Citation


Persistent URL

https://research.gold.ac.uk/id/eprint/25916/

Versions

The version presented here may differ from the published, performed or presented work. Please go to the persistent GRO record above for more information.

If you believe that any material held in the repository infringes copyright law, please contact the Repository Team at Goldsmiths, University of London via the following email address: gro@gold.ac.uk.

The item will be removed from the repository while any claim is being investigated. For more information, please contact the GRO team: gro@gold.ac.uk
Constructing Antidictionaries in Output-Sensitive Space

Lorraine A.K. Ayad*, Golnaz Badkobeh†, Gabriele Fici‡, Alice Héliou§ and Solon P. Pissis†

*Department of Informatics
King’s College London, London, UK
lorraine.ayad@kcl.ac.uk

†Department of Computing
Goldsmiths University of London, London, UK
g.badkobeh@gold.ac.uk

‡Dipartimento di Matematica e Informatica
Università di Palermo, Palermo, Italy
gabriele.fici@unipa.it

§Independent Researcher
alice.heliou@gmail.com

†‡CWI, Amsterdam, The Netherlands
solon.pissis@cwi.nl

Abstract

A word $x$ that is absent from a word $y$ is called 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 $M_{y_1 \# \ldots \# y_k}$ 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. The 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 $||M_{y_1 \# \ldots \# y_N}|| = o(n)$, for all $N \in [1, k]$. For instance, in the human genome, $n \approx 3 \times 10^9$ but $||M_{y_1 \# \ldots \# y_k}|| \approx 10^6$. We 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 $O(kn + \sum_{N=1}^k ||M_{y_1 \# \ldots \# y_N}||)$ 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 $\text{MaxOut} = \max\{|M_{y_1 \# \ldots \# y_N}|| : N \in [1, k]\}$. Proof-of-concept experimental results are also provided confirming our theoretical findings and justifying our contribution.

1 Introduction

The word $x$ is an absent word of the word $y$ if it does not occur in $y$. The absent word $x$ of $y$ is called 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 $M_y^\ell$. For example, if $y = \text{abaab}$, then $M_y = \{\text{aaa}, \text{aaba}, \text{bab}, \text{bb}\}$ and $M_y^3 = \{\text{aaa}, \text{bab}, \text{bb}\}$. The upper bound on the number of minimal absent words is $O(\sigma n)$ [1], where $\sigma$ is the size of the alphabet and $n$ is the length of $y$, and this is tight for integer alphabets [2]; 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 [3].

State-of-the-art algorithms compute all minimal absent words of $y$ in $O(\sigma n)$ time [1, 4] or in $O(n + |M_y|)$ time [5, 6] 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 [7, 8]. In many real-world applications of minimal absent words, such as in data compression [9, 10, 11, 12], in sequence comparison [2, 6], in on-line pattern matching [13], or in identifying pathogen-specific signatures [14], 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 $\Theta(\sigma n)$, $\Omega(\sigma n)$ space is required to represent them explicitly. In [6], 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 [4]. In theory, this problem can be addressed by using the external memory algorithm for computing minimal absent words presented in [15]. The I/O-optimal version of this algorithm, however, requires a lot of external memory to build the global data structures for the input [16]. One could also use the algorithm of [17] that computes $M_y$ in $O(n + |M_y|)$ 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 [9, 10]. 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 $||M_{y_1\ldots\#y_N}^{12}|| \approx 10^6$, 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 $M_{y_1}^\ell, \ldots, M_{y_1\ldots\#y_k}^\ell$ can be computed in $O(kn + \sum_{N=1}^k ||M_{y_1\ldots\#y_N}^\ell||)$ 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 $\text{MAXOUT} = \max\{||M_{y_1\ldots\#y_N}^\ell|| : N \in [1, k]\}$. Proof-of-concept experimental results are provided confirming our theoretical findings and justifying our contribution.

## 2 Preliminaries

We generally follow [18]. 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., $|\Sigma| = O(1)$. Given a word $y = u x v$ 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 $x \neq y$. In theory, this problem can be addressed by using the external memory algorithm for computing minimal absent words presented in [15]. The I/O-optimal version of this algorithm, however, requires a lot of external memory to build the global data structures for the input [16]. One could also use the algorithm of [17] that computes $M_y^\ell$ in $O(n + |M_y|)$ 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 [9, 10]. 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 $||M_{y_1\ldots\#y_N}^{12}|| \approx 10^6$, 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 $M_{y_1}^\ell, \ldots, M_{y_1\ldots\#y_k}^\ell$ can be computed in $O(kn + \sum_{N=1}^k ||M_{y_1\ldots\#y_N}^\ell||)$ 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 $\text{MAXOUT} = \max\{||M_{y_1\ldots\#y_N}^\ell|| : N \in [1, k]\}$. Proof-of-concept experimental results are provided confirming our theoretical findings and justifying our contribution.

## 2 Preliminaries

We generally follow [18]. 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., $|\Sigma| = O(1)$. Given a word $y = u x v$ 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 $x \neq y$. In theory, this problem can be addressed by using the external memory algorithm for computing minimal absent words presented in [15]. The I/O-optimal version of this algorithm, however, requires a lot of external memory to build the global data structures for the input [16]. One could also use the algorithm of [17] that computes $M_y^\ell$ in $O(n + |M_y|)$ 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 [9, 10]. 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 $||M_{y_1\ldots\#y_N}^{12}|| \approx 10^6$, 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 $M_{y_1}^\ell, \ldots, M_{y_1\ldots\#y_k}^\ell$ can be computed in $O(kn + \sum_{N=1}^k ||M_{y_1\ldots\#y_N}^\ell||)$ 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 $\text{MAXOUT} = \max\{||M_{y_1\ldots\#y_N}^\ell|| : N \in [1, k]\}$. Proof-of-concept experimental results are provided confirming our theoretical findings and justifying our contribution.
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 $M_y = \{a a, b b, b a, c\}$. 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 MAW 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 $y$ [18]. 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 $L(v)$ denote the path-label from the root node to node $v$. We say that node $v$ is path-labeled $L(v)$; i.e., the concatenation of the edge labels along the path from the root node to $v$. Additionally, $D(v) = |L(v)|$ is used to denote the word-depth of node $v$. A node $v$ such that the path-label $L(v) = y[i..n-1]$, 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 $L(v) = aw$ 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 $x[0..|x| - 1]$ with respect to word $y$ is an array $MS_x[0..|x| - 1]$, where $MS_x[i]$ is a pair $(f_i, p_i)$ such that $(i) x[i..i+f_i-1]$ is the longest prefix of $x[i..|x| - 1]$ that is a factor of $y$; and (ii) $y[p_i..p_i+f_i-1] = x[i..i+f_i-1]$ [19]. $T(y)$ is constructible in time $O(n)$, and, given $T(y)$, we can compute $MS_x$ in time $O(|x|)$ [19].

## 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, $x \in M_{y_3}^\ell$. 

Let $x$ be factors one of another. Inspect Figure 1 in this regard.

**Example 1.** Let $y_1 = abaab$, $y_2 = bbabaab$ and $\ell = 5$. We have $M_{y_1}^{\ell} = \{bb, aaa, bab, aaba\}$ 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 $R_{y_1}^{\ell} = \{bb, aaa\}$ 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 $x \in M_{y_3}^{\ell} \setminus (M_{y_1}^{\ell} \cup M_{y_2}^{\ell})$. Then $x$ has a prefix $x_i$ in $R_{y_i}^{\ell}$ and a suffix $x_j$ in $R_{y_j}^{\ell}$, for $i, j$ such that $\{i, j\} = \{1, 2\}$.

**Proof.** Let $x = aub$, $a, b \in \Sigma$, be a word in $M_{y_3}^{\ell} \setminus (M_{y_1}^{\ell} \cup M_{y_2}^{\ell})$. 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 $y_1 = abaab$, $y_2 = bbabaab$ and $\ell = 5$. Consider $x = abaaa \in M_{y_1}^{\ell} \setminus (M_{y_1}^{\ell} \cup M_{y_2}^{\ell})$ (Case 2 MAW). We have that $abaa$ occurs in $y_1$ but not in $y_2$ and $baaa$ occurs in $y_2$ but not
Theorem 1. Let \( M > 1 \), and let \( x \in M \setminus (M \cup M) \), we should consider all pairs \( (i, j) \) with \( x \in M \setminus y \), and \( y \in M \). In order to construct the final set \( M \), 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 \( x \in M \cup M \) (Case 1 MAWs) or, otherwise, \( x \in M \setminus (M \cup M) \) for some \( i \). Moreover, in this latter case, \( x \) has a prefix in \( R \) and a suffix in \( R \), or the converse, i.e., \( x \) has a prefix in \( R \) and a suffix in \( R \) (Case 2 MAWs).

**Proof.** Let \( x \in M \) and \( x \notin M \) (Case 1 MAWs). Then, \( x \notin M \) and \( x \notin M \). Let \( x \) be a word of length \( m \). By the definition of MAW, \( x[0..m-2] \) and \( x[1..m-1] \) must both be factors of \( y \). However, both cannot be factors of \( y \) and both cannot be factors of \( y \). Therefore, we have one of the two cases:

**Case 1:** \( x[0..m-2] \) is factor of \( y \) but not of \( y \) and \( x[1..m-1] \) is a factor of \( y \) but not of \( y \).

**Case 2:** \( x[0..m-2] \) is factor of \( y \) but not of \( y \) and \( x[1..m-1] \) is a factor of \( y \) but not of \( y \).

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

Because \( 0 \leq t < m - 1 \) then \( x[1..t+1] \) is a factor of \( y \). Therefore, \( x[0..t+1] \) is a factor of \( y \). In addition, all factors of \( x[0..t+1] \) occur in \( y \), so \( x[0..t+1] \) is a factor of \( y \).

Now, \( x[1..m-1] \) does not occur in \( y \) which means that \( x[m-1] \) does not occur in \( y \). Therefore, \( x[1..m-1] \) occurs in \( y \), and \( x[p..m-1] \) is the longest suffix of \( x[1..m-1] \) that occurs in \( y \).

Because \( 0 < p \leq m - 1 \) then \( x[p-1..m-2] \) occurs in \( y \), therefore \( x[p-1..m-1] \) occurs in \( y \). Since all factors of \( x[p-1..m-1] \) occur in \( y \), we have \( x[p-1..m-1] \in R \).

**4 Algorithm**

Let us first introduce an algorithmic tool. In the weighted ancestor problem, introduced in [20], 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 \( w \), 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** ([21]). *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 \(M_{y_1\#...\#y_{N-1}}^\ell\). Our algorithm works as follows:

1. We read word \(y_N\) from the disk and compute \(M_{y_N}^\ell\) in time \(O(|y_N|)\). We output the words in the following constant-space form: \(<i_1, i_2, \alpha>\) per word [4]; such that \(y_N[i_1..i_2] \cdot \alpha \in M_{y_N}^\ell\).

2. Here we compute Case 1 MAWs. We apply Lemma 1 to construct set \(M = \{w : w \in M_{y_1\#...\#y_{N-1}}^\ell, w \in M_{y_1\#...\#y_{N-1}}^\ell \cup M_{y_N}^\ell\}\) and the sets \(R_{y_1\#...\#y_{N-1}}^\ell, R_{y_N}^\ell\) as follows.

   (a) We first want to find the elements of \(M_{y_1\#...\#y_{N-1}}^\ell\) that are superwords of any word \(y_N[i_1..i_2] \cdot \alpha\). We build the generalised suffix tree \(T_{\ell} = T(M_{y_1\#...\#y_{N-1}}^\ell \cup \{y_N\})\) [19]. We find the locus of the longest proper prefix \(y_N[i_1..i_2]\) of each element of \(M_{y_N}^\ell\) in \(T_{\ell}\) via answering off-line a batch of weighted ancestor queries (Theorem 2). From there on, we spell \(\alpha\) and mark the corresponding node on \(T_{\ell}\), if any. After processing all \(<i_1, i_2, \alpha>\) in the same manner, we traverse \(T_{\ell}\) to gather all occurrences (starting positions) of words \(y_N[i_1..i_2] \cdot \alpha\) in the elements of \(M_{y_1\#...\#y_{N-1}}^\ell\), thus finding the elements of \(M_{y_1\#...\#y_{N-1}}^\ell\) that are superwords of any \(y_N[i_1..i_2] \cdot \alpha\). By definition, no MAW \(y_N[i_1..i_2] \cdot \alpha\) is a prefix of another MAW \(y_N[i'_1..i'_2] \cdot \alpha'\), thus the marked nodes form pairwise disjoint subtrees, and the whole process takes time \(O(|y_N| + |M_{y_1\#...\#y_{N-1}}^\ell|)\), the size of \(T_{\ell}\).

   (b) We next want to check if the words \(y_N[i_1..i_2] \cdot \alpha\) are superwords of any element of \(M_{y_1\#...\#y_{N-1}}^\ell\). 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 \(T(M_{y_1\#...\#y_{N-1}}^\ell)\) considering the tuples in ascending order (from left to right) at the same time. By definition, no element in \(M_{y_1\#...\#y_{N-1}}^\ell\) is a factor of another element in the same set. Thus if a factor of \(y_N[i_1..i_2] \cdot \alpha\) corresponds to an element in \(M_{y_1\#...\#y_{N-1}}^\ell\) this is easily located in \(T(M_{y_1\#...\#y_{N-1}}^\ell)\) while running the matching statistics algorithm. The whole process takes \(O(|y_N| + |M_{y_1\#...\#y_{N-1}}^\ell|)\) time: \(O(|M_{y_1\#...\#y_{N-1}}^\ell|)\) 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 \(R_{y_1\#...\#y_{N-1}}^\ell\) explicitly since it is a subset of \(M_{y_1\#...\#y_{N-1}}^\ell\). We create set \(R_{y_N}^\ell\) implicitly: every element \(x \in R_{y_N}^\ell\) is stored as a tuple \(<i_1, i_2, \alpha>\) such that \(x = y_N[i_1..i_2] \cdot \alpha\). We store every element of \(\{x_2 : x_2 \in M \cap M_{y_N}^\ell\}\) 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 \( R^\ell_{y_N} \) 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 same subword. This can be done in time \( O(|y_i| + |y_N|) \) by propagating upwards the labels of the terminal nodes (starting positions of suffixes) and updating the smallest and largest positions analogously.

   (c) Compute the set \( M^\ell_{y_i \# y_N} \) and output the words in the following constant-space form: \( < a, i_1, i_2, b > \) per word; such that \( a \cdot x[i_1 \ldots i_2] \cdot b \) is a MAW. This can be done in time \( O(|y_i| + |y_N|) \).

   (d) For each element of \( M^\ell_{y_i \# y_N} \), we need to locate the node representing word \( ax[i_1 \ldots i_2] = au \) and the node representing word \( x[i_1 \ldots i_2]b = ub \). This can be done in time \( O(|y_i| + |y_N|) \) via answering off-line a batch of weighted ancestor queries (Theorem 2). At this point, we have located the two nodes on \( T_x \). We assign a pointer from the stored starting position \( g \) of \( au \) to the ending position \( f \) of \( ub \), only if \( g \) is before \( \# \) and \( f \) is after \( \# \) (\( f \) can be trivially computed using the stored starting position of \( ub \) and the length of \( ub \)). Conversely, we assign a pointer from the ending position \( f \) of \( ub \) to the stored starting position \( g \) of \( au \), only if \( f \) is before \( \# \) and \( g \) is after \( \# \).

   (e) Suppose \( au \) occurs in \( y_i \) and \( ub \) in \( y_N \). We make use of the pointers as follows. Recall steps 3 and 4(a) and check whether \( au \) starts where a word \( r_1 \) of \( R^\ell_{y_N} \) starts and \( ub \) ends where a word \( r_2 \) of \( R^\ell_{y_1 \# \ldots \# y_{N-1}} \) ends. If this is the case and \( |u| \geq \max\{|r_1|, |r_2|\} - 1 \), then by Theorem 1 \( aub \) is added to our output set \( M \), otherwise discard it. Inspect Figure 2 in this regard. Conversely, if \( au \) occurs in \( y_N \) and \( ub \) in \( y_i \) check whether \( au \) starts where a word \( r_2 \) of \( R^\ell_{y_1 \# \ldots \# y_{N-1}} \) starts and whether \( ub \) ends where a word \( r_1 \) of \( R^\ell_{y_N} \) ends. If this is the case and \( |u| \geq \max\{|r_1|, |r_2|\} - 1 \), then \( aub \) is added to \( M \), otherwise discard it.

Finally, we set \( M^\ell_{y_1 \# \ldots \# y_N} = M \) as the output of the Nth step. Let \( \text{MAXIN} \) be the length of the longest word in \( \{y_1, \ldots, y_k\} \) and \( \text{MAXOUT} = \max\{|M^\ell_{y_1 \# \ldots \# y_N}| : N \in [1, k]\} \).
Figure 2: *au* starts where a word *r*₁ of *R*₁ *y* starts in *y*₁ and *ub* ends where a word *r*₂ of *R*₂ *y*₁ *#*...*#y*ₙ⁻¹ ends in *y*ₙ. Moreover, if |*u*| ≥ max{|*r*₁|, |*r*₂|} − 1, then *aub* is a Case 2 MAW.

**Theorem 3.** Given *k* words *y*₁, *y*₂, ..., *y*ₙ and an integer *ℓ* > 0, all *M*ₘₙ *y*₁, ..., *M*ₘₙ *y*ₙ can be computed in \( O(kn + \sum_{N=1}^{N-1} \| M_{y_1#...#y_N} \|) \) total time using \( O(\text{MaxIn} + \text{MaxOut}) \) space, where \( n = |y_1#...#y_k| \).

**Proof.** From the above discussion, the time is bounded by \( O(\sum_{N=1}^{N-1} \sum_{i=1}^{k} (|y_N| + |y_i|)) \)\). We can bound the first term as follows.

\[
\sum_{N=1}^{k} \sum_{i=1}^{N-1} (|y_N| + |y_i|) \leq \sum_{N=1}^{k} \sum_{i=1}^{k} (|y_N| + |y_i|) = \sum_{N=1}^{k} \sum_{i=1}^{k} |y_N| + \sum_{N=1}^{k} \sum_{i=1}^{k} |y_i| = 2k(|y_1| + \cdots + |y_k|).
\]

Therefore the time is bounded by \( O(kn + \sum_{N=1}^{k} \| M_{y_1#...#y_N} \|) \).

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 sets. Note that the total output size of the algorithm is the sum of all its output sets, that is \( \sum_{N=1}^{k} \| M_{y_1#...#y_N} \| \), and \( \text{MaxOut} \) could come from any intermediate set.

The correctness of the algorithm follows from Lemma 1 and Theorem 1. \( \Box \)

5 Proof-of-Concept Experiments

In this section, we do not directly compare against the fastest internal [4] or external [15] 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 *ℓ*, 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 [4]. As input dataset here we used the entire human genome (version hg38) [22], 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

\(^1\)The implementation can be made available upon request.
Figure 3: Elapsed time and peak memory usage using increasing $k$ blocks of the entire human genome for $\ell = 10, 11, 12$; notice that the peak memory usage is the same for all values of $\ell$.

$k = 2, 4, 6, 8, 10$ 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 [4] 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.

References


[17] Yuta Fujishige, Takuya Takagi, and Diptarama Hendrian, “Truncated DAWGs and their application to minimal absent word problem,” in Gagie et al. [23], pp. 139–152.


