BOX Project

Benjamin WOJTECKI[1] and Noé VINCENT[1]

[1]Département d'informatique, École Normale Supérieure de Rennes, France

LARGE scale genomic sequence analysis and the discovery of meaningful patterns in DNA have been made possible by advances in bioinformatics. An important biological system studied using sequence analysis is the adaptive immune system CRISPR-Cas, which enables bacteria to identify and eliminate foreign genetic material such as plasmids [3]. This recognition is based on short DNA sequences, making computational methods important for understanding these mechanisms. From sequence analysis, the absence of particular DNA words can indicate selective pressure acting on genomes. This observation led to the idea of minimal absent words (**MAWs**), defined as the shortest DNA sequences that do not occur in a genome while their shorter substrings do occur. Several bioinformatics studies have used MAWs, which offer a compact representation of sequence avoidance [2].

In this work, we focus on the computation of minimal absent words in a large dataset of plasmid DNA sequences. As is common in DNA sequence analysis, we use canonical representations of sequences that take reverse complements into account. In addition to MAWs, we consider minimal p-absent words (**pMAWs**), which are absent in a given proportion of sequences in the dataset. This generalization allows the identification of sequence patterns that are consistently avoided across multiple plasmids. This work has two main purposes. First, we aim to implement an algorithm to enumerate minimal absent words up to a user-defined maximum length. Second, we perform an analysis of the resulting MAWs and pMAWs to identify common patterns of sequence avoidance in plasmid genomes, with the broader goal of providing understanding into motifs potentially targeted by CRISPR-Cas systems.

Methods

This section describes the methodologies implemented to solve the objectives of the project. First, we present **Program1**, which focuses on the enumeration of minimal absent words for individual DNA sequences. The methodology of **Program2** is described in the next subsection.

Program1: Enumeration of Minimal Absent Words

The goal of Program1 is to enumerate all minimal absent words of length at most $k_{\max}$ for each DNA sequence provided in a FASTA file. Let us recall the formal definition of Minimal Absent Words.

Definition 1. *Let x be a string of length $|x| > 1$ and S be a string. We say that x is a minimal*

absent word of S if both the following conditions hold:

- *x is an absent word of S*
- *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*

Each DNA sequence is processed using the same computational pipeline. The overall workflow is composed of four main steps, illustrated in Figure 1.

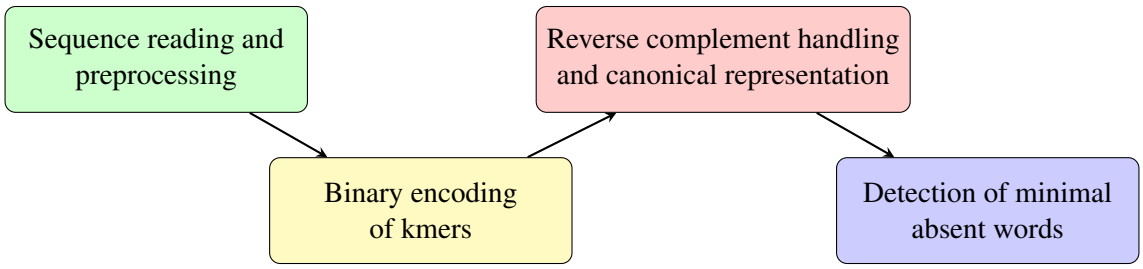


Figure 1: Overview of the computational workflow

Sequence reading and preprocessing
Input sequences are read from a FASTA file using the `readfq` function, which was provided during a practical session. This function allows efficient parsing of FASTA formatted data. Each DNA sequence is processed independently. Only the standard DNA alphabet $\Sigma = \{A, C, G, T\}$ is considered, any ambiguous nucleotide interrupts the kmer encoding process. We make this decision because, even though an ambiguous nucleotide may carry some information, it does not make sense to include ambiguous nucleotides in a biological context disambiguation seemed out of the scope of this study and our knowledge.

Binary encoding of k-mers
To achieve efficient memory usage and fast comparisons, DNA sequences are encoded using a 2 bit representation per nucleotide. Each nucleotide is mapped to an integer as follows: $A \to 0$, $C \to 1$, $G \to 2$, and $T \to 3$. Then, a kmer is represented as an integer in the range $[0, |\Sigma|^k - 1]$, where $|\Sigma| = 4$. For a given sequence and a fixed value of k, all kmers are generated using a sliding window. The encoding is performed incrementally using bitwise operations. At each step, the current encoded value is left-shifted by two bits, the new nucleotide is appended using a bitwise OR, and a bit mask is applied to keep only the last $2k$ bits. This allows all valid kmers to be extracted in a single pass over the sequence without recomputing the encoding from scratch. Letters outside the alphabet Σ reset the current window and prevent the generation of invalid kmers. This streaming encoding procedure runs in linear time with respect to the sequence length, with a time complexity of $\mathcal{O}(n)$, since a constant number of bitwise operations is performed per character. The encoding itself requires only constant memory, giving a space complexity of $\mathcal{O}(1)$.

Reverse complement handling and canonical representation
The reverse complement of a kmer is computed on its binary representation using bitwise operations. For each detected minimal absent word, its canonical form is defined by the following definition.

Definition 2. *The canonical version of a sequence $\mathcal{S}$ is the lexicographic minimum between itself and its reverse complement. Recall that the reverse complement of $\mathcal{S}$ is obtained by reversing it and transforming $A \to T$, $T \to G$, 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.*

This definition guarantees that each MAW is reported uniquely and independently of strand orientation.

Data structure for kmer presence
For each value of k, the set of kmers is stored using one of two data structures:

- a set of integers, when the number of $k$mers is small,
- a compact bit array, when the number of $k$mers becomes large.

The choice between these two representations is made based on the density of $k$mers. This strategy reduces memory usage while preserving constant time membership queries.

Detection of minimal absent words
Minimal absent words are detected iteratively for increasing values of k, from $k = 1$ up to $k_{\max}$. For $k = 1$, MAWs correspond to nucleotides from Σ that do not occur in the sequence. For $k \geq 2$, candidate words are generated by extending each $(k-1)$mer with one nucleotide from Σ. A candidate word x of length k corresponds to Definition 1. Candidate words that satisfy both conditions are stored in their canonical form.

Output
For each input sequence and each value of k, all detected minimal absent words are decoded back into DNA strings, lexicographically sorted, and written to a TSV file. Each output line contains the sequence identifier, the word length k, and a comma-separated list of canonical MAWs. Only values of k for which at least one MAW is detected are returned.

Program 2: Computation of pMAWs

The goal of Program 2 is to compute the pMAWs (of lenght upto k_{max} of a set $\mathcal{S}$ of sequences. To achieve this, the program is given, for each sequence $S \in \mathcal{S}$ the set of MAWs of S.
Definition 3. *Let x be a substring of $S \in \mathcal{S}$. We say that x is a Minimal p-Absent Word (pMAW) of $\mathcal{S}$ if the following condition hold:*

- $x > 1$ *is absent of at least* $p.|S|$ *sequences (x is p-absent)*
- $\forall w$ *proper substring of* x*, it holds that* w *is not* p*-absent.*

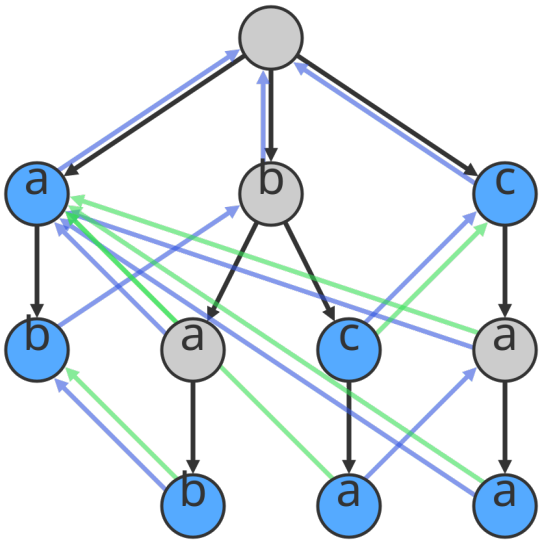


Figure 2: Aho-Corasick automaton for $H = \{a, ab, bab, bc, bca, c, caa\}$

Aho-Corasick algorithm
The Aho-Corasick [1] algorithm efficiently solves the problem of finding a needle in a haystack. Formally, let Σ be an alphabet, this algorithm solves:

INPUT : a haystack H (a finite set of words on Σ), a needle x (a word on Σ)

OUTPUT : $w \in H$ a substring of x.

To do so, this algorithm implements an Automaton (see, Figure 2) based on a trie, where each word of H is inserted and containing additional links between the nodes. In particular, those additional links are of two different kinds:

- Suffix links (in blue) linking a node to the longest possible suffix (of the word ending at said node) present in the graph.
- Dictionary links (in green) linking a node to the first terminating node (ie node where a word of H ends (blue nodes)) along a path of suffix links.

Finding pMAWs
Let x be a candidate, for x to be a pMAW, the two properties presented in definition 3 need to be verified.

To do so, we chose to maintain a trie T where for each word is stored an absenceMap as a bitarray of size $N = |\mathcal{S}|$:

$$B_w \in \{0,1\}^N, B_w[i] = 1 \leftrightarrow w \notin S_i$$

.

Then we had that, if $\#B_w \geq p.N$ then w is p-absent.

To initialize T, the algorithm starts with the canonical words of two letters and compute their AbsenceMaps. This is achieved by looking whether the word has a substring in the MAWs of each sequence using an Aho-Corasick automaton seeded with the set of MAWs (of length up to k_{max}) of that sequence.

The words of size two that are p-absent are then added to the set of compute p-maws. They verify the minimality condition, because each of their proper substring are not p-absent.

The set of candidates of size 3, is the canonical extensions of words of size 2 that are not p-absent.

Then the following general procedure is followed, here at round k :

candidatesNextRound $\leftarrow \emptyset$
for w in candidates **do**
    AbsMap = T.longestPrefix(w)
    for i, AbsMap$[i] \neq 1$ **do**
        subword = findNeedle($ACAutomata_i, w$)
        AbsMap = AbsMap $\bigvee$ T.longestPrefix(subword) $\bigvee M_i$
    end for
    $T[w]$ = AbsMap
    if w is p-absent **then**
        pMAWS$[k]$ = pMAWs$[k] \cup \{w\}$
    else
        candidatesNextRound = candidatesNextRound $\cup\{w\}$
    end if
end for
newCandidates $\leftarrow \emptyset$
for w in candidatesNextRound **do**
    for c in "ATCG" **do**
        newCandidates $\leftarrow$ newCandidates $\cup$\{canonical(w+c)\} $\cup$ \{ canonical(c+w)\}
    end for
end for
candidates = newCandidates

In the above algorithm $\bigvee$ is the bitwise OR, M_i is the bitarray with a 1 on the i-th bit.

This procedure is then repeated until the size of the candidates reaches k_{max} or the size of the set of candidates reaches 0 (ie, every smaller words are p-absent).

Note how T allows absMap to take into account the previously calculated absences for the prefixes of a word, but also the absences of potential subwords found with the call to Aho-Corasick algorithm for each sequence.

OUTPUT
When a pMAW of size k is found, it is added to pMAWs$[k]$, writing the data into an output file is then straightforward.

Results

In this section, we present the results obtained with Program1 and Program2.

Program1

Complexity analysis

The time and space complexity of Program1 mostly depend on the sequence length n and the maximum word length k. For a fixed value of k, the algorithm first scans the sequence once to generate all kmers, which takes $\mathcal{O}(n)$ time. Then, candidate words of length k are generated by extending all present $(k-1)$mers with the four possible nucleotides. In the worst case, the number of possible $(k-1)$mers is 4^{k-1}, which suggests 4^k candidate words. Each candidate is tested in constant time using either a set or a bit array. Therefore, the worst case time complexity for a given k is $\mathcal{O}(n+4^k)$. When considering all values from $k=1$ to $k_{\max}$, the time complexity is bounded by the largest value of k, making the complexity in the worst case $\mathcal{O}(n+4^{k_{\max}}) \equiv \mathcal{O}(4^{k_{\max}})$.

In terms of space complexity, storing the presence of all kmers may require up to $\mathcal{O}(4^k)$ memory. This explains why both memory usage and computation time grow very quickly for large values of k, and why the use of Program1 is limited to moderate values of $k_{\max}$. This analysis is confirmed by the graph of the computation time as a function of k (Figure 3).

Computation time as a function of k

Here, the computation time tests were done on a common laptop with AMD Ryzen 5 7535HS (6*3.33GHz) and 16 GiB of DDR5 4800MHz RAM, on the standard dataset "all_ebi_plasmids" (see GitHub).

Figure 4 shows the number of minimal absent words detected as a function of the maximum word length k.

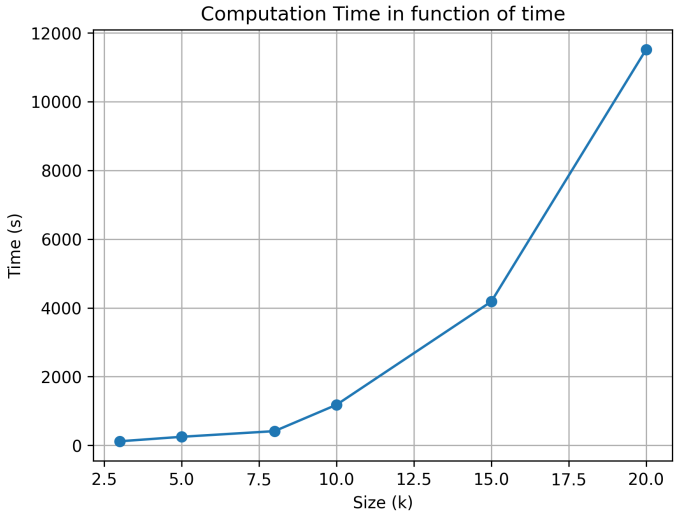


Figure 3: Computation time as a function of the maximum length k

We observe that the computation time increases significantly with k. This increase in runtime is correlated with the growth in the number of MAWs. Indeed, the algorithm must generate and test a large number of candidate words. Even with efficient data structures, the overall computational cost becomes dominated by the total number of MAWs. To conclude, these results highlight the practical limitations of MAW enumeration for large values of k.

Number of MAWs as a function of k
Figure 4 shows the number of minimal absent words detected as a function of the maximum word length k.

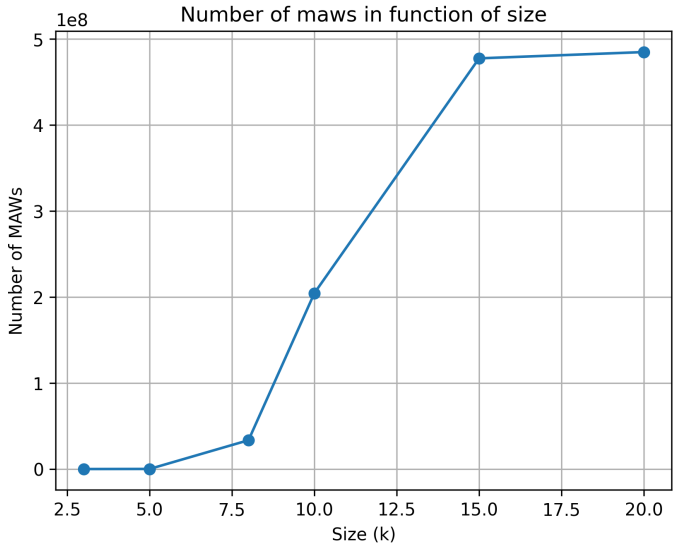


Figure 4: Number of minimal absent words as a function of the maximum length k

We observed that the number of MAWs increases very quickly when k becomes larger. For small

values of k, the number of MAWs is low. However, starting from $k = 8$, the number of MAWs grows extremely fast and reaches several hundreds of millions. This form can be explained because DNA sequences are not random. Indeed, certain motifs are frequent and these very specific kmers are always present while others are rare or absent, causing the number of minimal absent words (MAWs) to increase irregularly with k.

Program 2

Complexity analysis
First, the algorithm needs to create the $N = |\mathcal{S}|$ automata which has complexity in $O(\Sigma_{i=1}^{N} L_i)$ ([1]) where L_i is the number of MAW from sequence S_i. From [4] we have that $L_i \in O(|S_i|)$. Then computing the automata has time complexity in $O(size(\mathcal{S}))$ where $size(\mathcal{S})$ is the sum of the lengths of the sequences in $\mathcal{S}$.

The initializing of some data structures have time complexities in $O(N)$ which is negligible regarding the creation of the automata, others have time complexities in $O(k_{max})$ which is negligible regarding the complexity of the main loop.

The main loop is a loop of at most k_{max} round. Each round needs to consider each of the $O(4^k)$ candidates and in the worst case does N checks to the automaton, which has complexity in $O(k)$ (bound on the size of the needle). Then this main loop has complexity in $O(\Sigma_{i=2}^{k_{max}} 4^i N i) \in O(4^{k_{max}} N)$.

Outputting the pmaws is negligible regarding the main loop (mainly because each operation of the main loop cannot create more than one pMAW).

In a nutshell, program 2 has time complexity in $O(size(\mathcal{S}) + 4^{k_{max}} N)$.

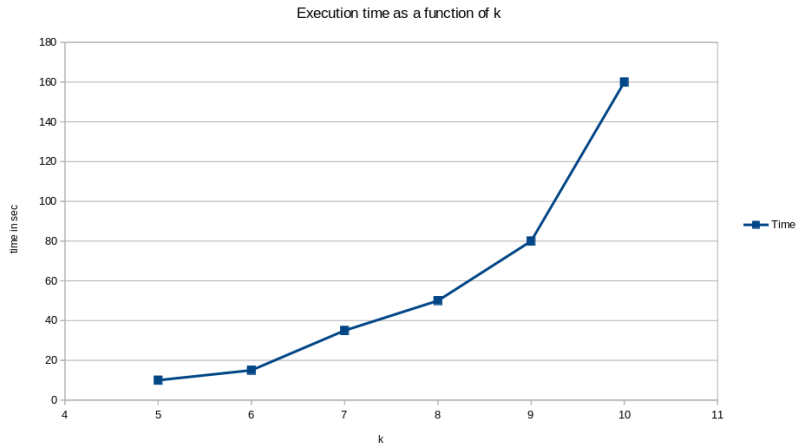
The space complexity is mainly driven by storing the automata and the trie. The automata has complexity in $O(L_i)$ which then scales up to $O(size(\mathcal{S}))$. The trie has space complexity in $O(4^{k_{max}} N)$.

Note that in reality the bounds on the automata are rarely reached because only the MAWs of length up to k_{max} are added in the automata. Seeing Figure 4, one can understand how this deeply improves real execution time.

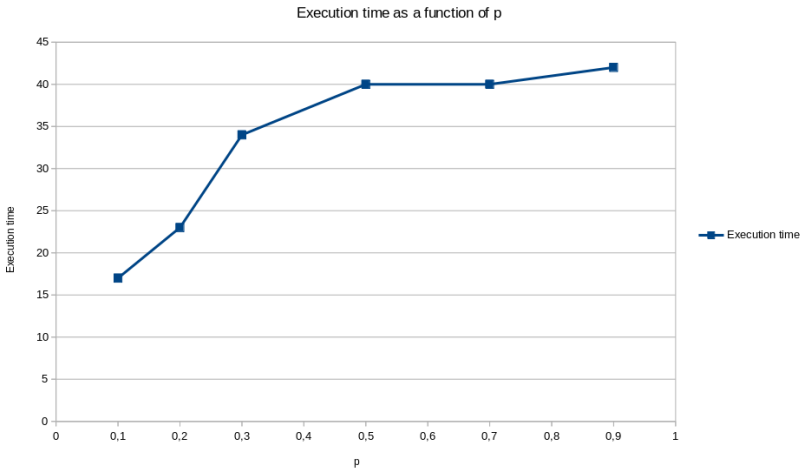
Computation time as a function of k and p
Here, the computation time tests were done on a common laptop with Intel Core i5-1135G7 (8*2.40GHz) and 8 GiB of DDR4 4267MHz RAM, on the output of program1 from the standard dataset "all_ebi_plasmids" (see GitHub).

On Figure 5a, one can see how the time of execution follows some kind of exponential growth in the value of k. Indeed, a lower value of k means less data to feed into the automata and less operations in the main loop.

Figure 5b, illustrates the fact that the value of p has a great impact on the execution time of program 2. Indeed, a lower value of p implies that it is easier for a maw to be p-absent, then the amount of



(a) Execution time of program 2 (p = 0.3) as a function of k



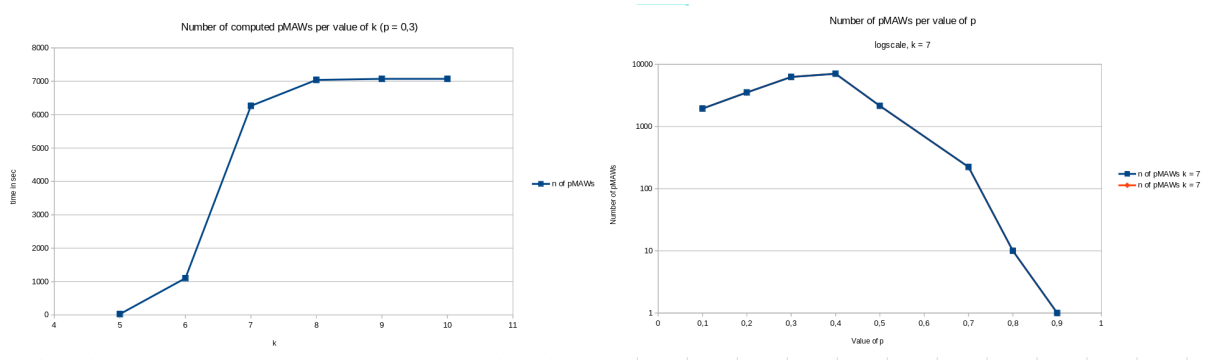
(b) Execution time of program 2 (k = 7) as a function of p

Figure 5: Execution time of program 2

candidates reduces substantially and the execution time too.

Number pMAWs produced per value of k and p

As one can see on Figure 6a, a value of k around 7 enables to provide the vast majority of pMAWs. Figure 6b shows how the value of p has a huge impact on the quantity of pMAWs produced. Indeed, the lowest p is, the easiest it is for a word to be p-absent and thus not to be extended, this implies lower amount of candidates. This low number of candidates limits the number of produced pMAWs for low values of p. When p, exceeds 0,3 it does not limit the number of candidates in the same way but a higher value of p makes it harder for candidates to reach the p-absent condition which then diminish the number of produced pMAWs.



(a) Number of pMAW per value of k, with p = 0,3 (b) Number of pMAW per value of p, with k = 7

Figure 6: Amounts of produced pMAWs

Discussion

From the different figures one can see that choosing a value of k around 7 or 8 seems to enable the different programs to reach reasonable execution time while providing sufficient levels of production of pMAWs. Moreover, a value of p lower than 0.3 seems to artificially limit the amount of produced pMAWs, but the choice of this parameter seems to be made with regard to the biological meaning of such choice.

These conclusions should be followed cautiously, indeed, even though the dataset is a standard in comparable benchmark, we could have used other samples and try to vary the length and variability of reads.

The complexity of Program 2 is quite deceptive, even though it runs fairly rapidly in reality. Indeed, according to our opinion the use of N automata seems to be exagerated and a simplification of the algorithm seems feasible. For instance, we think about merging all these automata in one, which would be used to find substrings and then verifying in a modified version of the trie the substring's absence bitArray. This version was not implemented due to the Lack of time and the relatively satisfying real-world performance of the provided version of the algorithm.

Program 1 could also use some refining, using parallelism, for instance. But, as for program 2, we chose to provide high quality prototyping-grade code to make it easily adaptable, understandable and portable instead of the low quality production code we would have struggled to reach due to our "theoretician" profiles. One could easily derive compiled and optimized version of our code to reach better performance.

Moreover, for the sake of clarity, we chose here not to take ambiguous bases into account. This can be sorted via some sort of preprocessing by the users or via changes to the core algorithms which could be considered as future works.

References

[1] Alfred V. Aho and Margaret J. Corasick. "Efficient string matching: an aid to bibliographic search". In: *Commun. ACM* 18.6 (June 1975), pp. 333–340. ISSN: 0001-0782. DOI: 10.1145/360825.360855. URL: https://doi.org/10.1145/360825.360855.

[2] Pinho AJ et al. "On the computation of minimal absent words". In: *BMC Bioinformatics* (2009).

[3] Rodolphe Barrangou et al. "CRISPR Provides Acquired Resistance against Viruses in Prokaryotes". In: *Science (New York, N.Y.)* (2007). DOI: 10.1126/science.1138140.

[4] Maxime Crochemore et al. "Absent words in a sliding window with applications". In: *Information and Computation* 270 (Sept. 2019), p. 104461. DOI: 10.1016/j.ic.2019.104461.