Internal Shortest Absent Word Queries

Golnaz Badkobeh
Department of Computing, Goldsmiths University of London, UK

Panagiotis Charalampopoulos
Efi Arazi School of Computer Science, The Interdisciplinary Center Herzliya, Israel

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, we are to preprocess T so that given a range $[i, j]$, we can return a representation of a shortest string over Σ that is absent in the fragment $T[i] \cdots T[j]$ of T. For any positive integer $k \in [1, \log\log_\sigma n]$, we present an $\mathcal{O}((n/k) \cdot \log\log_\sigma n)$-size data structure, which can be constructed in $\mathcal{O}(n \log_\sigma n)$ time, and answers queries in time $\mathcal{O}(\log\log_\sigma k)$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string algorithms, internal queries, shortest absent word, bit parallelism

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.6

Related Version *Extended Version (with Dmitry Kosolobov)*: `https://arxiv.org/abs/2106.01763`

Funding *Panagiotis Charalampopoulos*: Supported by the Israel Science Foundation grant 592/17.

1 Introduction

Range queries are a classic data structure topic [63, 13, 12]. In 1d, a range query $q = f(A, i, j)$ on an array of n elements over some set U, denoted by $A[1 \mathinner{.\,.} n]$, takes two indices $1 \le i \le j \le n$, a function f defined over arrays of elements of U, and outputs $f(A[i \mathinner{.\,.} j]) = f(A[i], \ldots, A[j])$. Range query data structures in 1d can thus be viewed as data structures answering queries on a string in the internal setting, where U is the considered alphabet.

Asking internal queries on a string has received much attention in recent years. In the internal setting, we are asked to preprocess a string T of length n over an alphabet Σ of size σ, so that queries about substrings of T can be answered efficiently. Note that an arbitrary substring of T can be encoded in $\mathcal{O}(1)$ words of space by the indices i, j of an occurrence of it as a fragment $T[i] \cdots T[j] = T[i \mathinner{.\,.} j]$ of T. Data structures for answering internal queries are interesting in their own sake, but also have numerous applications in the design of algorithms and (more sophisticated) data structures in stringology. Because of these numerous applications, we usually place particular emphasis on the construction time – other than on space/query-time tradeoffs, which is the main focus in the classic data structure literature.

The most widely-used internal query is that of asking for the *longest common prefix* of two suffixes $T[i \mathinner{.\,.} n]$ and $T[j \mathinner{.\,.} n]$ of T. The classic data structure for this problem [48] consists of the suffix tree of T [25] and a lowest common ancestor data structure [37] over the suffix tree. It occupies $\mathcal{O}(n)$ space, it can be constructed in $\mathcal{O}(n)$ time, and it answers queries in $\mathcal{O}(1)$ time. In the word RAM model of computation with word size $\Theta(\log n)$ bits the construction time is not necessarily optimal. A sequence of works [61, 52, 14] has culminated in the recent optimal data structure of Kempa and Kociumaka [40]: it occupies $\mathcal{O}(n/\log_\sigma n)$ space, it can be constructed in $\mathcal{O}(n/\log_\sigma n)$ time, and it answers queries in $\mathcal{O}(1)$ time.

Another fundamental problem in this setting is the *internal pattern matching* (IPM) problem. It consists in preprocessing T so that we can efficiently compute the occurrences of a substring U of T in another substring V of T. For the decision version of the IPM problem, Keller et al. [39] presented a data structure of nearly-linear size supporting sublogarithmic-time queries. Kociumaka et al. [45] presented a data structure of linear size supporting constant-time queries when the ratio between the lengths of V and U is bounded by a constant. The $\mathcal{O}(n)$-time construction algorithm of the latter data structure was derandomised in [42]. In fact, Kociumaka et al. [45], using their efficient IPM queries as a subroutine, managed to show efficient solutions for other internal problems, such as for computing the periods of a substring (*period queries*, introduced in [44]), and for checking whether two substrings are rotations of one another (*cyclic equivalence queries*). Other problems that have been studied in the internal setting include string alignment [62, 18], approximate pattern matching [21], dictionary matching [20, 19], longest common substring [4], counting palindromes [59], range longest common prefix [3, 1, 49, 34], the computation of the lexicographically minimal or maximal suffix, and minimal rotation [7, 41], as well as of the lexicographically kth suffix [8]. We refer the interested reader to the Ph.D dissertation of Kociumaka [42], for a nice exposition.

In this work, we extend this line of research by investigating the following basic internal query, which, to the best of our knowledge, has not been studied previously. Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$, preprocess T so that given a range $[i, j]$, we can return a shortest string over Σ that does not occur in $T[i \mathinner{.\,.} j]$. The latter shortest string is also known as a shortest absent word in the literature. We work on the standard unit-cost word RAM model with machine word-size $w = \Theta(\log n)$ bits. We measure the space used by our algorithms and data structures in machine words, unless stated otherwise. We assume that we have random access to T and so our algorithms return a constant-space representation of a shortest string (a witness) consisting of a substring of T and a letter. A naïve solution for this problem precomputes a table of size $\mathcal{O}(n^2)$ that stores the answer for every possible query $[i, j]$. Our main result is the following.

▶ **Theorem 1.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, for any positive integer $k \in [1, \log\log_\sigma n]$, we can construct a data structure of size $\mathcal{O}((n/k) \cdot \log\log_\sigma n)$, in $\mathcal{O}(n \log_\sigma n)$ time, so that if query $[a, b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a \mathinner{.\,.} b]$ in $\mathcal{O}(\log\log_\sigma k)$ time.*

By setting $k = 1$ we get an $\mathcal{O}(n \log\log_\sigma n)$-size data structure with $\mathcal{O}(1)$ query time.

In the related *range shortest unique substring* problem, defined by Abedin et al. [2], the task is to construct a data structure over T to be able to answer the following type of online queries efficiently. Given a range $[i, j]$, return a shortest string with exactly one occurrence (starting position) in $[i, j]$. Abedin et al. presented a data structure of size $\mathcal{O}(n \log n)$ supporting $\mathcal{O}(\log_w n)$-time queries, where $w = \Theta(\log n)$ is the word size. Additionally, Abedin et al. [2] presented a data structure of size $\mathcal{O}(n)$ supporting $\mathcal{O}(\sqrt{n} \log^\epsilon n)$-time queries, where ϵ is an arbitrarily small positive constant.

Our Techniques

For clarity of exposition, in this overview, we skip the time-efficient construction algorithms of our data structures and only describe how to compute the *length* of a shortest absent word (without a witness) in $T[a \mathinner{.\,.} b]$; note that this length is at most $\log_\sigma n$. Let us also note that the length of a shortest absent word of T can be computed in $\mathcal{O}(n)$ time using the suffix tree of T [25]. It suffices to traverse the suffix tree of T recording the shortest string-depth ℓ, where an implicit or explicit node has less than σ outgoing edges.

First approach: We precompute, for each position i and for each length $j \in [1, \log_\sigma n]$, the ending position of the shortest prefix of $T[i \mathinner{.\,.} n]$ that contains an occurrence of each of the σ^j distinct words of length j. Then, a query for the length of a shortest absent word of $T[a \mathinner{.\,.} b]$ reduces to a predecessor query among the ending positions we have precomputed for position a. By maintaining these $\mathcal{O}(\log_\sigma n)$ ending positions in a fusion tree [32], we obtain a data structure of size $\mathcal{O}(n \log_\sigma n)$ supporting queries in $\mathcal{O}(\log_w \log n) = \mathcal{O}(1)$ time.

Second approach: We precompute, for each length $j \in [1, \log_\sigma n]$, all minimal fragments of T that contain an occurrence of each of the distinct σ^j words of length j. As these fragments are inclusion-free, we can encode them using two n-bit arrays storing their starting and ending positions in T, respectively. We thus require $\mathcal{O}(n)$ words of space in total over all js. Observe that $T[a \mathinner{.\,.} b]$ does not have an absent word of length j if and only if it contains a minimal fragment for length j; we can check this condition in $\mathcal{O}(1)$ time after augmenting the computed bit arrays with succinct rank and select data structures [38]. Finally, due to monotonicity (if $T[a \mathinner{.\,.} b]$ contains all strings of length $j + 1$ then T contains all strings of length j), we can binary search for the answer in $\mathcal{O}(\log \log_\sigma n)$ time.

Third approach: We rely on the following combinatorial observation: if the length of a shortest absent word of a string X over Σ is λ, we need to prepend $\Omega(\sigma^{d-1} \cdot \lambda)$ letters to X in order to obtain a string with a shortest absent word of length $\lambda + d$. (For intuition, think of $|X|$ as a constant; then, we essentially need to prepend the de Bruijn sequence of order d over Σ to X in order to achieve the desired result.) This observation allows us to sparsify the information we stored in our first approach: for each length $j \in [1, \log_\sigma n]$, we use the value (previously) stored for some position i for an interval of positions. We then maintain a dynamic fusion tree over the stored values, which are now $o(n \log_\sigma n)$ in total, and make it persistent so that we can later query any version of it. As we will show, in the end we get the correct answer up to a small additive error, which we then eliminate by utilising the data structure developed in our second approach.

Let us remark that our partially persistent fusion trees allow us to obtain an alternative time-optimal data structure for the weighted ancestors problem [26] when the input tree of size n is of depth polylogarithmic in n. Such a data structure can be also easily derived from [47, 36].

Other Related Work

Let us recall that a string S that does not occur in T is called *absent* from T, and if all its proper substrings appear in T it is called a *minimal absent word* of T. It should be clear that every shortest absent word is also a minimal absent word. Minimal absent words (MAWs) are used in many applications [60, 56, 29, 35, 15, 54, 24] and their theory is well developed [51, 28, 30], also from an algorithmic and data structure point of view [50, 22, 9, 17, 16, 6, 33, 10, 23]. For example, it is well known that, given two strings X and Y, one has $X = Y$ if and only if X and Y have the same set of MAWs [51].

Paper Organization

Section 2 provides some preliminaries. The first approach is detailed in Section 3 and the second one in Section 4. Section 5 provides the combinatorial foundations for the third approach, which is detailed in Section 6.

2 Preliminaries

An *alphabet* Σ is a finite nonempty set whose elements are called *letters*. A *string* (or *word*) $S = S[1\mathinner{.\,.} n]$ is a sequence of *length* $|S| = n$ over Σ. The *empty* string ε is the string of length 0. The *concatenation* of two strings S and T is the string composed of the letters of S followed by the letters of T. It is denoted by $S \cdot T$ or simply by ST. The set of all strings (including ε) over Σ is denoted by Σ^*. The set of all strings of length $k > 0$ over Σ is denoted by Σ^k. For $1 \le i \le j \le n$, $S[i]$ denotes the ith letter of S, and the fragment $S[i\mathinner{.\,.} j]$ denotes an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$. We say that P *occurs* at (starting) *position* i in S. P is called *absent* from S if it does not occur in S. A substring $S[i\mathinner{.\,.} j]$ is a *suffix* of S if $j = n$ and it is a *prefix* of S if $i = 1$.

The following proposition is straightforward (as explained in Section 1).

▶ **Proposition 2.** *Let T be a string of length n. A shortest absent word of T can be computed in $\mathcal{O}(n)$ time.*

Given an array A of n items taken from a totally ordered set, the *range minimum query* $\mathsf{RMQ}_A(\ell, r) = \arg\min A[k]$ (with $1 \le \ell \le k \le r \le n$) returns the position of the minimal element in $A[\ell\mathinner{.\,.} r]$. The following result is known.

▶ **Theorem 3** ([12]). *Let A be an array of n integers. A data structure of size $\mathcal{O}(n)$ can be constructed in $\mathcal{O}(n)$ time supporting RMQs on A in $\mathcal{O}(1)$ time.*

We make use of *rank and select* data structures constructed over bit vectors. For a bit vector H we define $\mathsf{rank}_q(i, H) = |\{k \in [1, i] : H[k] = q\}|$ and $\mathsf{select}_q(i, H) = \min\{k \in [1, n] : \mathsf{rank}_q(k, H) = i\}$, for $q \in \{0, 1\}$. The following result is known.

▶ **Theorem 4** ([38, 53]). *Let H be a bit vector of n bits. A data structure of $o(n)$ additional bits can be constructed in $\mathcal{O}(n)$ time supporting* rank *and* select *queries on H in $\mathcal{O}(1)$ time.*

The *static predecessor* problem consists in preprocessing a set Y of integers, over an ordered universe U, so that, for any integer $x \in U$ one can efficiently return the predecessor $\mathsf{pred}(x) := \max\{y \in Y : y \le x\}$ of x in Y. The successor problem is defined analogously: upon a queried integer $x \in U$, the successor $\min\{y \in Y : y \ge x\}$ of x in Y is to be returned. Willard and Fredman designed the *fusion tree* data structure for this problem [32]. In the dynamic variant of the problem, updates to Y are interleaved with predecessor and successor queries. Pătraşcu and Thorup [57] presented a dynamic version of fusion trees, which, in particular yields an efficient construction of this data structure.

▶ **Theorem 5** ([32, 57]). *Let Y be a set of at most n w-bit integers. A data structure of size $\mathcal{O}(n)$ can be constructed in $\mathcal{O}(n \log_w n)$ time supporting insertions, deletions, and predecessor queries on Y in $\mathcal{O}(\log_w n)$ time.*

If $|U| = \mathcal{O}(n)$, then, after an $\mathcal{O}(n)$-time preprocessing, we can answer predecessor queries in $\mathcal{O}(1)$ time. For each $y \in Y$, we set the yth bit of an initially all-zeros $|U|$-size bit vector. We then preprocess this bit vector as in Theorem 4. Then, a predecessor query for any integer x can be answered in $\mathcal{O}(1)$ time due to the following readily verifiable formula: $\mathsf{pred}(x) = \mathsf{select}_1(\mathsf{rank}_1(x))$.

The main problem considered in this paper is formally defined as follows.

Internal Shortest Absent Word (ISAW)
Input: A string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size $\sigma > 1$.
Output: Given integers a and b, with $1 \le a \le b \le n$, output a shortest string in Σ^* with no occurrence in $T[a\mathinner{.\,.} b]$.

If $a = b$ then the answer is trivial. So, in what follows we assume that $a < b$. Let us also remark that the output (shortest absent word) can be represented in $\mathcal{O}(1)$ space using: either a range $[i, j] \subseteq [1, n]$ and a letter α of Σ, such that the shortest string in Σ^* with no occurrence in $T[a\,.\,.\,b]$ is $T[i\,.\,.\,j]\alpha$; or simply a range $[i, j] \subseteq [1, n]$ such that the shortest string in Σ^* with no occurrence in $T[a\,.\,.\,b]$ is $T[i\,.\,.\,j]$.

▶ **Example 6.** Given the string T = `abaabaaabbabbbaaab` and the range $[a, b] = [8, 14]$ (shown in red), the only shortest absent word of $T[8\,.\,.\,14]$ is $T[i\,.\,.\,j] = T[7\,.\,.\,8] =$ `aa`.

3 $\mathcal{O}(n \log_\sigma n)$ Space and $\mathcal{O}(1)$ Query Time

Let T be a string of length n. We define $S_T(j)$ as the function counting the cardinality of the set of length-j substrings of T. This is known as the *substring complexity* function [27, 58]. Note that $S_T(j) \leq n$, for all j. We have the following simple fact.

▶ **Fact 7.** *The length ℓ of a shortest absent word of a string T of length n over an alphabet of size σ is equal to the smallest j for which $S_T(j) < \sigma^j$ and hence $\ell \in [1, \lfloor \log_\sigma n \rfloor]$.*

We denote the set of shortest absent words of T by SAW_T. Recall that, by Proposition 2, a shortest absent word of T can be computed in $\mathcal{O}(n)$ time. We denote the length of the shortest absent words of T by ℓ. By Fact 7, $\ell \leq \lfloor \log_\sigma n \rfloor$. Since ℓ is an upper bound on the length of the answer for any ISAW query on T, in what follows, we consider only lengths in $[1, \ell - 1]$. Let one such length be denoted by j. By constructing and traversing the suffix tree of T, we can assign to each $T[i\,.\,.\,i + j - 1]$ its lexicographic rank in Σ^j. The time required for each length j is $\mathcal{O}(n)$, since the suffix tree of T can be constructed within this time [25]. Thus, the total time for all lengths $j \in [1, \ell - 1]$ is $\mathcal{O}(n \log_\sigma n)$ by Fact 7.

We design the following warm-up solution to the ISAW problem. For all $j \in [1, \ell - 1]$ we store an array RNK_j of n integers such that $\mathsf{RNK}_j[i]$ is equal to the lexicographic rank of $T[i\,.\,.\,i + j - 1]$ in Σ^j. Then, given a range $[a, b]$, in order to check if there is an absent word of length j in $T[a\,.\,.\,b]$ we only need to compute the number of distinct elements in $\mathsf{RNK}_j[a\,.\,.\,b - j + 1]$. It is folklore that using a persistent segment tree, we can preprocess an array A of n integers in $\mathcal{O}(n \log n)$ time so that upon a range query $[a, b]$ we can return the number of distinct elements in $A[a\,.\,.\,b]$ in $\mathcal{O}(\log n)$ time. Thus, we could use this tool as a black box for every array RNK_j resulting, however, in $\Omega(\log n)$-time queries. We improve upon this solution as follows.

We employ a range minimum query (RMQ) data structure [12] over a slight modification of RNK_j. For each j, we have an auxiliary procedure checking whether all strings from Σ^j occur in $T[a\,.\,.\,b]$ or not (i.e., it suffices to check whether any lexicographic rank is absent from the corresponding range). Similar to the previous solution, we rank the elements of Σ^j by their lexicographic order. We append RNK_j with all integers in $[1, \sigma^j]$. Let this array be APP_j. By Fact 7, we have that $|\mathsf{APP}_j| \leq 2n$. Then, we construct an array PRE_j of size $|\mathsf{APP}_j|$: $\mathsf{PRE}_j[i]$ stores the position of the rightmost occurrence of $\mathsf{APP}_j[i]$ in $\mathsf{APP}_j[1\,.\,.\,i - 1]$ (or 0 if such an occurrence does not exist). This can be done in $\mathcal{O}(n)$ time per j by sorting the list of pairs $(T[i\,.\,.\,i + j - 1], i)$, for all i, using the suffix tree of T to assign ranks for $T[i\,.\,.\,i + j - 1]$ and then radix sort to sort the list of pairs.

We now rely on the following fact.

▶ **Fact 8.** $S_{T[a\,.\,.\,b]}(j) = \sigma^j$ *if and only if* $\min\{\mathit{PRE}_j[i] : i \in [b - j + 2, |\mathit{PRE}_j|]\} \geq a$.

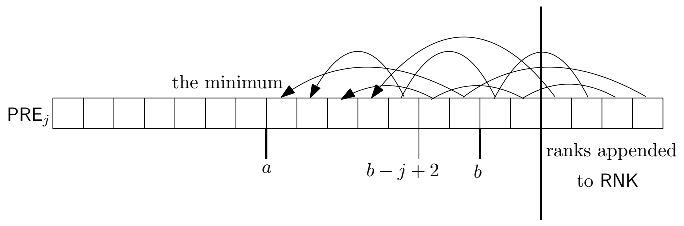

Figure 1 Illustration of the setting in Fact 8.

Proof. If the smallest element in $\mathsf{PRE}_j[b-j+2\,..\,|\mathsf{PRE}_j|]$, say $\mathsf{PRE}_j[k]$, is such that $\mathsf{PRE}_j[k] \geq a$, then all ranks of elements in Σ^j occur in $\mathsf{APP}_j[a\,..\,b-j+1]$. This is because all elements (ranks) in Σ^j occur at least once after $b-j+2$ (due to appending all integers in $[1, \sigma^j]$ to RNK_j), thus all must have a representative occurrence after $b-j+2$. Inspect Figure 1 for an illustration. (The opposite direction is analogous.) ◀

The following two examples illustrate the construction of arrays RNK_j, APP_j, and PRE_j as well as Fact 8.

▶ **Example 9** (Construction). Let $T = \texttt{abaabaaabbabbbaaab}$ and $\Sigma = \{\texttt{a}, \texttt{b}\}$. The set SAW_T of shortest absent words of T over Σ, each of length $\ell = 4$, is $\{\texttt{aaaa}, \texttt{abab}, \texttt{baba}, \texttt{bbbb}\}$. Arrays RNK_j, APP_j, and PRE_j, for all $j \in [1, \ell-1]$, are as follows: For instance, $\mathsf{RNK}_2[15] =$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T	a	b	a	a	b	a	a	a	b	b	a	b	b	b	a	a	a	b						
RNK_1	1	2	1	1	2	1	1	1	2	2	1	2	2	2	1	1	1	2						
APP_1	1	2	1	1	2	1	1	1	2	2	1	2	2	2	1	1	1	2	1	2				
PRE_1	0	0	1	3	2	4	6	7	5	9	8	10	12	13	11	15	16	14	17	18				
RNK_2	2	3	1	2	3	1	1	2	4	3	2	4	4	3	1	1	2							
APP_2	2	3	1	2	3	1	1	2	4	3	2	4	4	3	1	1	2	1	2	3	4			
PRE_2	0	0	0	1	2	3	6	4	0	5	8	9	12	10	7	15	11	16	17	14	13			
RNK_3	3	5	2	3	5	1	2	4	7	6	4	8	7	5	1	2								
APP_3	3	5	2	3	5	1	2	4	7	6	4	8	7	5	1	2	1	2	3	4	5	6	7	8
PRE_3	0	0	0	1	2	0	3	0	0	0	8	0	9	5	6	7	15	16	4	11	14	10	13	12

$\mathsf{APP}_2[15] = 1$ denotes that the lexicographic rank of $\texttt{aa}$ in Σ^2 is 1; and $\mathsf{PRE}_2[15] = 7$ denotes that the previous rightmost occurrence of $\texttt{aa}$ is at position 7.

▶ **Example 10** (Fact 8). Let $[a, b] = [7, 11]$ and $j = 2$ (see Example 9). The smallest element in $\{\mathsf{PRE}_2[11], \ldots, \mathsf{PRE}_2[21]\}$ is $\mathsf{PRE}_2[15] = 7 \geq a = 7$, which corresponds to rank $\mathsf{APP}_2[15] = 1$. Indeed all other ranks $2, 3, 4$ have at least one occurrence within $\mathsf{APP}_2[7\,..\,11] = 1, 2, 4, 3, 2$.

To apply Fact 8, we construct an RMQ data structure over PRE_j. By Theorem 3 it takes $\mathcal{O}(n)$ time and space and answers RMQs in $\mathcal{O}(1)$ time. This results in $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ preprocessing time and space for all j.

For querying, let us observe that $\sigma^j - S_{T[a\,..\,b]}(j)$, for any T, a, b and increasing j, is non-decreasing. We can thus apply binary search on j to find the smallest length j such that $S_{T[a\,..\,b]}(j) < \sigma^j$. This results in $\mathcal{O}(\log \ell) = \mathcal{O}(\log \log_\sigma n)$ query time. We obtain the following proposition (retrieving a witness shortest absent word is detailed later).

▶ **Proposition 11.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, we can construct a data structure of size $\mathcal{O}(n \log_\sigma n)$ in $\mathcal{O}(n \log_\sigma n)$ time, so that if query $[a, b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a\,..\,b]$ in $\mathcal{O}(\log \log_\sigma n)$ time.*

We further improve the query time via employing fusion trees as follows. We create a 2d array $\mathsf{FTR}[1\mathinner{.\,.}\ell-1][1\mathinner{.\,.}n]$ of integers, where

$$\mathsf{FTR}[j][i] = \min\{\mathsf{PRE}_j[i-j+2], \ldots, \mathsf{PRE}_j[|\mathsf{PRE}_j|]\},$$

for all $j \in [1, \ell-1]$ and $i \in [1, n]$. Intuitively, $\mathsf{FTR}[j][i]$ is the rightmost index of T such that $T[\mathsf{FTR}[j][i]\mathinner{.\,.}i]$ contains all strings of length j over Σ.

Array FTR can be constructed in $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ time by scanning each array PRE_j from right to left maintaining the minimum. Within the same complexities we also maintain satellite information specifying the index $k \in [i-j+2, |\mathsf{PRE}_j|]$ where the range minimum $\mathsf{FTR}[j][i]$ came from in the sub-array $\mathsf{PRE}_j[i-j+2\mathinner{.\,.}|\mathsf{PRE}_j|]$. We then construct n fusion trees, one for every collection of $\ell-1$ integers in $\mathsf{FTR}[1\mathinner{.\,.}\ell-1][i]$. This takes total preprocessing time and space $\mathcal{O}(n\ell) = \mathcal{O}(n\log_\sigma n)$ by Theorem 5. Given the range query $[a, b]$, we need to find the smallest $j \in [1, \ell-1]$ such that $\mathsf{FTR}[j][b] < a$. By Theorem 5, we find where the predecessor of a lies in $\mathsf{FTR}[1\mathinner{.\,.}\ell-1][b]$ in $\mathcal{O}(\log_w \ell)$ time, where w is the word size; this time cost is $\mathcal{O}(1)$ since $w = \Theta(\log n)$.

We finally retrieve a *witness* shortest absent word as follows. If there is no $j < \ell$ such that $\mathsf{FTR}[j][b] < a$, then we output any shortest absent word of length ℓ of T arbitrarily. If such a $j < \ell$ exists, by the definition of $\mathsf{FTR}[j]$, we output $T[\mathsf{FTR}[j][b]\mathinner{.\,.}\mathsf{FTR}[j][b]+j-1]$ if $\mathsf{FTR}[j][b] > 0$ or $T[k\mathinner{.\,.}k+j-1]$ if $\mathsf{FTR}[j][b] = 0$, where k is the index of PRE_j, where the minimum came from. Inspect the following illustrative example.

▶ **Example 12** (Querying). We construct array FTR for T from Example 9. For a given $[a, b]$ we look up column b, and find the topmost entry whose value is less than a. If all entries have values greater than or equal to a, we output any element from SAW_T arbitrarily.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T	a	b	a	a	b	a	a	a	b	b	a	b	b	b	a	a	a	b
FTR[1]	0	1	2	2	4	5	5	5	8	8	10	11	11	11	14	14	14	17
FTR[2]	0	0	0	0	0	0	0	0	0	5	7	7	7	7	7	11	11	13
FTR[3]	0	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4	4

If $[a, b] = [3, 14]$ then no entry in column $b = 14$ is less than $a = 3$, which means the length of the shortest absent word is 4; we output one from $\{\mathtt{aaaa}, \mathtt{abab}, \mathtt{baba}, \mathtt{bbbb}\}$ arbitrarily. If $[a, b] = [5, 14]$ then $\mathsf{FTR}[3][14] = 4 < 5$ so the length of a shortest absent word of $T[5\mathinner{.\,.}14]$ is 3; a shortest absent word is $T[\mathsf{FTR}[3][14]\mathinner{.\,.}\mathsf{FTR}[3][14] + 3 - 1] = T[4\mathinner{.\,.}6] = \mathtt{aba}$.

If $[a, b] = [7, 9]$, $\mathsf{FTR}[2][9] = 0 < 7$ so the length of a shortest absent word is 2; a shortest absent word is $T[k\mathinner{.\,.}k+j-1] = T[9\mathinner{.\,.}10] = \mathtt{bb}$ because $\mathsf{FTR}[2][9] = \min\{\mathsf{PRE}_2[9], \ldots, \mathsf{PRE}_2[|\mathsf{PRE}_2|]\} = \mathsf{PRE}_2[9] = 0$ tells us that the minimum in this range came from index $k = 9$.

We obtain the following result.

▶ **Theorem 13.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, we can construct a data structure of size $\mathcal{O}(n\log_\sigma n)$ in $\mathcal{O}(n \log_\sigma n)$ time, so that if query $[a, b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a\mathinner{.\,.}b]$ in $\mathcal{O}(1)$ time.*

4 $\mathcal{O}(n)$ Space and $\mathcal{O}(\log\log_\sigma n)$ Query Time

▶ **Definition 14** (Order-j Fragment). *Given a string T over an alphabet of size σ and an integer j, V is called an* order-j fragment *of T if and only if V is a fragment of T and $S_V(j) = \sigma^j$. V is further called a* minimal order-j fragment *of T if $S_U(j) < \sigma^j$ and $S_Z(j) < \sigma^j$ for $U = V[1\mathinner{.\,.}|V|-1]$ and $Z = V[2\mathinner{.\,.}|V|]$.*

In particular, minimal order-j fragments are pairwise not included in each other. The following fact follows directly.

▶ **Fact 15.** *Given a string T of length n over an alphabet of size σ and an integer j we have $\mathcal{O}(n)$ minimal order-j fragments. Moreover, an arbitrary fragment F of T has $S_F[j] = \sigma^j$ if and only if it contains at least one of these minimal fragments.*

For each $j \in [1, \log_\sigma n]$, we consider all minimal order-j fragments T, separately. We encode the minimal order-j fragments of T using two bit vectors SP_j and EP_j, standing for starting positions and ending positions. Inspect the following example.

▶ **Example 16.** We consider T from Example 9 and $j = 2$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
T	a	b	a	a	b	a	a	a	b	b	a	b	b	b	a	a	a	b			
APP_2	2	3	1	2	3	1	1	2	4	3	2	4	4	3	1	1	2	1	2	3	4
PRE_2	0	0	0	1	2	3	6	4	0	5	8	9	12	10	7	15	11	16	17	14	13
SP_2	0	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	0	0			
EP_2	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	1			

For instance, $\mathsf{SP}_2[13] = 1$ and $\mathsf{EP}_2[18] = 1$ denote the minimal order-2 fragment $V = T[13\mathinner{.\,.}18] = \texttt{bbaaab}$.

We construct a rank and select data structure on SP_j and EP_j, for all $j \in [1, \ell - 1]$ supporting $\mathcal{O}(1)$-time queries. The overall space is $\mathcal{O}(n)$ by Theorem 4 and Fact 7.

Let us now explain how this data structure enables fast computation of absent words of length j. Given a range $[a, b]$, by Fact 15, we only need to find whether $T[a\mathinner{.\,.}b]$ contains a minimal order-j fragment. We can do this in $\mathcal{O}(1)$ time using one rank and two select queries: $t = \mathsf{rank}_1(a-1, \mathsf{SP}_j) + 1$; and $\mathsf{select}_1(t, \mathsf{SP}_j)$ and $\mathsf{select}_1(t, \mathsf{EP}_j)$.

▶ **Example 17.** We consider T, SP_2 and EP_2 from Example 16. Let $[a, b] = [5, 14]$. We have $t = \mathsf{rank}_1(a-1, \mathsf{SP}_2)+1 = \mathsf{rank}_1(4, \mathsf{SP}_2)+1 = 1$, $\mathsf{select}_1(t, \mathsf{SP}_2) = \mathsf{select}_1(1, \mathsf{SP}_2) = 5 < b = 14$ and $\mathsf{select}_1(t, \mathsf{EP}_2) = \mathsf{select}_1(1, \mathsf{EP}_2) = 10 < b = 14$, which means $T[5, 14]$ contains a minimal order-2 fragment.

Let us now describe a time-efficient construction of SP_j and EP_j. We use arrays PRE_j and APP_j of T, which are constructible in $\mathcal{O}(n)$ time (see Section 3). Recall that $\mathsf{PRE}_j[i]$ stores the starting position of the rightmost occurrence of rank $\mathsf{APP}_j[i]$ in $\mathsf{APP}_j[1\mathinner{.\,.}i-1]$ (or 0 if such an occurrence does not exist). We apply Fact 8 as follows. We start with all bits of SP_j and EP_j unset. Then, for each $b \in [1, n]$ for which $\mathsf{PRE}_j[b-j+1] < \min\{\mathsf{PRE}_j[i] : i \in [b-j+2, |\mathsf{PRE}_j|]\} = a$, we set the bth bit of EP_j and the ath bit of SP_j. This can be done online in a right-to-left scan of PRE_j in $\mathcal{O}(n)$ time.

▶ **Example 18.** We consider T, SP_2 and EP_2 from Example 16. We start by setting $b = n = 18$ and scan PRE_2 from right to left: we have $a = 13$ because $\min\{\mathsf{PRE}_2[21] = 13 : i \in [18, 21]\} \geq a = 13$. This gives fragment $T[13\mathinner{.\,.}18]$. Then we set $b = n-1 = 17$ and have $a = 11$ because

$\min\{\mathsf{PRE}_2[21] = 13 : i \in [17,20]\} \geq a = 11$. This gives fragment $T[11\,..\,17]$. Then we set $b = n-2 = 16$ and have $a = 11$ because $\min\{\mathsf{PRE}_2[21] = 13 : i \in [16,19]\} \geq a = 11$. This gives fragment $T[11\,..\,16]$. At this point note that $T[11\,..\,17]$ contains $T[11\,..\,16]$, and so $T[11\,..\,17]$ is removed as it is non-minimal.

▶ **Lemma 19.** *SP_j and EP_j can be constructed in $\mathcal{O}(n)$ time.*

For all j, the construction time is $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ by Theorem 4, Lemma 19, and Fact 7. We obtain the following lemma.

▶ **Lemma 20.** *Given a string T of length n over an alphabet $\Sigma \subset \{1,2,\ldots,n^{\mathcal{O}(1)}\}$ of size σ, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n\log_\sigma n)$ time, so that if query $(j,[a,b])$ is given, we can check in $\mathcal{O}(1)$ time whether there is any string in Σ^j that does not occur in $T[a\,..\,b]$, and if so return such a string.*

We can now apply Lemma 20 using binary search on j to find the smallest length j such that $S_{T[a\,..\,b]}(j) < \sigma^j$. This results in $\mathcal{O}(\log \ell) = \mathcal{O}(\log\log_\sigma n)$ query time by Fact 7. It should now be clear that when we find the j corresponding to the length of a shortest absent word, we can output the length-j suffix of the leftmost minimal order-j fragment starting after a. Note that outputting this suffix is correct by the definition of minimal order-j fragments.

▶ **Example 21.** We consider T, SP_2 and EP_2 from Example 16. Let $[a,b] = [2,7]$. The length of a shortest absent word of $T[2\,..\,7]$ is 2. We output `bb`, which is the length-2 suffix of the leftmost minimal order-2 fragment $T[5\,..\,10] =$ `baaabb` starting after $a = 2$.

We obtain the following result.

▶ **Theorem 22.** *Given a string T of length n over an alphabet $\Sigma \subset \{1,2,\ldots,n^{\mathcal{O}(1)}\}$ of size σ, we can construct a data structure of size $\mathcal{O}(n)$ in $\mathcal{O}(n\log_\sigma n)$ time, so that if query $[a,b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a\,..\,b]$ in $\mathcal{O}(\log\log_\sigma n)$ time.*

5 Combinatorial Insights

A positive integer p is a *period* of a string S if $S[i] = S[i+p]$ for all $i \in [1, |S|-p]$. We refer to the smallest period as *the period* of the string. Let us state the periodicity lemma, one of the most elegant combinatorial results on strings.

▶ **Lemma 23** (Periodicity Lemma (weak version) [31]). *If a string S has periods p and q such that $p+q \leq |S|$, then $\mathsf{gcd}(p,q)$ is also a period of S.*

▶ **Lemma 24.** *If all strings in $\{WU : U \in \Sigma^k\}$ for $W \neq \varepsilon$ occur in some string S, then $|S| \geq |W| \cdot \sigma^k/4$.*

Proof. Let p be the period of W, and let $a \in \Sigma$ be such so that the period of Wa is also p. All strings WbZ for a letter $b \neq a$ and $Z \in \Sigma^{k-1}$ must occur in S. Let $A = \{WU : U \in \Sigma^k\} \setminus \{WaZ : Z \in \Sigma^{k-1}\}$, and note that it is of size $\sigma^k - \sigma^{k-1} \geq \sigma^k/2$. The following claim immediately implies the statement of the lemma.

▷ Claim. Let i and j be starting positions of occurrences of different strings $WU, WV \in A$ in S, respectively. Then, we have $|j-i| \geq |W|/2$.

Proof. Let us assume, without loss of generality, that $j > i$. Further, let us assume towards a contradiction that $j - i < |W|/2$. Then, $j - i$ is a period of W and $p + j - i \leq |W|$ since $p \leq j - i$. Therefore, due to the periodicity lemma (Lemma 23), $j - i$ must be divisible by the period p of W. Hence, U starts with the letter a and $WU \notin A$, a contradiction. ◁

This concludes the proof of this lemma. ◀

▶ **Lemma 25.** *If a shortest absent word of a string Y is of length λ, then the length of a shortest absent word of XY is in $[\lambda, \lambda + \max\{10, 4 + \log_\sigma(|X|/\lambda)\}]$.*

Proof. Let W and W' be shortest absent words of Y and XY, respectively. Further, let $d = |W'| - |W|$. In order to have $d > 0$, all strings WU for $U \in \Sigma^{d-1}$ must occur in XY, and hence in $X \cdot Y[1 \,..\, |WU| - 1]$, since none of them occurs in Y. Lemma 24 implies that $|X| + \lambda + d > \lambda \cdot \sigma^{d-1}/4$. Then, since $\lambda + d \leq 2\lambda d$ for any positive integers λ, d, we have $|X| > \lambda \cdot (\sigma^{d-1}/4 - 2d)$. Assuming that $d \geq 10$, and since $\sigma \geq 2$, we conclude that $|X| > \lambda \cdot \sigma^{d-1}/8$. Consequently, $\log_\sigma(8|X|/\lambda) + 1 > d$. Since $\log_\sigma 8 \leq 3$ we get the claimed bound. ◀

▶ **Lemma 26.** *If a shortest absent word of XY is of length m, a shortest absent word of Y is of length λ, and $|X| \leq m \cdot \tau$, for a positive integer $\tau \geq 16$, then $m - \lambda \leq 10 + 2\log_\sigma \tau$.*

Proof. From Lemma 25 we have $\lambda \in [m - \max\{10, 4 + \log_\sigma(|X|/\lambda)\}, m]$. If $\max\{10, 4 + \log_\sigma(|X|/\lambda)\} = 10$, then $m - \lambda \leq 10$ and we are done.

In the complementary case, since $|X| \leq m \cdot \tau$, we get the following:

$$\lambda \geq m - \log_\sigma(m \cdot \tau/\lambda) - 4 \iff \lambda \geq m + \log_\sigma \lambda - \log_\sigma m - \log_\sigma \tau - 4.$$

In particular, $\lambda \geq m - \log_\sigma m - \log_\sigma \tau - 4$.

From the above, if $m \leq \tau$, then $m - \lambda \leq 4 + 2\log_\sigma \tau$.

In what follows we assume that $m > \tau \geq 16$. Rearranging the original equation, and since $\log_\sigma(\cdot)$ is an increasing function and $\lambda \geq m - \log_\sigma m - \log_\sigma \tau - 4$, we have

$$\begin{aligned} m - \lambda \leq 4 + \log_\sigma(m \cdot \tau/\lambda) &\leq 4 + \log_\sigma\left(\frac{m}{m - \log_\sigma m - \log_\sigma \tau - 4}\right) + \log_\sigma \tau \\ &\leq 4 + \log_\sigma\left(\frac{m}{m - 2\log_\sigma m - 4}\right) + \log_\sigma \tau. \end{aligned}$$

Then, we have $m - 2\log_\sigma m - 4 \geq m/5$ since, for any $\sigma \geq 2$, $4x/5 - 2\log_\sigma x - 4$ is an increasing function on $[16, \infty)$ and positive for $x = 16$. Hence, $m - \lambda \leq 4 + \log_\sigma 5 + \log_\sigma \tau \leq 7 + \log_\sigma \tau$.

By combining the bounds on $m - \lambda$ we get the claimed bound. ◀

6 $\mathcal{O}(n \log\log_\sigma n)$ Space and $\mathcal{O}(1)$ Query Time

Recall that we denote by ℓ the length of a shortest absent word of T. We start by constructing the 2d array $\mathsf{FTR}[1 \,..\, \ell - 1][1 \,..\, n]$ from Section 3 in time $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$. Further recall that $\mathsf{FTR}[j][i]$ is the rightmost index of T such that $T[\mathsf{FTR}[j][i] \,..\, i]$ contains all strings of length j over Σ. Then, to answer a query $[a, b]$, it suffices to find the smallest j such that $\mathsf{FTR}[j][b] < a$. We do this by finding where the predecessor of a lies in $\mathsf{FTR}[1 \,..\, \ell - 1][b]$. To this end, we construct n fusion trees: one per $\mathsf{FTR}[1 \,..\, \ell - 1][i]$, resulting in a data structure of size $\Theta(n\ell) = \mathcal{O}(n \log_\sigma n)$ with $\mathcal{O}(1)$ query time.

The main idea in this section is to rather maintain a collection of lazy dynamic fusion trees going from position n to 1, and apply the combinatorial lemmas from Section 5 to answer the query. Using the lazy dynamic fusion trees, for a parameter τ, we will compute an interval of size $\Theta(\log_\sigma \tau)$ that contains the length of a shortest absent word of $T[a\,..\,b]$. Then we will perform binary search employing Lemma 20 to compute the desired length and output a shortest absent word.

6.1 Lazy FTR Arrays

With *lazy* we mean that instead of array FTR, we consider array

$$\mathsf{LFTR}[j][i] = \mathsf{FTR}[j][i + ((n-i) \pmod{\tau \cdot j})],$$

for an integer parameter $\tau \geq 16$. We will later set τ to be some function of n – the reader may think of it as a constant. Intuitively, for an integer k, we use the value $\mathsf{FTR}[j][n-\tau\cdot j\cdot k]$ for $\tau\cdot j$ positions, namely for $\mathsf{LFTR}[j][i]$, $i \in (n-\tau\cdot j\cdot(k-1), n-\tau\cdot j\cdot k]$. Overall, the number of values that we consider is

$$\sum_{j\in[1,\ell-1]} \frac{n}{\tau\cdot j} = \mathcal{O}\left(\frac{n\log\log_\sigma n}{\tau}\right).$$

Inspect Figure 2 in this regard.

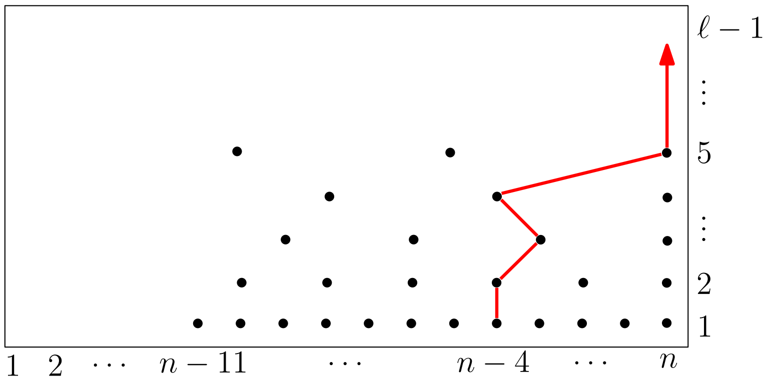

Figure 2 For simplicity, we have set $\tau = 1$. The black dots represent the entries stored in $\mathsf{LFTR}[1\,..\,5][n-11\,..\,n]$ and the red line represents $\mathsf{LFTR}[1\,..\,5][n-4]$.

We implement LFTR array using a collection of 1d arrays that occupy $\mathcal{O}((n/\tau)\cdot\log\log_\sigma n)$ space in total and allow $\mathcal{O}(1)$-time access to $\mathsf{LFTR}[j][i]$ for any j, i. Specifically, we store in array R_j, for all $j \in [1, \ell-1]$ and in decreasing order of i, the entries $\mathsf{FTR}[j][i]$ with $n-i \equiv 0 \pmod{\tau\cdot j}$. Then, we have $\mathsf{LFTR}[j][i] = R_j[1 + \lfloor (n-i)/(\tau\cdot j)\rfloor]$.

▶ **Fact 27.** *$\mathsf{LFTR}[j][i] \geq \mathsf{FTR}[j][i]$, for all i, j.*

Proof. It follows by the definition of LFTR and the fact that $\mathsf{FTR}[j][i]$ is monotonically non-decreasing for increasing i and fixed j. ◀

Note that the fact that $\mathsf{FTR}[1\,..\,\ell-1][i]$ is decreasing for all i allowed us to use predecessor queries in our previous solution. We prove an analogous, slightly weaker, statement for LFTR.

▶ **Lemma 28.** *Let $j_1, j_2 \in [1, \ell-1]$ such that $j_2 - j_1 > 10 + 2\log_\sigma \tau$, and suppose that $\tau \geq 16$. Then, for all i, $\mathsf{LFTR}[j_1][i] > \mathsf{LFTR}[j_2][i]$.*

Proof. Let $X_1 = T[\mathsf{LFTR}[j_1][i]+1\,..\,i]$. Then, X_1Y_1 has a shortest absent word of length j_1, for some Y_1 with $|Y_1| \leq \tau \cdot j_1$.

Let $X_2 = T[\mathsf{LFTR}[j_2][i]+1\,..\,i]$. Then, X_2Y_2 has a shortest absent word of length j_2, for some Y_2 with $|Y_2| \leq \tau \cdot j_2$.

Note that $||Y_2| - |Y_1|| \leq \tau \cdot j_2$.

Suppose, towards a contradiction, that $\mathsf{LFTR}[j_1][i] \leq \mathsf{LFTR}[j_2][i]$, and hence $|X_1| \geq |X_2|$. Then, we must have $|Y_2| > |Y_1|$ as otherwise X_2Y_2 would be a substring of X_1Y_1 and its shortest absent word cannot be longer than the one of X_1Y_1. Let $Y_2 = Y_1V$, with $|V| \leq \tau \cdot j_2$.

Then, we have that a shortest absent word of X_1Y_2 is of length at least j_2, since X_2Y_2 is a suffix of X_1Y_2. By Lemma 26 applied to $X_1Y_2 = X_1Y_1V$ and X_1Y_1, we have $j_2 - j_1 \leq 10 + 2\log_\sigma \tau$, a contradiction. ◀

In particular, Lemma 28 tells us that in column i of LFTR, we cannot have too many values that are equal. More formally, for each j, we have $\mathcal{O}(\log_\sigma \tau)$ indices $j' \neq j$ such that $\mathsf{LFTR}[j'][i] = \mathsf{LFTR}[j][i]$. Our goal is to have, for every position i, a snapshot of one of our dynamic fusion trees to contain as keys the entries of $\mathsf{LFTR}[1\,..\,\ell-1][i]$. The satellite information (value) of key $\mathsf{LFTR}[j][i]$ is a bit vector of size $\ell - 1$ bits. For each key, we maintain the corresponding lengths j in the bit vector. Whenever a key is returned, we can also return the largest of the lengths j stored in the bit vector: it corresponds to the highest set bit. In the next subsection we show the following result.

▶ **Lemma 29.** *We can preprocess LFTR in $\mathcal{O}((n/\tau) \cdot \log\log_\sigma n)$ time and space to answer predecessor and successor queries over LFTR$[1\,..\,\ell-1][i]$, for any $i \in [1, n]$, in $\mathcal{O}(1)$ time.*

We denote by $\mathsf{top}(a, b)$ the largest j such that $\mathsf{LFTR}[j][b]$ is equal to the successor of a in $\mathsf{LFTR}[1\,..\,\ell-1][b]$.

▶ **Lemma 30.** *Given LFTR, after $\mathcal{O}((n/\tau) \cdot \log\log_\sigma n)$ time and space preprocessing, we can answer top(a, b) queries in $\mathcal{O}(1)$ time.*

Proof. At preprocessing, construct the data structure underlying Lemma 29. Upon a query $\mathsf{top}(a, b)$, answer a successor query for a in $\mathsf{LFTR}[1\,..\,\ell-1][b]$ using this data structure. For the corresponding bit vector, we find the highest set bit in $\mathcal{O}(1)$ time, thus retrieving $\mathsf{top}(a, b)$. ◀

Our data structure mainly relies on what we show next using Lemma 26: the sought answer is "close" to $\mathsf{top}(a, b)$. Let us denote the length of a shortest absent word of $T[c\,..\,b]$ by $\ell_{[c,b]}$ and the length of a shortest absent word of $T[a\,..\,b]$ by $\ell_{[a,b]}$.

By the definition of $\mathsf{top}(a, b)$, we have that for some $c = \mathsf{LFTR}[\mathsf{top}(a, b)][b] \geq a$ and a prefix X of $T[b+1\,..\,n]$ with $|X| \leq \mathsf{top}(a, b) \cdot \tau$, the length of a shortest absent word of $T[c\,..\,b]X$ is $\mathsf{top}(a, b)+1$. By Lemma 26, $\mathsf{top}(a, b) - \ell_{[c,b]} \leq 10 + 2\log_\sigma \tau$. Thus $\ell_{[a,b]} \geq \ell_{[c,b]} \geq \mathsf{top}(a, b) - 10 - 2\log_\sigma \tau$.

In addition, we have $\mathsf{LFTR}[\mathsf{top}(a, b)][b] \geq a > \mathsf{LFTR}[j][b]$, for all $j > \mathsf{top}(a, b)+10+2\log_\sigma \tau$, by Lemma 28 and the definition of $\mathsf{top}(a, b)$. Hence, $\ell_{[a,b]} \leq \mathsf{top}(a, b)+11+2\log_\sigma \tau$ by Fact 27.

Thus, the sought answer $\ell_{[a,b]}$ is in $[\mathsf{top}(a, b) - 10 - 2\log_\sigma \tau, \mathsf{top}(a, b) + 11 + 2\log_\sigma \tau]$. We employ Lemma 20 to perform binary search over this interval in $\mathcal{O}(\log\log_\sigma \tau)$ time – after an $\mathcal{O}(n)$-time preprocessing. Recall that Lemma 20 also gives us a witness shortest absent word.

We thus arrive at the main result of this paper, by setting $\tau = 15 + k$, for any positive integer $k \in [1, \log\log_\sigma n]$.

▶ **Theorem 1.** *Given a string T of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ of size σ, for any positive integer $k \in [1, \log\log_\sigma n]$, we can construct a data structure of size $\mathcal{O}((n/k) \cdot \log\log_\sigma n)$, in $\mathcal{O}(n \log_\sigma n)$ time, so that if query $[a, b]$ is given, we can compute a shortest string over Σ that does not occur in $T[a \mathinner{.\,.} b]$ in $\mathcal{O}(\log\log_\sigma k)$ time.*

In particular, we get the following tradeoffs:

- an $\mathcal{O}(n \log\log_\sigma n)$-size data structure with $\mathcal{O}(1)$ query time (for $k = 1$);
- an $\mathcal{O}(n)$-size data structure with $\mathcal{O}(\log\log_\sigma \log\log_\sigma n)$ query time (for $k = \lfloor \log\log_\sigma n \rfloor$).

6.2 Partially Persistent Fusion Trees (using Fusion Trees)

We now describe the construction of the fusion trees over the LFTR array, which underlies Lemma 29. We provide the description for answering predecessor queries but it can be trivially adapted for successor queries. We will make our fusion trees "static partially persistent": for each position $i \in [1, n]$, we will be able to answer predecessor queries in the version of the fusion tree corresponding to $\mathsf{LFTR}[1 \mathinner{.\,.} \ell - 1][i]$.

Recall that we work in the word RAM model. For implementing partially persistent fusion trees, we view each memory cell as a collection of pairs of values and timestamps; a timestamp indicates when the respective value was written in the cell. (This is a standard persistence trick, see e.g. [55].) For each cell, we want to construct a predecessor data structure over the timestamps to simulate these operations. The key idea is that, in each such cell, we would like to keep the number of updates small so as to employ fusion trees for implementing it as a predecessor data structure. Let us stress that the latter fusion trees should not be confused with the partially persistent fusion trees we construct over the LFTR array for our problem.

We now process T from right to left to construct the collection of partially persistent fusion trees. For each position of T, we perform the updates as per the LFTR array. Specifically, for position n, we initialise the partially persistent fusion tree with keys $\mathsf{LFTR}[1 \mathinner{.\,.} \ell - 1][n]$. Then, for position i from $n - 1$ to 1, for all $j \in [1, \ell - 1]$, such that $(n - i) \pmod{\tau \cdot j} = 0$, we remove key $\mathsf{LFTR}[j][i + \tau \cdot j]$ and insert key $\mathsf{LFTR}[j][i]$. However, after processing every $\tau \cdot \log n / \log\log n$ positions of T, and hence $\sum_{j \in [1, \ell - 1]} (\tau \log n / \log\log n)/(\tau \cdot j) = \Theta(\log n)$ updates have been performed, we create a completely new instance of a partially persistent fusion tree. We initialise this new instance with the LFTR values of the currently unprocessed position of T. Let us note that the $\mathcal{O}(\log_\sigma n)$ time cost for reinitialisation amortises, because we can charge it to the $\Theta(\log n)$ $\mathcal{O}(1)$-time updates we have previously performed. For each position i of T, we store the timestamp $t(i)$ at which its processing ended and a pointer to the partially persistent fusion tree of the collection corresponding to it.

Upon a query $[a, b]$, we wish to find the smallest j such that $\mathsf{LFTR}[j][b] < a$. We thus need to find where the predecessor of a lies in $\mathsf{LFTR}[1 \mathinner{.\,.} \ell - 1][b]$. We retrieve the partially persistent fusion tree and ask the query, using timestamp $t(b)$. Note that each memory access performed by the query requires $\mathcal{O}(1)$ time, as it translates to a predecessor query in a fusion tree with $\mathcal{O}(\log n)$ keys; and there are $\mathcal{O}(1)$ such accesses because this is a (partially persistent) fusion tree with $\mathcal{O}(\log_\sigma n)$ keys. The query thus takes $\mathcal{O}(1)$ time.

6.3 Weighted Ancestor Data Structure for Shallow Trees

A tree is a *weighted tree* if it is a rooted tree with an integer weight on each node v, denoted by $w(v)$, such that the weight of the root is zero and $w(u) < w(v)$ if u is the parent of v. We say that a node v is a *weighted ancestor at depth δ* of a node u if v is the highest ancestor of u with weight of at least δ. The problem of constructing a data structure to answer weighted ancestor queries was introduced by Farach and Muthukrishnan in [26].

After $\mathcal{O}(n)$-time preprocessing, weighted ancestor queries for nodes of a weighted tree $\mathcal{T}$ of size n with integer weights from a universe $[1\mathinner{.\,.}U]$ can be answered in $\mathcal{O}(\log\log U)$ time [26, 5]. Later, it was shown that a dynamic variant of the weighted ancestors problem admits a solution with the same time bounds as those for dynamic predecessor structures [47, 36]. Further, Kopelowitz et al. [46] introduced another $\mathcal{O}(n)$-size data structure that achieves faster query time in many special cases. For the offline version, Kociumaka et al. [43] showed how to answer a batch of q weighted ancestor queries in the optimal $\mathcal{O}(n+q)$ time.

The weighted ancestors problem has numerous applications if the input tree is a suffix tree of some string; see [36] for a nice exposition of these applications. In this context, the weighted ancestors problem translates to preprocessing the suffix tree of a string $T[1\mathinner{.\,.}n]$, so that, given i and j, we can retrieve the implicit or explicit node corresponding to substring $T[i\mathinner{.\,.}j]$. Observe that, since the weighted ancestor is a generalisation of the predecessor problem, it cannot admit better bounds. Nevertheless, the problem on suffix trees is a special case of the general problem. This led to the challenge of solving the problem on suffix trees in $\mathcal{O}(n)$ preprocessing time and $\mathcal{O}(1)$ query time [26]. Gawrychowski et al. [36] partly settled this question by presenting an $\mathcal{O}(n)$-size data structure with $\mathcal{O}(1)$ query time; the construction time, however, is superlinear in n. Very recently, Belazzougui et al. [11] have settled this question by presenting an $\mathcal{O}(n)$-size data structure for weighted ancestors in suffix trees with $\mathcal{O}(1)$ query time and an $\mathcal{O}(n)$-time construction algorithm.

Given a tree $\mathcal{T}$ of size n and depth d, we can do the following trick to reduce the weighted ancestors problem to $\mathcal{O}(n/d)$ instances on trees of both size and depth $\mathcal{O}(d)$: Cut along every dth root-to-leaf path in $\mathcal{T}$, and duplicate its nodes and edges. Then, we can apply the result of Kopelowitz and Lewenstein [47, 36] to each of the $\mathcal{O}(n/d)$ smaller trees. In the specific case of $d = \log^{\mathcal{O}(1)} n$, this gives an $\mathcal{O}(n)$-size data structure that answers weighted ancestor queries in $\mathcal{O}(1)$ time. By applying the machinery we have developed in the previous subsection, we achieve the same result for this special case in an alternative way. For each of the $\mathcal{O}(n/d)$ trees of size and depth $\mathcal{O}(d)$, we perform a depth-first traversal, maintaining a partially persistent fusion tree that when visiting node v stores the weights of all (weak) ancestors of v. We obtain the following corollary for shallow trees.

▶ **Corollary 31.** *Let $\mathcal{T}$ be a weighted tree of size n and depth $d = \log^{\mathcal{O}(1)} n$, with weights polynomial in n. We can preprocess $\mathcal{T}$ in $\mathcal{O}(n)$ time so that weighted ancestor queries on $\mathcal{T}$ can be answered in $\mathcal{O}(1)$ time.*

References

1 Paniz Abedin, Arnab Ganguly, Wing-Kai Hon, Yakov Nekrich, Kunihiko Sadakane, Rahul Shah, and Sharma V. Thankachan. A linear-space data structure for Range-LCP queries in poly-logarithmic time. In *Computing and Combinatorics - 24th International Conference, COCOON 2018*, pages 615–625, 2018. doi:10.1007/978-3-319-94776-1_51.

2 Paniz Abedin, Arnab Ganguly, Solon P. Pissis, and Sharma V. Thankachan. Efficient data structures for range shortest unique substring queries. *Algorithms*, 13(11):276, 2020. doi:10.3390/a13110276.

3 Amihood Amir, Alberto Apostolico, Gad M. Landau, Avivit Levy, Moshe Lewenstein, and Ely Porat. Range LCP. *J. Comput. Syst. Sci.*, 80(7):1245–1253, 2014. doi:10.1016/j.jcss.2014.02.010.

4 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/s00453-020-00744-0.

5 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

6 Lorraine A. K. Ayad, Golnaz Badkobeh, Gabriele Fici, Alice Héliou, and Solon P. Pissis. Constructing antidictionaries in output-sensitive space. In *Data Compression Conference, DCC 2019*, pages 538–547. IEEE, 2019. doi:10.1109/DCC.2019.00062.

7 Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, Ignat I. Kolesnichenko, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theor. Comput. Sci.*, 638:112–121, 2016. doi:10.1016/j.tcs.2015.08.023.

8 Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 572–591. SIAM, 2015. doi:10.1137/1.9781611973730.39.

9 Carl Barton, Alice Héliou, Laurent Mouchard, and Solon P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinform.*, 15:388, 2014. doi:10.1186/s12859-014-0388-9.

10 Carl Barton, Alice Héliou, Laurent Mouchard, and Solon P. Pissis. Parallelising the computation of minimal absent words. In *Parallel Processing and Applied Mathematics - 11th International Conference, PPAM 2015. Revised Selected Papers, Part II*, volume 9574 of *Lecture Notes in Computer Science*, pages 243–253. Springer, 2015. doi:10.1007/978-3-319-32152-3_23.

11 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021*, volume 191 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.8.

12 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

13 Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993. doi:10.1137/0222017.

14 Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 607–626. SIAM, 2020. doi:10.1137/1.9781611975994.37.

15 Supaporn Chairungsee and Maxime Crochemore. Using minimal absent words to build phylogeny. *Theor. Comput. Sci.*, 450:109–116, 2012. doi:10.1016/j.tcs.2012.04.031.

16 Panagiotis Charalampopoulos, Maxime Crochemore, Gabriele Fici, Robert Mercaş, and Solon P. Pissis. Alignment-free sequence comparison using absent words. *Inf. Comput.*, 262:57–68, 2018. doi:10.1016/j.ic.2018.06.002.

17 Panagiotis Charalampopoulos, Maxime Crochemore, and Solon P. Pissis. On extended special factors of a word. In *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018*, volume 11147 of *Lecture Notes in Computer Science*, pages 131–138. Springer, 2018. doi:10.1007/978-3-030-00479-8_11.

18 Panagiotis Charalampopoulos, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. An almost optimal edit distance oracle. *CoRR*, abs/2103.03294, 2021. arXiv:2103.03294.

19 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, volume 161 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.8.

20 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. In *30th International Symposium on Algorithms and Computation, ISAAC 2019*, volume 149 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ISAAC.2019.22.

21 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*, pages 978–989. IEEE, 2020. doi:10.1109/FOCS46700.2020.00095.

22 Maxime Crochemore, Alice Héliou, Gregory Kucherov, Laurent Mouchard, Solon P. Pissis, and Yann Ramusat. Absent words in a sliding window with applications. *Inf. Comput.*, 270, 2020. doi:10.1016/j.ic.2019.104461.

23 Maxime Crochemore, Filippo Mignosi, and Antonio Restivo. Automata and forbidden words. *Inf. Process. Lett.*, 67(3):111–117, 1998. doi:10.1016/S0020-0190(98)00104-5.

24 Maxime Crochemore, Filippo Mignosi, Antonio Restivo, and Sergio Salemi. Data compression using antidictionaries. *Proceedings of the IEEE*, 88(11):1756–1768, 2000. doi:10.1109/5.892711.

25 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

26 Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 1996*, volume 1075 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1996. doi:10.1007/3-540-61258-0_11.

27 Sébastien Ferenczi. Complexity of sequences and dynamical systems. *Discret. Math.*, 206(1-3):145–154, 1999. doi:10.1016/S0012-365X(98)00400-2.

28 Gabriele Fici and Pawel Gawrychowski. Minimal absent words in rooted and unrooted trees. In *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019*, volume 11811 of *Lecture Notes in Computer Science*, pages 152–161. Springer, 2019. doi:10.1007/978-3-030-32686-9_11.

29 Gabriele Fici, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Word assembly through minimal forbidden words. *Theor. Comput. Sci.*, 359(1-3):214–230, 2006. doi:10.1016/j.tcs.2006.03.006.

30 Gabriele Fici, Antonio Restivo, and Laura Rizzo. Minimal forbidden factors of circular words. *Theor. Comput. Sci.*, 792:144–153, 2019. doi:10.1016/j.tcs.2018.05.037.

31 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. URL: http://www.jstor.org/stable/2034009.

32 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

33 Yuta Fujishige, Yuki Tsujimaru, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing DAWGs and minimal absent words in linear time for integer alphabets. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016*, volume 58 of *LIPIcs*, pages 38:1–38:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.MFCS.2016.38.

34 Arnab Ganguly, Manish Patil, Rahul Shah, and Sharma V. Thankachan. A linear space data structure for range LCP queries. *Fundam. Inform.*, 163(3):245–251, 2018. doi:10.3233/FI-2018-1741.

35 Sara P. Garcia, Armando J. Pinho, João M. O. S. Rodrigues, Carlos A. C. Bastos, and Paulo J. S. G. Ferreira. Minimal absent words in prokaryotic and eukaryotic genomes. *PLoS ONE*, 6, 2011. doi:10.1371/journal.pone.0016065.

36 Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In *Algorithms - ESA 2014 - 22th Annual European Symposium*, volume 8737 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

37 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

38 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, FOCS 1989*, pages 549–554. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63533.

39 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theor. Comput. Sci.*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.

40 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.

41 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, pages 28:1–28:12, 2016. doi:10.4230/LIPIcs.CPM.2016.28.

42 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts.* PhD thesis, University of Warsaw, 2018. URL: https://mimuw.edu.pl/~kociumaka/files/phd.pdf.

43 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear-time algorithm for seeds computation. *ACM Trans. Algorithms*, 16(2):27:1–27:23, 2020. doi:10.1145/3386369.

44 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Efficient data structures for the factor periodicity problem. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012*, volume 7608 of *Lecture Notes in Computer Science*, pages 284–294, 2012. doi:10.1007/978-3-642-34109-0_30.

45 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

46 Tsvi Kopelowitz, Gregory Kucherov, Yakov Nekrich, and Tatiana Starikovskaya. Cross-document pattern matching. *J. Discrete Algorithms*, 24:40–47, 2014. doi:10.1016/j.jda.2013.05.002.

47 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007*, pages 565–574. SIAM, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283444.

48 Gad M. Landau and Uzi Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

49 Kotaro Matsuda, Kunihiko Sadakane, Tatiana Starikovskaya, and Masakazu Tateshita. Compressed orthogonal search on suffix arrays with applications to range LCP. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, pages 23:1–23:13, 2020. doi:10.4230/LIPIcs.CPM.2020.23.

50 Takuya Mieno, Yuki Kuhara, Tooru Akagi, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Minimal unique substrings and minimal absent words in a sliding window. In *46th SOFSEM*, volume 12011 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 2020. doi:10.1007/978-3-030-38919-2_13.

51 Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Words and forbidden factors. *Theor. Comput. Sci.*, 273(1-2):99–117, 2002. doi:10.1016/S0304-3975(00)00436-9.

52 J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Text indexing and searching in sublinear time. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, volume 161 of *LIPIcs*, pages 24:1–24:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.24.

53 Gonzalo Navarro. *Compact Data Structures - A Practical Approach.* Cambridge University Press, 2016. URL: http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB.

54 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*, pages 72–77. IEEE, 2010. doi:10.1109/ISITA.2010.5649621.

55 Mihai Patrascu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011. doi:10.1137/09075336X.

56 Diogo Pratas and Jorge M Silva. Persistent minimal sequences of SARS-CoV-2. *Bioinformatics*, July 2020. btaa686. doi:10.1093/bioinformatics/btaa686.

57 Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014*, pages 166–175. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.26.

58 Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. doi:10.1007/s00453-012-9618-6.

59 Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017*, volume 10508 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.

60 Raquel M. Silva, Diogo Pratas, Luísa Castro, Armando J. Pinho, and Paulo J. S. G. Ferreira. Three minimal sequences found in Ebola virus genomes and absent from human DNA. *Bioinform.*, 31(15):2421–2425, 2015. doi:10.1093/bioinformatics/btv189.

61 Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-space LCE data structure with constant-time queries. In *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017*, volume 83 of *LIPIcs*, pages 10:1–10:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.MFCS.2017.10.

62 Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Math. Comput. Sci.*, 1(4):571–603, 2008. doi:10.1007/s11786-007-0033-3.

63 Andrew C. Yao. Space-time tradeoff for answering range queries (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC 1982, pages 128–136. ACM, 1982. doi:10.1145/800070.802185.