Enhancing Generalized Compressed Suffix Trees, with Applications

Sankardeep Chakraborty
The University of Tokyo, Japan

Kunihiko Sadakane
The University of Tokyo, Japan

Wiktor Zuba
CWI, Amsterdam, The Netherlands

Abstract

Generalized suffix trees are data structures for storing and searching a set of strings. Though many string problems can be solved efficiently using them, their space usage can be large relative to the size of the input strings. For a set of strings with n characters in total, generalized suffix trees use $\mathrm{O}(n \log n)$ bit space, which is much larger than the strings that occupy $n \log \sigma$ bits where σ is the alphabet size. Generalized compressed suffix trees use just $\mathrm{O}(n \log \sigma)$ bits but support the same basic operations as the generalized suffix trees. However, for some sophisticated operations we need to add auxiliary data structures of $\mathrm{O}(n \log n)$ bits. This becomes a bottleneck for applications involving big data. In this paper, we enhance the generalized compressed suffix trees while still retaining their space efficiency. First, we give an auxiliary data structure of $\mathrm{O}(n)$ bits for generalized compressed suffix trees such that given a suffix s of a string and another string t, we can find the suffix of t that is closest to s. Next, we give a $\mathrm{o}(n)$ bit data structure for finding the ancestor of a node in a (generalized) compressed suffix tree with given string depth. Finally, we give data structures for a generalization of the document listing problem from arrays to trees. We also show their applications to suffix-prefix matching problems.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases suffix tree, compact data structure, suffix-prefix query, weighted level ancestor

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2024.18

Funding *Sankardeep Chakraborty*: Supported by MEXT Quantum Leap Flagship Program (MEXT Q-LEAP) Grant Number JPMXS0120319794.
Wiktor Zuba: Received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 101034253 and 872539.

1 Introduction

Suffix trees are data structures for string matching [29]. In addition to the basic pattern matching problem, they can also be used for other problems such as finding longest common extensions, maximal pairs, approximate string matching, etc. [14]. They can be further extended to generalized suffix trees (GSTs for short) storing suffixes of a set of strings, which gives them many applications in bioinformatics such as longest common substrings, maximal unique matches [5], and maximal exact matches [17]. Hereafter we do not distinguish suffix trees and GSTs unless specified because GST is the suffix tree of the string obtained by concatenating all the strings from the set.

Though suffix trees are the most basic data structures in string processing, one drawback is their space usage. Though the suffix tree of a string uses $\mathrm{O}(n)$ machine words, where n is the string length, that alone already requires huge memory. It was estimated that for a human genome, which has about 3 billion characters, the suffix tree uses more than 40 GB of memory [16]. Therefore there has been much research on reducing the space requirement of suffix trees. There are two approaches; one is to omit some of the components of suffix trees and the other is to compress the components. For the first approach, suffix arrays [19], enhanced suffix arrays [1], and space efficient suffix trees [21] have been proposed as space-efficient alternatives to the suffix trees. For the second approach, the compressed suffix arrays [13] and compressed suffix trees [25] have been proposed for compressing the respective standard structures. The first approach aims mainly at reduction of the practical space usage as asymptotically the space usage remains the same; suffix arrays and enhanced suffix arrays use $\mathrm{O}(n \log n)$ bits of space for a string of length n – the same as the suffix trees. The second approach, on the other hand, aims at improving the asymptotic bounds; compressed suffix arrays and trees use $\mathrm{O}(n \log \sigma)$ bits where σ is the size of the alphabet of the string. These data structures are truly linear space data structures – the space is linear to the actual input size – $n \log \sigma$ bits[1] (the space needed to represent the string).

Though the compressed suffix trees support the same set of basic operations as the suffix trees, some of auxiliary data structures for supporting extended operations still use $\mathrm{O}(n \log n)$ bit space, which dominates the space of the entire data structure.

1.1 Our contributions

In this paper, we enhance the generalized compressed suffix trees. First we add auxiliary data structures which for a given suffix t and a string ID i allow finding the suffix of the string i most similar to t. For two given suffixes it is easy to compute the length of their longest common prefix using the suffix tree even if the suffixes belong to different strings. However, if a suffix t of a string is fixed and the ID i of another string is given, finding the suffix of the string i with the longest common prefix with s takes time due to the multiplicity of the possible candidate suffixes.

▶ **Theorem 1** (Closest colored suffixes). *We are given a set of strings $S_1, S_2, \ldots, S_k$ on an alphabet of size σ. The total length of the strings is n. There exists a data structure using $\mathrm{SIZE}_{\mathrm{SA}}(2n, \sigma) + \mathrm{O}(n + k)$ bits so that given a suffix t of a string S_j and an index i, we can obtain the suffix s of S_i that has the maximum LCP with t in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log \log k)$ time, where $\mathrm{SIZE}_{\mathrm{SA}}(n, \sigma)$ is the size of a data structure storing a suffix array for a string of length n on an alphabet of size σ, and $\mathrm{TIME}_{\mathrm{SA}}$ is the time for obtaining an entry of a suffix array or its inverse.*

Note that $\mathrm{TIME}_{\mathrm{SA}}$ also depends on n and σ in general, but we omit them because they are fixed throughout the paper. If we use the data structure in the second row of Table 1, the space and the time complexities become $2n \log \sigma + \mathrm{O}(n + k)$ bits and $\mathrm{O}(\log n \log \log k)$ time, respectively. An existing solution [18] has $\mathrm{O}(\log \log k)$ query time using $\mathrm{O}(n \log n)$ bit space. Our algorithm is faster than the original GST, which is $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log n)$ time.

Next we give a succinct index for weighted level ancestors in compressed suffix trees.

▶ **Theorem 2** (Weighted level ancestors). *By adding an auxiliary data structure of $\mathrm{o}(n)$ bits to the compressed suffix tree, we can compute the nearest ancestor of a node with string depth smaller than a given value in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log \log n)$ time.*

This is faster than the original GST, which is $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log n)$ time. The proofs are given in Section 3.

[1] Throughout the paper the base of the logarithm is two.

Table 1 Size in bits and query time of suffix arrays and compressed suffix arrays, where n is the length of the string and σ is its alphabet size. H_k is the k-th order entropy of the string. Time for obtaining an entry of the suffix array is denoted by $\mathrm{TIME}_{\mathrm{SA}}$.

Index	Space ($\mathrm{SIZE}_{\mathrm{SA}}(n, \sigma)$)	Query time ($\mathrm{TIME}_{\mathrm{SA}}$)
Suffix array [19]	$n \log n$	$\mathrm{O}(1)$
Compressed suffix array [13]	$n \log \sigma + \mathrm{O}(n)$	$\mathrm{O}(\log n)$
Compressed suffix array [13]	$\mathrm{O}(\epsilon^{-1} n \log \sigma)$	$\mathrm{O}(\log^{\epsilon} n)$
FM-index [7]	$nH_k + \mathrm{o}(n \log \sigma)$	$\mathrm{O}(\frac{\log \sigma}{\log \log n})$

We also give applications of these two enhancements in Section 4. Our proposed data structures are used to solve the suffix-prefix matching problems [18] (see also [31] for their approximate version). Existing solutions use $\mathrm{O}(n \log n)$ bit space, whereas ours use linear ($\mathrm{O}(n \log \sigma)$ bit) space. For solving this problem, we generalize the document listing problem [22] from arrays to trees providing data structures that are of independent interest.

2 Preliminaries

2.1 Suffix arrays, suffix trees, and their compression

A string S of length n on an alphabet $\mathcal{A}$ is an array $S[1, n]$ of characters in $\mathcal{A}$. We assume the alphabet is an ordered set. We add a terminator \$ at the end of the string, that is, $S[n+1] = \$$, which is smaller than any character in $\mathcal{A}$. The character at position i in the string S is denoted by $S[i]$. A substring of S is the concatenation of characters $S[i], S[i+1], \ldots, S[j]$ and denoted by $S[i, j]$. Substrings of the form $S[i, n]$ and $S[1, i]$ are called suffixes and prefixes, respectively. For two strings s, t, $\mathrm{LCP}(s, t)$ is defined to be the length of the longest common prefix between them.

The suffix array [19] of a string S of length n is an integer array $\mathrm{SA}[0, n]$, where $\mathrm{SA}[i] = j$ means that the suffix $S[j, n]$ is lexicographically the i-th suffix among all the suffixes of S ($S[0] = n + 1$). The (classic) suffix array uses $n \log n$ bits of space for the array SA, and $n \log \sigma$ bits of space for the string S itself.

The suffix tree of a string is a compacted trie representing all the suffixes of the string [29]. The suffix tree has $n + 1$ leaves, each corresponding to a suffix of S (including the last suffix $S[n+1, n+1] = \$$). Each edge of the suffix tree has a string label. We define the string label of a node as the concatenation of the labels of the edges between the root and the node. The string label of the i-th leaf coincides with lexicographically the i-th suffix. The string depth of an internal node v of the suffix tree is the length of the string label of v. For two suffixes s and t, $\mathrm{LCP}(s, t)$ is equal to the string depth of the lowest common ancestor between their corresponding leaves.

The suffix array can be compressed to $\mathrm{O}(n \log \sigma)$ bits where $\sigma = |\mathcal{A}|$ so that each entry $\mathrm{SA}[i]$ can be computed in $\mathrm{polylog}(n)$ time [13]. The inverse suffix array $\mathrm{ISA}[1, n+1]$ of a string is an integer array such that $\mathrm{ISA}[j] = i$ if and only if $\mathrm{SA}[i] = j$. The inverse array can be computed within the same time complexity as the suffix array. Let $\mathrm{TIME}_{\mathrm{SA}}$ denote the time for computing a value of a suffix array or an inverse suffix array. The space and query time complexities for compressed versions of the suffix arrays are shown in Table 1.

The compressed suffix tree [25] of a string S consists of the compressed suffix array of S, a balanced parentheses (BP) representation [20, 23] of the compacted trie, and a bit-vector storing the information about the string depths of nodes. The second and the third components use $4n + \mathrm{o}(n)$ bits and $2n + \mathrm{o}(n)$ bits, respectively. Using these components, we can compute the string depth of a node in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time.

Using the compressed suffix tree, we can support the following operations:

- Finding the leaf corresponding to the lexicographically i-th suffix in constant time.
- Finding the lowest common ancestor of two nodes in constant time.
- Finding the level ancestor (the ancestor of a node with given depth) in constant time [23]. Note that this depth is not the string depth.
- Computing the string depth of a node in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time.
- Computing the edge labels of length ℓ in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} + \ell)$ time.

Note that the compressed suffix tree of [25] supports weighted level ancestor queries w.r.t. string depths in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log n)$ time by a binary search using (unweighted) level ancestor queries and string depth queries. If we can use $\mathrm{O}(n \log n)$ bits of space we can support this query in constant time [3]. In this paper we give an index supporting the queries in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log\log n)$ time using additional $\mathrm{o}(n)$ bits to the compressed suffix trees.

2.2 Bit-vectors and rank/select dictionaries

A bit-vector is a string $B[1, n]$ on alphabet $\{0, 1\}$ supporting the following three operations.

- $\mathrm{access}(B, i)$: returns $B[i]$.
- $\mathrm{rank}_c(B, i)$: returns the number of c's ($c \in \{0, 1\}$) in $B[1, i]$.
- $\mathrm{select}_c(B, i)$: returns the position of the i-th occurrence of $c \in \{0, 1\}$. If $i > \mathrm{rank}_c(B, n)$, we define $\mathrm{select}_c(B, i) = n + 1$. We also define $\mathrm{select}_c(B, 0) = 0$.

We can perform each of these operations in constant time using $n + \mathrm{o}(n)$ bits of space [24].

The predecessor $\mathrm{pred}_c(B, i)$ and the successor $\mathrm{succ}_c(B, i)$ are the positions of c closest to i. They can be computed in constant time as $\mathrm{pred}_c(B, i) := \mathrm{select}_c(B, \mathrm{rank}_c(B, i - 1))$ and $\mathrm{succ}_c(B, i) := \mathrm{select}_c(B, \mathrm{rank}_c(B, i) + 1)$. Note that $\mathrm{pred}_c(B, i) < i < \mathrm{succ}_c(B, i)$.

2.3 Generalized suffix arrays and trees

We are given a set of strings $S_1, S_2, \ldots, S_k$ on an alphabet of size σ. We concatenate them into string $\mathcal{S} = S_1\$S_2\$ \cdots S_k\$$. The generalized suffix array/tree of the set of the strings is just the suffix array/tree of $\mathcal{S}$ with the following modification.

We create a bit-vector D of length $n + k$ where $n = |S_1| + |S_2| + \cdots + |S_k|$, and set 1's for the positions of \$'s in $\mathcal{S}$. Then, given a position j in $\mathcal{S}$, we can compute the ID d of the string S_d containing the position j in constant time by $d := \mathrm{rank}_1(D, j) + 1$. We define the document array $A[0, n + k]$ as $A[i] := \mathrm{rank}_1(D, \mathrm{SA}_{\mathcal{S}}[i]) + 1$. we do not store the document array explicitly because it uses $n \log k$ bits and each entry can be computed from the (compressed) suffix array in $\mathrm{TIME}_{\mathrm{SA}}$ time.

Sadakane [26] enhanced generalized compressed suffix trees to compute the number of occurrences of a pattern in each of the strings efficiently. After creating (compressed) suffix arrays (and inverse suffix arrays) of $\mathcal{S}$ and each S_d we can convert the rank r_{l} of a suffix of S_d into the rank r_{g} in $\mathcal{S}$ and vice versa in $\mathrm{O}(t_{SA})$ time as follows.

$$r_{\mathrm{g}} = \mathrm{ISA}_{\mathcal{S}}[\mathrm{SA}_{S_d}[r_{\mathrm{l}}] + s_d] \quad (1)$$

$$r_{\mathrm{l}} = \mathrm{ISA}_{S_d}[\mathrm{SA}_{\mathcal{S}}[r_{\mathrm{g}}] - s_d] \quad (2)$$

where $s_d = |S_1| + |S_2| + \cdots + |S_{d-1}| + d - 1$. We can compute s_d in constant time using the bit-vector D, namely, $s_d = \mathrm{select}_1(D, d - 1)$. Recall that d is computed from r_{g} by $d := \mathrm{rank}_1(D, r_{\mathrm{g}}) + 1$. We call r_{g} and r_{l} as the global and the local rank of the suffix, respectively.

Figure 1 shows the generalized suffix tree for a set of strings ACAA, ACAG, ACGC, CACA. Note that we use the same example as [18]. Each string is appended with a terminator \$. To bound the degree of a node by $\sigma + 1$, we add an artificial node if a node has more than

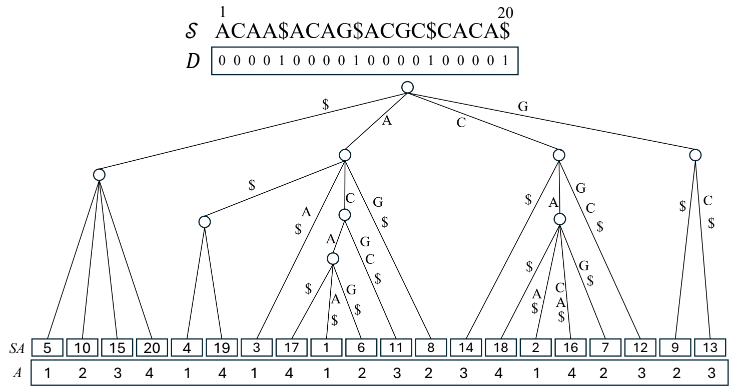

1
20
𝒮 ACAA$ACAG$ACGC$CACA$
D 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
SA 5 10 15 20 4 19 3 17 1 6 11 8 14 18 2 16 7 12 9 13
A 1 2 3 4 1 4 1 4 1 2 3 2 3 4 1 4 2 3 2 3

Figure 1 Generalized suffix tree for a set of strings ACAA, ACAG, ACGC, CACA. SA is the suffix array of the concatenated string $\mathcal{S}$, and $A$ is the array storing ID's of suffixes.

one edge with label starting from \$. For example, the root node has an edge with label \$ pointing to a node with four edges labeled \$. We can distinguish \$'s by their positions in $\mathcal{S}$. We define the string depth of an artificial node as that of its parent node. We can compute it using the same algorithm as that for normal nodes.

2.4 LCP arrays

For a string T of length n and its suffix array SA, we define the LCP (longest common prefix) array $L[1,n]$ as

$$L[i] = \mathrm{LCP}(T[SA[i-1],n], T[SA[i],n]).$$

If we store L in plain form, we need $n \log n$ bits of space. However, if we have access to the suffix array, we can compress it into $2n + \mathrm{o}(n)$ bits so that any entry of $L[i]$ is computed in $\mathrm{TIME}_{\mathrm{SA}} + \mathrm{O}(1)$ time [25].

The length of the LCP between two suffixes $T[SA[p],n]$ and $T[SA[q],n]$ of a string can be obtained by $\min_{p<i\le q} L[i]$. The index i attaining the minimum value can be computed in constant time using the $2n + \mathrm{o}(n)$ bit data structure for range minimum queries [8]. We can also compute it using the compressed suffix tree [25]. In this case, we use $4n + \mathrm{o}(n)$ bits for the tree topology of the suffix tree.

2.5 Rank and select data structures for large alphabets

Let $T[1,n]$ be a string on alphabet $\mathcal{A}$ of size k. As a generalization of the case of bit-vectors, we can define operations $\mathrm{access}(T,i)$, $\mathrm{rank}_c(T,i)$, and $\mathrm{select}_c(T,i)$ for $c \in \mathcal{A}$. The wavelet tree [12] supports all the operations in $\mathrm{O}(\log k)$ time using $(n+\mathrm{o}(n))\log k$ bit space. Golynski et al. [11] gave a data structure supporting select in constant time and rank and access in $\mathrm{O}(\log\log k)$ time using $(n + \mathrm{o}(n))\log k + 2n$ bit space[2].

These data structures encode the string in a specific form. Therefore we cannot further compress the string. Barbay et al. [2] considered another approach; they design succinct indexes for abstract data types. Their results are summarized as follows[3].

[2] There are other variants.
[3] The claim is slightly simplified from the original one.

▶ **Theorem 3** (Theorem 2.10 in [2]). *Given support for* select *in $f(n,k)$ time on a string S of length n on an alphabet of size k, we can support* access, rank*, predecessor, and successor (for any character) in $\mathrm{O}(f(n,k)\log\log k)$ time with a succinct index using $\mathrm{O}(n\log k/\log\log k)$ bits of space.*

Though this index uses asymptotically smaller space than the string itself, its size still depends on the alphabet size k. In this paper we follow the abstract data type approach and give a new index using less space (see Lemma 6).

2.6 Nearest marked ancestors

Let T be a rooted tree with some of the nodes marked. In the nearest marked ancestor problem, for a given node v of T we want to find its closest marked ancestor. Tsur [28] gave solutions for a generalized version of this problem – nearest colored ancestor – where each node has a color and we find the nearest ancestor with a given color. In this paper we only use the nearest marked ancestor queries, thus we provide a simplified statement.

▶ **Theorem 4** (A simplified version of Theorem 1 of [28]). *There exists a representation of T that uses $n + \mathrm{o}(n)$ bits in addition to a $2n + \mathrm{o}(n)$ bit representation of the tree topology that allows for answering the nearest marked ancestor queries in $\mathrm{O}(1)$ time.*

2.7 Weighted level ancestor queries

Consider a rooted tree with n nodes where each node has an integer weight in $[0, U]$ and on any path from a leaf to the root, the weights are non-increasing. The weighted level ancestor $\mathrm{WA}(v, w)$ of node v is the closest node on the path from v to the root that has weight smaller than w. Kopelowitz and Lewenstein [15] gave a data structure using linear space that answers a query in the same time complexity as finding the predecessor among n values in $[0, U]$. For the case of $U = \mathrm{O}(n)$, we obtain a data structure using $\mathrm{O}(n \log n)$ bits of space that answers a query in $\mathrm{O}(\log\log n)$ time. As shown above, for the case that weights are equal to the string depths in a suffix tree, there is a data structure using $\mathrm{O}(n \log n)$ bits of space that supports the query in constant time [3]. In Section 3.3 we show how to reduce the space at the cost of more expensive queries.

2.8 Tree Covering

Here we provide an overview of Farzan and Munro's tree covering representation and its application in creating a succinct tree data structure [6]. Their approach involves decomposing the input tree into smaller units called "mini-trees", which are further divided into "micro-trees." These micro-trees are stored efficiently in a compact precomputed table. By focusing on the connections and links between these subtrees, the entire tree can be represented using the shared roots of the mini-trees. The main result is the following theorem.

▶ **Theorem 5** ([6]). *For a rooted ordered tree with n nodes and a positive integer $1 \le \ell \le n$, one can decompose the trees into subtrees satisfying the following conditions: (1) each subtree contains at most 2ℓ nodes, (2) the number of subtrees is $\mathrm{O}(n/\ell)$, (3) each subtree has at most one outgoing edge, apart from those from the root of the subtree.*

Note that to achieve this, we allow subtrees to share their root nodes, hence the name "tree covering". Theorem 5 applied with $\ell = \log^2 n$ creates a tree covering representation for a tree with n nodes, resulting in $\mathrm{O}(n/\log^2 n)$ mini-trees. The resulting *tree over mini-trees*, with $O(n/\log^2 n)$ nodes, can be represented in $O(n/\log n) = o(n)$ bits. Theorem 5 can be then applied to each mini-tree with $\ell_1 = \frac{1}{6}\log n$, resulting in $O(n/\log n)$ micro-trees. The *mini-tree over micro-trees* is obtained by contracting each micro-tree into a node and adding dummy nodes for micro-trees sharing a common root, and it has $O(\log n)$ vertices, thus, it can be represented by $O(\log\log n)$-bit pointers. The total space for all mini-trees over micro-trees is $O(n\log\log n/\log n) = o(n)$ bits. Micro-trees are stored with two-level pointers in a precomputed table, which occupies $2n + o(n)$ bits. This representation, supplemented by auxiliary data structures requiring only $o(n)$ bits, enables fundamental tree navigation operations, such as accessing the parent, the i-th child, the lowest common ancestor, and many more in $O(1)$ time [6].

2.9 Document listing problems

The document listing problem [22] is, given an array of colors $A[1, n]$ and an interval $[i, j]$ of the indices of the array, to enumerate all distinct colors in the sub-array $A[i, j]$. The problem can be solved in optimal $O(1 + k)$ time where k is the output size (the number of distinct colors in $A[i, j]$), after $O(n)$ time preprocessing for A. Namely, the preprocessing first constructs another array $P[1, n]$ such that $P[i] = j$ if $j < i$ is the largest index such that $A[i] = A[j]$, and $P[i] = -1$ if no such j exists, and then constructs a range minimum query data structure for P. We can consider P to be a representation of linked lists connecting the same colors.

A query for $A[i, j]$ is done as follows. First we find the index m of the minimum value in $P[i, j]$. If $P[m] < i$, we output $A[m]$ and recurse for $A[i, m - 1]$ and $A[m + 1, j]$. If $P[m] \geq i$, we terminate. If we have access to the array P in time t, the query time complexity is $O((1 + k)t)$.

The algorithm is extended so that it works without storing P explicitly [26]. Instead we store a range minimum query data structure for P using $2n + o(n)$ bits. The query algorithm is changed to work without accesses to P. Instead of checking if $P[m] < i$ for finding answers without duplicates, we use a bit array D whose length is the number of possible colors in A, and set $D[c] = 1$ if color c is output. Therefore it can be checked in constant time if a color is already output or not. After outputting all the answers, we clear the bits of D. To do this, we need to keep the output in the memory.

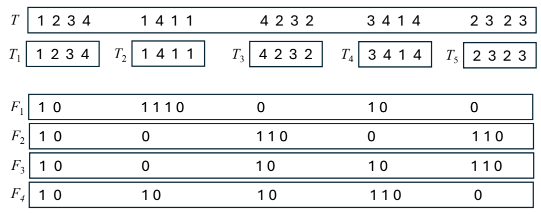
3 Enhancing Generalized Compressed Suffix Trees

3.1 New predecessor data structures

Let $T[1, n]$ be a string on alphabet $\mathcal{A}$ of size k. We give a simpler and more space-efficient index than [2] by omitting support for the access operation, which can be done using a GST. Our new index is summarized as follows.

▶ **Lemma 6.** *Given support for* select *in $f(n, k)$ time on a string S of length n on an alphabet of size k, there is a succinct index using $O(n + k)$ bits that supports* rank*, predecessor, and successor for any character in $O(f(n, k)\log\log k)$ time.*

Proof. Since predecessors and successors can be computed using rank and select operations as shown in Section 2.2, it is enough to show that we can compute ranks in $O(f(n, k)\log\log k)$ time. We use a similar approach to [2]. We partition T into blocks $T_1, T_2, \ldots, T_m$ ($m = \lceil n/k\rceil$) of length k each. Let F_c be a bit-vector storing frequencies of $c \in \mathcal{A}$ for each block using unary codes. That is, $F_c = 1^{f_c(1)}01^{f_c(2)}0 \cdots 1^{f_c(m)}0$ where $f_c(i)$ is the number of occurrences of c in



T 1 2 3 4 1 4 1 1 4 2 3 2 3 4 1 4 2 3 2 3
T_1 1 2 3 4
T_2 1 4 1 1
T_3 4 2 3 2
T_4 3 4 1 4
T_5 2 3 2 3
F_1 1 0 1 1 1 0 0 1 0 0
F_2 1 0 0 1 1 0 0 1 1 0
F_3 1 0 0 1 0 1 0 1 1 0
F_4 1 0 1 0 1 0 1 1 0 0

Figure 2 An example of our rank/select indexes of Lemma 6. The array T in the figure is the same as the array A in Figure 1.

T_i. The total length of F_c for all $c \in \mathcal{A}$ is $\sum_{c=1}^{k}(f_c+m) \le 2n+2k$ where $f_c = \sum_{i=1}^{m} f_c(i)$. We can compute the number of c's in $T_1, \dots, T_i$ in $f(n,k)$ time by $\mathrm{select}_0(F_c, i) - i$. We can also obtain the block containing the j-th occurrence of c in $f(n,k)$ time by $\mathrm{select}_1(F_c, j) - j + 1$. Therefore given an index j of T, we can obtain the number p of occurrences of c in blocks before the block containing j in $f(n,k)$ time by $p = \mathrm{select}_0(F_c, b) - b$ where $b = \lceil j/k \rceil$ is the index of the block. Then the problem is reduced to computing the rank of c in block T_b.

We assume that for each character $c \in \mathcal{A}$, we can compute select in τ time, and show that with this assumption we can compute rank in a block in $\mathrm{O}(\tau \log\log k)$ time using an auxiliary data structure of $\mathrm{O}(n)$ bits.

Let b be the index of the block containing $T[j]$ ($b = \lceil j/k \rceil$). If $f_c(b) \le \log k$, we can compute rank in $\mathrm{O}(\tau \log\log k)$ time by simply doing a binary search using select operations. If $f_c(b) > \log k$, we choose every $\log k$ values of the positions of c's in block T_b, and construct the y-fast trie [30]. The space is $\mathrm{O}\left(\left\lceil \frac{f_c(b)}{\log k} \right\rceil \cdot \log k\right)$ bits for each character c. Because there are less than $k/\log k$ such c's, the total space for block T_b is $\mathrm{O}\left(\sum_c \left(\frac{f_c(b)}{\log k} + 1\right) \log k\right) = \mathrm{O}(k)$ bits, and the total space for all the blocks is $\mathrm{O}(n)$ bits. Using the y-fast trie, we can obtain the predecessor among the samples in $\mathrm{O}(\log\log k)$ time, and then by a binary search we can obtain the true predecessor in $\mathrm{O}(\tau \log\log k)$ time. Successors and ranks are similarly computed in $\mathrm{O}(\tau \log\log k)$ time. ◀

3.2 Finding Closest Colored Suffixes

Given a suffix t of a string S_i and the index j of another string S_j, we compute a suffix of S_j that has the maximum LCP value with s.

We use the same framework as [18]. Let p and q be the lex-order of suffix s of S_i and the closest suffix t of S_j to t in the suffix array for $\mathcal{S}$, respectively. Then there are no suffixes of S_j whose lex-order in $\mathcal{S}$ is between p and q. Let $A[1, n+k]$ be an array of integers such that $A[i] = d$ if lexicographically the i-th suffix in $\mathcal{S}$ belongs to S_d. That is, $d = \mathrm{rank}_1(D, \mathrm{SA}_{\mathcal{S}}[i]) + 1$ is the ID of the string containing the suffix, as shown in Section 2.3. Then it holds that $A[q] = d$ and for all i between p and q exclusive $A[i] \ne d$. That is, q is either the predecessor or the successor of p representing d. In their paper, the data structure of [4] is used for computing predecessors and successors. We replace it with our predecessor data structure of Lemma 6. To use it, we need to give an algorithm for computing $\mathrm{select}_d(A, i)$.

▶ **Lemma 7.** *We can compute* $\mathrm{select}_d(A, i)$ *in* $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ *time.*

Proof. We can compute $\mathrm{select}_d(A, i)$ by the following formula.

$$\mathrm{select}_d(A, i) := \mathrm{ISA}_{\mathcal{S}}[\mathrm{SA}_{S_d}[i] + \mathrm{select}_1(D, d)]$$

This takes clearly $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time. ◀

To compute the closest suffix we first compute the predecessor q_1 and the successor q_2 in A, which correspond to suffixes s_1 and s_2 of S_j, then we decide which one is closer to t. We compute the lengths of $\mathrm{LCP}(s_1, t)$ and $\mathrm{LCP}(s_2, t)$ and choose the larger one (if they are the same, we choose one arbitrarily). The length of $\mathrm{LCP}(s_1, t)$ is computed as follows.

- Find the leaves of the suffix tree of $\mathcal{S}$ corresponding to s_1 and t. They are lexicographically the q_1-th and p-th suffixes.
- Find the lowest common ancestor v between the leaves.
- Compute the string depth of v.

All this can be done in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ total time. The length of $\mathrm{LCP}(s_2, t)$ is computed similarly.

Now we give a proof of Theorem 1.

Proof. We construct compressed suffix trees for each of $S_1, S_2, \ldots, S_k$, and the compressed suffix tree for their concatenation $\mathcal{S}$. The compressed suffix arrays of S_i have size $\mathrm{SIZE}_{\mathrm{SA}}(|S_i|, \sigma)$ for $i = 1, 2, \ldots, k$ and the total size is $\mathrm{SIZE}_{\mathrm{SA}}(n, \sigma)$ bits. The compressed suffix array of $\mathcal{S}$ uses additional $\mathrm{SIZE}_{\mathrm{SA}}(n, \sigma)$ bits. The suffix tree structures are stored in $\mathrm{O}(n + k)$ bits. Therefore the total space is $\mathrm{SIZE}_{\mathrm{SA}}(2n, \sigma) + \mathrm{O}(n + k)$ bits. We store the bit-vector D of the lengths of the strings in $n + k + \mathrm{o}(n + k)$ bits. We also construct the predecessor data structure of Lemma 6 for the document array A storing ID's of suffixes in $\mathcal{S}$. Note that we do not store the document array A explicitly; each entry of A is computed in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time using the compressed suffix arrays of $\mathcal{S}$ (see Section 2.3).

For a query, we compute the global rank p of t in $\mathcal{S}$ using Equation 1 in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time. Then we compute the predecessor q_1 and the successor q_2 in A such that $A[q_1] = A[q_2] = j$ in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log\log k)$ time using Lemma 6 where $f(n, k) = \mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$. We compute the LCP's between the suffix at q_1 and t and the suffix at q_2 and t in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time, and choose the suffix with longer LCP. The query complexity is $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log\log k)$ in total. ◀

3.3 Succinct index for weighted level ancestor queries

We prove Theorem 2. The solution of Kopelowitz and Lewenstein [15] for weighted level ancestor queries uses $\mathrm{O}(n \log n)$ bit space and supports a query in $\mathrm{O}(\log\log n)$ time. Though there are improved data structures [10, 3] that support a query in constant time for a suffix tree, they also use $\mathrm{O}(n \log n)$ bit space.

We give a succinct ($\mathrm{o}(n)$ bit) index for weighted level ancestors which can be used together with a (generalized) compressed suffix tree. The query time is $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \log\log n)$. The basic idea is to decompose the tree into small components using the tree covering [6] so that each component is a connected subgraph, called a mini tree, of the tree with $\mathrm{O}(\log^2 n)$ nodes. The number of components is $\mathrm{O}(n/\log^2 n)$. We create a tree, called tree over mini trees, connecting the components and use the $\mathrm{O}(n \log n)$ bit data structure (in our application we use $\mathrm{O}((n/\log^2 n)\log n) = \mathrm{O}(n/\log n)$ bits) for this new tree.

Given a query $\mathrm{WA}(v, w)$, we first find the mini tree containing v and check if the root of the mini tree has weight smaller than w. If so, the answer is inside the mini tree, and we can find it by a binary search using unweighted level ancestor queries. Because the mini tree contains $\mathrm{O}(\log^2 n)$ nodes, the path length from v to the mini tree root is also $\mathrm{O}(\log^2 n)$. Therefore the binary search takes $\mathrm{O}(\log\log n)$ steps and at each step we compute the string depth of a node in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time. If the root of the mini tree has weight larger than w, we find the nearest mini tree whose root has weight smaller than w. This is done by using the data structure of [15] in $\mathrm{O}(\log\log n)$ time. Finally we find the answer - the right node of this mini tree through binary search. The total time complexity is $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log\log n)$ and the space complexity is $\mathrm{o}(n)$ bits in addition to that of the compressed suffix tree.

3.4 Document listing problem in a rooted tree

We generalize the document listing problem from arrays to rooted trees, that is, given two nodes of the tree with an ancestor-descendant relation we output all the distinct colors appearing on the path connecting them. A single node can have one color, multiple colors, or no color at all.

To solve this problem, we use the heavy-path decomposition of the rooted tree [27]. That is we decompose the tree with n nodes into heavy paths so that any root-to-leaf path intersects $\mathrm{O}(\log n)$ heavy paths.

First we give an $\mathrm{O}(n+k)$ bit representation of heavy paths. See Figure 3 for an example. We assume the tree topology is given as a BP sequence. Its length is at most $4(n+k)$ because the tree has $n+k$ leaves and at most $n+k-1$ internal nodes. We encode heavy edges using bit-vector called "heavy". We set heavy$[i]=1$ iff the edge between the node with preorder i and its parent is heavy. We mark a node if it is the head of a heavy path using a bit-vector of length n, called "head". Then by a nearest marked ancestor query we can find the preorder of the head of the heavy path containing a given node and the distance between them in constant time. We can give a total order for the heavy paths by the preorders of the head nodes. Using the bit-vector "head", we can give numbers from 1 to $m \leq n$ to heavy paths. We can also store the lengths of the heavy paths using unary codes in at most $m+n$ bits. This is encoded in "path-len". In Figure 3, the heavy path from node a to the 8-th leaf from the left has the head at node a, and it is the first heavy path because the head has the smallest preorder among all the five heavy paths. Its length 5 is encoded by the unary code at the beginning of the bit-vector "path-len".

Next we give an encoding of the colors of the nodes. For each heavy path, we encode the number of colors in each node using unary codes. The first heavy path has nodes a, c, e, f plus a leaf and the number of colors of them is 4, 2, 0, 1, and 0, respectively. The numbers are encoded in bit-vector "multi" using $n+u$ bits where $u \leq n$ is the total number of colors. For other heavy paths, the numbers of colors are stored similarly. The array named "color" stores the colors of nodes. Note that this array is constructed in the preprocessing phase and deleted after we construct the range minimum query data structure for array P (see Section 2.9). The range minimum query data structure uses $\mathrm{O}(n+u)$ bits. Note that in the original algorithm for arrays, we set $P[i]=-1$ if there are no values $j<i$ such that $A[j]=A[i]$, whereas in our algorithm for trees, we set $P[i]=j$ if there exists $j<i$ such that color$[j]=$ color$[i]$ and the node with color$[j]$ is the nearest such ancestor of the node with color$[i]$. In the example, for the heavy path containing nodes g, h, and a leaf, the colors of the nodes in the path are stored in color$[8,9]$, and the corresponding P values are $3,4$ because the root node has colors $3,4$ and therefore we store the indices of P storing the same colors.

A document listing query is done as follows. We first give an algorithm for the case P is explicitly stored. We are given the head h and the tail t nodes of a sub-path p on a root-to-leaf path. First we partition the path into a set of heavy paths. This is done by first obtaining the nearest marked node of the tail node, that is, the head of the heavy path containing t. We go to the parent of the head node and repeat this process until we find the heavy path containing h. This is done in $\mathrm{O}(\log n)$ time because the sub-path p may contain $\mathrm{O}(\log n)$ heavy paths. Then for each heavy path which has an overlap with p, we find the minimum value $P[i]$ and choose the minimum among those values. We check whether the value of $P[i]$ is smaller than the index of h in the array color, and if it is, we output its color and continue the process. The first minimum value $P[i]$ is obtained in $\mathrm{O}(\log n)$ time, then we divide the heavy path containing $P[i]$ into two. To efficiently output all distinct

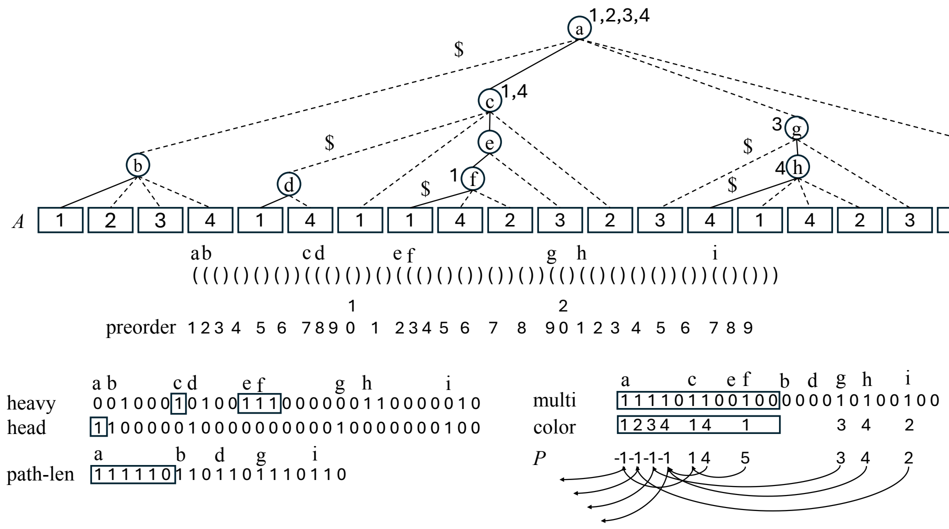


Figure 3 A data structure for document listing problem in a rooted tree. Node c has colors 1,4 because there are two suffixes of S_1 and S_4 ending at the node. The heavy-path from the node consists of nodes a, c, e, f, and a leaf, and it is represented by the boxes in the bit-vectors.

colors, we maintain divided paths using a Fibonacci heap [9]. Because $\mathrm{O}(\log n + z)$ values are stored in the Fibonacci heap where z is the output size, the query algorithm runs in $\mathrm{O}(\log n + z \log(z + \log n))$ time using $\mathrm{O}(\log n(z + \log n))$ bit space.

We modify the algorithm for the case where P is represented only through its range minimum data structure. We compute the color of a suffix using a GST. In this case, we use a similar algorithm to the one described in Section 2.9 using a bit array of length k to mark output colors. First we obtain $\mathrm{O}(\log n)$ paths representing p. Then for each path, we find the position i of the minimum value of P. We compute the color color$[i]$ in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time using the GST. If this color was not found yet, we output it and divide the path into two. It is not necessary to find the minimum of the minimum values because duplication is checked by a different mechanism. It is also not necessary to store the paths in a Fibonacci heap. The time complexity is $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}}(\log n + z))$ and the working space is $\mathrm{O}(\log n(\log n + z))$ bits.

4 Application to Suffix-prefix Matching

In the Suffix-Prefix problem we are given a set of strings $S_1, \ldots, S_k$. We want to preprocess this set of strings so that given $i, j \in [1, k]$ we can answer query "what is the length of the longest suffix of S_i that is also a prefix of S_j" fast.

For any $i, j \in [1, k]$, we define $\mathrm{SPL}_{i,j}$ as the longest string that is both a suffix of S_i and a prefix of S_j. We consider the following variants [18]:

- One-to-One(i, j): output $\mathrm{SPL}_{i,j}$.
- One-to-All(i): output $\mathrm{SPL}_{i,j}$ for every $j \in [1, k]$.
- Report(i, ℓ): output all distinct $j \in [1, k]$ such that $\mathrm{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- Count(i, ℓ): output the number of distinct $j \in [1, k]$ such that $\mathrm{SPL}_{i,j} \geq \ell$, where $\ell \geq 0$ is an integer.
- Top(i, K): output K distinct $j \in [1, k]$ with the highest values of $\mathrm{SPL}_{i,j}$, breaking ties arbitrarily.

Table 2 Time complexities for suffix-prefix problems. An existing solution uses $\mathrm{O}(n \log n)$ bits of space, while ours uses $\mathrm{SIZE}_{\mathrm{SA}}(2n, \sigma) + \mathrm{O}(n + k)$ bits of space. Typical values of $\mathrm{SIZE}_{\mathrm{SA}}$ and $\mathrm{TIME}_{\mathrm{SA}}$ are $n \log \sigma + \mathrm{O}(n)$ and $\mathrm{O}(\log n)$, respectively. The term z in the time complexity of the Report and Count queries is the output size of the Report query.

Query	Time ([18])	Time (ours)
One-to-One(i, j)	$\mathrm{O}(\log \log k)$	$\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log \log n)$
One-to-All(i)	$\mathrm{O}(k)$	$\mathrm{O}(k \cdot \mathrm{TIME}_{\mathrm{SA}} \cdot \log \log n)$
Report(i, ℓ)	$\mathrm{O}(\log n / \log \log n + z)$	$\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}}(\log n + z))$
Count(i, ℓ)	$\mathrm{O}(\log n / \log \log n)$	$\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}}(\log n + z))$

We give compact data structures for these problems except for Top(i, K). The results are summarized in Table 2.

4.1 Answering One-to-One and One-to-All queries

The base of the data structure consists of suffix trees ST_i of S_i for each $i \in [1, k]$ and a generalized suffix tree ST of the whole set of strings. ST is additionally enhanced with a rank-select queries data structure and the lowest common ancestor queries data structure. A node v of ST is colored j if the string label of v is equal to a suffix of j. Note that a node may have multiple colors.

Using Theorem 1 for the full string S_i and j we obtain the location of the closest suffix U of S_j in ST in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log \log k)$ time. We can convert the global rank of U to the local rank in ST_j in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time. Next, using the lowest common ancestor query for S_i and U in ST we can find the LCP of those two strings, that is the string depth of the lowest common ancestor of the nodes representing them, in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time. Next by using the weighted level ancestor query in ST_j for the leaf representing U and the LCP length we locate the node u of ST_j with the property that every ancestor of S_i in ST marked with color j is also an ancestor of u in ST_j, and every ancestor of u in ST_j is also an ancestor of S_i, in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log \log n)$ time. Thus we reduced the problem of finding the nearest ancestor marked with color j in ST to finding the nearest marked ancestor in ST_j - that is in a situation where all the marks have the same color.

For the topology of each ST_j, we use the nearest marked ancestor data structure [28]. The additional space is $n + \mathrm{o}(n)$ bits for all ST_j's, and the answer is obtained in constant time. In summary, a One-to-One suffix-prefix query is done in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log \log n)$ time.

For a One-to-All query, we naively repeat One-to-One queries for each $j \in [1, k]$. Then the time complexity is $\mathrm{O}(k \cdot \mathrm{TIME}_{\mathrm{SA}} \cdot \log \log n)$.

4.2 Report and Count queries

As shown in [18], Report(i, ℓ) and Count(i, ℓ) are the same as Report$^r(j, \ell)$ and Count$^r(j, \ell)$ for the reversed strings $S_1^r, \ldots, S_k^r$. Let ST^r be the generalized compressed suffix tree of the set of the reversed strings, and let v be the nearest node to the root that is on the path from the root node representing S_j^r (reversed sting S_j) and that has a string depth at least ℓ. We color node u by color i if a suffix of S_i^r (without the terminator) ends at u. Then Report$^r(j, \ell)$ is to output all distinct colors on the path from v to the leaf corresponding to S_j^r. That is, Report$^r(j, \ell)$ corresponds to the document listing problem in a tree. The node v is obtained in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log \log n)$ time using the weighted level ancestor query. We use the algorithm from Section 3.4. Note that the arrays "color" and P are not stored explicitly. By using range minimum queries on P, we obtain only the position in P of the minimum value. To obtain the color, we use the compressed suffix tree. If a node u has color i, then u has an edge labeled \$ and a leaf connected by the edge stores a suffix of S_i^r (nodes f, g, h, and i in Figure 3). If u has multiple colors, we create a child w of u connected by an edge labeled \$ and create a leaf as a child of w for each color (nodes a and c in Figure 3). Since we can obtain the global rank of the suffix using the BP sequence of the generalized suffix tree, we can obtain the color in $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}})$ time. The total time complexity becomes $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}}(\log n + z))$ time. $\mathrm{Count}(j, \ell)$ is done in the same time complexity as $\mathrm{Report}(j, \ell)$.

For Top-K, we can use the observation in [18] that there exists an integer $\ell \in [0, n-1]$ such that $\mathrm{Count}(i, \ell+1) \leq K < \mathrm{Count}(i, \ell)$. Therefore we can solve a Top-K query by a binary search based on the value of $\mathrm{Count}(i, \ell)$. Unfortunately the time for $\mathrm{Count}(j, \ell)$ by our algorithm depends on the value, hence such an algorithm for Top-K is inefficient.

5 Concluding Remarks

This paper has proposed auxiliary data structures to enhance generalized compressed suffix trees (GSTs). By adding $\mathrm{O}(n)$ bits of space, we improved the time complexity for finding the closest colored suffix from $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log n)$ to $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log\log k)$ time, where k is the number of strings and n is the total length of the strings, and $\mathrm{TIME}_{\mathrm{SA}}$ is the time to obtain an entry of the suffix array. We also improved the time complexity for finding weighted level ancestors in a compressed suffix tree from $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log n)$ to $\mathrm{O}(\mathrm{TIME}_{\mathrm{SA}} \cdot \log\log n)$ time. Using these enhanced GSTs, we obtained linear space ($\mathrm{O}(n \log \sigma)$ bits) data structures for suffix-prefix queries for a set of strings. Future work will be to give time-efficient algorithms for Count and Top-K queries using $\mathrm{O}(n)$ bits of space. A challenging open problem is to obtain a weighted level ancestor data structure for suffix trees using $\mathrm{O}(n)$ bits of space supporting a query in constant time.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. The 9th International Symposium on String Processing and Information Retrieval. `doi:10.1016/S1570-8667(03)00065-0`.

2 Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Trans. Algorithms*, 7(4), September 2011. `doi:10.1145/2000807.2000820`.

3 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted Ancestors in Suffix Trees Revisited. In Paweł Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, volume 191 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:15, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.CPM.2021.8`.

4 Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Algorithms*, 11(4), April 2015. `doi:10.1145/2629339`.

5 Arthur L. Delcher, Adam Phillippy, Jane Carlton, and Steven L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, June 2002. `doi:10.1093/nar/30.11.2478`.

6 Arash Farzan and J. Ian Munro. A uniform paradigm to succinctly encode various families of trees. *Algorithmica*, 68(1):16–40, 2014. `doi:10.1007/S00453-012-9664-0`.

7 P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 52(4):552–581, 2005. `doi:10.1145/1082036.1082039`.

8 Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In Bo Chen, Mike Paterson, and Guochuan Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74450-4_41.

9 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987. doi:10.1145/28869.28874.

10 Paweł Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014*, pages 455–466, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

11 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, SODA '06, pages 368–373, USA, 2006. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1109557.1109599.

12 R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

13 R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

14 D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

15 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 565–574, USA, 2007. Society for Industrial and Applied Mathematics. URL: http://dl.acm.org/citation.cfm?id=1283383.1283444.

16 S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999. doi:10.1002/(SICI)1097-024X(199911)29:13\%3C1149::AID-SPE274\%3E3.0.CO;2-O.

17 Stefan Kurtz, Adam Phillippy, Arthur Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven Salzberg. Versatile and open software for comparing large genomes. *Genome biology*, 5:R12, February 2004. doi:10.1186/gb-2004-5-2-r12.

18 Grigorios Loukides, Solon P. Pissis, Sharma V. Thankachan, and Wiktor Zuba. Suffix-Prefix Queries on a Dictionary. In Laurent Bulteau and Zsuzsanna Lipták, editors, *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2023.21.

19 U. Manber and G. Myers. Suffix arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

20 J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

21 J. I. Munro, V. Raman, and S. R. Satti. Space Efficient Suffix Trees. *Journal of Algorithms*, 39:205–222, 2001. doi:10.1006/JAGM.2000.1151.

22 S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 657–666, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545469.

23 G. Navarro and K. Sadakane. Fully-Functional Static and Dynamic Succinct Trees. *ACM Transactions on Algorithms (TALG)*, 10(3):Article No. 16, 39 pages, 2014.

24 R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4), 2007. doi:10.1145/1290672.1290680.

25 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

26 Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discrete Algorithms*, 5(1):12–22, 2007. doi:10.1016/J.JDA.2006.03.011.

27 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

28 Dekel Tsur. Succinct data structures for nearest colored node in a tree. *Information Processing Letters*, 132:6–10, 2018. doi:10.1016/j.ipl.2017.10.001.

29 P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

30 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

31 Wiktor Zuba, Grigorios Loukides, Solon P. Pissis, and Sharma V. Thankachan. Approximate suffix-prefix dictionary queries. In Rastislav Královic and Antonín Kucera, editors, *49th International Symposium on Mathematical Foundations of Computer Science, MFCS 2024, August 26-30, 2024, Bratislava, Slovakia*, volume 306 of *LIPIcs*, pages 85:1–85:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.MFCS.2024.85.