Improved Practical Algorithms to Compute Maximal Covers

Holly Koponen, Neerja Mhaskar, and W. F. Smyth*

Algorithms Research Group, Department of Computing & Software
McMaster University, Canada
koponeh@mcmaster.ca, pophlin@mcmaster.ca, smyth@mcmaster.ca

Abstract. A ***cover*** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a repeating substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that every position in $\boldsymbol{x}$ lies within an occurrence of $\boldsymbol{u}$. Since very few strings possess a cover, it becomes interesting to compute various kinds of cover generalizations. Here we describe algorithms to compute a ***maximal cover***[1]; that is, a repeating substring $\boldsymbol{u}$ of $\boldsymbol{x}$ that covers $M = M_{\boldsymbol{x}}$ positions, the maximum coverage attained by any repeating substring of $\boldsymbol{x}$. In 2015, an $O(n \log n)$-time algorithm to compute a maximal cover was proposed; but the algorithm was complex, making use of annotated suffix trees. In 2022, an $O(n^2)$-time maximal cover algorithm was implemented and evaluated on protein sequences. In this paper, we propose two simple $O(n^2)$-time algorithms for this problem that, nonetheless, as we show by experiment, execute in linear time in many cases that arise in practice, and are much faster than the algorithm recently implemented in 2022. On the other hand, when experiments are restricted to the highly repetitive Fibonacci strings, the behaviour of both algorithms is clearly quadratic.

Keywords: string, cover, Fibonacci, algorithm

1 Introduction

As introduced in [2,3], a ***cover*** of a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is a proper substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that every position of $\boldsymbol{x}$ lies within an occurrence of $\boldsymbol{u}$ — thus a cover occurs at least twice in $\boldsymbol{x}$. For example, $\boldsymbol{u} = aba$ is a cover of $\boldsymbol{x} = ababaaba$. Even though all the covers of every prefix of $\boldsymbol{x}$ can be computed, whenever they exist, in $O(n)$ time [12], nevertheless, since very few strings possess a cover, in order to provide a compact representation of $\boldsymbol{x}$, various alternate covering structures have been proposed:

- *k-cover* [9]: a minimum collection of substrings of $\boldsymbol{x}$, each of given length $k < n$, that covers $\boldsymbol{x}$ — this computation turns out to be NP-hard [5];
- *enhanced cover* [7,1]: the border $\boldsymbol{u}$ (both prefix and suffix) of $\boldsymbol{x}$ that, over all borders of $\boldsymbol{x}$, covers a maximum number of positions — computable in $\Theta(n)$ time, but its effectiveness depends on some border also being a good cover — a rare occurrence.
- *α-partial cover* [10]: the shortest substring $\boldsymbol{u}$ of $\boldsymbol{x}$ (if it exists) that, for $\alpha = 1, 2, \ldots, n$, covers at least α positions in $\boldsymbol{x}$ — this computation uses only $\mathcal{O}(n \log n)$ time over all values of α and thus also computes the *maximal cover* (next entry), but on the other hand requires space-consuming data structures ("augmented and annotated" suffix trees);

* Supported by Grant No. 105–36797 from the Natural Sciences & Engineering Research Council of Canada (NSERC).

[1] The term "optimal cover" was used in [13].

Holly Koponen, Neerja Mhaskar, W. F. Smyth: Improved Practical Algorithms to Compute Maximal Covers, pp. 30–41.
Proceedings of PSC 2023, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07206-6 © Czech Technical University in Prague, Czech Republic

- *maximal cover* [14]: a substring $\boldsymbol{u}$ of $\boldsymbol{x}$ that occurs at least twice and that, over all substrings, covers a maximum number $M_{\boldsymbol{x}}$ of positions — to compute $\boldsymbol{u}$, an $\mathcal{O}(n^2)$-time and $\Theta(n)$-space algorithm is described[2];
- *frequency cover* [13]: any substring $\boldsymbol{u}$ of $\boldsymbol{x}$, of length greater than 1, that occurs most frequently — this requires only $\Theta(n)$ time, but yields a good cover only if $\boldsymbol{x}$ contains many short repeating substrings;

For example, given $\boldsymbol{x} = abababaaaba$ of length 9, there is no cover as defined above, but the set $\{aba, aab\}$ is a 3-cover; aba is a maximal cover, also an 8-partial cover (there is no 9-partial cover); ab and aba are frequency covers; aba is an enhanced cover. For further reading, see the survey [15].

In this paper, we introduce terminology and symbolism in Section 2. Then in Section 3 we outline the overall methodology for calculating maximal covers of $\boldsymbol{x}$ from the "Overlapping Positions" $\mathcal{OLP}$ array, which is the fundamental data structure underlying this computation. Section 4 describes the three competing $\mathcal{O}(n^2)$ algorithms[3] that compute $\mathcal{OLP}$, one already published in [14], and implemented in [8]. Here we describe two new ones introduced in [11] and provide computational evidence that they are faster in practice, requiring only linear time on average over a wide range of test cases. Section 5 compares the execution times of all three $\mathcal{OLP}$ algorithms applied to random strings with alphabet sizes $\sigma = \{2, 3, 4\}$, to Fibonacci sequences, and to protein sequences. Section 6 summarizes our findings and proposes future work.

2 Preliminaries

A ***string*** $\boldsymbol{x}[1..n]$ is a concatenation of symbols drawn from a totally ordered set Σ, called an ***alphabet***, where $\sigma = |\Sigma|$. The ***length*** of $\boldsymbol{x}$ is $|\boldsymbol{x}| = n$. A string of length zero is called an ***empty string*** and denoted by ε. A string $\boldsymbol{u}$ is called a ***substring*** of $\boldsymbol{x}[1..n]$ if $\boldsymbol{u} = \varepsilon$ or $\boldsymbol{u} = \boldsymbol{x}[i..j]$ and $1 \leq i \leq j \leq n$, a ***proper substring*** if $|\boldsymbol{u}| < n$. A substring $\boldsymbol{u}$ of $\boldsymbol{x}[1..n]$ is called a ***prefix (suffix)*** of $\boldsymbol{x}$ if $\boldsymbol{u} = \varepsilon$ or $\boldsymbol{u} = \boldsymbol{x}[1..i]$ and $1 \leq i \leq n$ ($\boldsymbol{u} = \boldsymbol{x}[j..n]$ and $1 \leq j \leq n$). A ***border*** $\boldsymbol{u}$ of $\boldsymbol{x}$ is a proper substring of $\boldsymbol{x}$ that is both a prefix and suffix of $\boldsymbol{x}$. For example, $\boldsymbol{x} = abacaba$ has borders aba, a and ε. A substring $\boldsymbol{u}$ of $\boldsymbol{x}$ is a ***repeating substring*** of $\boldsymbol{x}$ if it occurs at least twice in $\boldsymbol{x}$. A ***left (right) extendible*** repeating substring $\boldsymbol{u}$ of $\boldsymbol{x}$ is a repeating substring of $\boldsymbol{x}$ whose each occurrence in $\boldsymbol{x}$ is preceded (followed) by the same symbol. If every occurrence of $\boldsymbol{u}$ is not preceded (followed) by the same symbol then it is called a ***non-left (non-right) extendible, NLE (NRE)*** repeating substring of $\boldsymbol{x}$.

A ***run*** in $\boldsymbol{x}$ is a substring $\boldsymbol{x}[i..j] = \boldsymbol{u}^e\boldsymbol{u'}$, where $e \geq 2$, $\boldsymbol{u'}$ is a prefix of $\boldsymbol{u}$, and the periodicity $|\boldsymbol{u}|$ cannot be extended left or right [18].

In this paper, we describe and compare algorithms that compute maximal covers $\boldsymbol{u}$; that is, repeating substrings that cover a maximum $M = M_{\boldsymbol{x}}$ positions of a given string $\boldsymbol{x}$. It may be that more than one substring $\boldsymbol{u}$ covers M positions; for example, $\boldsymbol{x} = aabaababaaba$ of length 12 has three maximal covers: $\boldsymbol{u}_1 = aba$, $\boldsymbol{u}_2 = aaba$, $\boldsymbol{u}_3 = (aba)^2$. Thus the longest (shortest) maximal cover may be of interest: the longest could provide more information about the structure of $\boldsymbol{x}$, while the shortest is more compact.

[2] An $\mathcal{O}(n \log n)$-time algorithm proposed in [14] was incorrect.

[3] All algorithms described in this paper are assumed to run on a word RAM model with word size $= k$ bits, $k \leq \log n$, where n is the input size.

The maximal cover algorithms proposed here both require two well-known data structures: the $\mathcal{SA}$ and $\mathcal{LCP}$ arrays. The ***suffix array*** $\mathcal{SA}$ is an integer array of length $|\boldsymbol{x}|$ such that $\mathcal{SA}[i]$ is the starting position of the i-th lexicographically least suffix in $\boldsymbol{x}$. The ***longest common prefix array*** $\mathcal{LCP}$ is also an integer array of length $|\boldsymbol{x}|$, where $\mathcal{LCP}[1] = 0$ and $\mathcal{LCP}[i]$, $i \in [2..n]$, is the length of the longest common prefix between the suffixes of $\boldsymbol{x}$ starting at $\mathcal{SA}[i-1]$ and $\mathcal{SA}[i]$. For convenience, we define $\boldsymbol{v}_i$ to be the NRE repeating substring of length $\mathcal{LCP}[i]$ occurring at positions $\mathcal{SA}[i-1], \mathcal{SA}[i]$ in $\boldsymbol{x}$ — that is,

$$\boldsymbol{v}_i = \boldsymbol{x}[SA[i] \mathinner{..} \mathcal{SA}[i] + \mathcal{LCP}[i] - 1]. \quad (1)$$

3 Computing Maximal Covers

In order to compute maximal covers of $\boldsymbol{x}$, we require the $\mathcal{SA}_x$ and $\mathcal{LCP}_x$ arrays, defined above. Also required are the following:

- The ***repeating substring frequency array*** $\mathcal{RSF}$ (introduced in [13]) is an integer array of length $|\boldsymbol{x}| = n$ such that $\mathcal{RSF}[i]$ is the number of occurrences, that is 'frequency', of the NRE repeating substring v_i in x. Thus, if $\mathcal{RSF}[i] = m$, then there exist m consecutive index positions $r \in \{r_1, r_1+1, ..., r_m-1, r_m\}$ in $\mathcal{SA}$ such that $\boldsymbol{x}[\mathcal{SA}[r]..n]$ has prefix $\boldsymbol{v}_i$. Note that for $\mathcal{LCP}[i] = 0$, $\mathcal{RSF}[i] = 0$.
- The ***overlapping positions array*** $\mathcal{OLP}$ (introduced in [14]) is an integer array of length $|\boldsymbol{x}|$ such that $\mathcal{OLP}[i]$ is the total number of overlapping positions between adjacent occurrences of the NRE repeating substring $\boldsymbol{v}_i$ in x.
- The ***repeating substring positions covered array*** $\mathcal{RSPC}$ (introduced in [14]) is an integer array of length $|\boldsymbol{x}|$ such that $\mathcal{RSPC}[i]$ is the number of distinct positions covered by $\boldsymbol{v}_i$. Hence,

$$\mathcal{RSPC}[i] = \mathcal{LCP}[i] * \mathcal{RSF}[i] - \mathcal{OLP}[i], \quad (2)$$

  a straightforward linear-time computation.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\boldsymbol{x}$	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
$\mathcal{OLP}$	0	0	1	1	4	4	0	1	1	0	0	0	2	2	0	0	0
$\mathcal{RSPC}$	0	9	14	14	17	17	9	14	14	0	10	10	16	16	0	12	12

Figure 1: $\mathcal{SA}$, $\mathcal{LCP}$, $\mathcal{RSF}$, $\mathcal{OLP}$ and $\mathcal{RSPC}$ arrays computed for the string $\boldsymbol{x} = abacababacabacaba$.

The $\mathcal{SA}$ and $\mathcal{LCP}$ arrays are computed in linear time using the well-known implementations of Yuta Mori[4] and Puglisi [17], respectively. To compute the $\mathcal{RSF}$ array, we use the $\Theta(n)$-time algorithm given in [13]. For the $\mathcal{OLP}$ array, we describe below two simple $O(n^2)$-time algorithms from [11], comparing their efficiency with the algorithm given in [14], and implemented in [8]. We optimize the computation of the $\mathcal{RSF}$ and $\mathcal{OLP}$ arrays by replacing duplicate values with zeros to avoid recomputing them for identical substrings $\boldsymbol{v}_i$.

Finally, the $\mathcal{RSPC}$ array is computed based on Equation 2, then scanned greedily [14, Algorithm 4] in linear time to complete the expected linear-time calculation of maximal cover. For more details on the MAXCOVER software architecture and data structures, see [11][5].

4 Computing the Overlapping Positions Array

In this section, we compare three worst-case $\mathcal{O}(n^2)$ algorithms to compute the $\mathcal{OLP}$ array that underlies the computation of the maximal cover.

4.1 Original $\mathcal{OLP}$ Algorithm

We call the original algorithm proposed in [14, Algorithm 1] and implemented in [8] the **OLP-1** algorithm. For completeness, we present it as Algorithm 1 and briefly explain it here.

Algorithm 1 OLP-1 Algorithm

```
procedure COMPUTE_OLP*_ARRAY_QUADRATIC(R1, RM)
    i ← 1
    for i ← 1 to n do                         ▷ u is a distinct NRE repeating substring
        if (RSF*[i] = 0) then OLP*[i] = 0
        else                                  ▷ u occurs in SA in range [R1[i], RM[i]]
            OLP*[i] ← Compute_OLPi*(i, R1[i], RM[i])
    return OLP*
```

The OLP-1 algorithm takes $R1$ and RM integer arrays of length n as input. The $R1[i]$ and $RM[i]$ stores the r_1 and r_m values for $\boldsymbol{v}_i$. In the implementation by [11], $R1[i]$ and $RM[i]$ are computed in linear time by traversing the $\mathcal{LCP}$ array and keeping track of the rise and fall of values using a stack. Then, as OLP-1 traverses the $\mathcal{LCP}$ array from left to right, for each $\boldsymbol{v}_i$, it computes the $\mathcal{OLP}[i]$ value using the $Compute_\mathcal{OLP}i^*()$ procedure.

The $Compute_\mathcal{OLP}i^*()$ procedure, first copies the m starting positions of $\boldsymbol{v}_i$ found in the range $\mathcal{SA}[r_1..r_m]$, to a temporary array $\mathcal{SA}^*[r_1..r_m]$ and sorts it. Then for an occurrence of $\boldsymbol{v}_i$ in the ordered range $\mathcal{SA}^*[r_1..r_m]$ (starting at index position r_1), it checks for an adjacent overlapping occurrence of $\boldsymbol{v}_i$ using the *exrun* function introduced and defined in [4] . The *exrun* function primarily returns the run $\boldsymbol{r}$ containing the adjacent occurrences of $\boldsymbol{v}_i$, if it exists. In which case, the procedure computes the

[4] Based on the SA-IS algorithm by [16] accessed from: `https://sites.google.com/site/yuta256/`

[5] **Availability:** Open source code, binaries, and test data are available on Github at `https://github.com/hollykoponen/MAXCOVER`. The software currently runs on Linux and is untested on other OS.

```
procedure COMPUTE_OLPi*(index, r_1, r_m)
    ℓ ← LCP[index]                                          ▷ ℓ is the length of v_i
    OLPi ← 0
    Copy elements SA[r_1..r_m] to SA*[r_1..r_m] and sort in ascending order.
    k ← r_1
    while k < r_m do
        if (SA*[k + 1] − SA*[k] < ℓ) then                   ▷ Test for overlap
            r ← exrun(SA*[k], SA*[k + 1] + ℓ − 1)           ▷ r = (i, j, p)
            f_{r,v_i} ← frequency of v_i in run r
            OLPi ← OLPi + (ℓ − p) × (f_{r,v_i} − 1)
            k ← k + f_{r,v_i} − 1
        else
            k ← k + 1
    return OLPi
```

Figure 2: Compute the set of eligible runs for an NRE repeating substring

frequency of $\boldsymbol{v}_i$ in the run $\boldsymbol{r}$ and computes the total overlapping positions between adjacent occurrences of $\boldsymbol{v}_i$ in the run $\boldsymbol{r}$. Note that, this results in skipping some of the positions in $\mathcal{SA}^*[r_1..r_m]$.

The OLP-1 algorithm executes in $\mathcal{O}(n^2)$ time. The *exrun* queries are answered in constant time by preprocessing the given string in linear time. However, the quadratic execution time results from multiple traversals of a range in $\mathcal{SA}[r_1..r_m]$ (possibly n times).

4.2 Improved Algorithms: OLP-2 and OLP-3

Here we propose two improved algorithms, **OLP-2** and **OLP-3**, that use simple techniques and data structures to compute the $\mathcal{OLP}$ array more efficiently, even though the worst-case time complexity remains $\mathcal{O}(n^2)$.

Similar to OLP-1, both algorithms traverse the $\mathcal{LCP}$ array to compute the overlaps between adjacent occurrences of $\boldsymbol{v}_i$. They also compute the values r_1 and $r_m = r_1 + m - 1$, then copy the values in $\mathcal{SA}[r_1..r_m]$ to $\mathcal{SA}^*[r_1..r_m]$ and sort them to compute the total $\mathcal{OLP}$ overlap.

Both OLP-2 and OLP-3 use:

$$\sum_{j=1}^{m-1} (\max\{0, \mathcal{LCP}[i] - \mathcal{SA}^*[r_1 + j] + \mathcal{SA}^*[r_1 + j - 1]\}) \tag{3}$$

to determine the total overlap, if any, between adjacent occurrences of $\boldsymbol{v}_i$.

However, OLP-2 and OLP-3 do not require complicated data structures to compute overlaps — thus significantly reducing execution time. The primary difference between OLP-2 and OLP-3 is the methodology used to compute r_1 for each $\boldsymbol{v}_i$ while tracking the changes in the $\mathcal{LCP}$ array values. OLP-2 traverses the $\mathcal{LCP}$ array in reverse multiple times to compute r_1, while OLP-3 (inspired by OLP-1) computes r_1 using a stack.

In OLP-3, we use a stack that stores pairs of integer values in the format: $(index,\ r_1)$. For a $\boldsymbol{v}_i$ when a pair of values is pushed onto the stack $index = i$ and r_1 identifies the starting position of $\boldsymbol{v}_i$ in $\mathcal{SA}$. When we reference the top of the stack we write $index = top_i$ and $r_1 = top_{r1}$.

We push the index i onto the stack for each increase in the $\mathcal{LCP}$ array values (i.e. $\mathcal{LCP}[top_i] < \mathcal{LCP}[i]$). We pop the stack when the $\mathcal{LCP}$ value decreases (i.e.

Algorithm 2 OLP-2: Compute $\mathcal{OLP}$ Using $\mathcal{LCP}$ Traversal

▷ *Note: arrays begin from 0 rather than 1, so $\mathcal{SA}$ values are decremented by one.*

```
 1: procedure COMPUTE_OLP()
 2:     Initialize arrays OLP and SA* of size n full of 0's.
 3:     for i from 1 to n − 1 do
 4:         v_i ← x[SA[i]..SA[i] + LCP[i] − 1]
 5:         if (RSF[i] > 1
 6:              and ((LCP[i] = 2 and v_i[0] = v_i[1]) or LCP[i] > 2)
 7:              and MAXBORDER(v_i))  then
                        ▷ Modified MAXBORDER(v_i) returns the longest border length for v_i [18].
 8:             i' ← i
 9:             while (LCP[i' − 1] ≥ LCP[i]) do
10:                 i' ← i' − 1
11:             r_1 ← i' − 1
12:             r_m ← r_1 + RSF[i] − 1
13:             Copy SA[r_1..r_m] to SA*[r_1..r_m]
14:             Sort SA*[r_1..r_m] in ascending order
15:             sum ← 0
16:             for k from 1 to RSF[i] − 1 do
17:                 diff ← SA*[r_1 + k] − SA*[r_1 + k − 1]
18:                 if (diff < LCP[i]) then
19:                     sum ← sum + LCP[i] − diff
20:             OLP[i] ← sum
21:     return OLP
```

$\mathcal{LCP}[top_i] > \mathcal{LCP}[i]$), which means we found r_m for the current NRE substring, $\boldsymbol{v}_i$. When we pop the stack, we compute the total overlap using the ProcessStack() procedure, which is a near replica of OLP-2. We also process any remaining values on the stack at the final index using ProcessStack().

The ordered pair on the top of the stack (top_i, top_{r1}) represent the values for $\boldsymbol{v}_{top_i}$. When we pop (top_i, top_{r1}) of the stack, we know that v_i represented by $\mathcal{LCP}[i]$ is a prefix of $\boldsymbol{v}_{top_i}$. In which case, the r_1 value for $\boldsymbol{v}_i$ would at least be equal to the r_1 value of $\boldsymbol{v}_{top_i}$. We store this tentative value of r_1 in $prev_{r1}$. We stop popping elements from the stack when $\boldsymbol{v}_i$ is no longer a prefix of $\boldsymbol{v}_{top_i}$ (i.e. when the stack is empty or $LCP[top_i] < LCP[i]$)). In which case, $prev_{r1}$ stores the final r_1 value of $\boldsymbol{v}_i$.

OLP-2 requires $\mathcal{O}(n^2)$ time in the worst case. The outer ***for*** loop executes $n-2$ times. Thus, for each $\boldsymbol{v}_i$, we sort $\mathcal{SA}^*[r_1..r_m]$ using Radix Sort in $\mathcal{O}(mk)$ time, where $m = RSF[i]$, and $k = \log_2 q$ is the key length in bits, and q is the largest value in $SA^*[r_1..r_m]$, which provides an upper bound on the number of bits in each element of $\mathcal{SA}$. However, in practice, we can treat k as a constant limited by computing capabilities. In our experiments, $k = 64$, sufficient for strings of length $n \leq 10^{18}$. Therefore, in practice, the time complexity may be treated as $\mathcal{O}(n^2)$.

Similarly, OLP-3 requires $\mathcal{O}(n^2)$ time. For each $\boldsymbol{v}_i$, we call ProcessStack() at most n times. This procedure also sorts the values in $\mathcal{SA}^*[r_1..r_m]$ for each $\boldsymbol{v}_i$ in $\mathcal{O}(mk)$ time, again yielding overall time complexity $\mathcal{O}(n^2)$.

Algorithm 3 OLP-3: Compute $\mathcal{OLP}$ using Stack.

```
                    ▷ Note: arrays begin from 0 rather than 1, so SA values are decremented by one.
 1: procedure COMPUTE_OLP()
 2:     Initialize arrays OLP and SA* of size n full of 0's.
 3:     Declare a stack of ordered pairs (index, r1) for a v_i     ▷ stack.top returns (top_i, top_r1)
 4:     push(1, 0)
 5:     prev_r1 ← 0
 6:     for i from 2 to n do
 7:         if (LCP[top_i] < LCP[i]) then
 8:             push(i, i − 1)
 9:         else if (LCP[top_i] > LCP[i]) then
10:             while (stack ≠ empty and LCP[top_i] > LCP[i]) do
11:                 prev_r1 ← top_r1
12:                 PROCESSSTACK()                              ▷ A near replica of OLP-2 [11].
13:                 pop()
14:             if (stack = empty or LCP[top_i] < LCP[i]) then
15:                 push(i, prev_r1)
16:     while (stack ≠ empty) do
17:         PROCESSSTACK()                                      ▷ A near replica of OLP-2 [11].
18:         pop()
19:     return OLP
```

```
procedure PROCESSSTACK()
    v_top_i ← input[SA[top_i]..SA[top_i] + LCP[top_i] − 1]      ▷ stack.top returns (top_i, top_r1)
    if (RSF[top_i] > 1)
        and (LCP[top_i] > 2 or (LCP[top_i] = 2 and v_top_i[0] = v_top_i[1]))
        and MAXBORDER(v_top_i, LCP[top_i]))  then
        r1 ← top_r1
        rm ← r1 + RSF[top_i] − 1
        Copy SA[r1..rm] to SA*[r1..rm]
        Sort SA*[r1..rm] in ascending order
        sum ← 0
        for k from 1 to RSF[top_i] do
            diff ← SA*[r1 + k] − SA*[r1 + k − 1]
            if (diff < LCP[top_i]) then
                sum ← sum + LCP[top_i] − diff
        OLP[top_i] ← sum
```

Figure 3: PROCESSSTACK() procedure to process the pairs $(index, r_1)$ on the stack. A near replica of OLP-2.

5 Comparison of OLP algorithms

All algorithms were developed in C++ and run on a Microsoft Windows 10 Pro machine with Intel Core i9-10980XE CPU @3.00 GHz (3000 MHz, 18 Cores, 36 Logical Processors) and CORSAIR Vengeance RGB PRO 128GB (4x32GB) DDR4 3600 (PC4-28800) RAM. Testing was performed on an Ubuntu Virtual Machine (Oracle VM VirtualBox Manager) with 81804 MB Base Memory and 18 Processors.

5.1 Random Strings

We created test data of 54 unique randomly generated strings. Although ideally, an average over multiple tests per string type would be performed, only one test was performed per string because of the time it took to compute *(see analysis below)*.

This data varied based on 18 string lengths: half in the thousands ($|\boldsymbol{x}| \in \{1000, .., 9000\}$) and the other half in the millions ($|\boldsymbol{x}| \in \{1M, .., 9M\}$). These datasets also varied in alphabet size ($|\Sigma| = \{2, 3, 4\}$) to simulate binary strings, triples, and DNA strings.

The execution time of OLP-1 was significantly longer than the improved algorithms (OLP-2 and OLP-3). For instance, for $|\boldsymbol{x}| \in \{1000, .., 9000\}$, OLP-1 took up to 5 seconds to compute. In contrast, the improved algorithms (OLP-2 and OLP-3) took no more than 0.03 seconds. For $|\boldsymbol{x}| \in \{1M, .., 9M\}$, OLP-1 took too long to complete computation. In particular, for 9 million long strings the execution time was $\sim$ 55 hours $\approx$ 2.3 days. As a result, its computation has several missing data points to compare with the improved algorithms. In contrast, for 9 million long strings, OLP-2 and OLP-3 took under a minute (maximum time taken was $\sim$ 45.7 seconds) to execute.

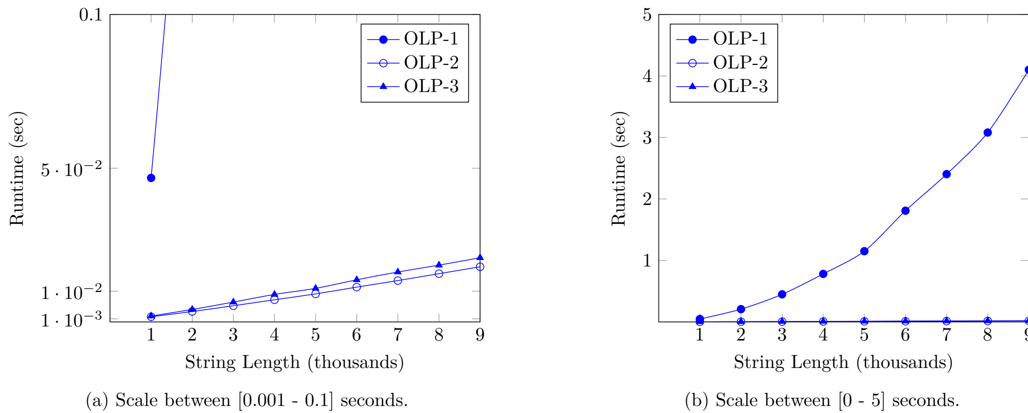

Figure 4: Scaled charts (a) zoomed in; and (b) zoomed out; of runtime comparison to compute maximal covers using OLP-1, OLP-2, or OLP-3 on random binary strings. See [11] for more charts and data tables.

As noted earlier, although all three algorithms have quadratic running times, the overall overhead for OLP-2 and OLP-3 is significantly reduced due to simplified algorithm structure and fewer and simpler data structures — thus resulting in linear execution time for very long strings in practice. This is most clearly seen when comparing the algorithms on random strings over a binary alphabet $|\Sigma| = 2$ (see Figure 4).

It might be assumed that OLP-3 would perform better in practice than OLP-2 due to the use of a stack to reduce duplication in the computation of r_1. However, this was not the case for the test data used. OLP-2 performed only marginally better – most clearly seen when the string lengths are 9M. This is most likely due to the overhead required to set up the stack and to implement the push and pop routines.

Notice also that the larger the alphabet size, the faster the OLP-2 and OLP-3 algorithms perform. This is because smaller alphabet sizes are more likely to result in highly repetitive strings, which require more processing to compute overlaps.

5.2 Fibonacci Strings

We generated the Fibonacci strings using $F_0 = b$, $F_1 = a$, $F_2 = ab$, $F_k = F_{k-1}F_{k-2}$. These strings ranged in length from 3 (F_3) to $121,393$ (F_{25}). We excluded the first four Fibonacci strings as they do not contain any repetitions. We stopped at F_{25} due to space limitations.

The experiments show that OLP-1 takes an order-of-magnitude longer to execute: nearly 9 hours to compute the maximal covers for F_{25}. In contrast, OLP-2 and OLP-3 take only 37 and 32 seconds, respectively.

Nevertheless, in Figure 5 we see that all three perform quadratically on Fibonacci strings, which are worst-case strings due to their highly repetitive structure. Interestingly, we see that OLP-2 performs slightly faster than OLP-3 on shorter Fibonacci strings (F_{3-15}), but slightly slower on longer Fibonacci strings (F_{16-25}). This is because:

1. OLP-2 traverses the $\mathcal{LCP}$ array backwards multiple times to determine r_1 — unlike OLP-3, which uses a stack to keep track of r_1 values during traversal, thereby eliminating duplicate computation;
2. OLP-2 performs $|x| = n$ computations of the border for v_i, whereas OLP-3 only does so when we pop off the stack, i.e. when there is a fall in $\mathcal{LCP}$ values.

We conclude that OLP-3 will be faster than OLP-2 on highly repetitive strings.

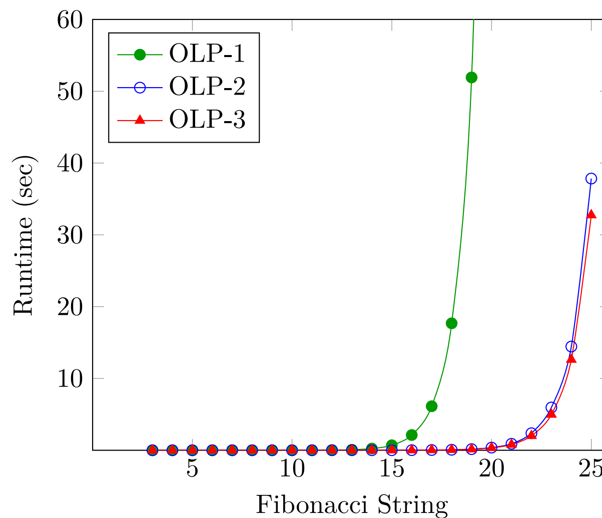

Figure 5: Plot of total runtime (seconds) when computing maximal covers of Fibonacci strings using OLP-1, OLP-2, or OLP-3.

5.3 Protein Sequences

We acquired FASTA files of protein sequence datasets from the NCBI (National Center for Biotechnology Information) databases of four taxonomically distinct model organisms: (1) *Arabidopsis thaliana* (thale cress plant), (2) *Caenorhabditis elegans* (roundworm), (3) *Drosophila melanogaster* (common fruit fly), and (4) *Homo sapiens* (human). Each protein dataset contained several thousand protein sequences (See Table 1) ranging in length from 19 to ~36k amino acids.

Note that some amino acid letters represented are indeterminate, i.e. they could represent multiple amino acids. For example, ‘Z’ represents either glutamine (‘Q’) or

glutamate ('E'). In addition, 'X' represents 'all proteins' or gaps. For simplicity, we treated them as regular strings where each letter is treated distinctly. In future work, this analysis can be improved by addressing the ambiguous nature of these letters for better matching.

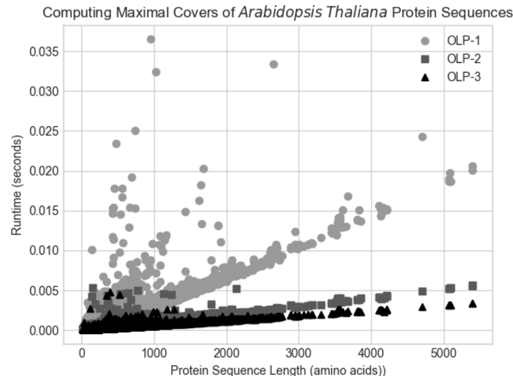

Figure 6: Runtime in seconds taken to compute maximal covers of *Arabidopsis Thaliana* protein sequences using OLP-1, OLP-2, or OLP-3 algorithms.

Species	No. of protein seq.	Protein seq. Length	Total Runtime in Seconds		
			OLP-1	OLP-2	OLP-3
Arabidopsis	~48K	19 - 5,399	64.2 s	22.3 s	16.5 s
C. elegans	~28K	19 - 15,187	47.4 s	14.5 s	10.8 s
D. melanogaster	~30K	20 - 22,948	73.7 s	23.6 s	15.3 s
human	~116K	23 - 35,990	246.3 s	79.7 s	56.6 s

Table 1: Total runtime (seconds) when computing maximal covers on sets of protein sequences of 4 different species using OLP-1, OLP-2, or OLP-3.

We computed maximal covers for each protein sequence within each set dedicated to a particular species, using OLP-1, OLP-2 and OLP-3. The algorithms perform in linear time when there are few repetitions, as shown in Fig. 6. However, when the protein sequence contains large repeats, they are processed quadratically. The outliers shown in Fig. 6 are examples of protein sequences with periodicity, for which OLP-1 took longer to compute maximal covers, unlike OLP-2 and OLP-3, which remained relatively close to linear time computation.

From Table 1, we see that OLP-3 was the fastest in each case. This is likely due to the stack data structure that better handles highly repetitive strings, as also shown by the results for Fibonacci Strings.

6 Conclusion

We conclude that OLP-2 and OLP-3 enable us to compute maximal covers in linear time for random strings and biological sequences, treated as regular strings rather than indeterminate. However, all OLP algorithms perform in quadratic time for highly repetitive strings, with OLP-2 and OLP-3 still significantly faster than OLP-1.

In the future, we plan to further improve the performance of OLP-3 by using the system stack instead of the program stack. Recently, Czajka & Radoszewski in [6]

gives an $\mathcal{O}(n \log n)$ implementation to compute maximal covers. We plan to compare the performance of our algorithms with this implementation soon. Moreover, it is worth considering that OLP-3 could potentially exhibit improved performance when utilizing a static stack instead of a dynamic stack. Therefore, conducting additional experiments to compare the two approaches would be beneficial.

Furthermore, we can conduct additional experiments to obtain an average by performing multiple tests for each string type. Additionally, when evaluating maximal covers, we can improve experiments involving protein sequences by incorporating considerations for indeterminate string matching.

An immediate question arises whether maximal cover computation of a string $\boldsymbol{x}$ (at least those on small σ) might be an interesting compression technique. However, our experimental evaluations show that several classes of strings have very short maximal covers (e.g. $|u| \leq 2$). This indicates that maximal covers may not be useful as a compression technique.

Another interesting question arises whether computing an *iterated* maximal cover set — that is, a set of covers covering the string $\boldsymbol{x}$ — might be useful as a compression technique. We plan to investigate this in future work.

7 Acknowledgements

The authors thank Viktor Melnyk for his contributions to this paper. In particular, for setting up the software to compute maximal covers using OLP-1, which included the incorporation of the $\mathcal{SA}/\mathcal{LCP}$ software, implementation of the optimal cover algorithm, including the implementation of the OLP-1 algorithm, and preliminary testing of its effectiveness.

References

1. A. Alatabbi, A. S. M. S. Islam, M. S. Rahman, J. Simpson, and W. F. Smyth: *Enhanced covers of regular & indeterminate strings using prefix tables.* J. Automata, Languages & Combinatorics, 21(3) 2016, pp. 131–147.
2. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasi–periodicities in strings*, Tech. Rep. 90.5, The Leonadro Fibonacci Institute, Trento, Italy, 1990.
3. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings.* Theoret. Comput. Sci., 119(2) 1993, pp. 247–265.
4. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "runs" theorem.* SIAM J. Comput., 46(5) 2017, pp. 1501–1514.
5. R. Cole, C. S. Iliopoulos, M. Mohamed, W. F. Smyth, and L. Yang: *The complexity of the minimum k-cover problem.* J. Automata, Languages & Combinatorics, 10-5/6 2005, pp. 641–653.
6. P. Czajka and J. Radoszewski: *Experimental evaluation of algorithms for computing quasiperiods.* Theoretical Computer Science, 854 2021, pp. 17–29.
7. T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. F. Smyth, and W. Tyczyński: *Enhanced string covering.* Theoretical Computer Science, 506 2013, pp. 102–114.
8. G. B. Golding, H. Koponen, N. Mhaskar, and W. F. Smyth: *Computing maximal covers for protein sequences.* Journal of Computational Biology, 30(2) 2023, pp. 149–160.
9. C. S. Iliopoulos and W. F. Smyth: *On-line algorithms for k-covering*, in Proc. 9th Australasian Workshop on Combinatorial Algs. (AWOCA), 1998, pp. 97–106.
10. T. Kociumaka, J. Radoszewski, W. Rytter, S. P. Pissis, and T. Waleń: *Fast algorithm for partial covers in words.* Algorithmica, 73(1) 2015, pp. 217 – 233.

11. H. Koponen: *Efficient implementation & application of maximal string covering algorithms.* MSc Thesis, McMaster University, 2022, p. 58.
12. Y. Li and W. F. Smyth: *Computing the cover array in linear time.* Algorithmica, 32(1) 2002, pp. 95–106.
13. N. Mhaskar and W. F. Smyth: *Frequency covers for strings.* Fundamenta Informaticae, 163(3) 2018, pp. 275–289.
14. N. Mhaskar and W. F. Smyth: *String covering with optimal covers.* Journal of Discrete Algorithms, 51 2018, pp. 26–38.
15. N. Mhaskar and W. F. Smyth: *String covering: A survey.* submitted for publication, 2022.
16. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced–sorting.* Data Compression Conference, 0 2009, pp. 193–202.
17. S. J. Puglisi and A. Turpin: *Spacetime tradeoffs for longest-common-prefix array computation.* Proc. 19th Internat. Symp. Algs. & Comp., 2008, pp. 124–135.
18. B. Smyth: *Computing Patterns in Strings*, Pearson/Addison–Wesley, 2003.