcal{O}(\log \log n)$ time. Consequently, $\log \log n$ is a frequent factor in our query times related to internal covers.

We construct a tree-structure (static range tree) of so-called *basic factors* of a string. For each basic factor F we store a compact representation of the set $SeedSet(F)$. The crucial point is that the total length of all these factors is $\mathcal{O}(n \log n)$ and every other factor can be represented, using the tree-structure, as a concatenation of $\mathcal{O}(\log n)$ basic factors. Representations of seed-sets of basic factors are precomputed. Then, upon an internal query related to a specific factor S , we decompose S into concatenation of basic factors F_1, F_2, \dots, F_k . Intuitively, the representation of the set of covers or (in easier queries) the shortest cover will be computed as a “composition” of $SeedSet(F_1), SeedSet(F_2), \dots, SeedSet(F_k)$, followed by adjusting it to border conditions using internal pattern matching. To get efficiency, when querying about covers of a factor S , we do not compute the whole representation of $SeedSet(S)$ (these representations are only precomputed for basic factors).

Finally, several stringology tools related to properties of covers and string periodicity are used to improve $\text{poly} \log n$ -factors in the query time that would result from a direct application of this approach.

2 Preliminaries

We consider a text T of length n over an integer alphabet $\{0, \dots, n^{\mathcal{O}(1)}\}$. If this is not the case, its letters can be sorted and renumbered in $\mathcal{O}(n \log n)$ time, which does not influence the preprocessing time of our data structure.

For a string S , by $|S|$ we denote its length and by $S[i]$ we denote its i th letter ($i = 1, \dots, |S|$). By $S[i..j]$ we denote the string $S[i] \dots S[j]$ called a factor of S ; it is a prefix if $i = 1$ and a suffix if $j = |S|$. A factor that occurs both as a prefix and as a suffix of S is called a border of S . A factor is proper if it is shorter than the string itself. A positive integer p is called a period of S if $S[i] = S[i + p]$ holds for all $i = 1, \dots, |S| - p$. By $\text{per}(S)$ we denote the smallest period of S . A string S is called periodic if $|S| \geq 2\text{per}(S)$ and aperiodic otherwise. If $S = XY$, then any string of the form YX is called a cyclic shift of S . We use the following simple fact related to covers.

Observation 1. *Let A, B, C be strings such that $|A| < |B| < |C|$.*

- (a) *If A is a cover of B and B is a cover of C , then A is a cover of C .*
- (b) *If B is a border of C and A is a cover of C , then A is a cover of B .*

Below we list several algorithmic tools used later in the paper.

Queries related to suffix trees and arrays.

A range minimum query on array $A[1..n]$ requires to compute $\min\{A[i], \dots, A[j]\}$.

Lemma 2 ([4]). *Range minimum queries on an array of size n can be answered in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ -time preprocessing.*

By $\text{lcp}(i, j)$ ($\text{lcs}(i, j)$) we denote the length of the longest common prefix of $T[i..n]$ and $T[j..n]$ (longest common suffix of $T[1..i]$ and $T[1..j]$, respectively). Such queries are called longest common extension (LCE) queries. The following lemma is obtained by using range minimum queries on suffix arrays.

Lemma 3 ([4,15]). *After $\mathcal{O}(n)$ -time preprocessing, one can answer LCE queries for T in $\mathcal{O}(1)$ time.*

The suffix tree of T , denoted as $\mathcal{T}(T)$, is a compact trie of all suffixes of T . Each implicit or explicit node of $\mathcal{T}(T)$ corresponds to a factor of T , called its *string label*. The *string depth* of a node of $\mathcal{T}(T)$ is the length of its string label.

We use *weighted ancestor (WA) queries* on a suffix tree. Such queries, given an explicit node v and an integer value ℓ that does not exceed the string depth of v , ask for the highest explicit ancestor u of v with string depth at least ℓ .

Lemma 4 ([1,17]). *Let $\mathcal{T}(T)$ be the suffix tree of T . WA queries on $\mathcal{T}(T)$ can be answered in $\gamma_n = \mathcal{O}(\log \log n)$ time after $\mathcal{O}(n)$ -time preprocessing. Moreover, any m WA queries on $\mathcal{T}(T)$ can be answered off-line in $\mathcal{O}(n + m)$ time.*

Internal Pattern Matching (IPM).

The data structure for IPM queries is built upon a text T and allows efficient location of all occurrences of one factor X of T inside another factor Y of T , where $|Y| \leq 2|X|$.

Lemma 5 ([20]). *The result of an IPM query is a single arithmetic sequence. After linear-time preprocessing one can answer IPM queries for T in $\mathcal{O}(1)$ time.*

A period query, for a given factor X of text T , returns a compact representation of all the periods of X (as a set of $\mathcal{O}(\log n)$ arithmetic sequences).

Lemma 6 ([20]). *After $\mathcal{O}(n)$ time and space preprocessing, for any factor of T we can answer a period query in $\mathcal{O}(\log n)$ time.*

The data structures of Lemmas 5 and 6 are constructed in $\mathcal{O}(n)$ expected time. These constructions were made worst-case in [16].

Static range trees.

A *basic interval* is an interval $[a..a+2^i)$ such that 2^i divides $a-1$. We assume w.l.o.g. that n is a power of two. We consider a static range tree structure whose nodes correspond to basic subintervals of $[1..n]$ and a non-leaf node has children corresponding to the two halves of the interval. (See e.g. [18]). The total number of basic intervals is $\mathcal{O}(n)$. Using the tree, every interval $[i..j]$ can be decomposed into $\mathcal{O}(\log n)$ pairwise disjoint basic intervals. The decomposition can be computed in $\mathcal{O}(\log n)$ time by inspecting the paths from the leaves corresponding to i and j to their lowest common ancestor. A *basic factor* of T is a factor that corresponds to positions from a basic interval.

$\overbrace{a a b a a b a b a a b a a b a}^{\text{aba}}$
 $\overbrace{a a b a a b a b a a b a a b a}^{\text{aba}}$

Fig. 2. The strings aba , $abaab$ are seeds of the given string (as well as strings $abaaba$, $abaababa$, $abaababaa$).

Seeds.

We say that a string S is a *seed* of a string U if S is a factor of U and S is a cover of a string U' such that U is a factor of U' ; see Fig. 2. The second point of the lemma below follows from Lemma 4.

Lemma 7 ([17]).

- (a) All the seeds of T can be represented as a collection of a linear number of disjoint paths in the suffix tree $\mathcal{T}(T)$. Moreover, this representation can be computed in $\mathcal{O}(n)$ time if T is over an integer alphabet.
- (b) After $\mathcal{O}(n)$ time preprocessing we can check if a given factor of T is a seed of T in $\mathcal{O}(\gamma_n)$ time.

Our main data structure is a static range tree $SeedSets(T)$ which stores all seeds of every basic factor of T represented as a collection of paths in its suffix tree. Actually, only seeds of length at most half of a string will be of interest; see Fig. 3.

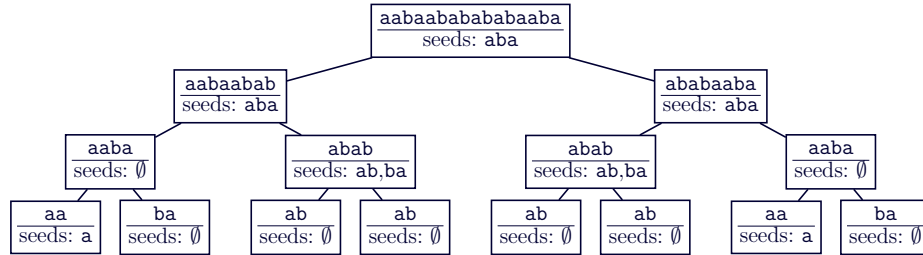


Fig. 3. A schematic view of tree $SeedSets$ of T (in the real data structure, seeds are stored on suffix trees of basic factors). For example, ba is a seed of $T[5..12]$ since it is a seed of basic factors $T[5..8]$ and $T[9..12]$ and its occurrence covers $T[8..9]$ (Lemma 10).

The sum of lengths of basic factors in T is $\mathcal{O}(n \log n)$. Consequently, due to Lemma 7, the tree $SeedSets(T)$ has total size $\mathcal{O}(n \log n)$ and can be computed in $\mathcal{O}(n \log n)$ time. (To use Lemma 7(a) we renumber letters in basic factors of T via bucket sort so that the letters of a basic factor S are from $\{0, \dots, |S|^{\mathcal{O}(1)}\}$.)

3 Internal Cover of a Given Length

In this section we show how to use $SeedSets(T)$ to answer internal queries related to computing the longest prefix of a factor S of T that is covered by its length- ℓ prefix. We start with the following, easier queries.

COVER OF A GIVEN LENGTH QUERY ($ISCOVER(\ell, S)$)

Input: A factor S of T and a positive integer ℓ

Query: Does S have a cover of length ℓ ?

The following three lemmas provide the building blocks of the data structure for answering $ISCOVER$ queries.

Lemma 8 (Seed of a basic factor). *After $\mathcal{O}(n \log n)$ -time preprocessing, for any factor C and basic factor B of T such that $2|C| \leq |B|$, we can check if C is a seed of B in $\mathcal{O}(\gamma_n)$ time.*

Proof. Let $|C| = c$ and $B = T[a..b]$. We first ask an IPM query to find an occurrence of C inside $T[a..a+2c-1]$. If such an occurrence does not exist, then C cannot be a seed of $T[a..b]$ as it is already not a seed of $T[a..a+2c-1]$ (there must be a full occurrence to cover the middle letter, and $a+2c-1 \leq b$). Otherwise, we can use the occurrence to check if C is a seed of B with Lemma 7(b). \square

For strings C and S , by $Cov(C, S)$ we denote the set of positions of S that are covered by occurrences of C .

Lemma 9 (Covering short factors). *After $\mathcal{O}(n)$ -time preprocessing, for any two factors C and F of T such that $|F|/|C| = \mathcal{O}(1)$, the set $Cov(C, F)$, represented as a union of maximal intervals, can be computed in $\mathcal{O}(1)$ time.*

Proof. We ask IPM queries for pattern C on length- $2|C|$ factors of F with step $|C|$. Each IPM query returns an arithmetic sequence of occurrences that corresponds to an interval of covered positions (possibly empty). It suffices to compute the union of these intervals. \square

Lemma 10 (Seeds of strings concatenation). *After $\mathcal{O}(n)$ -time preprocessing, for any three factors C , $F_1 = T[i..j]$ and $F_2 = T[j+1..k]$ of T such that $2|C| \leq |F_1|, |F_2|$ and C is a seed of both F_1 and F_2 , we can check if C is also a seed of F_1F_2 in constant time.*

Proof. For a string C of length c being a seed of both $T[i..j]$ and $T[j+1..k]$ to be a seed of $T[i..k]$, it is enough if its occurrences cover the string $U = T[j-c+1..j+c]$. We can check this condition if we apply Lemma 9 for C and $F = T[j-2c+1..j+2c]$. \square

Lemma 11. *After $\mathcal{O}(n \log n)$ time and space preprocessing of T , a query $ISCOVER(\ell, S)$ can be answered in $\mathcal{O}(\log(|S|/\ell)\gamma_n + 1)$ time.*

Proof. Let $S = T[i..j]$, $|S| = s$ and $C = T[i..i + \ell - 1]$.

We consider a decomposition of S into basic factors, but we are only interested in basic factors of length at least 2ℓ in the decomposition. Let F_1, \dots, F_k be those factors and $T[i..i']$, $T[j'..j]$ be the remaining prefix and suffix of length $\mathcal{O}(\ell)$. Note that $k = \mathcal{O}(\log(s/\ell))$. Moreover, this decomposition can be computed in $\mathcal{O}(k+1)$ time by starting from the leftmost and rightmost basic factors of length 2^b , where $b = \lceil \log \ell \rceil + 1$, that are contained in S .

If C is a cover of S , it must be a seed of each of the basic factors F_1, \dots, F_k . We can check this condition by using Lemma 8 in $\mathcal{O}(k\gamma_n)$ total time.

Next we check if C is a seed of $F_1 \cdots F_k$ in $\mathcal{O}(k)$ total time using Lemma 10. Finally, we use IPM queries to check if occurrences of C cover all positions in each of the strings $T[i..i' + c - 1]$, $T[j' - c + 1..j]$ and if C is a suffix of $T[i..j]$, using Lemma 9. This takes $\mathcal{O}(1)$ time.

The total time complexity is $\mathcal{O}(k\gamma_n + 1)$. □

As we will see in the next section, ISCOVER queries immediately imply a slower, $\mathcal{O}(\log^2 n \gamma_n)$ -time algorithm for answering MINCOVER queries. However, they are also used in our algorithm for answering ALLCOVERS queries. In the efficient algorithm for MINCOVER queries we use the following generalization of ISCOVER queries.

LONGEST COVERED PREFIX QUERY (COVEREDPREF(ℓ, S))

Input: A factor S of T and a positive integer ℓ

Query: The longest prefix P of S that is covered by $S[1.. \ell]$

To answer these queries, we introduce an intermediate problem that is more directly related to the range tree containing seeds representations.

SEEDED BASICPREF(C, ℓ, S) QUERY

Input: A length- ℓ factor C of T and a factor S being a concatenation of basic factors of T of length 2^p , where $p = \min\{q \in \mathbb{Z} : 2^q \geq 2\ell\}$

Output: The length m of the longest prefix of S which is a concatenation of basic factors of length 2^p such that C is a seed of this prefix

In other words, we consider only blocks of S which are basic factors of length $2^p = \Theta(\ell)$. Everything starts and ends in the beginning/end of a basic factor of length 2^p . The number of such blocks in the prefix returned by SEEDED BASICPREF is $\mathcal{O}(\text{result}'/\ell)$, where $\text{result}' = \text{SEEDED BASICPREF}(C, \ell, S)$, and, as we show in Lemma 13, it can be computed in $\mathcal{O}(\log(\text{result}'/\ell)\gamma_n + 1)$ time. This is how we achieve $\mathcal{O}(\log(\text{result}/\ell)\gamma_n + 1)$ time for COVEREDPREF(ℓ, S) queries. In a certain sense the computations behind Lemma 13 can work in a pruned range tree $\text{SeedSets}(T)$.

Lemma 12. *After $\mathcal{O}(n)$ -time preprocessing, a COVEREDPREF(ℓ, S) query reduces in $\mathcal{O}(1)$ time to a SEEDED BASICPREF(C, ℓ, S') query with $|S'| \leq |S|$.*

Proof. First, let us check if the answer to $\text{COVEREDPREF}(\ell, S)$ is small, i.e. at most 4ℓ , using Lemma 9. Otherwise, let p be defined as in a SEEDEDBASICPREF query, $C = S[1.. \ell]$ and S' be the maximal factor of S that is composed of basic factors of length 2^p (S' can be the empty string, if $|S| < 3 \cdot 2^p$). Let $S = T[i.. j]$ and $S' = T[i'.. j']$. Then

$$|(i' + \text{SEEDEDBASICPREF}(C, \ell, S')) - (i + \text{COVEREDPREF}(\ell, S))| < 2^p;$$

see Fig. 4. Hence, knowing $d = \text{SEEDEDBASICPREF}(C, \ell, S')$, we check in $\mathcal{O}(1)$ time, using Lemma 9 in a factor $T[i' + d - 2^p .. i' + d + 2^p - 1]$ of length 2^{p+1} , what is the exact value of $\text{COVEREDPREF}(\ell, S)$.

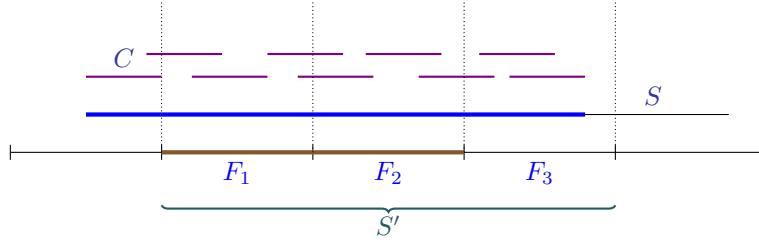


Fig. 4. F_1, F_2, F_3 are basic factors of length 2^p . The answers to $\text{COVEREDPREF}(\ell, S)$ and $\text{SEEDEDBASICPREF}(C, \ell, S')$ queries are shown in bold. Note that C is a seed of F_1 and F_2 and that it could be the case that C is also a seed of F_3 , even though it has no further full occurrence.

We compute p using the formula $p = 1 + \lceil \log \ell \rceil$. Then the endpoints of S' can be computed from the endpoints of S in $\mathcal{O}(1)$ time using simple modular arithmetic. The $\mathcal{O}(n)$ preprocessing is due to Lemma 9. \square

A proof of the following lemma is left for the full version.

Lemma 13. *After $\mathcal{O}(n \log n)$ time and space preprocessing of T , a query $\text{SEEDEDBASICPREF}(C, \ell, S)$ can be answered in $\mathcal{O}(\log(\text{result}/\ell) \gamma_n + 1)$ time, where $\text{result} = |\text{SEEDEDBASICPREF}(C, \ell, S)|$.*

As a corollary of Lemmas 12 and 13, we obtain the following result.

Lemma 14. *After $\mathcal{O}(n \log n)$ time and space preprocessing of T , a query $\text{COVEREDPREF}(\ell, S)$ can be answered in $\mathcal{O}(\log(\text{result}/\ell) \gamma_n + 1)$ time, where $\text{result} = |\text{COVEREDPREF}(\ell, S)|$.*

4 Internal Shortest Cover Queries

For a string S , by $\text{Borders}(S)$ we denote a decomposition of the set of all border lengths of S into $\mathcal{O}(\log |S|)$ arithmetic sequences A_1, \dots, A_k such that each

sequence A_i is either a singleton or, if p is its difference, then the borders with lengths in $A_i \setminus \{\min(A_i)\}$ are periodic with the shortest period p . Moreover, $\max(A_i) < \min(A_{i+1})$ for every $i \in [1..k-1]$. See e.g. [8]. The following lemma is shown by applying a period query (Lemma 6).

Lemma 15 ([16,20]). *For any factor S of T , $Borders(S)$ can be computed in $\mathcal{O}(\log n)$ time after $\mathcal{O}(n)$ -time preprocessing.*

4.1 Simple Algorithm with $\mathcal{O}(\log^2 n \gamma_n)$ Query Time

Let us start with a much simpler but slower algorithm for answering MINCOVER queries using ISCOVER queries. We improve it in Theorem 17 by using COVEREDPREF queries and applying an algorithm for computing shortest covers that resembles, to some extent, computation of the shortest cover from [2].

Proposition 16. *Let T be a string of length n . After $\mathcal{O}(n \log n)$ -time preprocessing, for any factor S of T we can answer a MINCOVER(S) query in $\mathcal{O}(\log^2 n \log \log n)$ time.*

Proof. Using Lemma 15 we compute the set $Borders(S) = A_1, \dots, A_k$ in $\mathcal{O}(\log n)$ time. Let us observe that the shortest cover of a string is aperiodic. This implies that from each progression A_i only the border of length $\min(A_i)$ can be the shortest cover of S . We use Lemma 11 to test each of the $\mathcal{O}(\log n)$ candidates in $\mathcal{O}(\log n \gamma_n)$ time. \square

4.2 Faster Queries

Theorem 17. *Let T be a string of length n . After $\mathcal{O}(n \log n)$ -time preprocessing, for any factor S of T we can answer a MINCOVER(S) query in $\mathcal{O}(\log n \log \log n)$ time.*

Proof. Again we use Lemma 15 we compute the set $Borders(S) = A_1, \dots, A_k$, in $\mathcal{O}(\log n)$ time. Let us denote the border of length $\min(A_i)$ by C_i and $C_{k+1} = S$. We assume that C_i 's are sorted in increasing order of lengths. Then we proceed as shown in Algorithm 1. See also Fig. 5.

Algorithm 1: MINCOVER(S) query.

```

i := 1;
while true do
  // Invariant:  $C_1, \dots, C_{i-1}$  are not covers of  $S$ 
  //  $C_i$  is an active border
   $P := \text{COVEREDPREF}(|C_i|, S)$ ;
  if  $P = S$  then return  $|C_i|$ ;
  while  $|C_i| \leq |P|$  do
     $i := i + 1$ ;

```

To argue for the correctness of the algorithm it suffices to show the invariant. The proof goes by induction.

The base case is trivial. Let us consider the value of i at the beginning of a step of the while-loop. If $P = S$, then by the inductive assumption C_i is the shortest cover of S and can be returned. Otherwise, C_i is not a cover of S .

Moreover, for each j such that $|C_i| < |C_j| \leq |P|$, since C_j is a prefix of P , C_i is a seed of C_j . Moreover, both C_i and C_j are borders of S , so C_i is a border of C_j . Consequently, C_j cannot be a cover of T , as then C_i would also be a cover of T by Observation 1. This shows that the inner while-loop correctly increases i .

The algorithm stops because at each point $|P| \geq |C_i|$ and i is increased.

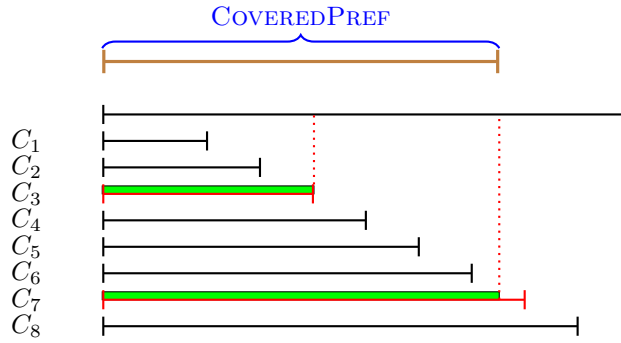


Fig. 5. If C_3 is an active border, then the next active one is C_7 . We skip C_4, C_5, C_6 as candidates for the shortest cover.

Let c_1, \dots, c_p be equal to the length of an active border in the algorithm at the start of subsequent outer while-loop iterations and let $c_{p+1} = |S|$.

Let us note that, for all $j = 1, \dots, p$, $|\text{COVEREDPREF}(c_j, S)| \leq c_{j+1}$. By Lemma 14, the total complexity of answering longest covered prefix queries in the algorithm is at most

$$\mathcal{O}\left(p + \gamma_n \sum_{j=1}^p \log \frac{c_{j+1}}{c_j}\right) = \mathcal{O}(\log n + \gamma_n(\log c_{p+1} - \log c_1)) = \mathcal{O}(\log n \gamma_n).$$

The preprocessing of Lemmas 14 and 15 takes $\mathcal{O}(n \log n)$ time. The conclusion follows. \square

If MINCOVER queries are to be answered in a batch, we can use off-line WA queries of Lemma 4 to save the γ_n -factor. We can also avoid storing the whole data structure *SeedSets* by using an approximate version of COVEREDPREF queries. Details are left for the full version.

Theorem 18. *For a string T of length n , any m queries $\text{MINCOVER}(T[i..j])$ can be answered in $\mathcal{O}((n+m) \log n)$ time and $\mathcal{O}(n+m)$ space.*

5 Internal All Covers Queries

In this section we refer to $\text{ALLCOVERS}(S)$ as to the set of lengths of all covers of S . This set consists of a logarithmic number of arithmetic sequences since the same is true for all borders. In each sequence of borders we show that it is needed only to check $\mathcal{O}(1)$ borders to be a cover of S . Hence we start with an algorithm testing any sequence of $\mathcal{O}(\log n)$ candidate borders.

5.1 Verifying $\mathcal{O}(\log n)$ Candidates

Assume that B is an increasing sequence b_1, \dots, b_k of lengths of borders of a given factor S (not necessarily all borders), with $b_k = |S|$. A *chain* in B is a maximal subsequence b_i, \dots, b_j of consecutive elements of B such that $S[1..b_t]$ is a cover of $S[1..b_{t+1}]$ for each $t \in [i..j)$. From Observation 1 we get the following.

Observation 19. *The set of elements of a chain that belong to $\text{ALLCOVERS}(S)$ is a prefix of this chain. Moreover, if the last element of a chain is not $|S|$, then it is not a cover of S .*

We denote by $\text{chains}(B)$ and $\text{covers}(B)$, respectively, the partition of B into chains and the set of elements $b \in B$ such that $S[1..b]$ is a cover of S . For $b \in B$ by $\text{prev}(b)$ we denote the previous element in its chain (if it exists). Moreover, for $C \subseteq B$ by $\text{next}_C(b)$ we denote the smallest $c \in C$ such that $c > b$.

Lemma 20. *Let T be a string of length n . After $\mathcal{O}(n \log n)$ -time preprocessing, for any factor S of T and a sequence B of $\mathcal{O}(\log n)$ borders of S we can compute $\text{covers}(B)$ in $\mathcal{O}(\log n \log \log n \gamma_n)$ time.*

Proof. We introduce two operations and use them in a recursive Algorithm 2.

refine(B): removes the last element of each chain in B and every second element of each chain, except $|S|$ (see Fig. 6). Note that $|\text{refine}(B)| \leq |B|/2 + 1$.

computeUsing(B, C): Assuming that we know the set C of all covers of S among $\text{refine}(B)$, for each element b of $B \setminus \text{refine}(B)$ we add it to C if $\text{prev}(b) \in C$ and $S[1..b]$ is a cover of $S[1..\text{next}_C(b)]$. The set of all elements that satisfy this condition together with C is returned as $\text{covers}(B)$.

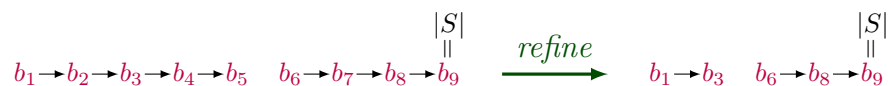


Fig. 6. There is an arrow from b_i to b_{i+1} iff $S[1..b_i]$ is a cover of $S[1..b_{i+1}]$. Note that all elements in the last chain b_6, b_7, b_8, b_9 are cover lengths of S , b_5 is not, but some prefix of b_1, b_2, b_3, b_4 may be.

Algorithm 2: $\text{covers}(B)$

```
Compute  $\text{chains}(B)$ ;  
if  $B$  is a single chain (ending with  $|S|$ ) then return  $B$ ;  
 $B' := \text{refine}(B)$ ; //  $|B'| \leq |B|/2 + 1$   
 $C := \text{covers}(B')$ ;  
return  $\text{computeUsing}(B, C)$ ;
```

If $B = (b_1, \dots, b_k)$, then $\text{chains}(B)$ can be constructed in $\mathcal{O}(\sum_{i=1}^{k-1} (\log \frac{b_{i+1}}{b_i} \gamma_n + 1)) = \mathcal{O}(\log n \gamma_n)$ time using Lemma 11. Similarly, operation $\text{computeUsing}(B, C)$ requires $\mathcal{O}(\log n \gamma_n)$ time since the intervals $[b, \text{next}_C(b)]$ for $b \in B \setminus \text{refine}(B)$ such that $\text{prev}(b) \in C$ are pairwise disjoint. The depth of recursion of Algorithm 2 is $\mathcal{O}(\log \log n)$. This implies the required complexity. \square

5.2 Computing Periodic Covers

Our tool for periodic covers are (as usual) *runs*. A *run* (also known as a *maximal repetition*) is a periodic factor $R = T[a..b]$ which can be extended neither to the left nor to the right without increasing the period $p = \text{per}(R)$, i.e., $T[a-1] \neq T[a+p-1]$ and $T[b-p+1] \neq T[b+1]$ provided that the respective positions exist. The following observation is well-known.

Observation 21. *Two runs in T with the same period p can overlap on at most $p-1$ positions.*

The *exponent* $\text{exp}(S)$ of a string S is $|S|/\text{per}(S)$. The *Lyndon root* of a string S is the minimal cyclic shift of $S[1.. \text{per}(S)]$.

If $S = T[a..b]$ is periodic, then by $\text{run}(S)$ we denote the run R with the same period that contains S . We say that S is *induced* by R . A periodic factor of T is induced by exactly one run [10]. The *run*-queries are essentially equivalent to two-period queries. By $\mathcal{R}(T)$ we denote the set of all runs in a string T .

Lemma 22 ([3,10,21]).

- (a) $|\mathcal{R}(T)| \leq n$ and $\mathcal{R}(T)$ can be computed in $\mathcal{O}(n)$ time.
- (b) After $\mathcal{O}(n)$ -time preprocessing, $\text{run}(S)$ queries can be answered in $\mathcal{O}(1)$ time.
- (c) The runs from $\mathcal{R}(T)$ can be grouped by their Lyndon roots in $\mathcal{O}(n)$ time.

The following lemma implies that indeed for any string S , $\text{ALLCOVERS}(S)$ can be expressed as a union of $\mathcal{O}(\log |S|)$ arithmetic sequences. It also shows a relation between periodic covers and runs in S .

Lemma 23. *Let S be a string, $A \in \text{Borders}(S)$ be an arithmetic sequence with difference p , $A' = A \setminus \{\min(A)\}$ and $a' = \min(A')$. Moreover, let x be the minimal exponent of a run in S with Lyndon root being a cyclic shift of $S[1..p]$.*

- (a) *If $a' \notin \text{ALLCOVERS}(S)$, then $A' \cap \text{ALLCOVERS}(S) = \emptyset$.*

(b) *Otherwise, there exists $c \in ((x-2)p, xp] \cap A'$ such that $A' \cap \text{ALLCOVERS}(S) = \{a', a' + p, \dots, c\}$.*

Proof. Part (a) follows from Observation 1. Indeed, assume that S has a cover of length $b \in A'$, with $b > a'$. As $S[1..a']$ is a cover of $S[1..b]$, we would have $a' \in \text{ALLCOVERS}(S)$.

We proceed to the proof of part (b). Let c be the maximum element of A' such that $C := S[1..c]$ is a cover of S . By the same argument as before, we have that $A' \cap \text{ALLCOVERS}(S) = A' \cap [1, c]$. It suffices to prove the bounds for c .

Let L be the minimum cyclic shift of $S[1..p]$. We consider all runs R_1, \dots, R_k in S with Lyndon root L . Each occurrence of C in S is induced by one of them. Each of the runs must hold an occurrence of C . Indeed, by Observation 21, no two of the runs overlap on more than $p-1$ positions, so the p th position of each run cannot be covered by occurrences of C that are induced by other runs. The shortest of the runs has length xp , so $c \leq xp$.

Furthermore, let $C' = S[1..c']$ be a prefix of S of length $c' = c + p$. If $p \cdot \text{exp}(R_i) \geq c' + p - 1$, then R_i induces an occurrence of C' and $\text{Cov}(C', R_i) = \text{Cov}(C, R_i)$. Hence, if $px \geq c' + p - 1$ would hold, C' would be a cover of S , which contradicts our assumption. Therefore, $px < c' + p - 1 = c + 2p - 1$, so $c > (x-2)p$. \square

Lemma 25 transforms Lemma 23 into a data structure. We use static dictionaries.

Lemma 24 (Ružić [25]). *A static dictionary of n integers that supports $\mathcal{O}(1)$ -time lookups can be stored in $\mathcal{O}(n)$ space and constructed in $\mathcal{O}(n \log \log n)^2$ time. The elements stored in the dictionary may be accompanied by satellite data.*

Lemma 25 (Computing $\mathcal{O}(\log n)$ Candidates).

For any factor S of T we can compute in $\mathcal{O}(\log n)$ time $\mathcal{O}(\log n)$ borders of S which are candidates for covers of S . After knowing which of these candidates are covers of S , we can in $\mathcal{O}(\log n)$ time represent (as $\mathcal{O}(\log n)$ arithmetic progressions) all borders which are covers of S . The preprocessing time is $\mathcal{O}(n \log \log n)^2$ and the space used is $\mathcal{O}(n)$.

Proof. It is enough to show that for any factor S of T and a single arithmetic sequence $A \in \text{Borders}(S)$ we can compute in $\mathcal{O}(1)$ time up to four candidate borders. Then, after knowing which of them are covers of S , we can in $\mathcal{O}(1)$ time represent (as a prefix subsequence of A) all borders in A which are covers of S . We first describe the data structure and then the query algorithm.

Data structure. Let $T[a_1..b_1], \dots, T[a_k..b_k]$ be the set of all runs in T with Lyndon root L , with $a_1 < \dots < a_k$ (and $b_1 < \dots < b_k$). The part of the data structure for this Lyndon root consists of an array A_L containing a_1, \dots, a_k , an array E_L containing the exponents of the respective runs, as well as a dictionary on A_L and a range-minimum query data structure on E_L . Formally, to each Lyndon root we assign an integer identifier in $[1, n]$ that is retained with every run with this Lyndon root and use it to index the data structures. We also store a dictionary of all the runs. The data structure takes $\mathcal{O}(n)$ space and can

be constructed in $\mathcal{O}(n(\log \log n)^2)$ time by Lemmas 2, 22 and 24. We also use LCE-queries on T (Lemma 3).

Queries. Let us consider a query for $S = T[i..j]$ and $A \in \text{Borders}(S)$. If $|A| = 1$, we have just one candidate. Otherwise, A is an arithmetic sequence with difference p . Let $a = \min(A)$, $A' = A \setminus \{a\}$, and $a' = \min(A')$. We select borders of length a and a' as candidates. If $a' \notin \text{ALLCOVERS}(S)$, then Lemma 23(a) implies that $A \cap \text{ALLCOVERS}(S) \subseteq \{a\}$. We also select borders of lengths in $A \cap ((x-2)p, xp]$ as candidates, where x is defined as in Lemma 23. Note that there are at most two of them. Let c be the maximum candidate which turned out to be a cover of S . Then $A \cap \text{ALLCOVERS}(S) = A \cap [1, c]$ by Lemma 23(b).

What is left is to compute x , that is, the minimum exponent of a run in S with Lyndon root L that is a cyclic shift of $S[1..p]$. Since $|A| \geq 2$, S has a prefix run with Lyndon root L . Then $\ell = \min(p + d, |S|)$, where $d = \text{lcp}(i, i + p)$, is the length of the run. If $\ell = |S|$, then $x = \ell/p$ and we are done. Otherwise, let $i' = i + p + d$. We make the following observation.

Claim. If $a' \in \text{ALLCOVERS}(S)$, then $T[i'..i' + p]$ is contained in a run in T with Lyndon root L .

We identify the run $T[a..b]$ with period p containing $T[i'..i' + p]$ by asking $\text{lcp}(i', i' + p)$ and $\text{lcs}(i', i' + p)$ queries. This lets us recover the identifier of its Lyndon root L . Similarly we compute the suffix run with Lyndon root L in S and the previous run $T[a'..b']$ with Lyndon root L in T . Using the dictionary on A_L , we recover the range in the array that corresponds to elements from a to a' . This lets us use a range minimum query on this range in E_L and use it together with the exponents of the prefix and suffix runs of S to compute x . \square

5.3 Main Query Algorithm

The main result of this section follows from Lemma 20 and Lemma 25.

Theorem 26. *Let T be a string of length n . After $\mathcal{O}(n \log n)$ -time preprocessing, for any factor S of T we can answer a query $\text{ALLCOVERS}(S)$, with output represented as a union of $\mathcal{O}(\log n)$ pairwise disjoint arithmetic sequences, in $\mathcal{O}(\log n(\log \log n)^2)$ time.*

The transformation to the off-line model is similar as in Theorem 18.

Corollary 27. *For a string T of length n , any m queries $\text{ALLCOVERS}(T[i..j])$ can be answered in $\mathcal{O}((n + m) \log n \log \log n)$ time and $\mathcal{O}(n + m)$ space.*

6 Final Remarks

We showed an efficient data structure for computing internal covers. However, a similar problem for seeds, which are another well-studied notion in quasiperiodicity, seems to be much harder. We pose the following question.

Open problem.

Can one answer internal queries related to seeds in $\mathcal{O}(\text{polylog } n)$ time after $\mathcal{O}(n \cdot \text{polylog } n)$ time preprocessing?

References

1. Amir, A., Landau, G.M., Lewenstein, M., Sokol, D.: Dynamic text and static pattern matching. *ACM Trans. Algorithms* **3**(2), 19 (2007). <https://doi.org/10.1145/1240233.1240242>
2. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings. *Inf. Process. Lett.* **39**(1), 17–20 (1991). [https://doi.org/10.1016/0020-0190\(91\)90056-N](https://doi.org/10.1016/0020-0190(91)90056-N)
3. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. *SIAM J. Comput.* **46**(5), 1501–1514 (2017). <https://doi.org/10.1137/15M1011032>
4. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Panario, D., Viola, A. (eds.) *LATIN 2000: Theoretical Informatics*, 4th Latin American Symposium, Punta del Este, Uruguay, April 10–14, 2000, Proceedings. *Lecture Notes in Computer Science*, vol. 1776, pp. 88–94. Springer (2000). https://doi.org/10.1007/10719839_9
5. Berkman, O., Iliopoulos, C.S., Park, K.: The subtree max gap problem with application to parallel string covering. *Inf. Comput.* **123**(1), 127–137 (1995). <https://doi.org/10.1006/inco.1995.1162>
6. Breslauer, D.: An on-line string superprimitivity test. *Inf. Process. Lett.* **44**(6), 345–347 (1992). [https://doi.org/10.1016/0020-0190\(92\)90111-8](https://doi.org/10.1016/0020-0190(92)90111-8)
7. Breslauer, D.: Testing string superprimitivity in parallel. *Inf. Process. Lett.* **49**(5), 235–241 (1994). [https://doi.org/10.1016/0020-0190\(94\)90060-4](https://doi.org/10.1016/0020-0190(94)90060-4)
8. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., Tyczyński, W., Waleń, T.: The maximum number of squares in a tree. In: Kärkkäinen, J., Stoye, J. (eds.) *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3–5, 2012*. Proceedings. *Lecture Notes in Computer Science*, vol. 7354, pp. 27–40. Springer (2012). https://doi.org/10.1007/978-3-642-31265-6_3
9. Crochemore, M., Iliopoulos, C.S., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: Extracting powers and periods in a string from its runs structure. In: Chávez, E., Lonardi, S. (eds.) *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11–13, 2010*. Proceedings. *Lecture Notes in Computer Science*, vol. 6393, pp. 258–269. Springer (2010). https://doi.org/10.1007/978-3-642-16321-0_27
10. Crochemore, M., Iliopoulos, C.S., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: Extracting powers and periods in a word from its runs structure. *Theor. Comput. Sci.* **521**, 29–41 (2014). <https://doi.org/10.1016/j.tcs.2013.11.018>
11. Crochemore, M., Iliopoulos, C.S., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., Zuba, W.: Shortest covers of all cyclic shifts of a string. In: Rahman, M.S., Sadakane, K., Sung, W. (eds.) *Algorithms and Computation - 14th International Conference, WALCOM 2020, Singapore, March 31 - April 2, 2020*, Proceedings. *Lecture Notes in Computer Science*, vol. 12049, pp. 69–80. Springer (2020). https://doi.org/10.1007/978-3-030-39881-1_7
12. Czajka, P., Radoszewski, J.: Experimental evaluation of algorithms for computing quasiperiods. *CoRR* **abs/1909.11336** (2019), <http://arxiv.org/abs/1909.11336>
13. Gawrychowski, P., Radoszewski, J., Starikovskaya, T.: Quasi-periodicity in streams. In: Pisanti, N., Pissis, S.P. (eds.) *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18–20, 2019, Pisa, Italy*. *LIPICs*, vol. 128, pp. 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). <https://doi.org/10.4230/LIPICs.CPM.2019.22>

14. Iliopoulos, C.S., Moore, D.W.G., Park, K.: Covering a string. *Algorithmica* **16**(3), 288–297 (1996). <https://doi.org/10.1007/BF01955677>
15. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. *J. ACM* **53**(6), 918–936 (2006). <https://doi.org/10.1145/1217856.1217858>
16. Kociumaka, T.: Efficient Data Structures for Internal Queries in Texts. Ph.D. thesis, University of Warsaw (2018), <https://mimuw.edu.pl/~kociumaka/files/phd.pdf>
17. Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: A linear-time algorithm for seeds computation. *ACM Trans. Algorithms* **16**(2) (Apr 2020). <https://doi.org/10.1145/3386369>
18. Kociumaka, T., Radoszewski, J., Rytter, W., Straszypiński, J., Waleń, T., Zuba, W.: Efficient representation and counting of antipower factors in words. In: Martín-Vide, C., Okhotin, A., Shapira, D. (eds.) *Language and Automata Theory and Applications - 13th International Conference, LATA 2019, St. Petersburg, Russia, March 26-29, 2019, Proceedings. Lecture Notes in Computer Science*, vol. 11417, pp. 421–433. Springer (2019). https://doi.org/10.1007/978-3-030-13435-8_31
19. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Efficient data structures for the factor periodicity problem. In: Calderón-Benavides, L., González-Caro, C.N., Chávez, E., Ziviani, N. (eds.) *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings. Lecture Notes in Computer Science*, vol. 7608, pp. 284–294. Springer (2012). https://doi.org/10.1007/978-3-642-34109-0_30
20. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal pattern matching queries in a text and applications. In: Indyk, P. (ed.) *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*. pp. 532–551. SIAM (2015). <https://doi.org/10.1137/1.9781611973730.36>
21. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. pp. 596–604. IEEE Computer Society (1999). <https://doi.org/10.1109/SFFCS.1999.814634>
22. Li, Y., Smyth, W.F.: Computing the cover array in linear time. *Algorithmica* **32**(1), 95–106 (2002). <https://doi.org/10.1007/s00453-001-0062-2>
23. Moore, D.W.G., Smyth, W.F.: An optimal algorithm to compute all the covers of a string. *Inf. Process. Lett.* **50**(5), 239–246 (1994). [https://doi.org/10.1016/0020-0190\(94\)00045-X](https://doi.org/10.1016/0020-0190(94)00045-X)
24. Moore, D.W.G., Smyth, W.F.: A correction to "An optimal algorithm to compute all the covers of a string". *Inf. Process. Lett.* **54**(2), 101–103 (1995). [https://doi.org/10.1016/0020-0190\(94\)00235-Q](https://doi.org/10.1016/0020-0190(94)00235-Q)
25. Ružić, M.: Constructing efficient dictionaries in close to sorting time. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfssdóttir, A., Walukiewicz, I. (eds.) *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Track A: Algorithms, Automata, Complexity, and Games. Lecture Notes in Computer Science*, vol. 5125, pp. 84–95. Springer (2008). https://doi.org/10.1007/978-3-540-70575-8_8



String Covers of a Tree

Jakub Radoszewski^() , Wojciech Rytter , Juliusz Straszyński ,
Tomasz Walen , and Wiktor Zuba 

University of Warsaw, Warsaw, Poland
{jrad,rytter,jks,waleni,w.zuba}@mimuw.edu.pl

Abstract. We consider covering labeled trees by a collection of paths with the same string label, called a (string) cover of a tree. We show how to compute all covers of a directed (rooted) labeled tree in $\mathcal{O}(n \log n / \log \log n)$ time and all covers of an undirected labeled tree in $\mathcal{O}(n^2)$ time and space or in $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space. We also show several essential differences between covers in standard strings and covers in trees.

1 Introduction

We consider undirected and directed trees with at least 2 nodes and edges labeled by single symbols. The label of a simple path $v_1 \xrightarrow{a_1} v_2 \xrightarrow{a_2} v_3 \xrightarrow{a_3} \dots \xrightarrow{a_{k-1}} v_k$ is the string $W = a_1 a_2 \dots a_{k-1}$. Let us note that the label of the reverse path from v_k to v_1 is W^R , the reverse of W . We say that a non-empty string W **covers** a tree if each edge is on a simple path labeled by W . In case of rooted trees we consider only edges directed bottom-up towards the root (the symmetric ordering top-down is equivalent with respect to coverings). Figure 1 shows the covers of an example undirected tree.

A standard string can be considered as a directed path, hence covers of a directed path can be found using one of the known algorithms for computing covers of strings [4, 7, 24, 25]. However, covering an undirected simple path (see e.g. Fig. 2) is a very different problem and is much harder than covering a string. It is equivalent to covering with two strings W and W^R . A nontrivial almost linear time algorithm for covering an undirected path is implied by the algorithm in [27, Section 3] about covering a string with two equal-length strings.

Covers in directed trees are also quite different from covers in strings. A directed tree can be seen as a collection of strings corresponding to leaf-to-root paths, but a cover of a tree does not necessarily cover each of these strings; see Fig. 3, where the string *ababaaba* is not a cover of a string corresponding to the leftmost leaf-to-root path, though it is a cover of the whole tree. However, the following fact can be shown with the aid of induction.

Work supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

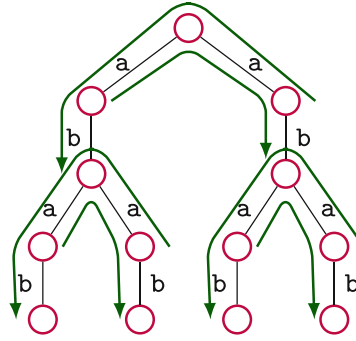


Fig. 1. The string aab is a cover of this undirected tree T ; all its occurrences are shown in green. The tree has 14 distinct covers: $aab, aabab, ab, abaabab, abab, ba, baa, baab, baabab, baba, babaa, babaab, babaaba, babaabab$. Observe that the bottom-up directed version of T rooted in the top node has only two covers: ba and $baba$.

$$1 \overset{a}{-} 2 \overset{a}{-} 3 \overset{a}{-} 4 \overset{a}{-} 5 \overset{a}{-} 6 \overset{b}{-} 7 \overset{a}{-} 8 \overset{a}{-} 9 \overset{a}{-} 10 \overset{a}{-} 11 \overset{a}{-} 12$$

Fig. 2. An undirected path $a^m b a^m$ of length $n = 2m + 1$ has $\Omega(n)$ different aperiodic covers, of the form $a^m b a^i$ or $a^i b a^m$, for $i = 0, \dots, m$. A standard string of length n can only have $\mathcal{O}(\log n)$ aperiodic covers; see [4].

Observation 1. *If S is a cover of a directed tree T , then it is a cover of at least one of the strings corresponding to leaf-to-root branches.*

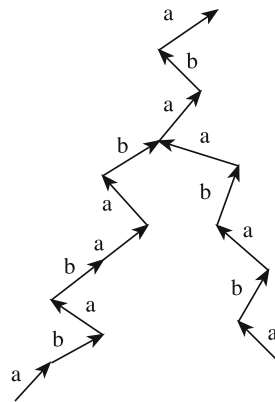


Fig. 3. The displayed directed tree has 3 covers $aba, ababa, ababaaba$. In case of standard strings a shorter cover is a suffix of the longer one. This does not work for covers of directed trees; $ababa$ is not a suffix of $ababaaba$

This work extends the rich study of covers in non-standard settings—e.g., 2-dimensional [9,10,26], Abelian [20,21,23], parameterized and order-preserving [17], and on indeterminate [1,3,11,16] and weighted strings [5,16]—to labeled trees. Moreover, we continue the line of work on algorithmic and combinatorial properties of palindromes, powers and runs in labeled trees, which have different properties than in strings [8,10,12–14,18,19,29].

Our Results: We show that all covers of a directed labeled tree can be computed in $\mathcal{O}(n \log n / \log \log n)$ time if the labels of tree edges belong to an integer alphabet, i.e., are integers of magnitude $n^{\mathcal{O}(1)}$. In case of undirected trees with labels over any alphabet, all covers can be computed in $\mathcal{O}(n^2)$ time and space or in $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space.

2 Preliminaries

In this section we introduce several algorithms and combinatorial properties related to labeled and unlabeled trees. If $u_1 \rightsquigarrow u_k = (u_1, u_2, \dots, u_{k-1}, u_k)$ is a (simple) path in a labeled tree T , then its *length* $\text{dist}(u_1, u_k)$ is defined as the number of edges (i.e., $k - 1$) and its *string label* is the concatenation of labels of edges $(u_1, u_2), \dots, (u_{k-1}, u_k)$.

Let T be a rooted labeled tree. We assume that all edges are directed towards the root. For a node v of T , by $\text{label}_d(v)$ we denote the string label of a path from v to its ancestor at distance d and by $\text{label}(v)$ we denote the string label of a path from v to the root.

2.1 The Table of Prefixes

For a string S , we denote $\text{TreePREF}_S[v] = \max\{d \geq 0 : \text{label}_d(v) = S[1..d]\}$, where $S[1..d]$ is a length- d prefix of string S ; see Fig. 4.

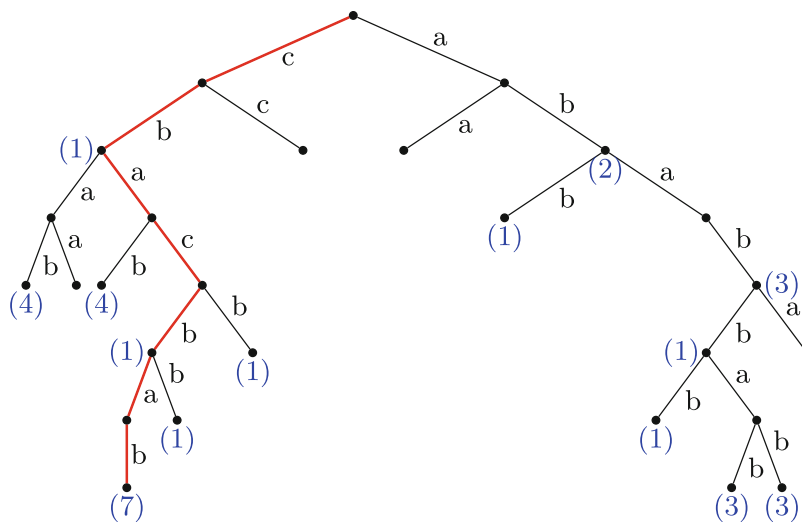


Fig. 4. An example of TreePREF_S array, where $S = babcabc$ (red path, bottom-up). The nonzero values of the array are shown in brackets. (Color figure online)

To compute this array we use the concept of a suffix tree of a labeled tree T that was introduced by Kosaraju [22]. A *compacted trie* is a trie in which maximal paths whose inner nodes have degree 2 are represented as single edges with string

labels. Usually such labels are not stored explicitly, but as pointers to a base string (or base strings); only the first letters are stored. The remaining nodes are called explicit, whereas the nodes that are removed due to compactification are called implicit. The *suffix tree of a rooted labeled tree* T is a compacted trie of the strings $label(v)$ for all nodes v in T . An efficient construction of the suffix tree of a tree was given by Shibuya [28].

Fact 1 (Shibuya [28]) The suffix tree of a rooted tree with n nodes over an integer alphabet has size $\mathcal{O}(n)$ and can be constructed in $\mathcal{O}(n)$ time.

Lemma 1. *For a directed labeled tree T with n nodes, the array TreePREF_S can be computed in $\mathcal{O}(n)$ time.*

Proof. We create a path from the root of T leading to a new leaf s with $label(s) = S$; let T' be the resulting tree. Then we compute the suffix tree \mathcal{T} of the tree T' . For each node u of T' , by $where(u)$ let us denote the node of \mathcal{T} with path label $label(u)$. For each node u of T' that originates from T , $\text{TreePREF}_S[u]$ is the depth of the lowest common ancestor (LCA) of $where(u)$ and $where(s)$. We use the fact that LCA queries can be answered in $\mathcal{O}(1)$ time after linear time preprocessing [6]. \square

2.2 Summing Second Heights of All Nodes

We are interested in computing sums of heights of nodes. Let us define $height(v)$ to be the number of nodes on the longest path from v to a leaf.

Remark 1. The sum of heights of all nodes can be quadratic, for example for a simple directed path.

The situation changes if we consider for each node its second highest child. Denote by $sec-height(v)$ the second largest height of a child of v (possibly $sec-height(v) = height(v) - 1$ if there are two children with the same largest height). If v has only one child then we define $sec-height(v) = 0$.

Proposition 1. $\sum_{v \in T} sec-height(v) \leq n$.

Proof. For a node v we define $MaxPath(v)$ as a longest path from v to its leaf. Initially we choose (one of possibly many) $MaxPath(root)$, then we remove this path (both nodes and edges) and choose longest paths for roots of resulting subtrees. We continue in this way and obtain a decomposition of the tree into node-disjoint longest paths; see Fig. 5.

Let $FirstChild(v)$ denote a child of v which belongs to the same path in the decomposition and $SecondChild(v)$ denote a child $w \neq FirstChild(v)$ of v of largest height. Let V' be the set of nodes with at least two children each. Then

$$\sum_v sec-height(v) = \sum_{v \in V'} |MaxPath(SecondChild(v))| \leq n,$$

since all selected longest paths are node-disjoint ($|p|$ denotes the number of nodes in a path p). \square

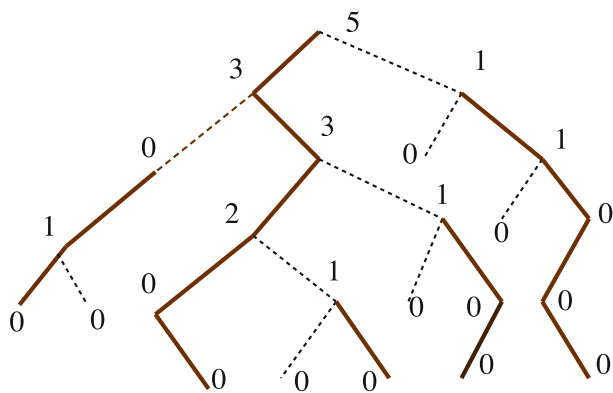


Fig. 5. A tree T with the values $sec\text{-height}(v)$. The longest paths are distinguished. Observe that $\sum_v sec\text{-height}(v) < |T|$.

3 Covers of Directed Trees

Let T be a rooted labeled tree with $n \geq 2$ nodes. Let $Subtree(v)$ denote the set of descendant nodes of v , including v . For a set M of marked nodes and nodes u, v in T we define

$$\text{Bottom}(v) = \text{a node } u \in M \cap Subtree(v) \text{ with minimal } dist(u, v),$$

$$\text{chain}(u) = \{v : \text{Bottom}(v) = u\}.$$

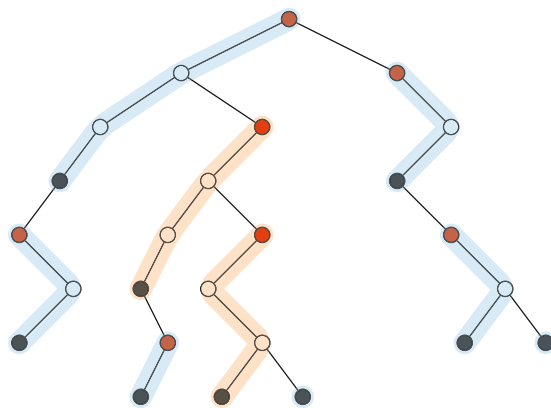


Fig. 6. Chain decomposition; each chain is the set of nodes v such that $\text{Bottom}(v) = u$, where u is its bottom node (marked node, black in the figure). Red nodes are top nodes of nontrivial chains; nodes constituting single element chains are black and red at the same time. We have $\text{maxgap} = 4$; the two chains of size 4 are drawn in orange. (Note that the chain containing the root has size 3, since the root node is not counted.) (Color figure online)

Let $u \in M$. Our algorithm keeps an invariant that $\text{chain}(u)$ is a path. We denote by $\|\text{chain}(u)\|$ the number of non-root nodes on $\text{chain}(u)$, called the *size*

of the chain. If $v \in \text{chain}(u)$, then by $\text{Top}(v)$ we denote the topmost node of $\text{chain}(u)$. We define:

$$\text{maxgap}(M) = \max\{\|\text{chain}(v)\| : v \in T\}.$$

In other words, $\text{maxgap}(M)$ is the maximal number of edges from a bottom node to the root, if the root is in the same chain, or to the lowest node in a different chain; see Fig. 6. Here $\|\text{chain}(v)\| = \infty$ if v does not belong to any chain.

Observation 2. Let S be the label of a chosen leaf-to-root path, d be a positive integer, and let $M = \{w : \text{TreePREF}_S(w) \geq d\}$. The tree T has a cover of length d if and only if $\text{maxgap}(M) \leq d$.

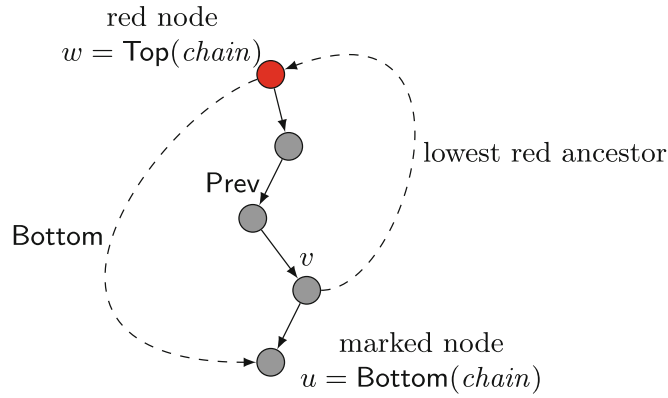


Fig. 7. Structure of a chain $u \rightsquigarrow w$. The top node w keeps the value $\text{Bottom}[w] = u$. (Color figure online)

Algorithm 1: Covers-in-directed-tree

```

 $S := \text{label}(\text{leaf}, \text{root})$  for some leaf;
Compute  $\text{TreePREF}_S[v]$  for all  $v \in T$ ;
 $m := \min\{\text{TreePREF}_S[v] : v \text{ is a leaf}\}$ ;
foreach  $v$  in  $T$  do  $\text{TreePREF}_S[v] := \min(\text{TreePREF}_S[v], m)$ ;
Initialize;
for  $d := m$  down to  $1$  do
     $NEW := \{v : \text{TreePREF}_S[v] = d\} \setminus \text{leaves}(T)$ ;
    foreach  $v$  in  $NEW$  do  $\text{MarkAndUpdate}(v)$ ;
    if  $\text{maxgap}(M) \leq d$  then Report a cover of length  $d$ ;

```

For each $u \in M$, the values $\text{len}[u] := \|\text{chain}(u)\|$ and $\text{Top}[u] := \text{Top}(u)$ are stored. Moreover, we assume that the node $w = \text{Top}[u]$ is colored in red and stores u , as $\text{Bottom}[w] := \text{Bottom}(u)$. If $v \notin M$, we store, as $\text{Prev}[v]$, the child

of v on the path from v to $\text{Bottom}(v)$. If $v \notin M$, we will compute $\text{Top}(v)$ as the lowest red ancestor of v ; this will be the bottleneck of the algorithm. See Fig. 7.

We use the algorithm Covers-in-directed-tree shown as Algorithm 1 with auxiliary functions. The algorithm considers all possible lengths d of a cover in descending order and stores the chain decomposition implied by the set M of Observation 2. Non-leaf nodes are sorted by their TreePREF_S values using buckets in $\mathcal{O}(n)$ time. To some extent this algorithm resembles Moore and Smyth’s algorithm [24,25] for computing covers of strings (working in a reversed order).

Function Initialize

Comment: Creates chains for $M = \text{leaves}(T)$

foreach v **in** T **do**

 Compute $\text{Leaf}[v]$ as the leftmost nearest leaf in the subtree of v ;

foreach leaf **in** $\text{leaves}(T)$ **do**

$\text{NewChain}(\text{leaf}, v)$ where v is the topmost node with $\text{Leaf}[v] = \text{leaf}$;

 Compute the initial $\text{Prev}[\cdot]$ values;

The function $\text{NewChain}(u, w)$ creates a chain $u \rightsquigarrow w$, whereas the function $\text{FindChain}(v) = (u \rightsquigarrow w)$ first computes $w = \text{Top}(v)$ as the lowest red ancestor of v and then computes $u = \text{Bottom}[w]$. We assume that the nodes of T are pre-ordered from left to right in Initialize.

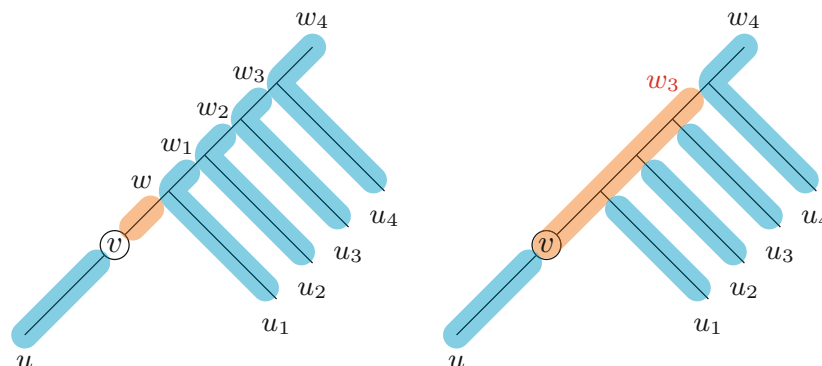


Fig. 8. Illustration of execution of one call to $\text{MarkAndUpdate}(v)$. The successive values of $\text{Top}(\text{chain}_1)$ are w, w_1, \dots, w_3 . Finally $\text{Top}(\text{chain}_1) = w_3$. The nodes $p = \text{parent}(w)$, $p_1 = \text{parent}(w_1)$, $p_2 = \text{parent}(w_2)$ are changing their chains and Prev values. For each of them chain-height (for a definition, see below) is decreasing, also their chain goes through a different child. The newly marked node v is *winning* with u_1, u_2, u_3 .

In addition to the pseudocodes, the $\text{len}[\cdot]$ array is updated whenever operations on chains are performed. Moreover, the maximum (finite value) in this array, i.e., $\text{maxgap}(M)$, never increases because a newly created chain only extends if it “wins” with an existing one (see Fig. 8). Hence, it is sufficient to store an array of n buckets containing all $\text{len}[\cdot]$ values and retain as $\text{maxgap}(M)$

Function MarkAndUpdate(v)

Comment: Inserts v to M and updates the chains decomposition; see Fig. 8
 $chain := FindChain(v)$;
 $chain_1 := NewChain(v, Top(chain))$;
 $Top(chain) := Prev[v]$; color $Top(chain)$ in red;
while $Top(chain_1) \neq root$ **do**
 $chain_2 := FindChain(parent(Top(chain_1)))$;
 if not $Crossover(chain_1, chain_2)$ **then**
 break;
 // Otherwise we say that v wins with w
 update $maxgap(M)$;

the maximum index of a non-empty bucket, which never increases. Overall this works in time linear in the number of operations on chains plus $\mathcal{O}(n)$.

To analyze the complexity of the algorithm, let us define $chain\text{-}height(v) = dist(v, Bottom(v))$.

Observation 3. *In the algorithm Covers-in-directed-tree, after a node p changes its chain in the function Crossover, we have $chain\text{-}height(p) \leq sec\text{-}height(p) + 1$. Afterwards, each time p changes its chain in the function Crossover, its chain-height decreases. Consequently, p changes its chain $\mathcal{O}(sec\text{-}height(p))$ times.*

Lemma 2. *The algorithm Covers-in-directed-tree works in $\mathcal{O}(\beta(n))$ time, where $\beta(n)$ is the cost of answering on-line n lowest red ancestor queries.*

Proof. It is enough to show that the total number of “wins” performed in the algorithm is $\mathcal{O}(n)$. Then the total number of iterations of the while-loop in the function MarkAndUpdate is linear.

Due to Observation 3 the total number of wins in the function MarkAndUpdate is amortized by the sum of all numbers $sec\text{-}height(p)$. Now the thesis follows from Proposition 1. \square

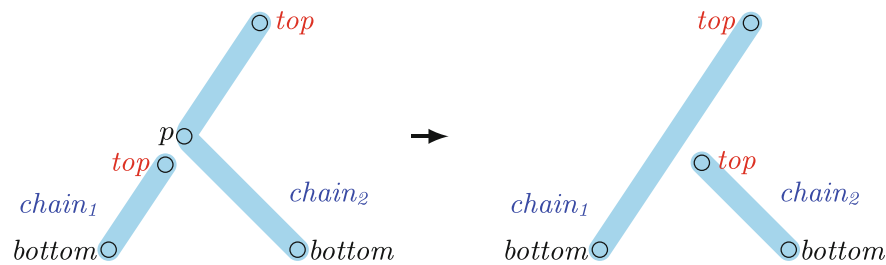


Fig. 9. Graphical illustration of one call to $Crossover(chain_1, chain_2)$.

To answer Top queries, we can use the following result from [2], where in this case the marked nodes are the red nodes.

Function Crossover($chain_1, chain_2$)

Comment: Works as shown in Fig. 9. We have $parent(\text{Top}(chain_1)) \in chain_2$.

$p := parent(\text{Top}(chain_1));$

if $dist(p, \text{Bottom}(chain_1)) \geq dist(p, \text{Bottom}(chain_2))$ **then**

return false;

else

 uncolor $\text{Top}(chain_1);$

$(\text{Top}(chain_2), \text{Prev}[p], \text{Top}(chain_1)) :=$

$(\text{Prev}[p], \text{Top}(chain_1), \text{Top}(chain_2));$

 color $\text{Top}(chain_2)$ in red; **return true;**

Fact 2. Lowest red ancestor queries for a dynamic set of marked nodes in a tree of size n can be answered in $\mathcal{O}(\log n / \log \log n)$ time.

Now, the main result of this section follows from Lemma 2 and Fact 2.

Theorem 1. All covers in a directed tree with n nodes can be computed in $\mathcal{O}(n \log n / \log \log n)$ time.

4 Covers of Undirected Trees

Covering an undirected tree is much harder than that of a directed tree. Even the case when the tree is a simple undirected path is nontrivial. For example, the shortest standard cover of a Fibonacci string $abaababaabaab$ (and the corresponding directed path) is $abaab$, however it is not true in the undirected case, where the shortest cover is ab . A serious difference between covers in directed and covers in undirected trees is shown in the generic example below.

Example 1. Take a full binary tree T_k of height k , subdivide each edge, and then label the higher edge obtained in this division with a , and the lower edge with b (Fig. 10 shows the tree T_4). T_k has $2^{k+2} - 3$ nodes. This tree has $\Omega(k) = \Omega(\log n)$ different covers of the same length (this cannot happen for covers of strings).

4.1 $\mathcal{O}(n^2)$ Time and Space Solution

We say that a set M of simple paths in a tree T covers T if each edge of T is on some path in M . We define an auxiliary problem that will be used to test candidates for a cover.

Strips Covering Problem

Input: A set M of simple paths in an undirected tree T (given by their endpoints).

Output: YES if M covers T , NO otherwise.

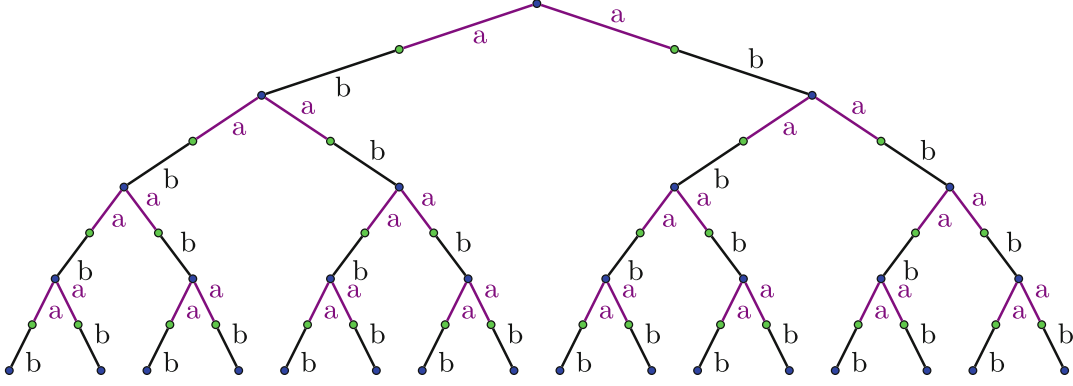


Fig. 10. The undirected tree T_4 . It has 44 distinct covers and there are many distinct aperiodic covers of the same length. Generally there are $k(2k+3)$ covers in T_k . The label of every path that starts in a leaf, has length at least two and if it ends with an edge with a letter b , then it has a subpath with label aa , is a cover of T_k . The reverses of such labels are also covers. There are 4 distinct covers of length 6: $bababa, ababab, babaab, baabab$.

Lemma 3. *The Strips Covering Problem can be solved in $\mathcal{O}(|T| + |M|)$ time.*

Proof. Let us root T in an arbitrarily chosen node. We can reduce the problem to the case when each path $\pi \in M$ is a bottom-up path. For a path $\pi = u \rightsquigarrow v$ we compute $w = LCA(u, v)$ and replace π by two paths $u \rightsquigarrow w, v \rightsquigarrow w$. This reduction works in linear time [6].

We use counters for nodes. For each node w initially $count[w] = 0$. Next for each non-empty bottom-up path (u, w) we set $count[u] += 1, count[w] -= 1$. An edge $(w, parent(w))$ is covered by M if and only if the sum of counters of all descendants of w (including w) is positive. These sums can be easily computed in a bottom-up manner within the required complexity. In the end M covers T if and only if all edges are covered. \square

Lemma 4. *An undirected labeled tree with n nodes has at most $2n - 2$ covers.*

Proof. Take any leaf; the only edge connected to this leaf must be covered by a first or last letter of the cover. An occurrence of the cover must appear on some path in the tree, and each such path is determined by the two nodes on its ends. As one end is fixed (the chosen leaf), there can only be $n - 1$ such paths. This gives at most $2(n - 1)$ possible cover candidates. \square

Theorem 2. *All covers of an undirected tree with n nodes can be computed in $\mathcal{O}(n^2)$ time and space.*

Proof. We group all $\mathcal{O}(n^2)$ paths in the tree by their string labels, and for each of the $k = \mathcal{O}(n)$ candidates from Lemma 4 check if all paths in the group with the same string label as the candidate cover the whole tree. If M_1, \dots, M_k are these sets of paths, then the solution using Lemma 3 works in $\mathcal{O}(kn + \sum_{i=1}^k |M_i|) = \mathcal{O}(n^2)$ time since the sets M_i are pairwise disjoint.

For grouping paths by their string labels we assign identifiers in $\{1, \dots, n^2\}$ to all paths using the following algorithm:

- The labels of edges are renumbered with integers in the range $\{1, \dots, n-1\}$ in $\mathcal{O}(n \log n)$ time.
- Given the identifiers of paths of length i , we compute paths of length $i+1$ and assign identifiers to them by forming pairs of the form: (the identifier of prefix path of length i , the identifier of the last edge), sorting them by radix sort and renumbering with integers from $\{1, \dots, n^2\}$. If p_{i+1} is the number of paths of length $i+1$, then this step takes $\mathcal{O}(p_{i+1} + n)$ time.

This gives $\mathcal{O}(n \log n + \sum_{i=2}^n (p_i + n)) = \mathcal{O}(n^2)$ time and space. \square

4.2 $\mathcal{O}(n^2 \log n)$ Time and $\mathcal{O}(n)$ Space

Let us recall that there are $\mathcal{O}(n)$ candidates for a cover (see Lemma 4). In this section we show how to check if each of them is a cover in $\mathcal{O}(n \log n)$ time. We use the following auxiliary problem together with a centroid decomposition.

Anchored Covering Problem

Input: Labeled tree T , its node r and a string C ; $|T| = n$, $|C| = m$.

Output: All edges of T that are covered by occurrences of C in T that pass through the node r .

Lemma 5. *Anchored Covering Problem for a tree with n nodes can be solved in $\mathcal{O}(n)$ time.*

Proof. Let us root T in r . We denote by T_1, T_2, \dots, T_t the subtrees of r . For each $i \in \{1, \dots, t\}$, we define P_i, S_i as the set of nodes $v \in T_i$ such that the label of the path $v \rightsquigarrow r$ is a prefix and a reverse of a suffix of C , respectively. For $v \in T_i$ we have $v \in P_i$ ($v \in S_i$) if $\text{TreePREF}_C[v] = \text{depth}(v)$ ($\text{TreePREF}_{C^R}[v] = \text{depth}(v)$), respectively, where $\text{depth}(v)$ denotes the distance of v from the root r), so these sets can be computed in $\mathcal{O}(n)$ time (cf. Lemma 1).

A node v is called **good** if it is an endpoint of an undirected path labeled C and passing through r .

Claim 1. $v \in T_i$ is good if there is a node $u \in T_j, j \neq i$ such that

$$(\text{depth}(u) + \text{depth}(v) = m) \text{ and } (u \in S_j, v \in P_i \text{ or } u \in P_j, v \in S_i).$$

We use the function `MarkNodes` to mark good nodes. In the pseudocode we denote by $\text{depths}(W)$ a list of depths of nodes in W . We represent each of the sets $\text{PrefDepths}, \text{SufDepths}$ by its characteristic vector so that adding a set $\text{depths}(W)$ to them can be done in time proportional to the number of inserted elements.

In the algorithm `AnchoredCovering` we are implementing the claim and process i 's in the ascending and descending orders; in the first pass $\text{PrefDepths}, \text{SufDepths}$ consist of depths of P_j, S_j of $u \in T_j$ for all $j < i$ and in the second pass for $j > i$. Each pass works in $\mathcal{O}(n)$ time. In the end all edges that are

Function MarkNodes(i)

Comment: (some) good nodes in the subtree T_i are marked

```

foreach  $v$  in  $P_i$  do
  if  $m - \text{depth}(v) \in \text{SufDepths}$  then
    mark  $v$  as a good node;
foreach  $v$  in  $S_i$  do
  if  $m - \text{depth}(v) \in \text{PrefDepths}$  then
    mark  $v$  as a good node;
 $\text{PrefDepths} := \text{PrefDepths} \cup \text{depths}(P_i)$ ;
 $\text{SufDepths} := \text{SufDepths} \cup \text{depths}(S_i)$ ;

```

Algorithm 2: AnchoredCovering

```

 $\text{PrefDepths} := \text{SufDepths} := \emptyset$ ;
for  $i := 1$  to  $t$  do MarkNodes( $i$ );
 $\text{PrefDepths} := \text{SufDepths} := \emptyset$ ;
for  $i := t$  down to  $1$  do MarkNodes( $i$ );
return all edges on paths from marked nodes to  $r$ ;

```

located on paths $v \rightsquigarrow \text{root}$ for good nodes v are computed in $\mathcal{O}(n)$ time with a simple bottom-up traversal. \square

Let T_1, T_2, \dots, T_t be the connected components obtained after removing a node r from T . The node r is called a *centroid* of T if $|T_i| \leq n/2$ for all T_i .

The *centroid decomposition* of T , $CDecomp(T)$, is defined recursively as:

$$CDecomp(T) = \{(T, r)\} \cup \bigcup_{i=1}^t CDecomp(T_i).$$

Every tree has a centroid, see [15], and a centroid of a tree can be computed in $\mathcal{O}(n)$ time.

The recursive definition of $CDecomp(T)$ implies the following bounds.

Fact 3. For a tree T with n nodes, the total size of all subtrees in $CDecomp(T)$ is $\mathcal{O}(n \log n)$. The decomposition $CDecomp(T)$ can be computed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ additional space (excluding the space used for storing the output).

Theorem 3. One can check if a given string covers an undirected tree with n nodes in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.

Proof. A known property of a centroid decomposition is that for every path π in T there exists an element $(T', r') \in CDecomp(T)$ such that π is a path in T' that passes through r' . We generate the pairs (T', r') forming $CDecomp(T)$

one by one, for each of them solve an instance of the Anchored Covering Problem, and mark all edges returned by each instance. The complexity follows by Fact 3. \square

Corollary 1. *All covers of an undirected tree can be computed in $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n)$ space.*

References

1. Alatabbi, A., Rahman, M.S., Smyth, W.F.: Computing covers using prefix tables. *Discret. Appl. Math.* **212**, 2–9 (2016). <https://doi.org/10.1016/j.dam.2015.05.019>
2. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: 39th Annual Symposium on Foundations of Computer Science, FOCS '98, pp. 534–544. IEEE Computer Society, Palo Alto, California, USA (1998). <https://doi.org/10.1109/SFCS.1998.743504>
3. Antoniou, P., Crochemore, M., Iliopoulos, C.S., Jayasekera, I., Landau, G.M.: Conservative string covering of indeterminate strings. In: Holub, J., Zdárek, J. (eds.) *Proceedings of the Prague Stringology Conference 2008*, Prague, Czech Republic, 1–3 September 2008, pp. 108–115. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague (2008). <http://www.stringology.org/event/2008/p10.html>
4. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings. *Inf. Process. Lett.* **39**(1), 17–20 (1991). [https://doi.org/10.1016/0020-0190\(91\)90056-N](https://doi.org/10.1016/0020-0190(91)90056-N)
5. Barton, C., Kociumaka, T., Liu, C., Pissis, S.P., Radoszewski, J.: Indexing weighted sequences: neat and efficient. *Inf. Comput.* **270**, 104462 (2020). <https://doi.org/10.1016/j.ic.2019.104462>
6. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. In: Rajsbaum, S. (ed.) *LATIN 2002*. LNCS, vol. 2286, pp. 508–515. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45995-2_44
7. Breslauer, D.: An on-line string superprimitivity test. *Inf. Process. Lett.* **44**(6), 345–347 (1992). [https://doi.org/10.1016/0020-0190\(92\)90111-8](https://doi.org/10.1016/0020-0190(92)90111-8)
8. Brlek, S., Lafrenière, N., Provençal, X.: Palindromic complexity of trees. In: Potapov, I. (ed.) *DLT 2015*. LNCS, vol. 9168, pp. 155–166. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21500-6_12
9. Charalampopoulos, P., Radoszewski, J., Rytter, W., Waleń, T., Zuba, W.: Computing covers of 2D-strings. In: Gawrychowski, P., Starikovskaya, T. (eds.) *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, 5–7 July 2021, Wrocław, Poland. LIPIcs, vol. 191, pp. 12:1–12:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPIcs.CPM.2021.12>
10. Crochemore, M., et al.: The maximum number of squares in a tree. In: Kärkkäinen, J., Stoye, J. (eds.) *CPM 2012*. LNCS, vol. 7354, pp. 27–40. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31265-6_3
11. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Covering problems for partial words and for indeterminate strings. *Theor. Comput. Sci.* **698**, 25–39 (2017). <https://doi.org/10.1016/j.tcs.2017.05.026>
12. Funakoshi, M., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Computing maximal palindromes and distinct palindromes in a trie. In: Holub, J., Zdárek, J. (eds.) *Prague Stringology Conference 2019*, Prague, Czech Republic, 26–28

- August 2019, pp. 3–15. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science (2019). <http://www.stringology.org/event/2019/p02.html>
13. Gawrychowski, P., Kociumaka, T., Rytter, W., Waleń, T.: Tight bound for the number of distinct palindromes in a tree. In: Iliopoulos, C.S., Puglisi, S., Yilmaz, E. (eds.) SPIRE 2015. LNCS, vol. 9309, pp. 270–276. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23826-5_26
 14. Gawrychowski, P., Kociumaka, T., Rytter, W., Waleń, T.: Tight bound for the number of distinct palindromes in a tree. CoRR **abs/2008.13209** (2020). [arXiv:2008.13209](https://arxiv.org/abs/2008.13209)
 15. Harary, F.: Graph Theory. Reading, Addison-Wesley, Boston, MA (1994)
 16. Iliopoulos, C.S., Mohamed, M., Mouchard, L., Perdikuri, K., Smyth, W.F., Tsakalidis, A.K.: String regularities with don't cares. Nordic J. Comput. **10**(1), 40–51 (2003)
 17. Kikuchi, N., Hendrian, D., Yoshinaka, R., Shinohara, A.: Computing covers under substring consistent equivalence relations. In: Boucher, C., Thankachan, S.V. (eds.) SPIRE 2020. LNCS, vol. 12303, pp. 131–146. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59212-7_10
 18. Kociumaka, T., Pachocki, J., Radoszewski, J., Rytter, W., Waleń, T.: Efficient counting of square substrings in a tree. Theor. Comput. Sci. **544**, 60–73 (2014). <https://doi.org/10.1016/j.tcs.2014.04.015>
 19. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: String powers in trees. Algorithmica **79**(3), 814–834 (2017). <https://doi.org/10.1007/s00453-016-0271-3>
 20. Kociumaka, T., Radoszewski, J., Wiśniewski, B.: Subquadratic-time algorithms for abelian stringology problems. In: Kotsireas, I.S., Rump, S.M., Yap, C.K. (eds.) MACIS 2015. LNCS, vol. 9582, pp. 320–334. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-32859-1_27
 21. Kociumaka, T., Radoszewski, J., Wiśniewski, B.: Subquadratic-time algorithms for abelian stringology problems. AIMS Med. Sci. **4**(3), 332–351 (2017). <https://doi.org/10.3934/ms.2017.3.332>
 22. Kosaraju, S.R.: Efficient tree pattern matching (preliminary version). In: 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October–1 November 1989, pp. 178–183. IEEE Computer Society (1989). <https://doi.org/10.1109/SFCS.1989.63475>
 23. Matsuda, S., Inenaga, S., Bannai, H., Takeda, M.: Computing abelian covers and abelian runs. In: Holub, J., Zdárek, J. (eds.) Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, 1–3 September 2014, pp. 43–51. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague (2014). <http://www.stringology.org/event/2014/p05.html>
 24. Moore, D.W.G., Smyth, W.F.: An optimal algorithm to compute all the covers of a string. Inf. Process. Lett. **50**(5), 239–246 (1994). [https://doi.org/10.1016/0020-0190\(94\)00045-X](https://doi.org/10.1016/0020-0190(94)00045-X)
 25. Moore, D.W.G., Smyth, W.F.: A correction to “An optimal algorithm to compute all the covers of a string”. Inf. Process. Lett. **54**(2), 101–103 (1995). [https://doi.org/10.1016/0020-0190\(94\)00235-Q](https://doi.org/10.1016/0020-0190(94)00235-Q)
 26. Popa, A., Tanasescu, A.: An output-sensitive algorithm for the minimization of 2-dimensional string covers. In: Gopal, T.V., Watada, J. (eds.) TAMC 2019. LNCS, vol. 11436, pp. 536–549. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-14812-6_33

27. Radoszewski, J., Straszyński, J.: Efficient computation of 2-covers of a string. In: Grandoni, F., Herman, G., Sanders, P. (eds.) 28th Annual European Symposium on Algorithms, ESA 2020, 7–9 September 2020, Pisa, Italy (Virtual Conference). LIPIcs, vol. 173, pp. 77:1–77:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/LIPIcs.ESA.2020.77>
28. Shibuya, T.: Constructing the suffix tree of a tree with a large alphabet. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E86–A(5)**, 1061–1066 (2003)
29. Sugahara, R., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Computing runs on a trie. In: Pisanti, N., Pissis, S.P. (eds.) 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18–20, 2019, Pisa, Italy. LIPIcs, vol. 128, pp. 23:1–23:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). <https://doi.org/10.4230/LIPIcs.CPM.2019.23>

Efficient Computation of Sequence Mappability

Panagiotis Charalampopoulos · Costas S.
Iliopoulos · Tomasz Kociumaka · Solon
P. Pissis · Jakub Radoszewski · Juliusz
5 Straszynski*

Abstract Sequence mappability is an important task in genome resequencing. In the (k, m) -mappability problem, for a given sequence T of length n , the goal is to compute a table whose i th entry is the number of indices $j \neq i$ such that the length- m substrings of T starting at positions i and j have at most k mismatches. Previous works on this problem focused on heuristics computing a rough approximation of the result or on the case of $k = 1$. We present several efficient algorithms for the general case of the problem. Our main result is an algorithm that, for $k = O(1)$, works in $O(n)$ space and, with high probability, in $O(n \cdot \min\{m^k, \log^k n\})$ time. Our algorithm requires a careful adaptation of the k -errata trees of Cole et al. [STOC 2004] to avoid multiple counting of pairs of substrings. Our technique can also be applied to solve the all-pairs Hamming distance problem introduced by Crochemore et al. [WABI 2017]. We further develop $O(n^2)$ -time algorithms to compute all (k, m) -mappability tables

* Corresponding author

Panagiotis Charalampopoulos
The Interdisciplinary Center Herzliya, Herzliya, Israel
E-mail: panagiotis.charalampopoulos@post.idc.ac.il

Costas S. Iliopoulos
Department of Informatics, King's College London, London, UK
E-mail: c.iliopoulos@kcl.ac.uk

Tomasz Kociumaka
University of California, Berkeley, USA
E-mail: kociumaka@berkeley.edu

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands
E-mail: solon.pissis@cwi.nl

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Warsaw, Poland
E-mail: jrad@mimuw.edu.pl

Juliusz Straszynski*
Institute of Informatics, University of Warsaw, Warsaw, Poland
E-mail: jks@mimuw.edu.pl

for a fixed m and all $k \in \{0, \dots, m\}$ or a fixed k and all $m \in \{k, \dots, n\}$. Finally, we show that, for $k, m = \Theta(\log n)$, the (k, m) -mappability problem cannot be solved in strongly subquadratic time unless the Strong Exponential Time Hypothesis fails.

This is an improved and extended version of a paper presented at SPIRE 2018.

Keywords sequence mappability · k -errata tree · Hamming distance

1 Introduction

The k -mappability problem. Analyzing data derived from massively parallel sequencing experiments often depends on the process of genome assembly via resequencing; namely, assembly with the help of a reference sequence. In this process, a large number of reads (or short sequences) derived from a DNA donor during these experiments must be mapped back to a reference sequence, comprising a few gigabases, to establish the section of the genome from which each read has been derived. An extensive number of short-read alignment techniques and tools have been introduced to address this challenge emphasizing on different aspects of the process [16].

In turn, the process of resequencing depends heavily on how mappable a genome is with respect to reads of some fixed length m . Thus, given a reference sequence, for every substring of length m in the sequence, we want to count how many additional times this substring appears in the sequence when allowing for a small number k of errors. This computational problem and a heuristic approach to approximate the solution were first proposed in [12] (see also [5]), where a great variance in genome mappability between species and gene classes was revealed.

More formally, for a string T , let T_i^m denote the length- m substring of T that starts at position i . In the (k, m) -mappability problem, for a given string T of length n , we are asked to compute a table $A_{\leq k}^m$ whose i th entry $A_{\leq k}^m[i]$ is the number of indices $j \neq i$ such that the substrings T_i^m and T_j^m are at Hamming distance at most k . In the previous study [12], the assumed values of parameters were $k \leq 4$, $m \leq 100$, and the alphabet of T was $\{A, C, G, T\}$.

Example 1.1 Consider a string $T = \text{aababba}$ and $m = 3$. The following table shows the (k, m) -mappability counts for $k = 1$ and $k = 2$.

position	i	1	2	3	4	5
substring	T_i^3	aab	aba	bab	abb	bba
(1, 3)-mappability	$A_{\leq 1}^3[i]$	2	2	1	2	1
(2, 3)-mappability	$A_{\leq 2}^3[i]$	3	3	3	4	3
difference	$A_{=2}^3[i]$	1	1	2	2	2

For instance, consider the position 1. The (1, 3)-mappability is 2 due to the occurrences of bab and abb at positions 3 and 4, respectively. The (2, 3)-mappability is 3 since only the substring bba, occurring at position 5, has three mismatches with aab.

For convenience, our algorithms compute an array $A_{=k}^m$ whose i th entry $A_{=k}^m[i]$ is the number of positions $j \neq i$ such that substrings T_i^m and T_j^m are at Hamming distance *exactly* k . Note that $A_{\leq k}^m[i] = \sum_{\kappa=0}^k A_{=\kappa}^m[i]$; see the “difference” row in the example above. Henceforth, we call this problem *the (k, m) -mappability problem*.

Solution	Time complexity
Manzini [31]	$O(mn \log n / \log \log n)$
Alzamel et al. [3]	$O(nm)$
Alzamel et al. [3]	$O(n \log n \log \log n)$
Alzamel et al. [3]	$O(n)$ on average for $m = \Omega(\log n)$
Hooshmand et al. [21], Amir et al. [4]	$O(n \log n)$
Amir et al. [4]	$O(n)$ for $m = \Omega(\sqrt{n})$

Table 1.1 Known algorithms for computing $(1, m)$ -mappability for strings over constant-sized alphabets. All algorithms use $O(n)$ space.

55 Using the suffix array and the LCP table [30,26,24], the $(0, m)$ -mappability problem can be solved in $O(n)$ time and space. Known solutions for computing $(1, m)$ -mappability are shown in Table 1.1; the $O(nm)$ -time and the $O(n)$ -average-time solutions of Alzamel et al. [3] work also on strings over *integer alphabets* $\{1, \dots, \sigma\}$ for $\sigma = n^{O(1)}$. Moreover, the latter algorithm was shown to be generalizable to arbitrary k ,
60 requiring $O(n)$ space and, on average, $O(kn)$ time if $m = \Omega(k \log_\sigma n)$. A practically fast algorithm for arbitrary k was presented in [32]. In [1], the authors introduced an efficient construction of a *genome mappability array* B_k in which $B_k[\mu]$ is the smallest length m such that at least μ of the length- m substrings of T do not occur elsewhere in T with at most k mismatches. This construction was further improved in [6].

65 *The all-pairs Hamming distance problem.* The evolutionary relationships between different species or taxa are usually inferred through phylogenetic analysis techniques. Some of these techniques rely on the inference of phylogenetic trees. A first step of these techniques is to compute the distances between all pairs of sequences representing the set of species or taxa under study [35]. This particular step, however,
70 often dominates the running time of these methods. Depending on the application, the underlying model of evolution, and the optimality criterion, it may not be strictly necessary to be aware of the complete distance matrix (see [17,11], for instance). Thus, in this preprocessing step, we are only interested in pairs with distances not exceeding a given threshold.

75 The computational problem can be formally defined as follows. Given a set \mathbf{R} of r length- m strings and an integer $k \in \{0, \dots, m\}$, return all pairs $(X_1, X_2) \in \mathbf{R} \times \mathbf{R}$, with $X_1 \neq X_2$, such that X_1 and X_2 are at Hamming distance at most k . This problem has been studied in the average-case model and efficient linear-time algorithms are known under some constraints on the value of k and some assumptions on the elements of \mathbf{R} [11,29,20]. In particular, these algorithms work in $O(rm)$ average-case time
80 if $k < \frac{(m-k-1) \log \sigma}{\log rm}$ and the elements of \mathbf{R} are over an integer alphabet Σ of size $\sigma > 1$ with the letters of the strings being independent and identically distributed random variables uniformly distributed over Σ . The indexing variant of the all-pairs

Hamming distance problem has further applications in bioinformatics for querying
 85 typing databases [8] and in information retrieval for searching similar documents in a
 collection [19].

Intuitively, there is a connection between the (k, m) -mappability problem and the
 all-pairs Hamming distance problem that allows to transfer the technique used in the
 solution to the former to a solution to the latter (it is not a formal reduction between
 90 problems). The connection is as follows: by first concatenating the r elements of \mathbf{R} to
 construct a new string T of length $n = rm$, solving the former considering only the r
 substrings of T starting at positions i with $i \bmod m = 1$, and summing up the resulting
 values, we would obtain the total size of the output of the latter.

Henceforth, we assume, as in the mappability problem, that we are to compute all
 95 pairs at Hamming distance *exactly* k . In the end, we run the algorithm for all values of
 k up to a given threshold of interest.

Our contributions. We present several algorithms for the general case of the (k, m) -
 mappability problem. More specifically, our contributions are as follows:

1. In Section 3, we show a randomized Las-Vegas algorithm for the (k, m) -mappa-
 100 bility problem that works in $O(n \binom{\log n + k}{k} 4^k k)$ time with high probability¹ and
 $O(n 2^k k)$ space for a string over any ordered alphabet. It requires a careful adapta-
 tion of the technique of recursive heavy-path decompositions in a tree [10].
2. In Section 4, we show an algorithm to solve the all-pairs Hamming distance prob-
 lem for strings over any ordered alphabet that works in $O(rm + r \binom{\log r + k}{k} 4^k k \log r +$
 105 $\text{output} \cdot 2^k k \log r)$ time and $O(rm + r 2^k k \log r)$ space.
3. In Section 5, we show an algorithm for the (k, m) -mappability problem that works
 in $O(nk \cdot (m + 1)^k)$ time and $O(n)$ space for a string over an integer alphabet.
 Together with the first result, this yields an $O(n \cdot \min\{m^k, \log^k n\})$ -time and $O(n)$ -
 space algorithm for $k = O(1)$.
- 110 4. In Section 6, we show $O(n^2)$ -time algorithms for a string over any ordered alphabet
 to compute *all* (k, m) -mappability tables for a fixed m and all $k \in \{0, \dots, m\}$, or
 for a fixed k and all $m \in \{k, \dots, n\}$.
5. Finally, in Section 7, we prove that the (k, m) -mappability problem for $k, m =$
 $\Theta(\log n)$ cannot be solved in strongly subquadratic time unless the Strong Expon-
 115 nential Time Hypothesis [22, 23] fails.

In contributions 1 and 5, we apply recent advances in the Longest Common Substring
 with k Mismatches problem that were presented in [9] and [27], respectively (see
 also [34]). In particular, compared to [9], our contribution 1 requires a careful counting
 of substring pairs to avoid multiple counting and a thorough analysis of the space
 120 usage. Technically this is the most involved contribution. Contributions 1, 2, and 4
 apply to strings over an arbitrary ordered alphabet; the running times of the respective
 algorithms are $\Omega(n \log n)$, which is sufficient to renumber the letters of the input text
 so that its alphabet becomes an integer alphabet.

This work is an extended version of [2]. In comparison to the conference version,
 125 in particular, we improve the complexity of the main algorithm by a $\Theta(\log n)$ -factor,

¹ With probability at least $1 - n^{-c}$ for an arbitrarily large predefined constant parameter $c > 0$.

remove the dependency on the alphabet size in contribution 3, and apply our techniques to solve the all-pairs Hamming distance problem (contribution 2).

2 Preliminaries

Let $T = T[1]T[2] \cdots T[n]$ be a *string* of length $|T| = n$ over a finite ordered alphabet Σ of size $|\Sigma| = \sigma$. The empty string is denoted by ε . In some algorithms we assume that the string is over an *integer alphabet*, i.e., $\Sigma = \{1, \dots, n^{O(1)}\}$. For two positions i and j on T , the *substring* (sometimes called *factor*) of T that starts at position i and ends at position j is $T[i] \cdots T[j]$ (it is of length 0 if $j < i$). A *prefix* of T is a substring that starts at position 1 and a *suffix* of T is a substring that ends at position n . We denote the suffix that starts at position i by T_i and its prefix of length m by T_i^m .

The *Hamming distance* between two strings S and T of the same length $|S| = |T|$ is defined as $d_H(S, T) = |\{i \in \{1, 2, \dots, |S|\} : S[i] \neq T[i]\}|$. If $|S| \neq |T|$, we set $d_H(S, T) = \infty$.

By $\text{lcp}(U, V)$ we denote the length of the longest common prefix of strings U and V . For a fixed string T , we also set $\text{lcp}(r, s) = \text{lcp}(T_r, T_s)$.

Compact trie. A *trie* of a collection of strings C is a labeled tree that contains a node for every distinct prefix of a string in C ; the root node is ε ; the set of *terminal* nodes is C ; and edges are of the form $u \xrightarrow{c} uc$, where u and uc are nodes and $c \in \Sigma$. A compact trie \mathbf{T} of a collection of strings C is obtained from the trie of C by dissolving all non-branching nodes, excluding the root and the terminals. The nodes of the trie which become nodes of \mathbf{T} are called *explicit* nodes, whereas the other nodes are called *implicit*. Each edge of \mathbf{T} can be viewed as an upward maximal path of implicit nodes starting with an explicit node. The string label of an edge is a substring of one of the strings in C ; the label of an edge is the first letter of the edge's string label. Each node of the trie can be represented in \mathbf{T} by the edge it belongs to and an index within the corresponding path. We let $\mathbf{L}(v)$ denote the *path-label* of a node v , i.e., the concatenation of the string labels of the edges along the path from the root to v . Additionally, $\mathbf{D}(v) = |\mathbf{L}(v)|$ is the *string-depth* of node v .

Suffix tree. The suffix tree of a string T is the compact trie representing all suffixes of T . The suffix tree of a string T of length n over an integer alphabet can be constructed in $O(n)$ time [14] and, after an $O(n)$ -time preprocessing [7], it can be used to answer $\text{lcp}(r, s)$ queries in $O(1)$ time.

Hashing. We use perfect hashing to implement dynamic dictionaries supporting insertions and deletions of entries (key-value pairs), as well as look-ups of entries with a given key. Technically, we maintain a single global dictionary (which may simulate multiple local dictionaries) implemented using the following result originating from the work of Dietzfelbinger and Meyer auf der Heide [13].

Theorem 2.1 (see [13, Theorem 5.5]) *For any constant $c > 1$ and positive integer n , there is a data structure that maintains a dynamic dictionary \mathbf{D} of size $|\mathbf{D}| \leq n$ with the following guarantees:*

1. The data structure occupies $O(n)$ space.
2. Handling any $m \leq n^c$ operations (look-ups, insertions, and deletions) in an on-line fashion costs $O(n+m)$ time in total² with probability at least $1 - n^{-c}$.

The constants in the time and space bounds depend on c .

170 The original data structure of [13, Theorem 5.5] supports n operations in $O(n)$ time with probability at least $1 - n^{-c}$. To allow $m \leq n^c$ operations, we use instances supporting $2n$ operations in $O(n)$ time with probability at least $1 - n^{1-2c}$, and we build such an instance from scratch after completing every n operations (using $|\mathbf{D}| \leq n$ insertions out of the allowance of $2n$ operations). By the union bound, all $m \leq n^c$ operations are thus handled in $O(n+m)$ total time with probability at least $1 - n^{-c}$.

175 When using strings as dictionary keys, we rely on Karp–Rabin fingerprints (polynomial hashing) [25] with collision probability bounded by n^{-C} for strings of length at most n (and a sufficiently large constant C). In order to obtain Las-Vegas algorithms, we provide mechanisms for detecting collisions and resort to naive polynomial-time solutions upon detecting any.

180

3 Computing Mappability in $O(n \log^k n)$ Time and $O(n)$ Space

Our algorithm operates on so-called *modified strings*. A modified string α is a pair $(U(\alpha), M(\alpha))$, where $U(\alpha)$ is a string and $M(\alpha)$ a set of modifications. Each element of the set $M(\alpha)$ is a pair of the form (i, c) which denotes a substitution “ $U(\alpha)[i] := c$ ”.

185 We assume that no two pairs in $M(\alpha)$ share the same index i . By $val(\alpha)$, we denote the string $U(\alpha)$ after all the substitutions. The sets $M(\alpha)$ for modified strings are implemented as (functional) lists. Whenever a modified string β is obtained by introducing an extra modification to a modified string α , the head of $M(\beta)$ represents the new modification whereas the tail points to $M(\alpha)$. We always introduce modifications

190 in the left-to-right order so that the lists $M(\alpha)$ are sorted according to the decreasing order of indices i .

The algorithm processes *modified substrings* of T that are modified strings originating from the substrings T_i^m . In this case, the strings $U(\alpha)$ are not stored explicitly. Instead, for a modified substring α originating from T_i^m , an index $idx(\alpha) = i$ is stored.

195 *Overview of the algorithm.* Intuitively, the algorithm proceeds by efficiently simulating transformations of a compact trie of modified substrings, initially containing all substrings T_i^m .³ The elementary transformations are guided by the *smaller-to-larger* principle, and each of them consists in copying one subtree unto its sibling, with an appropriate modification introduced to each copied substring in order to match the label of the edge leading to the sibling. This process effectively results in registering

200 one mismatch for a large batch of substrings at once, and therefore lays a foundation to solve the main problem in the aforementioned time.

² Through standard deamortization, we could achieve, with high probability, $O(1)$ -time operations after $O(n)$ -time initialization. However, this would not benefit the main results of this paper.

³ The true course of the algorithm will not actually perform much of its operations on a compact trie, but the intuition is best conveyed by visualizing them this way.

The trie is constructed top-down recursively, and the final set of modified substrings that are present in the trie is known only when all the leaves of the trie have been reached.

205

A node v of the trie stores a set of modified substrings $MS(v)$. Initially, the root r stores all substrings T_i^m in its set $MS(r)$. The path-label $\mathbf{L}(v)$ is the longest common prefix of (the values of) all the modified substrings in $MS(v)$ and the string-depth $\mathbf{D}(v)$ is the length of this prefix. None of the strings in $MS(v)$ contains a modification at a position greater than $\mathbf{D}(v)$. The children of v are determined by subsets of $MS(v)$ that correspond to different letters at position $\mathbf{D}(v) + 1$. Furthermore, additional modified substrings with modifications at position $\mathbf{D}(v) + 1$ are created and inserted into the children's MS -sets. This corresponds to the intuition of copying subtrees unto their siblings; see Fig. 3.1.

210

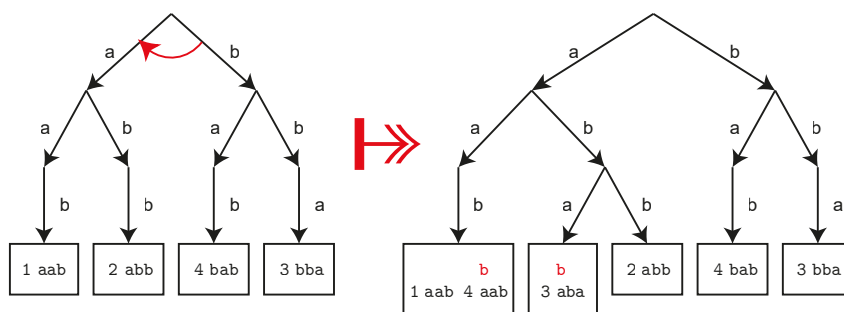


Fig. 3.1 To the left: a trie of all length-3 substrings of `aabbab`. To the right: an effect of copying the right subtree unto the left subtree, which corresponds to changing the first letter of all its substrings from `b` to `a`. In the original trie, there was exactly one pair of substrings from different subtrees of the root at Hamming distance 1; after the operation, there is a leaf containing modified substrings corresponding to these substrings. Such copy operations are performed in our algorithm top-down in the trie, making sure that each resulting modified substring has at most k modifications.

The goal is to propagate the modified substrings to the leaves and, by processing each leaf independently, register exactly once every pair of substrings (T_i^m, T_j^m) differing on exactly k positions.

215

Now, we will describe the recursive routine for visiting a node.

Processing an internal node. Assume that our node v has children u_1, \dots, u_a . First, we distinguish a child of v with maximum-size set MS , with ties being broken arbitrarily; let it be u_1 . We will refer to this child as *heavy* and to every other as *light*. We will recursively branch into each child to take care of all pairs of modified substrings contained in any single subtree.

220

For this, we create an extra child u_{a+1} so that $MS(u_{a+1})$ contains all modified substrings from $MS(u_2) \cup \dots \cup MS(u_a)$ with the letters at position $\mathbf{D}(v) + 1$ replaced by

225

a common wildcard character \$. By processing the subtree of u_{a+1} , we will consider pairs of modified substrings that originate from different light children.

230 Additionally, we insert all modified substrings from $MS(u_2) \cup \dots \cup MS(u_a)$ into $MS(u_1)$, substituting the letter at position $\mathbf{D}(v) + 1$ with the common letter at this position of modified substrings in $MS(u_1)$. This transformation will take care of pairs between the heavy child and the light ones.

As modified substrings with more than k substitutions are irrelevant for our algorithm, we refrain from creating them in the interest of time and space complexity.

235 Finally, the algorithm branches into the subtrees of u_1, \dots, u_{a+1} . A pseudocode of this process is presented as Algorithm 1.

Let us note that, in the special case of a binary alphabet, the child u_{a+1} need not be created. Indeed, in this case, each node has at most two children, hence at most one light one, whereas when processing the subtree of u_{a+1} , we consider pairs of modified substrings that originate from different light children.

Algorithm 1: A recursive procedure of processing a trie node

```

Procedure processNode( $v$ )
  lcp( $v$ ): computes the longest common prefix of all the strings in  $\{val(\alpha) : \alpha \in MS(v)\}$ 
  insert( $v, \alpha$ ): inserts  $\alpha$  into  $MS(v)$ 
  splitByLetter( $v, index$ ): splits  $MS(v)$  into groups having the same index-th letter,
    returning a list of sets of modified substrings

  depth  $\leftarrow$  lcp( $v$ )
  if depth =  $m$  then
    processLeaf( $v$ )
    return
  children  $\leftarrow$  splitByLetter( $v, depth + 1$ )
  heavyChild  $\leftarrow$  findHeaviest(children)
  heavyLetter  $\leftarrow$  val( $\alpha$ )[depth+1] for some  $\alpha \in$  heavyChild
  wildcardTree  $\leftarrow$   $\emptyset$ 
  foreach lightChild  $\in$  children  $\setminus$  {heavyChild} do
    foreach  $\alpha \in$  lightChild do
      if  $|M(\alpha)| < k$  then
         $\alpha' \leftarrow \alpha$ 
         $\alpha'$ [depth+1]  $\leftarrow$  $
        insert(wildcardTree,  $\alpha'$ )
         $\alpha'' \leftarrow \alpha$ 
         $\alpha''$ [depth+1]  $\leftarrow$  heavyLetter
        insert(heavyChild,  $\alpha''$ )
  foreach child  $\in$  children  $\cup$  {wildcardTree} do
    processNode(child)

```

240 *Processing a leaf.* Each modified substring α stores its index of origin $idx(\alpha)$ and the set of modifications $M(\alpha)$. As we have seen, the substitutions introduced in the recursion are of two types: of wildcard origin and of heavy origin. For a modified substring α , we introduce a partition $M(\alpha) = W(\alpha) \cup H(\alpha)$ into modifications of these kinds. For every leaf v , the modified substrings $\alpha \in MS(v)$ share the same value $val(\alpha)$, and hence $W(\alpha)$ is also the same. Finally, by $W^{-1}(\alpha)$ we denote the set

245

$\{(j, T_{idx(\alpha)}^m[j]) : (j, \$) \in W(\alpha)\}$. We call modified substrings $\alpha, \beta \in MS(v)$ *compatible* if they satisfy the following condition:

$$H(\alpha) \cap H(\beta) = \emptyset, \quad W^{-1}(\alpha) \cap W^{-1}(\beta) = \emptyset, \quad |H(\alpha)| + |H(\beta)| + |W(\alpha)| = k. \quad (3.1)$$

Lemma 3.3 below shows that if two modified substrings are compatible, then the original substrings were at Hamming distance at most k . Intuitively, α and β are compatible only if the positions of modifications in $M(\alpha) \cup M(\beta)$ do not contain any position j such that $T_{idx(\alpha)}^m[j] = T_{idx(\beta)}^m[j]$.

Example 3.1 Let us consider modified strings $\alpha_1, \dots, \alpha_6$ with the following original strings and sets of modifications such that $val(\alpha_i) = \text{aba}\c for all $i = 1, \dots, 6$.

i	$U(\alpha_i)$	$H(\alpha_i)$	$W(\alpha_i)$	$W^{-1}(\alpha_i)$	$d_H(U(\alpha_1), U(\alpha_i))$
1	aaabc	$\{(2, \text{b})\}$	$\{(4, \$)\}$	$\{(4, \text{b})\}$	0
2	bbacc	$\{(1, \text{a})\}$	$\{(4, \$)\}$	$\{(4, \text{c})\}$	3
3	abbc b	$\{(3, \text{a}), (5, \text{c})\}$	$\{(4, \$)\}$	$\{(4, \text{c})\}$	4
4	abacc	\emptyset	$\{(4, \$)\}$	$\{(4, \text{c})\}$	2
5	acacc	$\{(2, \text{b})\}$	$\{(4, \$)\}$	$\{(4, \text{c})\}$	2
6	ababa	$\{(5, \text{c})\}$	$\{(4, \$)\}$	$\{(4, \text{b})\}$	2

Let us notice that $W(\alpha_i)$ is the same for all i . Let $k = 3$. The only modified string that is compatible with α_1 is α_2 . Each of the remaining modified strings violates exactly one of the conditions from Equation (3.1): $|H(\alpha_1)| + |H(\alpha_3)| + |W(\alpha_1)| = 4$, $|H(\alpha_1)| + |H(\alpha_4)| + |W(\alpha_1)| = 2$, $H(\alpha_1) \cap H(\alpha_5) = \{(2, \text{b})\}$, $W^{-1}(\alpha_1) \cap W^{-1}(\alpha_6) = \{(4, \text{b})\}$. Indeed, we have $d_H(U(\alpha_1), U(\alpha_2)) = 3$ and $d_H(U(\alpha_1), U(\alpha_i)) \neq 3$ for $i \in \{3, \dots, 6\}$.

As proved in Lemma 3.8 below, for every $\alpha \in MS(v)$, we should increment $A_{=k}^m[idx(\alpha)]$ for each compatible $\beta \in MS(v)$. We next show how to efficiently count these modified substrings using the inclusion-exclusion principle and several precomputed values, as we cannot afford to count them naively.

For convenience, let $R(\alpha)$ denote the union of disjoint sets $H(\alpha)$ and $W^{-1}(\alpha)$. For a leaf v , let $Count(s, B)$ denote the number of modified substrings $\beta \in MS(v)$ such that $|H(\beta)| = s$ and $B \subseteq R(\beta)$. All the non-zero values $Count(\cdot, \cdot)$ are stored in a hash table. They can be generated by iterating through all the subsets of $R(\beta)$ for all modified substrings $\beta \in MS(v)$; this costs $O(2^k k |MS(v)|)$ time and space. Finally, the result for a modified substring α can be computed using the following direct consequence of the inclusion-exclusion principle.

Lemma 3.2 *The number of modified substrings $\beta \in MS(v)$ that are compatible with a modified substring $\alpha \in MS(v)$ is $\sum_{B \subseteq R(\alpha)} (-1)^{|B|} Count(k - |M(\alpha)|, B)$.*

Proof First, let $h = k - |M(\alpha)|$. We want to count the modified substrings $\beta \in MS(v)$ that satisfy $|H(\beta)| = h$ and $R(\alpha) \cap R(\beta) = \emptyset$. For $(i, x) \in R(\alpha)$, let $A_{(i,x)} = \{\beta \in MS(v) : |H(\beta)| = h \text{ and } (i, x) \in R(\beta)\}$. Then, we want to compute $Count(h, \emptyset) -$

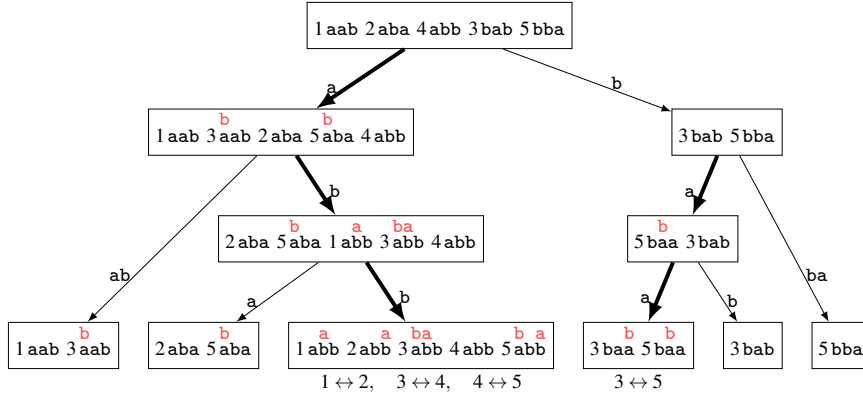


Fig. 3.2 Computation of $(2,3)$ -mappability for the string $T = aababba$ from Example 1.1. Edges leading to heavy children are drawn in bold. Note that the alphabet is binary in this case, so wildcard subtrees do not need to be introduced; the only substitutions are from the (at most one) light child to the heavy child. The letters shown above are the original letters before the substitutions. The pairs of compatible modified substrings are indicated with arrows; in the binary case, Equation (3.1) implies that these are substrings with modifications at different positions and exactly $k = 2$ modifications in total. In the end, $A_{=2}^3[1] = A_{=2}^3[2] = 1$ and $A_{=2}^3[3] = A_{=2}^3[4] = A_{=2}^3[5] = 2$ as expected.

$|\bigcup_{(i,x) \in R(\alpha)} A_{(i,x)}|$. By the inclusion-exclusion principle, we have

$$\begin{aligned} \left| \bigcup_{(i,x) \in R(\alpha)} A_{(i,x)} \right| &= \sum_{B \neq \emptyset, B \subseteq R(\alpha)} (-1)^{|B|+1} \left| \bigcap_{(i,x) \in B} A_{(i,x)} \right| \\ &= \sum_{B \neq \emptyset, B \subseteq R(\alpha)} (-1)^{|B|+1} \text{Count}(h, B), \end{aligned}$$

which concludes the proof. \square

Examples. Examples of the execution of the algorithm for a binary and a ternary string can be found in Figures 3.2 and 3.3, respectively.

²⁷⁵ *Correctness.* We will show that it is enough to count pairs of modified substrings obtained in the leaves. First we show that a pair of compatible modified substrings implies a pair of length- m substrings at Hamming distance at most k .

Lemma 3.3 *If $\alpha, \beta \in MS(v)$ are compatible with $i = \text{idx}(\alpha)$, and $j = \text{idx}(\beta)$, then $d_H(T_i^m, T_j^m) = k$.*

Proof By Equation (3.1) we have $W^{-1}(\alpha) \cap W^{-1}(\beta) = \emptyset$, so T_i^m and T_j^m differ at positions of modifications in $W(\alpha) = W(\beta)$. They also differ at positions of modifications in $H(\beta)$ since at the nodes corresponding to these positions, an ancestor of α (that is, the modified substring from which α originates) was in the heavy child and an ancestor of β originated from a light child (recall that Equation (3.1) includes

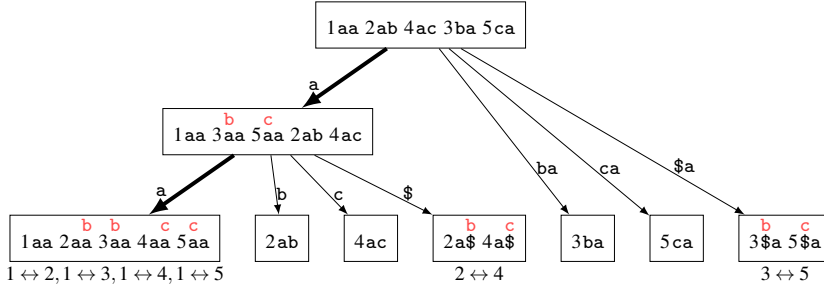


Fig. 3.3 Computation of $(1,2)$ -mappability for the string $T = aabaca$. This example illustrates the need to use of wildcard symbols for a non-binary alphabet, as otherwise pairs from different light children of the same node would not be registered. In this case $k = 1$, so modified substrings are compatible if and only if at most one of them has a modification or both have a modification of the wildcard-type which originate from different letters. We have $A_{-1}^2[1] = 4$ and $A_{-1}^2[2] = A_{-1}^2[3] = A_{-1}^2[4] = A_{-1}^2[5] = 2$.

$H(\alpha) \cap H(\beta) = \emptyset$. Symmetrically, T_i^m and T_j^m differ at positions of modifications in $H(\alpha)$. In conclusion, they differ at positions of modifications in $H(\alpha) \cup H(\beta) \cup W(\alpha)$. The three sets are disjoint, so $|H(\alpha) \cup H(\beta) \cup W(\alpha)| = |H(\alpha)| + |H(\beta)| + |W(\alpha)| = k$ by Equation (3.1). This shows that $d_H(T_i^m, T_j^m) \geq k$. With $val(\alpha) = val(\beta)$, we conclude that $d_H(T_i^m, T_j^m) = k$. \square

280 We proceed with a proof that if two length- m substrings are at distance at most k , then some leaf contains a pair of corresponding modified substrings that are compatible. Let us start with an observation that lists some basic properties of our algorithm. Both parts can be shown by straightforward induction.

- 285 **Observation 3.4** (a) *If a node v stores modified substrings $\alpha, \beta \in MS(v)$, then it has a descendant v' with $\mathbf{D}(v') = lcp(val(\alpha), val(\beta))$ and $\alpha, \beta \in MS(v')$.*
 (b) *Every node stores at most one modified substring originating from the same substring T_ℓ^m .*

We use the following auxiliary lemma.

290 **Lemma 3.5** *Assume that $d_H(T_i^m, T_j^m) = k$ and let $1 \leq x_1 < x_2 < \dots < x_k \leq m$ be the indices where the two substrings differ. Further let $x_{k+1} = m + 1$. For every $p \in \{1, \dots, k + 1\}$, there exist a node v_p and modified substrings $\alpha_p, \beta_p \in MS(v_p)$ such that:*

- 295 *– $idx(\alpha_p) = i$ and $idx(\beta_p) = j$;*
– $lcp(val(\alpha_p), val(\beta_p)) = x_p - 1 = \mathbf{D}(v_p)$;
– for each position x_1, \dots, x_{p-1} , both $M(\alpha_p)$ and $M(\beta_p)$ contain modifications of wildcard origin, or exactly one of these sets contains a modification of heavy origin;
– there are no other modifications in $M(\alpha_p)$ or $M(\beta_p)$.

300 *Proof* The proof goes by induction on p . As α_1 and β_1 , we take (un)modified substrings such that $idx(\alpha_1) = i$, $idx(\beta_1) = j$, and $M(\alpha_1) = M(\beta_1) = \emptyset$. They are stored

in the set $MS(r)$ for the root r , so Observation 3.4(a) guarantees the existence of a node v_1 with $\mathbf{D}(v_1) = \text{lcp}(\alpha_1, \beta_1)$ and $\alpha_1, \beta_1 \in MS(v_1)$.

Let $p > 1$. By the inductive hypothesis, the set $MS(v_{p-1})$ contains modified substrings α_{p-1} and β_{p-1} . The node v_{p-1} has children w_1, w_2 corresponding to letters $T_i^m[x_{p-1}]$ and $T_j^m[x_{p-1}]$, respectively. If w_1 is the heavy child, then w_2 is a light child and a modified substring β' such that $\text{idx}(\beta') = j$ and $M(\beta') = M(\beta_{p-1}) \cup \{(x_{p-1}, T_i^m[x_{p-1}])\}$ is inserted to $MS(w_1)$. Then, we take $\alpha' = \alpha_{p-1}$. The case that w_2 is the heavy child is symmetric. Finally, if both w_1 and w_2 are light children, a child u of v_{p-1} is created along the wildcard symbol $\$$. There exist modified substrings $\alpha', \beta' \in MS(u)$ such that: $\text{idx}(\alpha') = i$, $\text{idx}(\beta') = j$, $M(\alpha') = M(\alpha_{p-1}) \cup \{(x_{p-1}, \$)\}$, and $M(\beta') = M(\beta_{p-1}) \cup \{(x_{p-1}, \$)\}$.

In either case, we have $\text{lcp}(\text{val}(\alpha'), \text{val}(\beta')) = x_p - 1$. The set $(M(\alpha') \cup M(\beta')) \setminus (M(\alpha_{p-1}) \cup M(\beta_{p-1}))$ contains either a modification of heavy origin in one of the modified substrings or modifications of wildcard origin in both. Hence, by the inductive hypothesis, we can set $\alpha_p = \alpha'$ and $\beta_p = \beta'$. The node v_p with $\mathbf{D}(v_p) = \text{lcp}(\text{val}(\alpha_p), \text{val}(\beta_p))$ and $\alpha_p, \beta_p \in MS(v_p)$ must exist due to Observation 3.4(a). \square

Example 3.6 Let us consider strings $\alpha = \text{aab}$ and $\beta = \text{aba}$ from Figure 3.2 that differ at positions $x_1 = 2$ and $x_2 = 3$. The trie in the figure has a path that contains nodes storing the modified substrings from the following table.

p	α	$M(\alpha_p)$	$\text{val}(\alpha_p)$	β	$M(\beta_p)$	$\text{val}(\beta_p)$	$\mathbf{D}(v_p)$
1	aab	\emptyset	aab	aba	\emptyset	aba	1
2	aab	$\{(2, \text{b})\}$	abb	aba	\emptyset	aba	2
3	aab	$\{(2, \text{b})\}$	abb	abb	$\{(3, \text{b})\}$	abb	3

The following corollary is a direct consequence of Lemma 3.5.

Corollary 3.7 *If $d_H(T_i^m, T_j^m) = k$, then there is a leaf v and a pair of compatible modified substrings $\alpha, \beta \in MS(v)$ with $i = \text{idx}(\alpha)$ and $j = \text{idx}(\beta)$.*

Proof Lemma 3.5, applied for $p = k + 1$, yields a leaf v_{k+1} that contains compatible modified substrings $\alpha = \alpha_{k+1}$ and $\beta = \beta_{k+1}$ with $\text{idx}(\alpha) = i$ and $\text{idx}(\beta) = j$. \square

The following lemma, a stronger version of the corollary, together with Lemma 3.3 shows that Algorithm 1 correctly computes the mappability table $A_{=k}^m$.

Lemma 3.8 *If $d_H(T_i^m, T_j^m) = k$, then there is exactly one leaf v and exactly one pair of compatible modified substrings $\alpha, \beta \in MS(v)$ with $i = \text{idx}(\alpha)$ and $j = \text{idx}(\beta)$.*

Proof Corollary 3.7 implies that there is at least one leaf that contains compatible modified substrings α and β with $\text{idx}(\alpha) = i$ and $\text{idx}(\beta) = j$. Now, it suffices to check that there is no other pair of compatible modified substrings $(\alpha', \beta') \neq (\alpha, \beta)$ that would be present in some leaf u and satisfy $\text{idx}(\alpha') = i$ and $\text{idx}(\beta') = j$.

We apply Lemma 3.5. Let us first note that $M(\alpha') \cup M(\beta')$ must contain modifications at positions x_1, \dots, x_k (since $\text{val}(\alpha') = \text{val}(\beta')$) and no modifications at other positions (otherwise, $|H(\alpha')| + |H(\beta')| + |W(\alpha')|$ would exceed k). Let p be the greatest index in $\{1, \dots, k+1\}$ such that $x_p - 1 \leq \text{lcp}(\text{val}(\alpha), \text{val}(\alpha'))$. By Observation 3.4(b), $u \neq v_{k+1}$, so $p \leq k$.

Thus, the node v_p is an ancestor of the leaf u , but the node v_{p+1} is not. Let us consider the children w_1, w_2 of v_p obtained by following edges with labels $T_i^m[x_p]$ and $T_j^m[x_p]$, respectively. If w_1 is the heavy child, β' must contain a modification of heavy origin at position x_p , so v_{p+1} is an ancestor of u ; a contradiction. The same contradiction is obtained in the symmetric case that w_2 is the heavy child. Finally, if both w_1 and w_2 are light, then either both α' and β' contain a modification of wildcard origin at position x_p , which again gives a contradiction, or they both contain a modification of heavy origin, which contradicts the first part of Equation (3.1). \square

Remark 3.9 The recursive approach presented above is somewhat similar to the scheme used by Thankachan et al. [34] for computing the longest common substring with up to k mismatches of two strings. We attempted to adapt the approach of [34] to computing k -mappability, but failed. Another virtue of our approach is that we obtain time complexity better by a factor of $k!$ for super-constant k .

Implementation and complexity. Our Algorithm 1, excluding the counting phase in the leaves, has exactly the same structure as Algorithm 1 in [9]. This is verified in detail in Appendix A. Proposition 13 from [9] provides a bound on the total number of the generated modified strings and an efficient implementation based on finger-search trees. We apply that proposition for a family \mathbf{F} composed of substrings T_i^m to obtain the following bounds.

Fact 3.10 (see [9, Proposition 13]) *Algorithm 1 applied up to the leaves takes $O(n \binom{\log n + k + 1}{k+1} 2^k)$ time and generates $O(n \binom{\log n + k}{k} 2^k)$ modified substrings.*

Let us further analyze the space complexity of the algorithm.

Observation 3.11 *If a node v is a child of w , then every element of $MS(v)$ is either an element of $MS(w)$ or a modified substring originating from an element of $MS(w)$.*

Lemma 3.12 *Algorithm 1 applied up to the leaves uses $O(nk)$ working space.*

Proof We assume that, upon termination, the procedure `processNode` discards the set $MS(v)$ and all the modified strings created during its execution. This way, the whole memory allocated within a given call to `processNode` is freed. Since `processNode` returns no output and its only side effects are updates of the array A_{-}^k , no information is lost through such garbage collection.

A call to `processNode(v)` for node v partitions the list $MS(v)$ into sublists corresponding to u_1, \dots, u_a , creates $2(|MS(u_2)| + \dots + |MS(u_a)|)$ new modified substrings (each requiring constant space to be stored), appends them to sublists corresponding to u_1 and u_{a+1} , and then recurses on the sublists. In particular, the elements of the original list $MS(v)$ are not copied but reused in the recursive call.

Let us consider a root-to-leaf path ρ in the recursion. Each recursive call uses $O(1)$ local variables, which take $O(n)$ space overall. We also need to bound the total number of modified substrings created by calls to `processNode` for nodes on the path ρ .

By Observations 3.11 and 3.4(b), $|MS(v)|$ is non-increasing on ρ . Moreover, if v is a light child of its parent w , then $|MS(v)| \leq |MS(w)|/2$. Let us consider all nodes w on ρ such that the unique child of w that is on ρ is a light child. The total number of

365 modified strings created by the calls to `processNode(w)` for all such nodes w is $O(n)$ since we can bound it from above by a geometric series that sums to $O(n)$.

As for the calls to `processNode(w)` for the remaining nodes on ρ , for every two modified strings they create, they put one of them in the child of w that also belongs to ρ . Hence, it suffices to bound the total number of modified substrings originating from T_i^m for each position i that are in $MS(v)$ for some node v on ρ . For a given position i , let $\alpha_1, \dots, \alpha_b$ be all such modified substrings originating from T_i^m . By Observation 3.11, we have $M(\alpha_1) \subsetneq M(\alpha_2) \subsetneq \dots \subsetneq M(\alpha_b)$ and thus $b \leq k$. In total, we create $O(nk)$ modified substrings in calls to `processNode` on nodes of ρ . \square

Next, we show how to improve the time complexity of Algorithm 1 by a relatively small change in its execution. Intuitively, we will take advantage of the fact that the modified substrings in a leaf of the recursion do not need to be sorted lexicographically.

370 Namely, whenever a modified substring β with exactly k modifications is created at a node v (i.e., $|M(\alpha)| = k - 1$ in the if-statement), we do not include β in the recursive call of `wildcardTree` or `heavyChild`. Instead, an entry $(val(\beta), \beta)$ is inserted into a global hash table. When processing a leaf v containing modified substrings with a common value $val(\alpha)$, we need to move all modified substrings with value 375 $val(\alpha)$ from the global hash table to the set $MS(v)$. Finally, if any modified string β created while processing a given node v remains in the hash table upon completion of `processNode(v)`, then β is removed from the hash table together with all other modified substrings with the value $val(\beta)$. At this moment, an artificial leaf of the recursion containing all these modified substrings is created, and the standard routine 380 is applied to process this leaf.

Recall that the hash table uses Karp–Rabin fingerprints to index strings and collisions could incur incorrect results in the algorithm. To tackle this issue, whenever a modified substring β is inserted to the hash table and there is another modified substring with the same hash in the table, we pick any such modified substring α and check if $val(\alpha) = val(\beta)$ in $O(k)$ time using lcp queries on T with a method 385 that resembles kangaroo jumping [18, 28] (it requires $O(n)$ -time preprocessing). By Lemma 3.12, the hash table contains up to $O(nk)$ entries at any given time, so the collision probability is $O(nk \cdot n^{-C}) = O(n^{-C+2})$. Setting $C > c + 2$, we can make sure that this is dominated by the probability that the hash table fails to process the 390 underlying insertion in $O(1)$ amortized time.

Let us call the resulting algorithm Algorithm 1’.

Lemma 3.13 *The outputs of Algorithms 1 and 1’ are the same. Moreover, Algorithm 1’ works in $O(n \binom{\log n + k}{k} 2^k k)$ time with high probability (up to the leaves) and uses the same amount of space as Algorithm 1.*

395 *Proof* Let v be a leaf in the recursion of Algorithm 1. If $MS(v)$ contains at least one modified substring with up to $k - 1$ modifications, v will be identified by the recursive procedure of Algorithm 1’. Then, all modified substrings with exactly k modifications that belong to v are populated from the global hash table. If $MS(v)$ does not contain any modified substring with less than k modifications, v will be identified upon a 400 deletion from the global hash map at the lowest internal node u of the recursion in which a modified substring belonging to $MS(v)$ was created. Here, we use the fact

that the path-labels $\mathbf{L}(u)$ of all nodes u of the recursion are different. This shows that indeed the leaves of the recursion of Algorithms 1 and 1' are the same.

As for the time complexity, the total number of modified substrings created by Algorithm 1' is the same as in Algorithm 1, i.e., $O(n \binom{\log n + k}{k} 2^k)$ by Fact 3.10. However, the time necessary to conduct the whole recursive procedure corresponds to the time complexity of Algorithm 1 if it had been executed with $k - 1$ instead of k , i.e., also $O(n \binom{\log n + k}{k} 2^k)$ by Fact 3.10. After $O(n)$ -time preprocessing, for each modified substring, we can compute its Karp–Rabin fingerprint and check collisions in $O(k)$ time; this accounts for the additional factor k in the time complexity.

Finally, the space complexity stays the same because modified substrings with exactly k modifications are removed from the hash table at latest when the recursion rolls back. \square

Lemmas 3.12 and 3.13 yield the complexity of Algorithm 1'. Note that, due to the application of the inclusion-exclusion principle in the leaves, we need to multiply the time complexity of the algorithm by 2^k and increase the space complexity by $O(n2^k k)$.

Theorem 3.14 *There exists a Las-Vegas randomized algorithm that computes the (k, m) -mappability of a given length- n string in $O(n2^k k)$ space and, with high probability, in $O(n \binom{\log n + k}{k} 4^k k)$ time. For $k = O(1)$, the space is $O(n)$ and the time becomes $O(n \log^k n)$.*

4 All-Pairs Hamming Distance Problem

Let us recall that in the all-pairs Hamming distance problem, given a set \mathbf{R} of r length- m strings and an integer $k \in \{0, \dots, m\}$, we are to return all pairs $(X_1, X_2) \in \mathbf{R} \times \mathbf{R}$, with $X_1 \neq X_2$, such that X_1 and X_2 are at Hamming distance at most k . We will show how the algorithm from the previous section can be modified to solve this problem at the cost of an additional $\log r$ -factor in the complexity.

We run the algorithm from the previous section for T being a concatenation of all the strings in \mathbf{R} and only with substrings $\{T_i^m : i \bmod m = 1\}$ in the root. The algorithm needs to be updated only at the leaves of the compact trie. Henceforth, let us consider a trie leaf v with a set $MS(v) = \{\beta_1, \dots, \beta_p\}$ of modified substrings. We will further denote this set as MS ($|MS| = p$). Our goal is to list, for every $\beta \in MS$, all $\beta' \in MS$ that are compatible with β .

Let us construct a static balanced binary search tree (BST) in which the leaves correspond to the modified substrings β_i . This way, each node of the BST corresponds to a set of subsequent candidates from the leaves of its subtree. If β_i, \dots, β_j are the modified substrings in the leaves of the subtree of a BST node u , then we denote $set(u) = \{\beta_i, \dots, \beta_j\}$. A leaf will be responsible for storing information only for itself and an internal node stores merged information of its children.

Our goal is to store information in each node u of the BST in such a way that, for any modified substring $\alpha \in MS$, we will be able to decide whether there is any other candidate in $set(u)$ that is compatible with α . Therefore, in each node u , we will compute all the required machinery for using the inclusion-exclusion principle on the modified substrings in $set(u)$, that is, a dictionary that stores all non-zero

values of $\text{Count}(s, B)$ for modified substrings $\beta \in \text{set}(u)$. Since every $\beta \in MS$ is present in $O(\log p)$ sets $\text{set}(u)$, precomputing all mentioned information can be done in $O(2^k k p \log p)$ time and space.

Our query algorithm for a given modified substring β is a recursive procedure starting at the root of the BST. Assume that the algorithm is at some BST node u . We use Lemma 3.2 and the dictionary for $\text{set}(u)$ to count the elements $\beta' \in \text{set}(u)$ that are compatible with β . If this number is positive, the algorithm recursively descends to the children of node u . In the end, modified substrings β' that are compatible with β will be listed at the leaves of the BST. The correctness of this algorithm follows from Lemma 3.8.

Every application of Lemma 3.2 takes $O(2^k k)$ time. For each modified substring β' that is compatible with a modified substring β , the algorithm will visit $O(\log p)$ BST nodes, which gives $O(2^k k \log p)$ time for finding each compatible modified substring $\beta' \in MS$. Note that $p \leq r$ (see Observation 3.4(b)). Summing up over all trie nodes v and applying Lemmas 3.13 and 3.12, we obtain the following result. (Observe that [9, Proposition 13] is applied for a family \mathbf{F} of size r rather than n .)

Theorem 4.1 *There exists a Las-Vegas randomized algorithm that, given a set of r length- m strings and an integer k , solves the all-pairs Hamming distance problem in $O(rm + 2^k k r \log r)$ space and, with high probability, in $O(rm + r \binom{\log r + k}{k} 4^k k \log r + \text{output} \cdot 2^k k \log r)$ time. For $k = O(1)$, the space is $O(rm + r \log r)$ and the time becomes $O(rm + r \log^{k+1} r + \text{output} \cdot \log r)$.*

Notably, the algorithm underlying Theorem 4.1 works in $O(rm)$ time (with high probability) if $k = O(1)$, $m = \Omega(\log^{k+1} r)$, and $\text{output} = O(rm/\log r)$.

5 Computing Mappability in $O(nm^k)$ Time and $O(n)$ Space

In this section, we generalize the $O(nm)$ -time algorithm for $k = 1$ and integer alphabets from [3]. To this end, we make use of an approach from [6]. The high-level idea from [6] is to define a lexicographic order on the suffixes of T that ignores the same k fixed positions of every suffix. (In fact, the algorithm does the same for many such combinations of k positions.) The algorithm then uses the suffix tree of T to sort the modified suffixes according to this new lexicographic order. The focus of the original algorithm is not on counting substrings that are at Hamming distance at most k , and so we adapt it with some extra care to avoid multiple counting.

We first generate all $\binom{m}{\leq k}$ subsets of $\{1, \dots, m\}$ of size at most k . For each such subset F , we consider the length- m substrings of T with their f -th letter substituted with $\$ \notin \Sigma$ for all $f \in F$. We sort all these sets of strings in $O(nk \binom{m}{\leq k})$ total time using the approach of [6], also obtaining the maximal blocks of equal strings in the sorted lists.

We now briefly describe the algorithm for sorting one such set of strings in time $O(nk)$ for the sake of completeness. Let us assume for simplicity that $F = \{f\}$ as the algorithm can be generalized trivially for larger sets. We first retrieve the sorted list of T_i^{f-1} for all i from the suffix tree. We then give ranks to these strings after we check equality of adjacent strings in the sorted list using lcp queries. We similarly

rank strings T_j^{m-f} for all j . Finally, we sort the ranks of the pairs $(T_i^{f-1}, T_{i+f+1}^{m-f})$ using bucket sort.

485 Prior to running the above algorithm, we initialize arrays D_K for $K \in \{1, \dots, k\}$. For each maximal block, of size b , of equal strings obtained for some set F , we increment the b relevant entries of $D_{|F|}$ by $b-1$.

Note that if $d_H(T_i^m, T_j^m) = \kappa$, then this will contribute $\binom{m-\kappa}{K-\kappa}$ to each of $D_K[i]$ and $D_K[j]$ for $K \geq \kappa$, since there are this many size- K supersets of the set of mismatching positions in the power set of $\{1, \dots, m\}$. We thus compute $A_{=K}^m[i] = D_K[i] - \sum_{\kappa=0}^{K-1} \binom{m-\kappa}{K-\kappa} A_{=\kappa}[i]$ in increasing order with respect to K , and we are done. (We precompute all relevant binomial coefficients in $O(k^2)$ time.)

Theorem 5.1 *Given a string of length n , the (k, m) -mappability problem can be solved in $O(nk \binom{m}{\leq k})$ time and $O(n)$ space. For $k = O(1)$, the time becomes $O(nm^k)$.*

495 Combining Theorems 3.14 and 5.1 gives the following result.

Corollary 5.2 *For every $k = O(1)$, there exists a randomized algorithm that computes the (k, m) -mappability of a given length- n string in $O(n)$ space and in $O(n \cdot \min\{m^k, \log^k n\})$ time with high probability.*

6 Computing (k, m) -Mappability for All k or for All m

500 **Theorem 6.1** *The (k, m) -mappability for a given m and all $k \in \{0, \dots, m\}$ can be computed in $O(n^2)$ time using $O(n)$ space.*

Proof We first present an algorithm which solves the problem in $O(n^2)$ time using $O(n^2)$ space and then show how to reduce the space usage to $O(n)$.

505 We initialize an $n \times n$ matrix M in which $M[i, j]$ will store the Hamming distance between substrings T_i^m and T_j^m . Let us consider two letters $T[i] \neq T[j]$ of the input string, where $i < j$. Such a pair contributes to a mismatch between the following pairs of strings:

$$(T_{i-m+1}^m, T_{j-m+1}^m), (T_{i-m+2}^m, T_{j-m+2}^m), \dots, (T_i^m, T_j^m).$$

This list of strings is represented by a diagonal interval in M , the entries of which we need to increment by 1. We process all $O(n^2)$ pairs of letters and update the information on the respective intervals. Then $A_{=k}^m[i] = |\{j : M[i, j] = k\}|$.

510 To achieve $O(1)$ time for each single addition on a diagonal interval, we use a well-known trick from an analogous problem in one dimension. Suppose that we would like to add 1 on the diagonal interval from $M[x_1, y_1]$ to $M[x_2, y_2]$. Instead, we can simply add 1 to $M[x_1, y_1]$ and -1 to $M[x_2 + 1, y_2 + 1]$. Every cell will then represent the difference of its actual value to the actual value of its predecessor on the diagonal. After all such operations are performed, we can retrieve the actual values by computing prefix sums on each diagonal in a top-down manner.

To reduce the space usage to $O(n)$, it suffices to observe that the value of $M[i, j]$ depends only on the value of $M[i-1, j-1]$ and at most two letter comparisons which can add $+1$ and/or -1 to the cell. Recall that $M[i, j] = d_H(T_i^m, T_j^m)$. We need to subtract 1 from the previous result if the first characters of the previous substrings were

equal and add 1 if the last characters of the new substrings were different. Therefore, we can process the matrix row by row, from top to bottom, and compute the values $A_{=0}^m[i], \dots, A_{=m}^m[i]$ while processing the i th row. \square

By $\text{lcp}_k(i, j)$ we denote the length of the longest common prefix of T_i and T_j when up to k mismatches are allowed, that is, the maximum ℓ such that $d_H(T_i^\ell, T_j^\ell) \leq k$.
 520 Flouri et al. [15] proposed an $O(n^2)$ -time algorithm to compute the longest common substring of two strings S, T with at most k mismatches. Their algorithm actually computes the lengths of the longest common prefixes with at most k mismatches of every suffix of S and T and returns the maximum among them. Applied for $S = T$, it gives the following result.

525 **Lemma 6.2 ([15])** *For a string T of length n , the values $\text{lcp}_k(i, j)$ for all $i, j \in \{1, \dots, n\}$ can be computed in $O(n^2)$ time.*

Theorem 6.3 *The (k, m) -mappability for a given k and all $m \in \{k, \dots, n\}$ can be computed in $O(n^2)$ time and space.*

Proof First we compute all the values $\text{lcp}_k(i, j)$ using Lemma 6.2. We initialize an $n \times n$ matrix Q setting all entries to 0. Then, for a pair (i, j) such that $\text{lcp}_k(i, j) = \ell$, we increment the entries $Q[\ell, i]$ and $Q[\ell, j]$. Note that if $\text{lcp}_k(i, j) = \ell$, then i (resp. j) will contribute 1 to the (k, m) -mappability values $A_{\leq k}^m[j]$ (resp. $A_{\leq k}^m[i]$) for all $m \in \{k, \dots, \ell\}$. Thus, starting from the last row of Q , we iteratively add row ℓ to row $\ell - 1$. By the above observation, row m ends up storing the (k, m) -mappability array $A_{\leq k}^m$. \square

7 Conditional Hardness for $k, m = \Theta(\log n)$

530 We will show that (k, m) -mappability cannot be computed in strongly subquadratic time in case that the parameters are $\Theta(\log n)$, unless the Strong Exponential Time Hypothesis (SETH) of Impagliazzo, Paturi and Zane [23, 22] fails. Our proof is based on the conditional hardness of the following decision version of the Longest Common Substring with k Mismatches problem.

Common Substring of Length d with k Mismatches

535 **Input:** Strings S, T of length n over binary alphabet and integers k, d .

Output: Is there a factor of S of length d that occurs in T with k mismatches?

Lemma 7.1 ([27]) *Suppose there is $\varepsilon > 0$ such that Common Substring of Length d with k Mismatches can be solved in $O(n^{2-\varepsilon})$ time on strings over binary alphabet for $k = \Theta(\log n)$ and $d = 21k$. Then SETH is false.*

540 **Theorem 7.2** *If the (k, m) -mappability can be computed in $O(n^{2-\varepsilon})$ time for binary strings, $k, m = \Theta(\log n)$, and some $\varepsilon > 0$, then SETH is false.*

Proof We make a Turing reduction from Common Substring of Length d with k Mismatches. Let S and T be the input to the problem. We compute the (k, d) -mappabilities of strings $S \cdot T$ and $S \cdot T_1^{d-1}$ and store them in arrays A and B , respectively. Henceforth,

we consider only indices $i \in \{1, \dots, n - d + 1\}$ in the arrays. For each such index, $A[i]$ holds the number of length- d factors of S , $X := S_{n-d+2}^{d-1} T_1^{d-1}$, and T that are at Hamming distance k from S_i^d , and $B[i]$ holds the number of length- d factors of S and X that are at Hamming distance k from S_i^d . For each i , we subtract $B[i]$ from $A[i]$. Then, $A[i]$ holds the number of length- d factors of T that are at Hamming distance k from S_i^d . Hence, Common Substring of Length d with k Mismatches has a positive answer if and only if $A[i] > 0$ for any $i \in \{1, \dots, n - d + 1\}$.

By Lemma 7.1, an $O(n^{2-\varepsilon})$ -time algorithm for Common Substring of Length d with k Mismatches with $k = \Theta(\log n)$ and $d = 21k$ would refute SETH. By the shown reduction, an $O(n^{2-\varepsilon})$ -time algorithm for (k, m) -mappability with $k, m = \Theta(\log n)$ would also refute SETH. \square

8 Final Remarks

Our main contribution is an $O(n \cdot \min\{m^k, \log^k n\})$ -time $O(n)$ -space algorithm for solving the (k, m) -mappability problem for a length- n string over an integer alphabet. Let us recall that genome mappability, as introduced in [12], counts the number of substrings that are at Hamming distance at most k from every length- m substring of the text. One may also be interested to consider mappability under the edit distance model. This question relates also to recent contributions to computing approximate longest common prefixes and substrings under edit distance [33, 6]. In the case of the edit distance, in particular, a decision needs to be made whether sufficiently similar substrings only of length exactly m or of all lengths between $m - k$ and $m + k$ should be counted. We leave the mappability problem under edit distance for future investigation.

Acknowledgements Panagiotis Charalampopoulos is supported by the Israel Science Foundation grant 592/17. Tomasz Kociumaka is partly supported by NSF 1652303, 1909046, and HDR TRIPODS 1934846 grants, and an Alfred P. Sloan Fellowship. Jakub Radoszewski and Juliusz Straszynski are supported by the “Algorithms for text processing with errors and uncertainties” project carried out within the HOMING program of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund, project no. POIR.04.04.00-00-24BA/16, and by the Polish National Science Center, grant number 2018/31/D/ST6/03991.

This paper is part of the PANGAIA project that has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no. 872539. This paper is also part of the ALPACA project that has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no. 956229.



References

1. Alamro, H., Ayad, L.A.K., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P.: Longest common prefixes with k -mismatches and applications. In: A.M. Tjoa, L. Bellatreche, S. Biffl, J. van Leeuwen,

- J. Wiedermann (eds.) 44th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2018, *LNCS*, vol. 10706, pp. 636–649. Springer (2018). URL https://doi.org/10.1007/978-3-319-73117-9_45
2. Alzamel, M., Charalampopoulos, P., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Radoszewski, J., Straszynski, J.: Efficient computation of sequence mappability. In: T. Gagie, A. Moffat, G. Navarro, E. Cuadros-Vargas (eds.) 25th International Symposium on String Processing and Information Retrieval, SPIRE 2018, *LNCS*, vol. 11147, pp. 12–26. Springer (2018). URL https://doi.org/10.1007/978-3-030-00479-8_2
 3. Alzamel, M., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P., Radoszewski, J., Sung, W.: Faster algorithms for 1-mappability of a sequence. *Theoretical Computer Science* **812**, 2–12 (2020). URL <https://doi.org/10.1016/j.tcs.2019.04.026>
 4. Amir, A., Boneh, I., Kondratovsky, E.: The k -mappability problem revisited. In: P. Gawrychowski, T. Starikovskaya (eds.) 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, *LIPICs*, vol. 191, pp. 5:1–5:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2021). URL <https://doi.org/10.4230/LIPICs.CPM.2021.5>
 5. Antoniou, P., Daykin, J.W., Iliopoulos, C.S., Kourie, D., Mouchard, L., Pissis, S.P.: Mapping uniquely occurring short sequences derived from high throughput technologies to a reference genome. In: 9th International Conference on Information Technology and Applications in Biomedicine, ITAB 2009, pp. 1–4. IEEE (2009). URL <https://doi.org/10.1109/itab.2009.5394394>
 6. Ayad, L.A.K., Barton, C., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P.: Longest common prefixes with k -errors and applications. In: T. Gagie, A. Moffat, G. Navarro, E. Cuadros-Vargas (eds.) 25th International Symposium on String Processing and Information Retrieval, SPIRE 2018, *LNCS*, vol. 11147, pp. 27–41. Springer (2018). URL https://doi.org/10.1007/978-3-030-00479-8_3
 7. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms* **57**(2), 75–94 (2005). URL <https://doi.org/10.1016/j.jalgor.2005.08.001>
 8. Carriço, J.A., Crochemore, M., Francisco, A.P., Pissis, S.P., Ribeiro-Gonçalves, B., Vaz, C.: Fast phylogenetic inference from typing data. *Algorithms for Molecular Biology* **13**(1), 4 (2018). URL <https://doi.org/10.1186/s13015-017-0119-7>
 9. Charalampopoulos, P., Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T.: Linear-time algorithm for long LCF with k mismatches. In: G. Navarro, D. Sankoff, B. Zhu (eds.) 29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018, *LIPICs*, vol. 105, pp. 23:1–23:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2018). URL <https://doi.org/10.4230/LIPICs.CPM.2018.23>
 10. Cole, R., Gottlieb, L., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: L. Babai (ed.) 36th Annual ACM Symposium on Theory of Computing, STOC 2004, pp. 91–100. ACM (2004). URL <https://doi.org/10.1145/1007352.1007374>
 11. Crochemore, M., Francisco, A.P., Pissis, S.P., Vaz, C.: Towards distance-based phylogenetic inference in average-case linear-time. In: R. Schwartz, K. Reinert (eds.) 17th International Workshop on Algorithms in Bioinformatics, WABI 2017, *LIPICs*, vol. 88, pp. 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2017). URL <https://doi.org/10.4230/LIPICs.WABI.2017.9>
 12. Derrien, T., Estellé, J., Sola, S.M., Knowles, D.G., Raineri, E., Guigó, R., Ribeca, P.: Fast computation and applications of genome mappability. *PLoS ONE* **7**(1), e30377 (2012). URL <https://doi.org/10.1371/journal.pone.0030377>
 13. Dietzfelbinger, M., Meyer auf der Heide, F.: A new universal class of hash functions and dynamic hashing in real time. In: M. Paterson (ed.) 17th International Colloquium on Automata, Languages and Programming, ICALP 1990, *LNCS*, vol. 443, pp. 6–19. Springer (1990). URL <https://doi.org/10.1007/BFb0032018>
 14. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of suffix tree construction. *Journal of the ACM* **47**(6), 987–1011 (2000). URL <https://doi.org/10.1145/355541.355547>
 15. Flouri, T., Giaquinta, E., Kobert, K., Ukkonen, E.: Longest common substrings with k mismatches. *Information Processing Letters* **115**(6-8), 643–647 (2015). URL <https://doi.org/10.1016/j.ipl.2015.03.006>
 16. Fonseca, N.A., Rung, J., Brazma, A., Marioni, J.C.: Tools for mapping high-throughput sequencing data. *Bioinformatics* **28**(24), 3169–3177 (2012). URL <https://doi.org/10.1093/bioinformatics/bts605>

17. Francisco, A.P., Bugalho, M., Ramirez, M., Carriço, J.A.: Global optimal eBURST analysis of multi-locus typing data using a graphic matroid approach. *BMC Bioinformatics* **10**(1), 152 (2009). URL <https://doi.org/10.1186/1471-2105-10-152>
18. Galil, Z., Giancarlo, R.: Parallel string matching with k mismatches. *Theoretical Computer Science* **51**, 341–348 (1987). URL [https://doi.org/10.1016/0304-3975\(87\)90042-9](https://doi.org/10.1016/0304-3975(87)90042-9)
19. Gog, S., Venturini, R.: Fast and compact Hamming distance index. In: R. Perego, F. Sebastiani, J.A. Aslam, I. Ruthven, J. Zobel (eds.) 39th International ACM-SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2016, pp. 285–294. ACM (2016). URL <https://doi.org/10.1145/2911451.2911523>
20. Grabowski, S., Kowalski, T.M.: Algorithms for all-pairs Hamming distance based similarity. *Software: Practice and Experience* (2021). URL <https://doi.org/10.1002/spe.2978>
21. Hooshmand, S., Abedin, P., Gibney, D., Aluru, S., Thankachan, S.V.: Faster computation of genome mappability with one mismatch. In: 8th IEEE International Conference on Computational Advances in Bio and Medical Sciences, ICCABS 2018, p. 1. IEEE Computer Society (2018). URL <https://doi.org/10.1109/ICCABS.2018.8541897>
22. Impagliazzo, R., Paturi, R.: On the complexity of k -SAT. *Journal of Computer and System Sciences* **62**(2), 367–375 (2001). URL <https://doi.org/10.1006/jcss.2000.1727>
23. Impagliazzo, R., Paturi, R., Zane, F.: Which problems have strongly exponential complexity? *Journal of Computer and System Sciences* **63**(4), 512–530 (2001). URL <https://doi.org/10.1006/jcss.2001.1774>
24. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. *Journal of the ACM* **53**(6), 918–936 (2006). URL <https://doi.org/10.1145/1217856.1217858>
25. Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* **31**(2), 249–260 (1987). URL <https://doi.org/10.1147/rd.312.0249>
26. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: A. Amir, G.M. Landau (eds.) 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001, LNCS, vol. 2089, pp. 181–192. Springer (2001). URL https://doi.org/10.1007/3-540-48194-X_17
27. Kociumaka, T., Radoszewski, J., Starikovskaya, T.A.: Longest common substring with approximately k mismatches. *Algorithmica* **81**(6), 2633–2652 (2019). URL <https://doi.org/10.1007/s00453-019-00548-x>
28. Landau, G.M., Vishkin, U.: Efficient string matching with k mismatches. *Theoretical Computer Science* **43**, 239–249 (1986). URL [https://doi.org/10.1016/0304-3975\(86\)90178-7](https://doi.org/10.1016/0304-3975(86)90178-7)
29. Mäkinen, V., Norri, T.: Applying the positional Burrows–Wheeler transform to all-pairs Hamming distance. *Information Processing Letters* **146**, 17–19 (2019). URL <https://doi.org/10.1016/j.ipl.2019.02.003>
30. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* **22**(5), 935–948 (1993). URL <https://doi.org/10.1137/0222058>
31. Manzini, G.: Longest common prefix with mismatches. In: C.S. Iliopoulos, S.J. Puglisi, E. Yilmaz (eds.) 22nd International Symposium on String Processing and Information Retrieval, SPIRE 2015, LNCS, vol. 9309, pp. 299–310. Springer (2015). URL https://doi.org/10.1007/978-3-319-23826-5_29
32. Pockrandt, C., Alzamel, M., Iliopoulos, C.S., Reinert, K.: Genmap: ultra-fast computation of genome mappability. *Bioinformatics* **36**(12), 3687–3692 (2020). URL <https://doi.org/10.1093/bioinformatics/btaa222>
33. Thankachan, S.V., Aluru, C., Chockalingam, S.P., Aluru, S.: Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In: B.J. Raphael (ed.) 22nd Annual International Conference on Research in Computational Molecular Biology, RECOMB 2018, LNCS, vol. 10812, pp. 211–224. Springer (2018). URL https://doi.org/10.1007/978-3-319-89929-9_14
34. Thankachan, S.V., Apostolico, A., Aluru, S.: A provably efficient algorithm for the k -mismatch average common substring problem. *Journal of Computational Biology* **23**(6), 472–482 (2016). URL <https://doi.org/10.1089/cmb.2015.0235>
35. Vaz, C., Nascimento, M., Carriço, J.A., Rocher, T., Francisco, A.P.: Distance-based phylogenetic inference from typing data: a unifying view. *Briefings in Bioinformatics* **22**(3) (2021). URL <https://doi.org/10.1093/bib/bbaa147>

A Application of the Construction from [9]

In [9], a recursive procedure shown in Algorithm 2 was developed. This procedure takes as input a string P and a family \mathbf{F}_P that consists of tuples (S, F, b) such that $F \in \mathbf{F}$ for some string family \mathbf{F} , S is a suffix of F of length $|S| = |F| - |P|$, and $b = k - d_H(F, PS) \geq 0$. Intuitively, the parameter b can be seen as a “budget” of remaining letter substitutions that can be performed in the string (PS obtained from F) that prevents exceeding the threshold k of mismatches. In the first call, we have $P = \varepsilon$ and $\mathbf{F}_P = \{(F, F, k) : F \in \mathbf{F}\}$. For a non-empty string $S = aS'$, where $a \in \Sigma$, we denote $\text{suf}(S) = S'$.

Algorithm 2: A recursive procedure inserting strings with prefix P to sets $N(F)$.

```

Procedure Generate( $P, \mathbf{F}_P$ ) is
   $h :=$  a most frequent element of  $\{S[1] : (S, F, b) \in \mathbf{F}_P \text{ and } S \neq \varepsilon\}$ ;
  foreach  $(S, F, b) \in \mathbf{F}_P$  do                                     //  $b = k - d_H(F, PS) \geq 0$ 
    if  $S = \varepsilon$  then  $N(F) := N(F) \cup \{P\}$ ;
    else
       $c := S[1]$ ;
       $\mathbf{F}_{Pc} := \mathbf{F}_{Pc} \cup \{(\text{suf}(S), F, b)\}$ ;
      if  $c \neq h$  and  $b > 0$  then
         $\mathbf{F}_{Ph} := \mathbf{F}_{Ph} \cup \{(\text{suf}(S), F, b - 1)\}$ ;
         $\mathbf{F}_{P\$} := \mathbf{F}_{P\$} \cup \{(\text{suf}(S), F, b - 1)\}$ ;
      foreach  $c \in \Sigma \cup \{\$\}$  such that  $\mathbf{F}_{Pc} \neq \emptyset$  do
        Generate( $Pc, \mathbf{F}_{Pc}$ );

```

The following result from [9] shows that this abstract procedure can be implemented efficiently. In the statement below, we ignore the meaning of the resulting family of strings (which is important for computing the longest common substring of two strings with k mismatches) and focus only on its size and the complexity of its construction.

Theorem A.1 (see [9, Proposition 13]) *Let $\mathbf{F} \subseteq \Sigma^*$ be a finite family of strings and $k \geq 0$ be an integer. Then the family $\mathbf{F}' = \bigcup_{F \in \mathbf{F}} N(F)$ generated by Algorithm 2 has size at most $2^k \binom{\log |\mathbf{F}| + k}{k} |\mathbf{F}|$. Moreover, the compacted trie of \mathbf{F}' can be constructed in $O(2^k |\mathbf{F}| \binom{\log |\mathbf{F}| + k + 1}{k + 1})$ time provided constant-time lcp queries for suffixes of the strings $F \in \mathbf{F}$.*

Let us now inspect how the recursive procedure `processNode` in Algorithm 1 translates one-to-one to procedure `Generate` in Algorithm 2 applied for the family $\mathbf{F} = \{T_i^m : i \in \{1, \dots, n - m + 1\}\}$ to showcase that they are indeed equivalent. For this string family, if \mathbf{F}_P contains a triple (ε, F, b) for some F and b , then all the remaining triples have the first component equal to ε as well as all strings in \mathbf{F} are of the same length.

A node v corresponds to string P and the modified strings in $MS(v)$ correspond to the triples in \mathbf{F}_P in such a way that the set $MS(v)$ (composed of pairs $(U(\alpha), M(\alpha))$) is

$$\{(F, \{(i, P[i]) : i \in \{1, \dots, |P|\}, F[i] \neq P[i]\}) : (S, F, b) \in \mathbf{F}_P\}.$$

In both procedures, we construct light trees, a heavy tree (i.e., the call for Ph), and a wildcard tree (i.e., the call for $P\$$). These trees are constructed in the same manner. The modified strings $\alpha \in MS(v)$ that get copied to the heavy and wildcard trees in Algorithm 1 are those that satisfy $|M(\alpha)| < k$ and $U(\alpha)[\text{depth} + 1] \neq \text{heavyLetter}$. In Algorithm 2, the triples $(S, F, b) \in \mathbf{F}_P$ that get copied to families \mathbf{F}_{Ph} and $\mathbf{F}_{P\$}$ are those that satisfy $b > 0$ and $F[|P| + 1] \neq h$. For $\alpha \in MS(v)$ corresponding to $(S, F, b) \in \mathbf{F}_P$, we have $|M(\alpha)| = k - b$, $|P| = \text{depth}$, and $\text{heavyLetter} = h$.⁴ This yields a bijection between the family of copied modified strings and the family of copied triples. This concludes that indeed both constructions are equivalent.

⁴ Here, we assume that the two algorithms break ties for the heavy letter consistently, e.g., by choosing the lexicographically smallest letter.

715 Finally, the condition about constant-time lcp-queries on strings $F \in \mathbf{F}$ from Theorem A.1, in this case being substrings of the text T , is satisfied with the aid of LCA queries on a suffix tree of T , so the theorem implies Fact 3.10.

Circular Pattern Matching with k Mismatches

Panagiotis Charalampopoulos^{a,b,1}, Tomasz Kociumaka^{c,b,2}, Solon P. Pissis^{d,e}, Jakub Radoszewski^{b,3,4},
Wojciech Rytter^b, Juliusz Straszypiński^{b,3,4}, Tomasz Waleń^{b,4}, Wiktor Zuba^{b,4}

^a*Department of Informatics, King's College London, London, UK*

^b*Institute of Informatics, University of Warsaw, Warsaw, Poland*

^c*Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel*

^d*CWI, Amsterdam, The Netherlands*

^e*Vrije Universiteit, Amsterdam, The Netherlands*

Abstract

We consider the circular pattern matching with k mismatches (k -CPM) problem in which one is to compute the minimal Hamming distance of every length- m substring of T and any cyclic rotation of P , if this distance is no more than k . It is a variation of the well-studied k -mismatch problem. A multitude of papers have been devoted to solving the k -CPM problem, but only average-case upper bounds are known. In this paper, we present the first non-trivial worst-case upper bounds for this problem. Specifically, we show an $\mathcal{O}(nk)$ -time algorithm and an $\mathcal{O}(n + \frac{n}{m} k^4)$ -time algorithm. The latter algorithm applies in an extended way a technique that was very recently developed for the k -mismatch problem [Bringmann et al., SODA 2019].

A preliminary version of this work appeared at FCT 2019. In this version we improve the time complexity of the second algorithm from $\mathcal{O}(n + \frac{n}{m} k^5)$ to $\mathcal{O}(n + \frac{n}{m} k^4)$.

Keywords: circular pattern matching, k -mismatch problem, approximate pattern matching

1. Introduction

Pattern matching is a fundamental problem in computer science [1]. It consists in finding all substrings of a text T of length n that match a pattern P of length m . In many real-world applications, a measure of similarity is usually introduced allowing for *approximate* matches between the given pattern and substrings of the text. The most widely-used similarity measure is the Hamming distance between the pattern and all length- m substrings of the text.

Computing the Hamming distance between P and all length- m substrings of T has been investigated for the past 30 years. The first efficient solution requiring $\mathcal{O}(n\sqrt{m \log m})$ time was independently developed by Abrahamson [2] and Kosaraju [3] in 1987. The k -mismatch version of the problem asks for finding only the substrings of T that are close to P , specifically, at Hamming distance at most k . The first efficient solution to this problem running in $\mathcal{O}(nk)$ time was developed in 1986 by Landau and Vishkin [4].

Email addresses: panagiotis.charalampopoulos@kcl.ac.uk (Panagiotis Charalampopoulos), kociumaka@mimuw.edu.pl (Tomasz Kociumaka), solon.pissis@cwi.nl (Solon P. Pissis), jrad@mimuw.edu.pl (Jakub Radoszewski), rytter@mimuw.edu.pl (Wojciech Rytter), jks@mimuw.edu.pl (Juliusz Straszypiński), walen@mimuw.edu.pl (Tomasz Waleń), w.zuba@mimuw.edu.pl (Wiktor Zuba)

¹Supported by the ERC grant TOTAL agreement no. 677651, a Studentship from the Faculty of Natural and Mathematical Sciences at King's College London and an A. G. Leventis Foundation Educational Grant.

²Supported by ISF grants no. 1278/16 and 1926/19, a BSF grant no. 2018364, and an ERC grant MPM (no. 683064) under the EU's Horizon 2020 Research and Innovation Programme.

³Supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING program of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

⁴Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

It took almost 15 years for a breakthrough result by Amir et al. improving this to $\mathcal{O}(n\sqrt{k\log k})$ [5]. More recently, there has been a resurgence of interest in the k -mismatch problem. Clifford et al. gave an $\mathcal{O}((n/m)(k^2 \log k) + npolylogn)$ -time algorithm [6], which was subsequently improved further by Gawrychowski and Uznański to $\mathcal{O}((n/m)(m + k\sqrt{m})polylogn)$ [7]. Very recently, Chan et al. [8] improved the polylogn factors in the latter solution at the cost of (Monte-Carlo) randomization. Moreover, Gawrychowski and Uznański [7], showed that a significantly faster “combinatorial” algorithm for this problem is rather unlikely.

The k -mismatch problem has also been considered on compressed representations of the text [9, 10, 11, 12, 13], in the parallel model [14], in the streaming model [15, 6, 16], and in the setting of dynamic strings [13]. Furthermore, it has been considered in non-standard stringology models, such as the parameterized model [17] and the order-preserving model [18].

The matching relation (e.g., identity or Hamming distance at most k) in the standard pattern matching setting assumes that the leftmost and rightmost positions of the pattern are conceptually important. In many real-world applications, such as in bioinformatics [19, 20, 21, 22] or in image processing [23, 24, 25, 26], any cyclic shift (rotation) of P is a relevant pattern. In bioinformatics, the position where a sequence starts can be totally arbitrary due to, for instance, arbitrariness in the sequencing of a circular molecular structure or inconsistencies introduced into sequence databases due to different linearisation standards [19]. In image processing, the contours of a shape may be represented through a directional chain code; the latter can be interpreted as a cyclic sequence if the orientation of the image is not important [23]. Thus one is interested in computing the minimal distance of every length- m substring of T and any cyclic rotation of P , if this distance is no more than k . This is the circular pattern matching with k mismatches (k -CPM) problem.

A multitude of papers [27, 28, 29, 30, 31, 32] have thus been devoted to solving the k -CPM problem but, to the best of our knowledge, only average-case upper bounds are known; i.e., in these works the assumption is that text T is uniformly random. The main result states that, after preprocessing pattern P , the average-case optimal search time of $\mathcal{O}(n\frac{k+\log m}{m})$ [33] can be achieved for certain values of the error ratio k/m (see [31, 27] for more details on the preprocessing costs). Note that the exact (no mismatches allowed) version of the CPM problem can be solved as fast as exact pattern matching; namely, in $\mathcal{O}(n)$ time [34].

In this paper, we draw our motivation from (i) the importance of the k -CPM problem in real-world applications and (ii) the fact that no (non-trivial) worst-case upper bounds are known. Trivial here refers to running the fastest-known algorithm for the k -mismatch problem [7] separately for each of the m rotations of P . This yields an $\mathcal{O}(n(m + k\sqrt{m})polylogn)$ -time algorithm for the k -CPM problem. This is clearly unsatisfactory: it is a simple exercise to design an $\mathcal{O}(nm)$ -time or an $\mathcal{O}(nk^2)$ -time algorithm. In an effort to tackle this unpleasant situation, we present two much more efficient algorithms: a simple $\mathcal{O}(nk)$ -time algorithm and an $\mathcal{O}(n + \frac{n}{m}k^4)$ -time algorithm. Our second algorithm applies in an extended way a technique that was developed very recently for k -mismatch pattern matching in grammar compressed strings by Bringmann et al. [10]. We also show that both of our algorithms can be implemented in $\mathcal{O}(m)$ space.

A preliminary version of this work was published as [35].

Our approach. We first consider a simple version of the problem (called ANCHOR-MATCH) in which we are given a position in T (an *anchor*) which belongs to potential k -mismatch circular occurrences of P . A simple $\mathcal{O}(k)$ -time algorithm is given (after linear-time preprocessing) to compute all relevant occurrences. By considering separately each position in T as an anchor we obtain an $\mathcal{O}(nk)$ -time algorithm. The concept of an anchor is extended to the so-called *matching pairs*: when we know a pair of positions, one in P and the other in T , that are aligned. Then comes the idea of a *sample* \mathcal{S} , which is a fragment of P of length $\Theta(m/k)$ which supposedly exactly matches a corresponding fragment in T . We choose $\mathcal{O}(k)$ samples and work for each of them and for windows of T of size $2m$. As it is typical in many versions of pattern matching, our solution is split into periodic and non-periodic cases. If \mathcal{S} is non-periodic the sample occurs only $\mathcal{O}(k)$ times in a window and each occurrence gives a matching pair (and consequently two possible anchors). Then we perform ANCHOR-MATCH for each such anchor. The hard part is the case when \mathcal{S} is periodic. Here we compute all exact occurrences of \mathcal{S} and obtain $\mathcal{O}(k)$ groups of occurrences, each one being an arithmetic progression. Now each group is processed using the approach “few matches or almost periodicity” of Bringmann et al. [10]. In the latter case periodicity is approximate allowing up to k mismatches. Finally, we are able to decrease the exponent of k by one in the complexity using a marking trick.

2. Preliminaries

Let $S = S[0]S[1] \cdots S[n-1]$ be a *string* of length $|S| = n$ over an integer alphabet Σ . The elements of Σ are called *letters*. For two positions i and j on S , we denote by $S[i..j] = S[i] \cdots S[j]$ the *fragment* of S that starts at position i and ends at position j (the fragment is empty, denoted ε , if $j < i$). A *prefix* of S is a fragment that starts at position 0, i.e., of the form $S[0..j]$, and a *suffix* is a fragment that ends at position $n-1$, i.e., of the form $S[i..n-1]$. For an integer p , we define the p th *power* of S , denoted by S^p , as the string obtained from concatenating p copies of S . The string obtained by concatenating infinitely many copies of S is denoted by S^∞ . If S and S' are two strings of the same length, then by $S =_k S'$ we denote the fact that S and S' have at most k mismatches, that is, that the Hamming distance between S and S' does not exceed k .

We say that an integer $1 \leq q \leq |S|$, is a *period* of a string S if $S[i] = S[i+q]$ for $i = 0, \dots, |S| - q - 1$. We denote the smallest period of S by $\text{per}(S)$ and say that a string S is *periodic* if $2\text{per}(S) \leq |S|$. Fine and Wilf's periodicity lemma [36] asserts that if a string of length n has periods p and q and $n \geq p+q - \text{gcd}(p, q)$, then the string has a period $\text{gcd}(p, q)$.

For a string S and integer $0 \leq x < |S|$, by $\text{rot}_x(S)$ we denote the string that is obtained from S by moving the prefix of S of length x to the end. More formally,

$$\text{rot}_x(S) = VU, \text{ where } S = UV \text{ and } |U| = x.$$

We call the string $\text{rot}_x(S)$ a *rotation* of S and often represent rotations of S using the underlying values x .

2.1. Anatomy of Circular Occurrences

In what follows, we denote by m the length of the pattern P and by n the length of the text T . We say that P has a k -*mismatch circular occurrence* (in short, a k -*occurrence*) in T at position p if $T[p..p+m-1] =_k \text{rot}_x(P)$ for some rotation x . In this case, the position x in the pattern is called the *split point* and the position $p + (m-x) \bmod m$ in the text¹ is called the *anchor*. In other words, if $P = UV$ and its rotation VU occurs in T , then the first position of V in P is the split point of this occurrence, and the first position of U in T is the anchor of this occurrence (see Fig. 1).

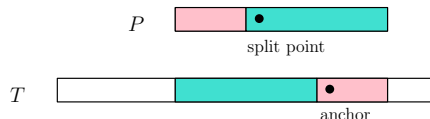


Figure 1: The split point and the anchor for a k -occurrence of P in T .

The main problem in scope can now be stated as follows.

k -CPM PROBLEM

Input: A text T of length n , a pattern P of length m , and a positive integer k .

Output: The positions of all k -occurrences of P in T .

For an integer z , let us denote $\mathbf{W}_z = [z..z+m-1]$ (a *window* of size m). Intuitively, this window corresponds to a length- m fragment of the text T . For a k -occurrence at position p of T with rotation x , we introduce a set of pairs of positions in the fragment of the text and the corresponding positions from the original (unrotated) pattern P :

$$M(p, x) = \{(i, (i - p + x) \bmod m) : i \in \mathbf{W}_p\}.$$

The pairs $(i, j) \in M(p, x)$ are called *matching pairs* of an occurrence p with rotation x . In particular, $(p + ((m-x) \bmod m), 0) \in M(p, x)$. An example is provided in Fig. 2.

¹The modulo operation is needed to handle the trivial rotation with $x = 0$.

$$\begin{array}{ccc}
P = \begin{array}{cccccc} \color{red}{a} & \color{red}{a} & \color{blue}{b} & \color{blue}{b} & \color{blue}{b} & \color{blue}{b} \\ 0 & 1 & \textcircled{2} & 3 & 4 & 5 \\ & & \text{split point}=2 & & & \end{array} &
T = \begin{array}{cccccccccccc} a & a & c & c & \color{blue}{b} & \color{blue}{b} & \color{red}{x} & \color{blue}{b} & \color{red}{a} & \color{red}{a} & a & b \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \textcircled{8} & 9 & 10 & 11 \\ & & & & & & & & \text{anchor}=8 & & & \end{array} & \\
\text{rot}_2(P) = \begin{array}{cccccc} \color{blue}{b} & \color{blue}{b} & \color{blue}{b} & \color{blue}{b} & \color{red}{a} & \color{red}{a} \\ 2 & 3 & 4 & 5 & 0 & 1 \end{array} &
\end{array}$$

Figure 2: A 1-occurrence of a pattern $P = \text{aabbbb}$ in a text $T = \text{aacbbxbaaab}$ at position $p = 4$ with rotation $x = 2$; $M(4, 2) = \{(4, 2), (5, 3), (6, 4), (7, 5), (8, 0), (9, 1)\}$.

3. An $\mathcal{O}(nk)$ -time Algorithm

We first introduce an auxiliary problem in which one wants to compute all k -occurrences of P in T with a given anchor \mathbf{a} . This problem describes the core computational task in our first solution.

ANCHOR-MATCH PROBLEM

Input: A text T of length n , a pattern P of length m , a positive integer k , and a position \mathbf{a} .

Output: All k -occurrences of P in T with anchor \mathbf{a} , represented as a collection of $\mathcal{O}(k)$ intervals.

For a binary string X , by $\|X\|$ we denote the arithmetic sum of characters in X . We define the following auxiliary problem.

LIGHT-FRAGMENTS PROBLEM

Input: Positive integers m , k and a string V of length n over alphabet $\{0, 1\}$ containing $\mathcal{O}(k)$ non-zero characters. The string V is specified by its positions with non-zero characters (sorted increasingly).

Output: The set $A = \{i : \|V[i..i+m-1]\| \leq k\}$ represented as a collection of $\mathcal{O}(k)$ intervals.

Lemma 1. *The LIGHT-FRAGMENTS problem can be solved in $\mathcal{O}(k)$ time.*

Proof. Let I be the set of positions of V with non-zero characters; $|I| = \mathcal{O}(k)$ by definition. We define a piecewise constant function $f : [0..|V| - m + 1] \rightarrow \mathbb{Z}$ such that $f(x) = |I \cap [0..x]|$. Let $g(x) = f(x + m - 1) - f(x - 1)$. Then $g(x) = |I \cap [x..x + m - 1]|$. The function f has $\mathcal{O}(k)$ pieces, so g has $\mathcal{O}(k)$ pieces as well, and both can be computed in $\mathcal{O}(k)$ time since I is sorted. In the end, we report the pieces where g has value at most k . \square

Let us recall a standard algorithmic tool for preprocessing text T . We denote the length of the longest common prefix (resp. suffix) of two strings U and V by $\text{lcp}(U, V)$ (resp. $\text{lcs}(U, V)$). There is an $\mathcal{O}(n)$ -sized data structure answering such queries over suffixes (resp. prefixes) of T in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$ -time preprocessing. It consists of the suffix array of T and a data structure for answering range minimum queries; see [1]. Using the kangaroo method [4, 14], these queries can handle mismatches; after an $\mathcal{O}(n)$ -time preprocessing of T , longest common prefix (resp. suffix) queries with up to k mismatches can be answered in $\mathcal{O}(k)$ time.

Lemma 2. *After $\mathcal{O}(n)$ -time preprocessing of T and P , the ANCHOR-MATCH problem can be solved in $\mathcal{O}(k)$ time for any given k and \mathbf{a} .*

Proof. In the preprocessing, we prepare a data structure for lcp queries in $P\#T$, where $\#$ is a special character that occurs neither in P nor in T .

Consider now a query for an anchor \mathbf{a} over T . Let $L = T[a - m..a - 1]$ and $R = T[a..a + m - 1]$, with $\#$ at out-of-bounds positions of T . We define a binary string L' such that $L'[i] = 1$ if and only if $L[i] \neq P[i]$. We define R' analogously. Let L'' be the longest suffix of L' such that $\|L''\| \leq k$ and R'' be the longest prefix of R' such that $\|R''\| \leq k$.

Using the kangaroo method [4, 14], the strings L'' , R'' can be constructed in $\mathcal{O}(k)$ time. The ANCHOR-MATCH problem now reduces to the LIGHT-FRAGMENTS problem for the string $V = L''R''$. \square

into intervals Z_1, \dots, Z_{n_1} . We obtain a similar partition of $[0..|V| - m]$ into intervals Z'_1, \dots, Z'_{n_2} . We have $n_1, n_2 = \mathcal{O}(k)$.

Let us now fix Z_j and $Z'_{j'}$. First, we check if the condition on the total number of non-zero characters is satisfied for arbitrary $z \in Z_j$ and $z' \in Z'_{j'}$. If so, we compute the set $X = Z'_j \bmod q = \{z' \bmod q : z' \in Z'_{j'}\}$. It is a single circular interval and can be computed in constant time. The required result is

$$\{z \in Z_j : z \bmod q \in X\}.$$

By Lemma 7, this set can be represented as a union of three chains, each with difference q , and, as such, it can be computed in $\mathcal{O}(1)$ time. The conclusion follows. \square

5. An $\mathcal{O}(n + k^5)$ -time Algorithm for Short Texts

In this section, we proceed by assuming that $m \leq n \leq 2m$ and aim at an $\mathcal{O}(n + k^5)$ -time algorithm. In the next sections, we remove this assumption and reduce the exponent of k to 4.

A (*deterministic*) *sample* is a short fragment \mathcal{S} of the pattern P . An occurrence in the text without any mismatch is called *exact*. We introduce a problem of SAMPLE-MATCH that consists in finding all k -occurrences of P in T such that \mathcal{S} matches exactly the corresponding length- $|\mathcal{S}|$ fragment of T .

We split the pattern P into $2k + 3$ samples of length $\lfloor \frac{m}{2k+3} \rfloor$ or $\lceil \frac{m}{2k+3} \rceil$ each. In any k -occurrence of P in T , at least $k + 2$ of the samples match exactly the corresponding fragments of T (up to k samples may match with a mismatch and at most one sample may contain the split point).

Remark 9. We require at least $k + 2$ samples (instead of just one) to match exactly for two reasons: (1) in order to have more than a half of them match exactly, which will guarantee that the interval chains that are obtained from applications of the ALIGNED-LIGHT-SUM problem have the same difference and thus can be unioned using Lemma 5 (see the proof of Proposition 21); and (2) for the marking trick in the next section.

5.1. Matching Non-Periodic Samples

We solve the SAMPLE-MATCH problem for a non-periodic sample \mathcal{S} in $\mathcal{O}(k^2)$ time in two steps. First, we show an $\mathcal{O}(k)$ -time solution of following PAIR-MATCH problem, asking to compute all k -occurrences of P in T which align $T[i]$ with $P[j]$.

PAIR-MATCH PROBLEM

Input: A text T of length n , a pattern P of length m , a positive integer k , and two integers $i \in [0..n-1]$ and $j \in [0..m-1]$.

Output: The set $A(i, j)$ of all positions in T where we have a k -mismatch occurrence of $\text{rot}_x(P)$ for some x such that (i, j) is a matching pair.

We then reduce the SAMPLE-MATCH problem to $\mathcal{O}(k)$ instances of the PAIR-MATCH problem, where j is the starting position of \mathcal{S} in P and i is an occurrence of \mathcal{S} in T ; notice that there are $\mathcal{O}(k)$ such occurrences.

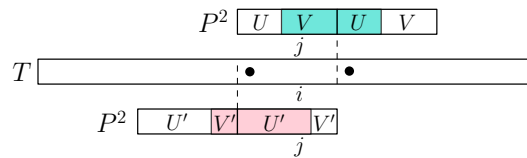


Figure 5: The two possible anchors for the matching pair of positions (i, j) are shown as bullet points. A possible k -occurrence of P in T corresponding to the left (resp. right) anchor is shown below T (above T , resp.). Note that $P^2 = PP$.

Lemma 10. *After $\mathcal{O}(n)$ -time preprocessing of T and P , the PAIR-MATCH problem can be solved in $\mathcal{O}(k)$ time for any given k, i, j , with the output represented as a collection of $\mathcal{O}(k)$ intervals.*

Proof. Recall that the ANCHOR-MATCH problem returns all k -occurrences of P in T with a given anchor. The PAIR-MATCH problem can be essentially reduced to the ANCHOR-MATCH problem, since for a given matching pair of characters in P and T , there are at most two ways of choosing the anchor depending on the relation between j and a split point: these are $i - j$ and $i + |P| - j$ (see Fig. 5). Clearly, we choose $i - j$ as an anchor only if $i - j \geq 0$ and $i + |P| - j$ only if $i + |P| - j < |T|$. We then have to take the intersection of the answer with $[i - m + 1 .. i]$ to ensure that the k -occurrence contains position i . \square

Lemma 11. *After $\mathcal{O}(n)$ -time preprocessing, the SAMPLE-MATCH problem for a non-periodic sample \mathcal{S} can be solved in $\mathcal{O}(k^2)$ time, which the output represented as a union of $\mathcal{O}(k^2)$ intervals of occurrences.*

Proof. Since \mathcal{S} is non-periodic, it has $\mathcal{O}(k)$ occurrences in T , which can be computed in $\mathcal{O}(k)$ time after an $\mathcal{O}(n)$ -time preprocessing using IPM queries [37, 38] in $P\#T$. Let j be the starting position of \mathcal{S} in P and i be a starting position of an occurrence of \mathcal{S} in T . For each of the $\mathcal{O}(k)$ such pairs (i, j) , the computation reduces to the PAIR-MATCH problem for i and j . The statement follows by Lemma 10. \square

5.2. Matching Periodic Samples

Let us assume that \mathcal{S} is periodic with $q := \text{per}(\mathcal{S}) \leq \frac{1}{2}|\mathcal{S}|$. A fragment of T containing an inclusion-maximal arithmetic sequence of occurrences of \mathcal{S} in T with difference q is called here an \mathcal{S} -run. If \mathcal{S} matches a fragment in the text, then the match belongs to an \mathcal{S} -run. For example, the underlined fragment of $T = \text{bbabababaa}$ is an \mathcal{S} -run for $\mathcal{S} = \text{abab}$.

Lemma 12. *If \mathcal{S} is periodic, the number of \mathcal{S} -runs in the text is $\mathcal{O}(k)$ and they can all be computed in $\mathcal{O}(k)$ time after $\mathcal{O}(n)$ -time preprocessing of T and P .*

Proof. We construct the data structure for IPM queries on $P\#T$. This allows us to compute the set of all occurrences of \mathcal{S} in T as a collection of $\mathcal{O}(k)$ arithmetic sequences with difference $\text{per}(\mathcal{S})$. We then check for every two consecutive sequences if they can be joined together. This takes $\mathcal{O}(k)$ time and results in $\mathcal{O}(k)$ \mathcal{S} -runs. \square

For two equal-length strings S and S' , we denote the set of their *mismatches* by

$$\text{Mis}(S, S') = \{i \in [0 .. |S| - 1] : S[i] \neq S'[i]\}.$$

We say that position a in S is a *misperiod* with respect to a fragment $S[i .. j]$ if $S[a] \neq S[b]$ where b is the unique position such that $b \in [i .. j]$ and $(j - i + 1) \mid (b - a)$. We define the set $\text{LeftMisper}_k(S, i, j)$ as the set of k maximal misperiods that are smaller than i and $\text{RightMisper}_k(S, i, j)$ as the set of k minimal misperiods that are greater than j . Each of the sets can have less than k elements if the corresponding misperiods do not exist. We further define

$$\text{Misper}_k(S, i, j) = \text{LeftMisper}_k(S, i, j) \cup \text{RightMisper}_k(S, i, j)$$

and $\text{Misper}(S, i, j) = \bigcup_{k=0}^{\infty} \text{Misper}_k(S, i, j)$.

The following lemma captures a combinatorial property behind the new technique of Bringmann et al. [10]. The intuition is shown in Fig. 6.

Lemma 13. *Assume that $S =_k S'$ and that $S[i .. j] = S'[i .. j]$. Let*

$$I = \text{Misper}_{k+1}(S, i, j) \text{ and } I' = \text{Misper}_{k+1}(S', i, j).$$

If $I \cap I' = \emptyset$, then $\text{Mis}(S, S') = I \cup I'$, $I = \text{Misper}(S, i, j)$, and $I' = \text{Misper}(S', i, j)$.

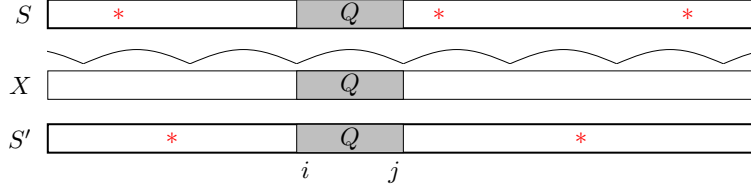


Figure 6: Let S , S' , and X be equal-length strings such that X is a factor of Q^∞ and $S[i..j] = S'[i..j] = X[i..j] = Q$. The asterisks in S denote the positions in $\text{Mis}(S, X)$, or equivalently, the misperiods with respect to $S[i..j]$. Similarly for S' . One can observe that $\text{Mis}(S, X) \cap \text{Mis}(S', X) = \emptyset$ holds in this situation, and therefore $\text{Mis}(S, X) \cup \text{Mis}(S', X) = \text{Mis}(S, S')$.

Proof. Let $J = \text{Misper}(S, i, j)$ and $J' = \text{Misper}(S', i, j)$. We first observe that $I \cup I' \subseteq \text{Mis}(S, S')$ since $I \cap I' = \emptyset$. Then, $S =_k S'$ implies that $|\text{Mis}(S, S')| \leq k$ and hence $|I| \leq k$ and $|I'| \leq k$, which in turn implies that $I = J$ and $I' = J'$. The observation that $\text{Mis}(S, S') \subseteq J \cup J'$ concludes the proof. \square

A string S is k -periodic w.r.t. a fragment $S[i..i+q-1]$ if $|\text{Misper}(S, i, i+q-1)| \leq k$. In this case, q is called the k -period. In particular, in the conclusion of the above lemma, S is $|I|$ -periodic w.r.t. $S[i..j]$ and S' is $|I'|$ -periodic w.r.t. $S'[i..j]$. This notion forms the basis of the following auxiliary problem, where we search for k -occurrences in which the rotation of the pattern and the fragment of the text are k -periodic for the same period q .

Let U and V be two strings and J and J' be sets containing positions in U and V , respectively. We say that length- m fragments $U[p..p+m-1]$ and $V[x..x+m-1]$ are (J, J') -disjoint if the sets $(\mathbf{W}_p \cap J) \ominus p$ and $(\mathbf{W}_x \cap J') \ominus x$ are disjoint.

Example 14. If $J = \{2, 4, 11, 15, 16, 17\}$, $J' = \{5, 6, 15, 18, 19\}$, and $m = 12$, then $U[3..14]$ and $V[6..17]$ are (J, J') -disjoint for the following strings, with characters at positions in J and J' replaced by bullets:

$$\begin{aligned}
 U &= \text{ab} \bullet \boxed{\text{a} \bullet \text{b abc ab} \bullet \text{abc}} \bullet \bullet \bullet \\
 V &= \text{abc ab} \bullet \boxed{\bullet \text{bc abc abc} \bullet \text{bc}} \bullet \bullet \bullet
 \end{aligned}$$

Let us introduce an auxiliary problem that is obtained in the case that is shown in the conclusion of the above lemma (i.e., when misperiods in the rotation of P and the corresponding fragment of T are not aligned); see also Fig. 7.

PERIODIC-PERIODIC-MATCH PROBLEM

Input: Positive integers k and m , strings U and V such that $m \leq |U|, |V| \leq 2m$, integers i, i', q such that $U[i..i+q-1]$ matches $V[i'..i'+q-1]$, and two sets of size $\mathcal{O}(k)$:

$$J = \text{Misper}(U, i, i+q-1), \quad J' = \text{Misper}(V, i', i'+q-1).$$

(The strings U and V are not stored explicitly.)

Output: The set of positions p in U for which there exists a (J, J') -disjoint k -occurrence $U[p..p+m-1]$ of $V[x..x+m-1]$ for x such that

$$i - p \equiv i' - x \pmod{q}.$$

Intuitively, the modulo condition on the output of the PERIODIC-PERIODIC-MATCH problem corresponds to the fact that the approximate periodicity is aligned.

In the PERIODIC-PERIODIC-MATCH problem, we search for k -occurrences in which none of the misperiods in J and J' are aligned. In this case, each of the misperiods accounts for one mismatch in the k -occurrence. In the lemma below, we reduce the PERIODIC-PERIODIC-MATCH problem to the ALIGNED-LIGHT-SUM problem, in which we only require that the total number of misperiods in an occurrence is at most k . This

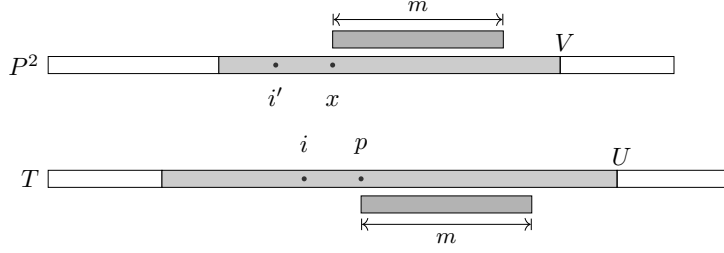


Figure 7: In the case of a periodic sample, we have two $\mathcal{O}(k)$ -periodic fragments U and V . We find all p in U such that for some position x in V , the fragments of length m starting at positions p and x are at Hamming distance at most k . If q is a period, then $i - p \equiv i' - x \pmod{q}$ due to synchronization of periodicities.

way, all the (J, J') -disjoint k -occurrences can be found. Also additional occurrences where two misperiods are aligned can be reported, but they are still valid k -occurrences (actually, k' -occurrences for some $k' < k$).

Lemma 15. *We can compute in $\mathcal{O}(k^2)$ time a set of k -occurrences of P in T represented as $\mathcal{O}(k^2)$ interval chains, each with difference q , that is a superset of the solution to the PERIODIC-PERIODIC-MATCH problem.*

Proof. In the PERIODIC-PERIODIC-MATCH problem, the modulo condition forces the exact occurrences of the approximate period to match. Hence, it guarantees that all the positions except the misperiod positions match. Now, the ALIGNED-LIGHT-SUM problem highlights these positions inside the string.

Claim 16. The PERIODIC-PERIODIC-MATCH problem can be reduced in $\mathcal{O}(k)$ time to the ALIGNED-LIGHT-SUM problem so that we obtain a superset of the desired result. The potential extra positions do not satisfy only the (J, J') -disjointness condition.

Proof. Let the parameters m, k , and q remain unchanged. We create binary strings U' and V' of length $|U|$ and $|V|$, respectively, with positions with non-zero characters in the sets J and J' , respectively. Then we prepend U' with $z = (i' - i) \bmod q$ zeros. Let A be the solution to the ALIGNED-LIGHT-SUM problem for U' and V' . Then $(A \ominus z) \cap \mathbb{Z}_{\geq 0}$ is a superset of the solution to PERIODIC-PERIODIC-MATCH; the elements of the set that correspond to matches where non-zero elements of the strings U', V' were aligned do not satisfy the disjointness condition. \square

Now, the thesis follows from Lemma 8. \square

Let us further define

$$\text{PAIRS-MATCH}(T, I, P, J) = \bigcup_{i \in I, j \in J} \text{PAIR-MATCH}(T, i, P, j).$$

Let A be a set of positions in a string S and m be a positive integer. We then denote $A \bmod m = \{a \bmod m : a \in A\}$ and by $\text{frag}_A(S)$ we denote the fragment $S[\min(A) \dots \max(A)]$. We provide pseudocode for an algorithm that computes all k -occurrences of P such that \mathcal{S} matches an exact occurrence in T contained in a given \mathcal{S} -run (see Algorithm 1); inspect also Fig. 8. Let $p_{\mathcal{S}}$ denote the starting position of \mathcal{S} in P , and let $m_{\mathcal{S}} = |\mathcal{S}|$.

Lemma 17. *After $\mathcal{O}(n)$ -time preprocessing of T and P , algorithm Run-Sample-Matching works in $\mathcal{O}(k^3)$ time and returns a compact representation that consists of $\mathcal{O}(k^3)$ intervals and $\mathcal{O}(k^2)$ interval chains, each with difference q . Moreover, if there is at least one interval chain, then some rotation of the pattern P is k -periodic with a k -period $\text{per}(\mathcal{S})$.*

Proof. See Algorithm 1. The sets J and J' can be computed in $\mathcal{O}(k)$ time:

Claim 18. If S is a string of length n , then the sets $\text{RightMisper}_k(S, i, j)$ and $\text{LeftMisper}_k(S, i, j)$ can be computed in $\mathcal{O}(k)$ time after $\mathcal{O}(n)$ -time preprocessing.

Data: A periodic fragment \mathcal{S} of pattern P , an \mathcal{S} -run R in the text T , integers $q = \text{per}(\mathcal{S})$ and k .

Result: A compact representation of k -occurrences of P in T including all k -occurrences where \mathcal{S} matches exactly a fragment of R in T .

Let $R = T[s..s + |R| - 1]$;

$J := \text{Misper}_{k+1}(T, s, s + q - 1)$; $\{ \mathcal{O}(k) \text{ time} \}$

$J' := \text{Misper}_{k+1}(P^2, m + p_{\mathcal{S}}, m + p_{\mathcal{S}} + q - 1)$; $\{ \mathcal{O}(k) \text{ time} \}$

$U := \text{frag}_J(T)$; $V := \text{frag}_{J'}(P^2)$;

$Y := \text{PERIODIC-PERIODIC-MATCH}(U, V)$; $\{ \mathcal{O}(k^2) \text{ time} \}$

$Y := Y \oplus \min(J)$;

$J' := J' \bmod m$;

$X := \text{PAIRS-MATCH}(T, J, P, J')$; $\{ \mathcal{O}(k^3) \text{ time} \}$

return $X \cup Y$;

Algorithm 1: Run-Sample-Matching

Proof. For $\text{RightMisper}_k(S, i, j)$, we use the kangaroo method [4, 14] to compute the longest common prefix with at most k mismatches of $S[j + 1..n - 1]$ and U^∞ for $U = S[i..j]$. The value $\text{lcp}(X^\infty, Y)$ for a fragment X and a suffix Y of a string S , occurring at positions a and b , respectively, can be computed in constant time as follows. If $\text{lcp}(S[a..n - 1], S[b..n - 1]) < |X|$ then we are done. Otherwise the answer is given by $|X| + \text{lcp}(S[b..n - 1], S[b + |X|..n - 1])$. The computations for $\text{LeftMisper}_k(S, i, j)$ are symmetric. \square

The $\mathcal{O}(k^3)$ and $\mathcal{O}(k^2)$ time complexities of computing X and Y follow from Lemmas 10 and 15, respectively (after $\mathcal{O}(n)$ -time preprocessing). The sets X and Y consist of $\mathcal{O}(k^3)$ intervals and $\mathcal{O}(k^2)$ interval chains, each with difference q .

As for the “moreover” statement, by Lemma 13, if any occurrence q is reported in the PERIODIC-PERIODIC-MATCH problem, then it implies the existence of x such that $V[x..x + m - 1]$ is k -periodic with a k -period q . However, $V[x..x + m - 1]$ is a rotation of the pattern P . This concludes the proof. \square

The correctness of the algorithm follows from Lemma 13, as shown in the lemma below.

Lemma 19. *Assume $n \leq 2m$. Let \mathcal{S} be a periodic sample in P with smallest period q and R be an \mathcal{S} -run in T . Let X and Y be defined as in the pseudocode of Run-Sample-Matching. Then $X \cup Y$ is a set of k -occurrences of P in T which is a superset of the solution to SAMPLE-MATCH for \mathcal{S} in R .*

Proof. Both PAIR-MATCH and PERIODIC-PERIODIC-MATCH problems return positions of k -occurrences of P in T . It suffices to show that $p \in X \cup Y$ if $\text{rot}_x(P)$ has a k -mismatch occurrence in T at position p such that the designated fragment \mathcal{S} matches a fragment of R exactly. We assume that the split point x in P is to the right of \mathcal{S} , i.e., that $x \geq p_{\mathcal{S}} + m_{\mathcal{S}}$. The opposite case—that $x < p_{\mathcal{S}}$ —can be handled analogously.

Let $J = \text{Misper}_{k+1}(T, s, s + q - 1)$ and $J' = \text{Misper}_{k+1}(P^2, m + p_{\mathcal{S}}, m + p_{\mathcal{S}} + q - 1)$. We define L_1 and L_2 as the subsets of J and J' , respectively, that are relevant for this k -occurrence, i.e.,

$$L_1 = J \cap \mathbf{W}_p, \quad L_2 = J' \cap \mathbf{W}_x.$$

Further let $L'_2 = L_2 \bmod m$. If any $i \in L_1$ and $j \in L'_2$ are a matching pair for this k -occurrence, then it will be found in the PAIRS-MATCH problem, i.e., $p \in X$. Let us henceforth consider the opposite case.

Let $S = T[p..p + m - 1]$, $S' = \text{rot}_x(P)$, and $i = m - x + p_{\mathcal{S}}$ be the starting position of $Q = S[0..q - 1]$ in both strings. Further let $I = L_1 \ominus p$ and $I' = L_2 \ominus x$. We have that $I \cap I' = \emptyset$ by our assumption that misperiods do not align. We make the following claim.

Claim 20. $\text{Mis}(S, S') = I \cup I'$.

Proof. Note that $I = \text{Misper}_{k+1}(S, i, i + q - 1)$ and $I' = \text{Misper}_{k+1}(S', i, i + q - 1)$. The former equality follows from the fact that $\text{Misper}_{k+1}(T, s, s + q - 1) = \text{Misper}_{k+1}(T, t, t + q - 1)$ for any $t \in [s..s + |R| - q]$. We can thus directly apply Lemma 13 to strings S and S' . \square

In particular, $|I| + |I'| \leq k$. Moreover, $\min(J) < p$ and $p + m - 1 < \max(J)$ as well as $\min(J') < x$ and $x + m - 1 < \max(J')$, since otherwise we would have $|I| \geq k + 1$ or $|I'| \geq k + 1$. In conclusion, this k -occurrence will be found in the PERIODIC-PERIODIC-MATCH problem, i.e., $p \in Y$. \square

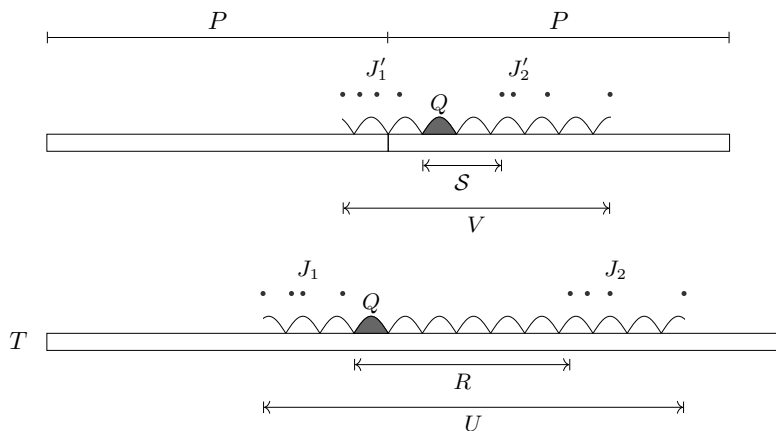


Figure 8: More detailed setting in Algorithm 1; $J = J_1 \cup J_2$, $J' = J'_1 \cup J'_2$.

5.3. Algorithm Summary

Proposition 21. *If $m \leq n \leq 2m$, the k -CPM problem can be solved in $\mathcal{O}(n + k^5)$ time and $\mathcal{O}(n)$ space.*

Proof. We split the pattern into $2k + 3$ fragments and choose a sample \mathcal{S} among them in every possible way.

If the sample \mathcal{S} is not periodic, we use the algorithm of Lemma 11 for SAMPLE-MATCH in $\mathcal{O}(k^2)$ time (after $\mathcal{O}(n)$ -time preprocessing). It returns a representation of k -occurrences as a union of $\mathcal{O}(k^2)$ intervals.

If the sample \mathcal{S} is periodic, we need to find all \mathcal{S} -runs in T . By Lemma 12, there are $\mathcal{O}(k)$ of them and they can all be computed in $\mathcal{O}(k)$ time (after $\mathcal{O}(n)$ -time preprocessing). For every such \mathcal{S} -run R , we apply the Run-Sample-Matching algorithm. Its correctness follows from Lemma 19. By Lemma 17, it takes $\mathcal{O}(k^3)$ time and returns $\mathcal{O}(k^3)$ intervals and $\mathcal{O}(k^2)$ interval chains, each with difference $\text{per}(\mathcal{S})$, of k -occurrences of P in T (after $\mathcal{O}(n)$ -time preprocessing). Over all \mathcal{S} -runs, this takes $\mathcal{O}(k^4)$ time after the preprocessing and returns $\mathcal{O}(k^4)$ intervals and $\mathcal{O}(k^3)$ interval chains.

By Lemma 17, if any interval chains are reported in Run-Sample-Matching, then some rotation of the pattern is k -periodic with a k -period $\text{per}(\mathcal{S})$. Then, at least $k + 2$ of the $2k + 3$ pattern fragments do not contain misperiods and hence they must have a period $q = \text{per}(\mathcal{S})$. This is actually their smallest period, for if one of these fragments \mathcal{S}' had a period $q' < q$, then $|\mathcal{S}'| \geq |\mathcal{S}| - 1$ and, by Fine and Wilf's periodicity lemma [36], \mathcal{S}' would have a period $q'' = \text{gcd}(q, q') < q$, which would imply that Q would also have a period q'' , and hence \mathcal{S} as well. Thus, throughout the course of the algorithm, Run-Sample-Matching can only return interval chains of period $\text{per}(\mathcal{S})$ by the pigeonhole principle.

In total, SAMPLE-MATCH takes $\mathcal{O}(k^4)$ time for a given sample (after preprocessing), $\mathcal{O}(n + k^5)$ time in total, and returns $\mathcal{O}(k^5)$ intervals and $\mathcal{O}(k^4)$ interval chains of k -occurrences, each with the same difference q . Let us note that an interval is a special case of an interval chain with an arbitrary difference, say, 1. We then apply Lemma 5 to compute the union of all chains of occurrences and the union of all intervals in $\mathcal{O}(n + k^5)$ total time. In the end, we return the union of the two unions.

In order to bound the space required by our algorithm by $\mathcal{O}(n)$, we do not store each interval chain explicitly throughout the execution of the algorithm. Instead, for each interval chain, we increment/decrement

a constant number of cells in a (\mathcal{G}_q -shaped for interval chains or \mathcal{G}_1 -shaped for intervals) 2D array of size $\mathcal{O}(n)$ as in the proof of Lemma 5, and compute the union of all such interval chains in the end. \square

6. An $\mathcal{O}(n + k^4)$ -time Algorithm for Short Texts

Let us observe that for each non-periodic fragment \mathcal{S} , we have to solve $\mathcal{O}(k)$ instances of PAIR-MATCH, while for each periodic fragment \mathcal{S} and each \mathcal{S} -run, we obtain two sets J and J' , each of cardinality $\mathcal{O}(k)$, where each pair of elements in $J \times J'$ requires us to solve an instance of PAIR-MATCH. Further, recall that each instance of PAIR-MATCH reduces to two calls to our $\mathcal{O}(k)$ -time algorithm for ANCHOR-MATCH. We thus solve $\mathcal{O}(k^4)$ ANCHOR-MATCH instances in total, yielding a total time complexity of $\mathcal{O}(k^5)$. As can be seen in the proof of Proposition 21 and the pseudocode, this is the only bottleneck of our algorithm, with everything else requiring $\mathcal{O}(n + k^4)$ time. We will decrease the number of calls to ANCHOR-MATCH by using a marking trick.

We first present a simple application of the marking trick. Suppose that we are in the standard (non-circular) k -mismatch problem, where we are to find all k -mismatch occurrences of a pattern P of length m in a text T of length n , and $n \leq 2m$. Further, suppose that P is square-free, or, in other words, that it is nowhere periodic. Let us consider the following algorithm: We split the pattern into $k + 1$ fragments of length roughly m/k each. Then, at each k -occurrence of P in T , at least one of the $k + 1$ fragments must match exactly. We then find the $\mathcal{O}(k)$ such exact matches of each fragment in T and each of them nominates a position for a possible k -occurrence of P . We thus have $\mathcal{O}(k^2)$ candidate positions in total to verify.

Now consider the following refinement of this algorithm: We split the pattern into $2k$ fragments (instead of $k + 1$), each of length roughly $m/(2k)$. Then, at each k -occurrence of P in T , at least k of the fragments must match exactly; we exploit this fact as follows: Each exact occurrence of a fragment in T gives a mark to the corresponding position for a k -occurrence of P . There are thus $\mathcal{O}(k^2)$ marks given in total. However, we only need to verify positions with at least k marks and these are now $\mathcal{O}(k)$ in total. An illustration is provided in Fig. 9.



Figure 9: We consider $k = 4$. To the left: a candidate starting position for P given by an exact match of one of the 5 fragments. To the right: a candidate starting position for P given by exact matches of 4 of the 8 fragments.

Let us get back to the k -CPM problem. We have the following fact.

Fact 22. *The algorithm underlying Proposition 21 returns each k -occurrence either through a call to PERIODIC-PERIODIC-MATCH or through at least $k + 2$ calls to ANCHOR-MATCH.*

Proof. Let us fix a k -occurrence p of P in T . Since at least $k + 2$ out of the $2k + 3$ samples must match exactly in this k -occurrence, the algorithm must return p through at least $k + 2$ calls to SAMPLE-MATCH. For each of these calls, the k -occurrence p is returned through a call to PERIODIC-PERIODIC-MATCH or through (at least one) call to ANCHOR-MATCH. Thus, if p is not returned by any of the calls to PERIODIC-PERIODIC-MATCH, it must be returned by at least $k + 2$ calls to ANCHOR-MATCH. \square

We run the algorithm yielding Proposition 21 with a single difference: Instead of processing each instance of PAIR-MATCH separately, we apply the marking trick in order to decrease the exponent of k by one. This is achieved by a reduction in the number of calls to the algorithm that solves ANCHOR-MATCH. For each of the $\mathcal{O}(k^4)$ instances of PAIR-MATCH, we mark the two possible anchors for a k -occurrence and note that only anchors with at least $k + 2$ marks need to be verified; these are $\mathcal{O}(k^3)$ in total. Finally, for each such anchor, we apply our solution to the ANCHOR-MATCH problem, which requires $\mathcal{O}(k)$ time, hence obtaining an $\mathcal{O}(n + k^4)$ -time algorithm. The correctness of this approach follows from Fact 22. We arrive at the following result.

Proposition 23. *If $m \leq n \leq 2m$, the k -CPM problem can be solved in $\mathcal{O}(n + k^4)$ time and $\mathcal{O}(n)$ space.*

7. Final Result

Both Propositions 3 and 23 use $\mathcal{O}(n)$ space. Moreover, Proposition 23 assumes that $n \leq 2m$. In order to solve the general version of the k -CPM problem, where n is arbitrarily large, efficiently and using $\mathcal{O}(m)$ space, we use the so-called *standard trick*: we split the text into $\mathcal{O}(n/m)$ fragments, each of length $2m$ (perhaps apart from the last one), starting at positions equal to $0 \bmod m$.

We need, however, to ensure that the data structures for answering lcp, lcs, and other internal queries over each such fragment of the text can be constructed in $\mathcal{O}(m)$ time when the input alphabet Σ is large. As a preprocessing step, we hash the letters of the pattern using perfect hashing. For each key, we assign a unique identifier from $\{1, \dots, m\}$. This takes $\mathcal{O}(m)$ time (with high probability) and space [41]. When reading a fragment F of length (at most) $2m$ of the text, we look up its letters using the hash table. If a letter is in the hash table, we replace it in F by its rank value; otherwise, we replace it by rank $m + 1$. We can now construct the data structures in $\mathcal{O}(m)$ time, and thus our algorithms can be implemented in $\mathcal{O}(m)$ space.

If $\Sigma = \{1, \dots, n^{\mathcal{O}(1)}\}$, the same bounds can be achieved deterministically. Specifically, we consider two cases. If $m > \sqrt{n}$, we sort the letters of every text fragment and of the pattern in $\mathcal{O}(m)$ time per fragment because n is polynomial in m and $|\Sigma|$ is polynomial in n . Then, we can merge the two sorted lists and replace the letters in the pattern and the text fragments by their ranks. Otherwise (if $m \leq \sqrt{n}$), we construct a deterministic dictionary for the letters of the pattern in $\mathcal{O}(m \log^2 \log m)$ time [42]. The dictionary uses $\mathcal{O}(m)$ space and answers queries in constant time; we use it instead of perfect hashing in the previous solution.

We combine Propositions 3 and 23 with the above discussion to get our final result.

Theorem 24. *Circular Pattern Matching with k Mismatches can be solved in $\mathcal{O}(\min(nk, n + \frac{n}{m} k^4))$ time and $\mathcal{O}(m)$ space.*

Our algorithms output all positions in the text where some rotation of the pattern occurs with k mismatches. It is not difficult to extend the algorithms to output, for each of these positions, a corresponding witness rotation of the pattern.

References

- [1] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on strings, Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.
- [2] K. R. Abrahamson, Generalized string matching, SIAM J. Comput. 16 (6) (1987) 1039–1051. doi:10.1137/0216067.
- [3] S. R. Kosaraju, Efficient string matching, manuscript (1987).
- [4] G. M. Landau, U. Vishkin, Efficient string matching with k mismatches, Theor. Comput. Sci. 43 (1986) 239–249. doi:10.1016/0304-3975(86)90178-7.
- [5] A. Amir, M. Lewenstein, E. Porat, Faster algorithms for string matching with k mismatches, J. Algorithms 50 (2) (2004) 257–275. doi:10.1016/S0196-6774(03)00097-X.
- [6] R. Clifford, A. Fontaine, E. Porat, B. Sach, T. Starikovskaya, The k -mismatch problem revisited, in: R. Krauthgamer (Ed.), 27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, SIAM, 2016, pp. 2039–2052. doi:10.1137/1.9781611974331.ch142.
- [7] P. Gawrychowski, P. Uznański, Towards unified approximate pattern matching for Hamming and L_1 distance, in: I. Chatzigiannakis, C. Kaklamani, D. Marx, D. Sannella (Eds.), Automata, Languages, and Programming, ICALP 2018, Vol. 107 of LIPIcs, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, pp. 62:1–62:13. doi:10.4230/LIPIcs.ICALP.2018.62.
- [8] T. M. Chan, S. Golan, T. Kociumaka, T. Kopelowitz, E. Porat, Approximating text-to-pattern hamming distances, in: K. Makarychev, Y. Makarychev, M. Tulsiani, G. Kamath, J. Chuzhoy (Eds.), Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, ACM, 2020, pp. 643–656. doi:10.1145/3357713.3384266.
- [9] P. Bille, R. Fagerberg, I. L. Gørtz, Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts, ACM Trans. Algorithms 6 (1) (2009) 3:1–3:14. doi:10.1145/1644015.1644018.
- [10] K. Bringmann, P. Wellnitz, M. Künnemann, Few matches or almost periodicity: Faster pattern matching with mismatches in compressed texts, in: T. M. Chan (Ed.), 30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, SIAM, 2019, pp. 1126–1145. doi:10.1137/1.9781611975482.69.
- [11] P. Gawrychowski, D. Straszak, Beating $\mathcal{O}(nm)$ in approximate LZW-compressed pattern matching, in: L. Cai, S. Cheng, T. W. Lam (Eds.), Algorithms and Computation, ISAAC 2013, Vol. 8283 of LNCS, Springer, 2013, pp. 78–88. doi:10.1007/978-3-642-45030-3_8.

- [12] A. Tiskin, [Threshold approximate matching in grammar-compressed strings](#), in: J. Holub, J. Zdárek (Eds.), Prague Stringology Conference 2014, PSC 2014, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014, pp. 124–138.
URL <http://www.stringology.org/event/2014/p12.html>
- [13] P. Charalampopoulos, T. Kociumaka, P. Wellnitz, Faster approximate pattern matching: A unified approach, CoRR abs/2004.08350. [arXiv:2004.08350](#).
- [14] Z. Galil, R. Giancarlo, Parallel string matching with k mismatches, *Theor. Comput. Sci.* 51 (1987) 341–348. [doi:10.1016/0304-3975\(87\)90042-9](#).
- [15] B. Porat, E. Porat, Exact and approximate pattern matching in the streaming model, in: 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009, IEEE Computer Society, 2009, pp. 315–323. [doi:10.1109/FOCS.2009.11](#).
- [16] R. Clifford, T. Kociumaka, E. Porat, The streaming k -mismatch problem, in: T. M. Chan (Ed.), 30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, SIAM, 2019, pp. 1106–1125. [doi:10.1137/1.9781611975482.68](#).
- [17] C. Hazay, M. Lewenstein, D. Sokol, Approximate parameterized matching, *ACM Trans. Algorithms* 3 (3) (2007) 29. [doi:10.1145/1273340.1273345](#).
- [18] P. Gawrychowski, P. Uznański, Order-preserving pattern matching with k mismatches, *Theor. Comput. Sci.* 638 (2016) 136–144. [doi:10.1016/j.tcs.2015.08.022](#).
- [19] L. A. K. Ayad, S. P. Pissis, MARS: improving multiple circular sequence alignment using refined sequences, *BMC Genomics* 18 (1) (2017) 86. [doi:10.1186/s12864-016-3477-5](#).
- [20] R. Grossi, C. S. Iliopoulos, R. Mercas, N. Pisanti, S. P. Pissis, A. Retha, F. Vayani, Circular sequence comparison: algorithms and applications, *Algorithms Mol. Biol.* 11 (2016) 12. [doi:10.1186/s13015-016-0076-6](#).
- [21] C. S. Iliopoulos, S. P. Pissis, M. S. Rahman, Searching and indexing circular patterns, in: M. Elloumi (Ed.), *Algorithms for Next-Generation Sequencing Data, Techniques, Approaches, and Applications*, Springer, 2017, pp. 77–90. [doi:10.1007/978-3-319-59826-0_3](#).
- [22] C. Barton, C. S. Iliopoulos, R. Kundu, S. P. Pissis, A. Retha, F. Vayani, Accurate and efficient methods to improve multiple circular sequence alignment, in: E. Bampis (Ed.), *Experimental Algorithms, SEA 2015, Vol. 9125 of LNCS*, Springer, 2015, pp. 247–258. [doi:10.1007/978-3-319-20086-6_19](#).
- [23] L. A. K. Ayad, C. Barton, S. P. Pissis, A faster and more accurate heuristic for cyclic edit distance computation, *Pattern Recognit. Lett.* 88 (2017) 81–87. [doi:10.1016/j.patrec.2017.01.018](#).
- [24] V. Palazón-González, A. Marzal, Speeding up the cyclic edit distance using LAESA with early abandon, *Pattern Recognit. Lett.* 62 (2015) 1–7. [doi:10.1016/j.patrec.2015.04.013](#).
- [25] V. Palazón-González, A. Marzal, J. M. Vilar, On hidden Markov models and cyclic strings for shape recognition, *Pattern Recognit.* 47 (7) (2014) 2490–2504. [doi:10.1016/j.patcog.2014.01.018](#).
- [26] V. Palazón-González, A. Marzal, On the dynamic time warping of cyclic sequences for shape retrieval, *Image Vision Comput.* 30 (12) (2012) 978–990. [doi:10.1016/j.imavis.2012.08.012](#).
- [27] K. Fredriksson, G. Navarro, Average-optimal single and multiple approximate string matching, *ACM J. Exp. Algorithmics* 9 (1.4) (2004) 1–47. [doi:10.1145/1005813.1041513](#).
- [28] C. Barton, C. S. Iliopoulos, S. P. Pissis, Fast algorithms for approximate circular string matching, *Algorithms Mol. Biol.* 9 (2014) 9. [doi:10.1186/1748-7188-9-9](#).
- [29] M. A. R. Azim, C. S. Iliopoulos, M. S. Rahman, M. Samiruzzaman, A fast and lightweight filter-based algorithm for circular pattern matching, in: P. Baldi, W. Wang (Eds.), 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB 2014, ACM, 2014, pp. 621–622. [doi:10.1145/2649387.2660804](#).
- [30] M. A. R. Azim, C. S. Iliopoulos, M. S. Rahman, M. Samiruzzaman, A filter-based approach for approximate circular pattern matching, in: R. W. Harrison, Y. Li, I. I. Mandoiu (Eds.), *Bioinformatics Research and Applications, ISBRA 2015, Vol. 9096 of LNCS*, Springer, 2015, pp. 24–35. [doi:10.1007/978-3-319-19048-8_3](#).
- [31] C. Barton, C. S. Iliopoulos, S. P. Pissis, Average-case optimal approximate circular string matching, in: A. Dediu, E. Formenti, C. Martín-Vide, B. Truthe (Eds.), *Language and Automata Theory and Applications, LATA 2015, Vol. 8977 of LNCS*, Springer, 2015, pp. 85–96. [doi:10.1007/978-3-319-15579-1_6](#).
- [32] T. Hirvola, J. Tarhio, Bit-parallel approximate matching of circular strings with k mismatches, *ACM J. Exp. Algorithmics* 22. [doi:10.1145/3129536](#).
- [33] W. I. Chang, T. G. Marr, Approximate string matching and local similarity, in: M. Crochemore, D. Gusfield (Eds.), *Combinatorial Pattern Matching, CPM 1994, Vol. 807 of LNCS*, Springer, 1994, pp. 259–273. [doi:10.1007/3-540-58094-8_23](#).
- [34] M. Lothaire, [Applied Combinatorics on Words](#), Cambridge University Press, 2005.
URL http://www.cambridge.org/gb/knowledge/isbn/item1172552/?site_locale=en_GB
- [35] P. Charalampopoulos, T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, J. Straszypiński, T. Waleń, W. Zuba, Circular pattern matching with k mismatches, in: L. A. Gąsieniec, J. Jansson, C. Levcopoulos (Eds.), *Fundamentals of Computation Theory - 22nd International Symposium, FCT 2019, Vol. 11651 of Lecture Notes in Computer Science*, Springer, 2019, pp. 213–228. [doi:10.1007/978-3-030-25027-0_15](#).
- [36] N. J. Fine, H. S. Wilf, Uniqueness theorems for periodic functions, *Proceedings of the American Mathematical Society* 16 (1) (1965) 109–114. [doi:10.2307/2034009](#).
- [37] T. Kociumaka, J. Radoszewski, W. Rytter, T. Waleń, Internal pattern matching queries in a text and applications, in: P. Indyk (Ed.), 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM, 2015, pp. 532–551. [doi:10.1137/1.9781611973730.36](#).
- [38] T. Kociumaka, [Efficient data structures for internal queries in texts](#), Ph.D. thesis, University of Warsaw (Oct. 2018).
URL <https://www.mimuw.edu.pl/~kociumaka/files/phd.pdf>

- [39] T. Kociumaka, J. Radoszewski, W. Rytter, J. Straszyński, T. Waleń, W. Zuba, Efficient representation and counting of antipower factors in words, in: C. Martín-Vide, A. Okhotin, D. Shapira (Eds.), Language and Automata Theory and Applications, LATA 2019, Vol. 11417 of LNCS, Springer, 2019, pp. 421–433, full version at <https://arxiv.org/abs/1812.08101>. doi:10.1007/978-3-030-13435-8_31.
- [40] M. Pătraşcu, Unifying the landscape of cell-probe lower bounds, SIAM J. Comput. 40 (3) (2011) 827–847. doi:10.1137/09075336X.
- [41] M. L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with $\mathcal{O}(1)$ worst case access time, J. ACM 31 (3) (1984) 538–544. doi:10.1145/828.1884.
- [42] M. Ružić, Constructing efficient dictionaries in close to sorting time, in: L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfssdóttir, I. Walukiewicz (Eds.), Automata, Languages and Programming, ICALP 2008, Part I, Vol. 5125 of LNCS, Springer, 2008, pp. 84–95. doi:10.1007/978-3-540-70575-8_8.