Fundamenta Informaticae XXI (2018) 1–16
DOI 10.3233/FI-2016-0000
IOS Press

Frequency Covers for Strings

Neerja Mhaskar [C]
Dept. of Computing and Software, McMaster University, Canada
pophlin@mcmaster.ca

W. F. Smyth[*]
Dept. of Computing and Software, McMaster University, Canada
School of Engineering & Information Technology, Murdoch University, Perth, WA, Australia
smyth@mcmaster.ca

Abstract. We study a central problem of string processing: the compact representation of a string by its frequently-occurring substrings. In this paper we propose an effective, easily-computed form of quasi-periodicity in strings, the *frequency cover*; that is, the longest of those repeating substrings $\boldsymbol{u}$ of $\boldsymbol{w}$, $|\boldsymbol{u}| > 1$, that occurs the maximum number of times in $\boldsymbol{w}$. The advantage of this generalization is that it is not only applicable to all strings but also that it is the only generalized notion of cover yet proposed, which can be computed efficiently in linear time and space. We describe a simple data structure called the *repeating substring frequency* array ($\mathcal{RSF}$ array) for the string $\boldsymbol{w}$, which we show can be constructed in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space, where $|\boldsymbol{w}| = n$. We then use $\mathcal{RSF}$ to compute all the frequency covers of $\boldsymbol{w}$ in linear time and space. Our research also allows us to give an alternate algorithm to compute all non-extendible repeating substrings in $\boldsymbol{w}$, also in $\mathcal{O}(n)$ time and space.

Keywords: quasi-periodicities, covers, non-extendible repeating substring.

[C]Corresponding author
[*]Research supported in part by an NSERC Discovery Grant.

1. Introduction

In 1990 the idea of a "cover" or "quasiperiodicity" was introduced in [1] and later published in [2]: a *cover* of a string $\boldsymbol{w} = \boldsymbol{w}[1..n]$ is a proper substring $\boldsymbol{u}$ of $\boldsymbol{w}$ such that $\boldsymbol{w}$ can be constructed from possibly overlapping instances of $\boldsymbol{u}$. For example, the string $\boldsymbol{w} = ababaaba$ has a cover $\boldsymbol{u} = aba$. Although strings having covers are succinctly defined by their covers, such strings are rare. To generalize the notion of covers an enhanced cover of a string was proposed in [3]. An *enhanced cover* of $\boldsymbol{w} = \boldsymbol{w}[1..n]$ is a non-empty border of $\boldsymbol{w}$ that covers the maximum number of positions possible by any border of $\boldsymbol{w}$. For example, $\boldsymbol{w} = abaaabaabaaaaba$ has an enhanced cover $\boldsymbol{u} = aba$ covering a maximum of 12 positions. Several linear-time algorithms have been proposed to compute covers [4], [5], [6], [7] (updated in [8]), and enhanced covers [3], [8]. However, the expected maximum length of any border of any prefix of a string is approximately 1.64 for binary alphabets, and this number decreases with larger alphabet sizes [8]. Therefore, given a random string it is unusual for a string to have a non-empty border, and therefore to have an enhanced cover.

Given the need for a generalized notion of covers which is applicable to all strings, in [9] the α-partial cover was proposed. An α-partial cover of a string $\boldsymbol{w}$, $\alpha \leq n$, is the shortest substring $\boldsymbol{u}$ of $\boldsymbol{w}$ covering at least α positions in $\boldsymbol{w}$. For example, given the string $\boldsymbol{w} = ccabaacabaacabaaaba$ and $\alpha = 15$, $cabaa$ is the 15-partial cover of $\boldsymbol{w}$ as it covers $\alpha = 15$ positions in $\boldsymbol{w}$. The same paper [9] also shows that computing α-partial covers for all values of α can be done in $\mathcal{O}(n \log n)$ time using annotated suffix trees, thereby avoiding the need to guess an appropriate α value. In this paper we introduce a notion of the cover of a string which is based on its frequently occurring substrings (Section 3). An advantage of this definition is that it is not only applicable to all strings, but can be computed efficiently using simple data structures in linear time and space. In fact it is the only generalized notion of cover that is computable in linear time and space.

The outline of the paper is as follows: in Section 2 we introduce the terminology. Next, in Section 3, we define the main concept of the paper, the frequency cover, and the repeating substring frequency array ($\mathcal{RSF}$) data structure to compute it. We also give a linear time algorithm to compute $\mathcal{RSF}$, and use it to compute the frequency cover(s) in linear time and space. In Section 4 we summarize the results of the experiments conducted on many sample strings to compute their frequency covers, and the percentage of positions covered by them. Finally in section 5 we give a linear algorithm using the inverse $\mathcal{RSF}$ array ($\mathcal{IRSF}$) to compute all non-extendible repeating substrings in the given string.

2. Definitions

A string is a finite array $\boldsymbol{w}[1..n]$ whose entries $\boldsymbol{w}[i]$, $1 \leq i \leq n$, are drawn from a finite set of totally ordered symbols Σ, called an *alphabet*. The *length* of $\boldsymbol{w}$ is written $|\boldsymbol{w}| = n$. A string $\boldsymbol{w}[i..j]$ is a *substring* of $\boldsymbol{w}[1..n]$ if $1 \leq i \leq j \leq n$. A substring $\boldsymbol{w}[i..j]$ is a *proper substring* of $\boldsymbol{w}$ if $j - i + 1 < n$. A *prefix* of $\boldsymbol{w}$ is a substring $\boldsymbol{w}[i..j]$, where $i = 1$. A *suffix* of $\boldsymbol{w}$ is a substring $\boldsymbol{w}[i..j]$, where $j = n$. The suffix starting at index i is called *suffix* i. A *suffix array* $SA_{\boldsymbol{w}}$ of $\boldsymbol{w}$ is an integer array of length n, where $\mathcal{SA}_{\boldsymbol{w}}[i]$ is the starting position of the i-th lexicographically least suffix in $\boldsymbol{w}$. The *longest common prefix array* $\mathcal{LCP}_{\boldsymbol{w}}$ of string $\boldsymbol{w}$ is an integer array of length n, where $\mathcal{LCP}[i]$, $1 < i \leq n$, is the length of the

longest common prefix of suffixes $\mathcal{SA}_{\boldsymbol{w}}[i-1]$ and $\mathcal{SA}_{\boldsymbol{w}}[i]$. We assume that $\mathcal{LCP}_{\boldsymbol{w}}[1] = 0$. The *inverse suffix array* $\mathcal{ISA}_{\boldsymbol{w}}$ of $\boldsymbol{w}$ is an integer array of length n, where $\mathcal{ISA}_{\boldsymbol{w}}[i]$ is the lexicographic ranking of suffix i in $\boldsymbol{w}$. The *inverse longest common prefix array* $\mathcal{ILCP}_{\boldsymbol{w}}$ of a string $\boldsymbol{w}$ is an integer array of length n, where $\mathcal{ILCP}_{\boldsymbol{w}}[i] = \mathcal{LCP}_{\boldsymbol{w}}[\mathcal{ISA}_{\boldsymbol{w}}[i]]$, $1 \leq i \leq n$. We use $\mathcal{SA}, \mathcal{LCP}, \mathcal{ISA}$ and $\mathcal{ILCP}$ (without the subscript $\boldsymbol{w}$) when there is no ambiguity. Similarly for other arrays defined below.

We define the *frequency* $f_{\boldsymbol{w},\boldsymbol{u}}$ of a non-empty substring $\boldsymbol{u}$ in a string $\boldsymbol{w}$, to be the number of times the substring $\boldsymbol{u}$ occurs in the string $\boldsymbol{w}$. For example, in the string $\boldsymbol{w} = abababa$, the substrings ab, ba and aba all occur three times in $\boldsymbol{w}$, and their frequencies are $f_{\boldsymbol{w},ab} = 3$, $f_{\boldsymbol{w},ba} = 3$ and $f_{\boldsymbol{w},aba} = 3$, respectively. A *repeating substring* $\boldsymbol{u}$ in a string $\boldsymbol{w}$ is a substring $\boldsymbol{u}$ of $\boldsymbol{w}$ that occurs more than once; that is, $f_{\boldsymbol{w},\boldsymbol{u}} \geq 2$. A repeating substring of $\boldsymbol{w}$ is left (right) extendible if every instance of its occurrence in $\boldsymbol{w}$ is preceded (followed) by the same symbol, otherwise it is non-left (non-right) extendible, NLE (NRE) for short. For example, in the string $\boldsymbol{w} = ccabaacabaacabaaab$, the repeating string aba is both left and right extendible because its every occurrence is preceded by c and followed by a. A repeating substring of $\boldsymbol{w}$ is said to be *non-extendible* (NE) if and only if it is not left or right extendible. For example in the string $\boldsymbol{w} = ccabaacabaacabaaab$, the repeating substring $cabaa$ is non-extendible. In this paper, whenever we speak of a repeating substring, we mean an NRE repeating substring.

3. Frequency Cover in Strings

We now define the main concept of this paper. A *frequency cover* of $\boldsymbol{w}$ is the longest of those repeating substrings $\boldsymbol{u}$ of $\boldsymbol{w}$, $|\boldsymbol{u}| > 1$, that occurs the maximum number of times in $\boldsymbol{w}$.

For example in $\boldsymbol{w} = abababa$, the substring aba is the frequency cover of $\boldsymbol{w}$, occurring three times, as do the shorter substrings ab and ba. Note that a frequency cover is not unique. Consider the string $\boldsymbol{w} = ababcdcd$. It has two frequency covers ab and cd. Moreover, not all strings have a frequency cover. For example $\boldsymbol{w} = abcdefgh$ does not have a frequency cover. Note that we require the frequency covers of a string to be of length greater than one, as computing frequencies of each distinct letter in the string is quick and easy, at least on an ordered alphabet of reasonable size (simply scan the string from left to right and count the number of occurrences of each distinct letter).

Our definition of frequency cover requires that the repeating substring be longest; that is, we are interested in the longest frequency cover. We can define the shortest frequency cover as the shortest of those repeating substrings $\boldsymbol{u}$ of $\boldsymbol{w}$, $|\boldsymbol{u}| > 1$, that occurs the maximum number of times in $\boldsymbol{w}$. However, from Lemma 3.1 the longest frequency cover always covers more number of positions than the shortest frequency cover. Since we are interested in covers covering more number of positions, we are always interested in the longest frequency cover.

Lemma 3.1. Suppose $\boldsymbol{x}$ and $\boldsymbol{y}$ are the longest and shortest frequency covers of $\boldsymbol{w}$ respectively. Then $\boldsymbol{x}$ always covers more positions in $\boldsymbol{w}$ than $\boldsymbol{y}$ does.

Proof:
Since both $\boldsymbol{x}$ and $\boldsymbol{y}$ are frequency covers, $f_{\boldsymbol{w},\boldsymbol{x}} = f_{\boldsymbol{w},\boldsymbol{y}}$. Observe that the shortest frequency cover $\boldsymbol{y}$ will always be of size two; that is, $|\boldsymbol{y}| = 2$. For if $|\boldsymbol{y}| > 2$, any substring of $\boldsymbol{y}$ of length two would have the same frequency as that of $\boldsymbol{y}$ in $\boldsymbol{w}$ and be shorter than $\boldsymbol{y}$, thus contradicting the assumption that $\boldsymbol{y}$ is the shortest frequency cover.

For $\boldsymbol{x}$ to cover fewer positions than $\boldsymbol{y}$ does, some occurrences of $\boldsymbol{x}$ in $\boldsymbol{w}$ must overlap. Note that the overlap between any two instances of $\boldsymbol{x}$ cannot be greater than $\lfloor \frac{\boldsymbol{x}}{2} \rfloor$ as it would create a repetition in $\boldsymbol{x}$ which leads to $\boldsymbol{x}$ not being the frequency cover – a contradiction. Therefore, $\boldsymbol{x} = \boldsymbol{v}a\boldsymbol{v}$ (where a is a symbol). Additionally a is non-empty as otherwise it would create a repetition in $\boldsymbol{x}$ which leads to $\boldsymbol{x}$ not being the frequency cover – a contradiction. If $|\boldsymbol{v}| > 1$, then $\boldsymbol{v}$ would be the frequency cover and not $\boldsymbol{x}$. Therefore, $|\boldsymbol{x}| = 3$. Note that the least positions covered by $\boldsymbol{x}$ is when all occurrences of $\boldsymbol{x}$ in $\boldsymbol{w}$ overlap. However, assuming this worst case, $\boldsymbol{x}$, where $|\boldsymbol{x}| = 3$, still covers one more position in $\boldsymbol{w}$ than $\boldsymbol{y}$ does. Therefore, it is not possible for a shortest frequency cover to cover more positions than the positions covered by the longest frequency cover. □

3.1. Computing the $\mathcal{RSF}$ array

We propose a new data structure in order to efficiently compute frequency covers:

Definition 3.2. ($\mathcal{RSF}$ array)
The **"repeating substring frequency array"** $\mathcal{RSF}$ of $\boldsymbol{w}$ is an integer array of length n, where $\mathcal{RSF}[i]$ is the frequency of the repeating substring of length $\mathcal{LCP}[i]$ starting at index $\mathcal{SA}[i]$ in $\boldsymbol{w}$; that is, the repeating substring $\boldsymbol{w}[SA[i] \ldots \mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$.

To compute the $\mathcal{RSF}$ array, we use the next smaller value of the $\mathcal{LCP}$ array $\mathcal{NSV}_{\mathcal{LCP}}$, and the next smaller value of the reverse $\mathcal{LCP}$ array $\mathcal{NSV}_{\mathcal{LCP}^R}$, defined by $\mathcal{LCP}^R[i] = \mathcal{LCP}[n - i + 1]$, $1 \leq i \leq n$.

Definition 3.3. (Next smaller value of an array A ($\mathcal{NSV}_A$))
Given a non-negative integer array $A[1..n]$, the next smaller value of A ($\mathcal{NSV}_A$) is an integer array of length n, where $\mathcal{NSV}_A[i]$, $1 \leq i \leq n$, is defined as follows:

1. if $A[i] = 0$ then $\mathcal{NSV}_A[i] = 0$
2. otherwise, $NSV_A[i] = j$, where
   (a) for every $h \in [1..j-1]$, $A[i] \leq A[i+h]$, and
   (b) either $i + j = n + 1$ or $A[i] > A[i + j]$.

This definition differs from the standard definition of $\mathcal{NSV}$ [10] in its handling of zero values in A. For various approaches to compute the standard $\mathcal{NSV}$ array see [11]. See Figure 1 for an example.

In a suffix array all suffixes having identical prefixes are grouped together (appear contiguously), and as a result they are grouped together in the $\mathcal{LCP}$ array. In this group when there is an increase in the $\mathcal{LCP}$ value going from index i to $i + 1$, the longest common prefix at index $\mathcal{SA}[i]$ and of length $\mathcal{LCP}[i]$ also occurs at $\mathcal{SA}[i + 1]$. In fact, this holds for any substring identified by the $\mathcal{LCP}$ array before index i in this group. For example, in Figure 1 when the $\mathcal{LCP}$ value increases going from index four to five, the longest common prefix at $\mathcal{SA}[4]$; that is, aba also occurs at $\mathcal{SA}[5]$. Similarly, the substring 'a' occurring at $\mathcal{SA}[2]$ and length $1 < \mathcal{LCP}[5]$ also occurs at $\mathcal{SA}[5]$.

When there is a drop in the $\mathcal{LCP}$ value between index i and $i + 1$, all the substrings of length greater than $\mathcal{LCP}[i + 1]$ no longer appear in the subsequent suffixes in the suffix array; therefore

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\boldsymbol{w}$	a	b	a	c	a	b	a	b	a	c	a	b	a	c	a	b	a
$\mathcal{SA}$	17	15	5	11	1	7	13	3	9	16	6	12	2	8	14	4	10
$\mathcal{LCP}$	0	1	3	3	7	7	1	5	5	0	2	2	6	6	0	4	4
$\mathcal{NSV}_{\mathcal{LCP}}$	0	8	4	3	2	1	3	2	1	0	4	3	2	1	0	2	1
$\mathcal{LCP}^R$	4	4	0	6	6	2	2	0	5	5	1	7	7	3	3	1	0
$\mathcal{NSV}_{\mathcal{LCP}^R}$	2	1	0	2	1	2	1	0	2	1	6	2	1	2	1	1	0
$\mathcal{RSF}$	0	9	5	5	3	3	9	3	3	0	5	5	3	3	0	3	3

Figure 1. $\mathcal{NSV}_{\mathcal{LCP}}$, $\mathcal{NSV}_{\mathcal{LCP}^R}$ and $\mathcal{RSF}$ array (computed using Algorithm 1) values computed for $\boldsymbol{w} = abacababacabacaba$.

we can safely stop counting the occurrences of these repeating substrings. For example, in Figure 1 $\mathcal{LCP}[7] = 1$ implies that the repeating substrings aba and $abacaba$ no longer appear in the subsequent suffixes in the suffix array, and therefore we no longer need to count their occurrences.

Recall that $\mathcal{NSV}[i]$ is the number of positions after index i where the value of $\mathcal{LCP}$ first drops. As seen above, it also marks the index past which the repeating substring at $\mathcal{SA}[i]$ does not occur. Therefore, $\mathcal{NSV}[i]$ is the total number of following occurrences of the substring starting at index $\mathcal{SA}[i]$ and length $\mathcal{LCP}[i]$.

Note that $\mathcal{NSV}[i]$ is not equal to the frequency of the repeating substring occurring at $\mathcal{SA}[i]$ in $\boldsymbol{w}$. To compute its frequency, we need to compute the total number of its previous occurrences in the suffix array before index i. By simply counting its following occurrences there is a possibility of missing its previous occurrences. This could happen when the repeating substring is followed by a subword having smaller lexicographical rank and at other times followed by symbols with higher lexicographical rank.

Counting the previous occurrences is the same as counting the occurrences of this repeating substring forward in $\mathcal{LCP}^R$. Therefore, we compute $\mathcal{NSV}_{\mathcal{LCP}^R}$ analogous to computing $\mathcal{NSV}_{\mathcal{LCP}}$. Since by definition $\mathcal{LCP}[i] = \mathcal{LCP}^R[n-i+1]$, $\mathcal{NSV}_{\mathcal{LCP}^R}[n-i+1]$ is equal to the number of previous occurrences of the substring $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i]+\mathcal{LCP}[i]-1]$. Hence, adding the values $\mathcal{NSV}_{\mathcal{LCP}}[i]$ and $\mathcal{NSV}_{\mathcal{LCP}^R}[n-i+1]$ gives the frequency of the repeating substring $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i]+\mathcal{LCP}[i]-1]$, and this value is recorded in $\mathcal{RSF}[i]$ as seen in Algorithm 1. In view of this discussion, we claim the correctness of Algorithm 1, as stated in Lemma 3.4. Figure 1 shows the $\mathcal{NSV}_{\mathcal{LCP}}$, $\mathcal{NSV}_{\mathcal{LCP}^R}$ and $\mathcal{RSF}$ array values computed for $\boldsymbol{w} = abacababacabacaba$.

Based on the above discussion, we propose Algorithm 1 to compute $\mathcal{RSF}$ as follows:

1. First we compute $\mathcal{NSV}_{\mathcal{LCP}}[i]$, thus counting the number of *following* occurrences of the substring of $\boldsymbol{w}$ starting at $\mathcal{SA}[i]$ and of length $\mathcal{LCP}[i]$ in the $\mathcal{LCP}$ array.

2. We then compute $\mathcal{NSV}_{\mathcal{LCP}^R}[n-i+1]$, thus counting the *previous* occurrences of the substring of $\boldsymbol{w}$ starting at $\mathcal{SA}[i]$ and of length $\mathcal{LCP}[i]$ in the $\mathcal{LCP}$ array.

3. The total number of occurrences $\mathcal{RSF}[i]$ of the substring starting at $\mathcal{SA}[i]$ and length $\mathcal{LCP}[i]$ equals the total number of following and previous occurrences of the substring in $\boldsymbol{w}$, as shown in Algorithm 1.

Algorithm 1 Algorithm for computing $\mathcal{RSF}$ Array

```
procedure COMPUTE_RSF_ARRAY()          ▷ Precomputed NSV_LCP and NSV_LCP^R
    i ← 1
    while i ≤ n do
        RSF[i] = NSV_LCP[i] + NSV_LCP^R[n − i + 1]
        i ← i + 1
```

Lemma 3.4. Algorithm 1 correctly computes the frequency of all NRE repeating substrings in $\boldsymbol{w}$.

Lemma 3.5. Algorithm 1 executes in linear time and space in the length of the string.

Proof:
From [11], $\mathcal{NSV}_{\mathcal{LCP}}$ and $\mathcal{NSV}_{\mathcal{LCP}^R}$ can both be computed in $\mathcal{O}(n)$ time and space. Since computing $\mathcal{RSF}$ does simple addition and has a single **while** loop, it also executes in linear time and space. □

3.2. Algorithm to compute $\mathcal{RSF}$ optimized for space ($\mathcal{RSF}^*$)

By Lemma 3.5, Algorithm 1 executes in $\mathcal{O}(n)$ space. To be precise, the space required for computing Algorithm 1 is $9n$ bytes for storing $\boldsymbol{w}$, $\mathcal{LCP}$ and $\mathcal{NSV}_{\mathcal{LCP}}$, $8n$ bytes for storing $\mathcal{LCP}^R$ and $\mathcal{NSV}_{\mathcal{LCP}^R}$; in addition, storing these results in $\mathcal{RSF}$ array requires an additional $4n$ bytes, thus summing to $21n$ bytes. Each entry in $stack$ is 4 bytes, and the largest number of entries in $stack$ is the maximum depth of the suffix tree – thus $\mathcal{O}(n)$ in the worst case.

However, expected depth on an alphabet of size $\alpha > 1$ is $2\log_\alpha n$ [12]. Thus even for $\alpha = 4$ (DNA alphabet size), expected space for $stack$ is $4\log_4 n$ bytes — if $n = 4^{20}$, the expected stack space would be 80 bytes. Thus in practice, the stack space requirement is negligible. Therefore, the expected space requirement for Algorithm 1 is $21n$ bytes. This can be significantly and easily reduced to $9n$ bytes, by simply not explicitly storing the $\mathcal{LCP}^R, \mathcal{NSV}_{\mathcal{LCP}}$ and $\mathcal{NSV}_{\mathcal{LCP}^R}$ arrays. In particular, we can store the computed $\mathcal{NSV}_{\mathcal{LCP}}$ values directly in $\mathcal{RSF}$, then compute $\mathcal{NSV}_{\mathcal{LCP}^R}$ values and add it to an appropriate element in $\mathcal{RSF}$. Furthermore, to avoid unnecessary duplication, we set $\mathcal{RSF}[i] = 0$ whenever there exists $i' < i$ such that $\mathcal{LCP}[i'] = \mathcal{LCP}[i]$ and $\boldsymbol{w}[\mathcal{SA}[i']..\mathcal{SA}[i'] + \mathcal{LCP}[i'] - 1] = \boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i]+\mathcal{LCP}[i]-1]$. For the sake of clarity from this point on we denote the $\mathcal{RSF}$ optimized for space and time as discussed above as $\mathcal{RSF}^*$.

We propose Algorithm 2 to compute $\mathcal{NSV}_{\mathcal{LCP}}$ optimized for space for $\mathcal{RSF}^*$. It is similar to the algorithm proposed to compute $\mathcal{NSV}$ using a stack [11], with a few minor modifications: instead of storing the values in a separate $\mathcal{NSV}_{\mathcal{LCP}}$ array, we store these values directly in $\mathcal{RSF}^*$ array. Furthermore, to avoid unnecessary duplication, when the identified substring at $\mathcal{SA}[i]$ is identical to a substring previously identified by the $\mathcal{LCP}$ array, we set $\mathcal{RSF}^*[i] = 0$. Note that after Algorithm 2

Algorithm 2 Algorithm to compute $\mathcal{NSV}_{\mathcal{LCP}}\_\mathcal{RSF}^*$

```
procedure COMPUTE_NSV_LCP_RSF*()                          ▷ Precompute LCP array
    i ← 1
    top ← 0
    Empty stack
    while (i ≤ n) do
        if (stack.isempty()) then stack.push(i)
        else
            top ← stack.top()
            if (LCP[top] < LCP[i]) then stack.push(i)
            else if (LCP[top] == LCP[i]) then RSF*[i] ← 0
            else
                while LCP[top] > LCP[i] do
                    RSF*[i] ← i − top
                    stack.pop()
                    top ← stack.top()
                if (LCP[top] == LCP[i] and LCP[i] ≠ 0) then RSF*[i] ← 0
                else
                    stack.push(i)
        i ← i + 1
    Compute_NSV_stack_RSF*
```

```
procedure COMPUTE_NSV_STACK_RSF*()
    while (!stack.isempty()) do
        top ← stack.top()
        if (LCP[top] == 0) then RSF*[top] ← 0
        else
            RSF*[top] ← n + 1 − top
        stack.pop()
```

Figure 2. $Compute\_\mathcal{NSV}\_stack\_\mathcal{RSF}^*$ procedure to compute $\mathcal{NSV}$ for elements remaining in the stack after scanning $\mathcal{LCP}$ from left to right in Algorithm 2.

executes, the $\mathcal{RSF}^*$ array contains only partial values, as we still need to compute the $\mathcal{NSV}_{\mathcal{LCP}^R}$ values and add these to the appropriate element in the $\mathcal{RSF}^*$ array.

Note that Algorithm 2 with its modifications does not compute $\mathcal{NSV}_{\mathcal{LCP}^R}$ correctly. To address this, we propose Algorithm 3. Algorithm 3 computes the $\mathcal{NSV}_{\mathcal{LCP}^R}$ similar to [11], but has the following differences:

1. Since we are interested in computing $\mathcal{NSV}$ for $\mathcal{LCP}^R$ we scan the $\mathcal{LCP}$ array from right to left. Therefore we begin with $i = n$ instead of $i = 1$.

Algorithm 3 Algorithm to compute $\mathcal{RSF}^*$

```
procedure COMPUTE_RSF*()                                   ▷ Precompute LCP array
    COMPUTE_NSV_{LCP^R}_RSF*()
    Empty stack
    i ← n
    top ← 0
    while (i ≥ 1) do
        if (stack.isempty()) then stack.push(i)
        else
            top ← stack.top()
            if (LCP[top] ≤ LCP[i]) then stack.push(i)
            else
                top ← stack.top()
                while (!stack.isempty() and (LCP[top] > LCP[i]) do
                    if RSF*[top] ≠ 0 then
                        RSF*[top] ← RSF*[top] + top − i
                    stack.pop()
                    top ← stack.top()
                stack.push(i)
        i ← i − 1
```

2. After Algorithm 3 executes, the $\mathcal{LCP}$ values and consequently the $\mathcal{RSF}^*$ values of all the indices in the stack are zero. Since Algorithm 2 already computes these $\mathcal{RSF}^*$ values, and to avoid duplication, we do not compute these values again.

3. Since Algorithm 3 is primarily for computing the $\mathcal{RSF}^*$ array, the computed $\mathcal{NSV}_{\mathcal{LCP}^R}[top]$, $1 \leq i \leq n$, value $(top - i)$ is added to $\mathcal{RSF}^*[top]$ to get the frequency of the repeating substring $\boldsymbol{w}[\mathcal{SA}[top]..\mathcal{SA}[top] + \mathcal{LCP}[top] - 1]$.

Figure 3 shows the $\mathcal{NSV}_{\mathcal{LCP}}$, $\mathcal{NSV}_{\mathcal{LCP}^R}$, and $\mathcal{RSF}^*$ array values computed for the string $\boldsymbol{w} = abacababacabacaba$ after executing Algorithms 2 and 3. From the discussion above and because the modifications to the $\mathcal{NSV}$ algorithm in [11] are minor we claim the correctness of Algorithms 2 and 3, and we get the following lemmas.

Lemma 3.6. Algorithms 2 and 3 correctly compute the frequencies of all NRE repeating substrings in $\boldsymbol{w}$.

Lemma 3.7. Algorithms 2 and 3 execute in $\mathcal{O}(n)$ time and require a total of $9n$ bytes of space.

3.3. Computing frequency covers using $\mathcal{RSF}^*$ array

We propose Algorithm 4 to compute the frequency cover of $\boldsymbol{w}$ using the $\mathcal{RSF}^*$ array. The outline of Algorithm 4 is as follows: firstly, we assume that $\mathcal{SA}$, $\mathcal{LCP}$, and $\mathcal{RSF}^*$ arrays are precomputed. We

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\boldsymbol{w}$	a	b	a	c	a	b	a	b	a	c	a	b	a	c	a	b	a
$\mathcal{SA}$	17	15	5	11	1	7	13	3	9	16	6	12	2	8	14	4	10
$\mathcal{LCP}$	0	1	3	3	7	7	1	5	5	0	2	2	6	6	0	4	4
$\mathcal{NSV}_{\mathcal{LCP}}$	0	8	4	0	2	0	0	2	0	0	4	0	2	0	0	2	0
$\mathcal{NSV}_{\mathcal{LCP}^R}$	0	1	1	0	1	0	0	1	0	0	1	0	1	0	0	1	0
$\mathcal{RSF}^*$	0	9	5	0	3	0	0	3	0	0	5	0	3	0	0	3	0

Figure 3. $\mathcal{NSV}_{\mathcal{LCP}}$, $\mathcal{NSV}_{\mathcal{LCP}^R}$ and $\mathcal{RSF}^*$ array values computed for $\boldsymbol{w} = abacababacabacaba$ by Algorithms 2 and 3. Note that, $\mathcal{NSV}_{\mathcal{LCP}}$ values are stored in $\mathcal{RSF}^*$. And, $\mathcal{NSV}_{\mathcal{LCP}^R}[i]$ values are computed and added to the existing $\mathcal{RSF}^*[i]$.

do a left to right scan of the $\mathcal{RSF}^*$ array, and while doing so we maintain an integer array of length n, called FC, which stores the starting index of all the local frequency covers. By the end of the scan, FC stores the starting indices of all the frequency covers of $\boldsymbol{w}$. In the example shown in Figure 3, the string $\boldsymbol{w}$ has just one frequency cover aba covering 14 of 17 positions in $\boldsymbol{w}$.

procedure OUTPUT_FREQUENCY_COVERS(k)
  $i \leftarrow 1$
  $index \leftarrow 1$
  while ($i \leq k$) **do**
    $index = FC[i]$
    Output Frequency cover pairs: $(\mathcal{SA}[index], \mathcal{SA}[index] + \mathcal{LCP}[index] - 1])$
    $i \leftarrow i + 1$

Figure 4. Algorithm to output frequency covers. The FC array is assumed to be precomputed and available.

As part of proving the correctness of algorithm 4, we state and prove Lemmas 3.8 – 3.10.

Lemma 3.8. If for some $i \in [1..n]$, $\mathcal{LCP}[i] > 0$, then the substring $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$ is an NRE repeating substring of $\boldsymbol{w}$.

Proof:
The proof is by contradiction. Suppose $\mathcal{LCP}[i] > 0$ and the substring $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$ is right extendible. Then by definition every occurrence of $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$ is followed by the same symbol. Since $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$ is a substring of $\boldsymbol{w}$ all its occurrences are prefixes of distinct suffixes. Since $\mathcal{SA}$ orders the suffixes of $\boldsymbol{w}$ lexicographically, all suffixes with $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$ as the prefix appear contiguously in $\mathcal{SA}$. By definition $\mathcal{LCP}$ identifies the longest common prefix between two adjacent suffixes in $\mathcal{SA}$, in which case it would identify $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i]]$ as the repeating substring. This contradicts our assumption that $\boldsymbol{w}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$ is a repeating substring identified by $\mathcal{LCP}$ and is right extendible. □

Algorithm 4 Algorithm for computing all frequency covers of a string

```
procedure COMPUTE_FREQUENCY_COVERS()          ▷ Precompute, SA, LCP, RSF*
    max_frequency ← 1
    max_cover_length ← 1
    next_pointer ← 0                  ▷ Stores the total no. of frequency covers at i-th iteration
    i ← 1
    while i ≤ n do
        if (RSF*[i] > 1 and LCP[i] > 1) then
            if (RSF*[i] > max_frequency) then
                max_frequency = RSF*[i]
                max_cover_length = LCP[i]
                next_pointer ← 1
                FC[next_pointer] ← i
            else if (RSF*[i] == max_frequency) then
                if (max_cover_length == LCP[i]) then
                    next_pointer ← next_pointer + 1
                    FC[next_pointer] ← i
                else if (max_cover_length < LCP[i]) then
                    max_cover_length ← LCP[i]
                    next_pointer ← 1
                    FC[next_pointer] = i
        i ← i + 1
    Output_Frequency_Covers(next_pointer)
```

Let M be the maximum of the frequencies of all substrings in $\boldsymbol{w}$. Let the substrings that occur M times; that is, having frequency M, be $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$. Let L be the maximum length of any substring in $\mathcal{S}$. Recall that for a string $\boldsymbol{w}$ to have a frequency cover, $M > 1$ and $L > 1$, which we therefore assume.

Lemma 3.9. Suppose $\mathcal{V}$ is the set of all NRE repeating substrings of $\boldsymbol{w}$. If $\boldsymbol{u} \in \mathcal{V}$, where $f_{\boldsymbol{w},\boldsymbol{u}} = M$ and $|\boldsymbol{u}| = L$, then $\boldsymbol{u}$ is an NE repeating substring of $\boldsymbol{w}$.

Proof:
The proof is by contradiction. Suppose $\boldsymbol{u}$ is a left extendible repeating substring of $\boldsymbol{w}$. Then every occurrence of $\boldsymbol{u}$ in $\boldsymbol{w}$ is preceded by the same symbol (say $a \in \Sigma$). Then by definition, $f_{\boldsymbol{w},\boldsymbol{u}} = f_{\boldsymbol{w},a\boldsymbol{u}} = M$. Since $\boldsymbol{u}$ is an NRE repeating substring, $a\boldsymbol{u}$ is also an NRE repeating substring, and $a\boldsymbol{u} \in \mathcal{V}$, where $|a\boldsymbol{u}| > L$. But this contradicts our assumption that $|\boldsymbol{u}| = L$, where L is the maximum length of a substring occurring M times in $\boldsymbol{w}$. Therefore $\boldsymbol{u}$ is an NE repeating substring. □

Lemma 3.10. $\boldsymbol{u}$ is a frequency cover of $\boldsymbol{w} \Leftrightarrow \boldsymbol{u}$ is an NE repeating substring of $\boldsymbol{w}$, where $|\boldsymbol{u}| = L$ and $f_{\boldsymbol{w},\boldsymbol{u}} = M$.

Proof:
($\Leftarrow$) Proof of this direction follows from the definition of frequency cover.
($\Rightarrow$) Proof of this direction is by contradiction. Let $\boldsymbol{u}$ be a frequency cover of $\boldsymbol{w}$. Then $|\boldsymbol{u}| = L$ and $f_{\boldsymbol{w},\boldsymbol{u}} = M$ by definition. Suppose $\boldsymbol{u}$ is right extendible (or left extendible). This means that every occurrence of $\boldsymbol{u}$ is succeeded (or preceded) by the same symbol (say $a \in \Sigma$). But then, $\boldsymbol{u}a$ (or $a\boldsymbol{u}$) of length $> L$ is the frequency cover and not $\boldsymbol{u}$ – a contradiction. □

Theorem 3.11. Algorithm 4 correctly computes all frequency covers in the given string $\boldsymbol{w}$.

Proof:

By Lemma 3.8 and 3.9, Algorithm 4 stores only those NE repeating substrings of length L and frequency M in $\boldsymbol{w}$. By Lemma 3.10, these NE repeating substrings represent all the frequency covers of the string $\boldsymbol{w}$. Therefore Algorithm 4 correctly computes all the frequency covers in $\boldsymbol{w}$. □

Theorem 3.12. Algorithm 4 computes all the frequency covers in the given string $\boldsymbol{w}$ in $\mathcal{O}(n)$ time and space.

Proof:
By Lemma 3.5 it is clear that Algorithm 4 can be computed in $\mathcal{O}(n)$ time. The space required for Algorithm 4 is $17n$ bytes for storing $\boldsymbol{w}, \mathcal{SA}, \mathcal{LCP}, \mathcal{RSF}^*$ and FC arrays. This can be easily reduced to $13n$ bytes by merging Algorithms 3 and 4 and not storing $\mathcal{RSF}^*$ explicitly. □

Furthermore, the space required by FC array is rarely n for large strings over small alphabet (e.g., DNA Sequences). Therefore this reduces the space requirement further; for all practical purposes computing frequency covers takes $9n$ bytes.

4. Experimental results

In order to get some idea of the effectiveness/utility of frequency covers, we computed the frequency covers of sample strings found at

`http://www.cas.mcmaster.ca/~bill/strings/`.

The results are summarized in Table 5. In the table the % of positions covered by the frequency cover is based on the assumption that no two instances of the frequency cover in the string overlap. When the % of positions covered by a frequency cover is more than 100, it implies that the frequency cover, covers the entire string and also some of its occurrences overlap. For example, the frequency cover ata for some of the Fibonacci strings seen in Table 5 cover the entire string. In fact it is observed that when the Fibonacci string ends with a t the frequency cover is at and covers a certain percent of the string. However, when the the Fibonacci string ends with an a the frequency cover of the string is ata which covers the entire Fibonacci string. Note that in the table, the sample run-rich strings are highly periodic string and are explained in [13].

Text Type	Description	No. of letters (n)	σ	Frequency Cover	occ	% PC (computed based on no overlap) $\leq$
DNA	Chromosome 19	63,811,650	4	TT	4,762,324	14.93
	Chromosome 22	34,553,757	4	CA	2,646,891	15.32
	Escherichia coli genome (E.coil)	4,638,689	4	gc	383,734	16.54
Protein	Protein sequences	1,048,575	20	LL	11,829	2.26
		2,097,151	20	LL	24,046	2.29
		4,194,303	20	LL	45,471	2.17
		8,388,607	20	XX	204,436	4.87
		16,777,215	20	XX	204787	2.44
		33,554,431	20	AA	357107	2.13
Highly Periodic	Fibonacci strings	2,178,308	2	ata	832,040	114.6
		3,524,577	2	at	1,346,269	76.4
		5,702,886	2	ata	2,178,309	114.6
		9,227,464	2	at	3,524,578	76.4
		14,930,351	2	ata	5,702,887	114.6
	Run-rich strings [13]	2,851,442	2	at & ta	2,851,443	76.4
		12,078,907	2	at & ta	4,613,732	76.4
Random		4,194,303	2	aa	1,049,423	50.04
		8,388,607	2	at & ta	2,097,319	50
		4,194,303	21	in	9814	0.47
		8,388,607	21	om	19,424	0.46
English	King James bible	4,047,391	63	th	148,979	7.36
	LINUX howto files	39,422,104	197	(two contiguous spaces)	2,881,448	14.62

Figure 5. Experimental results – FC denotes frequency cover, occ denotes no. of occurrences of the frequency cover in the string, and % PC denotes the percentage of positions covered computed based on no overlap.

5. Computing non-extendible repeating substrings in strings using $\mathcal{RSF}$

It turns out that $\mathcal{RSF}$ can be used to compute all the non-extendible repeating substrings in $\boldsymbol{w}$. These data structures are important in bioinformatics applications; algorithms to compute them were described in [14, 15, 16] using suffix trees or suffix arrays. We introduce the inverse $\mathcal{RSF}$ array $\mathcal{IRSF}$ to compute all non-extendible repeating substrings in $\boldsymbol{w}$.

Definition 5.1. ($\mathcal{IRSF}$ array)
The **"inverse repeating substring frequency array"** $\mathcal{IRSF}$ of $\boldsymbol{w}$ of length n is an integer array of length n, where $\mathcal{IRSF}[i] = \mathcal{RSF}[\mathcal{ISA}[i]]$.

To compute all non-extendible repeating substrings in $\boldsymbol{w}$ we need the $\mathcal{ILCP}$ and $\mathcal{IRSF}$ arrays precomputed; these can be easily computed from the $\mathcal{ISA}$, $\mathcal{LCP}$ and $\mathcal{RSF}$ arrays, respectively, in linear time and space. We propose Algorithm 5 to compute all non-extendible repeating substrings in $\boldsymbol{w}$

Algorithm 5 Algorithm to compute NE repeating substrings in a string $\boldsymbol{w}$

```
procedure COMPUTE_NE()                          ▷ Precompute ILCP and IRSF arrays
    if IRSF[1] ≠ 0 then
        Output NE repeating substring pair: (1, ILCP[1])
    i ← 2
    while (i ≤ n) do
        if IRSF[i] ≠ 0 then
            if !(IRSF[i] == IRSF[i − 1] && ILCP[i] + 1 == ILCP[i − 1]) then
                Output NE repeating substring pair: (i, i + ILCP[i] − 1)
        i ← i + 1
```

outlined as follows:

1. We scan the $\mathcal{IRSF}$ array from left to right.
2. If $\mathcal{IRSF}[1] \neq 0$, then it implies that the prefix of $\boldsymbol{w}$ of length $\mathcal{ILCP}[1]$ is an NRE repeating substring. From Lemma 5.2, it is also an NE repeating substring. Therefore we output the integer pair corresponding to this substring. The first and second integers of this pair correspond to the starting and ending index of the NE repeating substring.
3. For all $\mathcal{IRSF}[i] > 0$, $2 \leq i \leq n$, we check if the substring at i is left extendible. If it is ***not*** left extendible we output the starting and ending indices of the repeating substring in $\boldsymbol{w}$.

Lemma 5.2. Suppose $\boldsymbol{u}$ is an NRE repeating substring of $\boldsymbol{w}$. If $\boldsymbol{u}$ is a prefix of $\boldsymbol{w}$ then $\boldsymbol{u}$ is an NE repeating substring.

Proof:
The proof is by contradiction. Suppose $\boldsymbol{u}$ is left extendible. Then, by definition each occurrence of $\boldsymbol{u}$ in $\boldsymbol{w}$ is preceded by the same symbol. However, $\boldsymbol{u}$ is a prefix of $\boldsymbol{w}$, and this occurrence of $\boldsymbol{u}$ has

no preceding symbol. This contradicts our assumption that $\boldsymbol{u}$ is a left extendible repeating substring. Since $\boldsymbol{u}$ is both an NRE and NLE repeating substring, it follows that $\boldsymbol{u}$ is an NE repeating substring. □

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
w	a	b	a	c	a	b	a	b	a	c	a	b	a	c	a	b	a
$\mathcal{SA}$	17	15	5	11	1	7	13	3	9	16	6	12	2	8	14	4	10
$\mathcal{LCP}$	0	1	3	3	7	7	1	5	5	0	2	2	6	6	0	4	4
$\mathcal{RSF}$	0	9	5	5	3	3	9	3	3	0	5	5	3	3	0	3	3
$\mathcal{ISA}$	5	13	8	16	3	11	6	14	9	17	4	12	7	15	2	10	1
$\mathcal{ILCP}$	7	6	5	4	3	2	7	6	5	4	3	2	1	0	1	0	0
$\mathcal{IRSF}$	3	3	3	3	5	5	3	3	3	3	5	5	9	0	9	0	0
	NE	X	X	X	NE	X	NE	X	X	X	NE	X	NE	X	NE	X	X

Figure 6. $\mathcal{ISA}$, $\mathcal{ILCP}$ and $\mathcal{IRSF}$ array values computed for $\boldsymbol{w} = abacababacabacaba$. '$NE$' stands for non-extendible repeating substring and 'X' stands for left-extendible repeating substrings.

Lemma 5.3. Suppose $\boldsymbol{u} = \boldsymbol{w}[i..j]$ and $\boldsymbol{u'} = \boldsymbol{w}[i-1..j]$, $2 \leq i < n$, are two NRE repeating substrings of $\boldsymbol{w}$. If $f_{\boldsymbol{w},\boldsymbol{u}} = f_{\boldsymbol{w},\boldsymbol{u'}}$, then $\boldsymbol{u}$ is left extendible.

Proof:
The proof is by contradiction. Suppose $\boldsymbol{u}$ is an NLE repeating substring. Then by definition there is at least one occurrence of $\boldsymbol{u}$ which is not preceded by the symbol $\boldsymbol{u'}[1]$. But this contradicts our assumption that $f_{\boldsymbol{w},\boldsymbol{u}} = f_{\boldsymbol{w},\boldsymbol{u'}}$. Therefore, $\boldsymbol{u}$ is left extendible. □

The correctness of Algorithm 5 follows from Lemmas 3.8, 5.2 and 5.3. It is easy to see that Algorithm 5 can be computed in $\mathcal{O}(n)$ time and space. Consequently, we get Theorems 5.4 and 5.5.

Theorem 5.4. Algorithm 5 correctly computes all NE repeating substrings of $\boldsymbol{w}$.

Theorem 5.5. Algorithm 5 computes all NE repeating substrings of $\boldsymbol{w}$ in $\mathcal{O}(n)$ time and space.

Note that there are other linear time algorithms proposed to compute non-extendible repeating substrings in a string that are more space efficient [16]. We present this algorithm to show the usefulness of the $\mathcal{RSF}$ data structure.

6. Conclusions

The advantage of the frequency cover introduced in the paper is that it is applicable to all strings and can be computed in linear time as opposed to the other generalized notion of cover — the α-partial cover [9] which can be computed in $\mathcal{O}(n \log n)$ time. Additionally, its computation does not make use of space consuming annotated suffix trees. However, the frequency cover has a drawback. The frequency cover does not always cover the maximum number of positions possibly covered by any repeating substring. For example, in $\boldsymbol{w} = abacababacabacaba$, the frequency cover of $\boldsymbol{w}$ is aba which covers 14 of 17 positions in $\boldsymbol{w}$. However, the substring $abacaba$ covers a total of 17 positions in $\boldsymbol{w}$, thus covering the entire string. Although [9] shows that computing the $\alpha-$partial cover of a given string for all values of α can be computed in $\mathcal{O}(n \log n)$ time, it has the disadvantage of using annotated suffix trees. Therefore there still remains a need for a notion of an "optimal cover" that not only is applicable to all strings but that also is a substring covering a maximum number of positions in the given string, and that can be computed efficiently.

References

[1] Apostolico A, Ehrenfeucht A. Efficient Detection of Quasi–periodicities in Strings. Technical Report 90.5, The Leonadro Fibonacci Institute, Trento, Italy, 1990.

[2] Apostolico A, Ehrenfeucht A. Efficient Detection of Quasiperiodicities in Strings. *Theoretical Computer Science*, 1993. **119**(2):247–265.

[3] Flouri T, Iliopoulos CS, Kociumaka T, Pissis SP, Puglisi SJ, Smyth WF, Tyczyński W. Enhanced string covering. *Theoretical Computer Science*, 2013. **506**:102–114.

[4] Apostolico A, Farach M, Iliopoulos C. Optimal superprimitivity testing for strings. *Inform. Process. Lett.*, 1991. **39**:17–20.

[5] Breslauer D. An on-line string superprimitivity test. *Information Processing Letters*, 1992. **46**(6):345–347.

[6] Moore D, Smyth WF. An optimal algorithm to compute all the covers. *Information Processing Letters*, 1994. **50**:239–246.

[7] Li Y, Smyth WF. Computing the cover array in linear time. *Algorithmica, 32–1, 95–106*, 2002.

[8] Alatabbi A, Islam ASMS, Rahman MS, Simpson J, Smyth WF. Enhanced Covers of Regular & Indeterminate Strings using Prefix Tables. *Journal of Automata, Languages & Combinatorics*, 2016. **21**(3):131–147.

[9] Kociumaka T, Radoszewski J, Rytter W, Pissis SP, Waleń T. Fast Algorithm for Partial Covers in Words. *Algorithmica*, 2015. **73**(1):217 – 233.

[10] Franek F, Islam ASMS, Rahman MS, Smyth WF. Algorithms to Compute the Lyndon Array. In: Proceedings of the Prague Stringology Conference 2016. 2016 pp. 172–184.

[11] Goto K, Bannai H. Simpler and faster Lempel-Ziv factorization. *Data Compression Conference*, 2013. pp. 133–142.

[12] Karlin S, Ghandour G, Ost F, Tavare S, Korn LJ. New approaches for computer analysis of nucleic acid sequences. *Proc. Natl. Acad. Sci. USA*, 1983. **80**:5660–5664.

[13] Franek F, Simpson R, Smyth WF. The maximum number of runs in a string. *Proceeding of 14th Australian Workshop on Combinatorial Algorithms*, 2003. pp. 26–35.

[14] Gusfield D. Algorithms on Strings, Trees, and Sequences: Computer Science And Computational Biology. Cambridge University Press, 1997. ISBN 0-521-58519-8(hc).

[15] Smyth B. Computing Patterns in Strings. Pearson/Addison–Wesley, 2003. ISBN 9780201398397.

[16] Puglisi SJ, Smyth WF, Yusufu M. Fast, optimal algorithms for computing all the repeats in a string. *Mathematics in Computer Science*, 2010. **3–4**:373–389.