Implementation and Analysis of Minimal (p-)Absent Words in DNA Sequence

Amory ANTAO
École Normale Supérieure Rennes
Rennes, France
amory.antao@ens-rennes.fr

***Abstract*—The identification of Minimal Absent Words (MAWs) and p-MAWs in DNA sequences is a major challenge in bioinformatics, particularly for understanding defense mechanisms in organisms such as CRISPR systems. This project presents the design and implementation of efficient algorithms to detect these patterns in large genomic databases.**
After theoretical analysis, the computation of MAWs is found to have a complexity of $\mathcal{O}(n \times \textbf{max\_length}^2)$, while the computation of p-MAWs has a complexity of $\mathcal{O}(n \times \textbf{max\_length}^4 \times (1 + (4\gamma_p\delta_p)^{\textbf{max\_length}}))$, where γ_p and δ_p are introduced parameters, and $4\gamma_p\delta_p$ remains practically below 1, thus avoiding exponential behavior. This work represents an advancement in the analysis of absent patterns in biological sequences, particularly regarding p-MAWs, with potential applications in genetics and molecular biology.

I. Introduction

A. Context

One of the recent major breakthroughs in computational biology is the discovery of the CRISPR system (Clustered Regularly Interspaced Short Palindromic Repeats) [1], a discovery that earned E. Charpentier and J. Doudna the 2020 Nobel Prize in Chemistry for the development of the CRISPR-Cas9 genome-editing technique. This system is used by bacteria as a defense mechanism against bacterial viruses and predatory plasmids. It relies on an intern specific recognition of DNA sequences to eliminate potential threats.
In this context, identifying absent patterns in DNA sequences — known as the search for “minimal absent words“ (MAWs) — is particularly relevant for understanding these defense mechanisms.

B. Definition

Before defining MAWs, it is essential to introduce the concept of the reverse complement. The reverse complement (RC) of a word x is obtained by reversing the sequence and replacing each nucleotide base with its complement:

- $A \longrightarrow T$
- $T \longrightarrow A$
- $C \longrightarrow G$
- $G \longrightarrow C$

This concept is critical because, in practice, when sequencing DNA, it is often impossible to know from which strand the information is extracted. To account for this bias, it is necessary to consider the reverse complement of every observed pattern.
An “absent word“ (AW) of a sequence s is defined as a pattern x such that neither x nor its reverse complement $\mathrm{RC}(x)$ appears in s as a substring.
However, as the size of the considered patterns increases, the number of AWs grows exponentially. To limit this combinatorial explosion, we focus on MAWs, which are minimal patterns that cannot be reduced without losing their absence property.
Formally, a pattern x of length $|x| > 1$ is a MAW of a sequence s if:

- x is an AW of s, and
- every substring w of x, with $|w| < |x|$, is present in s (or its reverse complement is present in s).

As an extension of this concept, it may be useful to introduce the notions of p-AW and p-MAW. For $p \in]0, 1]$, a pattern x is considered a p-AW of a set of sequences $\mathcal{S}$ if x is an AW in at least $p \cdot |\mathcal{S}|$ sequences $s \in \mathcal{S}$. Similarly, x is a p-MAW of $\mathcal{S}$ if it is a p-AW of $\mathcal{S}$ and all of its substrings w that are different from x are not p-AWs of $\mathcal{S}$.

C. First Examples

Let’s take two simple examples on the alphabet $\{a, b\}$ to illustrate these definitions.These examples are simplified because they use a smaller alphabet and do not consider reverse complements, but they will help us clearly understand the concepts discussed.

a) Example 1: Consider the sequence $s = abaaba$. The pattern bab does not appear in s, making it an AW of s. Furthermore, the substrings ba, ab, a, and b are all present in s, which allows us to conclude that bab is a MAW of s.

b) Example 2: Now, consider the set of sequences $\mathcal{S} = abbaa, bbb, bbba$ with $p = 0.5$.
In this case, a pattern must be absent from at least two sequences $(0.5 \times 3 = 1.5)$.
The pattern $bbbb$ does not appear in any of the sequences in $\mathcal{S}$. Therefore, it is a p-AW. Moreover, the substrings of $bbbb$ (b, bb, bbb) appear in at least two sequences in $\mathcal{S}$, allowing us to conclude that $bbbb$ is a p-MAW.

D. Important Remarks

From the previous examples, two key observations can be made:

- To check whether a pattern $x = x_1 x_2 \dots x_n$ is a MAW of a sequence s, it is sufficient to verify that:
  - x is an AW of s, and
  - the sub-patterns $x' = x_2 \dots x_n$ and $x'' = x_1 \dots x_{n-1}$ are not AWs of s.

  This is because any substring w of x such as $w \neq x$, can be expressed as a substring of either x' or x''. If both x' and x'' are not AWs of s, then none of the substrings of x will be AWs either.
- A pattern can be a p-MAW without being a MAW of any individual sequence in $\mathcal{S}$.
  For instance, consider the case of $\mathcal{S} = s_1 = abbaa, s_2 = bbb, s_3 = bbba$ with $p = 0.5$, and the pattern $x = bbba$:
  - x is not a MAW of s_1 because bbb is an AW of s_1
  - x is not a MAW of s_2 because a is an AW of s_2
  - x is not an AW of s_3 because $x = s_3$

  However, x is a p-AW of $\mathcal{S}$ because it is absent from two sequences: s_1 and s_2. Moreover, bbb is not an p-AW of $\mathcal{S}$ because it is in s_2 and s_3, and bba is not an p-AW of $\mathcal{S}$ because it is in s_1 and s_2.
  Therefore, x is a p-MAW of $\mathcal{S}$.

E. Objectives

The main objective of this work is to design and implement efficient algorithms to identify MAWs and p-MAWs in a set of DNA sequences.

II. Methods: Finding Limited-Length MAWs

A. General Algorithm

The general outline of the algorithm is as follows:

Algorithm 1 Find_MAWs(S, max)

```
Require: S : DNA sequence of length $n$
Require: max : maximum MAW length
Ensure: $MAWs$ : list of minimal absent words
 1: $sa \leftarrow$ SuffixArray($S$)
 2: $st \leftarrow$ SuffixTree($S$)
 3: $MAWs \leftarrow \emptyset$
 4: $potMAWs \leftarrow$ PotentialMAWs(S, $sa$, max))
 5: for all $maw \in potMAWs$ do
 6:     if Verify(S, $st$, $maw$, max) then
 7:         $MAWs \leftarrow MAWs \cup \{maw\}$
 8:     end if
 9: end for
10: return $MAWs$
```

The algorithm is divided into several parts:

- Building the suffix array and suffix tree of the input DNA sequence
- Searching for potential MAWs between successive pairs of suffixes in the suffix array
- Verifying whether the potential MAWs are actual minimal absent words

B. Suffix Array Construction

The suffix array is a data structure that lists all suffixes of a sequence, sorted in lexicographical order. Each suffix is represented by its starting index in the original sequence.

a) Example: For the sequence $s = abaaba$, the different suffixes are:

Table I
List of suffixes of s

Suffix	**Index**
abaaba	0
baaba	1
aaba	2
aba	3
ba	4
a	5

Table II
List of suffixes of s sorted lexicographically

Suffix	**Index**
a	5
aaba	2
aba	3
abaaba	0
ba	4
baaba	1

Table III
Suffix array of s

5	2	3	0	4	1

b) Building the Suffix Array: In practice, the suffix array is computed using the PySAIS library [2]. This library is based on the SA-IS (Suffix Array Induced Sorting) algorithm, which constructs the suffix array of a sequence of length n in $\mathcal{O}(n)$ time [3].

C. Suffix Tree

The suffix tree encodes similar information to the suffix array. However, it is less practical for implementing the next part of the algorithm and requires significantly more memory space. Its main advantage is that pattern searches in the suffix tree can be performed in $\mathcal{O}$(max_length) time.
To build the suffix tree, the suffix-tree library [4] was used. This library implements the Ukkonen algorithm, which constructs the suffix tree in $\mathcal{O}(n)$ time [5].

D. Finding Potential MAWs Between Consecutive Suffix Pairs

A pattern x of length n can only be a potential MAW if it is not present in the original sequence s, but all its substrings of length $n-1$ are present. Therefore, the idea is to identify patterns that are prefixes of suffixes in the original sequence, which can be easily found using the suffix array.
To avoid redundancy, we follow the suffix array in lexicographical order. Patterns will be processed in that order. The algorithm's approach is to examine the differences between two consecutive suffixes, making changes from the initial suffix to reach the final suffix, without ever adding more than one letter that deviates from a substring of the original sequence.
These modifications are made in three phases:

1) Start with the initial suffix and check if a letter can be added.
2) Descend from the initial suffix, considering words not present in the sequence, until reaching the prefix of the final suffix.
3) Ascend from the prefix to the final suffix, considering words not present in the sequence.

a) Example:

- Between ”” and a :
  - There are no words between ”” and a, so no potential MAWs are added.
- Between a and $aaba$:
  - We are directly in phase (3) because a is a prefix of $aaba$. We ascend to $aaba$ while adding words not present in the original sequence.
    $a \longrightarrow aa$
    $aa \longrightarrow aab$ - pattern addable: **aaa**
    $aab \longrightarrow aaba$
- Between $aaba$ and aba :
  - Start phase (1). Check if a letter can be added to the initial suffix.
    Both a and b can be added, remaining between the two suffixes. Patterns added: **aabaa** and **aabab**.
  - Start the descending phase:
    $aaba \longrightarrow aab$ - pattern addable: **aabb**
    $aab \longrightarrow aa$
    $aa \longrightarrow a$ - prefix found: ab
  - Start the ascending phase:
    $ab \longrightarrow aba$
- Between aba and $abaaba$:
  - We are directly in phase (3). We can start the ascending phase:
    $aba \longrightarrow abaa$
    $abaa \longrightarrow abaab$ - pattern addable: **abaaa**
    $abaab \longrightarrow abaaba$
- Between $abaaba$ and ba :
  - Start phase (1). Both a and b can be added, remaining between the two suffixes. Patterns added: **abaabaa** and **abaabab**.
  - Start the descending phase:
    $abaaba \longrightarrow abaab$ - pattern addable: **abaabb**
    $abaab \longrightarrow abaa$
    $abaa \longrightarrow aba$
    $aba \longrightarrow ab$ - pattern addable: **abb**
    $ab \longrightarrow a$
    $a \longrightarrow$ ”” - prefix found: b
  - Start the ascending phase:
    $b \longrightarrow ba$
- Between ba and $baaba$:
  - We are directly in phase (3). We can start the ascending phase:
    $ba \longrightarrow baa$
    $baa \longrightarrow baab$ - pattern addable **baaa**
    $baab \longrightarrow baaba$
- Between $baaba$ and ”” :
  - Start in phase (1). Both a and b can be added. Patterns added: **baabaa** and **baabab**.
  - Start the descending phase:
    $baaba \longrightarrow baab$ - pattern addable: **baabb**
    $baab \longrightarrow baa$
    $baa \longrightarrow ba$
    $ba \longrightarrow b$ - patterb addable: **bb**
    $b \longrightarrow$ ””

Thus, we obtain:

Table IV
Potential MAWs found

Suffix	Index	Potential MAWs found
a	5	*aaa*
aaba	2	*aabaa - aabab - aabb*
aba	3	*abaaa*
abaaba	0	*abaabaa - abaabab - abaabb - abb*
ba	4	*baaa*
baaba	1	*baabaa - baabab - baabb - bb*

b) Shortcut: In practice, since the input includes a maximum pattern length, we can cut the descents and ascents to handle only cases where the length remains below the desired limit.

c) Complexity: Let n be the length of the sequence and max_length the maximum pattern length.
We know that this search must be performed $n+1$ times because there are n suffixes, resulting in $n+1$ pairs of suffixes (including the empty string ””).
For the search process:

- Phase (1) is done in constant time (linear with respect to the size of the alphabet).
- Phase (2) is done in $\mathcal{O}(\text{max\_length})$. Indeed, we start by cutting all descents that exceed the maximum length. Then, we spend a constant amount of time at each pattern length (a number of comparisons linear

with the size of the alphabet). This is done for pattern lengths ranging from 1 to at most max_length. Therefore, Phase (2) is done in $\mathcal{O}(\text{max\_length})$.
- Phase (3) is also done in $\mathcal{O}(\text{max\_length})$. Overall, the same analysis as in Phase (2) applies.

Thus, the complexity of finding potential MAWs is $\mathcal{O}(n \times \text{max\_length})$.

E. Verification of Potential MAWs

To complete the algorithm, each potential MAW found must be verified to ensure that it is indeed a minimal absent word.
To do this, it is sufficient to check that both substrings of length $|x| - 1$ are not AWs. This verification is performed in $\mathcal{O}(\text{max\_length})$ using the suffix tree that we built.
Additionally, we have seen that the number of potential MAWs is in $\mathcal{O}(n \times \text{max\_length})$. Therefore, verifying all potential MAWs results in a complexity of $\mathcal{O}(n \times \text{max\_length}^2)$.

F. Final Complexity

Finally, the overall complexity of this algorithm is $\mathcal{O}(n \times \text{max\_length}^2)$.

III. Methods: Finding Limited-Length p-MAWs

A. General Algorithm

Using the previously described algorithm, it is possible to obtain the set of MAWs of length less than max_length for a set of DNA sequences. These results can then be used to find all p-MAWs for the same set of sequences.
To achieve this, the following algorithm can be implemented: The algorithm is composed of several parts:

Algorithm 2 Find_pMAWs($MAWs$, p, nb_seq)

```
Require: $MAWs$: list of MAWs for $nb\_seq$ sequences
Require: $p$: proportion threshold
Require: $max$: maximum MAW length
Ensure: $pMAWs$: list of $p$-MAWs
 1: $threshold \leftarrow \lceil p \times nb\_seq \rceil$
 2: $pMAWs \leftarrow \emptyset$
 3: $dictLength \leftarrow$ CreateDictPerLength(MAWs)
 4: $cache \leftarrow$ PreCountAW($dictLength$)
 5: $pot\_pMAWs \leftarrow$ PotentialpMAW($dictLength, cache$)
 6: for all $pmaw \in pot\_pMAWs$ do
 7:     if Verify($dictLength, pmaw, n, cache$) then
 8:         $pMAWs \leftarrow pMAWs \cup \{pmaw\}$
 9:     end if
10: end for
11: return $pMAWs$
```

- Modifying the list of MAWs
- Pre-computing a cache to reduce redundant calculations
- Identifying all potential p-MAWs to be considered
- Verifying whether they are indeed p-MAWs

B. Modification of the MAW List

This step may seem anecdotal, but it is actually essential. The goal is to convert the input, which gives us the global list of MAWs for each sequence, into a list of different MAWs categorized by their lengths. Additionally, it is necessary to keep track of the original sequence each MAW comes from.

a) Principle: To achieve this, we associate a flag with each MAW. We use a bitarray of size nb_seq, where a 1 at index i indicates that the given MAW is a MAW of sequence i. By slightly modifying this flag, as we will see in the following section, we can count how many sequences a given pattern is absent from, which is necessary to obtain the p-MAWs.
The modification of the MAW list and the addition of the flag can be done simultaneously. For each MAW in each sequence, we modify its position to a set that categorizes it by length and add or update the corresponding flag.

b) Example: If we take the previous Example 2, with $S = \{abbaa, bbb, bba\}$, and apply the first algorithm, we obtain the following MAWs:

Table V
MAWs by Sequence

Sequence	MAWs
abbaa	$aaa - aab - aba - bab - bbb$
bbb	$bbbb$
bbba	$aa - ab - bbbb$

The algorithm then produces the following results after processing each MAW:

Table VI
Reorganized MAWs and Flags

Length	MAW	Flag
2	*aa*	001
	ab	001
3	*aaa*	100
	aab	100
	aba	100
	bab	100
	bbb	100
4	*bbbb*	011

c) Complexity: For each MAW, the calculation is performed in constant time. Let n be the total number of bases. As previously mentioned, the number of potential MAWs is $\mathcal{O}(n \times \text{max\_length})$. Therefore, the complexity of this reorganization step is $\mathcal{O}(n \times \text{max\_length})$.

C. Cache Precomputation

This step is also very important. The main goal is to count how many sequences a specific pattern is absent from. Since we do not have direct access to the sequences, we start from the given pattern and check each of its substrings to update the flag associated with the pattern.

a) Pattern Update: The pattern update is done using a simple logical OR operation. For example, to find the AW-flag for the pattern *aab*, a logical OR is applied to the flags of each substring of *aab*, which in this case are *aa* and *ab* (the other substrings, which are not MAWs, have a null flag). As a result, we get 101. The pattern is thus absent from sequences 1 and 3. Note that the pattern is also absent from sequence 2, but this does not appear because sequence 2 contains only the letter b, and by definition, a cannot be considered a MAW for that sequence. In practice, we restrict ourselves to sequences that contain at least one occurrence of each letter, which is not a significant issue for long DNA sequences.

In terms of complexity, a word of size m has $\sum_{k=1}^{m}(m+1-k)$ substrings (1 of size m, 2 of size $m-1$, 3 of size $m-2$, etc.), which can be rewritten as $\sum_{k'=1}^{m} k'$, where $k' = m+1-k$. This sum equals $\frac{m(m+1)}{2}$ substrings.

Additionally, the logical OR operation between two bitarrays of size m is performed in $\mathcal{O}(m)$.

Therefore, updating a pattern of length up to max_length is done in $\mathcal{O}(\text{max\_length}^3)$.

b) Cache: The cache corresponds to the update of all patterns that are MAWs. The total number of MAWs is in $\mathcal{O}(n \times \text{max\_length})$. Thus, the complexity of creating the cache is $\mathcal{O}(n \times \text{max\_length}^4)$.

It should be noted that this does not improve the theoretical complexity of the algorithm, since we still need to count the number of absences in the same way for many patterns that are not in the cache. However, in practice, it allows for a significant time-saving gain.

D. Obtaining Potential p-MAWs

a) Principle: To find all potential p-MAWs, we start from each MAW (a p-MAW must have at least one of its substrings that is a MAW for a sequence, otherwise it would not be absent from any sequence). We count how many sequences the pattern is absent from. If it is absent from fewer sequences than the *threshold*, it cannot be a p-MAW. In this case, we need to test the patterns obtained by extending the MAW with each of the 4 different letters. Conversely, if the pattern is absent from at least *threshold* sequences, it is considered a potential p-MAW, and there is no need to test its extensions, as these would contain a substring that is a p-AW.

b) Complexity: The complexity calculation for this part is quite complicated. First, for each pattern tested, we need to determine how many sequences it is absent from. Even though the cache allows us to get a result in constant time fairly often, when the pattern is not a MAW for any sequence, the counting phase is in $\mathcal{O}(\text{max\_length}^3)$ as previously discussed.

Moreover, finding a precise upper bound on the number of tests required is complex. A rough upper bound would be $4^{\text{max\_length}}$, but for this to happen, no pattern of length up to max_length should be absent from at least *threshold* sequences, which is unlikely unless max_length is very small, in which case the bound is not so large.

To provide more interesting insights, we introduce two parameters: γ_p and δ_p. The parameter γ_p represents the probability that a tested pattern is absent from fewer than *threshold* sequences. The parameter δ_p represents the average fraction of new tests to be conducted when a tested pattern is absent from fewer than *threshold* sequences.

Using these parameters, an upper bound on the number of tests to be conducted for each MAW is given by:

$$\sum_{k=1}^{\text{max\_length}} (\gamma_p \times 4 \times \delta_p)^k = \frac{1-(4\gamma_p\delta_p)^{\text{max\_length}}}{1-4\gamma_p\delta_p}$$

Thus, for all $\mathcal{O}(n \times \text{max\_length})$ MAWs, we need to perform $\mathcal{O}(1+(4\gamma_p\delta_p)^{\text{max\_length}})$ tests, each with a complexity of $\mathcal{O}(\text{max\_length}^3)$. Therefore, the overall complexity becomes $\mathcal{O}(n \times \text{max\_length}^4 \times (1+(4\gamma_p\delta_p)^{\text{max\_length}}))$.

E. Verification

All these potential p-MAWs must then go through a verification step. To do this, it is sufficient to count how many sequences are missing the two largest strict substrings of the pattern. The pattern is validated if its substrings are not p-AWs.

a) Complexity: As previously discussed, the theoretical complexity of this calculation is $\mathcal{O}(\text{max\_length}^3)$, although in practice this number is much lower, as the caching system avoids a significant number of redundant calculations.

Performing the verification for all potential p-MAWs results in an overall complexity of $\mathcal{O}(n \times \text{max\_length}^4 \times (1+(4\gamma_p\delta_p)^{\text{max\_length}}))$.

F. Final Complexity

Combining all parts, the final complexity of the algorithm is $\mathcal{O}(n \times \text{max\_length}^4 \times (1+(4\gamma_p\delta_p)^{\text{max\_length}}))$.

IV. Results

This section aims to address several objectives:

1) To analyze how max_length influences the performance of MAWs computation.
2) To investigate the impact of sequence length on the performance of MAWs computation.
3) To determine the optimal max_length value.

4) To evaluate how p affects the performance of p-MAWs computation.
5) To experimentally estimate the parameters γ_p and δ_p.

For this purpose, we rely on the EBI plasmid database [6].

A. *Influence of max_length on MAWs Computation*

To examine the influence of max_length, we fix the number of sequences to be processed. Here, we consider the first 10 sequences from the database, corresponding to 134,764 bases, and vary the max_length parameter.
The obtained results are as follows:

Table VII
Influence of max_length on MAWs Computation

max_length	**User Time**
8	0m3.392s
10	0m12.229s
12	0m24.119s
15	0m46.560s
18	1m15.932s
20	1m24.937s
22	1m41.464s
25	2m10.284s
27	2m26.221s
30	2m58.021s

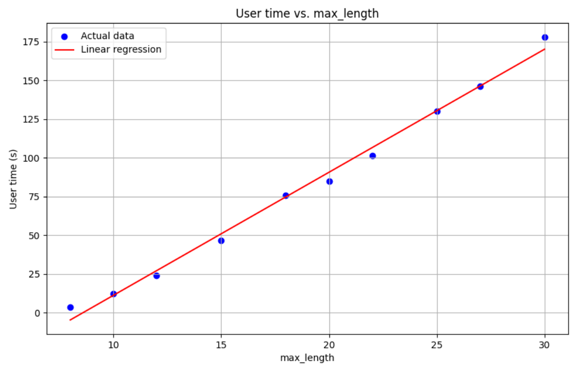

Figure 1. User time vs. max_length

The corresponding equation is: time $=$ 7.953 $\times$ max_length $-$ 68.413, with a determination coefficient $R^2 = 0.9932$.
Thus, the influence of max_length on MAWs computation time is linear.

B. *Influence of Sequence Length on MAWs Computation*

We compute the MAWs for several sequences of different lengths, using a fixed max_length of 30.
The obtained results are:

Table VIII
Influence of Sequence Length on MAWs Computation

|S|	**User Time**
$1,022$	0m1.708s
$2,467$	0m3.211s
$3,828$	0m5.346s
$7,005$	0m9.341s
$10,031$	0m12.465s
$15,398$	0m20.273s
$25,014$	0m32.328s
$50,100$	1m7.120s
$100,021$	2m22.317s
$200,231$	4m25.720s

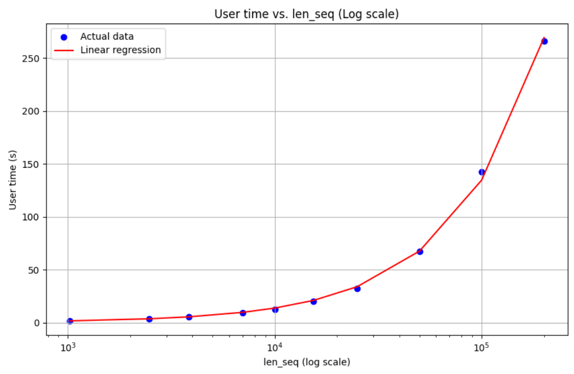

Figure 2. User time vs. Sequence Length (log scale)

The corresponding equation is: time $= 0.001343 \times |S| + 0.223761$, with a determination coefficient $R^2 = 0.9988$.
Thus, the influence of sequence length on MAWs computation time is linear.

C. *Optimal max_length*

From the previous experiments, we observe that max_length has a significant impact. Therefore, we aim to determine the value that optimally balances the number of captured MAWs and computation time.
Using high values, as in the second experiment, we identify the max_length that captures more than 99.9% of MAWs. We choose such a high percentage because the distribution of MAWs follows a bell curve.
This is evident in the following graphs, both in linear and logarithmic scales, for sequences of length 200,000 bases:

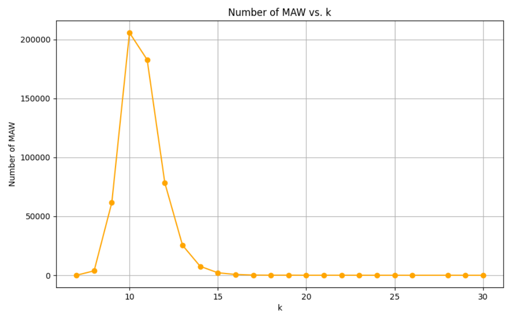

Figure 3. Number of MAWs vs. k

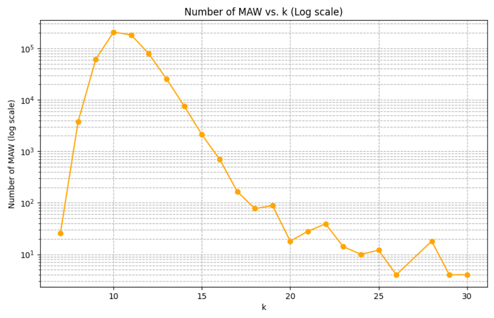

Figure 4. Number of MAWs vs. k (log scale)

The results are summarized in the table below:

Table IX
Optimal max_length for Capturing 99.9% of MAWs

$\|S\|$	Optimal max_length
$1,022$	13
$2,467$	14
$3,828$	19
$7,005$	22
$10,031$	16
$15,398$	14
$25,014$	18
$50,100$	16
$100,021$	16
$200,231$	16

We thus choose max_length $= 20$ as the optimal value.

D. *Influence of p on p-MAWs Computation*

We reuse the previous experiments to evaluate the influence of p.
We fix max_length to 20 and consider the first 10 sequences from the database (134,764 bases). We first compute the MAWs, then the p-MAWs for varying p.
The results are as follows:

Table X
Influence of p on p-MAWs Computation

p	User Time
0.3	0m13.544s
0.4	0m13.444s
0.5	0m13.949s
0.6	0m14.133s
0.7	0m13.555s
0.8	0m13.994s
0.9	0m21.993s

E. *Experimental Evaluation of γ_p and δ_p*

Under the same conditions as the previous experiment, we experimentally evaluate the two parameters.
The results are as follows:

Table XI
Experimental Values of γ_p and δ_p

p	γ_p	δ_p
0.3	0.01	0.77
0.4	0.02	0.65
0.5	0.03	0.56
0.6	0.06	0.52
0.7	0.12	0.48
0.8	0.23	0.45
0.9	0.43	0.42

From both experiments, we observe that p influences γ_p and δ_p. As p approaches 1, γ_p increases while δ_p decreases. Thus, p has a (slight) impact on computation time through the two parameters.
However, we note that $4 \times \gamma_p \times \delta_p$ remains well below 1 (for instance, it is 0.23 for $p = 0.7$), indicating that the computation does not enter an exponential regime.

V. Discussion

The experiments conducted show a linear influence of both sequence length and maximum motif size max_length on the computation time of MAWs. These results confirm the theoretical analyses presented in the methods section. However, some aspects need to be further explored and optimized. Notably, we observed in practice a complexity depending on max_length, while the worst-case scenario theoretically depends on max_length^2. It would be interesting to identify the conditions under which the worst-case complexity is reached.

Further work is required to eliminate more redundant computations during the search for p-MAWs. Although the implementation of a caching mechanism has limited some repetitive calculations, a more efficient data structure or a more optimized filtering method could further reduce these redundancies. This approach is particularly promising for improving algorithm performance on large sequence datasets or with high *max_length* values.

Additionally, the experimental evaluation of the parameters γ_p and δ_p shows that these values vary depending on the threshold p. However, a more in-depth theoretical analysis is needed to establish precise bounds and, if possible, demonstrate that $4\gamma_p\delta_p$ is less than 1, thus avoiding exponential complexity (at least for a reasonably chosen p).

Future work should focus on improving the verification step for p-MAWs and conducting a more thorough theoretical analysis of the parameters introduced in this study.

References

[1] F. Mojica, C. Diez-Villaseñor, E. Soria, and G. Juez, "Biological significance of a family of regularly spaced repeats in the genomes or Archæa, Bacteria and mitochondria", in Molecular Microbiology, vol. 36, april 2000, pp. 244-246

[2] https://github.com/AlexeyG/PySAIS

[3] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting", in 2009 data compression conference, pp. 193-202, IEEE

[4] https://pypi.org/project/suffix-tree/

[5] E. Ukkonen, "On-line construction of suffix trees", in Algorithmica, vol. 14, no 3, 1995, pp. 249-260

[6] https://t.ly/uXirO