Computing Maximal Covers for Protein Sequences

*G. Brian Golding[1a], Holly Koponen[2b],

Neerja Mhaskar[2c†], W. F. Smyth[2d]

[1]Department of Biology, McMaster University, Canada

[2]Department of Computing and Software, McMaster University, Canada

E-mail: [a]golding@mcmaster.ca, [b]koponeh@mcmaster.ca,

[c†]pophlin@mcmaster.ca, [d]smyth@mcmaster.ca.

January 26, 2023

Keywords: string covers, repeats, MUMmer, MAXCOVER, protein.

Abstract: A ***partial cover*** of a ***string*** or ***sequence*** of length n, which we model as an array $\boldsymbol{x} = \boldsymbol{x}[1..n]$, is a repeating substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that "many" positions in $\boldsymbol{x}$ lie within occurrences of $\boldsymbol{u}$. A ***maximal cover u**** — introduced in (Mhaskar and Smyth, 2018b) as ***optimal cover*** — is a partial cover that, over all partial covers $\boldsymbol{u}$, maximizes the positions covered. Applying data structures introduced in (Mhaskar and Smyth, 2018b), our software MAXCOVER for the first time enables efficient computation of $\boldsymbol{u}^*$ for any $\boldsymbol{x}$ – in particular, as described here, for protein sequences of Arabidopsis,

*Authors in alphabetical order by surname.

†To whom correspondence should be addressed

C. elegans, *D. melanogaster* and humans. In this protein context, we also compare an extended version of MAXCOVER to existing software (MUMmer's `repeat-match`) for the closely related task of computing non-extendible repeating substrings (a.k.a. *maximal repeats*). In practice, MAXCOVER is an order-of-magnitude faster than MUMmer, with much lower space requirements, while producing more compact output that nevertheless yields a more exact and user-friendly specification of the repeats.

1 Introduction

As introduced in (Apostolico and Ehrenfeucht, 1990, 1993), 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 $\boldsymbol{u}$. For example, $\boldsymbol{u} = aba$ is a cover of $\boldsymbol{x} = ababaaba$. Even though all the covers of every prefix of $\boldsymbol{x}$ are computable in $\mathcal{O}(n)$ time (Li and Smyth, 2002), nevertheless, since very few strings possess a cover, related generalizations of cover have been proposed that also yield a compressed representation of $\boldsymbol{x}$ and are more useful in practice: such as the k-cover (Iliopoulos and Smyth, 1998), α-partial cover (Kociumaka et al., 2015), frequency cover (Mhaskar and Smyth, 2018a), and enhanced cover (Flouri et al., 2013, Alatabbi et al., 2016). The survey (Mhaskar and Smyth, 2021) provides further detail.

We are particularly interested in a ***maximal cover*** (a.k.a. ***optimal cover***) (Mhaskar and Smyth, 2018b), a substring $\boldsymbol{u}$ of $\boldsymbol{x}$ that occurs at least twice and that, over all such substrings, covers a maximum number of positions — this computation requires $\mathcal{O}(n \log n)$ time and $\Theta(n)$ space.

Thus, in general, covers provide more information about a string and its structure than repeats. We can therefore use covers to study sequence structures

and compression, while requiring less storage/processing. However, covers are hard to compute. Hence, genome-wide analyses of repeats have been limited largely due to the difficulty of identifying maximal covers on a large scale. In this paper we implement and describe MAXCOVER, the first software to compute maximal covers for a given string, then apply it to compression of numerous protein sequences.

Section 2 of this paper introduces the required terminology and symbolism. In Section 3 we present a simple outline of the Maximal Cover (MC) algorithm[1], describing its implementation on arbitrary strings. Section 4 describes the application of MC to protein sequences, in particular those in which very long covers are identified. Then in Section 5 we display the efficiency and adaptability of our software by comparing a modified version of MAXCOVER that computes ***non-extendible repeating substrings*** with existing software (**MUMmer**) for the same task. Along with other advantages, MAXCOVER executes an order-of-magnitude (typically 20+ times) faster. Section 6 outlines future work.

2 Preliminaries

A ***string*** is a finite array $\boldsymbol{x} = \boldsymbol{x}[1..n]$ of ***letters*** chosen from a totally ordered set Σ, called an ***alphabet***, of cardinality $\sigma = |\Sigma|$. The ***length*** of $\boldsymbol{x}$ is $|\boldsymbol{x}| = n$. The ***empty string*** ε is a string of length zero. The string $\boldsymbol{x}[i..j]$ is a ***substring*** of $\boldsymbol{x}[1..n]$ iff $1 \leq i \leq j \leq n$, a ***proper substring*** iff $j - i + 1 < n$, a ***prefix*** of $\boldsymbol{x}$ for $i = 1$, a ***suffix*** for $j = n$. The suffix starting at index i is called ***suffix*** i. A ***border*** of $\boldsymbol{x}$ is a proper substring of $\boldsymbol{x}$ that is both prefix and suffix; thus, every string has the empty border ε. For example, $\boldsymbol{x} = ababaaba$ has borders aba, a and ε.

[1]The only algorithm currently implemented to compute the maximal cover of given $\boldsymbol{x}$.

The ***frequency*** $f_{x,u}$ of a substring u in x is the number of times u occurs in x; if $f_{x,u} > 1$, then u is said to be a ***repeating substring*** in x. Let u_1 and u_2 be occurrences of u in x; if u has one or more nonempty borders, then it may happen that u_1 and u_2 ***overlap***. For example, *abcabcab* is an overlap of 2 for $u_1 = u_2 = abcab$ resulting from the border *ab* of length 2.

A repeating substring u of w is ***left (right) extendible*** if every instance of u in w is preceded (followed) by the same symbol, otherwise it is ***non-left (non-right) extendible***. For example, in the string $w = ccabaacabaac$, 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 w is said to be ***non-extendible*** (NE, a.k.a. *maximal repeat*)[2] if and only if it is non-left and non-right extendible . For example in $w = ccabaacabaac$, the repeating substring *cabaac* is non-extendible.

A ***repetition*** is a string of the form $x = u^e$, where $e \geq 2$. For example, $x = abaaba = (aba)^2$ is a repetition. If w is not a repetition, it is said to be ***primitive***.

A string x is an ***overlapping string*** if $x = utv$, and $ut = tv$, and $u, t, v \neq \varepsilon$.

Each position in x that occurs within a repeating substring u of x is said to be ***covered*** by u. Hence:

- Let M denote the maximum number of positions in w covered by any repeating substring of w. Then a longest (shortest) ***maximal cover*** u is a longest (shortest) repeating substring of w that covers M positions.

This definition identifies both a longest and a shortest substring because both may be of interest: the longest could provide more information about the structure of w, while the shortest is more compact. In this paper we consider

[2]This definition is commonly referred to in literature as *'maximal repeat'* (Gusfield et al., 1997), but for the sake of clarity we use the term *'NE repeat'* (Smyth, 2003), to avoid confusion with *'maximal cover'*.

the longest throughout; as noted in Section 3, the change to shortest is trivial.

Algorithm MC depends on several important integer arrays of length n computed from the given string $\boldsymbol{x}[1..n]$, of which the first two, both computable in $\mathcal{O}(n)$ time, are well known:

- In the ***suffix array*** $\mathcal{SA}_{\boldsymbol{x}}$, $\mathcal{SA}_{\boldsymbol{x}}[i]$ is the starting position of the i-th lexicographically least suffix (that is, i-th in dictionary order) in $\boldsymbol{x}$.

- In the ***longest common prefix array*** $\mathcal{LCP}_{\boldsymbol{x}}$, $\mathcal{LCP}_{\boldsymbol{x}}[1] = 0$ and $\mathcal{LCP}_{\boldsymbol{x}}[i]$, $1 < i \leq n$, is the length of the longest common prefix of the suffixes of $\boldsymbol{x}$ starting at $\mathcal{SA}_{\boldsymbol{x}}[i-1]$ and $\mathcal{SA}_{\boldsymbol{x}}[i]$.

Three others, the first and last also computable in linear time, the second, $\mathcal{OLP}$, in time $\mathcal{O}(n \log n)$[3], were more recently introduced:

- In the ***repeating substring frequency array*** $\mathcal{RSF}_{\boldsymbol{x}}$ (Mhaskar and Smyth, 2018a), $\mathcal{RSF}[i]$ is the frequency in $\boldsymbol{x}$ of the substring of length $\mathcal{LCP}[i]$ starting at index $\mathcal{SA}[i]$ in $\boldsymbol{x}$; that is, the substring $\boldsymbol{x}[SA[i] \ldots \mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$. Note that for $\mathcal{LCP}[i] = 0$, $\mathcal{RSF}[i] = 0$.

- In the ***overlapping positions array*** $\mathcal{OLP}_{\boldsymbol{x}}$ (Mhaskar and Smyth, 2018b), $\mathcal{OLP}[i]$ is the total number of positions that are overlapping (*'total overlap'*) over all occurrences of the substring $\boldsymbol{u} = \boldsymbol{x}[\mathcal{SA}[i]..\mathcal{SA}[i] + \mathcal{LCP}[i] - 1]$ in $\boldsymbol{x}$. Note that for $\mathcal{LCP}[i] = 0$, $\mathcal{OLP}[i] = 0$.

- In the ***repeating substring positions covered array*** $\mathcal{RSPC}_{\boldsymbol{x}}$ (Mhaskar and Smyth, 2018b), $\mathcal{RSPC}[i]$ is the number of positions covered by the repeating substring $\boldsymbol{u}$ of length $\mathcal{LCP}[i]$ that occurs at $\mathcal{SA}[i]$; zero otherwise. Note that the value of $\mathcal{RSPC}[i]$ depends upon prior computation of $\mathcal{OLP}[i]$.

[3]See the discussion of algorithmic complexity vs. execution time at the end of Section 3.

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

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

We illustrate these arrays in Figure 1 using the invented protein $\boldsymbol{x} = ADAQADADAQADAQADA$, identical in structure to the example *abacababacabacaba* in (Mhaskar and Smyth, 2018b).

To understand this better, consider position $i = 3$:

- $\mathcal{SA}[2] = 15$, refers to the suffix $\boldsymbol{x}[15..17] = ADA$
- $\mathcal{SA}[3] = 5$, refers to the suffix $\boldsymbol{x}[5..17] = ADADAQADAQADA$.
- $\mathcal{LCP}[3] = 3$, tells us that the longest common prefix of $\boldsymbol{x}[15..17]$ and $\boldsymbol{x}[5..17]$ is of length 3, thus ADA.
- $\mathcal{RSF}[3] = 5$, the number of occurrences of ADA in $\boldsymbol{x}$.
- $\mathcal{OLP}[3] = 1$, the total overlap among all 5 occurrences of ADA in $\boldsymbol{x}$.
- $\mathcal{RSPC}[3] = (3 * 5) - 1 = 15 - 1 = 14$, the total number of positions covered by all occurrences of ADA in $\boldsymbol{x}$.

3 Computing Maximal Covers with MAXCOVER

In this section we present the very simple pseudocode of MAXCOVER: Algorithm MC operating on a given string $\boldsymbol{x}$ — in fact, this algorithm processes only the $\mathcal{SA}$, $\mathcal{LCP}$ and $\mathcal{RSPC}$ arrays that have already been computed from $\boldsymbol{x}$ in a preprocessing phase. It computes *all* of the maximal covers: there may in an extreme (and uninteresting) case be $\mathcal{O}(n)$ of them.

Algorithm 1: Computing all Maximal Covers of a String (Based on Algorithm 4 from (Mhaskar and Smyth, 2018b))

```
max_pc ← 1;
max_cover_length ← 0;
i ← 1;
MCList ← Empty;
while i ≤ n do
    if (RSPC[i] > 1) then
        if (RSPC[i] > max_pc) then
            max_pc = RSPC[i];
            max_cover_length = LCP[i];
            MCList ← Empty;
            Add i to MCList;
        end
        if (RSPC[i] = max_pc) then
            if (max_cover_length = LCP[i]) then
                Add i to MCList;
            end
            if (max_cover_length < LCP[i]) then
                max_cover_length ← LCP[i];
                MCList ← Empty;
                Add i to MCList;
            end
        end
    end
    i ← i + 1;
end
Output MCList ;
```

The algorithm scans the $\mathcal{RSPC}$ array from left to right. During this scan it maintains local maximal covers in a list called $MCList$. At the end of the scan

all the maximal covers for $\boldsymbol{x}$ are stored in $MCList$, which can in constant time be updated by a new entry or emptied entirely using an integer pointer. Thus the **while** loop executes n times with constant time for each execution, and so Algorithm 1 runs in $\mathcal{O}(n)$ time. Since it uses only the $\mathcal{SA}, \mathcal{LCP}, \mathcal{RSPC}$ and $MCList$ arrays, altogether occupying $16n$ bytes, it also requires $\mathcal{O}(n)$ space. Therefore, we state:

Theorem 1 ((Mhaskar and Smyth, 2018b)). *All maximal covers of a string $\boldsymbol{w} = \boldsymbol{w}[1..n]$ can be computed in $\mathcal{O}(n)$ time and space given the precomputed $\mathcal{RSPC}$ array.*

Based on the analysis given in (Mhaskar and Smyth, 2018b), we have claimed above that $\mathcal{OLP}$ is computable in $\mathcal{O}(n \log n)$ time, hence $\mathcal{RSPC}$ and maximal covers also require $\mathcal{O}(n \log n)$ time. However, the version of MAXCOVER tested here is based on an $\mathcal{O}(n^2)$ implementation. The reasons for this are as follows:

- The $\mathcal{O}(n \log n)$ implementation of $\mathcal{OLP}$ is complex and difficult.
- Over randomly-generated strings, the worst-case quadratic MAXCOVER executes in linear time on average.
- Moreover, we anticipate that, due to its avoidance of complex data structures, the quadratic version will in fact execute faster on average than the $\mathcal{O}(n \log n)$ version.

These results are in preparation for future publication.

Note, the *shortest* maximal cover for a given string $\boldsymbol{w}$ can be computed by trivially modifying Algorithm 1 to store the shortest substring (instead of the longest substring) covering the maximum number M of positions. To our knowledge, MAXCOVER is the only software currently available to compute the maximal cover of given $\boldsymbol{x}$.

4 Applying MAXCOVER to Protein Sequences

In this section we describe the application of MAXCOVER (Algorithm 1) to the identification of repeats in protein sequences.

4.1 Repeats in Proteins

Repeating amino acids in protein sequences are a common cause of variations in protein length. These repeats are often functional domain repeats that provide organisms with the opportunity to develop and adapt new functions within the domain. Knowledge of such repeats are necessary for computing alignments, to understand diseases, and for an understanding of protein function and evolution. Alignments of protein sequences are extremely difficult to do correctly across repeats since an accurate alignment can only be determined when the repeats differ in some substitutions. Some repeats are the cause of human diseases. For example, Huntington's disease is caused by the expansion of CAG repeats. Repeats are also critically important for protein evolution. Most large proteins are caused by the duplication of smaller segments. The duplication often copies a functional domain, creating two copies of this domain. While one copy carries out the original function, the second copy is free to adapt to a new altered function.

In large proteins such as Titin (see below), the protein has evolved by the hierarchical duplication of repeats, by sequence divergence of these repeats and then followed by more cycles of duplication and divergence. Finding the maximal cover within such proteins is challenging.

```
1 string of length 17
2
3 Optimal Covers:
4         ADAQADA
5         OC length=7
6         number of positions covered=17
7         % covered = 100%
8         time elapsed 0.000307 seconds
9
```

Figure 2: MAXCOVER output to compute the maximal cover of protein $\boldsymbol{x}$ = ADAQADADAQADAQADA.

4.2 MAXCOVER computing Maximal Covers

While computing the maximal covers, MAXCOVER outputs the length of the string, a maximal cover $\boldsymbol{u^*}$, the length of $\boldsymbol{u^*}$, the number of positions $\boldsymbol{u^*}$ covers in $\boldsymbol{x}$, the percentage of positions $\boldsymbol{u^*}$ covers in $\boldsymbol{x}$, and the time elapsed to compute $\boldsymbol{u^*}$. See Figure 2 for a simple example.

4.3 Data

We acquired protein data from the NCBI (National Center for Biotechnology Information) databases. We chose proteins from four taxonomically distinct model organisms and collected the entire complement of proteins from these organisms. We used FASTA files of the protein sequences for four species: *Arabidopsis thaliana* (48,265 protein sequences), *Caenorhabditis elegans* (28,350 protein sequences), *Drosophila melanogaster* (30,717 protein sequences), and *Homo sapiens* (116,263 protein sequences). To ensure good quality sequences, proteins that contained unknown amino acids (X's) were removed from the dataset. The proteins processed ranged in size from 20 to 100,000 amino acids.

To provide a good representative sample and to avoid excessive run times, we sampled 10,000 protein sequences per species, thus altogether 40,000. This ensured that the time required for testing was less than thirty minutes. The sequences were chosen randomly using the Python `random.sample` function[4].

[4]https://docs.python.org/3/library/random.html

4.4 Experimental Results

Representative examples of maximal covers for each of the four taxa are shown in Tables 1 – 4 (see Appendix 6), which use ***amino acids*** (aa) as the unit of measure for the length of the strings. As expected *C. elegans* had the least extensive repetitive proteome while *Homo sapiens* had the most extensive repetitive proteome. Nevertheless, a maximal cover of length 33 is a highly repeated substring within the *C. elegans* proteome, being repeated 83 times. It can be seen that within *C. elegans* the maximum number of positions covered occurs at an intermediate cover length (30-39aa) and the maximum percent coverage is also at an intermediate length (30-39aa). With Arabidopsis again the maximum number of positions covered occurs at an intermediate cover length (20-29aa) and the maximum percent coverage at 70-79aa. For *D. melanogaster* the maximum number of positions covered is for a cover of 50-59aa and the maximum percent coverage is at 400-699. For humans, both of these occur with large covers of length 700-799aa and 600-699aa. A possible reason for this discrepancy is that with the large population size of *C. elegans* only smaller protein repeats can exist before mutations destroy their similarity but within the historically much smaller human population large repeats can survive due to higher amounts of random genetic drift fixing the repeats before mutations can disrupt them. This hypothesis could be tested by more extensive applications of MAXCOVER and carefully chosen data. For instance, a comparative data sampling could be chosen where the population genetics is better understood.

5 Performance Evaluation of MAXCOVER

Since, as noted above, MAXCOVER is currently the only software available for the computation of maximal covers, we needed to find another basis for evaluating its efficiency. To do this, we made use of the fact that NE repeats and maximal covers are both computed using identical data structures: we made minor extensions to MAXCOVER so that it also computed NE repeating substrings.

We were thus able to test MAXCOVER against an existing alignment tool MUMmer (Marcais et al., 2018), that computes NE repeats in order to identify a minimum match length for a given genome. Although the primary software functions are different — MAXCOVER computes maximal covers whereas MUMmer is an alignment tool — we can assess MAXCOVER's performance by comparing its computation of NE repeats to that of MUMmer.

5.1 Machine Specifications

The MAXCOVER software was developed in C++ and run on a Microsoft Windows 10 Pro (10.0.19042 Build 19042) machine with Intel(R) Core(TM) i9-10980XE CPU @3.00GHz (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 81804MB Base Memory and 18 CPUs. The implementation used for $\mathcal{SA}$ array is by Yuta Mori[5] (Nong et al., 2011), $\mathcal{LCP}$ array by Simon Puglisi (Puglisi and Turpin, 2008), and $\mathcal{RSF}$ array by Neerja Mhaskar (Mhaskar and Smyth, 2018a). The software was made available on Github[6] as an open-source software.

[5] https://sites.google.com/site/yuta256/
[6] https://github.com/hollykoponen/MAXCOVER

```
1 Long Exact Matches:
2    Start1     Start2    Length
3         7         11         7
4         1         11         7
5         1          7         7
6         7         15         3
7         1         15         3
8         5          7         3
9         1          5         3
10       15         17         1
11        5         17         1
12       11         17         1
13        1         17         1
14       13         15         1
15        5         13         1
16       11         13         1
17        1         13         1
18        9         15         1
19        5          9         1
20        9         11         1
21        1          9         1
22        3         15         1
23        3          5         1
24        3         11         1
25        1          3         1
```

Figure 3: MUMmer's `repeat-match` output to compute the NE repeat of $\boldsymbol{x}$= ADAQADADAQADAQADA.

5.2 MUMmer's `repeat-match`

The program `repeat-match` from MUMmer uses suffix-trees to identify maximal exact repeat regions of a given minimum match length for a given genome. (See Figure 3.) The output of `repeat-match` has three columns: Start1 and Start2 specify the starting positions of exact matching region pairs, while the last column gives the length of the region. Note that if the given minimum match length was '2' then only lines 1-9 would be returned.

We discovered that `repeat-match` does not report redundancies consistently. In Figure 3 lines 3-4 report the start positions {1, 7, 11} of length 7, while in line 5, positions {1, 7} are redundantly repeated. However, when we look at lines 6-9, the pair (5, 15) is not redundantly reported. Similarly, we know that at $\boldsymbol{x}[11..17]$ = ADAQADA, so we know $\boldsymbol{x}[11..13]$ = ADA, but position 11 is not reported between lines 6-9. Therefore, `repeat-match` does not consistently report redundancies and/or all repeat-matches.

```
1 Substring Repeat    Length    Position
2         1             7       11, 1, 7
3         2             3       15, 5, 11, 1, 7
4         3             1       17, 15, 5, 11, 1, 7, 13, 3, 9
```

Figure 4: MAXCOVER output to compute the NE repeat of the string $\boldsymbol{x}$ = ADAQADADAQADAQADA.

5.3 MAXCOVER computing NE repeating substrings

Figure 4 shows the corresponding output from MAXCOVER. It has a more condensed format: 'NE Substring Repeat #' is a label for the non-extendible substring region to differentiate between matching regions of the same length; 'Length' is the length of the substring region; and 'Position' is a list of the start positions of the substring regions in $\boldsymbol{x}$. This format consistently reports all occurrences and uses a condensed tabular format.

Titin is the longest protein known. It is composed of 363 exons which are joined together to form the complete protein. (Bang et al., 2001) It is a structural protein used in the formation of muscles. We applied MAXCOVER to identify the top ten longest non-extendable repeats in this protein. Despite being the longest protein and a highly repetitive protein, there are only a maximum of four repeats of length 37 amino acids within the top ten longest. There are more numerous repeats in much smaller proteins within *C. elegans* (see table 1). Given the highly repetitive nature of titin, the small number of repeats might indicate that sequence divergence has occurred among the repeats. This suggests that a future variant of the current software that includes approximate matches would be useful. The software might then identify many more long repeats within this protein.

Substring Repeat	Length	Position
1	138	12529, 12780
2	110	12418, 12669
3	108	12220, 12471
4	88	12329, 12580
5	87	12329, 12580, 12831
6	48	12171, 12422, 12673
7	37	12175, 12426, 12677, 13040
8	33	12411, 12997
9	29	12917, 12973
10	28	12417, 12919

Figure 5: MAXCOVER output to compute the top ten longest NE repeating substrings of NP_001254479.2 titin isoform IC [*Homo sapiens*].

5.4 Comparing MAXCOVER & MUMmer's `repeat-match`

We compare the output of MUMmer's `repeat-match` and of MAXCOVER in Table 5. `Repeat-match` took 29.5 minutes to compute the NE substring regions, whereas MAXCOVER required only 1.25 minutes – faster by a factor of more than 20. This difference in speed is no doubt largely due to the use of suffix trees by `repeat-match`, whereas MAXCOVER uses suffix arrays. In terms of accuracy, `repeat-match` does not report every pair of start positions for matching regions, whereas MAXCOVER reports all start positions for matching regions. As we have seen, the output of `repeat-match` is lengthy and makes it difficult to recognize useful information, whereas MAXCOVER is condensed and it is clear which set of positions refer to the same region.

6 Conclusions & Future Work

MAXCOVER is the only software that we are aware of that computes a maximal cover. We show that the MAXCOVER software, when compared to MUMmer's `repeat-match`, is an order-of-magnitude faster and has much lower space requirements (largely due to the construction of simpler data structures such as suffix arrays), produces a more compact output, and yields a more exact and user-friendly specification of the repeats.

Future development and application of MAXCOVER include the following:

- Improve the complexity of a maximal cover computation from $O(n^2)$ to $O(n \log n)$ based on the analysis in (Mhaskar and Smyth, 2018b);
- Extend MAXCOVER to accommodate approximate matching under edit distance, so as to consider approximate alignments in two or more sequences;
- Apply MAXCOVER to strings arising in contexts other than bioinformatics, such as information security or image analysis.

Acknowledgements

Authorship Contribution

G. Brian Golding: Conceptualization, Methodology, Formal analysis, Writing - Original Draft, Funding acquisition.

Holly Koponen: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data Curation, Writing - Original Draft, Visualization.

Neerja Mhaskar: Conceptualization, Methodology, Validation, Formal analysis, Writing - Original Draft, Supervision, Project administration, Funding acquisition.

W. F. Smyth: Conceptualization, Methodology, Formal analysis, Writing - Review & Editing, Supervision, Funding acquisition.

Authors' Disclosure

Not applicable as there is no conflict of interest.

Funding Statement

The first author was funded by the Natural Sciences & Engineering Research Council of Canada (NSERC) [Grant No. RGPIN-2015-04477], the second and fourth authors were funded by NSERC [Grant No. 10536797], and the third author was funded by the department of Computing and Software, McMaster University.

References

Alatabbi A, Islam ASMS, Rahman MS, et al. Enhanced Covers of Regular & Indeterminate Strings Using Prefix Tables. Journal of Automata, Languages & Combinatorics 2016;21(3):131–147.

Apostolico A. and Ehrenfeucht A. Efficient Detection of Quasiperiodicities in Strings. Theoretical Computer Science 1993;119(2):247–265.

Apostolico A. and Ehrenfeucht A. Efficient Detection of Quasi–Periodicities in Strings. The Leonadro Fibonacci Institute: Trento, Italy; 1990.

Bang ML, Centner T, Fornoff F, et al. The Complete Gene Sequence of Titin, Expression of an Unusual Approximately 700-KDa Titin Isoform, and Its Interaction with Obscurin Identify a Novel Z-Line to I-Band Linking System. Circ Res 2001;89:1065–1072; doi: 10.1161/hh2301.100981.

Cole R, Iliopoulos CS, Mohamed M, et al. The Complexity of the Minimum K-Cover Problem. Journal of Automata, Languages & Combinatorics 2005;10–5/6:641–653.

Flouri T, Iliopoulos CS, Kociumaka T, et al. Enhanced String Covering. Theoretical Computer Science 2013;506:102–114.

Gusfield D, Press CU and Fund JD& PSHL. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. EBL-Schweitzer. Cambridge University Press; 1997.

Iliopoulos CS and Smyth WF. On-Line Algorithms for k-Covering. In: Proc. Ninth Australas. Workshop Comb. Algorithms 1998; pp. 97–106.

Kociumaka T, Radoszewski J, Rytter W, et al. Fast Algorithm for Partial Covers in Words. Algorithmica 2015;73(1):217–233.

Li Y and Smyth WF. Computing the Cover Array in Linear Time. Algorithmica 2002;32(1):95–106.

Marcais G, Delcher AL, Phillippy AM, et al. MUMmer4: A Fast and Ver-

satile Genome Alignment System. PLoS Comput Biol 2018;14:e1005944; doi: 10.1371/journal.pcbi.1005944.

Mhaskar N and Smyth WF. Frequency Covers for Strings. Fundamenta Informaticae 2018a;161–3:275–289.

Mhaskar N and Smyth WF. String Covering with Optimal Covers. Journal of Discrete Algorithms 2018b;51:26–38.

Mhaskar N and Smyth WF. String Covering: A Survey. Submitted for publication 2021;20.

Nong G, Zhang S and Chan DWH. Two Efficient Algorithms for Linear Time Suffix Array Construction. IEEE Transactions on Computers 2011;60:1471–1484; doi: 10.1109/TC.2010.188.

Puglisi SJ and Turpin A. Space-Time Tradeoffs for Longest-Common-Prefix Array Computation. In: Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08) 2008.

Smyth WF. Computing Patterns in Strings. ACM Press Bks. Pearson/Addison-Wesley; 2003.

Appendix

Tables

Table 1: Examples of maximal covers for *C. elegans* proteins within a particular range of maximal cover lengths computed using MAXCOVER.

MC-length range	protein ID	protein length	MC-length	positions covered	% coverage	time to compute
1	NP_001254049.1_3963	117	1	14	11 %	0.311 ms
2	NP_001024113.1_1693	73	2	16	21 %	0.244 ms
3	NP_001255514.1_4822	43	3	6	13 %	0.267 ms
4	NP_001368117.1_12245	76	4	8	10 %	0.26 ms
5	NP_001309530.1_8612	98	5	10	10 %	0.292 ms
6	NP_496365.2_17882	98	6	18	18 %	0.362 ms
7	NP_001334228.1_8988	351	7	114	32 %	0.595 ms
8	NP_494447.1_16558	604	8	256	42 %	1.561 ms
9	NP_001337314.1_9082	80	9	18	22 %	0.283 ms
10 - 19	NP_506718.2_24803	74	13	26	35 %	0.381 ms
20 - 29	NP_001257068.1_6103	223	24	96	43 %	0.454 ms
30 - 39	NP_001350976.1_9553	3317	33	2739	82 %	16.846 ms
40 - 49	NP_001368148.1_12271	826	41	160	19 %	1.46 ms
50 - 59	NP_503527.1_22569	274	57	145	52 %	0.624 ms
60 - 69	NP_493059.2_15693	468	64	74	15 %	0.74 ms
70 - 79	NP_001033466.2_2263	257	75	95	36 %	0.511 ms
80 - 89	-	-	-	-	-	-
90 - 99	-	-	-	-	-	-
100 - 199	NP_494641.2_16683	677	105	168	24 %	1.341 ms
200 - 299	NP_494430.2_16543	626	219	339	54 %	1.299 ms
300 - 399	NP_493601.2_16025	4647	378	1134	24 %	12.724 ms
400 - 499	-	-	-	-	-	-
500 - 599	-	-	-	-	-	-
600 - 699	-	-	-	-	-	-
700 - 799	NP_001366805.1_11196	13083	768	1536	11 %	54.72 ms

[aa] Length of protein sequences and maximal covers are measured in amino acids (aa).
[ms] Time to compute measured in milliseconds (ms).

Table 2: Examples of maximal covers for *D. melanogaster* proteins within a particular range of maximal cover lengths computed using MAXCOVER.

MC-length range	protein ID	protein length	MC-length	positions covered	% coverage	time to compute
1	NP_001260924.1_8063	369	1	38	10 %	0.687 ms
2	NP_001285973.1_11292	43	2	6	13 %	0.252 ms
3	NP_001261881.2_8903	51	3	6	11 %	0.245 ms
4	NP_524256.1_15218	62	4	28	45 %	0.351 ms
5	NP_001285816.1_11135	66	5	10	15 %	0.25 ms
6	NP_001262537.1_9508	126	6	60	47 %	0.424 ms
7	NP_523815.2_14812	1334	7	172	12 %	2.876 ms
8	NP_609716.2_18399	78	8	24	30 %	0.361 ms
9	NP_572939.1_16848	185	9	59	31 %	0.488 ms
10 - 19	NP_729001.2_26993	241	14	92	38 %	0.458 ms
20 - 29	NP_476941.2_13802	111	25	33	29 %	0.421 ms
30 - 39	NP_650373.1_22567	173	33	66	38 %	0.478 ms
40 - 49	NP_572186.1_16259	407	44	176	43 %	0.75 ms
50 - 59	NP_610937.4_19403	4011	52	1404	35 %	11.068 ms
60 - 69	-	-	-	-	-	-
70 - 79	NP_729000.2_26992	414	79	158	38 %	0.742 ms
80 - 89	NP_647938.1_20661	431	88	221	51 %	0.75 ms
90 - 99	NP_572306.1_16355	300	97	173	57 %	0.592 ms
100 - 199	NP_996410.1_30113	1583	184	318	20 %	3.744 ms
200 - 299	-	-	-	-	-	-
300 - 399	-	-	-	-	-	-
400 - 499	NP_727078.1_26214	533	456	532	99 %	0.821 ms
500 - 599	-	-	-	-	-	-
600 - 699	NP_995994.1_29765	762	684	760	99 %	1.579 ms

[aa] Length of protein sequences and maximal covers are measured in amino acids (aa).
[ms] Time to compute measured in milliseconds (ms).

Table 3: Examples of maximal covers for Arabidopsis proteins within a particular range of maximal cover lengths computed using MAXCOVER.

MC-length range	protein ID	protein length	MC-length	positions covered	% coverage	time to compute
1	NP_001031361.2_685	653	1	67	10 %	1.182 ms
2	NP_001332341.1_20487	26	2	4	15 %	0.232 ms
3	NP_850977.1_46441	58	3	6	10 %	0.325 ms
4	NP_001031813.1_1106	35	4	7	20 %	0.279 ms
5	NP_001329516.1_17663	66	5	10	15 %	0.334 ms
6	NP_172240.1_21498	44	6	12	27 %	0.304 ms
7	NP_189459.1_30049	111	7	14	12 %	0.377 ms
8	NP_001322312.1_10459	124	8	24	19 %	0.31 ms
9	NP_001321372.1_9520	68	9	15	22 %	0.353 ms
10 - 19	NP_568552.1_43794	321	19	57	17 %	0.543 ms
20 - 29	NP_001323842.1_11989	564	24	360	63 %	1.64 ms
30 - 39	NP_174445.1_23044	463	31	53	11 %	0.801 ms
40 - 49	NP_195887.2_34639	371	48	96	25 %	0.536 ms
50 - 59	NP_001154390.2_4083	744	57	114	15 %	1.278 ms
60 - 69	NP_176714.4_24533	318	61	122	38 %	0.72 ms
70 - 79	NP_001328999.1_17146	152	76	152	100 %	0.408 ms
80 - 89	NP_001325858.1_14005	411	89	159	38 %	0.695 ms
90 - 99	-	-	-	-	-	-
100 - 199	NP_849291.1_44918	228	152	228	100 %	0.425 ms
200 - 299	NP_851029.1_46486	305	228	304	99 %	0.643 ms
300 - 399	NP_173553.1_22430	430	304	332	77 %	1.026 ms

[aa] Length of protein sequences and maximal covers are measured in amino acids (aa).
[ms] Time to compute measured in milliseconds (ms).

Table 4: Examples of maximal covers for human proteins within a particular range of maximal cover lengths computed using MAXCOVER.

MC-length range	protein ID	protein length	MC-length	positions covered	% coverage	time to compute
1	XP_006716126.1_69540	707	1	79	11 %	1.304
2	NP_001341932.1_30631	70	2	22	31 %	0.326
3	NP_006266.2_48075	343	3	77	22 %	0.512
4	NP_001135954.1_6750	67	4	8	11 %	0.318
5	XP_011524892.1_80014	474	5	50	10 %	0.744
6	NP_009048.1_48830	241	6	36	14 %	0.427
7	XP_016882728.1_107042	389	7	42	10 %	0.633
8	NP_001340727.1_29802	403	8	64	15 %	0.718
9	XP_016882449.1_106829	602	9	63	10 %	0.936
10 - 19	NP_001334947.1_25817	515	18	108	20 %	1.047
20 - 29	XP_005255792.1_64442	395	27	54	13 %	0.55
30 - 39	NP_001159711.1_8644	317	33	50	15 %	0.541
40 - 49	NP_001123991.1_5873	605	44	88	14 %	1.179
50 - 59	XP_011521003.1_78175	893	55	110	12 %	1.671
60 - 69	NP_031372.2_49017	676	64	124	18 %	1.041
70 - 79	XP_016883383.1_107510	557	78	139	24 %	1.118
80 - 89	XP_016878127.1_103859	533	85	170	31 %	0.787
90 - 99	NP_001375298.1_41921	1110	94	188	16 %	1.81
100 - 199	XP_011529785.1_82360	528	116	232	43 %	0.841
200 - 299	XP_011514536.1_75210	597	292	365	61 %	0.958
300 - 399	XP_011537716.1_86000	1782	327	456	25 %	4.41
400 - 499	XP_011509528.1_72895	7795	485	970	12 %	22.539
500 - 599	-	-	-	-	-	-
600 - 699	NP_066289.3_53510	684	608	684	100 %	1.46
700 - 799	NP_001289300.1_18913	3794	724	1692	44 %	13.823
800 - 899	NP_056198.2_50743	2818	893	1137	40 %	8.226

^{aa} Length of protein sequences and maximal covers are measured in amino acids (aa).
^{ms} Time to compute measured in milliseconds (ms).

Table 5: Comparison between MUMmer's `repeat-match` and MAXCOVER software to compute NE substrings on a random sample of 40,000 protein sequences.

	MUMmer `repeat-match`	**MAXCOVER**
Data Structure	Suffix Trees	Suffix Arrays
Time	Slower (29 min. 28 sec.)	Faster (1 min. 15 sec.)
Accuracy	Does not report every pair of start positions for matching regions	Reports ALL start positions for matching regions
Output Format	Lengthy; Hard to decipher useful information	Condensed; Clear which set of positions refer to the same region