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 Straszyński
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 $\mathcal{O}(n \log n \log\log n + \mathsf{output})$ expected time or $\mathcal{O}(n \log n \log^2 \log n / \log\log\log n + \mathsf{output})$ 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].

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].

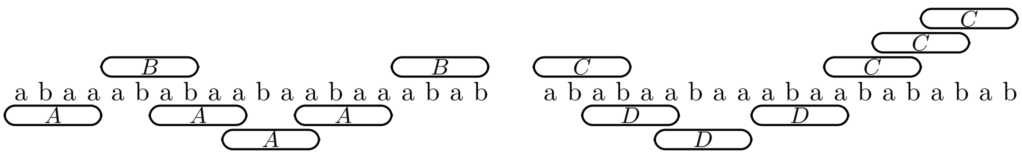

Figure 1 Two examples of a 2-cover of a string: a ps-cover (left) and a b-cover (right). Note that none of these strings has a proper cover.

In this paper, we present an $\mathcal{O}(n \log n \log\log n + \texttt{output})$ 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 $T = \texttt{a}^m\texttt{ba}^m\texttt{ba}^m$ 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 $\mathcal{O}(n \log\log n/\ell + \mathsf{output})$ expected time using predecessor queries.

Our algorithms easily generalize to the λ-covers problem, achieving $\mathcal{O}(n^{\lambda-1}\text{polylog}\, n + \mathsf{output})$ 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 $\{i, \ldots, j\}$; 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 $\mathit{intervals}_k(S)$ 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 $T[i] \ldots T[j]$ 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 $\mathit{Occ}_T(X)$ we denote the set of starting positions of occurrences of X in T and by $\mathit{Cov}_T(X)$ the set of positions that are covered by occurrences of X in T, i.e.,

$$\mathit{Cov}_T(X) = \bigcup \mathit{intervals}_{|X|}(\mathit{Occ}_T(X)).$$

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 $X \in \mathcal{S}$
- $\bigcup_{X \in \mathcal{S}} \mathit{Cov}(X) = [1\,..\,|T|]$

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 $p + q \le |S|$, then it has a period* $\gcd(p, q)$.

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.

▶ **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 $X = T[1\mathinner{.\,.}\ell]$ and $Y = T[n-\ell+1\mathinner{.\,.}n]$. We apply IPM queries to compute the sets of occurrences $\mathit{Occ}(X)$ and $\mathit{Occ}(Y)$, represented as unions of $\mathcal{O}(n/\ell)$ of arithmetic sequences, in $\mathcal{O}(n/\ell)$ time. This lets us compute the sets $\mathit{Cov}(X)$ and $\mathit{Cov}(Y)$, represented as unions of $\mathcal{O}(n/\ell)$ maximal intervals, sorted left-to-right. Then we need to check if $\mathit{Cov}(X)\cup\mathit{Cov}(Y) = [1\mathinner{.\,.}n]$, 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 $\mathit{Cov}(P_\ell)\cup\mathit{Cov}(S_\ell)$ 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 $\rhd_\ell$ of left endpoints of occurrences of P_ℓ
- the set $\lhd_\ell$ 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.

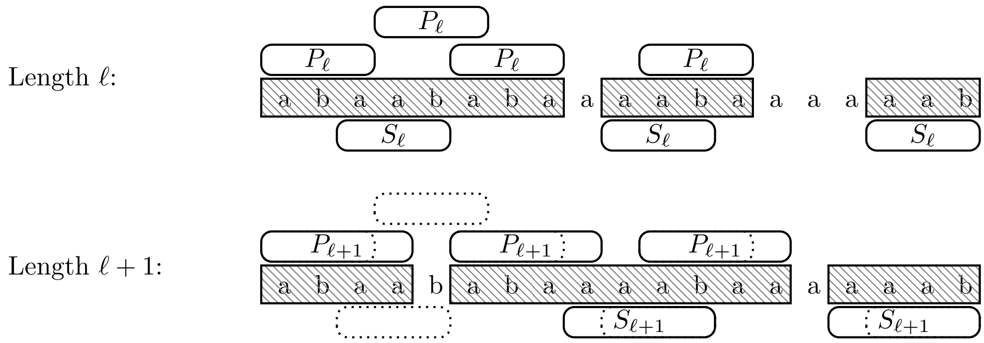

Figure 2 Illustration of gap dynamics. After incrementation of ℓ, a gap `b` was created, a gap `a` disappeared, and a gap `aaa` shrunk to `a`.

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 $i \in \rhd_\ell$ ($i \in \lhd_\ell$) persists as long as $\ell \leq \mathit{PREF}[i]$ ($\ell \leq \mathit{PREF}^R[i]$, respectively). Therefore, for $\ell = \mathit{PREF}[i]+1$ ($\ell = \mathit{PREF}^R[i]+1$), 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:

- $L_\rhd = \max\{a : a \in \rhd_\ell, a < p\}$ and $L_\lhd = \max\{a : a \in \lhd_\ell, a < p\}$
- $R_\rhd = \min\{b : b \in \rhd_\ell, b > p\}$ and $R_\lhd = \min\{b : b \in \lhd_\ell, b > p\}$.

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 $[i \mathinner{.\,.} j]$, 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 $\ell = \min(R_\lhd - i + 1, j - L_\rhd + 1)$, or

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

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

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

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 $\mathit{Cov}(P_\ell) \cup \mathit{Cov}(S_\ell)$, 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 $\mathit{Occ}(X)$, represented as a union of $\mathcal{O}(n/\ell)$ of arithmetic sequences, and the set $\mathit{Cov}(X)$, represented as a union of $\mathcal{O}(n/\ell)$ maximal intervals, in $\mathcal{O}(n/\ell)$ time. If $n - \ell + 1 \notin \mathit{Occ}(X)$, there is no b-cover of length ℓ, and if $\mathit{Cov}(X) = [1\,..\,n]$, 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 $\mathit{Cov}(Y)$, 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 $a \in [0\,..\,\ell)$.

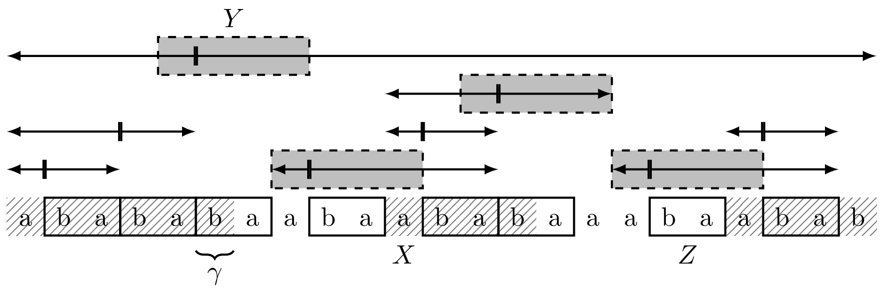

Figure 3 This string has a b-cover ($X = abab, Y = abaa$). The sets $Cov(X)$ and $Cov(Y)$ are shown in gray. We have $Z = ba$, $\Gamma = Occ_T(Z)$, and γ is the position of the occurrence of Z that ends at the first position that is not covered by $Cov(X)$. The set P_Y of occurrences of Y that is generated 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 $Z_1, \ldots, Z_t$ of T length $z = \lceil \ell/2 \rceil$ such that if (X, Y) is a b-cover, then Y contains at least one of $Z_1, \ldots, Z_t$ 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 $\lvert Y\rvert = \ell$.

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 $j \le i-z+1$, then it covers the factor Z_1. Otherwise, $j+\ell-1 \ge i+z$ and $j \le i$, 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., $4p \le \ell$, then p would also be a period of its factor Z and $2p \le z$, so Z would be periodic.

Assume now that Z is periodic. Let p be the smallest period of Z; we have $2p \le z$. Assume to the contrary that Y is 4-periodic with smallest period p' such that $p' \ne p$. We have $4p' \le \ell$, so $2p' \le z$. 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.*

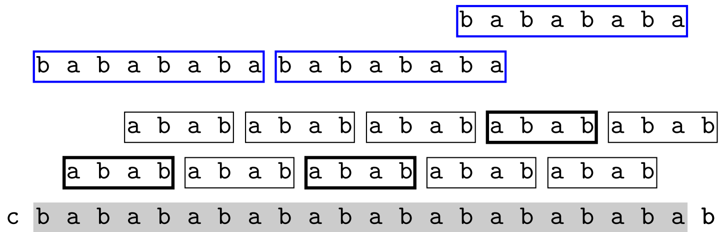

Figure 4 $Z = \texttt{abab}$ (black rectangles), $Y = \texttt{babababa}$ (blue rectangles); gray color shows $Cov(Y)$. The occurrences of Y that are 1-anchored at marked occurrences of Z are shown and cover $Cov(Y)$.

Proof. Let $p = \mathsf{per}(Z)$. We apply IPM queries to compute the set $Occ(Z)$, 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 $\Gamma = Occ(Z)$. 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 $Occ_Y(Z)$ is an arithmetic sequence with difference p and first element $t \in [1\,..\,p)$. Let $m = |Occ_Y(Z)|$; we have $2 \le m \le \ell/p$. 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 $T = \texttt{abababab}$, $Z = \texttt{abab}$ and $Y = \texttt{bababa}$.)

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 $\Gamma \subseteq Occ_T(Z)$, 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 $|Y| = \ell$.

▶ **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 $\mathcal{O}(n\tau_n/\ell + \mathsf{output})$ time and $\mathcal{O}(n/\ell)$ space.*

Proof. Let $A = Cov(X)$, $A' = [1\..n] \setminus A$, and $\mathcal{A} \subseteq [1\..n]^2$ be the set of maximal intervals of A. We have $|\mathcal{A}| \le n/\ell$ 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

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

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 $i \in Occ(Z)$ if and only if $a \le x$ and $|Z| \le \ell - a \le y$. See the arrows in Figure 3.

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

$$A \cup \bigcup intervals_\ell(S'_a \ominus a) = [1\..n], \tag{1}$$

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

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 $S'_a \ne S'_{a+1}$. 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 $i' = succ(i, D_S)$. 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 $i' - i \le \ell$ 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 $a \ge i' - j$ or $\ell - a \ge j' - i + 1$, where $j = succ(i, A')$ and $j' = pred(i' - 1, A')$.

(c) Otherwise, if there is an interval $[u\..v] \in \mathcal{A}$ such that $[u\..v] \subseteq [i\..i')$, then $a \ge i' - v - 1$ and $\ell - a \ge u - i$.

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,

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

When i is inserted to D_S, we remove the constraints $C(i', i'')$ imposed by the pair $i' = pred(i, D_S)$ and $i'' = succ(i, D_S)$ 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).

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 $\mathcal{O}(\mathsf{output})$ time. The complexity follows. ◁

This concludes the solution to Positioned Cover problem. ◀

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 $\mathcal{O}(n\tau_n/\ell + \mathsf{output})$ time.*

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

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 $\mathcal{O}(n \log n \log \log n + \mathsf{output})$ expected time or $\mathcal{O}(n \log n \log^2 \log n / \log \log \log n + \mathsf{output})$ 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 $\mathcal{O}(n^{\lambda-1} \log n \log \log n + \mathsf{output})$ expected time or $\mathcal{O}(n^{\lambda-1} \log n \log^2 \log n / \log \log \log n + \mathsf{output})$ 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 $\mathit{Cov}(X_1) \cup \mathit{Cov}(X_2)$, 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 $\bigcup \mathit{intervals}_\ell(B)$ represented as a union of maximal intervals. Finally, we merge this set with $\mathit{Cov}(X_1) \cup \mathit{Cov}(X_2)$ 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

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

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 $\mathcal{C} := \mathit{Cov}(X_1) \cup \dots \cup \mathit{Cov}(X_{\lambda-1})$ 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 $\mathit{Cov}_T(X)$. 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 $\mathit{Cov}_T(X)$. 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.iospress.com/articles/fundamenta-informaticae/fi84-1-04.

A Supplementary Figure

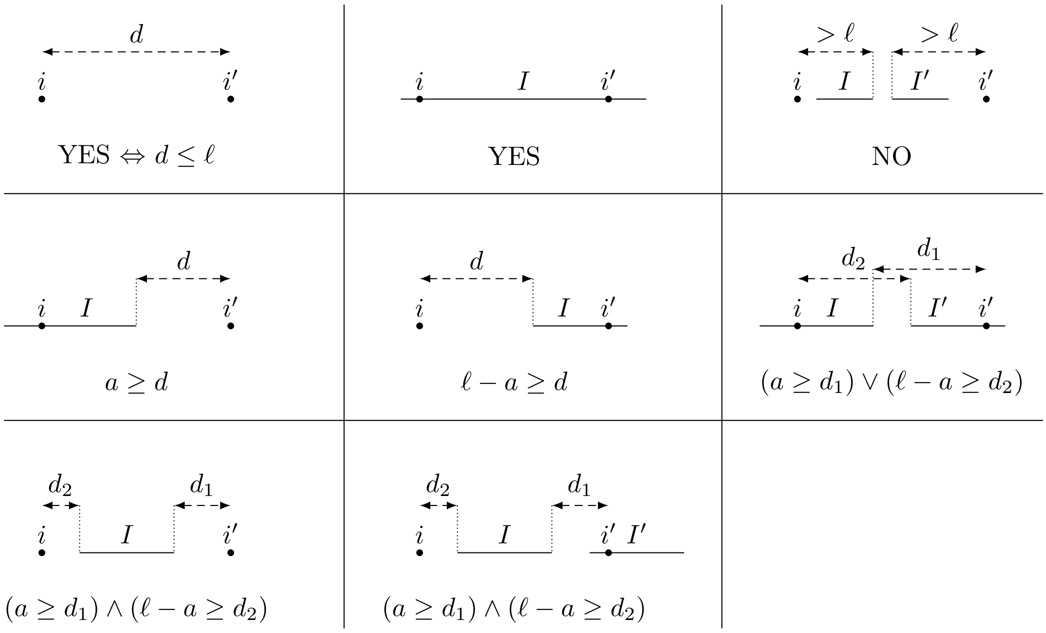

Figure 5 Sets of constraints $C(i, i')$ generated depending on the interactions with intervals $I, I' \in \mathcal{A}$. The respective rows correspond to items (a)–(c).