Internal Shortest Absent Word Queries in Constant Time and Linear Space

Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, Solon P Pissis

Internal Shortest Absent Word Queries in Constant Time and Linear Space[⋆]

Golnaz Badkobeh[a], Panagiotis Charalampopoulos[1b], Dmitry Kosolobov[2c], Solon P. Pissis[d,e]

[a] *Department of Computing, Goldsmiths University of London, UK*
[b] *Efi Arazi School of Computer Science, The Interdisciplinary Center Herzliya, Israel*
[c] *Ural Federal University, Ekaterinburg, Russia*
[d] *CWI, Amsterdam, The Netherlands*
[e] *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. We present an $\mathcal{O}(n)$-space data structure that answers such queries in constant time and can be constructed in $\mathcal{O}(n \log_\sigma n)$ time.

Keywords: string algorithms, internal queries, shortest absent word, bit parallelism

1. Introduction

Range queries are a classic data structure topic [59, 12, 11]. 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.

Internal queries on a string have 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 its occurrence 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 right, but also have numerous applications in the design of algorithms and (more sophisticated) data structures. Because of these numerous applications, we usually place particular emphasis on the construction time—other than on the tradeoff between space and query time, which is the main focus in the classic data structure literature.

In data structures on strings it is typically assumed that the input alphabet is integer and polynomially bounded, i.e., it is a subset of $\{1, 2, \ldots, n^{\mathcal{O}(1)}\}$ where n is the length of the input string T. One of the most widely-used internal queries 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 [45] 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 when the input alphabet is $\{1, 2, \ldots, \sigma\}$ and

[1]Supported by the Israel Science Foundation grant 592/17.
[2]Supported by the Ministry of Science and Higher Education of the Russian Federation (Ural Mathematical Center project No. 075-02-2021-1387)
[⋆]The present paper is an extended and improved version of an earlier text that appeared in the 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021 [8].

Email addresses: g.badkobeh@gold.ac.uk (Golnaz Badkobeh), panagiotis.charalampopoulos@post.idc.ac.il (Panagiotis Charalampopoulos), dkosolobov@mail.ru (Dmitry Kosolobov), solon.pissis@cwi.nl (Solon P. Pissis)

the string is packed into $\mathcal{O}(n/\log_\sigma n)$ machine words. A sequence of works [57, 49, 13] 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. [44] 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 derandomized in [42]. In fact, Kociumaka et al. [44], 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 [43]), 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 [58, 18], approximate pattern matching [21], dictionary matching [20, 19], longest common substring [4], counting palindromes [55], range longest common prefix [3, 1, 46, 34], the computation of the lexicographically minimal or maximal suffix, and minimal rotation [6, 41], as well as of the lexicographically kth suffix [7]. 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 $T[i\,..\,j]$. 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.

Theorem 1. *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 in $\mathcal{O}(n \log_\sigma n)$ time a data structure of size $\mathcal{O}(n)$ that, for any given query $[a, b]$, can compute in $\mathcal{O}(1)$ time a shortest string over Σ that does not occur in $T[a\,..\,b]$.*

In an earlier conference version of the present paper [8], we have obtained a weaker result: a data structure of size $\mathcal{O}((n/k) \cdot \log\log_\sigma n)$ that can answer queries in $\mathcal{O}(\log\log_\sigma k)$ time, where k is a user-defined parameter from $[1, \log\log_\sigma n]$. The improved data structure presented in this manuscript combines ideas from the conference version and the utilization of succinct fusion trees introduced by Grossi et al. [36].

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 $T[a\,..\,b]$; note that this length is at most $\log_\sigma n$. Let us also recall 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 starting position of the shortest suffix of $T[1\,..\,i]$ 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 $T[a\,..\,b]$ reduces to computing the predecessor of a among the starting positions we have precomputed for position b. By maintaining these

$\mathcal{O}(\log_\sigma n)$ starting 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 optimize the first approach by utilizing succinct fusion trees to store the sets of size $\mathcal{O}(\log_\sigma n)$ associated with positions of T, thus reducing the space on top of the sets to $\mathcal{O}(n \log_\sigma \log n)$. Instead of storing the $\mathcal{O}(\log_\sigma n)$-size sets explicitly, we compute their elements on demand using $\mathcal{O}(\log_\sigma n)$ select data structures, each occupying $\mathcal{O}(n)$ bits. This leads to an $\mathcal{O}(n \log_\sigma \log n)$-space solution. In order to optimize it further, 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 append $\Omega(\sigma^{d-1} \cdot \lambda)$ 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 append the de Bruijn sequence of order d over Σ to X in order to achieve the desired result.) This observation allows us to lower the memory consumption by truncating all succinct fusion trees at positions that are not multiples of $\log \log n$, by building them only for their first $\mathcal{O}(\log n / \log \log n)$ entries. The total space thus reduces to $\mathcal{O}(n)$ words. A query for the length of a shortest absent word of $T[a \mathinner{.\,.} b]$ is performed by first checking whether the answer is at most $\log n / \log \log n$, which is done using the (truncated) fusion tree stored at b, and, if not, a query on $T[a \mathinner{.\,.} b']$ is performed, where b' is the closest multiple of $\log \log n$ after b. It can be shown using the combinatorial observation that the answer for $T[a \mathinner{.\,.} b]$ is within an $\mathcal{O}(1)$-length range of the answer for $T[a \mathinner{.\,.} b']$, and it is computed by the data structure from the second approach.

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 [56, 53, 28, 35, 14, 51, 24] and their theory is well developed [48, 27, 29], also from an algorithmic and data structure point of view [47, 22, 9, 17, 16, 5, 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 [48].

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. Sections 3–5 have essentially already appeared in the conference version [8] of our paper; the main difference and novelty lie in Section 6. We conclude with open problems in Section 7.

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. A string 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 1. *Let T be a string of length n over an alphabet $\Sigma \subset \{1, 2, \ldots, n^{\mathcal{O}(1)}\}$. 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 2 ([11, 31])**.** *Let A be an array of n integers. A data structure of size $2n + o(n)$ bits that supports RMQs on A in $\mathcal{O}(1)$ time without the need to store and access A itself can be constructed in $\mathcal{O}(n)$ 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 3 ([38, 50])**.** *Let H be a bit vector of n bits. A data structure of $o(n)$ additional bits that supports rank and select queries on H in $\mathcal{O}(1)$ time can be constructed in $\mathcal{O}(n)$ 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 [52] presented a dynamic version of fusion trees, which, in particular, yields an efficient construction of this data structure.

Theorem 4 ([32, 52])**.** *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.*

We also use a succinct version of the (static) fusion tree that utilizes only $\mathcal{O}(n \log w)$ *bits* on top of a read-only array Y of length n (in contrast, the fusion tree from Theorem 4 uses $\mathcal{O}(nw)$ *bits*). In this data structure there is no need to store the array Y explicitly. Instead, Y can be "emulated" by computing its elements on demand in $\mathcal{O}(1)$ time. Albeit it is not explicitly stated in [36, 15], it follows from their construction that the succinct version can be constructed from a (usual) fusion tree in linear time.

Theorem 5 ([36, 15])**.** *Let Y be a read-only array of at most n w-bit integers and $n \le w^{\mathcal{O}(1)}$. A data structure of size $\mathcal{O}(n \log w)$ bits can be constructed in $\mathcal{O}(n \log_w n)$ time supporting predecessor queries on the elements of Y in $\mathcal{O}(\log_w n)$ time, provided that a table computable in $o(2^w)$ time and independent of the array has been precomputed.*

Note that if we build multiple predecessor queries for sets of w-bit integers using the above theorem, they can all share a unique table computable in $o(2^w)$ time.

If $|U| = \mathcal{O}(n)$, then, after an $\mathcal{O}(n)$-time preprocessing, we can answer predecessor queries over the integer universe U in $\mathcal{O}(1)$ time as follows. 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 3. 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 $T[a \mathinner{.\,.} b]$ is $T[i \mathinner{.\,.} j]\alpha$; or simply a range $[i, j] \subseteq [1, n]$ such that the shortest string in Σ^* with no occurrence in $T[a \mathinner{.\,.} b]$ is $T[i \mathinner{.\,.} j]$.

Example 1. Given the string $T =$ abaabaaabbabbbaaab and the range $[a, b] = [8, 14]$ (shown in red), the only shortest absent word of $T[8 \mathinner{.\,.} 14]$ is $T[i \mathinner{.\,.} j] = T[7 \mathinner{.\,.} 8] =$ 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 [26, 54]. Note that $S_T(j) \le n$, for all j. We have the following simple fact.

Fact 6. *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 1, 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 6, $\ell \le \lfloor \log_\sigma n \rfloor$. 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 $T[i \mathinner{.\,.} i+j-1]$ 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 6.

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 $T[i \mathinner{.\,.} i+j-1]$ in Σ^j. Then, given a range $[a, b]$, in order to check if there is an absent word of length j in $T[a \mathinner{.\,.} b]$ we only need to compute the number of distinct elements in $\mathsf{RNK}_j[a \mathinner{.\,.} b-j+1]$. 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 $A[a \mathinner{.\,.} b]$ 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 [11] over a slight modification of RNK_j. For each j, we have an auxiliary procedure checking whether all strings from Σ^j occur in $T[a \mathinner{.\,.} b]$ 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 6, we have that $|\mathsf{APP}_j| \le 2n$. 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 $\mathsf{APP}_j[1 \mathinner{.\,.} i-1]$ (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 $(T[i \mathinner{.\,.} i+j-1], i)$, for all i, using the suffix tree of T to assign ranks for $T[i \mathinner{.\,.} i+j-1]$ and then radix sort to sort the list of pairs.

We now rely on the following fact.

Fact 7. $S_{T[a \mathinner{.\,.} b]}(j) = \sigma^j$ *if and only if* $\min\{\mathsf{PRE}_j[i] : i \in [b-j+2, |\mathsf{PRE}_j|]\} \ge a$.

Proof. If the smallest element in $\mathsf{PRE}_j[b-j+2 \mathinner{.\,.} |\mathsf{PRE}_j|]$, say $\mathsf{PRE}_j[k]$, is such that $\mathsf{PRE}_j[k] \ge a$, then all ranks of elements in Σ^j occur in $\mathsf{APP}_j[a \mathinner{.\,.} b-j+1]$. 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), and thus all must have a representative occurrence after $b-j+2$. Inspect Figure 1 for an illustration. (The opposite direction is analogous.) □

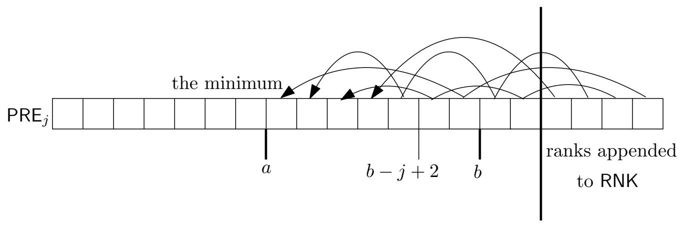

Figure 1: Illustration of the setting in Fact 7.

The following two examples illustrate the construction of arrays RNK_j, APP_j, and PRE_j as well as Fact 7.

Example 2 (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 depicted in Table 1. For instance, $\mathsf{RNK}_2[15] = \mathsf{APP}_2[15] = 1$ denotes that the

Table 1: Arrays RNK_j, APP_j, and PRE_j in Example 2.

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

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 3 (Fact 7)**.** Let $[a, b] = [7, 11]$ and $j = 2$ (see Example 2). 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 7, we construct, in $\mathcal{O}(n)$ time, an $\mathcal{O}(n)$-space, $\mathcal{O}(1)$-query-time RMQ data structure over PRE_j; see Theorem 2. This results in $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ preprocessing time and space over 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 2. *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 $\mathsf{FTR}[1\,..\,\ell-1][1\,..\,n]$ 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 $T[\mathsf{FTR}[j][i]\,..\,i]$ contains all strings of length j over Σ if such an index exists and 0 otherwise.

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 $\mathsf{PRE}_j[i-j+2\,..\,|\mathsf{PRE}_j|]$. We then construct n fusion trees, one for every collection of $\ell - 1$ integers in $\mathsf{FTR}[1\,..\,\ell-1][i]$. This takes total preprocessing time and space $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ by Theorem 4. 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 4, we find where the predecessor of a lies in $\mathsf{FTR}[1\,..\,\ell-1][b]$ 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 $\mathsf{FTR}[j][b]$, we output $T[\mathsf{FTR}[j][b]\,..\,\mathsf{FTR}[j][b]+j-1]$ if $\mathsf{FTR}[j][b] > 0$ or $T[k\,..\,k+j-1]$ 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 4 (Querying)**.** We construct array FTR for T from Example 2. 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
$\mathsf{FTR}[1]$	0	1	2	2	4	5	5	5	8	8	10	11	11	11	14	14	14	17
$\mathsf{FTR}[2]$	0	0	0	0	0	0	0	0	0	5	7	7	7	7	7	11	11	13
$\mathsf{FTR}[3]$	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 $T[5\,..\,14]$ is 3; a shortest absent word is $T[\mathsf{FTR}[3][14]\,..\,\mathsf{FTR}[3][14] + 3 - 1] = T[4\,..\,6] = \mathtt{aba}$.

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 $T[k\,..\,k + j - 1] = T[9\,..\,10] = \mathtt{bb}$ 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 proposition.

Proposition 3. *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 $\mathcal{O}(1)$ time.*

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

Definition 1 (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 $U = V[1\,..\,|V| - 1]$ and $Z = V[2\,..\,|V|]$.

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

Fact 8. *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 5. We consider T from Example 2 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 $V = T[13\,..\,18] = \mathtt{bbaaab}$.

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 3 and Fact 6.

Let us now explain how this data structure enables fast computation of absent words of length j. Given a range $[a,b]$, by Fact 8, we only need to find whether $T[a\,..\,b]$ contains a minimal order-j fragment. We can do this in $\mathcal{O}(1)$ time using one rank and one select query: $t = \mathsf{rank}_1(a-1, \mathsf{SP}_j) + 1$ and $\mathsf{select}_1(t, \mathsf{EP}_j)$. The select query returns the ending position of the leftmost minimal order-j fragment that starts after the position $a-1$; it remains to check whether this minimal order-j fragment is inside $[a,b]$.

Example 6. We consider T, SP_2 and EP_2 from Example 5. 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 position of the rightmost occurrence of rank $\mathsf{APP}_j[i]$ in $\mathsf{APP}_j[1\,..\,i-1]$ (or 0 if such an occurrence does not exist). We apply Fact 7 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 7. We consider T, SP_2 and EP_2 from Example 5. We start by setting $b = n = 18$ and scan PRE_2 from right to left: we have $a = 13$ because $\min\{\mathsf{PRE}_2[i] : i \in [18,21]\} = 13$. This gives fragment $T[13\,..\,18]$, which is minimal since $\mathsf{PRE}_2[b-1] = \mathsf{PRE}_2[17] < 13$. Then we set $b = n-1 = 17$ and have $a = 11$ because $\min\{\mathsf{PRE}_2[i] : i \in [17,21]\} = 11$. This gives fragment $T[11\,..\,17]$, which is not minimal since $\mathsf{PRE}_2[b-1] = \mathsf{PRE}_2[16] \geq 11$. Then we set $b = n-2 = 16$ and have $a = 11$ because $\min\{\mathsf{PRE}_2[i] : i \in [16,21]\} = 11$. This gives fragment $T[11\,..\,16]$, which is minimal since $\mathsf{PRE}_2[b-1] = \mathsf{PRE}_2[15] < 11$.

Lemma 1. *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 3, Lemma 1, and Fact 6. All the arrays SP_j and EP_j in total occupy $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ *bits* of space, which is $\mathcal{O}(n)$ space when measured in $\Theta(\log n)$-bit machine words. We obtain the following lemma.

Lemma 2. *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 perform binary search on j using Lemma 2 to find the smallest j for which $S_{T[a\,..\,b]}(j) < \sigma^j$. This results in $\mathcal{O}(\log \ell) = \mathcal{O}(\log\log_\sigma n)$ query time by Fact 6. 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 8. We consider T, SP_2 and EP_2 from Example 5. 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.

Proposition 4. *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 3 (Periodicity Lemma (weak version) [30]). *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 4. *If all strings in $\{UW : U \in \Sigma^k\}$ 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 that the period of aW is also p. All strings ZbW for a letter $b \neq a$ and $Z \in \Sigma^{k-1}$ must occur in S. Let $A = \{UW : U \in \Sigma^k\} \setminus \{ZaW : Z \in \Sigma^{k-1}\}$, 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 $UW, VW \in A$ 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 3), $j-i$ must be divisible by the period p of W. Hence, V ends with the letter a and $VW \notin A$, a contradiction. □

This concludes the proof of this lemma. □

Lemma 5. *If a shortest absent word of a string X is of length λ, then the length of a shortest absent word of XY is in $[\lambda, \lambda + \max\{10, 4 + \log_\sigma(|Y|/\lambda)\}]$.*

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

Lemma 6. *If a shortest absent word of XY is of length m, a shortest absent word of X is of length λ, and $|Y| \leq m \cdot \tau$, for a positive integer $\tau \geq 16$, then $m - \lambda \leq 10 + 2\log_\sigma \tau$.*

Proof. From Lemma 5 we have $\lambda \in [m - \max\{10, 4 + \log_\sigma(|Y|/\lambda)\}, m]$. If $\max\{10, 4+\log_\sigma(|Y|/\lambda)\} = 10$, then $m - \lambda \leq 10$ and we are done.

In the complementary case, since $|Y| \leq m \cdot \tau$, 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. $\mathcal{O}(n)$ Space and $\mathcal{O}(1)$ Query Time

Our linear-space solution of the ISAW problem with constant query time is an optimization of the $\mathcal{O}(n \log_\sigma n)$-space solution from Section 3 with some "boundary" cases processed using the data structure of Section 4. Let us first describe a simpler $\mathcal{O}(n \log_\sigma \log n)$-space data structure, which will be then optimized using the combinatorial insights from Section 5.

Recall that we denote by ℓ the length of a shortest absent word of T. The issue with the solution of Section 3 is that the 2d array $\mathsf{FTR}[1\,..\,\ell-1][1\,..\,n]$, equipped with fusion trees, occupies $\mathcal{O}(n \log_\sigma n)$ space. In order to reduce the memory consumption, we store the array FTR implicitly, computing its entries on demand, and utilize succinct fusion trees from Theorem 5 instead of usual fusion trees.

Recall that $\mathsf{FTR}[j][i]$ is the rightmost index of T such that $T[\mathsf{FTR}[j][i]\,..\,i]$ contains as substrings all strings of length j over Σ and it is equal to $\min\{\mathsf{PRE}_j[i-j+2], \ldots, \mathsf{PRE}_j[|\mathsf{PRE}_j|]\}$. Therefore, the content of the 2d array FTR can be "emulated" without storing it explicitly if one can compute in $\mathcal{O}(1)$ time the minima $\min\{\mathsf{PRE}_j[a], \ldots, \mathsf{PRE}_j[|\mathsf{PRE}_j|]\}$, for any $a \in [1, n]$. For $j \in [1, \ell-1]$ and $a \in [1, n]$, denote $M_{j,a} = \min\{\mathsf{PRE}_j[a], \ldots, \mathsf{PRE}_j[|\mathsf{PRE}_j|]\}$. Let us fix some j. Since the sequence $M_{j,1}, M_{j,2}, \ldots, M_{j,n}$ is non-decreasing, we can encode it in a $2n$-bit array B_j using the select data structure from Theorem 3 as follows: we construct B_j (initially empty) by considering $a = 1, 2, \ldots, n$ in increasing order and, for each a, we append to the end of B_j exactly $M_{j,a} - M_{j,a-1}$ zeroes followed by 1, setting $M_{j,0} = 0$ (i.e., we append the number $M_{j,a} - M_{j,a-1}$ written in unary); then, we have $M_{j,a} = \mathsf{select}_1(a, B_j) - a$.

Example 9. We consider T from Example 2 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
$M_{2,i}$	0	0	0	0	0	0	0	0	0	5	7	7	7	7	7	11	11	13			

In this case, we have $B_2 = \mathtt{11111111100000100111111000011001}$.

Besides access to the 2d array FTR, the algorithm of Section 3 also required access to the values $\arg\min\{\mathsf{PRE}_j[a], \ldots, \mathsf{PRE}_j[|\mathsf{PRE}_j|]\}$ in order to retrieve a witness shortest absent word. To this end, we build the $2n$-bit RMQ data structure from Theorem 2 on each array PRE_j; the data structure does not need to store the array PRE_j itself to compute $\arg\min$. The arrays B_j, for $j \in [1, \ell-1]$, equipped with select data structures, and the RMQ data structures on arrays PRE_j, for $j \in [1, \ell-1]$, can be constructed in total $\mathcal{O}(n\ell) = \mathcal{O}(n \log_\sigma n)$ time and they altogether occupy $\mathcal{O}(n \log_\sigma n)$ *bits* of space, which is $\mathcal{O}(n)$ space when measured in machine words.

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 constructed 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. But now we do not store the arrays $\mathsf{FTR}[1\,..\,\ell-1][i]$ explicitly, while still having $\mathcal{O}(1)$-time "oracle" access to their entries on demand. Hence, we can construct a succinct fusion tree of Theorem 5, for each array $\mathsf{FTR}[1\,..\,\ell-1][i]$, which takes $\mathcal{O}(\ell \log\log n)$ *bits* of space since the size of machine words is $w = \Theta(\log n)$ bits (a shared table mentioned in Theorem 5 is also precomputed for all the trees in $o(2^w) = o(n)$ time).

Thus, all the succinct fusion trees can be constructed in $\mathcal{O}(n \log_\sigma n)$ time and occupy $\mathcal{O}(n \log_\sigma n \log\log n)$ *bits*, which is $\mathcal{O}(n \log_\sigma \log n)$ space when measured in $\Theta(\log n)$-bit machine words. The ISAW queries are answered in $\mathcal{O}(1)$ time by the same algorithm as in Section 3.

Now we are to further reduce the memory usage of the data structure. We truncate all the arrays $\mathsf{FTR}[0\,..\,\ell-1][i]$ except those where i is a multiple of $\lfloor \log\log n \rfloor$ or $i = n$: namely, if i is a multiple of $\lfloor \log\log n \rfloor$ or $i = n$, then the succinct fusion tree for the whole array $\mathsf{FTR}[1\,..\,\ell-1][i]$ is stored, occupying $\mathcal{O}(\ell \log\log n)$ bits, by Theorem 5; otherwise ($i \neq n$ is not a multiple of $\lfloor \log\log n \rfloor$), we store the succinct fusion tree only for the subarray $\mathsf{FTR}[1\,..\,\lceil \log n / \log\log n \rceil][i]$, thus taking $\mathcal{O}(\log n)$ bits, by Theorem 5. In total, the space used is $\mathcal{O}(\frac{n}{\log\log n} \ell \log\log n + n \log n) = \mathcal{O}(n \log n)$ in bits or $\mathcal{O}(n)$ in words.

In order to answer an ISAW query for $T[a\,..\,b]$, we first check whether the length λ of a shortest absent word in $T[a\,..\,b]$ is smaller than $\log n/\log\log n$ by querying the fusion tree of $\mathsf{FTR}[1\,..\,\lceil \log n/\log\log n\rceil][b]$. If it is the case, then we have computed the length λ and we find the absent word itself using RMQs exactly as in the $\mathcal{O}(n\log_\sigma \log n)$-space solution described above.

Suppose that $\lambda \geq \log n/\log\log n$. We compute b', the successor of b among the positions i for which we have not truncated $\mathsf{FTR}[1\,..\,\ell-1][i]$: $b' = \min\{n, \lceil b/\lfloor\log\log n\rfloor\rceil \cdot \lfloor \log\log n\rfloor\}$. Observe that $[a,b] \subseteq [a,b']$. Then, using the fusion tree of $\mathsf{FTR}[1\,..\,\ell-1][b']$, we compute the smallest m such that $\mathsf{FTR}[m][b'] < a$. Then, m is the length of a shortest absent word in $T[a\,..\,b']$. Denote $X = T[a\,..\,b]$ and $T[a\,..\,b'] = XY$ where Y is a suffix of $T[a\,..\,b']$ of length $b'-b$. We obviously have $\lambda \leq m$. Since $|Y| = b'-b < \log\log n$ and $m \geq \log n/\log\log n$, we have $|Y| < m$. It follows from Lemma 6 that the answer λ is within a range of length 18 from m. Therefore, λ belongs to the range $[m-18, m]$ and we can find it in $\mathcal{O}(1)$ time using $\mathcal{O}(1)$ queries of the $\mathcal{O}(n)$-space data structure encapsulated by Lemma 2. We thus arrive at the main result of the paper.

Theorem 1. *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 in $\mathcal{O}(n\log_\sigma n)$ time a data structure of size $\mathcal{O}(n)$ that, for any given query $[a,b]$, can compute in $\mathcal{O}(1)$ time a shortest string over Σ that does not occur in $T[a\,..\,b]$.*

7. Open Problems

It remains open whether a data structure for the ISAW problem with the same query time and space complexities as the one encapsulated in Theorem 1 can be constructed in linear time. Also, it is natural to pose the following related open problem, which may require the development of fundamentally different techniques. 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 representation of a shortest string over $\Sigma_{[i,j]}$ that is absent in the fragment $T[i]\cdots T[j]$ of T, where $\Sigma_{[i,j]}$ is the set of letters from Σ occurring in the fragment $T[i]\cdots T[j]$.

References

[1] Abedin, P., Ganguly, A., Hon, W., Nekrich, Y., Sadakane, K., Shah, R., Thankachan, S.V., 2018. A linear-space data structure for Range-LCP queries in poly-logarithmic time, in: Computing and Combinatorics - 24th International Conference, COCOON 2018, pp. 615–625. URL: `https://doi.org/10.1007/978-3-319-94776-1_51`, doi:`10.1007/978-3-319-94776-1\_51`.

[2] Abedin, P., Ganguly, A., Pissis, S.P., Thankachan, S.V., 2020. Efficient data structures for range shortest unique substring queries. Algorithms 13, 276. URL: `https://doi.org/10.3390/a13110276`, doi:`10.3390/a13110276`.

[3] Amir, A., Apostolico, A., Landau, G.M., Levy, A., Lewenstein, M., Porat, E., 2014. Range LCP. J. Comput. Syst. Sci. 80, 1245–1253. URL: `https://doi.org/10.1016/j.jcss.2014.02.010`, doi:`10.1016/j.jcss.2014.02.010`.

[4] Amir, A., Charalampopoulos, P., Pissis, S.P., Radoszewski, J., 2020. Dynamic and internal longest common substring. Algorithmica 82, 3707–3743. URL: `https://doi.org/10.1007/s00453-020-00744-0`, doi:`10.1007/s00453-020-00744-0`.

[5] Ayad, L.A.K., Badkobeh, G., Fici, G., Héliou, A., Pissis, S.P., 2019. Constructing antidictionaries in output-sensitive space, in: Data Compression Conference, DCC 2019, IEEE. pp. 538–547. URL: `https://doi.org/10.1109/DCC.2019.00062`, doi:`10.1109/DCC.2019.00062`.

[6] Babenko, M.A., Gawrychowski, P., Kociumaka, T., Kolesnichenko, I.I., Starikovskaya, T., 2016. Computing minimal and maximal suffixes of a substring. Theor. Comput. Sci. 638, 112–121. URL: `https://doi.org/10.1016/j.tcs.2015.08.023`, doi:`10.1016/j.tcs.2015.08.023`.

[7] Babenko, M.A., Gawrychowski, P., Kociumaka, T., Starikovskaya, T., 2015. Wavelet trees meet suffix trees, in: Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, SIAM. pp. 572–591. URL: `https://doi.org/10.1137/1.9781611973730.39`, doi:`10.1137/1.9781611973730.39`.

[8] Badkobeh, G., Charalampopoulos, P., Pissis, S.P., 2021. Internal shortest absent word queries, in: Gawrychowski, P., Starikovskaya, T. (Eds.), 32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. pp. 24:1–24:18.

[9] Barton, C., Héliou, A., Mouchard, L., Pissis, S.P., 2014. Linear-time computation of minimal absent words using suffix array. BMC Bioinform. 15, 388. URL: `https://doi.org/10.1186/s12859-014-0388-9`, doi:`10.1186/s12859-014-0388-9`.

[10] Barton, C., Héliou, A., Mouchard, L., Pissis, S.P., 2015. Parallelising the computation of minimal absent words, in: Parallel Processing and Applied Mathematics - 11th International Conference, PPAM 2015. Revised Selected Papers, Part II, Springer. pp. 243–253. URL: `https://doi.org/10.1007/978-3-319-32152-3_23`, doi:`10.1007/978-3-319-32152-3\_23`.

[11] Bender, M.A., Farach-Colton, M., 2000. The LCA problem revisited, in: LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Proceedings, Springer. pp. 88–94. URL: `https://doi.org/10.1007/10719839_9`, doi:`10.1007/10719839\_9`.

[12] Berkman, O., Vishkin, U., 1993. Recursive star-tree parallel data structure. SIAM J. Comput. 22, 221–242. URL: `https://doi.org/10.1137/0222017`, doi:`10.1137/0222017`.

[13] Birenzwige, O., Golan, S., Porat, E., 2020. Locally consistent parsing for text indexing in small space, in: Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, SIAM. pp. 607–626. URL: `https://doi.org/10.1137/1.9781611975994.37`, doi:`10.1137/1.9781611975994.37`.

[14] Chairungsee, S., Crochemore, M., 2012. Using minimal absent words to build phylogeny. Theor. Comput. Sci. 450, 109–116. URL: `https://doi.org/10.1016/j.tcs.2012.04.031`, doi:`10.1016/j.tcs.2012.04.031`.

[15] Chan, T.M., Larsen, K.G., Pătraşcu, M., 2011. Orthogonal range searching on the ram, revisited, in: Hurtado, F., van Kreveld, M.J. (Eds.), Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011, ACM. pp. 1–10. URL: `https://doi.org/10.1145/1998196.1998198`, doi:`10.1145/1998196.1998198`.

[16] Charalampopoulos, P., Crochemore, M., Fici, G., Mercaş, R., Pissis, S.P., 2018a. Alignment-free sequence comparison using absent words. Inf. Comput. 262, 57–68. URL: `https://doi.org/10.1016/j.ic.2018.06.002`, doi:`10.1016/j.ic.2018.06.002`.

[17] Charalampopoulos, P., Crochemore, M., Pissis, S.P., 2018b. On extended special factors of a word, in: String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Springer. pp. 131–138. URL: `https://doi.org/10.1007/978-3-030-00479-8_11`, doi:`10.1007/978-3-030-00479-8\_11`.

[18] Charalampopoulos, P., Gawrychowski, P., Mozes, S., Weimann, O., 2021. An almost optimal edit distance oracle. CoRR abs/2103.03294. `arXiv:2103.03294`.

[19] Charalampopoulos, P., Kociumaka, T., Mohamed, M., Radoszewski, J., Rytter, W., Straszynski, J., Walen, T., Zuba, W., 2020a. Counting distinct patterns in internal dictionary matching, in: 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 8:1–8:15. URL: `https://doi.org/10.4230/LIPIcs.CPM.2020.8`, doi:`10.4230/LIPIcs.CPM.2020.8`.

[20] Charalampopoulos, P., Kociumaka, T., Mohamed, M., Radoszewski, J., Rytter, W., Walen, T., 2019. Internal dictionary matching, in: 30th International Symposium on Algorithms and Computation, ISAAC 2019, Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 22:1–22:17. URL: `https://doi.org/10.4230/LIPIcs.ISAAC.2019.22`, doi:`10.4230/LIPIcs.ISAAC.2019.22`.

[21] Charalampopoulos, P., Kociumaka, T., Wellnitz, P., 2020b. Faster approximate pattern matching: A unified approach, in: 61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, IEEE. pp. 978–989. URL: `https://doi.org/10.1109/FOCS46700.2020.00095`, doi:`10.1109/FOCS46700.2020.00095`.

[22] Crochemore, M., Héliou, A., Kucherov, G., Mouchard, L., Pissis, S.P., Ramusat, Y., 2020. Absent words in a sliding window with applications. Inf. Comput. 270. doi:`10.1016/j.ic.2019.104461`.

[23] Crochemore, M., Mignosi, F., Restivo, A., 1998. Automata and forbidden words. Inf. Process. Lett. 67, 111–117. doi:`10.1016/S0020-0190(98)00104-5`.

[24] Crochemore, M., Mignosi, F., Restivo, A., Salemi, S., 2000. Data compression using antidictionaries. Proceedings of the IEEE 88, 1756–1768. doi:`10.1109/5.892711`.

[25] Farach, M., 1997. Optimal suffix tree construction with large alphabets, in: 38th Annual Symposium on Foundations of Computer Science, FOCS 1997, IEEE Computer Society. pp. 137–143. URL: `https://doi.org/10.1109/SFCS.1997.646102`, doi:`10.1109/SFCS.1997.646102`.

[26] Ferenczi, S., 1999. Complexity of sequences and dynamical systems. Discret. Math. 206, 145–154. URL: `https://doi.org/10.1016/S0012-365X(98)00400-2`, doi:`10.1016/S0012-365X(98)00400-2`.

[27] Fici, G., Gawrychowski, P., 2019. Minimal absent words in rooted and unrooted trees, in: String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Springer. pp. 152–161. doi:`10.1007/978-3-030-32686-9\_11`.

[28] Fici, G., Mignosi, F., Restivo, A., Sciortino, M., 2006. Word assembly through minimal forbidden words. Theor. Comput. Sci. 359, 214–230. doi:`10.1016/j.tcs.2006.03.006`.

[29] Fici, G., Restivo, A., Rizzo, L., 2019. Minimal forbidden factors of circular words. Theor. Comput. Sci. 792, 144–153. URL: `https://doi.org/10.1016/j.tcs.2018.05.037`, doi:`10.1016/j.tcs.2018.05.037`.

[30] Fine, N.J., Wilf, H.S., 1965. Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society 16, 109–114. URL: `http://www.jstor.org/stable/2034009`.

[31] Fischer, J., Heun, V., 2011. Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. 40, 465–492. URL: `https://doi.org/10.1137/090779759`, doi:`10.1137/090779759`.

[32] Fredman, M.L., Willard, D.E., 1993. Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. 47, 424–436. doi:`10.1016/0022-0000(93)90040-4`.

[33] Fujishige, Y., Tsujimaru, Y., Inenaga, S., Bannai, H., Takeda, M., 2016. Computing DAWGs and minimal absent words in linear time for integer alphabets, in: 41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 38:1–38:14. doi:`10.4230/LIPIcs.MFCS.2016.38`.

[34] Ganguly, A., Patil, M., Shah, R., Thankachan, S.V., 2018. A linear space data structure for range LCP queries. Fundam. Inform. 163, 245–251. URL: `https://doi.org/10.3233/FI-2018-1741`, doi:`10.3233/FI-2018-1741`.

[35] Garcia, S.P., Pinho, A.J., Rodrigues, J.M.O.S., Bastos, C.A.C., Ferreira, P.J.S.G., 2011. Minimal absent words in prokaryotic and eukaryotic genomes. PLoS ONE 6. doi:`10.1371/journal.pone.0016065`.

[36] Grossi, R., Orlandi, A., Raman, R., Rao, S.S., 2009. More haste, less waste: Lowering the redundancy in fully indexable dictionaries, in: Albers, S., Marion, J.Y. (Eds.), 26th International Symposium on Theoretical Aspects of Computer Science, STACS 2009, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. pp. 517–528. doi:`10.4230/`

LIPIcs.STACS.2009.1847.

[37] Harel, D., Tarjan, R.E., 1984. Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13, 338–355. URL: https://doi.org/10.1137/0213024, doi:10.1137/0213024.

[38] Jacobson, G., 1989. Space-efficient static trees and graphs, in: 30th Annual Symposium on Foundations of Computer Science, FOCS 1989, IEEE Computer Society. pp. 549–554. doi:10.1109/SFCS.1989.63533.

[39] Keller, O., Kopelowitz, T., Feibish, S.L., Lewenstein, M., 2014. Generalized substring compression. Theor. Comput. Sci. 525, 42–54. URL: https://doi.org/10.1016/j.tcs.2013.10.010, doi:10.1016/j.tcs.2013.10.010.

[40] Kempa, D., Kociumaka, T., 2019. 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, ACM. pp. 756–767. URL: https://doi.org/10.1145/3313276.3316368, doi:10.1145/3313276.3316368.

[41] Kociumaka, T., 2016. Minimal suffix and rotation of a substring in optimal time, in: 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, pp. 28:1–28:12. URL: https://doi.org/10.4230/LIPIcs.CPM.2016.28, doi:10.4230/LIPIcs.CPM.2016.28.

[42] Kociumaka, T., 2018. Efficient Data Structures for Internal Queries in Texts. Ph.D. thesis. University of Warsaw. URL: https://mimuw.edu.pl/~kociumaka/files/phd.pdf.

[43] Kociumaka, T., Radoszewski, J., Rytter, W., Walen, T., 2012. Efficient data structures for the factor periodicity problem, in: String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, pp. 284–294. URL: https://doi.org/10.1007/978-3-642-34109-0_30, doi:10.1007/978-3-642-34109-0_30.

[44] Kociumaka, T., Radoszewski, J., Rytter, W., Walen, T., 2015. 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, SIAM. pp. 532–551. URL: https://doi.org/10.1137/1.9781611973730.36, doi:10.1137/1.9781611973730.36.

[45] Landau, G.M., Vishkin, U., 1988. Fast string matching with k differences. J. Comput. Syst. Sci. 37, 63–78. URL: https://doi.org/10.1016/0022-0000(88)90045-1, doi:10.1016/0022-0000(88)90045-1.

[46] Matsuda, K., Sadakane, K., Starikovskaya, T., Tateshita, M., 2020. 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, pp. 23:1–23:13. URL: https://doi.org/10.4230/LIPIcs.CPM.2020.23, doi:10.4230/LIPIcs.CPM.2020.23.

[47] Mieno, T., Kuhara, Y., Akagi, T., Fujishige, Y., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M., 2020. Minimal unique substrings and minimal absent words in a sliding window, in: 46th SOFSEM, Springer. pp. 148–160. doi:10.1007/978-3-030-38919-2_13.

[48] Mignosi, F., Restivo, A., Sciortino, M., 2002. Words and forbidden factors. Theor. Comput. Sci. 273, 99–117. doi:10.1016/S0304-3975(00)00436-9.

[49] Munro, J.I., Navarro, G., Nekrich, Y., 2020. Text indexing and searching in sublinear time, in: 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 24:1–24:15. URL: https://doi.org/10.4230/LIPIcs.CPM.2020.24, doi:10.4230/LIPIcs.CPM.2020.24.

[50] Navarro, G., 2016. Compact Data Structures - A Practical Approach. Cambridge University Press.

[51] Ota, T., Morita, H., 2010. 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, IEEE. pp. 72–77. doi:10.1109/ISITA.2010.5649621.

[52] Pătraşcu, M., Thorup, M., 2014. Dynamic integer sets with optimal rank, select, and predecessor search, in: 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, IEEE Computer Society. pp. 166–175. URL: https://doi.org/10.1109/FOCS.2014.26, doi:10.1109/FOCS.2014.26.

[53] Pratas, D., Silva, J.M., 2020. Persistent minimal sequences of SARS-CoV-2. Bioinformatics doi:10.1093/bioinformatics/btaa686. btaa686.

[54] Raskhodnikova, S., Ron, D., Rubinfeld, R., Smith, A.D., 2013. Sublinear algorithms for approximating string compressibility. Algorithmica 65, 685–709. URL: https://doi.org/10.1007/s00453-012-9618-6, doi:10.1007/s00453-012-9618-6.

[55] Rubinchik, M., Shur, A.M., 2017. Counting palindromes in substrings, in: String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Springer. pp. 290–303. URL: https://doi.org/10.1007/978-3-319-67428-5_25, doi:10.1007/978-3-319-67428-5_25.

[56] Silva, R.M., Pratas, D., Castro, L., Pinho, A.J., Ferreira, P.J.S.G., 2015. Three minimal sequences found in Ebola virus genomes and absent from human DNA. Bioinform. 31, 2421–2425. URL: https://doi.org/10.1093/bioinformatics/btv189, doi:10.1093/bioinformatics/btv189.

[57] Tanimura, Y., Nishimoto, T., Bannai, H., Inenaga, S., Takeda, M., 2017. Small-space LCE data structure with constant-time queries, in: 42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017, Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 10:1–10:15. URL: https://doi.org/10.4230/LIPIcs.MFCS.2017.10, doi:10.4230/LIPIcs.MFCS.2017.10.

[58] Tiskin, A., 2008. Semi-local string comparison: Algorithmic techniques and applications. Math. Comput. Sci. 1, 571–603. URL: https://doi.org/10.1007/s11786-007-0033-3, doi:10.1007/s11786-007-0033-3.

[59] Yao, A.C., 1982. Space-time tradeoff for answering range queries (extended abstract), in: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, ACM. pp. 128–136. doi:10.1145/800070.802185.