Minimal Absent Words in a Sliding Window and Applications to On-Line Pattern Matching

Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon Pissis, Yann Ramusat

Minimal absent words in a sliding window & applications to on-line pattern matching

Maxime Crochemore[1,2], Alice Héliou[3], Gregory Kucherov[2], Laurent Mouchard[4], Solon P. Pissis[1], and Yann Ramusat[5]

[1] Department of Informatics, King's College London, London, UK
{maxime.crochemore,solon.pissis}@kcl.ac.uk
[2] CNRS & Université Paris-Est, France gregory.kucherov@univ-mlv.fr
[3] LIX, École Polytechnique, CNRS, INRIA, Université Paris-Saclay, France
alice.heliou@polytechnique.org
[4] University of Rouen, LITIS EA 4108, TIBS, Rouen, France
laurent.mouchard@univ-rouen.fr
[5] DI ENS, ENS, CNRS, PSL Research University & INRIA Paris, France
yann.ramusat@ens.fr

Abstract. An *absent* (or forbidden) word of a word y is a word that does not occur in y. It is then called *minimal* if all its proper factors occur in y. There exist linear-time and linear-space algorithms for computing all minimal absent words of y (Crochemore et al., 1998, Belazzougui et al., 2013, Barton et al., 2014). Minimal absent words are used for data compression (Crochemore et al., 2000, Ota and Morita, 2014) and for alignment-free sequence comparison by utilizing a metric based on minimal absent words (Chairungsee and Crochemore, 2012). They are also used in molecular biology; for instance, three minimal absent words of the human genome were found to play a functional role in a coding region in Ebola virus genomes (Silva et al., 2015). In this article we introduce a new application of minimal absent words for on-line pattern matching. Specifically, we present an algorithm that, given a pattern x and a text y, computes the distance between x and every window of size $|x|$ on y. The running time is $\mathcal{O}(\sigma|y|)$, where σ is the size of the alphabet. Along the way, we show an $\mathcal{O}(\sigma|y|)$-time and $\mathcal{O}(\sigma|x|)$-space algorithm to compute the minimal absent words of every window of size $|x|$ on y, together with some new combinatorial insight on minimal absent words.

1 Introduction

Pattern matching is the problem of finding a *pattern* in a usually much longer *text*. Both pattern and text are words (or strings) drawn over some alphabet. This problem has been studied for a long time and efficient solutions have been proposed (see for example [1, 20, 22, 13] or also [16, 9]). A related problem is the approximate pattern matching problem: it is the same problem but allowing some *errors* in the matching process (see [16, 9, 27]). This problem depends mainly on how errors are interpreted and thus which metric is used for the comparison.

Pattern matching algorithms are classified into on-line and off-line. With *off-line* algorithms the text can be processed before searching; a survey of such algorithms was written by Navarro et al. [26]. A more recent algorithm based on a bidirectionnal index has been proposed by Kucherov et al. [21]. With *on-line* algorithms the text cannot be processed before searching. A famous such algorithm is *bitap*, one of the underlying algorithms of Unix utility *agrep*; it was first invented by Dömölki in 1964 [12] and it underwent several improvements among them the last one was done by Myers [24]. A survey on on-line algorithms for approximate pattern matching was written by Navarro [25] (see also [27]).

In this article we propose a new on-line pattern matching scheme using a metric that is based on minimal absent words. This notion of negative information has first been coined as minimal forbidden words by Béal et al. [5]. A *minimal absent word* of word y is a word absent from y whose all proper factors occur in y. A tight upper bound on the number of minimal absent words of a word y of length n over an alphabet of size σ is known to be $\mathcal{O}(\sigma n)$ [10, 23]. Moreover it was shown that the set of all minimal absent words of y is sufficient to uniquely reconstruct y [10, 14]. The notion has been used in data compression [11, 29] and in molecular biology [17, 19, 34, 32, 8, 2, 18], where authors often focus on the computation of the shortest absent words (sometimes called *unwords*).

Chairungsee and Crochemore introduced the Length Weighted Index (LWI), a metric based on the symmetric difference of minimal absent words sets [7]. The LWI was then applied by Crochemore et al. [8] to devise an $\mathcal{O}(m+n)$-time and $\mathcal{O}(m+n)$-space algorithm for alignment-free comparison of two sequences of length m and n on a constant-sized alphabet. More recently, different such indices have been studied for sequence comparison and phylogeny reconstruction [30]. We base our new pattern matching algorithm on this LWI. To maintain the LWI across the word y for a pattern x, we need to compute the set of minimal absent words in a sliding window of size $m = |x|$ of y. Several linear-time and linear-space algorithms have been proposed to compute the set of minimal absent words [10, 6, 3, 4, 15]. Ota et al. presented an on-line algorithm that requires linear time and linear space [28]. However, to the best of our knowledge, the problem of computing minimal absent words in a sliding window has not been addressed.

Our contributions. Here we present the *first* algorithm to compute minimal absent words in a sliding window. For a window of size m and a word of length n on an alphabet of size σ, our algorithm performs $\mathcal{O}(\sigma n)$ insert and delete operations on the set of minimal absent words. With a careful implementation of the data structures, it requires $\mathcal{O}(\sigma n)$ time overall using $\mathcal{O}(\sigma m)$ space. We apply this algorithm for on-line approximate pattern matching using the LWI for a pattern of length m over every window of size m of the text. This yields the *first* algorithm for the classical on-line exact pattern matching problem that uses some form of negative information (minimal absent words) for the comparison.

Definitions and Notation

Let $y = y[0]y[1]\cdots y[n-1]$ be a *word* of length $n = |y|$ on a finite ordered *alphabet* of size $\sigma = |\Sigma|$. We denote by $y[i\,..\,j] = y[i]\cdots y[j]$ the *factor* of y

whose occurrence *starts* at position i and *ends* at position j on y, and by ε the *empty word*, the word of length 0. The set of all possible words on Σ (including the empty word) is denoted by Σ^*. A *prefix* of y is a factor that starts at position 0 ($y[0\,..\,j]$) and a *suffix* is a factor that ends at position $n-1$ ($y[i\,..\,n-1]$). A factor x of y is *proper* if $x \neq y$.

Let u be a non-empty word. An integer p such that $0 < p \leq |u|$ is called a *period* of u if $u[i] = u[i+p]$, for $i = 0, 1, \ldots, |u| - p - 1$. For every word u and every natural number k, we define the kth *power* of the word u, denoted by u^k, by $u^0 = \varepsilon$ and $u^k = u^{k-1}u$, for $k = 1, 2, \ldots, n$.

Let x be a word of length $m \leq n$. We say that there exists an *occurrence* of x in y when x is a factor of y. Opposingly, we say that the word x is an *absent* word of y if it does not occur in y. We consider absent words of length at least 2 only. An absent word x of length m, $m \geq 2$, of y is *minimal* if and only if all its proper factors occur in y. This is equivalent to saying that a minimal absent word (MAW) of y is of the form aub, $a, b \in \Sigma, u \in \Sigma^*$, such that au and ub are factors of y but aub is not. We can easily see that, if x is a MAW of y, then $2 \leq |x| \leq |y| + 1$. Note that $|x| = |y| + 1$ if and only if $y = a^{|y|}$ for some $a \in \Sigma$.

Example 1. Let $y = \mathsf{ABAACA}$. Its factors of lengths 1 and 2 are A, B, C, AA, AB, AC, BA, and CA. The set of MAWs of y is obtained by combining the aforementioned factors: $\{\mathsf{BB}, \mathsf{BC}, \mathsf{CB}, \mathsf{CC}, \mathsf{AAA}, \mathsf{AAB}, \mathsf{BAB}, \mathsf{BAC}, \mathsf{CAA}, \mathsf{CAB}, \mathsf{CAC}\}$.

Let U and V be two sets. We denote by $U \triangle V$ their symmetric difference, that is, $U \triangle V = (U \setminus V) \cup (V \setminus U)$. We consider the LWI, a distance on Σ^*, for two words x and y on Σ^* [7]. It is based on the set $M(x) \triangle M(y)$, where $M(x)$ is the set of minimal absent words of x, and it is defined by:

$$\mathsf{LWI}(x, y) = \sum_{w \in M(x) \triangle M(y)} \frac{1}{|w|^2}.$$

2 Combinatorial results

In this section we consider a word z of fixed length m on an alphabet Σ of size σ and denote by $M(z)$ its set of MAWs. The word z essentially represents the content of the window on word y used in the algorithm of Section 3. We first discuss changes to be done on the set of MAWs when appending and removing letters on the word of interest. Then we show bounds on the number of changes on the set of MAWs when moving forward the current window by one position.

2.1 Changes when appending one letter to the window

We denote by $M(z)|\alpha$, $\alpha \in \Sigma$, the operation on the set of MAWs when concatenating the letter α to the, possibly empty, word z. The operation creates $M(z\alpha)$ from $M(z)$. We introduce some bounds on the number of insertions/deletions for the on-line computation of the set of MAWs. These results have already been shown in [28] and we briefly present them for completeness.

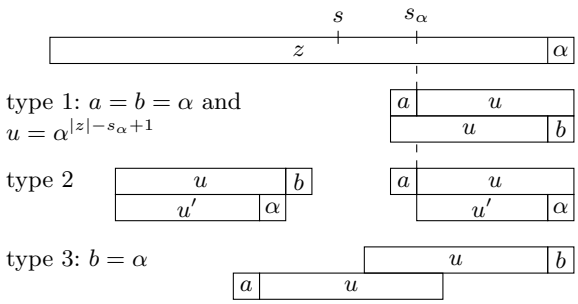

Fig. 1. Illustration of the three different types of MAWs that are added when letter α is appended to z.

We denote by s the starting position of the longest suffix of z that repeats in z; when this suffix is empty we set $s = |z|$. We also denote by s_α the starting position of the longest suffix that occurs in z followed by α; when this suffix is empty we set $s_\alpha = |z|$. Note that we have $s \leq s_\alpha$ because the latter suffix obviously repeats in z. This is illustrated in Figure 1.

The next two lemmas state bounds of the number of insert and delete operations performed by $M(z)|\alpha$.

Lemma 1. *$M(z)|\alpha$ deletes exactly one MAW from $M(z)$, namely, $z[s_\alpha-1\,..\,|z|-1]\alpha$*

Proof. Let $w = aub$, $a, b \in \Sigma$ and $u \in \Sigma^*$, be a MAW to be removed. This means that aub is absent in z but present in $z\alpha$. Thus $b = \alpha$ and au is a suffix of z that does not occur followed by α in z. The word $ub = u\alpha$ is also present in z, so u is a suffix of z that occurs in z followed by α. Then the starting position of the suffix occurrence of u in z is s_α and $w = z[s_\alpha - 1\,..\,|z| - 1]\alpha$. □

To establish an upper bound on the number of MAWs added by the operation $M(z)|\alpha$, we first divide the new MAWs of the form aub, $a, b \in \Sigma$ and $u \in \Sigma^*$, into three types (see also Figure 1):

1. au and ub are absent in z.
2. au is absent in z and ub is present in z.
3. au is present in z and ub is absent in z.

Lemma 2. *There are at most one MAW of type 1, σ MAWs of type 2, and $(s_\alpha - s)(\sigma - 1)$ MAWs of type 3 added by the operation $M(z)|\alpha$.*

Proof. We consider a new MAW $w = aub$, $a, b \in \Sigma$ and $u \in \Sigma^*$, created by the operation. Let w be of type 1, that is, au and ub do not occur in z. Then they are both suffixes of $z\alpha$, and because they have same length, are equal. This implies that u is both a prefix and a suffix of $ub = u\alpha$. Thus the latter has period 1, w is of the form $\alpha^{|w|}$, and $u = \alpha^{|w|-2}$. But then $u\alpha$ is absent in z. Therefore, $\alpha^{|w|-3}$

is the longest repeated suffix of z that occurs followed by α in z. Consequently $|w| = |z| - s_\alpha + 3$.

Let w be of type 2, that is, ub occurs in z and au occurs in $z\alpha$ but not in z. Then au is a suffix of $z\alpha$ and u can be written $u'\alpha$. As ub occurs in z, u' is a suffix of z that occurs in z followed by α. Moreover, since $au = au'\alpha$ does not occur in z, u' is the longest suffix of z that occurs in z followed by α, therefore its starting position as a suffix is s_α. The letter b can be any letter of the alphabet of z that occurs after an occurrence of u in z. Consequently there are at most σ such MAWs.

Let w be of type 3, that is, au occurs in z and ub occurs in $z\alpha$ but not in z. This implies that $b = \alpha$, u is a suffix of z not preceded by a, and au occurs elsewhere in z. Since no occurrence of u in z is followed by α, we have that the starting position k of u as a suffix satisfies $s \leq k < s_\alpha$. Therefore, there are at most $s_\alpha - s$ possible words u and for each of them, there are at most $\sigma - 1$ possibilities for the letter a to obtain a MAW. Consequently, there are at most $(s_\alpha - s)(\sigma - 1)$ such MAWs. □

The previous lemma shows that during one step of the computation of MAWs for a sliding window of size m we may have to handle $\mathcal{O}(\sigma m)$ new MAWs. However, the total number of insertions when computing the set of MAWs for a word y of length n get amortized to $\mathcal{O}(\sigma n)$ in an on-line computation.

Proposition 1 ([28]). *Starting with the empty word, and applying n times the operation $|$ leads to a total number of insertions/deletions of MAWs in $\mathcal{O}(\sigma n)$.*

Proof. The number of MAWs of the whole word of length n is in $\mathcal{O}(\sigma n)$ [10]. As stated by Lemma 1 at most one MAW can be deleted by each application of the operation $|$. Thus the total number of insertions/deletions is still in $\mathcal{O}(\sigma n)$. □

2.2 Changes when removing the first letter of the window

We denote by $M(\alpha z) \to M(z)$, $\alpha \in \Sigma$, the operation on the set of MAWs when deleting the letter α from the word αz. Removing the leftmost letter of the window is a dual question to what is done previously. We now focus on the longest repeated prefix instead of the longest repeated suffix.

Let us denote by p the ending position of the longest repeated prefix of z and by p_α the ending position of the longest prefix of z that occurs in z preceded by α. We set them to 0 when the prefixes are empty. Note that $p_\alpha \leq p$. Similar to Lemma 1, removing a letter from the left creates exactly one MAW.

Lemma 3. *The operation $M(\alpha z) \to M(z)$ creates exactly one MAW, which is $\alpha z[0\,..\,p_\alpha + 1]$.*

Similar to Section 2.1, we distinguish among three types of MAWs to be deleted by the operation:

1. au and ub are absent in z.
2. au is absent in z and ub present in z.
3. ub is absent in z and au present in z.

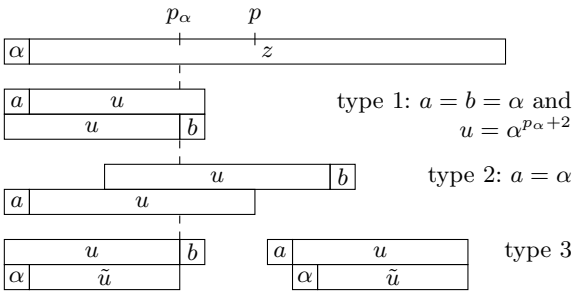

Fig. 2. Illustration of the three different types of MAWs that are deleted when removing α, the letter before z.

We note that types 1, 2, and 3 behave respectively similarly to type 1, 3, and 2 in Section 2.1; see Figure 2 for an illustration. The following result is similar to that stated in Lemma 2.

Lemma 4. *There are at most one MAW of type 1,* $(\sigma - 1)(p - p_\alpha)$ *MAWs of type 2, and* σ *MAWs of type 3 to be deleted by the operation* $M(\alpha z) \to M(z)$.

2.3 Changes when sliding a window over a text

We now focus on our main problem: MAWs in a sliding window. For $m < n$ and for all i, $0 \leq i \leq n - m$, we consider the window $y[i\,..\,i + m - 1]$ and define:

- s_i the starting position of its longest repeated suffix,
- $\tilde{s_i}$ the starting position of its longest suffix that occurs followed by $y[i + m]$,
- ss_i the starting position of its longest suffix that is a power,
- p_i the ending position of its longest repeated prefix,
- $\tilde{p_i}$ the ending position of its longest prefix that occurs preceded by $y[i - 1]$,
- pp_i the ending position of its longest prefix that is a power.

In what follows, we make use of this notation considering the case of a sliding window. The following lemma shows that we cannot output in linear time the set of MAWs in the sliding window at each step of the process.

Lemma 5. *The upper bound of* $\sum_{i=0}^{n-m} |M(y[i\,..\,i + m - 1])|$ *is* $\mathcal{O}(\sigma nm)$ *and this bound is tight.*

Proof. For every factor z of length m of y, $|M(z)|$ is $\mathcal{O}(\sigma m)$. Thus the upper bound of their sum is $\mathcal{O}(\sigma nm)$. Now consider $y = (\mathsf{A}^{m-1}\mathsf{C}^{m-1})^{\frac{n}{2m-2}}$ of length n and its factors of length $2m$. In each factor w of length $2m$, this kind of pattern occurs: $XY^{m-1}X$, with $\{X, Y\} = \{\mathsf{A}, \mathsf{C}\}$. Thus $\{XY^iX | 1 \leq i \leq m-1\} \subseteq M(w)$, so $|M(w)| \geq m - 1$. Consequently the bound is tight. One can generalize this construction of y to obtain a tight bound for larger alphabets (Lemma 1 in [2]). □

However, as shown below, we can bound the number of changes necessary to maintain the set of MAWs for a sliding window. We obtain the following result.

Theorem 1. *The upper bound of* $\sum_{i=0}^{n-m-1} |M(y[i\,..\,i+m-1])\triangle M(y[i+1\,..\,i+m])|$ *is in* $\mathcal{O}(\sigma n)$.

Proof. Let us consider the set $M(y[i\,..\,i+m-1])\triangle M(y[i\,..\,i+m])$ with $0 \leq i < n-m$. From Lemmas 1 and 2 we get

$$|M(y[i\,..\,i+m-1])\triangle M(y[i\,..\,i+m])| \leq (\tilde{s_i} - s_i)(\sigma - 1) + \sigma + 2.$$

Then,

$$\sum_{i=0}^{n-m-1} |M(y[i\,..\,i+m-1])\triangle M(y[i\,..\,i+m])| \leq \sum_{i=0}^{n-m-1} (\tilde{s_i} - s_i)(\sigma - 1) + n\sigma + 2n.$$

We note that $\tilde{s_i} \leq s_{i+1} \leq \tilde{s_i} + 1$ and we have $s_i \leq \tilde{s_i}$ thus

$0 \leq \sum_{i=0}^{n-m-1} (\tilde{s_i} - s_i) = \sum_{i=0}^{n-m-1} \tilde{s_i} - \sum_{i=0}^{n-m-1} s_i$

$0 \leq \sum_{i=0}^{n-m-1} (\tilde{s_i} - s_i) = \tilde{s}_{n-m-1} - s_0 + \sum_{i=0}^{n-m-2} (\tilde{s_i} - s_{i+1}) \leq n$

Then $\sum_{i=0}^{n-m-1} |M(y[i\,..\,i+m-1])\triangle M(y[i\,..\,i+m])| \leq 2n\sigma+n$. Now, we consider the set $M(y[i\,..\,i+m])\triangle M(y[i+1\,..\,i+m])$. From Lemmas 3 and 4 we obtain a similar inequality: $\sum_{i=0}^{n-m-1} |M(y[i\,..\,i+m])\triangle M(y[i+1\,..\,i+m])| \leq 2n\sigma + n$. Thus we obtain the desired bound by the triangle inequality. □

3 Minimal absent words in a sliding window

For a general introduction to suffix trees, see [9]. The *suffix tree* T of a non-empty word w of length n is a compact trie representing all suffixes of w. The nodes of the trie which become nodes of the suffix tree (i.e., branching nodes and leaves) are called *explicit* nodes, while the other nodes are called *implicit*. We use $L(v)$ to denote the path-label of a node v, i.e., the concatenation of the edge labels along the path from the root to v. Node v is a *terminal* node if and only if $L(v) = w[i\,..\,n-1]$, $0 \leq i < n$; here v is also labelled with index i. The *suffix link* of a node v with path-label $L(v) = \alpha s$ is a pointer to the node path-labelled s, where $\alpha \in \Sigma$ is a single letter and s is a word. The suffix link of v exists if v is a non-root internal node of T. Our algorithm relies on Senft's on-line construction algorithm of the suffix tree for a sliding window [31] that is itself based on Ukkonen's on-line construction algorithm of the suffix tree [33].

3.1 An overview of Senft's algorithm

The algorithm of Ukkonen constructs the suffix tree on-line in $\mathcal{O}(n)$ time for a word of length n on a constant-sized alphabet by processing the word from left to right. To adapt it for a sliding window with amortized constant time per one window shift, two additional problems need to be resolved: (i) deleting the leftmost letter of a window; and (ii) maintaining edge labels under window shifts.

Deleting the leftmost letter. Consider the longest repeated prefix of the current window. When the leftmost letter is deleted, all prefixes that are longer than this prefix need to be removed from the tree but the longest repeated prefix and all shorter prefixes will remain in the tree. To remove these prefixes we delete the leaf corresponding to the whole window and its incoming edge as follows:

- If the longest repeated prefix corresponds to an explicit node, this node is the parent of the leaf to be deleted. If this node has only one child remaining, we delete the node and merge the two edges. Otherwise, we do nothing.
- If the longest repeated prefix corresponds to an implicit node, it is equal to the longest repeated suffix. We create a new leaf in the place of the one we have deleted. We label it with the starting position of what was the longest repeated suffix and its incoming edge is labelled accordingly.

Maintaining Edge Labels. Assume by induction that all edge labels are correctly positioned relative to the current window. For the next m shifts of the window, we still maintain the same relative positioning of edge labels. After the m shifts, edge labels are recomputed by a bottom-up traversal of the tree. Since m shifts create at most $2m$ nodes, the amortized time spent on one shift is $O(1)$.

3.2 Our algorithm

Consider a word y of length n on an alphabet Σ of size σ. Our goal is to maintain the set of MAWs for a sliding window of size m. That is, for all successive $i \in [0, \ldots, n-m]$, we want to compute $M_m(i) = M(y[i \,..\, i+m-1])$.

For a word z, by $\Sigma(z)$ we denote the alphabet of z and by $V(z)$ the set of explicit nodes in the suffix tree of z. Consider a mapping $f : M(z) \to \Sigma(z) \times V(z)$ defined by $f(aub) = (a, v_{ub})$, where $a \in \Sigma$ and v_{ub} is either the explicit node corresponding to the factor ub or the immediate explicit descendant node if this node is implicit.

Lemma 6. *Mapping f is an injection.*

Proof. Let $w, w' \in M(z)$, $w \neq w'$, $w = aub$ and $w' = a'u'b'$, with $a, b, a', b' \in \Sigma(z)$ and $u, u' \in \Sigma(z)^*$.

Suppose that $f(w) = f(w')$, then $a = a'$ and $v_{ub} = v_{u'b'}$. Thus ub and $u'b'$ are distinct prefixes of the factor corresponding to v_{ub}, consequently one is prefix of the other, without loss of generality ub is prefix of $u'b'$. Then aub is a prefix of $au'b'$, this is impossible as they are both MAWs of z. Thus two distinct elements of $M(z)$ cannot share the same image by f, so f is an injection. □

Lemma 6 allows us to represent all MAWs by storing a set of letters in each explicit node of the tree. We will call this set the *maw*-set. Moreover, a letter a in the *maw*-set will be *tagged* if and only if u corresponds to an implicit node in the tree. Observe that a can become tagged only when u is a repeated suffix of y. This is because factors au and ub define distinct occurrences of u, and the occurrence of au must be a suffix, otherwise u would be followed by two distinct letters and would then be an explicit node. Besides *maw*-sets, we will also need to store at each explicit node another set of letters: the set of all letters preceding the occurrences of the factor corresponding to the node.

By induction, assume we are at position i, the suffix tree $T_m(i)$ for $y[i\,..\,i+m-1]$ is built and the set of MAWs $M_m(i)$ has been computed. We now explain how to update $T_m(i)$ and $M_m(i)$ to obtain $T_m(i+1)$ and $M_m(i+1)$. The tree is updated based on Senft's algorithm, by first adding a letter to the right of the current window and then deleting the leftmost letter. The set of MAWs is updated using Lemmas 1, 2 and 3, 4 respectively. The algorithm will maintain positions $s_i, p_i, \tilde{s_i}, \tilde{p_i}, ss_i, pp_i$ as defined in Section 2.3. We store the leaf nodes in a list so that the last created leaf and the "oldest" leaf currently in the tree can be accessed in constant time.

Adding a letter to the right. We follow Ukkonen's algorithm for updating the suffix tree. Recall that Ukkonen's algorithm proceeds by updating the *active node* in the tree. At the beginning of each iteration, the active node corresponds to the longest repeated suffix, i.e. to factor $y[s_i\,..\,i+m-1]$. The node corresponding to the longest repeated prefix is called the *head node*.

The algorithm starts from the active node and updates it following the suffix links until reaching a node with an outgoing edge starting with $y[i+m]$ – this node corresponds to the suffix starting at $\tilde{s_i}$. At the same time, we compute MAWs of type 3 that are created. For each $s_i \leq j < \tilde{s_i}$, we perform the following.

- If the active node is implicit we make it explicit. We set its set of preceding letters equal to its child's set. We move the untagged letters of the *maw*-set of its child to the *maw*-set of the active node. We untag the tagged letters of the *maw*-set of its child. If the last node created at this window shift does not have a suffix link, we add a suffix link from this node to the active node. We add the letter corresponding to this suffix link to the set of preceding letters of the active node.
- We create a leaf labelled j, with $y[j-1]$ in its set of preceding letters. We create an edge from the active node to this leaf with the label $y[i+m]$.
- For each letter $a \neq y[j-1]$ in the set of preceding letters of the active node, $ay[s_i+j\,..\,i+m] \in M_{m+1}(i) \backslash M_m(i)$ (type 3 in Lemma 2), therefore we add a in the *maw*-set of the leaf.

The current active node corresponds to the factor $y[\tilde{s_i}\,..\,i+m-1]$. According to Lemma 1, there is exactly one MAW to be deleted which is $y[\tilde{s_i}-1\,..\,i+m]$. This MAW is stored in the child of the active node by following the edge starting with $y[i+m]$; we remove $y[\tilde{s_i}-1]$ (tagged or not) from its *maw*-set.

Then we update the active node by following the edge starting with $y[i+m]$; now it corresponds to the factor $y[\tilde{s}_i \,..\, i+m]$. If the head node was also corresponding to the factor $y[\tilde{s_i} \,..\, i+m-1]$, we move it down with the active node; we have $\tilde{p}_{i+1} = p_i + 1$, otherwise we have $\tilde{p}_{i+1} = p_i$. If the active node is explicit, we update its set of preceding letters by adding $y[\tilde{s}_i - 1]$.

Then, for each letter b occurring after an occurrence of $y[\tilde{s}_i \,..\, i+m]$ in $y[i \,..\, i+m-1]$, $y[\tilde{s}_i - 1 \,..\, i+m]b \in M_{m+1}(i) \backslash M_m(i)$ (type 2 in Lemma 2). These MAWs are stored in their corresponding child of the active node. If the active node is implicit, there is only one of them and we tag the letter.

By Lemma 2, if $ss_i = \tilde{s_i} - 1$, then $y[i+m]y[\tilde{s_i} - 1 \,..\, i+m]$ is the new MAW of type 1. We store it in the *maw*-set of the child of the active node by following the edge starting with $y[i+m]$.

Deleting the leftmost letter. We note that the longest repeated prefix of $y[i \,..\, i+m]$ is $y[i \,..\, \tilde{p}_{i+1}]$, and its longest repeated suffix is $y[\tilde{s}_i \,..\, i+m]$. At the beginning of this step they correspond respectively to the head node and the active node. Consider the parent of the oldest leaf of the tree, similarly as in Senft's algorithm two cases are distinguished.

- If the head node is an explicit node, then it is the parent of the oldest leaf. We remove the leaf and its incoming edge. If the head node has only one remaining child, we delete the node and merge the two edges; the *maw*-set associated to the node is added to the leaf.
- Otherwise, the head node is on the edge leading to the oldest leaf. We replace the leaf with a new one labelled by $\tilde{s}_i$, with $y[\tilde{s}_i - 1]$ as the only preceding letter, and the edge is relabelled by $y[\tilde{s}_i - 1]$. We add $y[\tilde{s}_i - 1]$ to the set of preceding letters of the parent of the leaf.

The MAWs associated to the leaf we have deleted were those of type 3 (Lemma 4). We now update the tree and compute the other MAWs to remove and add.

We visit the oldest leaf in the tree and empty its set of preceding letters. Then we move up in the tree following back the edges until we have covered $\tilde{p}_{i+1} - i$ letters. We move the head node to this node: it corresponds to the factor $y[i+1 \,..\, \tilde{p}_{i+1}]$. If the active node was equal to the head node, we move the active node to this node; we have $s_{i+1} = \tilde{s}_i - 1$, otherwise we have $s_{i+1} = \tilde{s}_i$. Each of the explicit nodes visited on the path from the oldest leaf to the head node corresponds to a factor $y[i+1 \,..\, j]$, with $p_{i+1} \geq j > \tilde{p}_{i+1}$. For each of them, we remove $y[i]$ from their set of preceding letters. For each of their children, we remove letter $y[i]$ (tagged or not) from their *maw*-set (type 2 Lemma 4).

There is at most one MAW of type 1 that has to be deleted (Lemma 4). It exists if and only if $y[i] = y[i+1]$ and $pp_{i+1} = \tilde{p}_{i+1} + 1$, in which case we remove it from the *maw*-set of the child of the head node by following the edge starting with $y[i]$. According to Lemma 3, removing the leftmost letter creates one MAW, which is $y[i]y[i+1 \,..\, \tilde{p}_{i+1} + 1]$, thus we add $y[i]$ to the *maw*-set of the child of the head node by following the edge starting with $y[\tilde{p}_{i+1} + 1]$. If the head node is implicit and thus equal to the active node we tag the letter $y[i]$.

Finally if the head node is above the parent of the oldest leaf of the tree, we move it down to this node. If the active node is implicit and on the edge leading to the oldest leaf of tree we set the head node equal to the active node.

Complexity. The algorithm extends Senft's algorithm for the construction of the suffix tree in a sliding window. For both addition and deletion of a letter, the number of operations is $\mathcal{O}(\sigma(\tilde{s}_i - s_i))$ and $\mathcal{O}(\sigma(p_{i+1} - \tilde{p}_{i+1}))$. Similar to the proof of Theorem 1, we obtain that the total number of operations is $\mathcal{O}(\sigma n)$. We use $\mathcal{O}(\sigma m)$ space to store the suffix tree for the factor inside the window. The σ factor is to store an array of size σ at each explicit node for constant-time child queries. We also use up to $4m$ arrays of size σ each to store the two sets of letters – the suffix tree has no more than $2m$ explicit nodes. We also store the word itself over two windows. Thus the total space complexity is bounded by $\mathcal{O}(\sigma m)$. We thus obtain the following result.

Theorem 2. *Given a word of length n on an alphabet of size σ, our algorithm computes the set of minimal absent words in a sliding window of size m in $\mathcal{O}(\sigma n)$ time and $\mathcal{O}(\sigma m)$ space.*

4 Applications to on-line pattern matching

As a consequence of Theorem 2 we obtain the following result.

Theorem 3. *Given a word x of length m on an alphabet Σ of size σ, one can find on-line all occurrences of x in a word y of length $n \geq m$ on alphabet Σ in $\mathcal{O}(\sigma n)$ time and $\mathcal{O}(\sigma m)$ space. Within the same complexities, one can also compute on-line $\mathsf{LWI}(x, y[i\,..\,i+m-1])$, for all $0 \leq i \leq n-m$.*

Proof. As a pre-processing step, we build the suffix tree of x and compute the MAWs of x. At the same time, by Lemma 6, we represent all MAWs of x by storing a set of letters in each explicit node of the tree. This can be done in $\mathcal{O}(\sigma m)$ time and space [10]. We then apply Theorem 2 to build the suffix tree for a sliding window of size m over y on top of the suffix tree of x. This way when a MAW is created or deleted we can update LWI in $\mathcal{O}(1)$ time as we can check if it is a MAW of x or not. For the first part, note that two words x and z are equal if and only if $\mathsf{LWI}(x, z) = 0$ [10, 14]. We thus obtain the result. □

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18:333–340, 1975.
2. Y. Almirantis, P. Charalampopoulos, J. Gao, C. S. Iliopoulos, M. Mohamed, S. P. Pissis, and D. Polychronopoulos. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology*, 12(1):5:1–5:12, 2017.

3. C. Barton, A. Heliou, L. Mouchard, and S. P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinformatics*, 15:11, 2014.
4. C. Barton, A. Heliou, L. Mouchard, and S. P. Pissis. Parallelising the computation of minimal absent words. In *PPAM, Part II*, volume 9574 of *LNCS*, pages 243–253. Springer, 2015.
5. M. Béal, F. Mignosi, and A. Restivo. Minimal forbidden words and symbolic dynamics. In *STACS*, volume 1046 of *LNCS*, pages 555–566. Springer, 1996.
6. D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Versatile succinct representations of the bidirectional burrows-wheeler transform. In *ESA*, volume 8125 of *LNCS*, pages 133–144. Springer, 2013.
7. S. Chairungsee and M. Crochemore. Using minimal absent words to build phylogeny. *Theoretical Computer Science*, 450:109–116, 2012.
8. M. Crochemore, G. Fici, R. Mercas, and S. P. Pissis. Linear-time sequence comparison using minimal absent words. In *LATIN*, volume 9644 of *LNCS*, pages 334–346. Springer, 2016.
9. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
10. M. Crochemore, F. Mignosi, and A. Restivo. Automata and forbidden words. *Information Processing Letters*, 67(3):111–117, 1998.
11. M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Data compression using antidictonaries. *Proceedings of the IEEE*, 88(11):1756–1768, 2000.
12. B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics 3*, pages 29–46, 1964.
13. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS*, pages 390–398. IEEE Computer Society, 2000.
14. G. Fici. *Minimal Forbidden Words and Applications*. Thèse, Université de Marne la Vallée, 2006.
15. Y. Fujishige, Y. Tsujimaru, S. Inenaga, H. Bannai, and M. Takeda. Computing dawgs and minimal absent words in linear time for integer alphabets. In *MFCS*, volume 58 of *LIPIcs*, pages 38:1–38:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
16. D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.
17. G. Hampikian and T. L. Andersen. Absent sequences: Nullomers and primes. In *PSB*, pages 355–366. World Scientific, 2007.
18. A. Heliou, S. P. Pissis, and S. J. Puglisi. emMAW: Computing minimal absent words in external memory. *Bioinformatics*, 2017.
19. J. Herold, S. Kurtz, and R. Giegerich. Efficient computation of absent words in genomic sequences. *BMC Bioinformatics*, 9, 2008.
20. D. E. Knuth, Jr, and V. R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
21. G. Kucherov, K. Salikhov, and D. Tsur. Approximate string matching using a bidirectional index. *Theor. Comput. Sci.*, 638:145–158, 2016.
22. G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM J. Comput.*, 27–2:557–582, 1998.
23. F. Mignosi, A. Restivo, and M. Sciortino. Words and forbidden factors. *Theoretical Computer Science*, 273(1-2):99–117, 2002.
24. G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999.
25. G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

26. G. Navarro, R. A. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.
27. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings : Practical Online Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2008.
28. T. Ota, H. Fukae, and H. Morita. Dynamic construction of an antidictionary with linear complexity. *Theor. Comput. Sci.*, 526:108–119, 2014.
29. T. Ota and H. Morita. On a universal antidictionary coding for stationary ergodic sources with finite alphabet. In *ISITA*, pages 294–298. IEEE, 2014.
30. M. S. Rahman, A. Alatabbi, T. Athar, M. Crochemore, and M. S. Rahman. Absent words and the (dis)similarity analysis of DNA sequences: an experimental study. *BMC Bioinformatics Notes*, 9(1):1–8, 2016.
31. M. Senft. Suffix tree for a sliding window: An overview. In *WDS*, pages 41–46. Matfyzpress, 2005.
32. R. M. Silva, D. Pratas, L. Castro, A. J. Pinho, and P. J. S. G. Ferreira. Three minimal sequences found in Ebola virus genomes and absent from human DNA. *Bioinformatics*, 31(15):2421–2425, 2015.
33. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
34. Z. Wu, T. Jiang, and W. Su. Efficient computation of shortest absent words in a genomic sequence. *Inf. Process. Lett.*, 110(14-15):596–601, 2010.