Project Report

A contribution to the Minimal absent words identification problem

Victor Jourdan and Armand Ledoux

After the discovery of CRISPR, one of the most important discoveries in the field of bioinformatics, working as an adaptive immunity mechanism, used by bacteria to fend off phages and predatory plasmids. To avoid being eliminated by this biological antivirus, plasmids and viruses have developed strategies, such as avoiding distinct patterns, and developed an internal database, like CRISPR. We tried to design an algorithm to efficiently identify minimal absent ords in a plasmid dataset to better understand the patterns potentially targeted by CRISPR.

1 Introduction

The discovery of the CRISPR (Clustered Regularly Interspaced Short Palindromic Repeats), a mechanism used by bacteria to fight against phages and plasmids has been a major breakthrough in bioinformatics. Plasmids and viruses have developed counter-measures, such as avoiding certain patterns thanks to an intern database. The goal of this project is to develop efficient algorithms to list the minimal absent words of a DNA sequence or a set of DNA sequences.

1.1 Paper overview

In this paper we present different versions of algorithms to solve the MAW problem. First, we will introduce the concepts manipulated in this paper, then we will present the naïve and unsatisfying solution. Then we will study our various improvements of the naïve solution, with a complexity analysis of them. Comparison in terms of computation time and space usage will conclude our report.

2 Framework

We will consider the following alphabet, $\Sigma = \{\text{A, C, G, T, N}\}$, the four nucleotides and the N, signifying unknown. A DNA sequence is a string over this alphabet $s \in \Sigma$. A k-mer is a substring of length k of s. An absent word is a word that is not a substring of s. A minimal absent word (MAW) is an absent word of s such that every proper substring of it is a substring of s. For a given $p \in (0, 1]$, and a set of sequences S, a pMAW x is a word that is an absent word in at least $p \cdot |S|$ sequences of S, and such that every proper substring of x is not a pMAW of S. Note that for selecting substrings of any s, we will write it $s[i : j]$ which means the letters i...j-1 of the string, the first letter having index 0. We will write the relation "x' substring of x" as $x' \subset x$, and being a proper substring will be written $x' \subsetneq x$.

3 First Problem

Let us first investigate algorithms to detect the MAWs of a string, and find the one with an optimal complexity.

3.1 Naïve algorithm

The only optimization we allowed ourselves to do was to test, for each MAW candidate x, only its two maximal proper substrings : $x[0 : |x| - 1]$ and $x[1 : |x|]$, because if each of them is a substring of s, each proper substring of x being itself a substring of those two, we verify the MAW requirements and if one of them in not in s then we clearly do not verify the requirements. We saved a bit on complexity.

```
function NAÏVE MAW ALGORITHM(s, k)
    ▷ Brute force over all possible strings of size <= k
    maw_set ← {}
    for $i \in [\![1, k]\!]$ do
        for $x \in \Sigma^i$ do
            if $x \not\subset s$ and $x[0 : |x| - 1] \subset s$ and $x[1 : |x|] \subset s$ then
                maw_set ← maw_set ∪ {canonical($x$)}
    return maw_set
```

The algorithm's time complexity is as follows : we consider every word of length $\leq k$, which are of number $\sum_{i=0}^{k} 5^i = \frac{5^{k+1}-1}{5-1} = \Theta(5^k)$, and for each of them we do a linear complexity test in the length of s, considering we use an optimal substring testing algorithm. which results in a complexity of $O(5^k \cdot n)$. As we only need to store the MAWs, the space complexity is of $O(\sigma \cdot n)$, as proven by [1]. We could have given them sequentially if needed to reduce space usage. This algorithm, although very simple, fares too poorly even on small inputs. We had to design a second algorithm.

3.2 First improvement

This simple improvement relies on the aforementioned observation: for a canonical word x to be an MAW of s, both $x[0 : |x| - 1]$ and $x[1 : |x|]$ have to be in s. So let us list all substrings of length $\leq k$, and let us consider each one of them, add a letter in front or at the end (we will only do at the end because if this substring is the end maximal proper substring of another MAW, it will be handled by the front maximal substring.), and let us see if the completed word is not in s and if the corresponding second maximal substring is in s.

```
function FIRST IMPROVED ALGORITHM(s, k)
    for $i$ in $[\![1, k]\!]$ do
        dic$_i$ ← all i-mers of $s$
    pmaw_set ← {}
    for $i$ in $[\![1, k]\!]$ do
        for $x$ in dic$_i$ do
            for $l \in \Sigma$ do
                if $x + l \notin$ dic$_{i+1}$ and $x[1 : |x|] + l \in$ dic$_i$ then
                    pmaw_set ← pmaw_set ∪ {canonical($x$)}
    return pmaw_set
```

*Corresponding author

The algorithm's time complexity is equivalent to $O(k^2 \cdot n)$, $k \cdot n$ for the worst case number of i-mers, and each i-mer needs a test of time linear in k on average, with a good dictionary implementation. So it gives us on average a complexity of $O(k^2 \cdot n)$. This is much better than the naïve version. In terms of space complexity, it is, as the time complexity, $O(k^2 \cdot n)$ in the worst case. To reduce this space complexity, we could only keep, for i, doc_i and doc_{i+1}, to transform this complexity in $O(k \cdot n)$.

3.3 Number comparisons

The brute force version is horrible even for small values of k, needing more than 30secs for just one sequence and a kmax of 6. For the first improvement, with the first 10 sequences of the eba_plasmids file :

kmax	number of sequences	time
10	10	$0.403s$
20	10	$1.574s$
40	10	$4.10s$

it obviously depends on the size of each sequence.

4 Second Problem

4.1 Naïve algorithm

Now that we have good algorithms to find the MAWs of a sequence, let us also consider first the naive version of the algorithm. Given $p \in (0, 1]$, we find all pMAWs in S a set of sequences.

```
1: function NAÏVE PMAW ALGORITHM(s, k)
2:     ▷ Brute force over all possible strings of size <= k
3:     pmaw_set ← {}
4:     for $i \in [\![1, k]\!]$ do
5:         for $x \in \Sigma^i$ do
6:             if $\frac{|\{s \in S:\ x \subset s\}|}{|S|} < (1-p)$ and $\forall x' \subsetneq x, x' \notin$ pmaw_set then
7:                 pmaw_set ← pmaw_set ∪ {canonical($x$)}
8:     return maw_set
```

Complexity analysis :

4.2 First improvement

Let us consider the presence ratio $r(x) = \frac{|\{s \in S:\ x \in s\}|}{|S|}$. Firstly, for a given word x, we observe that it is decreasing with the $\subset$ relation: for each string in which x appears, $x[0 : |x| - 1]$ and $x[1 : |x|]$ do also appear, while the converse is not necessarily true. Secondly, we observe that if $r(x) < 1 - p$ and $\forall x' \subsetneq x, r(x') \geq 1 - p$, then no x' can be a pMAW, and therefore x is a pMAW. Now if one of the two maximal proper substrings of x has a rate inferior to $1 - p$, it either has a pMAW as a substring, because if every one of its substrings has a rate inferior to 1 - p, then the words of length one will evidently be pMAWs. So to determine if x is a pMAW, we just need to know that $x[0 : |x| - 1]$ and $x[1 : |x|]$ have a rate superior or equal to $1 - p$ and that $r(x) < (1 - p)$. Thus our inductive reasoning : knowing the words of length i that have a rate superior or equal to $1 - p$, we consider all possible combinations of two of them as two maximal proper substrings of an x, and we calculate $r(x)$ by combining looking at every set where both of them appear, and if it is not a MAW, then it appears in it.

4.3 Number comparisons

The brute force version is horrible even for small values of k, needing more than 30secs for just one sequence and a kmax of 6. For the first improvement, we selected the k high enough to store almost all pmaws without spending too much time on kmer generation. Jith a the first sequences of the eba_plasmids file :

kmax	number of sequences	p	time
10	10	0.25	$0.269s$
10	20	0.25	$1.04s$
10	100	0.25	$7.71s$
10	10	0.5	$0.330s$
10	20	0.5	$1.08s$
10	100	0.5	$8.55s$
10	10	0.75	$0.342s$
10	20	0.75	$1.33s$
10	100	0.75	$10.7s$

We observe that the time increases slightly with p, albeit not significatively, which is logical because we allow words to stay in the "present_words" longer with a higher p. As for the complexity regarding the number of sequences, it depends on the size of the sequences, so I cannot give a result for that.

```
1: function FIRST IMPROVED ALGORITHM(s, k)
2:     for $i$ in $[\![1, k]\!]$ do
3:         $\mathrm{dic}_i$ ← all i-mers of $s$
4:     ▷ We initialize by doing manually the calculus for the letters, i.e. the 1 letter words
5:     pmaw_set ← {every letter that is p-absent}
6:     present_words ← {every letter that is not p-absent}
7:     for $i$ in $[\![1, k]\!]$ do
8:         for $x \in$ present_words of length i do
9:             for $x' \in$ present_words of length i $s.t. x'[0 : |x'|] = x[1 : |x|]$ do
10:                if $|\{x, x' \in s \land x$ not a maw of $s\}| > p \cdot |S|$ then
11:                    pmaw_set ← pmaw_set ∪ {canonical($x + x'[-1]$)}
12:                else
13:                    present_word ← present_word ∪ {canonical ($x + x'[-1]$)}
14:    return pmaw_set
```

Complexity analysis : it is difficult to evaluate the complexity, but if we take the bound of MAWS being of number $O(\sigma \cdot n)$, we may have a complexity

We observe that the higher p is, the bigger the longest pMAW's length will be, that is logical because we allow more absence before turning a kmer into a pMAW.

Conflicts of Interest

The authors have no conflicts of interest to declare. All co-authors have seen and agree with the contents of the manuscript and there is no financial interest to report.

Remarks

This project has been jointly led by Armand and Victor. The estimated contribution time is of 25 to 30 hours for Victor and 3 to 5 hours for Armand. There has been no exchange of code with other groups, we only had brief theoretical discussions with Erwan about different strategies, albeit no time to implement a complex algorithm for the first question as time was dedicated to trying to find a link between MAWs and pMAWs… The project was fun to do overall, with theoretical and practical issues. Just stumbling on the link problem has given me a headache.

Notes and References

The algorithms have been thoughts and designed before reading the state of the art.

[1] M. Crochemore, F. Mignosi, and A. Restivo, "Automata and forbidden words," *Information Processing Letters*, vol. 67, no. 3, pp. 111–117, 1998.