Constructing Antidictionaries of Long Texts in Output-Sensitive Space
Lorraine a K Ayad, Golnaz Badkobeh, Gabriele Fici, Alice Héliou, Solon P Pissis

To cite this version:
Lorraine a K Ayad, Golnaz Badkobeh, Gabriele Fici, Alice Héliou, Solon P Pissis. Constructing Antidictionaries of Long Texts in Output-Sensitive Space. Theory of Computing Systems, 2021, 65 (5), pp.777-797. 10.1007/s00224-020-10018-5. hal-03498331

HAL Id: hal-03498331
https://inria.hal.science/hal-03498331
Submitted on 21 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Constructing Antidictionaries in Output-Sensitive Space

Lorraine A.K. Ayad\(^1\), Golkz Badkobeh\(^2\), Gabriele Fici\(^3\), Alice Héliou\(^4\), and Solon P. Pissis\(^5\)

\(^1\)Department of Informatics, King’s College London, London, UK, 
lorraine.ayad@kcl.ac.uk
\(^2\)Department of Computing, Goldsmiths University of London, London, UK, 
g.badkobeh@gold.ac.uk
\(^3\)Dipartimento di Matematica e Informatica, Università di Palermo, Palermo, Italy, 
gabriele.fici@unipa.it
\(^4\)Independent Researcher, alice.heliou@gmail.com
\(^5\)CWI, Amsterdam, The Netherlands, solon.pissis@cwi.nl

Abstract

A word \(x\) that is absent from a word \(y\) is called \textit{minimal} if all its proper factors occur in \(y\). Given a collection of \(k\) words \(y_1, y_2, \ldots, y_k\) over an alphabet \(\Sigma\), we are asked to compute the set \(M^\ell_{y_1 \ldots y_k}\) of minimal absent words of length at most \(\ell\) of word \(y = y_1 \# y_2 \# \ldots \# y_k\), \# \notin \Sigma.\) In data compression, this corresponds to computing the antidictionary of \(k\) documents. In bioinformatics, it corresponds to computing words that are absent from a genome of \(k\) chromosomes. This computation generally requires \(\Omega(n)\) space for \(n = |y|\) using any of the plenty available \(O(n)\)-time algorithms. This is because an \(\Omega(n)\)-sized text index is constructed over \(y\) which can be impractical for large \(n\). We do the identical computation incrementally using output-sensitive space. This goal is reasonable when \(|\{M^\ell_{y_1 \ldots y_k}\}| = o(n)\), for all \(N \in [1, k]\). For instance, in the human genome, \(n \approx 3 \times 10^9\) but \(|\{M^\ell_{y_1 \ldots y_k}\}| \approx 10^6\). We consider a constant-sized alphabet for stating our results. We show that all \(M^\ell_{y_1, \ldots, y_k}\) can be computed in \(O(kn + \sum_{1=1}^{k} |\{M^\ell_{y_1 \ldots y_N}\}|)\) total time using \(O(\text{MAXIN} + \text{MAXOUT})\) space, where \(\text{MAXIN}\) is the length of the longest word in \(\{y_1, \ldots, y_k\}\) and \(\text{MAXOUT} = \max\{|\{M^\ell_{y_1 \ldots y_N}\}| : N \in [1, k]\}\). Proof-of-concept experimental results are also provided confirming our theoretical findings and justifying our contribution.

1 Introduction

The word \(x\) is an \textit{absent word} of the word \(y\) if it does not occur in \(y\). The absent word \(x\) of \(y\) is called \textit{minimal} if and only if all its proper factors occur in \(y\). The set of all minimal absent words for a word \(y\) is denoted by \(M_y\). The set of all minimal absent words of length at most \(\ell\) of a word \(y\) is denoted by \(M^\ell_y\). For example, if \(y = abaab\), then \(M_y = \{aaa, aaba, bab, bb\}\) and \(M^3_y = \{aaa, bab, bb\}\). The upper bound on the number of minimal absent words is \(O(\sigma n)\) [10], where \(\sigma\) is the size of the alphabet and \(n\) is the length of \(y\), and this is tight for integer alphabets [6]; in fact, for large alphabets, such as when \(\sigma \geq \sqrt{n}\), this bound is also tight even for minimal absent words having the same length [1].

State-of-the-art algorithms compute all minimal absent words of \(y\) in \(O(\sigma n)\) time [2, 10] or in \(O(n + |M_y|)\) time [7, 16] for integer alphabets. There also exist space-efficient data structures based on the Burrows-Wheeler transform of \(y\) that can be applied for this computation [4, 5]. In
many real-world applications of minimal absent words, such as in data compression [11, 12, 14, 22], in sequence comparison [6, 7], in on-line pattern matching [9], or in identifying pathogen-specific signatures [23], only a subset of minimal absent words may be considered, and, in particular, the minimal absent words of length (at most) \( \ell \). Since, in the worst case, the number of minimal absent words of \( y \) is \( \Theta(n) \), \( \Omega(n) \) space is required to represent them explicitly. In [7], the authors presented an \( O(n) \)-sized data structure for outputting minimal absent words of a specific length in optimal time for integer alphabets.

The problem with existing algorithms for computing minimal absent words is that they make use of \( \Omega(n) \) space; and the same amount is required even if one is merely interested in the minimal absent words of length at most \( \ell \). This is because all of these algorithms construct global data structures, such as the suffix array [2]. In theory, this problem can be addressed by using the external memory algorithm for computing minimal absent words presented in [19]. The I/O-optimal version of this algorithm, however, requires a lot of external memory to build the global data structures for the input [20]. One could also use the algorithm of [15] that computes \( M_y^\ell \) in \( O(n + |M_y^\ell|) \) time using \( O(\min\{n, \ell z\}) \) space, where \( z \) is the size of the LZ77 factorisation of \( y \). This algorithm also requires constructing the truncated DAWG, a type of global data structure which could take space \( \Omega(n) \). Thus, in this paper, we investigate whether \( M_y^\ell \) can be computed efficiently in output-sensitive space.

As \( y \) can be “decomposed” into a collection of \( k \) words—with a suitable overlap of length \( \ell \) so as not to lose information—we consider the following, general, computational problem.

**Problem** Given \( k \) words \( y_1, y_2, \ldots, y_k \) over an alphabet \( \Sigma \) and an integer \( \ell > 0 \), compute the set \( M_{y_1\#\ldots\#y_k}^\ell \) of minimal absent words of length at most \( \ell \) of \( y = y_1\#y_2\#\ldots\#y_k \), \( \# \notin \Sigma \).

In data compression, this scenario corresponds to computing the antidictionary of \( k \) documents [11, 12]. In bioinformatics, it corresponds to computing words that are absent from a genome of \( k \) chromosomes. As discussed above, this computation generally requires \( \Omega(n) \) space for \( n = |y| \). We do the identical computation incrementally using output-sensitive space. This goal is reasonable when \( |M_{y_1\#\ldots\#y_N}^\ell| = o(n) \), for all \( N \in [1, k] \). In the human genome, \( n \approx 3 \times 10^9 \) but \( |M_{y_1\#\ldots\#y_k}^{12}| \approx 10^6 \), where \( k \) is the total number of chromosomes.

**Our Results** Antidictionary-based compressors work on \( \Sigma = \{0, 1\} \) and in bioinformatics we have \( \Sigma = \{A, C, G, T\} \); we thus consider a constant-sized alphabet for stating our results. We show that all \( M_{y_1}^\ell, \ldots, M_{y_k}^\ell \) can be computed in \( O(kn + \sum_{N=1}^k |M_{y_N}^\ell|) \) total time using \( O(\text{MAXIN} + \text{MAXOUT}) \) space, where \( \text{MAXIN} \) is the length of the longest word in \( \{y_1, \ldots, y_k\} \) and \( \text{MAXOUT} = \max\{|M_{y_N}^\ell| : N \in [1, k]\} \). Proof-of-concept experimental results are provided confirming our theoretical findings and justifying our contribution.

## 2 Preliminaries

We generally follow [8]. An **alphabet** \( \Sigma \) is a finite ordered non-empty set of elements called **letters**. A **word** is a sequence of elements of \( \Sigma \). The set of all words over \( \Sigma \) of length at most \( \ell \) is denoted by \( \Sigma^{\leq \ell} \). We fix a constant-sized alphabet \( \Sigma \), i.e., \( |\Sigma| = \Theta(1) \). Given a word \( y = uxv \) over \( \Sigma \), we say that \( u \) is a **prefix** of \( y \), \( x \) is a **factor** (or subword) of \( y \), and \( v \) is a **suffix** of \( y \). We also say that \( y \) is a **superword** of \( x \). A factor \( x \) of \( y \) is called **proper** if \( x \neq y \).
Given a word $y$ over $\Sigma$, the set of minimal absent words (MAWs) of $y$ is defined as

$$
M_y = \{aub \mid a, b \in \Sigma, au \text{ and } ub \text{ are factors of } y \text{ but } aub \text{ is not} \}
$$

$$
\cup \{ c \in \Sigma \mid c \text{ does not occur in } y \}.
$$

For instance, over $\Sigma = \{a, b, c\}$, for $y = ab$ we have $M_y = \{aa, bb, ba, c\}$. MAWs of length 1 for $y$ can be found in $O(|y| + |\Sigma|) = O(|y|)$ time using $O(|\Sigma|) = O(1)$ working space, and so, in what follows, we focus on the computation of MAWs of length at least 2.

The suffix tree $T(y)$ of a non-empty word $y$ of length $n$ is the compact trie representing all suffixes of $y$ [8]. The branching nodes of the trie as well as the terminal nodes, that correspond to non-empty suffixes of $y$, become explicit nodes of the suffix tree, while the other nodes are implicit. We let $\mathcal{L}(v)$ denote the path-label from the root node to node $v$. We say that node $v$ is path-labeled $\mathcal{L}(v)$; i.e., the concatenation of the edge labels along the path from the root node to $v$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the word-depth of node $v$. A node $v$ such that the path-label $\mathcal{L}(v) = y[i \ldots n - 1]$, for some $0 \leq i \leq n - 1$, is terminal and is also labeled with index $i$. Each factor of $y$ is uniquely represented by either an explicit or an implicit node of $T(y)$ called its locus. The suffix-link of a node $v$ with path-label $\mathcal{L}(v) = aw$ is a pointer to the node path-labeled $w$, where $a \in \Sigma$ is a single letter and $w$ is a word. The suffix-link of $v$ exists by construction if $v$ is a non-root branching node of $T(y)$. The matching statistics of a word $x[0 \ldots |x| - 1]$ with respect to word $y$ is an array $\text{MS}_x[0 \ldots |x| - 1]$, where $\text{MS}_x[i]$ is a pair $(f_i, p_i)$ such that (i) $x[i \ldots i + f_i - 1]$ is the longest prefix that is a factor of $y$; and (ii) $y[p_i \ldots p_i + f_i - 1] = x[i \ldots i + f_i - 1]$ [18]. $T(y)$ is constructible in time $O(n)$, and, given $T(y)$, we can compute $\text{MS}_x$ in time $O(|x|)$ [18].

## 3 Combinatorial Properties

For convenience, we consider the following setting. Let $y_1, y_2$ be words over the alphabet $\Sigma$ and let $y_3 = y_1 \# y_2$, with $\# \notin \Sigma$. Let $\ell$ be a positive integer and set $M_{y_1}^\ell = M_{y_1} \cap \Sigma^{\leq \ell}$ and $M_{y_2}^\ell = M_{y_2} \cap \Sigma^{\leq \ell}$. We want to construct $M_{y_3}^\ell = M_{y_3} \cap \Sigma^{\leq \ell}$. Let $x \in M_{y_3}^\ell$. We have two cases:

**Case 1**: $x \in M_{y_1}^\ell \cup M_{y_2}^\ell$.

**Case 2**: $x \notin M_{y_1}^\ell \cup M_{y_2}^\ell$.

The following auxiliary fact follows directly from the minimality property.

**Fact 1.** Word $x$ is absent from word $y$ if and only if $x$ is a superword of a MAW of $y$.

For Case 1, we prove the following lemma.

**Lemma 1** (Case 1). A word $x \in M_{y_1}^\ell$ (resp. $x \in M_{y_2}^\ell$) belongs to $M_{y_3}^\ell$ if and only if $x$ is a superword of a word in $M_{y_2}^\ell$ (resp. in $M_{y_1}^\ell$).

**Proof.** Let $x \in M_{y_1}^\ell$ (the case $x \in M_{y_2}^\ell$ is symmetric). Suppose first that $x$ is a superword of a word in $M_{y_2}^\ell$, that is, there exists $v \in M_{y_2}^\ell$ such that $v$ is a factor of $x$. If $v = x$, then $x \in M_{y_1}^\ell \cap M_{y_2}^\ell$ and therefore, using the definition of MAW, $x \in M_{y_3}^\ell$. If $v$ is a proper factor of $x$, then $x$ is an absent word of $y_2$ and again, by definition of MAW, $x \in M_{y_3}^\ell$.

Suppose now that $x$ is not a superword of any word in $M_{y_2}^\ell$. Then $x$ is not absent in $y_2$ by Fact 1, and hence in $y_3$, thus $x$ cannot belong to $M_{y_1}^\ell$. 


It should be clear that the statement of Lemma 1 implies, in particular, that all words in $M_{y_1}^\ell \cap M_{y_2}^\ell$ belong to $M_{y_3}^\ell$. Furthermore, Lemma 1 motivates us to introduce the reduced set of MAWs of $y_1$ with respect to $y_2$ as the set $R_{y_1}^\ell$ obtained from $M_{y_1}^\ell$ after removing those words that are superwords of words in $M_{y_2}^\ell$. The set $R_{y_2}^\ell$ is defined analogously.

Example 1. Let $y_1 = abaaab$, $y_2 = bbabaab$ and $\ell = 5$. We have $M_{y_1}^\ell = \{ bbb, baaa, baab, aba, bab, abb, aaba \}$ and $M_{y_2}^\ell = \{ bbb, aaaa, baab, aba, bab, abb \}$. The word bab is contained in $M_{y_1}^\ell \cap M_{y_2}^\ell$ so it belongs to $M_{y_3}^\ell$. The word aaba $\in M_{y_1}^\ell$ is a superword of aba $\in M_{y_2}^\ell$ hence aaba $\in M_{y_3}^\ell$. On the other hand, the words bbb, aaaa and abb are superwords of words in $M_{y_1}^\ell$, hence they belong to $M_{y_3}^\ell$. The remaining MAWs are not superwords of MAWs of the other word. The reduced sets are therefore $R_{y_1}^\ell = \{ bbb, aaaa \}$ and $R_{y_2}^\ell = \{ baab, aba \}$. In conclusion, we have for Case 1 that $M_{y_3}^\ell \cap (M_{y_1}^\ell \cup M_{y_2}^\ell) = \{ aaaa, bab, aaba, abb, bbb \}$.

We now investigate the set $M_{y_3}^\ell \setminus (M_{y_1}^\ell \cup M_{y_2}^\ell)$ (Case 2).

Fact 2. Let $x =aub$, $a,b \in \Sigma$, be such that $x \in M_{y_1}^\ell$ and $x \notin M_{y_1}^\ell \cup M_{y_2}^\ell$. Then $au$ occurs in $y_1$ but not in $y_2$ and $ub$ occurs in $y_2$ but not in $y_1$, or vice versa.

The rationale for generating the reduced sets should become clear with the next lemma.

Lemma 2 (Case 2). Let $x \in M_{y_1}^\ell \setminus (M_{y_1}^\ell \cup M_{y_2}^\ell)$. Then $x$ has a prefix $x_i$ in $R_{y_1}^\ell$ and a suffix $x_j$ in $R_{y_2}^\ell$, for $i,j$ such that $\{i,j\} = \{1,2\}$.

Proof. Let $x =aub$, $a,b \in \Sigma$, be a word in $M_{y_1}^\ell \setminus (M_{y_1}^\ell \cup M_{y_2}^\ell)$. By Fact 2, $au$ occurs in $y_1$ but not in $y_2$ and $ub$ occurs in $y_2$ but not in $y_1$, or vice versa. Let us assume the first case holds (the other case is symmetric). Since $au$ does not occur in $y_2$, there is a MAW $x_2 \in M_{y_2}^\ell$ that is a factor of $au$. Since $ub$ occurs in $y_2$, $x_2$ is not a factor of $ub$. Consequently, $x_2$ is a prefix of $au$.

Analogously, there is an $x_1 \in M_{y_1}^\ell$ that is a suffix of $ub$. Furthermore, $x_1$ and $x_2$ cannot be factors one of another. Inspect Figure 1 in this regard.

Example 2. Let $y_1 = abaaab$, $y_2 = bbabaab$ and $\ell = 5$. Consider $x = abaaa \in M_{y_3}^\ell \setminus (M_{y_1}^\ell \cup M_{y_2}^\ell)$ (Case 2 MAW). We have that abaa occurs in $y_1$ but not in $y_2$ and baaa occurs in $y_2$ but not in $y_1$. Since abaa does not occur in $y_2$, there is a MAW $x_2 \in R_{y_2}^\ell$ that is a factor of abaa. Since baaa occurs in $y_2$, $x_2$ is not a factor of baaa. So $x_2$ is a prefix of abaa and this is aba. Analogously, there is MAW $x_1 \in R_{y_1}^\ell$ that is a suffix of abaaa and this is aaa.

As a consequence of Lemma 2, in order to construct the set $M_{y_3}^\ell \setminus (M_{y_1}^\ell \cup M_{y_2}^\ell)$, we should consider all pairs $(x_i, x_j)$ with $x_i$ in $R_{y_1}^\ell$ and $x_j$ in $R_{y_2}^\ell$, $\{i,j\} = \{1,2\}$. In order to construct the final set
$M^\ell_{y_1\ldots y_N}$, we use incrementally Lemmas 1 and 2. We summarise the whole approach in the following general theorem, which forms the theoretical basis of our technique.

**Theorem 1.** Let $N > 1$, and let $x \in M^\ell_{y_1\ldots y_N}$. Then, either $x \in M^\ell_{y_1\ldots y_{N-1}} \cup M^\ell_{y_N}$ (Case 1 MAWs) or, otherwise, $x \in M^\ell_{y_1\ldots y_N} \setminus (M^\ell_{y_1} \cup M^\ell_{y_N})$ for some $i$. Moreover, in this latter case, $x$ has a prefix in $R^\ell_{y_1\ldots y_{N-1}}$ and a suffix in $R^\ell_{y_N}$, or the converse, i.e., $x$ has a prefix in $R^\ell_{y_N}$ and a suffix in $R^\ell_{y_1\ldots y_{N-1}}$ (Case 2 MAWs).

**Proof.** Let $x \in M^\ell_{y_1\ldots y_N}$ and $x \notin M^\ell_{y_1\ldots y_{N-1}} \cup M^\ell_{y_N}$. Then, $x \notin M^\ell_{y_1\ldots y_{N-1}}$ and $x \notin M^\ell_{y_N}$. Let $x$ be a word of length $m$. By the definition of MAW, $x[0 \ldots m - 2]$ and $x[1 \ldots m - 1]$ must both be factors of $y_1\ldots y_N$. However, both cannot be factors of $y_1\ldots y_{N-1}$ and both cannot be factors of $y_N$. Therefore, we have one of the two cases:

**Case 1 :** $x[0 \ldots m - 2]$ is factor of $y_1\ldots y_{N-1}$ but not of $y_N$ and $x[1 \ldots m - 1]$ is a factor of $y_N$ but not of $y_1\ldots y_{N-1}$.

**Case 2 :** $x[0 \ldots m - 2]$ is factor of $y_N$ but not of $y_1\ldots y_{N-1}$ and $x[1 \ldots m - 1]$ is a factor of $y_1\ldots y_{N-1}$ but not of $y_N$.

These two cases are symmetric, thus only proof of Case 1 will be presented here. If $x[0]$ does not occur in $y_N$ then $x[0] \in R^\ell_{y_N}$. Otherwise, let $x[0 \ldots t]$ be the longest prefix of $x[0 \ldots m - 2]$ that is a factor of $y_N$.

Because $0 \leq t < m - 1$ then $x[1 \ldots t + 1]$ is a factor of $y_N$. Therefore, $x[0 \ldots t + 1] \in M^\ell_{y_N}$. In addition, all factors of $x[0 \ldots t + 1]$ occur in $y_1\ldots y_{N-1}$, so $x[0 \ldots t + 1] \in R^\ell_{y_{N-1}}$.

Now, $x[1 \ldots m - 1]$ does not occur in $y_1\ldots y_{N-1}$, so either $x[m - 1]$ does not occur in $y_1\ldots y_{N-1}$ which means that $x[m - 1] \in R^\ell_{y_1\ldots y_{N-1}}$, or let $x[p \ldots m - 1]$ be the longest suffix of $x[1 \ldots m - 1]$ that occurs in $y_1\ldots y_{N-1}$.

Because $0 < p \leq m - 1$ then $x[p - 1 \ldots m - 2]$ occurs in $y_1\ldots y_{N-1}$, therefore $x[p - 1 \ldots m - 1] \in M^\ell_{y_1\ldots y_{N-1}}$. Since all factors of $x[p - 1 \ldots m - 1]$ occur in $y_N$, we have $x[p - 1 \ldots m - 1] \in R^\ell_{y_1\ldots y_{N-1}}$. \qed

**4 Algorithm**

Let us first introduce an algorithmic tool. In the *weighted ancestor* problem, introduced in [13], we consider a rooted tree $T$ with an integer weight function $\mu$ defined on the nodes. We require that the weight of the root is zero and the weight of any other node is strictly larger than the weight of its parent. A weighted ancestor query, given a node $v$ and an integer value $w \leq \mu(v)$, asks for the highest ancestor $u$ of $v$ such that $\mu(u) \geq w$, i.e., such an ancestor $u$ that $\mu(u) \geq w$ and $\mu(u)$ is the smallest possible. When $T$ is the suffix tree of a word $y$ of length $n$, we can locate the locus of any factor $y[i..j]$ using a weighted ancestor query. We define the weight of a node of the suffix tree as the length of the word it represents. Thus a weighted ancestor query can be used for the terminal node decorated with $i$ to create (if necessary) and mark the node that corresponds to $y[i..j]$.

**Theorem 2 ([3]).** Given a collection $Q$ of weighted ancestor queries on a weighted tree $T$ on $n$ nodes with integer weights up to $n^{O(1)}$, all the queries in $Q$ can be answered off-line in $O(n + |Q|)$ time.

**4.1 The Algorithm**

At the $N$th step, we have in memory the set $M^\ell_{y_1\ldots y_{N-1}}$. Our algorithm works as follows:
1. We read word $y_N$ from the disk and compute $M^\ell_{y_N}$ in time $O(|y_N|)$. We output the words in the following constant-space form: $<i_1, i_2, \alpha>$ per word [2]; such that $y_N[i_1 \ldots i_2] \cdot \alpha \in M^\ell_{y_N}$.

2. Here we compute Case 1 MAWs. We apply Lemma 1 to construct set $M = \{ w : w \in M^\ell_{y_1 \ldots \#y_N} \}$ and the sets $R^\ell_{y_1 \ldots \#y_{N-1}}$ as follows.

(a) We first want to find the elements of $M^\ell_{y_1 \ldots \#y_{N-1}}$ that are superwords of any word $y_N[i_1 \ldots i_2] \cdot \alpha$. We build the generalised suffix tree $T_1 = T(M^\ell_{y_1 \ldots \#y_{N-1}} \cup \{y_N\})$ [18]. We find the locus of the longest proper prefix $y_N[i_1 \ldots i_2]$ of each element of $M^\ell_{y_N}$ in $T_1$ via answering off-line a batch of weighted ancestor queries (Theorem 2). From there on, we spell $\alpha$ and mark the corresponding node on $T_1$, if any. After processing all $<i_1, i_2, \alpha>$ in the same manner, we traverse $T_1$ to gather all occurrences (starting positions) of words $y_N[i_1 \ldots i_2] \cdot \alpha$ in the elements of $M^\ell_{y_1 \ldots \#y_{N-1}}$, thus finding the elements of $M^\ell_{y_1 \ldots \#y_{N-1}}$ that are superwords of any $y_N[i_1 \ldots i_2] \cdot \alpha$. By definition, no MAW $y_N[i_1 \ldots i_2] \cdot \alpha$ is a prefix of another MAW $y_N[i'_1 \ldots i'_2] \cdot \alpha'$, thus the marked nodes form pairwise disjoint subtrees, and the whole process takes time $O(|y_N| + |M^\ell_{y_1 \ldots \#y_{N-1}}|)$, the size of $T_1$.

(b) We next want to check if the words $y_N[i_1 \ldots i_2] \cdot \alpha$ are superwords of any element of $M^\ell_{y_1 \ldots \#y_{N-1}}$. We first sort all tuples $<i_1, i_2, \alpha>$ using radixsort and then check this using the matching statistics algorithm for $y_N$ with respect to $T(M^\ell_{y_1 \ldots \#y_{N-1}})$ considering the tuples in ascending order (from left to right) at the same time. By definition, no element in $M^\ell_{y_1 \ldots \#y_{N-1}}$ is a factor of another element in the same set. Thus if a factor of $y_N[i_1 \ldots i_2] \cdot \alpha$ corresponds to an element in $M^\ell_{y_1 \ldots \#y_{N-1}}$ this is easily located in $T(M^\ell_{y_1 \ldots \#y_{N-1}})$ while running the matching statistics algorithm. The whole process takes $O(|y_N| + |M^\ell_{y_1 \ldots \#y_{N-1}}|)$ time: $O(|M^\ell_{y_1 \ldots \#y_{N-1}}|)$ time to construct the suffix tree and a further $O(|y_N|)$ time for the matching statistics algorithm and for processing the $O(|y_N|)$ tuples.

We create set $R^\ell_{y_1 \ldots \#y_{N-1}}$ explicitly since it is a subset of $M^\ell_{y_1 \ldots \#y_{N-1}}$. We create set $R^\ell_{y_N}$ implicitly: every element $x \in R^\ell_{y_N}$ is stored as a tuple $<i_1, i_2, \alpha>$ such that $x = y_N[i_1 \ldots i_2] \cdot \alpha$. We store every element of $\{x_2 : x_2 \in M \cap M^\ell_{y_N}\}$ with the same representation. All other elements of $M$ are stored explicitly.

3. Construct the suffix tree of $y_N$ and use it to locate all occurrences of words in $R^\ell_{y_1 \ldots \#y_{N-1}}$ in $y_N$ and store the occurrences as pairs (starting position, ending position). This step can be done in time $O(|y_N| + |R^\ell_{y_1 \ldots \#y_{N-1}}|)$. By definition, no element in $R^\ell_{y_1 \ldots \#y_{N-1}}$ is a prefix of another element in $R^\ell_{y_1 \ldots \#y_{N-1}}$, and thus this can be done within the claimed time complexity.

4. For every $i \in [1, N - 1]$, we perform the following to compute Case 2 MAWs:

(a) Read word $y_i$ from the disk. Construct the suffix tree $T_x$ of word $x = y_i \#y_N$ in time $O(|y_i| + |y_N|)$. Use $T_x$ to locate all occurrences of elements of $M^\ell_{y_N}$ in $y_i$ and store the occurrences as pairs (starting position, ending position). This step can be done in time $O(|y_i| + |y_N|)$ similar to step 2. By definition, no element in $R^\ell_{y_N}$ is a prefix of another element in $R^\ell_{y_N}$, and thus this can be done within the claimed time complexity.

(b) During a bottom-up traversal of $T_x$ mark, at each explicit node of $T_x$, the smallest starting position of the subword represented by that node, and the largest starting position of
The space is bounded by the maximum time spent at a single step; namely, the length of the longest word in the collection plus the maximum total size of set elements across all output

\[ |y| + |y_N| \leq \sum_{N=1}^{k} (|y_N| + |y_i|) = \sum_{N=1}^{k} |y_N| + \sum_{N=1}^{k} |y_i| = 2k(|y_1| + \cdots + |y_k|). \]
5 Proof-of-Concept Experiments

In this section, we do not directly compare against the fastest internal [2] or external [19] memory implementations because the former assumes that we have the required amount of internal memory, and the latter assumes that we have the required amount of external memory to construct and store the global data structures for a given input dataset. If the memory for constructing and storing the data structures is available, these linear-time algorithms are surely faster than the method proposed here. In what follows, we rather show that our output-sensitive technique offers a space-time tradeoff, which can be usefully exploited for specific values of \( \ell \), the maximal length of MAWs we wish to compute.

The algorithm discussed in Section 4 (with the exception of storing and searching the reduced set words explicitly rather than in the constant-space form previously described) has been implemented in the C++ programming language\(^1\). The correctness of our implementation has been confirmed against that of [2]. As input dataset here we used the entire human genome (version hg38) [21], which has an approximate size of 3.1GB. The following experiment was conducted on a machine with an Intel Core i5-4690 CPU at 3.50 GHz and 128GB of memory running GNU/Linux. We ran the program by splitting the genome into \( k = 2, 4, 6, 8, 10 \) blocks and setting \( \ell = 10, 11, 12 \). Figure 3 depicts the change in elapsed time and peak memory usage as \( k \) and \( \ell \) increase (space-time tradeoff).

Graph (a) shows an increase of time as \( k \) and \( \ell \) increase; and graph (b) shows a decrease in memory as \( k \) increases (as proved in Theorem 3). Notice that the space to construct the block-wise data structures bounds the total space used for the specific \( \ell \) values and that is why the memory peak is essentially the same for the \( \ell \) values used. This can specifically be seen for \( \ell = 10 \) where all words of length 10 are present in the genome. The same dataset was used to run the fastest internal memory implementation for computing MAWs [2] on the same machine. It took only 2242 seconds to compute all MAWs but with a peak memory usage of 60.80GB. The results confirm our theoretical findings and justify our contribution.

\(^{1}\)The implementation can be made available upon request.
References

[1] Yannis Almirantis, Panagiotis Charalamposopoulos, Jia Gao, Costas S. Iliopoulos, Manal Mohamed, Solon P. Pissis, and Dimitris Polychronopoulos. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology*, 12(1):5:1–5:12, 2017.

[2] Carl Barton, Alice Héliou, Laurent Mouchard, and Solon P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinformatics*, 15:388, 2014.

[3] Carl Barton, Tomasz Kociumaka, Chang Liu, Solon P. Pissis, and Jakub Radoszewski. Indexing weighted sequences: Neat and efficient. *CoRR*, abs/1704.07625, 2017.

[4] Djamal Belazzougui and Fabio Cunial. A framework for space-efficient string kernels. *Algorithmica*, 79(3):857–883, 2017.

[5] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Versatile succinct representations of the bidirectional Burrows-Wheeler transform. In Hans L. Bodlaender and Giuseppe F. Italiano, editors, *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2013.

[6] Panagiotis Charalamposopoulos, Maxime Crochemore, Gabriele Fici, Robert Mercaş, and Solon P. Pissis. Alignment-free sequence comparison using absent words. *Information and Computation*, 262(1):57–68, 2018.

[7] Panagiotis Charalamposopoulos, Maxime Crochemore, and Solon P. Pissis. On extended special factors of a word. In Gagie et al. [17], pages 131–138.

[8] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[9] Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon P. Pissis, and Yann Ramusat. Minimal absent words in a sliding window and applications to on-line pattern matching. In Ralf Klasing and Marc Zeitoun, editors, *Fundamentals of Computation Theory - 21st International Symposium, FCT 2017, Bordeaux, France, September 11-13, 2017, Proceedings*, volume 10472 of *Lecture Notes in Computer Science*, pages 164–176. Springer, 2017.

[10] Maxime Crochemore, Filippo Mignosi, and Antonio Restivo. Automata and forbidden words. *Information Processing Letters*, 67(3):111–117, 1998.

[11] Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Sergio Salemi. Data compression using antidictionaries. *Proceedings of the IEEE*, 88(11):1756–1768, 2000.

[12] Maxime Crochemore and Gonzalo Navarro. Improved antidictionary based compression. In *22nd International Conference of the Chilean Computer Science Society (SCCC 2002)*, 6-8 November 2002, Copiapo, Chile, pages 7–13, 2002.

[13] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In Daniel S. Hirschberg and Eugene W. Myers, editors, *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings*, volume 1075 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1996.
[14] Martin Fiala and Jan Holub. DCA using suffix arrays. In 2008 Data Compression Conference (DCC 2008), 25-27 March 2008, Snowbird, UT, USA, page 516. IEEE Computer Society, 2008.

[15] Yuta Fujishige, Takuya Takagi, and Diptarama Hendrian. Truncated DAWGs and their application to minimal absent word problem. In Gagie et al. [17], pages 139–152.

[16] Yuta Fujishige, Yuki Tsujimaru, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing DAWGs and minimal absent words in linear time for integer alphabets. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland, volume 58 of LIPIcs, pages 38:1–38:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[17] Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors. String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings, volume 11147 of Lecture Notes in Computer Science. Springer, 2018.

[18] Dan Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, New York, NY, USA, 1997.

[19] Alice Héliou, Solon P. Pissis, and Simon J. Puglisi. emMAW: computing minimal absent words in external memory. Bioinformatics, 33(17):2746–2749, 2017.

[20] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Parallel external memory suffix sorting. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings, volume 9133 of Lecture Notes in Computer Science, pages 329–342. Springer, 2015.

[21] W. James Kent, Charles W. Sugnet, Terrence S. Furey, Krishna M. Roskin, Tom H. Pringle, Alan M. Zahler, Haussler, and David. The human genome browser at UCSC. Genome Research, 12(6):996–1006, 2002.

[22] Takahiro Ota and Hiroyoshi Morita. On the adaptive antidictionary code using minimal forbidden words with constant lengths. In Proceedings of the International Symposium on Information Theory and its Applications, ISITA 2010, 17-20 October 2010, Taichung, Taiwan, pages 72–77. IEEE, 2010.

[23] Raquel M. Silva, Diogo Pratas, Luísa Castro, Armando J. Pinho, and Paulo Jorge S. G. Ferreira. Three minimal sequences found in Ebola virus genomes and absent from human DNA. Bioinformatics, 31(15):2421–2425, 2015.