# Covers in Optimal Space

**Itai Boneh**
Reichman University, Herzliya, Israel
University of Haifa, Israel

**Shay Golan**
Reichman University, Herzliya, Israel
University of Haifa, Israel

## Abstract

A cover of a string $S$ is a string $C$ such that every index of $S$ is contained in some occurrence of $C$. First introduced by Apostolico and Ehrenfeucht [TCS'93] over 30 years ago, covers have since received significant attention in the string algorithms community. In this work, we present a space-efficient algorithm for computing a compact representation of all covers of a given string. Our algorithm requires only $O(\log n)$ additional memory while accessing the input string of length $n$ in a read-only manner. Moreover, it runs in $O(n)$ time, matching the best-known time complexity for this problem while achieving an exponential improvement in space usage.

**2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms

**Keywords and phrases** Cover, Read-only random access, small space

**Digital Object Identifier** 10.4230/LIPIcs.CPM.2025.5

**Funding** This research was supported by Israel Science Foundation grant 810/21.

## 1 Introduction

A *cover* $C$ of a string $S$ is a substring of $S$ such that every index in $S$ is covered by some occurrence of $C$. The definition of covers was first introduced by Apostolico and Ehrenfeucht [2], where they also present an $O(n \log^2 n)$ time algorithm that reports all maximal substrings of an input string $S$ of length $n$ that have a non-trivial cover. In particular, the algorithm of [2] decides whether $S$ has a non-trivial cover. In 1991, Apostolico, Farach and Iliopoulos [3] introduce the first $O(n)$-time algorithm that computes the shortest cover of $S$. Breslauer [8] generalizes the result and shows how to compute the shortest cover for every prefix of $S$ in linear time. Finally, Moore and Smyth [28, 29] introduce an $O(n)$ time algorithm that returns all covers of $S$. In a very recent work, Radoszewski and Zuba [32] show how to obtain a sub-linear time algorithm when the string is over a sub polynomial alphabet and is given in packed representation.

The time complexity of computing all covers of a string has been thoroughly studied, with optimal algorithms well established. This motivates us to shift our focus to space complexity, where significant questions remain. Our goal is to explore the fundamental limits of space efficiency and develop algorithms that minimize memory usage while maintaining optimal running time. In particular, we consider the classical model of *read-only random access model*. In this model, random access to the input is allowed, and the goal is to design a well-performing algorithm that uses a small, typically sub-linear, amount of working space. The study of space efficient string algorithms in the read-only random access model has given rise to many interesting results throughout the years. A partial list of fundamental string problems that have been studied in the read-only random access model include Pattern Matching [15, 13], Lempel-Ziv Factorization [20], Longest Increasing Substring [21], Longest Common Extension Data Structure [6, 25], Longest Common Substring [5] and Internal Pattern Matching [4].


In this work, we study space-efficient computation of all the covers of an input string $S$ of length $n$ over a polynomial alphabet, given in read-only memory. We introduce an algorithm that computes an $O(\log n)$ size representation of all covers of $S$ using $O(\log n)$ space. The running time of our algorithm is $O(n)$, matching the running time of the state of the art algorithm for polynomial alphabet [29], which inherently uses linear space. As a subroutine, we also develop an algorithm with the same complexities for computing all the borders of an input string. Our main result is stated in the following theorem.

▶ **Theorem 1.** *There exists an algorithm that given a string $S$ of length $n$ in read-only memory, outputs a representation of* $\mathsf{Covers}(S)$ *by $O(\log n)$ arithmetic progressions. The algorithm runs in $O(n)$ time and uses $O(\log n)$ working space.*

In Section 6 we justify the $O(\log n)$ space complexity by showing that any representation of the covers of a strings requires $\Omega(\log n)$ machine words for some string of length $n$ for a sufficiently large $n$.

**Related work.** Covers have been investigated in many computational settings and numerous variations have been introduced (for a comprehensive recent survey, see Mhaskar and Smyth [27]). As previously mentioned, the best algorithm for computing all covers of an input string over general alphabet is by Moore and Smyth [28, 29], who presented an $O(n)$ time algorithm. Li and Smyth [26] consider the more general problem of computing the longest cover of each prefix of $S$, providing an $O(n)$ time algorithm for this problem. Crochemore et al. [12] considered the problem of constructing an efficient data structure reporting the shortest cover or all covers of a query substring. That is, they show how to preprocess a string in $O(n \log n)$ time to construct an $O(n \log n)$-space data structure that can report the shortest cover of an input substring in $O(\log n \log \log n)$ time.

The study of covers has led to the introduction of numerous variations and generalizations. A 2-cover of $S$ ([18, 33]) is a pair $(X, Y)$ of strings, such that every index in $S$ is covered either by an occurrence of $X$ or by an occurrence of $Y$. This can be generalized for a $\lambda$-cover, which is a set of $\lambda$ strings that, together, cover every index in $S$. While computing all $\lambda$-covers of a string is NP-complete for general $\lambda$ ([11]), efficient algorithms have been recently introduced for computing 2-covers ([31, 7]). Charalampopoulos et al. [10] introduced natural notions of covers in 2-dimensional strings, and provided efficient algorithms to compute all such covers of an input 2-dimensional string. Other variations of covers that were introduced and studied are enhanced covers [14], approximate covers [1], Cyclic Covers [16, 19], and Tree Covers [30, 24].

## 2 Preliminaries

**Integer Notations.** We use bracket notation to denote consecutive sets of integers, i.e. for two integers $i, j$ we denote $[i..j] = \{i, i+1, \ldots, j\}$. For a positive integer $n$, we denote $[n] = [1..n]$. An arithmetic progression of integers is a set defined by a triplet of the first element, the difference between elements and the number of elements. We denote it by $\mathsf{AP}(\ell_{\min}, p, m) = \{\ell_{\min} + k \cdot p \mid k \in [0, m-1]\}$.

**Strings.** A string $S = S[1]S[2]S[3]\ldots S[n]$ of length $|S| = n$ is a sequence of symbols over some alphabet $\Sigma$. For $i \leq j$, $S[i..j] = S[i]S[i+1]\ldots S[j]$ is a *substring* of $S$. If $i = 1$ it is a *prefix*, and if $j = n$ it is a *suffix*. A string that appears in $S$ both as a prefix and as a suffix is called a *border*. A positive integer $p$ is called a *period* of $S$ if $S[i] = S[i+p]$ for every $i \in [1..n-p]$. The minimal period of $S$ is called *the period* of $S$. We say that $S$ is *periodic*, if the period of $P$ is at most $|S|/2$.

For two strings $S$ and $P$ of lengths $n$ and $m$, we say that $P$ *occurs* at index $i \in [1..n-m+1]$ of $S$ if $P = S[i..i+m-1]$. We call the index $i$ an *occurrence* of $P$ in $S$. For an index $i \in S$, we say that $i$ is covered by a string $C$ of length $c$ if there is an occurrence of $C$ in $[i-c+1..i]$. If every index in $[1..n]$ in $S$ is covered by $C$, we say that $C$ is a *cover* of $S$.

Each prefix $S[1..\ell]$ can be uniquely represented by its length $\ell$. We say that $\ell$ is a border-length if $S[1..\ell]$ is a border of $S$ and cover-length if $S[1..\ell]$ is a cover of $S$. $\mathsf{Borders}(S) = \{\ell \mid S[1..\ell] \text{ is a border}\}$ similarly $\mathsf{Covers}(S) = \{\ell \mid S[1..\ell] \text{ is a cover}\}$. Since every cover must cover index 1, it must be a prefix of $S$, and since it must cover index $n$, it must also be a suffix of $S$. Therefore, every cover is a border, i.e $\mathsf{Covers}(S) \subseteq \mathsf{Borders}(S)$.

**Read-Only Random Access Model.** We assume that the input string is given in a read-only format, meaning the algorithm can query $S[i]$ in constant time but cannot modify the string. In this model, the input string does not occupy $O(n)$ space explicitly, allowing for algorithms with sublinear space complexity. Therefore, the space complexity of the algorithm consists of the working space of the algorithm and of the space dedicated to writing the output. Our algorithm operates in the word RAM model under the standard assumption that both an index from $[n]$ and a symbol from $\Sigma$ fit within a single machine word.

As a subroutine, our algorithm uses a pattern matching algorithm that uses $O(1)$ space [15]. The algorithm is formally stated in the following lemma.

▶ **Lemma 2.** *There exists an algorithm* PM *that, given a text $T$ and a pattern $P$ stored in read-only memory, reports all occurrences of $P$ in $T$ sequentially in $O(|T|+|P|)$ time while using only $O(1)$ working space. Moreover, the algorithm is online, meaning that it processes each character of $T$ in $O(1)$ time and determines whether the suffix of $T$ up to that character forms an occurrence of $P$.*

**Useful facts.** Here are several known facts that are useful in our algorithms.

▶ **Fact 3** ([9, see Lemma 3.1])**.** *Let $T$ and $P$ be two strings. If $|T| \leq 2|P|$ then all the occurrences of $P$ in $T$ form an arithmetic progression. Moreover, if there are more than two occurrences of $P$ in $T$ then the difference of the arithmetic progression is the period of $P$.*

▶ **Fact 4** (Folklore)**.** *Let $T$ and $P$ be two strings and let $p$ be the period of $P$. Let $i \neq j$ be two different occurrences of $P$ in $T$, then we must have $|i-j| \geq p$.*

▶ **Fact 5** ([28, Lemma 2])**.** *Let $S$ be a string with cover $C_1$. Then, every string $C_2$ with $|C_2| < |C_1|$ is a cover of $S$ if and only if $C_2$ is a cover of $C_1$.*

▶ **Fact 6** ([8, Fact 1.3])**.** *Let $S$ be a string with a border $B$ and let $C$ be a cover of $S$ with $|C| < |B|$. Then, $C$ is a cover of $B$.*

▶ **Fact 7** (cf. [7, Lemma 8])**.** *Let $S$ be a string with a cover $S[1..\ell]$ such that $S[1..\ell]$ has a period length $p \leq \frac{\ell}{2}$. Then, $S[1..\ell-p]$ is also a cover of $S$.*

## 3 Reporting All Borders

In this section we present an algorithm that computes a representation of $\mathsf{Borders}(S)$ in $O(n)$ time, using $O(\log n)$ space. It is well known that $\mathsf{Borders}(S)$ can be represented by $O(\log n)$ arithmetic progressions [22, 17, 23], and we follow this idea. However, we focus on a special arithmetic progression which we call *generating arithmetic progression* of border-lengths.

▶ **Definition 8.** *Let $S$ be a string. We say that $\mathsf{AP}(\ell_{\min}, p, m) \subseteq \mathsf{Borders}(S)$ is a generating arithmetic progression if $\ell_{\min} \geq p$ and for every $\ell \in L \setminus \{\ell_{\min}\}$ the period length of $S[1..\ell]$ is $p$.*

The following lemma introduces a partition of all borders into $O(\log n)$ generating arithmetic progressions and an optimal algorithm that computes these generating arithmetic progressions in $O(n)$ time, using only $O(\log n)$ working space.

▶ **Lemma 9.** *Let $S$ be a string. The interval $[1, n]$ can be partitioned into $O(\log n)$ sub-intervals $I_1, I_2, \ldots, I_{O(\log n)}$, and let $L_i = \mathsf{Borders}(S) \cap I_i$ such that the following properties hold:*

1. *For every $i$, the set $L_i$ is a generating arithmetic progression and we denote $L_i = \mathsf{AP}(\ell^i_{\min}, p_i, m_i)$.*
2. *Let $\ell^i_{\min}$ and $\ell^j_{\min}$ be the minimum elements in two different intervals such that $\ell^i_{\min} < \ell^j_{\min}$. Then, $\ell^i_{\min} \leq \frac{3}{4}\ell^j_{\min}$.*

*Moreover, there exists an algorithm that given $S$ in read-only memory outputs the partition and the arithmetic progressions. The algorithm takes $O(n)$ time and uses $O(\log n)$ working space.*

In order to prove Lemma 9 we first consider a sub-interval of $[1..n]$ of the form $[k..2k-1]$ and show that all border-lengths in such an interval forms a single generating arithmetic progression. Moreover, this arithmetic progression can be found in linear time and constant working space.

▶ **Lemma 10.** *There exists an algorithm that, given a string $S$ of length $n$ stored in read-only memory and an integer $k \leq n$, outputs a triplet $(\ell_{\min}, p, m)$ such that $L = \mathsf{AP}(\ell_{\min}, p, m) = \mathsf{Borders}(S) \cap [k, 2k-1]$, or $\mathsf{null}$ if $\mathsf{Borders}(S) \cap [k, 2k-1] = \emptyset$. Moreover, $\mathsf{AP}(\ell_{\min}, p, m)$ is a generating arithmetic progression. The algorithm runs in $O(k)$ time and uses $O(1)$ working space.*

**Proof.** For every $\ell \in \mathsf{Borders}(S) \cap [k, 2k-1]$ the border $S[1..\ell]$ starts with $S[1..k]$ and implies an occurrence of $S[1..k]$ at position $n - \ell + 1$. Thus, the algorithm starts by finding all the occurrences of $P = S[1..k]$ in $T = S[n - 2k + 2..n]$, using Lemma 2. If there are no such occurrences, we conclude that $\mathsf{Borders}(S) \cap [k, 2k-1]$ is empty. Otherwise, we distinguish between two cases. If there is a single occurrence of $P$ in $T$, the algorithm is trying to extend this occurrence to a border. Namely, if $S[1..k] = S[n - \ell + 1..n - \ell + k]$, the algorithm checks whether $S[1..\ell] = S[n - \ell + 1..n]$ by straightforward characters comparisons. Notice that since in this case there is a single occurrence, the time required for all those comparisons is $O(k)$. In this case, the algorithm outputs $\ell_{\min} = \ell$, $p = 1$ and $m = 1$ if it discovers that $S[1..\ell]$ is a border, or returns $\mathsf{null}$ otherwise.

In case where there are multiple occurrences of $S[1..k]$, we exploit periodicity as follows. If there are at least two occurrences of $P$ in $T$, all the occurrences form an arithmetic progression. This claim is obvious if there are exactly two occurrences and follows from Fact 3 if there are at least three occurrences. This allows us to represent all the occurrences reported by Lemma 2 in $O(1)$ space (creating the arithmetic progression when the second occurrence is reported and extending the arithmetic progression with every additional occurrence). Denote by $L' = \{\ell'_{\min} + c \cdot p \mid c \in [0, m-1]\}$ the set of indices $\ell$ such that $P$ occurs in $n - \ell + 1$. Due to the occurrences of $P$, it must be that the period length of $S[n - \ell'_{\min} - (m-1) \cdot p + 1..S - \ell'_{\min} + k]$ is exactly $p$. The algorithm finds where this period breaks both in the prefix and in the suffix, as follows. In the prefix, let $v_1 \in [k+1, 2k-1]$ be the smallest index where $S[v_1] \neq S[v_1 - p]$, if such $v_1$ does not exists $v_1 = 2k+1$. For the

suffix, recall that $S[n-\ell_{\min}-(m-1)\cdot p+1..n-\ell_{\min}+k]$ is periodic with period $p$. The algorithm finds the smallest index $v_2 > n-\ell_2+k$ such that $S[v_2] \neq S[v_2-p]$, if such an index does not exist, $v_2=-1$.

If $v_2=-1$, then every $\ell$ such that $S[n-\ell+1..n-\ell+k]=P$ is a border of $S$ if and only if $\ell < v_1$. Thus, the set of borders is an arithmetic progression, which can be computed in constant time from $L'$ (it is $L' \cap [1..v_1-1]$). Notice that in this case, we have indeed that $p$ is the period length of every element except maybe for $\ell_{\min}$. Clearly we also have $p < k \leq \ell_{\min}$. Thus, we indeed obtain a generating arithmetic progression.

If $v_2 \neq -1$ and $v_1 = 2k+1$, there are no borders whose length is some $\ell \in [k, 2k-1]$ because for any $\ell \in [k, 2k-1]$ the prefix of length $\ell$ has period-length $p$, while the suffix of length $\ell$ does not have period-length $p$.

Finally, if $v_2 \neq -1$ this means that $p$ is not a period of $S[n-\ell'_{\min}+1..n]$. In this case, if $v_1 \neq 2k+1$, we have only one candidate which is $\ell = v_1 + (n-v_2)$. This is because $\ell$ is only length allowing the position of the first violation of the period $p$ to be both at offset $v_1$ from the beginning and at offset $n-v_2$ from the end of the border.

If $\ell \geq 2k$ the algorithm concludes that there is no border whose length is in $[k, 2k-1]$. Otherwise, the algorithm checks whether $\ell$ is indeed a border, by performing $\ell = O(k)$ comparisons. To see why $\ell$ is the only candidate for a border, notice that $\ell$ is the only position where the first violation of period $p$ of the prefix and suffix match each other.

The running time of the algorithm is $O(k)$ (both for the pattern matching algorithm of Lemma 2, for verifying $O(1)$ candidates and for finding $v_1$ and $v_2$). The space usage of the algorithm is $O(1)$. ◀

We are now ready to prove Lemma 9.

**Proof of Lemma 9.** To obtain Lemma 9 we consider a partition of the border-lengths into exponential intervals of the form $[2^i, 2^{i+1}-1]$. We are using the algorithm of Lemma 10 for every $k$ which is a power of 2, from $2^0$ to $2^{\lfloor \log n \rfloor}$ to obtain $O(\log n)$ arithmetic progressions. It only remains to prove that the third property of Lemma 9 holds. That is, if $\ell^i_{\min}$ and $\ell^j_{\min}$ are the minimum elements in two different intervals such that $\ell^i_{\min} < \ell^j_{\min}$ then, $\ell^i_{\min} \leq \frac{3}{4}\ell^j_{\min}$. If $\ell^i_{\min} \in [k, 2k-1]$ and $\ell^j_{\min} \in [k', 2k'-1]$ such that $k' > 2k$ the claim follows immediately. Thus, we need to prove that the claim holds for two consecutive intervals $\ell^i_{\min} \in [k, 2k-1]$ and $\ell^j_{\min} \in [2k, 4k-1]$. Let $x = \ell^i_{\min}$ and $y = \ell^j_{\min}$. Assume to the contrary that $x > \frac{3}{4}y$. Let $p = y - x < \frac{1}{4}y$ since $S[1..x]$ is a border of $S[1..y]$ we have that $S[1..y]$ has period $p$. Since $S[1..x]$ is a prefix of $S[1..y]$, $p$ is also a period length of $S[1..x]$ and it holds that $p < \frac{x}{3}$. By Fact 7, it must be that $S[1..x-p] = S[1..y-2p]$ is also a cover of $S$. However, since $y \in [2k..4k-1]$ and $p < \frac{y}{4}$ we get that $y - 2p > y - \frac{y}{2} = \frac{y}{2}$ which implies that $y-2p$ is in $[k, 2k-1]$ and is smaller than $x = y-p$. This contradicts $x$ being minimum in $\mathsf{Borders}(S) \cap [k, 2k-1]$. Finally, the time complexity of the algorithm is $\sum_{i=1}^{\log n} O(2^i) = O(2^{2\log n}) = O(n)$. The algorithm uses $O(1)$ space per each call to Lemma 10 and reuses the space on each call, for a total of $O(1)$ space. Storing the intervals and the arithmetic progression representation of $L_i$'s requires $O(\log n)$ space. ◀

## 4 Warm Up - Reporting All Covers in $O(n \log n)$ Time

As a warm up, we show how to report all covers in $O(n \log n)$ time, which we later improve in Section 5 to an $O(n)$-time algorithm. The main idea of our algorithm is to process separately each arithmetic progression of border-lengths reported by Lemma 9. We will show that for a given arithmetic progression $L$ of border-lengths, with minimum length $\ell_{\min}$, difference $p$

and $m$ elements, the set of **cover-lengths** among those borders is a (possibly empty) prefix of the set, i.e. it is either an empty set or an arithmetic progression starting with $\ell_{\min}$, with difference $p$ and $m_c \leq m$ elements i.e., $L^{cover} = \mathsf{Covers}(S) \cap L = \mathsf{AP}(\ell_{\min}, p, m_c)$. Moreover, we will show that we can compute $m_c$, the number of elements in $L^{cover}$, in $O(n)$ time. We first show that if there exists some cover-length $\ell$ in the set $L$, then every smaller $\ell'$ in the $L$ is also a cover-length.

▶ **Lemma 11.** *Consider a generating arithmetic progression $L = \mathsf{AP}(\ell_{\min}, p, m)$ and let $\ell \in L$. If $S[1..\ell]$ is a cover of $S$ then for every $\ell' \in L$ with $\ell' < \ell$ we have $S[1..\ell']$ is also a cover of $S$.*

**Proof.** Recall that $L$ is a generating arithmetic progression, for every $\ell' \in L$ we have $\ell' \geq p$ and that $p$ is the period length of $S[1..\ell]$. By Fact 5 it is enough to show that $S[1..\ell']$ covers $S[1..\ell]$. By definition of arithmetic progression, there exists two non-negative integers $m_1 < m_2$ such that $\ell' = \ell_{\min} + m_1 \cdot p$ and $\ell = \ell_{\min} + m_2 \cdot p$. It is easy to see that for every $i < m - 1$ (where $m$ is the number of elements in the arithmetic progression) $S[1..\ell_{\min} + i \cdot p]$ is a cover of $S[1..\ell_{\min} + (i+1) \cdot p]$. This is because by $p$ being the period of $S[1..\ell_{\min} + (i+1) \cdot p]$, we have that $S[1..\ell_{\min} + i \cdot p]$ is a border of $S[1..\ell_{\min} + (i+1) \cdot p]$ and that $\ell_{\min} + i \cdot p \geq \ell_{\min}/2 + p/2 + i \cdot p \geq (\ell_{\min} + (i+1)p)/2$. Therefore, every index of $S[1..\ell_{\min}+(i+1)\cdot p]$ is covered either by the prefix, or by the suffix occurrence of $S[1..\ell_{\min}+i\cdot p]$. Thus, by a simple induction we get that $S[1..\ell']$ covers $S[1..\ell]$, as required. ◀

The following corollary follows immediately from Lemma 11.

▶ **Corollary 12.** *Let $L = \mathsf{AP}(\ell_{\min}, p, m) \subseteq \mathsf{Borders}(S)$ be a generating arithmetic progression. The set of border-lengths from $L$ that their corresponding borders covers $S$ is $L^{cover} = L \cap \mathsf{Covers}(S) = \mathsf{AP}(\ell_{\min}, p, m_c)$ for some integer $m_c \in [0, m]$.*

Thus, to find all the cover-lengths in a given arithmetic progression, the algorithm first verifies whether $S[1..\ell_{\min}]$ covers $S$, which implies whether or not $L^{cover} \neq \emptyset$. In the following simple lemma, we show that such a verification can be done in $O(n)$ time using $O(1)$ working space.

▶ **Lemma 13.** *Given a length $\ell$ there exists an algorithm that decides whether $S[1..\ell]$ covers $S$ in $O(n)$ time, using $O(1)$ working space.*

**Proof.** The algorithm uses $\mathsf{PM}$ - the pattern matching algorithm of Lemma 2 with $S$ as the text and $P = S[1..\ell]$ as the pattern. Recall, that this algorithm is a real-time algorithm that outputs all the occurrences of $P$ in $S$ from left to right. At any moment, the algorithm maintains the position of the last occurrence. If at some point there are $|P|$ consecutive characters without any occurrence of $P$ or if $P$ is not a suffix of $S$, the algorithm halts and reports that $S[1..\ell]$ does not cover $S$. Otherwise, the algorithm reports $S[1..\ell]$ is a cover of $S$. The complexities of the algorithm are dominated by the algorithm of Lemma 2, and are as stated. The correctness of the algorithm follows from the definition of a cover. ◀

In case $S[1..\ell_{\min}]$ is indeed a cover of $S$, it remains to find $m_c$ - the number of elements in $L^{cover} = L \cap \mathsf{Covers}(S)$, the arithmetic progression of covers (see Corollary 12). In the following lemma, we show how to do it in $O(n)$ time and $O(1)$ working space.

▶ **Lemma 14.** *There exists an algorithm such that its input is a string $S$ in read-only memory and a generating arithmetic progression $L = \mathsf{AP}(\ell_{\min}, p, m)$. The algorithm outputs $m_c$ such that the border-lengths of $L$ which are also cover-lengths are exactly $L^{cover} = L \cap \mathsf{Covers}(S) = \mathsf{AP}(\ell_{\min}, p, m_c)$. The algorithm uses $O(1)$ space and runs in $O(n)$ time.*

**Proof.** The algorithm first determines whether $S[1..\ell_{\min}]$ and $P = S[1..\ell_{\min}+p]$ cover $S$, using Lemma 13. If one of them does not cover $S$, the answer is simple by Corollary 12 (it is $m_c = 0$ if $S[1..\ell_{\min}]$ is not a cover and $m_c = 1$ if it is a cover and $P$ is not a cover). The algorithm finds all occurrences of $P$ in $S$ one after the other. The algorithm partitions the occurrences into maximal (non-overlapping) arithmetic progressions with a difference of exactly $p$, and maintains the shortest maximal arithmetic progression of occurrences. Let *minOccs* denote the number of occurrences in the shortest arithmetic progression. Then, the algorithm returns $\min\{minOccs+1, m\}$ as $m_c$. See Algorithm 1.

**Algorithm 1** **Compute_$\mathbf{m_c}$**$(S, \ell_{\min}, p, m)$.

**Input** : A string $S$ in read-only memory, and a generating arithmetic progression $\mathsf{AP}(\ell_{\min}, p, m)$.
**Output** : $m_c$.

```
Verify that S[1..ℓ_min] is a cover (using Lemma 13);
if S[1..ℓ_min] is not a cover then
    return 0
Verify that P = S[1..ℓ_min + p] is a cover (Lemma 13);
if P is not a cover then
    return 1
lastOcc ← 0; current ← 0; minOccs ← +∞;
Initialize PM with P = S[1..ℓ_min + p];
for i = 1 to n do
    isOccur ← process S[i] with PM;
    if isOccur = True then
        if i − lastOcc = p then
            current ← current + 1;
        else
            minOccs ← min{minOccs, current};
            current ← 1;
        lastOcc ← i;
minOccs ← min{minOccs, current};
return min{minOccs + 1, m};
```

**Complexities.** Since PM uses $O(1)$ space and takes $O(1)$ time per character, the algorithm uses $O(1)$ working space and runs in $O(n)$ time.

**Correctness.** If $S[1..\ell_{\min}]$ is not a cover of $S$, clearly, $m_c = 0$, and the algorithm reports the correct answer. Similarly, if $S[1..\ell_{\min}]$ is a cover and $S[1..\ell_{\min}+p]$ is not a cover then by Lemma 11 $m_c = 1$ and the algorithm reports the right value.

Let us consider the case where $S[1..\ell_{\min}+p]$ is a cover of $S$. Let $m' = \min(minOccs+1, m)$ and let $\ell' = \ell_{\min} + (m'-1)\cdot p$. We have to prove that $m_c = m'$. Let $\ell^* = \ell_{\min} + (m_c - 1)\cdot p$ be the maximum element in $L$ such that $S[1..\ell^*]$ covers $S$.

Since $L$ is a generating arithmetic progression, the period of $P = S[1, \ell_{\min}+p]$ is $p$. Therefore, due to Fact 4, when the algorithm find that the next occurrence is not at difference $p$ from the previous one, it is at difference strictly more than $p$. Algorithm 1 essentially

iterates these arithmetic progressions, and keeps track on the shortest arithmetic progression encountered throughout the iteration as $minOccs$. It follows that there is an index $i'$ such that:

1. For every $t \in [0..minOccs - 1]$ there is an occurrence of $P$ at index $i' + p \cdot t$.
2. Both $i' - p$ and $i' + p \cdot (minOccs)$ are not occurrences of $P$.

We start by showing that $minOccs+1 \geq m_c$. Assume to the contrary that $minOccs+1 < m_c$. Since $S[1..\ell^*]$ is a cover, it must hold that an occurrence of $S[1..\ell^*]$ covers the index $i'+p-1$, say at index $j'$. Since $j'$ is in particular an occurrence of $P$, and $i'$ is also an occurrence of $P$, Fact 4 implies that either $j' = i'$ or $j' \notin [i' - p + 1..i' + p - 1]$. If $j' \leq i' - p$, then, we have that $S[i' - p..i' + p - 1]$ has period $p$, which together with the fact that $S[i'..i' + |P| - 1]$ has period $p$ implies that $S[i' - p..i' + |P| - p - 1] = S[i'..i' + |P| - 1]$, a contradiction to $i' - p$ not being an occurrence of $P$. If $i' = j'$, we have that there are $m_c - 1 > minOccs$ consecutive occurrences of $P$ following $i'$, a contradiction to $i' + minOccs \cdot p$ not being an occurrence of $P$.

We proceed to show that $S[1..\ell']$ is a cover of $S$, which implies that $m' \leq m_c$. Let $i \in [n]$ be an index. Since $P$ is a cover of $S$, $i$ is covered by some occurrence of $P$ at index $j$. Let $c'$ be the value of $current$ after Algorithm 1 iterated the occurrence $j$. By the definition of $current$, we know that there is an occurrence of $P$ at index $j - p \cdot t$ for every $t \in [0..c' - 1]$. We also know that throughout the following $m' - 1 - c'$ iterations of the algorithm, the value of $current$ will increase to at least $m' - 1$ before being reset to 1. Therefore, there are occurrences of $P$ in $j + p \cdot c'$ for every $p \in [0, m' - 1 - c']$ as well. In conclusion, for $j' = j - (c' - 1) \cdot p$, there are occurrences of $P$ in $j' + p \cdot t$ for every $t \in [0, m' - 1]$ which means that there is an occurrence of $S[1..\ell']$ at index $j'$ containing all of these occurrences. In particular, this occurrence of $S[1..\ell']$ covers the occurrence of $P$ that covers $i$, which means that $S[1..\ell']$ covers $i$. ◀

To conclude this section, one can use Lemmas 9 and 14 to output all covers of $S$ in $O(n \log n)$ time and $O(\log n)$ space, by first obtaining the arithmetic progressions from Lemma 9 and then apply Lemma 14 on each one of them. In Section 5 we will show how to reduce the running time of finding all covers to $O(n)$ while preserving $O(\log n)$ working space.

## 5 Linear Time Algorithm

In this section, we introduce an improved algorithm that computes all covers of $S$ with $O(\log n)$ space and takes only $O(n)$ time.

**Sequence of First elements.** Let $L_1, L_2, \ldots, L_t$ be the arithmetic progressions that Lemma 9 outputs. Their union forms $\mathsf{Borders}(S)$, ordered by increasing value of first element. Let $\ell_1, \ell_2, \ldots, \ell_t$ be the sequence of elements, such that $\ell_i$ is the first element in $L_i$. In addition, let $\ell_{t+1} = n$ be the length of $S$. Recall that by Lemma 9 we have $\ell_i < \frac{3}{4}\ell_{i+1}$ for every $1 \leq i < t$ and that $t = O(\log n)$. We denote $F = F_S = (\ell_1, \ell_2, \ldots, \ell_{t+1})$.

We introduce a recursive algorithm that finds for a given $i$ the largest $j < i$ such that $S[1..\ell_j]$ is a cover of $S[1..\ell_i]$. In particular, when applying the algorithm with $i = t + 1$ the output is the largest $j$ such that $S[1..\ell_j]$ covers $S[1..\ell_{t+1}] = S[1..n] = S$. Later in Section 5.1 we will use this lemma to find $F \cap \mathsf{Covers}(S)$ and report all covers-lengths of $S$.

▶ **Lemma 15.** *There exists an algorithm that given $S$ in read-only memory and the sequence $F$, for a given $i$ computes the largest $j < i$ such that $S[1..\ell_j]$ is a proper cover of $S[1..\ell_i]$, or* null *if there is no such $j$. The algorithm runs in $O(\ell_i)$ time and uses $O(i)$ space.*

**Proof.** (See Algorithm 2.) We first consider the base case, where $i = 1$, in this case there is no value $j < i$, and the algorithm returns null.

Now, we consider the general case where $i > 1$. The algorithm runs in iterations, checking a candidate $j$ in each iteration (starting from $j = i - 1$) to determine whether $S[1..\ell_j]$ covers $S[1..\ell_i]$. At the end of each iteration, the algorithm either finds that $S[1..\ell_j]$ indeed covers $S[1..\ell_i]$, or identifies the largest prefix $S[1..x]$ of $S[1..\ell_i]$ that is covered by $S[1..\ell_j]$. The next candidate is the largest $j' < j$ such that $S[1..\ell_{j'}]$ covers $S[1..\ell_j]$, which is retrieved via a recursive call. In the next iteration, the algorithm does not need to verify that $S[1..\ell_{j'}]$ covers $S[1..x]$, since it follows from the fact that $S[1..\ell_{j'}]$ covers $S[1..\ell_j]$ and $S[1..\ell_j]$ covers $S[1..x]$ by Fact 5. Thus, the algorithm proceeds to the next iteration, starting to verify that $S[1..\ell_{j'}]$ covers $S[1..\ell_i]$ from position $x - \ell_{j'} + 1$. We note that the verification starts at $x - \ell_{j'} + 1$ and not at position $x + 1$, since position $x + 1$ can be covered by occurrences of $S[1..\ell_j]$ in the interval $[x - \ell_{j'} + 2..x + 1]$. If the verification fails, we are again in a situation where a maximal prefix covered by $S[1..\ell_{j'}]$ was found.

In each iteration, to determine the largest prefix covered by the current candidate $S[1..\ell_j]$, the algorithm employs the PM algorithm from Lemma 2. We utilize the fact that PM is a real-time algorithm to halt the algorithm the moment we recognize the largest prefix covered by the current candidate border. This way, the running time of the algorithm is linear in the length of the candidate and the progress we made in covering $S$.

**Algorithm 2** **max_j_cover**$(S, F, i)$.

**Input** : A string $S$ in read-only memory, an increasing sequence $F \subseteq \mathsf{Borders}(S)$ and an index $i \in [1, |F|]$
**Output** : The largest $j < i$ such that $S[1..\ell_j]$ covers $S[1..\ell_i]$

1 **if** $i = 1$ **then**
  `// Base case: If i = 1, there is no previous cover, return null`
2  **return** null;
3 $j \leftarrow i - 1$; $x \leftarrow \ell_j$;
4 **while** *true* **do**
5  Run PM$(S[x - \ell_j + 1..\ell_i], S[1..\ell_j])$ until reaching $\ell_j$ consecutive indices without an occurrence;
  `// update x to store largest prefix covered by` $\ell_j$
6  $x \leftarrow$ end of the last found occurrence;
7  **if** $x = \ell_i$ **then**
    `// If the entire prefix is covered, return j`
8  **return** $j$;
9  $j \leftarrow$ **max_j_cover**$(S, F, j)$;
10  **if** $j =$ null **then**
    `// If no valid j is found, return null`
11  **return** null;

**Correctness.** We prove correctness by induction on $i$. The base case $i = 1$ follows immediately. Assume that the algorithm is correct for all $j < i$, and we prove that it is also correct for $i$. We consider the iterations of the while loop (Line 4) and show that at the beginning of each iteration, the following property holds.

▷ Claim 16. At the beginning of every iteration, $S[1..x]$ is covered by $S[1..\ell_j]$.

Proof. We prove the claim by induction on the iterations. At the beginning of the first iteration, $x = \ell_j$ implies $S[1..x] = S[1..\ell_j]$ is trivially covered by $S[1..\ell_j]$. Assuming the property holds at the beginning of some iteration, we should prove it holds at the end of the iteration as well. Let $j_{\mathsf{before}}, x_{\mathsf{before}}$ and $j_{\mathsf{after}}, x_{\mathsf{after}}$ be the values of $j$ and $x$, at the beginning and end of the iteration, respectively. During the iteration, $x_{\mathsf{after}}$ is set such that $S[x_{\mathsf{before}} - \ell_{j_{\mathsf{before}}} + 1..x_{\mathsf{after}}]$ is the largest prefix of $S[x_{\mathsf{before}} - \ell_{j_{\mathsf{before}}} + 1..\ell_i]$ covered by $S[1..\ell_{j_{\mathsf{before}}}]$. By the induction hypothesis $S[1..\ell_{j_{\mathsf{before}}}]$ also covers $S[1..x_{\mathsf{before}}]$. Thus, $S[1..x_{\mathsf{after}}]$ is covered by $S[1..\ell_{j_{\mathsf{before}}}]$. Finally $j_{\mathsf{after}} = \mathbf{max\_j\_cover}(S, F, j_{\mathsf{before}})$, implying (by induction hypothesis regarding the correctness of $\mathbf{max\_j\_cover}(j)$) that $S[1..j_{\mathsf{after}}]$ also covers $S[1..x_{\mathsf{after}}]$, by Fact 5, as required. ◁

Due to Claim 16, at the beginning of each iteration $S[1..\ell_j]$ covers $S[1..x]$. By running the pattern matching algorithm of Lemma 2 from position $x - \ell_j + 1$ we guarantee that all occurrences of $S[1..\ell_j]$ that can be used to cover position $x+1$ are found. Thus, at Line 6, the algorithm assigns to $x$ the largest prefix of $S[1..\ell_i]$ that is covered by $S[1..\ell_j]$. In particular, for the value $j$ reported by the algorithm $S[1..\ell_j]$ indeed covers $S[1..\ell_i]$.

On the other hand, let $j^* < i$ be the largest integer such that $S[1..\ell_{j^*}]$ covers $S[1..\ell_i]$. By Fact 6, $S[1..\ell_{j^*}]$ covers all borders $S[1..\ell_i]$ for $i > j$. Therefore, the algorithm at Line 9 never assigns to $j$ values smaller than $j^*$ (and also not the value $\mathsf{null}$). Thus, the answer returned by the algorithm is never $\mathsf{null}$ and never smaller than $j^*$.

**Space complexity.** The space usage of the algorithm in every level of the recursion is clearly $O(1)$. Every recursive call has different integer $i = O(\log n)$. Thus, there are $O(\log n)$ levels of recursion, and the total space usage of the algorithm is $O(\log n)$ space.

**Time complexity.** We prove by induction that the time complexity is $O(\ell_i)$. For $i = 1$ the algorithm runs in $O(1) \subseteq O(\ell_1)$ time. Let us assume that the running time for every $j < i$ is $O(\ell_j)$, and we prove that the running time for $i$ is $O(\ell_i)$. Let $j_k$ and $x_k$ be the values of $j$ and $x$, respectively at the beginning of the $k$th iteration (Line 4), and let $x_{k+1}$ be the value of $x_k$ at the end of the $k$th iteration. The runtime of Line 5 is $O((x_{k+1} - x_k + \ell_{j_k}) + \ell_{j_k}) = O((x_{k+1} - x_k) + \ell_{j_k})$. The runtime of Line 9 is $O(\ell_j)$ by the induction hypothesis. All other lines of the loop take $O(1)$ time. Thus, the $k$th iteration costs $O((x_{k+1} - x_k) + \ell_{j_k})$ time. Summing over all iterations, the term $x_{k+1} - x_k$ sums up to $O(\ell_i)$ and the sum of $\ell_{j_k}$ is bounded by $O(\sum_{j=1}^{i-1} \ell_j) = O(\ell_i)$, since for every $1 \leq j < t \leq i-1$ we have $\ell_j \leq \frac{3}{4}\ell_{j+1}$. Thus, the algorithm runs in $O(\ell_i)$ time, as required. ◀

## 5.1 Reporting All Covers

Recall that $F = (\ell_1, \ell_2, \ldots, \ell_t, \ell_{t+1})$ is the sequence of all the first elements in the arithmetic progressions of Lemma 9, appended with $n = |S|$. The algorithm that reports all covers has two phases. In the first phase, the algorithm finds the subset of $F$ of border-lengths which are also cover-lengths, denoted as $F^{cover} = F \cap \mathsf{Covers}(S)$. Then, the algorithm computes for every $j$ with $\ell_j \in F^{cover}$ the arithmetic progression $L_j^{cover} = L_j \cap \mathsf{Covers}(S)$, where $L_j$ is the arithmetic progression of borders with minimum element $\ell_j$. A naïve implementation of this approach would cost $O(n \log n)$ time, since the the computation of $L_j^{cover}$ with the algorithm of Lemma 14 takes $O(n)$ time per arithmetic progression. The key idea for improvement, is that by Fact 5 if $\ell_{j_1}$ and $\ell_{j_2}$ are two consecutive cover-lengths in $F^{cover}$, to extend $S[1..\ell_{j_1}]$

it is enough to make sure the extension covers $S[1..\ell_{j_2}]$, and we do not need to compute the extension with respect to the complete string $S$. Thus, the costs for each computation of $L_j^{cover}$ is proportional to the successor cover in $F^{cover}$, which implies a geometric sequence of costs that sums up to $O(n)$ time.

▶ **Lemma 17.** *There exists an algorithm that given $S$ in read-only memory and $F$, computes $F^{cover}$ in $O(n)$ time, using $O(\log n)$ space.*

**Proof.** The algorithm starts by finding $\max F^{cover}$, the maximum cover-length in $F$ by Lemma 15 with $i = t + 1$. Then, as long as the last found value $j$ is not null, the algorithm uses Lemma 15 with $i = j$ to find the longest cover-length of $F$ which is smaller than $i$.

**Algorithm 3** Compute $F^{cover}(S, F)$.

**Input:** A string $S$ in read-only memory and the sequence $F$
**Output:** $F^{cover}$

```
1 j ← |F|, F^cover ← ∅;
2 while j ≠ null do
3 |   j ← max_j_cover(S, F, j);
4 |   Append ℓ_j to F^cover;
5 return F^cover;
```

**Correctness.** In the first iteration of the loop the algorithm clearly finds the maximum length $\ell_j \in F$ such that $S[1..\ell_j]$ covers $S[1..\ell_{t+1}] = S$, hence finds $\max F^{cover}$. In general, in the $i$th iteration, the algorithm appends the largest (among the covers in $F$) cover of the cover found in the $(i-1)$th iteration. By Fact 5, this is exactly the next largest cover of $S$ in $F$. Thus, at the end the algorithm returns all cover-lengths of $S$ in $F$, that is $F^{cover}$.

**Comlexities.** The space usage of the algorithm is dominated by the sizes of $F$ and $F^{cover}$, which are both $O(\log n)$ space. The time for every iteration of the loop is $O(\ell_j)$ by Lemma 15. Thus, in total the running time $O(n + \sum_{j \in F^{cover}} \ell_j) = O(n + \sum_{j \in F} \ell_j) = O(n + \sum_{j=1}^{\log n} (\frac{3}{4})^j n) = O(n)$. ◀

Finally, we are ready to prove Theorem 1 by combining Lemmas 9, 14, and 17.

▶ **Theorem 1.** *There exists an algorithm that given a string $S$ of length $n$ in read-only memory, outputs a representation of $\mathsf{Covers}(S)$ by $O(\log n)$ arithmetic progressions. The algorithm runs in $O(n)$ time and uses $O(\log n)$ working space.*

**Proof.** The algorithm first applies Lemma 9 and computes $F$ to be the increasing sequence of first elements in the arithmetic progressions. Then, the algorithm applies Lemma 17 to obtain $F^{cover} = f_1 < f_2 < \ldots < f_z$ (and let us artificially denote $f_{z+1} = n$). For every $j \in [z]$, the algorithm uses Lemma 14 with the string $S[1..f_{j+1}]$ and the arithmetic progression $\mathsf{AP}(\ell_j, p_j, m_j)$ of $f_j$ to find $m_j^{cover}$ such that $L_j \cap \mathsf{Covers}(S) = \mathsf{AP}(\ell_j, p_j, m_j^{cover})$. The algorithm returns all arithmetic progressions found in this process.

**Correctness.** First, it is easy to see that every length $\ell$ reported by the algorithm is indeed a cover-length of $S$. For $\ell \in F^{cover}$ it follows from the correctness of Lemma 17 For $\ell' \notin F^{cover}$ it follows from Fact 5 that $L_j$ is a generating arithmetic progression with respect to $S[1..f_{j+1}]$. Therefore, for $\ell' \notin F^{cover}$ the correctness follows from Lemmas 14 and 17.

On the other hand let $\ell \in \mathsf{Covers}(S)$, clearly $\ell \in \mathsf{Borders}(S)$ and by Lemma 9 there exists some $i$ such that $\ell \in \mathsf{AP}(\ell^i_{\min}, p_i, m_i)$. By the correctness of Lemma 17 it must be that $\ell^i_{\min} \in F^{cover}$ and by the correctness of Lemma 14 it must be that $\ell \in \mathsf{AP}(\ell^i_{\min}, p_i, m_i^{cover})$

**Complexities.** Since each of the algorithms in Lemmas 9, 14, and 17 runs in $O(n)$ time and uses $O(\log n)$ space this bounds hold for the algorithm, as required. ◀

## 6 Lower Bound on Representation Size of Covers($S$)

To complement our result we establish the claim that any representation of the set of cover lengths of a string of length $n$ must take $\Omega(\log n)$ words of space, that is $\Omega(\log^2 n)$ bits.

▶ **Lemma 18.** *For an integer $n$, let $C_n = \{\mathsf{Covers}(S) \mid S \in \{a,b\}^{\leq n}\}$. Then $\log |C_n| = \Omega(\log^2 n)$.*

Lemma 18 indicates that any algorithm using $o(\log n)$ machine words (i.e. $o(\log^2 n)$ bits) on inputs with length $n$, necessarily returns the same output for some pair of strings $S_1$,$S_2 \in \{a,b\}^{\leq n}$ with $\mathsf{Covers}(S_1) \neq \mathsf{Covers}(S_2)$, for a sufficiently large $n$. Hence, the algorithm of Theorem 1 is optimal in this model.

**Proof.** Let $\mathcal{A} = [1..\sqrt{n}]^{\frac{\log n}{10}}$ be set of arrays of size $t = \frac{\log n}{10}$ with each entry being a number in $[1..\sqrt{n}]$. Clearly, $\log |\mathcal{A}| = \Theta(\log^2 n)$. We prove the statement by showing an injection from $\mathcal{A}$ to $C_n$.

Let $A = (a_1, a_2, \dots a_t) \in \mathcal{A}$. We construct a string $S_A$ corresponding to $A$ as follows. $S_0 = a^{\sqrt{n}} b a^{\sqrt{n}}$. For every $i \in [1..t]$ we define $S_i = S_{i-1} \cdot S_{i-1}[a_i..|S_{i-1}|]$. Finally, we define $S_A = S_t$.

By the construction for every $i \in [1..t]$ we have $|S_i| \leq 2|S_{i-1}|$. Since $|S_0| = 2\sqrt{n}+1$ and $t = \frac{\log n}{10}$ we get by simple induction that $|S_A| = |S_t| \leq 2^{\frac{\log n}{10}} |S_0| \leq n^{\frac{1}{10}} \sqrt{n} \leq n$.

Notice that for every $i \in [0..t]$, the string $S_i$ starts and ends with $a^{\sqrt{n}}$. It follows that $S_{i-1}$ is both a prefix and a suffix of $S_i$ of length at least $|S_i|/2$. As a consequence, $S_{i-1}$ is a cover of $S_i$. Therefore, by Fact 5 we have that for every $i \in [0,t]$ we have $|S_i| \in \mathsf{Covers}(S)$. We next show how to recover all lengths of $|S_i|$s from $\mathsf{Covers}(S)$.

▷ Claim 19. For every $i \in [0, t-1]$ $\mathsf{Covers}(S_A) \cap [2|S_i| - \sqrt{n} + 1..2|S_i|] = \{|S_{i+1}|\}$.

Proof. Recall that $|S_{i+1}| = 2|S_i| - a_i + 1 \in [2|S_i| - \sqrt{n} + 1..2|S_i|]$ and $S_{i+1}$ covers $S$. Assume by contradiction that there is another cover length $\ell' \in \mathsf{Covers}(S_A) \cap [2|S_i| - \sqrt{n} + 1..2|S_i|]$. It follows that one of $\ell'$, $|S_{i+1}|$ is a border of the other with length difference at most $\sqrt{n} - 1$ from each other. It is well known [8, Fact 1.1] that a string $T$ with border-length $x$ has period length $|T| - x$, which indicates that $S_{i+1}$ is periodic with period less than $\sqrt{n}$. This is a contradiction as the prefix $S_0$ of $S_{i+1}$ has $b = S_0[\sqrt{n}+1] \neq S_0[\sqrt{n}+1+p] = a$ for any $p < \sqrt{n}$. ◁

We now show that the mapping of $A$ to $\mathsf{Covers}(S_A)$ is indeed an injection. Let $A$ and $A'$ be two different arrays in $\mathcal{A}$. Let $i$ be the smallest index where $a_i \neq a'_i$. It is clear by the construction that $|S_{i-1}| = |S'_{i-1}|$ and therefore $|S_i| \neq |S'_i|$. Since both $|S_i|$ and $|S'_i|$ are in $[2|S_i| - \sqrt{n} + 1..2|S_i|]$, by Claim 19 it must be that $\mathsf{Covers}(A) \neq \mathsf{Covers}(A')$. ◀

## References


1 Amihood Amir, Avivit Levy, Ronit Lubin, and Ely Porat. Approximate cover of strings. *Theor. Comput. Sci.*, 793:59–69, 2019. doi:10.1016/J.TCS.2019.05.020.

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

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

4 Gabriel Bathie, Panagiotis Charalampopoulos, and Tatiana Starikovskaya. Internal pattern matching in small space and applications. In Shunsuke Inenaga and Simon J. Puglisi, editors, *35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, June 25-27, 2024, Fukuoka, Japan*, volume 296 of *LIPIcs*, pages 4:1–4:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CPM.2024.4.

5 Stav Ben-Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. Time-space tradeoffs for finding a long common substring. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CPM.2020.5.

6 Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 607–626. SIAM, 2020. doi:10.1137/1.9781611975994.37.

7 Itai Boneh, Shay Golan, and Arseny M. Shur. String 2-covers with no length restrictions. In Timothy M. Chan, Johannes Fischer, John Iacono, and Grzegorz Herman, editors, *32nd Annual European Symposium on Algorithms, ESA 2024, September 2-4, 2024, Royal Holloway, London, United Kingdom*, volume 308 of *LIPIcs*, pages 31:1–31:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ESA.2024.31.

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

9 Dany Breslauer and Zvi Galil. Real-time streaming string-matching. In Raffaele Giancarlo and Giovanni Manzini, editors, *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings*, volume 6661 of *Lecture Notes in Computer Science*, pages 162–172. Springer, 2011. doi:10.1007/978-3-642-21458-5_15.

10 Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Computing covers of 2d-strings. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 12:1–12:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.12.

11 Richard Cole, Costas S. Iliopoulos, Manal Mohamed, William F. Smyth, and Lu Yang. The complexity of the minimum k-cover problem. *J. Autom. Lang. Comb.*, 10(5/6):641–653, 2005. doi:10.25596/JALC-2005-641.

12 Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Internal quasiperiod queries. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2020. doi:10.1007/978-3-030-59212-7_5.

13 Maxime Crochemore and Dominique Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991. doi:10.1145/116825.116845.

14 Tomás Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, W. F. Smyth, and Wojciech Tyczynski. Enhanced string covering. *Theor. Comput. Sci.*, 506:102–114, 2013. doi:10.1016/J.TCS.2013.08.013.

15 Zvi Galil and Joel I. Seiferas. Time-space-optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983. doi:10.1016/0022-0000(83)90002-8.

16 Roberto Grossi, Costas S. Iliopoulos, Jesper Jansson, Zara Lim, Wing-Kin Sung, and Wiktor Zuba. Finding the cyclic covers of a string. In Chun-Cheng Lin, Bertrand M. T. Lin, and Giuseppe Liotta, editors, *WALCOM: Algorithms and Computation - 17th International Conference and Workshops, WALCOM 2023, Hsinchu, Taiwan, March 22-24, 2023, Proceedings*, volume 13973 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2023. doi:10.1007/978-3-031-27051-2_13.

17 Leo J Guibas and Andrew M Odlyzko. Periods in strings. *Journal of Combinatorial Theory, Series A*, 30(1):19–42, 1981. doi:10.1016/0097-3165(81)90038-8.

18 Qing Guo, Hui Zhang, and Costas S. Iliopoulos. Computing the $\lambda$-covers of a string. *Inf. Sci.*, 177(19):3957–3967, 2007. doi:10.1016/J.INS.2007.02.020.

19 Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Linear-time computation of cyclic roots and cyclic covers of a string. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, June 26-28, 2023, Marne-la-Vallée, France*, volume 259 of *LIPIcs*, pages 15:1–15:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CPM.2023.15.

20 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight lempel-ziv parsing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2013. doi:10.1007/978-3-642-38527-8_14.

21 Masashi Kiyomi, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, and Jun Tarui. Space-efficient algorithms for longest increasing subsequence. In Rolf Niedermeier and Brigitte Vallée, editors, *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 28 to March 3, 2018, Caen, France*, volume 96 of *LIPIcs*, pages 44:1–44:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.STACS.2018.44.

22 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

23 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. *SIAM J. Comput.*, 53(5):1524–1577, 2024. doi:10.1137/23M1567618.

24 Lukasz Kondraciuk. String covers of a tree revisited. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 297–309. Springer, 2023. doi:10.1007/978-3-031-43980-3_24.

25 Dmitry Kosolobov and Nikita Sivukhin. Construction of sparse suffix trees and LCE indexes in optimal time and space. In Shunsuke Inenaga and Simon J. Puglisi, editors, *35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, June 25-27, 2024, Fukuoka, Japan*, volume 296 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CPM.2024.20.

26 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.

27 Neerja Mhaskar and W. F. Smyth. String covering: A survey. *Fundam. Informaticae*, 190(1):17–45, 2022. doi:10.3233/FI-222164.

28 Dennis W. G. Moore and William F. Smyth. Computing the covers of a string in linear time. In Daniel Dominic Sleator, editor, *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms. 23-25 January 1994, Arlington, Virginia, USA*, pages 511–515. ACM/SIAM, 1994. URL: http://dl.acm.org/citation.cfm?id=314464.314636.

**29** Dennis W. G. Moore and William F. Smyth. A correction to "an optimal algorithm to compute all the covers of a string". *Inf. Process. Lett.*, 54(2):101–103, 1995. doi:10.1016/0020-0190(94)00235-Q.

**30** Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. String covers of a tree. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 68–82. Springer, 2021. doi:10.1007/978-3-030-86692-1_7.

**31** Jakub Radoszewski and Juliusz Straszynski. Efficient computation of 2-covers of a string. In Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 77:1–77:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.ESA.2020.77.

**32** Jakub Radoszewski and Wiktor Zuba. Computing string covers in sublinear time. In Zsuzsanna Lipták, Edleno Silva de Moura, Karina Figueroa, and Ricardo Baeza-Yates, editors, *String Processing and Information Retrieval - 31st International Symposium, SPIRE 2024, Puerto Vallarta, Mexico, September 23-25, 2024, Proceedings*, volume 14899 of *Lecture Notes in Computer Science*, pages 272–288. Springer, 2024. doi:10.1007/978-3-031-72200-4_21.

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