Project Report

Axel Stengel and Romain de Beaucorps

December 2023

1 Introduction

1.1 Context

With the discovery of CRISPR, the need for identifying minimal absent words in a dataset has appeared. Indeed, viruses and plasmids have evolved to avoid certain patterns that are known in databases, since CRISPR functions like a software antivirus : it compares DNA with a database of known "malicious" code, and destroys any molecule that matches.

1.2 The problem

The problem we aim to solve is finding all the Minimal Absent Words (MAWs) in a collection of sequences of characters in the alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$

Reverse Complement. The reverse complement of a word is obtained by reversing it and switch A with T, C with G and vice versa.

Absent Word. A word ω is absent of a sequence S if neither ω nor its reverse complement are substrings of S. ω is absent from a set of sequences if it is absent of all the sequences.

Minimal Absent Word. A minimal absent word is a word of which all the substrings are present in the dataset, while being absent itself.

2 Methods

2.1 Naive approach

The first algorithm we implemented was very naive : generate all the strings of length k, then check manually if all their substrings are present and if the word itself is absent.
The implementation is detailed in Algorithm 1

2.1.1 Pseudocode

Algorithm 1 Naive algorithm for finding all MAWs of length up to k_{max} in a collection of sequences S

```
$MAWS \leftarrow \emptyset$
for $k = 0 \rightarrow k_{max}$ do
    $words \leftarrow$ all_combinations($\Sigma, k$)    ▷ all_combinations($\Sigma, k$) returns all the possible strings of length $k$ on the alphabet $\Sigma$
    for $word \in words$ do
        if is_MAW($word, S$) then    ▷ The implementation of is_MAW is described in the next algorithm
            $MAWS.append(word)$
        end if
    end for
end for
return $MAWS$
```

Algorithm 2 Naive implementation of `is_Maw` for deciding if a word x is a MAW in a collection of sequences S

```
for $s \in S$ do
    if ¬is_absent_in($x, s$) then
        return False
    end if
end for
$subs \leftarrow$ substrings($x$)
for $sub \in subs$ do
    for $seq \in S$ do
        if is_absent_in($sub, s$) then
            return False
        end if
    end for
end for
return $True$
```

However, this algorithm performs very poorly even for relatively low values of k on a very small dataset comprising only a single sequence of length $n \approx 9000$.

2.1.2 Complexity analysis

Notations. In this section and the other complexity studies, we will be referring to the total number of base pairs in the dataset as n, the length at which the MAWS are searched as k, and the first k at which MAWs are found as k_{min}.

The poor performances of this algorithm can be explained by its very bad time complexity, starting with `is_Maw` :

- There are $O(k^2)$ substrings of x, which we must all check in the worst case (see Algorithm 2)
- Checking if a word is absent in the dataset is done in $O(n)$ time in the worst case
- However, since every substring needs to be copied, the generation of the substrings happens in $O(k^3)$ time

- This makes the time complexity of `is_Maw` $O(k^3 + nk^2) = O(k^2(k + n))$ in the worst case

Now for the algorithm itself :

- We iterate on all the possible strings of length k for alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$, of which there are 4^k
- We call `is_Maw` inside the loop, of complexity $O(nk^2)$
- We obtain that the time complexity of the naive algorithm is $O(nk^2 4^k)$ in the worst case

The memory complexity however is $O(n)$, we just need to store the dataset which is negligible in our case as the bottleneck is the time complexity.

2.2 Breadth-First Search

Upon seeing the poor performances of the naive algorithm, we considered using an algorithm inspired by breadth first search.
The idea is to start with only the words of length 1 (the alphabet, Σ) and construct the words of length 2 by appending all the possible characters at the end of each word, and so on for each length k.
When we encounter an absent word, we do not keep exploring it. We therefore know it is almost a MAW, but it still could contain another absent word at its end that we have already seen (since it is of smaller length).
The algorithm is detailed in Algorithm 3

2.2.1 Complexity analysis

For all $k < k_{min}$, we simply iterate over all the 4^k possible words of length k, and check if they appear in the sequences. This has time complexity $O(n4^k)$, as discussed before.

As soon as we encounter absent words (so for $k \geq k_{min}$), we start pruning the search tree. This will depend on the proportion of words of length k that are absent from the input sequence, which we will call $\alpha(k)$. We define the number of words that are absent of length less than k as $\Psi(k) = \sum_{i=k_{min}}^{k-1} \alpha(i) \times 4^i$. We know that we have seen all these words before, as they have length $< k$, and they have therefore been pruned. This means that instead of the 4^k words to search, we only have $4^k - 4\Psi(k)$, and that the time complexity is reduced to $O(n(4^k - 4\Psi(k)))$

We can even extend this result to $k < k_{min}$, as an empty product has value 1 and we find $O(n4^k)$ again. The BFS algorithm therefore has $O(n(4^k - 4\Psi(k)))$.

Since it stores all the words to explore, this algorithm also has memory complexity of $O(4^k - 4\Psi(k)$.

2.2.2 Pseudocode

Algorithm 3 BFS algorithm for finding MAWs up to $kmax$ in a collection of sequences S

```
$current \leftarrow \Sigma$
$k \leftarrow 1$
$next \leftarrow \emptyset$
$MAWs \leftarrow \emptyset$
while $k \leq kmax \wedge \#current > 0$ do
    for $word \in current$ do
        for $maw \in MAWs$ do
            if $maw$ in $word \vee maw$ in reverse_complement($word$) then
                go to next word                ▷ It is absent but not minimal, stop exploring it
            end if
        end for
        for $seq \in S$ do
            if ¬is_absent_in($word, seq$) then          ▷ It is not a MAW, we keep exploring
                for $c \in \Sigma$ do
                    $next.append(word + c)$              ▷ + represents string concatenation
                end for
                go to next word
            end if
        end for
        $MAWs.append(word)$
    end for
    $k \leftarrow k + 1$
    $current \leftarrow next$
    $next \leftarrow \emptyset$
end while
return $MAWs$
```

2.3 Unword

The broad idea behind the `unword` algorithm is based on the fact that accessing an element in an array is constant in time. So if we translate each word into unique integers we can then store information about them in an array access that information in constant time. So the `unword` algorithm will use bit-array for each word size to store the information of presence : $A[index(w)]$ is 1 if and only if the word w is present in the sequences S. For the index translation we use the natural approach :

$$index(w) = \sum_{k=1}^{|w|} c(w[i]) \cdot 4^{|w|-i}$$

where $c(w[i])$ is the cost of the i-th letter of w. We set $c(A) = 0,\ c(C) = 1,\ c(G) = 2,\ c(T) = 3$.

2.3.1 Pseudo-code

The `scan(S,k)` function defined in Algorithm 4 scans the sequences of S for words of length k and stores the information in an array A of bits of length 4^k (number of different word of length k) where $A[index(w)]$ is 1 if and only if w is present in S.

The `is_MAW(w,A)` function defined in Algorithm 5 checks if the absent word w of length $k > k_{min}$ is a MAW. Because we know that all words of length under k_{min} are present in S, we only have to check the sub-strings of length over k_{min}. We can then use the list of bit array $A = [A_{k_{min}}, ..., A_k]$

Algorithm 4 Algorithm for scanning a list of sequences S for words of length k

```
$A_k \leftarrow bit\_array(4^k)$                    ▷ $A_k$ is initialized with all bits to 0
for $i = 0 \rightarrow |S| - 1$ do
    $j \leftarrow 0$
    $s \leftarrow S[i]$
    while $j < |s| - k$ and $A_k.count(1) \neq 4^k$ do
        $w = s[j : j + k]$
        $A_k[index(w)] \leftarrow 1$
        $A_k[index(\overline{w})] \leftarrow 1$
    end while
end for
    return $A_k$
```

to know if a certain sub-string is present is S or not, and we stop at the first sub-string not present in S. Because larger words have a higher chance of being absent, by generating the sub-strings from largest to smallest we statistically stop sooner, saving costly compute time.

Algorithm 5 Algorithm for verifying if an absent word w of length $k > k_{min}$ is a MAW of S using the list of bit array $A = [A_{k_{min}}, ..., A_k]$

```
$substrings \leftarrow [w[i : j] | j - i > k_{min}]$          ▷ $suubstrings$ is sorted by decreasing size
for $sub \in substrings$ do
    if $A_{|sub|}[index(sub)] = 0$ then return False
    end if
end for
    return True
```

Finally, the `unword` algorithm is described in Algorithm 6. The algorithm is divided into two sections, the first one finds the first $k = k_{min}$ where there are MAWs of length k and the second computes all MAWs of length $k \leq k_{max}$.

2.3.2 Time Complexity

Before looking at the complexity of the global `unword` algorithm, we first need to analyze all auxiliary functions.

- The `scan` function defined in Algorithm 4. `scan(S,k)` first creates an array of size 4^k which is in $O(4^k)$ time. It then scans the sequences of S with a sliding window of size k and computes the index of the current word and its reverse complement until it either finds 4^k different words or reaches the end of S. The computation of the first index is trivially in $O(k)$ but once we know the index of a word $w = s[j : j + k]$ we can compute the index of $w' = s[j + 1 : j + 1 + k]$ in $O(1)$ time using the following formula :

  $$index(s[j + 1 : j + 1 + k]) = 4 \times (index(s[j : j + k]) - 4^{q-1} \times c(s[j])) + c(s[j + k])$$

  In the same way we can compute the index of its reverse complement using the following formula :

  $$index(\overline{s[j + 1 : j + 1 + k]}) = index(\overline{s[j : j + k]}//4 + 4^{q-1} \times c(\overline{s[j + k]})$$

  Therefore, the amortized computation cost of indexes is $O(1)$ and so the `scan` function has a time complexity of $O(4^k + n)$.

Algorithm 6 Global Algorithm for finding all MAWs of length up to k_{max} of a list of sequences S

```
$k \leftarrow 2$
$A_k \leftarrow scan(S, k)$
while $A_k.count(1) = 4^k$ and $k < k_{max}$ do                ▷ first loop to find $k_{min}$
    $k \leftarrow k + 1$
    $A_k \leftarrow scan(S, k)$
end while
$k_{min} \leftarrow k$
$A \leftarrow [A_{k_{min}}]$
$MAW_S \leftarrow [(k_{min}, absent\_words(A_{k_{min}}))]$
while $k < k_max$ do                ▷ second loop to find all MAW of length $k \leq k_{max}$
    $k \leftarrow k + 1$
    $A_k \leftarrow scan(S, k)$
    $A \leftarrow A + [A_k]$
    $MAW_k \leftarrow [\ ]$
    for $w \in absent\_words(A_k)$ do
        if $is\_MAW(w, A)$ then
            $MAW_k \leftarrow MAW_k + [w]$
        end if
    end for
    $MAW_S \leftarrow MAW_S + [(k, MAW_k)]$
end while
    return $MAW_S$
```

- The `absent_words` function. `absent_word(A_k)` iterates 0(1) computations through a 4^k sized array, so is in $O(4^k)$ time.
- The `is_MAW` function defined in Algorithm 5. `is_MAW(w,A)` first generates all $O(k^2)$ sub-strings of w of length greater than k_min which is in $O(k^3)$ time because of list copy time. It then iterates over at most of all sub-strings and calculates its index. Here we cannot use the $O(1)$ amortized computation cost of indexes because the words are not necessarily in consecutive order, so the index computation is in $O(k)$. So the `is_MAW` function is in $O(k^3)$ time.

As explained in the previous subsection, the `unword` algorithm is divided into two distinct sections, the first one finds the first $k = k_{min}$ where there are MAWs of length k and the second computes all MAWs of length $k \leq k_{max}$.

If we consider $k_max > k_min$ then the first loop iterates `scan(S,k)` for $k = 2$ to k_min so its complexity is $O\left(\sum_{k=2}^{k_{min}} 4^k + n\right) = O\left(4^{k_{min}} + k_{min}n\right)$. Moreover, we can find an upper bound for k_{min}. Indeed, if we consider $k' = \lceil \log_4(n) \rceil$ then $4^{k'} \geq n > n + 1 - k'$, however S only contains $n + 1 - k'$ sub-strings of length k' so there are some absent words of length k' and so $k_{min} \leq k' = \lceil \log_4(n) \rceil$. This means that we can further simplify the time complexity of the first loop as follows :

$$O\left(4^{k_{min}} + k_{min}n\right) = O\left(4^{\log_4(n)} + \log_4(n)n\right) = O(\log(n)n)$$

In the second section of the algorithm, for each $k = k_{min} + 1$ to k_{max}, we first call `scan(S,k)` and then check `is_MAW(w,A)` for all absent words (found by calling `absent_words(A_k)`). So if we call z_k the number of absent words of length k we have the following complexity :

$$O\left(\sum_{k=k_{min}}^{k_{max}} 4^k + n + 4^k + z_k(k^3)\right) = O\left(4^{k_{max}} + k_{max}n + \sum_{k=k_{min}}^{k_{max}} z_k(k^3)\right)$$

However, the only upper bound for z_k is 4^k so the worst case time complexity of the second loop is $O\left(4^{k_{max}} + k_{max}n + {k_{max}}^3 4^{k_{max}}\right) = O\left(k_{max}(n + {k_{max}}^2 4^{k_{max}})\right)$.

2.3.3 Space Complexity

The `unword` algorithm stores each result of its `scan(S,k)` calls for $k = k_{min}$ to k_{max} in memory. We can easily find that the size of this result is in $O(4^k)$. It also stores the input file in memory, so the space complexity of the `unword` algorithm is in $O\left(n + \sum_{k=k_{min}}^{k_{max}} 4^k\right) = O\left(4^{k_{max}} + n\right)$

3 Results

All experiments were conducted on a desktop PC using one core of an AMD Ryzen 9 5900X at 3.700GHz, and with 32Gb of DDR4 RAM.

As previously mentioned, the naive algorithm is really slow ; it still manages to find absent words for k up to 10 in a reasonable time (10 minutes) on a single input sequence of around 9000 base pairs. Its memory usage is basically negligible, especially compared to the processing time.
The BFS algorithm performs relatively well, especially compared to the naive algorithm. Since we encounter a growing amount of absent words with each k, we can prune that many branches each k. In fact, as we see in Table 1, since we start finding a lot of absent words at around k = 6, we can prune a lot of branches and maintain an almost constant execution time with regard to k. However, on more complete datasets, since absent words do not start appearing before larger values of k ($k = 11$ for the second dataset), it can not prune and must search exhaustively which as we can see in Table 2 is not feasible.
The unword algorithm finds the first minimal absent words virtually instantly, but struggles to keep searching deeper for two reasons : it resorts to "manually" verifying if an absent word is minimal by checking its substrings (recall there is k^2), and it has to allocate a bitarray size 4^k, which we can observe in the execution times in both Tables 1 and 2 (notice the $4\times$ factor between subsequent runs starting at $k = 12$).

k	naive	bfs	unword
5	00:00.03	00:00.07	00:00.00
6	00:00.48	00:00.89	00:00.00
7	00:04.10	00:05.78	00:00.02
8	00:23.96	00:26.98	00:00.15
9	01:50.46	01:00.99	00:00.72
10	07:47.39	01:20.64	00:03.26
11	31:31.06	01:28.56	00:15.11
12	too long	01:31.04	01:06.57
13	too long	01:31.55	04:56.05
14	too long	01:32.21	21:40.94

Table 1: Results for about 9000 base pairs shared across one sequence

k	bfs	unword
5	00:00.05	00:00.00
6	00:00.76	00:00.02
7	00:12.63	00:00.24
8	03:45.38	00:01.06
9	1:04:15	00:04.11
10	too long	00:43.63
11	too long	02:00.74
12	too long	02:08.29
13	too long	02:47.72
14	too long	13:16.21

Table 2: Results for about 223 million base pairs shared across around 2800 sequences

4 Discussion

As expected, the naive algorithm is pretty much worthless as soon as we step into real data. We are very happy with the performances of the BFS algorithm on small data, however its inability to scale to larger data is very displeasing.
The unword algorithm is very satisfying for finding the first MAWs, and can push a bit further but starts to slow down and most importantly uses a lot of memory for $k = 15$ and above on the largest dataset.
The first improvement to try would intuitively be to combine the two approaches : use the unword algorithm for finding the first few levels of MAWs, and switch to a BFS implementation when the search space has been reduced enough.

Regarding the limitations of our algorithms, we didn't include the timings over $k = 14$ for a simple reason : it either became too long to run, or used too much memory, or both. A simple fix would be to implement the algorithms in a more efficient language than python, both in time and memory, like c++ or rust. Another possibility would simply be to find better algorithms, we read some about some implementations using suffix trees and suffix arrays.

In conclusion our most efficient algorithm, unword, allowed us to find the Minimal Absent Words of length k up to 15 on a large collection of sequences, but is nearly not enough to search through something as big as the entire genome for example.

5 Bibliography

References

[1] J. Herold, S. Kurtz, and R. Giegerich, "Efficient computation of absent words in genomic sequences," *BMC Bioinformatics*, vol. 9, no. 1, p. 167, Mar. 2008, ISSN: 1471-2105. DOI: `10.1186/1471-2105-9-167`. [Online]. Available: `https://doi.org/10.1186/1471-2105-9-167`.

[2] C. Barton, A. Heliou, L. Mouchard, and S. P. Pissis, "Linear-time computation of minimal absent words using suffix array," *BMC Bioinformatics*, vol. 15, no. 1, p. 388, Dec. 2014, ISSN: 1471-2105. DOI: `10.1186/s12859-014-0388-9`. [Online]. Available: `https://doi.org/10.1186/s12859-014-0388-9`.