Minimal Absent Words in Plasmids

Veronika Hendrychová, Nazar Misyats

December 2023

1 Introduction

In recent years, significant attention has been dedicated to the study of CRISPR (Clustered Regularly Interspaced Short Palindromic Repeats), an evolutionary system that serves as an adaptive immunity mechanism within bacteria. This natural defense mechanism is employed by bacteria to guard against bacterial viruses (phages) and predatory plasmids, and can be compared to the bacteria holding a database of harmful sequences. Upon discovering a match of the incoming molecule within the database, the system eliminates it before it can perform any damage.

As an evolutionary consequence, the viruses and plasmids have developed sophisticated anti-defense mechanisms to avoid being detected. They achieve this by avoiding specific patterns, analogous to the words stored in the "internal database" of the immunity system.

The objective of our project is to develop an algorithm to detect so-called *minimal absent words*. They provide valuable insights into the potential targets of CRISPR defense systems across bacteria, as they mark patterns that are avoided by plasmids. For example, three minimal absent words of the human genome were found in Ebola virus genomes. Understanding these specific words that evade detection can shed light on the adaptability and evolution of bacterial defense mechanisms, contributing to the broader understanding of microbial interactions.

1.1 Preliminaries

We consider sequences (strings) on the DNA alphabet $\Sigma = \{A, C, G, T\}$. Given a string $s \in \Sigma^*$, we denote its length by $|s|$. The canonical version of a sequence s is the lexicographic minimum between itself and its reverse complement. Recall that the reverse complement of s is obtained by reversing it and transforming A into T, C into G, and vice versa. As an example, the reverse complement of AGGTT is AACCT. The canonical version of AGGTT is therefore AACCT because AACCT $<_{lex}$ AGGTT.

Definition 1.1 (Absent word)**.** *Let x and s be two strings on the alphabet Σ. We say that x is an absent word of s if neither x nor its reverse complement occur as a substring of s. We say that x is an absent word of a set S of strings if x is an absent word with respect to all $s \in S$.*

Definition 1.2 (Minimal absent word (MAW))**.** *Let x be a string of length $|x| \geq 3$ and S be a set of strings. We say that x is a minimal absent word (MAW, for short) of S if both of the following conditions hold:*

1. *x is an absent word of S.*
2. *For every substring w of x such that $|w| < |x|$, it holds that w or its reverse complement is a substring of at least one element of S.*

Theorem 1.1 (MAW characterized, Barton et al. [3])**.** *String $x = x[0 \ldots n-1]$ is a MAW of string s if and only if x is not a factor of s, and both $x[1 \ldots n-1]$ and $x[0 \ldots n-2]$ or their reverse complements are factors of s.*

2 Methods

Here we present three possible approaches to create an algorithm obtaining the minimal absent words. All of these methods are designed for the following input and output.

- **Input:**
  1. FASTA file containing the set S of DNA sequences.
  2. Parameter $k_{\max}$, specifying the maximum length of Minimal Absent Words (MAWs) to be reported.
  3. (optional) Integer `max_seqs` restricting the number of sequences taken from the provided file (taken in original order from the beginning).
- **Output:** TSV (Tab-separated values) file, which contains the following two fields:
  1. Length k (from 3 up to $k_{\max}$).
  2. Lexicographically sorted, comma-separated list of all distinct canonical sequences of MAWs of length k for the set S, ommitting the k's for which there are no MAW's.

2.1 Naive approach

To establish a baseline of possible code efficiency, we first present a naive approach algorithm to identify minimal absent words. The algorithm consists of first generating all possible strings over a given alphabet, then checking the compliance with the minimal absent word definition 1.2 of each of them with the `is_maw` function. The algorithm outline is stated in 1 as a function `find_maws`.

Algorithm 1 Function `find_maws`

```
1:  function FIND_MAWS(sequences, kmax)
2:      for k ← 3 to kmax do
3:          for x in all possible strings of length k over ALPHABET do
4:              if IS_MAW(x, sequences) then
5:                  add TO_CANONICAL(x) to maws
6:              end if
7:          end for
8:      end for
9:      return maws
10: end function
```

2.2 String-extensions approach

The naive approach, which generates all possible strings and checks their minimality, can be improved by generating the MAW candidates more efficiently. To do so, we use string extensions as described for left side in function `extended_left` 2, and similarly for right side. These are the only candidates for MAWs: if we had a MAW candidate b that has not been created as a left or right extension of some of the substrings from the sequence, the factor $b[0 \dots len(b)-1]$ could not be contained in any of the sequences, therefore b would not be minimal. Using the property 1.1 and the knowledge of to which side the original substring was extended, we are then able to confirm the MAW candidate by checking only one substring. The algorithm is described as function `get_extended_maws` 3.

2.3 Suffix-array approach

We propose another approach using suffix arrays and derivative data structures to generate substrings faster, which accelerate the string-extension approach described earlier. For each sequence s in the dataset, we apply the algorithm proposed by Kärkkäinen and Sanders [1] to

Algorithm 2 Function EXTENDED_LEFT

```
function EXTENDED_LEFT(seq: str) : set of str
    exts ← empty set
    for a in ALPHABET do
        exts.add(a + seq)
    end for
    return exts
end function
```

Algorithm 3 Function GET_EXTENDED_MAWS

```
function GET_EXTENDED_MAWS(sequences: set of str, kmax: int) : set of str
    all_candidates ← union of all left and right extensions of all substrings of all sequences
    for x in all_candidates do
        if x is a left extension then
            verify whether x[0 : len(x) − 1] or its reverse complement is contained in some of the sequences
            if it is not verified then
                continue
            end if
        end if
        if x is a right extension then
            verify whether x[1 : len(x)] or its reverse complement is contained in some of the sequences
            if it is not verified then
                continue
            end if
        end if
        verify whether x and its reverse complement are not present in any of the sequences
        if it is not verified then
            continue
        end if
        add TO_CANONICAL(x) to maws
    end for
    return maws
end function
```

compute the suffix array (SA) p_s of s in linear time. $p_s[i]$ is the sorted position of the suffix of s starting at i. From this, we construct the Longest Common Prefix (LCP) array of size $|s|-1$ such that such that $\text{lcp}_s[i]$ is the length of the longest common prefix between $p_s[i]$ and $p_s[i+1]$. The LCP array is built using Kasai's algorithm [2], which is a linear-time and constant memory construction. Algorithm 4 Using the LCP array of the SA of s, we can iterate over all substrings of a given range of size in s. while avoiding iterating over duplicate substrings that appear in s. While having an $O(|s|^2)$ time complexity, the same as the naive enumeration of substrings, this algorithm requires fewer steps in practice.

Algorithm 4 Function `get_substrings`

```
1: function GET_SUBSTRINGS(s, p_s, lcp_s, L_min, L_max)
2:     for i to |s| − 1 do
3:         if n − p_s[i] ≥ L_min then
4:             if i = 0 then
5:                 L_0 ← L_min
6:             else
7:                 L_0 ← max(L_min, lcp_s[i − 1])
8:             end if
9:             for L ← L_0 to min(n − p_s[i], L_max) do
10:                yield s[p_s[i]..p_s[i] + L]
11:            end for
12:        end if
13:    end for
14: end function
```

All the substrings of length 2 to k_{max} are generated and stored in sets (hashtables), organized by the sizes of the substrings, for all s of S. We then iterate over the substrings w of sizes $k=2$ to $k_{max}-1$. For each $a \in \Sigma$, $x = aw$ is a candidate MAW of length $k+1 \in [3, k_{max}]$. Thus, since w is a substring in S, if $x[0..|x|-2] = aw[0..|w|-2]$ (or its reverse complement) is a substring of any s of S, and x is not, then x is a MAW of S as a consequence of theorem 1.1. The mechanism is the same for $x' = wa$.

3 Results

As our contribution, we provide comparison between these proposed methods. To test the performance of our algorithms, we used the dataset of the EBI plasmid database. For different numbers of sequences from the dataset, we measured the execution time of each method depending on the parameter k_{max}. We ran the experiments on a 16 GB RAM device, and stopped it once the execution time exceeded approximately 30 minutes. To obtain some real results, we applied our Extensions algorithm on the whole dataset of EBI plasmid database to find MAWs of length $kmax = 6$ and we got the result that there aren't any.

4 Discussion

In comparing the performance of the implemented algorithms (Naive, Extensions, and Suffix-Array) we can observe that the naive approach outperforms in absolute runtime for smaller datasets. However, as the k_{max} increases, the computational cost of the naive algorithm grows steeply, as expected from generating all possible substrings. The Extensions algorithm has initially a higher absolute runtime than the naive one, however, it shows a lower slope of growth, making it more efficient for larger k_{max} values. On the other hand, the SA-based algorithm maintains almost steady values for the increasing k_{max} parameter. For lower values of k_{max}, the absolute computational time is typically one order of magnitude higher than for the Extensions algorithm. This is caused by the computational cost of calculating the SA and LCP for each sequence and generating the set of substrings during the pre-processing step.

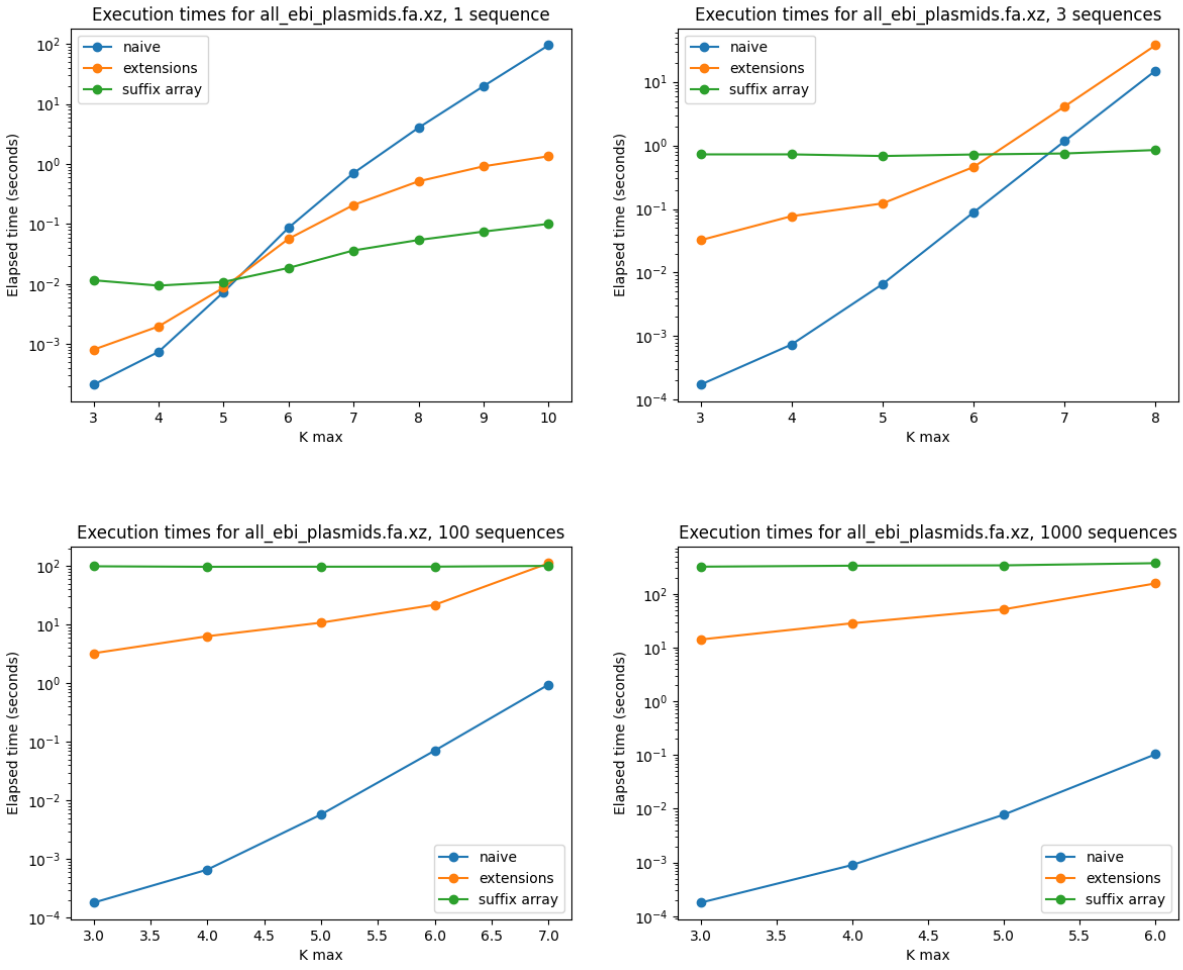

Figure 1: Comparison of the execution times of algorithms for various number of sequences.

While our proposed algorithms are able to asses the minimal absent words of small-scale data, it is important to acknowledge their limitations. The problem of MAWs has been studied and the state-of-the-art methods outperform the algorithms here; i. e. the external memory MAW computation [4], studying MAWs on Run-Length Encoded Strings [5], or general algorithm development as in [6].

References

[1] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt, *Linear Work Suffix Array Construction*, *J. ACM*, vol. 53, no. 6, pp. 918–936, Nov 2006.

[2] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, *Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*, *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, Vol. 2089, pp. 181–192, 2001. `https://doi.org/10.1007/3-540-48194-X_17`

[3] Barton, C., Heliou, A., Mouchard, L. et al. *Linear-time computation of minimal absent words using suffix array.* BMC Bioinformatics 15, 388 (2014). `https://doi.org/10.1186/s12859-014-0388-9`

[4] Alice Héliou, Solon P. Pissis, Simon J. Puglisi, *emMAW: computing minimal absent words in external memory*, *Bioinformatics*, Volume 33, Issue 17, September 2017, Pages 2746–2749, https://doi.org/10.1093/bioinformatics/btx209

[5] Tooru Akagi, Kouta Okabe, Takuya Mieno, Yuto Nakashima, and Shunsuke Inenaga, *Minimal Absent Words on Run-Length Encoded Strings*, *arXiv:2202.13591 [cs.DS]*, 2022.

[6] A. J. Pinho, P. J. Ferreira, S. P. Garcia, et al., *On finding minimal absent words*, *BMC Bioinformatics*, vol. 10, p. 137, 2009. https://doi.org/10.1186/1471-2105-10-137