Top Tree Compression of Tries

Philip Bille
Technical University of Denmark, DTU Compute, Denmark
phbi@dtu.dk

Paweł Gawrychowski
University of Wrocław, Poland
gawry@cs.uni.wroc.pl

Inge Li Gørtz
Technical University of Denmark, DTU Compute, Denmark
inge@dtu.dk

Gad M. Landau
University of Haifa, Israel
landau@cs.haifa.ac.il

Oren Weimann
University of Haifa, Israel
oren@cs.haifa.ac.il

Abstract

We present a compressed representation of tries based on top tree compression [ICALP 2013] that works on a standard, comparison-based, pointer machine model of computation and supports efficient prefix search queries. Namely, we show how to preprocess a set of strings of total length n over an alphabet of size σ into a compressed data structure of worst-case optimal size $O(n/\log_\sigma n)$ that given a pattern string P of length m determines if P is a prefix of one of the strings in time $O(\min(m \log \sigma, m + \log n))$. We show that this query time is in fact optimal regardless of the size of the data structure.

Existing solutions either use $\Omega(n)$ space or rely on word RAM techniques, such as tabulation, hashing, address arithmetic, or word-level parallelism, and hence do not work on a pointer machine. Our result is the first solution on a pointer machine that achieves worst-case $o(n)$ space. Along the way, we develop several interesting data structures that work on a pointer machine and are of independent interest. These include an optimal data structures for random access to a grammar-compressed string and an optimal data structure for a variant of the level ancestor problem.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases pattern matching, tree compression, top trees, pointer machine

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2019.4

Related Version Draft of full version available at `https://arxiv.org/abs/1902.02187`.

Funding *Philip Bille*: Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
Inge Li Gørtz: Supported by the Danish Research Council (DFF – 4005-00267, DFF – 1323-00178).
Oren Weimann: Supported by the Israel Science Foundation grant 592/17.

1 Introduction

A *string dictionary* compactly represents a set of strings $S = S_1, \ldots, S_k$ to support efficient *prefix queries*, that is, given a pattern string P determine if P is a prefix of some string in S. Designing efficient string dictionaries is a fundamental data structural problem dating back to the 1960's. String dictionaries are a key component in a wide range of applications in areas such as computational biology, data compression, data mining, information retrieval, natural language processing, and pattern matching.

30th International Symposium on Algorithms and Computation (ISAAC 2019).
Editors: Pinyan Lu and Guochuan Zhang; Article No. 4; pp. 4:1–4:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A key challenge and the focus of most of the recent work is to design efficient *compressed string dictionaries*, that take advantage of repetitions in the strings to minimize space, while still supporting efficient queries. While many efficient solutions are known, they all rely on powerful word-RAM techniques, such as tabulation, address arithmetic, word-level parallelism, hashing, etc., to achieve efficient bounds. A natural question is whether or not such techniques are necessary for obtaining efficient compressed string dictionaries or if simpler and more basic computational primitives such as pointer-based data structures and character comparison suffice.

In this paper, we answer this question to the affirmative by introducing a new compressed string dictionary based on *top tree compression* that works on a standard comparison-based, pointer machine model of computation. We achieve the following bounds: let $n = \sum_{i=1}^{k} |S_i|$ be the total length of the strings in S, let σ be the size of the alphabet, and m be the length of a query string P. Our compressed string dictionary uses $O(n/\log_\sigma n)$ space (space is measured as the number of words and not bits, see discussion below) and supports queries in $O(\min(m \log \sigma, m + \log n))$ time. The space matches the information-theoretic worst-case space lower bound, and we further show that the query time is optimal for any comparison-based query algorithm regardless of the space. Compared to previous work our string dictionary is the first $o(n)$ space solution in this model of computation.

1.1 Computational Models

We consider three computational models. In the *comparison-based model* algorithms only interact with the input by comparing elements. Hence they cannot exploit the internal representation of input elements, e.g., for hashing or word-level parallelism. The comparison-based model is a fundamental and well-studied computational model, e.g., in textbook results for sorting [44], string matching [43], and computational geometry [53]. Modern programming languages and libraries, such as the C++ standard template library, implement comparison-based algorithms by supporting abstract and user-specified comparison functions as function arguments. In our context, we say that a string dictionary is comparison-based if the query algorithm can only access the input string P via single character comparisons of the form $P[i] \leq c$, where c is a character.

In the *pointer machine model*, a data structure is a directed graph with bounded out-degree. Each node contains a constant number of data fields or pointer to other nodes and algorithms must access the data structure by traversing the graph. Hence, a pointer machine algorithm cannot implement random access structures such as arrays or perform address arithmetic. The pointer machine captures linked data structures such as linked-lists and search trees. The pointer machine model is a classic and well-studied model, see e.g. [1, 20, 21, 36, 59].

Finally, in the word RAM model of computation [35] the memory is an array of memory words, that each contain a logarithmic number of bits. Memory words can be operated on in unit-time using a standard set of arithmetic operations, boolean operations, and shifts. The word RAM model is strictly more powerful than the comparison-based model and the pointer-machine model and supports random access, hashing, address arithmetic, word-level parallelism, etc. (these are not possible in the other models).

The space of a data structure in the word RAM model is the number of memory words used and the space in the pointer machine model is the total number of nodes. To compare the space of the models, we assume that each field in a node in the pointer machine stores a logarithmic number of bits. Hence, the total number of bits we can represent in a given space in both models is within a constant factor of each other.

1.2 Previous work

The classic textbook string dictionary solution, due to Fredkin [30] from 1960, is to store the *trie* T of the strings in S and to answer prefix queries using a top-down traversal of T, where at each step we match a single character from P to the labels of the outgoing edges of a node. If we manage to match all characters of P then P is a prefix of a string in S and otherwise it is not.

Depending on the representation of the trie and the model of computation we can obtain several combinations of space and time complexity. On a comparison-based, pointer machine model of computation, we can store the outgoing edges of each in a biased search tree [14], leading to an $O(n)$ space solution with query time $O(\min(m \log \sigma, m + \log n))$.

We can compress this solution by merging maximal identical complete subtrees of T [27], thus replacing T by a directed acyclic graph (DAG) D that represents T. This leads to a solution with the same query time as above but using only $O(d)$ space, where d is the size of the smallest DAG D representing T. The size of D can be exponentially smaller than n, but may not compress at all. Consider for instance the case where T is a single path of length n where all edges have the same label (i.e., corresponding to a single string of the same letter). Even though T is highly compressible (we can represent it by the label and the length of the path) it does not contain any identical subtrees and hence its smallest DAG has size $\Omega(n)$.

Using the power of the word RAM model improved representations are possible. Benoit et al. [13] and Raman et al. [54] gave *succinct* representations of tries that achieve $O(n/\log_\sigma n)$ space and $O(m)$ query time, thus simultaneously achieving optimal query time and matching the worst-case information theoretic space lower bounds. These results rely on powerful word RAM techniques to obtain the bounds, such as tabulation and hashing. Numerous trie representations are known, see e.g., [4,5,6,7,8,17,25,33,39,40,52,58,60,61,62], but these all use word RAM techniques to achieve near optimal combinations of time and space.

Another approach is to compress the strings according to various measures of repetitiveness, such as the empirical k-th order entropy [34,45,49,55], the size of the Lempel-Ziv parse [9, 15,22,31,32,41,51], the size of the smallest grammar [23,24,31], the run-length encoded Burrows-Wheeler transform, [46,47,48,56], and others [5,10,11,29,50,57]. The above solutions are designed to support more general queries on the strings, but as noted by Ars and Fischer [5] they are straightforward to adapt to prefix queries. For example, if z is size of the Lempel-Ziv parse of the concatenation of the strings in S, the result of Christiansen and Etienne [22] implies a string dictionary of size $O(z \log(n/z))$ that supports queries in time $O(m + \log^\epsilon n)$. Since z can be exponentially smaller than n, the space is significantly improved on highly-compressible strings. Since $z = O(n/\log_\sigma n)$ in the worst-case, the space is always $O(\frac{n}{\log_\sigma n} \log(\frac{n}{n/\log_\sigma n})) = O(\frac{n \log\log_\sigma n}{\log_\sigma n})$ and thus almost optimal compared to the information theoretic lower bound. Similar bounds are known for the other measures of repetitiveness. As in the case of succinct representations of tries, all of these solutions use word RAM techniques.

1.3 Our results

We propose a new compressed string dictionary that achieves the following bounds:

▶ **Theorem 1.** *Let S be a set of strings of total length n over an alphabet of size σ. On a comparison-based, pointer machine model of computation, we can construct a compressed string dictionary that uses $O(n/\log_\sigma n)$ space and answer queries in $O(\min(m \log \sigma, m{+}\log n))$ time.*

Note that the space bound for Theorem 1 matches the information theoretic lower bound and the time bound matches the classic linear space implementation of tries with biased search trees. The result is the first $o(n)$ space solution in this model of computation. Furthermore, we show that this time bound is optimal.

▶ **Theorem 2.** *For any n, $m \leq n$, and $\sigma \geq 2$, there exists a set S of strings of total length n over an alphabet of size σ such that any comparison-based algorithm that checks if a given pattern P of length m belongs to S needs to perform $\Omega(\min(m \log \sigma, m + \log n))$ comparisons in the worst case.*

Note that Theorem 2 holds regardless of the space used, holds even for weaker membership queries, and only assumes that the algorithm is a comparison-based algorithm. We note that the upper bound holds on a pointer machine with comparisons and additions as arithmetic operations, while the lower bound only assumes comparisons.

1.4 Techniques

In *top tree compression* [18] one transforms a labeled tree T into another tree $\mathcal{T}$ (called a *top tree*) that is of height $O(\log n)$ and represents a hierarchical decomposition of T into connected subgraphs (called *clusters*). Each cluster overlaps with other clusters in at most two nodes. Every leaf in $\mathcal{T}$ corresponds to a cluster consisting of a single edge in T and every internal node in $\mathcal{T}$ corresponds to a merge of two clusters. The top tree $\mathcal{T}$ is then compressed using the classical DAG compression resulting in the *top DAG $\mathcal{TD}$*. The top DAG supports basic navigational queries on T in $O(\log n)$ time, has size $O(n/\log_\sigma n)$, can compress exponentially better than DAG compression, and is never worse than DAG compression by more than a $O(\log n)$ factor [16, 18, 28, 38].

Our main technical contribution is implementing prefix search optimally on the top DAG. To this end, we develop several optimal pointer machine data structures of independent interest:

- A data structure for the *path extraction problem*, that asks to compactly represent an edge-labeled tree T such that given a node v we can efficiently return the labels on the root-to-v path in T. While an optimal solution for this problem can be obtained by plugging in known tools, more specifically a fully persistent queue [37], we believe that our self-contained solution is simpler and elegant.
- A data structure for the *weighted level ancestor* problem, that asks to compactly represent an edge-weighted tree T such that given a node v and a positive number x we can efficiently return the rootmost ancestor of v whose distance from the root is at least x. An immediate implication of our weighted level ancestor data structure is an optimal data structure for the *random access problem* on grammar compressed strings. This improves a SODA'11 result [19] that required word RAM bit tricks.
- A data structure for the *spine path extraction problem*, that asks to compactly represent a top-tree compression $\mathcal{TD}$ such that given a cluster C we can efficiently return the characters of the unique path between the two boundary nodes of C.
- For the lower bound, we show that any algorithm that given a string $P[1, m]$ checks if $\sum_{i=1}^{m} P[i] = 0 \pmod 2$ needs to perform $\Omega(m \log \sigma)$ comparisons in the worst case. We then show that when $n \geq m\sigma^m$ this implies the $\Omega(m \log \sigma)$ bound for our problem and when $n < m\sigma^m$ it implies the $\Omega(m + \log n)$ bound for our problem.

1.5 Roadmap

In Section 2 we recall *top trees* and how a top tree of a tree T is obtained by merging (either vertically or a horizontally) the top trees of two subtrees of T that overlap on a single node. In Section 3 we present a simple randomized Monte-Carlo word RAM solution to the compressed string indexing problem that is the basis of our deterministic pointer machine solutions in the following sections. The solution is based on top trees and efficiently handles horizontal merges (deterministically) and vertical merges (randomized Monte-Carlo). In Section 4 we show how to handle vertical merges deterministically on a pointer machine, and in Section 5 we show that this suffices to achieve the $O(m + \log n)$ query time in Theorem 1. We show a different way to handle vertical merges in Section 6 and horizontal merges in Section 7. In Section 8 we show that these suffice to achieve the $O(m \log \sigma)$ query time in Theorem 1. Due to space constraints we defer the details of the lower bound and all proofs to the full version of the paper.

2 Preliminaries

In this section we briefly review Karp-Rabin fingerprints [42], top trees [3], and top tree compression [18].

2.1 Karp-Rabin Fingerprints

The Karp-Rabin fingerprint [42] of a string x is defined as $\phi(x) = \sum_{i=1}^{|x|} x[i] \cdot c^i \bmod p$, where c is a randomly chosen positive integer, and $2N^{c+4} \leq p \leq 4N^{c+4}$ is a prime. Karp-Rabin fingerprints guarantee that given two strings x and y, if $x = y$ then $\phi(x) = \phi(y)$. Furthermore, if $x \neq y$, then with high probability $\phi(x) \neq \phi(y)$. Fingerprints can be composed and subtracted as follows.

▶ **Lemma 3.** *Let $x = yz$ be a string decomposable into a prefix y and suffix z. Given any two of the Karp-Rabin fingerprints $\phi(x)$, $\phi(y)$ and $\phi(z)$, it is possible to calculate the remaining fingerprint in constant time.*

2.2 Clustering

Let v be a node in T with children $v_1, \ldots, v_k$ in left-to-right order. Define $T(v)$ to be the subtree induced by v and all proper descendants of v. Define $F(v)$ to be the forest induced by all proper descendants of v. For $1 \leq s \leq r \leq k$ let $T(v, v_s, v_r)$ be the connected component induced by the nodes $\{v\} \cup T(v_s) \cup T(v_{s+1}) \cup \cdots \cup T(v_r)$.

A *cluster* with *top boundary node* v is a connected component of the form $T(v, v_s, v_r)$, $1 \leq s \leq r \leq k$. A *cluster* with *top boundary node* v and *bottom boundary node* u is a connected component of the form $T(v, v_s, v_r) \setminus F(u)$, $1 \leq s \leq r \leq k$, where u is a node in $T(v_s) \cup \cdots \cup T(v_r)$. We denote the top boundary node of a cluster C by $\text{top}(C)$. Clusters can therefore have either one or two boundary nodes. For example, let $p(v)$ denote the parent of v then a single edge $(v, p(v))$ of T is a cluster where $p(v)$ is the top boundary node. If v is a leaf then there is no bottom boundary node, otherwise v is a bottom boundary node. Nodes that are not boundary nodes are called *internal* nodes. The path between the top and bottom boundary nodes in a cluster C is called the cluster's spine, and the string obtained by concatenating the labels on the spine from top to bottom is denoted $\text{spine}(C)$.

Two edge disjoint clusters A and B whose vertices overlap on a single boundary node can be *merged* if their union $C = A \cup B$ is also a cluster. There are five ways of merging clusters (see Figure 1). Merges of type (a) and (b) are called *vertical* merges (C is then a *vertical*

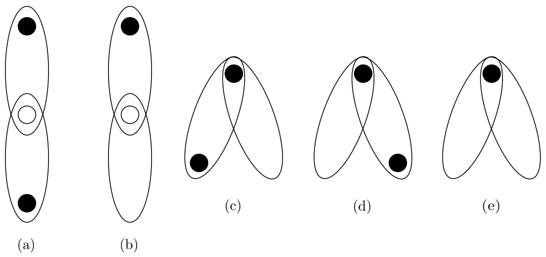

Figure 1 Five ways of merging clusters. The • nodes are boundary nodes that remain boundary nodes in the merged cluster. The ∘ nodes are boundary nodes that become internal (non-boundary) nodes in the merged cluster. Note that in the last four merges at least one of the merged clusters has a top boundary node but no bottom boundary node.

cluster) and can be done only if the common boundary node is not a boundary node of any other cluster except A and B. Merges of type (c),(d), and (e) are called *horizontal* merges (C is then a *horizontal* cluster) and can be done only if at least one of A or B does not have a bottom boundary node.

2.3 Top Trees

A *top tree* $\mathcal{T}$ of T is a hierarchical decomposition of T into clusters. It is an ordered, rooted, labeled, and binary tree defined as follows (see Figure 2(a)–(c)).

- The nodes of $\mathcal{T}$ correspond to clusters of T.
- The root of $\mathcal{T}$ corresponds to the cluster T itself. The top boundary node of the root of $\mathcal{T}$ is the root of T.
- The leaves of $\mathcal{T}$ correspond to the edges of T. The label of each leaf is the label of the corresponding edge (u, v) in T.
- Each internal node of $\mathcal{T}$ corresponds to the merged cluster of its two children. The label of each internal node is the type of merge it represents (out of the five merging options). The children are ordered so that the left child is the child cluster visited first in a preorder traversal of T.

▶ **Lemma 4** (Alstrup et al. [3]). *Given a tree T of size n_T, we can construct in $O(n_T)$ time a top tree $\mathcal{T}$ of T that is of size $O(n_T)$ and height $O(\log n_T)$.*

2.4 Top Dags

Every labeled tree can be represented with a directed acyclic graph (DAG) by identifying identical rooted subtrees and replacing them with a single copy. The *top DAG* of T, denoted $\mathcal{TD}$, is the minimal DAG representation of the top tree $\mathcal{T}$ of T. We can compute it in $O(n_\mathcal{T})$ time from $\mathcal{T}$ [27][1]. Top DAGs have important properties for compression and computation [16, 18, 28, 38]. We need the following optimal worst-case compression bound.

▶ **Lemma 5** (Dudek and Gawrychowski [28]). *Given an ordered tree with n_T nodes over an alphabet of size σ, we can construct a top DAG $\mathcal{TD}$ in $O(n_T)$ time of size $n_{\mathcal{TD}} = O(n_T / \log_\sigma n_T)$.*

[1] Here we use edge labels instead of nodes label. The two definitions are equivalent and edge labels are more natural for tries.

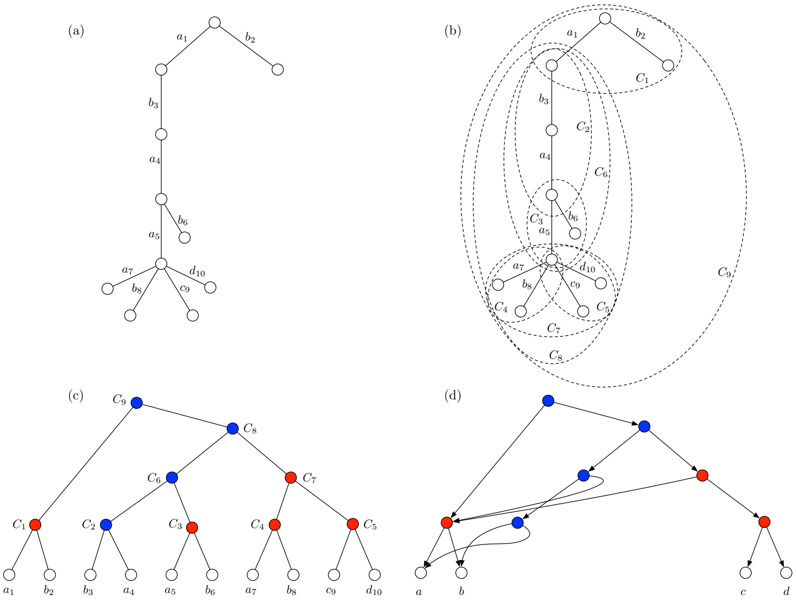

Figure 2 (a) A trie. Each edge label has a subscript to identify the corresponding leaf in the top tree in (c). (b) A hierarchical clustering of (a). (c) The top tree corresponding to (a). Blue nodes are vertical clusters and red nodes are horizontal clusters. (d) The top DAG of (c).

3 A Simple Index

We first present a simple randomized Monte-Carlo word RAM string index, that will be the starting point for our deterministic, comparison-based pointer machine solution in the later sections.

3.1 Data Structure

Let T be the trie of the strings $S = S_1, \ldots, S_k$ and let $\mathcal{TD}$ be the corresponding top DAG of T. Our data structure augments $\mathcal{TD}$ with additional information. For each cluster C in $\mathcal{TD}$ we store the following information.

- If C is a leaf cluster representing an edge e, we store the label of e.
- If C is an internal cluster with left and right child A and B, we store the label of the edge to the *rightmost child* of the top boundary node, the fingerprint $\phi(\mathrm{spine}(C))$, and the length $|\mathrm{spine}(C)|$.

This requires constant space for each cluster and hence $O(n_{\mathcal{TD}})$ space in total.

3.2 Searching

Given a pattern P of length m, we denote the unique node in T whose path from the root matches the longest prefix of P, the

Given a pattern P of length m we find the longest matching prefix of P in T, i.e., the unique node $\mathrm{locus}_T(P)$ in T whose path from the root matches the longest prefix of P, as follows. First, compute and store all fingerprints of prefixes of P in $O(m)$ time and space. By Lemma 3, we can then compute the fingerprint of any substring of P in $O(1)$ time.

Next, we traverse $\mathcal{TD}$ top-down while matching P. Initially, we search for $P[1, m]$ starting at the root of $\mathcal{TD}$. Suppose we have reached cluster C and have matched $P[1, i]$. If $i = m$ we return m. Otherwise ($i < m$) there are three cases:

Case 1: C is a leaf cluster. Let e be the edge stored in C. We compare $P[i+1]$ with the label of e. We return $i+1$ if they match and otherwise i.

Case 2: C is a horizontal cluster. Let A and B be the left and right child of C, respectively. We compare $P[i+1]$ with the label α of the edge to the rightmost child of A. If $P[i+1] \leq \alpha$, we continue the search in A for $P[i+1 \ldots m]$. Otherwise, we continue the search in B for $P[i+1 \ldots m]$.

Case 3: C is vertical cluster. Let A and B be the left and right child of C, respectively. If $|\text{spine}(A)| > m - i$ we continue the search in A for $P[i+1 \ldots m]$. Otherwise, we compare the fingerprint $\phi(\text{spine}(A))$ with $\phi(P[i+1 \ldots i+1+|\text{spine}(A)|])$. If they match, we continue the search in B for $P[i+1+|\text{spine}(A)| \ldots m]$. Otherwise, we continue the search in A for $P[i+1 \ldots m]$.

▶ **Lemma 6.** *The algorithm correctly computes the longest matching prefix of P in T.*

Next consider the running time. We compute all fingerprints of P in $O(m)$ time. Each step of top-down traversal requires constant time and since the depth of $\mathcal{TD}$ is $O(\log n)$ the total time is $O(m + \log n)$. In summary, we have the following theorem.

▶ **Theorem 7.** *Let $S = S_1, \ldots, S_k$ be a set of strings of total length n, and let $\mathcal{TD}$ be the corresponding top DAG for the trie of S. On a word RAM model of computation, we can solve the compressed string indexing problem in $O(n_{\mathcal{TD}}) = O(n/\log_\sigma n)$ space and $O(m + \log n)$ time for any pattern of length m. The solution is randomized Monte-Carlo.*

In the next sections we show how to convert the above algorithm from a randomized algorithm on a word RAM machine into a deterministic algorithm on a pointer machine. We note that Theorem 7 and our subsequent solutions can be extended to other variants of prefix queries, such as *counting queries*, that return the number of occurrences of P. To do so, we store the size of each cluster in $\mathcal{TD}$ and use the above top-down search modified to also record the highest cluster E whose top boundary is $\text{locus}_T(P)$. Since the size of E is the number of occurrences of P, we obtain a solution that also supports counting within the same complexities. From E we can also support *reporting queries*, that return the strings in S with prefix P, by simply decompressing E incurring additional linear time in the lengths of the strings with matching prefix.

4 Spine Extraction

We first consider how to handle vertical clusters (Case 3) deterministically on a pointer machine. The key challenge is to efficiently extract the characters on the spine path of a vertical cluster from top to bottom without decompressing the whole cluster. We will use this to efficiently compute longest common prefixes between spine paths and substrings of P in order to achieve total $O(m + \log n)$ time.

Given the top DAG $\mathcal{TD}$, the *spine path extraction problem* is to compactly represent $\mathcal{TD}$ such that given any vertical cluster C we can return the characters of $\text{spine}(C)$. We require that the characters are reported online and from top-to-bottom, that is, the characters must be reported in sequence and we can stop extraction at any point in time. The goal is to obtain a solution that is efficient in the length of the reported prefix. In the following sections we show how to solve the problem in $O(n_{\mathcal{TD}})$ space and $O(m + \log n)$ total time over all spine path extractions.

We present a new data structure derived from the top DAG called the *vertical top DAG* and show how to use this to extract characters from a spine path. We then use this to compute the longest common prefixes between a spine path and any string and plug this in to the top down traversal in the simple solution from Section 3 to obtain Theorem 1.

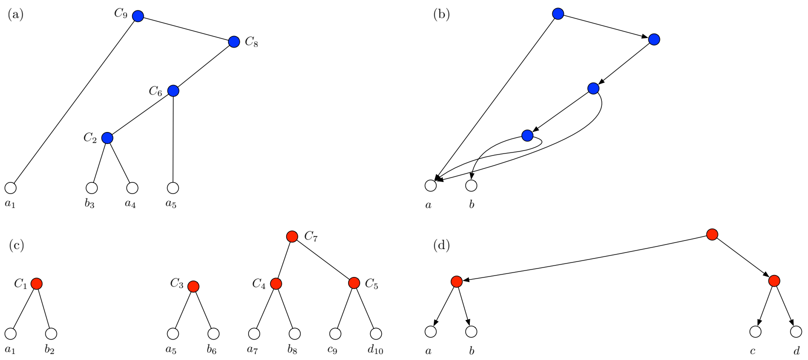

Figure 3 (a) The vertical top forest of the top tree from Figure 2(c). (b) The vertical top DAG of (a). (c) The horizontal top forest of Figure 2(a). (d) The horizontal top DAG of (a).

4.1 Vertical Top Forest and Vertical Top DAG

The *vertical top forest* $\mathcal{V}$ of $\mathcal{T}$ is a forest of ordered, rooted, and labeled binary trees. The nodes in $\mathcal{V}$ are all the vertical clusters of $\mathcal{T}$ and the leaf clusters of $\mathcal{T}$ that correspond to edges of a spine path of some cluster in $\mathcal{T}$. The edges of $\mathcal{V}$ are defined as follows. A cluster C of type (a) with children A and B in $\mathcal{T}$ has two children in $\mathcal{V}$. The left and right children are the unique vertical or leaf descendants of C in $\mathcal{T}$ whose spine path is spine(A) and spine(B), respectively. A cluster C of type (b) with children A and B in $\mathcal{T}$ has a single child in $\mathcal{V}$, which is the unique vertical or leaf descendant of C in $\mathcal{T}$ whose spine path is spine(A). See Figure 3(a). We have the following correspondence between spine paths and subtrees in $\mathcal{V}$.

▶ **Lemma 8.** *Let C be a vertical merge in $\mathcal{V}$ and L be the leaves of $\mathcal{V}(C)$. Then, L are the edges on* spine(C) *and $|\mathcal{V}(C)| = O(|L|)$. Furthermore, the left-to-right ordering of L corresponds to the top-down ordering of the edges on* spine(C)*.*

For instance in Figure 3(a), the descendant leaves of C_6 are b_3, a_4, a_5 in left-to-right ordering corresponding to the edges in the spine of C_6 in Figure 2(b).

The *vertical top DAG* $\mathcal{VD}$ is the DAG obtained by merging identical subtrees of $\mathcal{V}$ according to the DAG compression of $\mathcal{TD}$. See Figure 3(b).

4.2 Spine Extraction

We now show how to solve spine path extraction using the vertical top DAG $\mathcal{VD}$. The key idea is to simulate a depth-first left-to-right order traversal of $\mathcal{V}(C)$ using a recursive traversal of $\mathcal{VD}$. In order to use spine path extraction to search for a pattern we also need to be able to continue the search in some horizontal cluster of the top DAG after extracting characters on the spine. We will therefore define what we call a *vertical exit cluster*, from which we can quickly find the cluster to continue the search from.

Define the *vertical exit cluster*, vexit(C, ℓ), for C at position ℓ, $1 < \ell \leq |\text{spine}(C)|$ to be the lowest common ancestor of leaves $\ell - 1$ and ℓ in $\mathcal{V}(C)$. Intuitively, if we have extracted the first ℓ characters of spine(C), then vexit(C, ℓ) is the cluster such that all leaves in the left subtree have been extracted and only one leaf in the right subtree (corresponding to the ℓth character) has been extracted. Our goal is to implement spine path extraction in time $O(\ell + \text{height}(C) - \text{height}(\text{vexit}(C, \ell)))$. This will yield a telescoping sum when doing multiple extractions.

Our data structure consists of the vertical top DAG $\mathcal{VD}$. We augment each internal cluster by the label of the first edge on its spine path and each leaf cluster by the label of the stored edge. This uses $O(n_{\mathcal{VD}})$ space.

Given a cluster C we implement spine path extraction by simulating a depth-first left-to-right order traversal of $\mathcal{V}(C)$ using a recursive traversal of $\mathcal{VD}$. To extract the first character we return the stored label at C. Suppose we have extracted $\ell - 1$ characters, $1 < \ell \leq |\text{spine}(C)|$. To extract the next character continue the simulated depth-first search until we reach a cluster D in $\mathcal{V}(C)$ whose leftmost leaf is the ℓth leaf of $\mathcal{V}(C)$. Return the character stored at D and the parent of D in $\mathcal{V}(C)$ as $\text{vexit}(C, \ell)$. (Note the parent of D is the cluster visited right before D in the simulated depth-first search.)

By Lemma 8, the algorithm correctly solves spine path extraction and the total time to extract ℓ characters is $O(\ell + \text{height}(C) - \text{height}(\text{vexit}(C, \ell)))$. We need a stack to keep track of the current search path in the traversal using $O(\text{height}(\mathcal{V}(C))) = O(\log n_{\mathcal{T}}) = O(n_{\mathcal{TD}})$ space. In summary, we have the following lemma.

▶ **Lemma 9.** *Let $\mathcal{VD}$ be the vertical top DAG. We can represent $\mathcal{VD}$ in $O(n_{\mathcal{VD}})$ space such that given a vertical cluster C, we can support spine path extraction on C in $O(\ell + \text{height}(C) - \text{height}(\text{vexit}(C, \ell)))$ time, where ℓ is the length of the extracted prefix of* spine(C).

Note that we can use Lemma 9 to compute the longest common prefix of spine(C) and any string by reporting the characters on the spine path from top-to-bottom and comparing them with the string until we get a mismatch. This uses $O(\ell + 1 + \text{height}(C) - \text{height}(\text{vexit}(C, \ell + 1)))$ time, where ℓ is the length of the longest common prefix.

5 An $O(m + \log n)$ Time Solution

We now plug in our spine path extraction algorithm from Section 4 into the simple algorithm from Section 3.

Define the *horizontal entry cluster* for a vertical cluster C, denoted hentry(C), to be the highest horizontal cluster or leaf cluster in $\mathcal{T}(C)$ that contains all edges from top(C) to children within C. For a horizontal cluster or a leaf the horizontal exit cluster is the cluster itself. Note hentry(C) is the highest horizontal cluster or leaf cluster on the path from C to the leftmost leaf of C.

Our data structure consists of the data structures from Section 3 without fingerprints and Section 4. This uses $O(n_{\mathcal{TD}})$ space. To search for a string P of length m, we use the same algorithm as in Section 3, but with the following new implementation of the vertical merges.

Case 3: C is vertical cluster. Recall we have reached a vertical cluster C and have matched prefix $P[1, i]$. We check if the first character on spine(C) matches $P[i + 1]$. If it does not, we continue the algorithm from hentry(C). If it does, we extract characters from spine(C) in order to compute the length ℓ of the longest common prefix of spine(C) and $P[i + 1, m]$ and the corresponding vertical exit cluster $E = \text{vexit}(C, \ell + 1)$. Let B be the right child of E in $\mathcal{TD}$. We traverse the leftmost path from B to find hentry(B) and continue the search for $P[i + \ell + 1, m]$ from there.

▶ **Lemma 10.** *The algorithm correctly computes the longest matching prefix of P in T.*

Consider the time used in a vertical step from a cluster C . The time to compute the longest common prefix computation extracting ℓ characters and walking to the corresponding horizontal entry cluster hentry(vexit(C, ℓ)) is $O(\ell + h(C) - h(\text{vexit}(C, \ell) + h(\text{vexit}(C, \ell)) - h(\text{hentry}(\text{vexit}(C, \ell))) = O(\ell + h(C) - h(\text{hentry}(\text{vexit}(C, \ell)))$. Hence, if we have z vertical steps from clusters $C_1, \ldots, C_z$ extracting $\ell_1, \ldots, \ell_z$ characters ending in $E_i = \text{hentry}(\text{vexit}(C_i, \ell_i))$, respectively, we use time

$$\sum_{i=1}^{z} O(\ell_i + h(C_i) - h(E_i)) = O\left(\sum_{i=1}^{z} \ell_i + h(C_1) - h(E_z)\right) = O(m + \log n_{\mathcal{T}}) \ .$$

This follows from the fact that $C_1, \ldots, C_z$ and $E_1, \ldots, E_z$ all lie on the same root-to-leaf path in $\mathcal{T}$ and that $h(E_i) \geq h(C_{i+1})$. As in Section 3, the total time used at horizontal merges is $O(\log n_T)$, as $E_1, \ldots, E_z$ all lie on the same root-to-leaf path in $\mathcal{T}$ and we only walk down in the tree during the horizontal merges. This concludes the proof of the $O(m + \log n)$ query time in Theorem 1.

6 Spine Path Extraction with Constant Overhead

Next, we show how to achieve the $O(m \log \sigma)$ query time in Theorem 1. Our current solutions for horizontal merges (Case 2) from Section 3 and vertical merges (Case 3) from Section 5 both require $\Omega(m + \log n)$ and hence we need new techniques for both cases to achieve the $O(m \log \sigma)$ time bound. We consider vertical merges in this section and horizontal merges in the next section.

In this section, we improve the total time used on spine extraction to optimal $O(m)$ time. To do so we first introduce and present a novel solution to a new path extraction problem on trees in Section 6.1 and then show how to use this to extract characters from the spine in Section 6.2.

6.1 Path Extraction in Trees

Given a tree T with n nodes, the *path extraction problem* is to compactly represent T such that given a node v we can return the nodes on the path from the root of T to v in constant time per node. We require that the nodes are reported online and from top-to-bottom, that is, the nodes must be reported in sequence and we can stop the extraction at any point in time. The ordering of the nodes from top to bottom is essential. The other direction (from v to the root) is trivial since we can simply store parent pointers and traverse that path using linear space and constant time per node. If we allow word RAM tricks then we can easily solve the problem in the same bounds by using an existing *level ancestor data structure* [2, 12, 26]. We present an optimal solution that does not use word RAM tricks and works on a pointer machine. As mentioned in the introduction, an optimal solution can be also obtained by plugging in known tools, but we believe that our method is simpler and elegant.

Let depth(v) and height(v) be the distance from v to the root and to deepest leaf in v's subtree, respectively. Decompose T into a top part T_{top} consisting of nodes v, such that depth$(v) \leq$ height(v), and a bottom part T_{bot} consisting of the remaining nodes. For each leaf u in T_{top} we store the path from the root of T_{top} to u explicitly in a linked list sorted by increasing depth. (see Figure 4). Note that multiple copies of the same node may be stored across different lists. Each such path to a leaf u uses $O(\text{depth}(u))$ space, and hence the total space for all paths in T_{top} is

$$\sum_{u \text{ a leaf in } T_{\text{top}}} \text{depth}(u) \quad \leq \quad \sum_{u \text{ a leaf in } T_{\text{top}}} \text{height}(u) \quad = \quad O(n) \ ,$$

where the first equality follows by definition of the decomposition and the second follows since the longest paths from a descendant leaf in $T(u)$ to a leaf u in T_{top} are disjoint for all the leaves u in T_{top}. For all internal nodes in T_{top} we store a pointer to a leaf below it. For

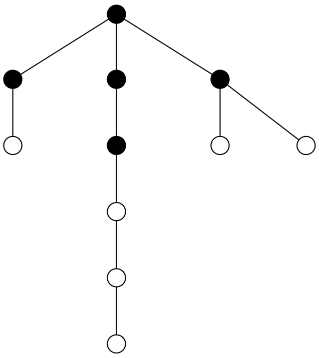

Figure 4 The tree decomposition for path extraction. The black nodes are the nodes in T_{top} and the white nodes are the nodes in T_{bot}. The three root-to-leaf paths are stored as three linked lists sorted by increasing depth. The total size of the lists is $2 + 3 + 2 = 7$.

all nodes v in T_{bot} we store a pointer to the unique ancestor v that is a leaf in T_{top}. We answer a path extraction query for a node v as follows. If v is in T_{top} we follow the leaf pointer and output the path stored in this leaf from the root until we reach v. If v is in T_{bot} we jump to the unique ancestor leaf u of v in T_{top}. We extract the path from the root to u, while simultaneously following parent pointers from v until we reach u storing these nodes on a stack. That is, each time we extract a node from the root-to-u path we follow a parent pointer and put the next node on the stack. We stop pushing nodes to the stack when we reach u. When we have output all nodes from the root to the leaf in T_{top} we output the nodes from the stack. Since $\text{depth}(u) \leq \text{height}(u)$ the path from the root to u is at least as long as the path from v to u plus 1. Therefore, the whole path is extracted. We spend $O(1)$ time per node and hence we have the following result.

▶ **Lemma 11.** *Given a tree T with n nodes, we can solve the path extraction problem in linear space and preprocessing and constant time per reported node.*

6.2 Optimal Spine Path Extraction

We plug the path extraction solution into our depth-first search traversal of the vertical top DAG $\mathcal{VD}$ to speed up spine extraction and longest common prefix computation. Recall that given a vertical cluster C, our goal is to simulate a depth-first left-to-right order traversal of the subtree $\mathcal{V}(C)$ using the vertical top DAG $\mathcal{VD}$.

We construct the *left-path suffix forest* L of $\mathcal{VD}$ as follows. The nodes of L are the nodes of $\mathcal{VD}$. If C has a left child A in $\mathcal{VD}$ then A is the parent of C in L. Hence, any leftmost path in $\mathcal{VD}$ corresponds to a path from a node to an ancestor of the node in L. We now store L with the path extraction data structure from Lemma 11. We implement the depth-first traversal as before except that whenever the traversal reaches an unexplored cluster C' in $\mathcal{V}(C)$ we begin path extraction for that cluster corresponding to the path from C' to the leftmost descendant leaf $\hat{C}$. We extract the leaf $\hat{C}$ and then continue the depth-first traversal from there. Hence, the current search path of the depth-first traversal is partitioned into an alternating sequence of leftmost paths and right edges. Whenever we need to go up on a left edge in the traversal we extract the next node for the corresponding path extraction instance.

To extract the topmost ℓ characters of $\text{spine}(C)$ we now use constant time to find the leftmost descendant leaf of $\mathcal{V}(C)$ and then $O(\ell)$ time to traverse the first ℓ leaves. Hence, we improve the time from $O(\text{height}(\mathcal{V}(C)) + \ell)$ to $O(\ell)$. At any point during the traversal we maintain ongoing path extractions instances along the current search path. The stacks each of these need are of size at most linear in the length of their corresponding subpath of the search path and hence this requires at most $O(\log n_{\mathcal{VD}})$ extra space.

▶ **Lemma 12.** *We can represent the vertical top DAG $\mathcal{VD}$ in $O(n_{\mathcal{VD}})$ space such that given a vertical cluster C, we can support spine path extraction on C in $O(\ell)$ time, where ℓ is the length of the extracted prefix of* $\mathrm{spine}(C)$*.*

7 Horizontal Access

We now show how to efficiently handle horizontal merges (Case 2). In the simple algorithm from Section 3 we use constant time at each horizontal merge leading to an $O(\log n_{\mathcal{T}})$ total time solution. Since we cannot afford $O(\log n_{\mathcal{T}})$ time we instead show how to handle all horizontal merges in $O(m \log \sigma)$ time. The key idea is to convert the problem into a variant of the *random access problem* for grammar compressed strings, and then design a linear-space logarithmic-query solution to the random access problem. We describe the random access problem in Section 7.1 and present our solution to it in Section 7.2, we introduce the horizontal top DAG in Section 7.3, and define and solve the *horizontal access problem* in Section 7.4.

7.1 Grammars and Random Access

Grammar compression replaces a long string S by a small context-free grammar (CFG) $\mathcal{G}$. We view a grammar $\mathcal{G}$ as a DAG, where each node is a grammar symbol and each rule defines directed ordered edges from the righthand side to the lefthand side. Given a node C in $\mathcal{G}$, we define $T(C)$ to be the parse tree rooted at C and $S(C)$ to be the string consisting of the leaves of $T(C)$ in left-to-right order. Note that given a rule $C \rightarrow C_1C_2 \ldots C_k$ we have that $S(C) = S(C_1) \cdot S(C_2) \cdots S(C_k)$, where $\cdot$ denotes concatenation. Given a grammar $\mathcal{G}$ representing a string S, the *random access problem* is to compactly represent $\mathcal{G}$ while supporting fast *access queries*, that is, given an index i in S report $S[i]$. Bille et al. [19] showed how to do random access in $O(\log |S|)$ time using $O(n_{\mathcal{G}} \cdot \alpha_k(n_{\mathcal{G}}))$ space[2] on a pointer machine model. Furthermore, given a node C in $\mathcal{G}$, access queries can be supported on the string $S(C)$ in time $O(\log |S(C)|)$.

For our purposes, we need to slightly extend this result to *gapped grammars.* A gapped grammar is a grammar except that each internal rule is now of the form $C \rightarrow C_1g_1C_2 \ldots g_{k-1}C_k$, where g_i is a non-negative integer called the *gap*. The string generated by $\mathcal{G}$ is now $S(C) = S(C_1)\mathtt{0}^{g_1}S(C_2)\cdots S(C_{k-1})\mathtt{0}^{g_{k-1}}S(C_k)$ and hence the resulting string generated is as before except for the inserted gaps of runs of 0's. Note that $|S(C)| = |S(C_1)| + g_1 + |S(C_2)| + \cdots + g_{k-1} + |S(C_k|$. The above random access result is straightforward to generalize to gapped grammars:

▶ **Lemma 13** (Bille et al. [19])**.** *Let S be a string compressed into a gapped grammar $\mathcal{S}$ of size $n_{\mathcal{S}}$. Given a node v in $\mathcal{S}$, we can support random access queries in $S(v)$ in $O(\log(|S(v)|))$ time using $O(n_{\mathcal{S}} \cdot \alpha_k(n_{\mathcal{S}}))$ space. The solution works on a pointer machine model of computation.*

[2] Here $\alpha_k(n)$ for any constant k denotes the inverse of the k^{th} row of Ackermann's function, defined as $\alpha_k(n) = 1 + \alpha_k(\alpha_{k-1}(n))$ so that $\alpha_1(n) = n/2$, $\alpha_2(n) = \log n$, $\alpha_3(n) = \log^* n$, and so on.

7.2 Horizontal Access in Linear Space

Bille et al. [19] further showed that the inverse-Ackermann factor in the space complexity of Lemma 13 can be removed if we assume a word RAM model of computation. In this section we show that this can also be achieved on a pointer machine. To this end, we need to replace a single component in the solution of Bille et al., their *weighted level ancestor* structure. In the weighted level ancestor problem, we are given a tree T on n nodes with positive weights on the edges. For every node $u \in T$, let $d(u)$ be its distance to the root, and let $\text{parent}(u)$ be its parent. Then, the goal is to preprocess T to answer the following *weighted level ancestor* queries: given a non-root node $u \in T$ and a positive number $x \leq d(u)$, find an ancestor v such that $d(v) \geq x$ but $d(\text{parent}(v)) < x$.

Without getting into the proof of Lemma 13, it suffices to say that (1) performing a random access query boils down to performing $O(\log(|S(v)|))$ weighted level ancestor queries, and (2) in order for all these $O(\log(|S(v)|))$ queries to be done in total $O(\log(|S(v)|))$ time, the time for each weighted level ancestor query should be proportional to $\log \frac{d(u)}{d(v)-d(\text{parent}(v))}$. Intuitively, we seek a position on an edge at distance x from the root, and the longer the found edge is the smaller the query time should be. We next show how to achieve such query time using linear space on a pointer machine, implying an inverse-Ackermann factor improvement to Lemma 13.

▶ **Lemma 14.** *A tree T on n nodes can be preprocessed in $O(n)$ space to answer a weighted level ancestor query for a node $u \in T$ and a number x in $O(1 + \log \frac{d(u)}{d(v)-d(\text{parent}(v))})$ time, where v is the found ancestor of u.*

▶ **Corollary 15.** *Let S be a string compressed into a gapped grammar $\mathcal{S}$ of size $n_{\mathcal{S}}$. Given a node v in $\mathcal{S}$, we can support random access queries in $S(v)$ in $\log(|S(v)|)$ time using $O(n_{\mathcal{S}})$ space. The solution works on a pointer machine model of computation.*

7.3 Horizontal Top Tree and Horizontal Top DAGs

Similar to the vertical top forest we define the *horizontal top forest* $\mathcal{H}$ of $\mathcal{T}$ as a forest of ordered and rooted trees that consists of all horizontal clusters of $\mathcal{T}$ and leaves of $\mathcal{T}$ whose top boundary is shared with a horizontal cluster. We define the edges in of C in $\mathcal{H}$ as follows. Let C be a horizontal cluster C with children A and B in $\mathcal{T}$. If A is a horizontal cluster or a leaf then the left child of C is A, and if A is a vertical cluster then the left child of C is $\text{hentry}(A)$. Similarly, the right child of C is either B or $\text{hentry}(B)$. See Figure 3. We have the following property of $\mathcal{H}$.

▶ **Lemma 16.** *Let C be a horizontal merge in $\mathcal{H}$. Then, the leaves of $\mathcal{H}(C)$ are the edges to children of the top boundary node of C and the left-to-right ordering of the leaves correspond to the left-to-right ordering of the children of C in T. All nodes in $\mathcal{H}(C)$ has $\text{top}(C)$ as top boundary node.*

For instance in Figure 3(c) the descendant leaves of C_7 are a_7, b_8, c_9, and d_{10} in left to right ordering corresponding to the edges to the children of $\text{top}(C_7)$. Given the horizontal top forest we define the *horizontal top DAG* $\mathcal{HD}$ as the DAG obtained by merging the subtrees of $\mathcal{H}$ according to the DAG compression of $\mathcal{T}$ into $\mathcal{TD}$.

7.4 Gapped Grammars and Horizontal Access

Let C be an internal cluster in $\mathcal{H}$. The *spine child* of C is the unique child of C that contains the first edge of spine(C). A descendant cluster D of C is a *spine descendant* of C if all clusters on the path from C to D are spine children of their parent. Define the *horizontal exit cluster* for a horizontal cluster C and character α, denoted hexit(C, α), to be the highest cluster in $\mathcal{H}(C)$ that has the unique leaf in $\mathcal{H}(C)$ labeled α as a spine descendant.

Given the horizontal top DAG $\mathcal{HD}$, the *horizontal access problem*, is to compactly represent $\mathcal{HD}$ such that given a horizontal merge C and a character $\alpha \in \Sigma$, we can efficiently determine if top(C) has an edge to a child labeled α within C and if so return the horizontal exit cluster hexit(C, α). In this section, we show how to solve the horizontal access problem in $O(n_{\mathcal{HD}})$ space and $O(\log \sigma)$ time.

The *characteristic vector* of a cluster C is a binary string encoding the labels of edges to children of top(C). More precisely, given a character $\alpha \in \Sigma$ define rank$(\alpha) \in \{1, \ldots, \sigma\}$ as the rank of α in the sorted order of characters of Σ. Also, given a cluster C in $\mathcal{H}$ define rank(C) to be the set of ranks of leaf labels in $\mathcal{H}(C)$. We define the characteristic vector $S(C)$ recursively as follows. If C is a leaf cluster $S(C) = \mathtt{1}$ and if C is an internal cluster with children $C_1, \ldots, C_k$, then $S(C) = S(C_1)\mathtt{0}^{g_1} S(C_2) \cdots S(C_{k-1})\mathtt{0}^{g_{k-1}} S(C_k)$, where $g_i = \min(\text{rank}(C_{i+1})) - \max(\text{rank}(C_i)) + 1$. Note that $|S(C)| \leq \sigma$ for any cluster C. From the definition we have the following correspondence between the characteristic vector and the leaf labels of a cluster.

▶ **Lemma 17.** *Given a cluster C in $\mathcal{H}$ and a character $\alpha \in \Sigma$, α is a leaf label in $\mathcal{H}(C)$ iff $S(C)[\text{rank}(\alpha) - \min(\text{rank}(C))] = \mathtt{1}$.*

Let $R_1, \ldots, R_z$ be the root clusters of the trees in $\mathcal{H}$ and note that if we add a virtual root cluster R as the parent of $R_1, \ldots, R_z$, $\mathcal{H}$ is a gapped parse tree for the string $S = S(R_1) \cdots S(R_z)$. Hence, the horizontal top DAG $\mathcal{HD}$ is a gapped grammar for the same string. By Lemma 17 we can determine if there is an edge labeled α out of top(C) in C using a random access query on the corresponding gapped grammar using time $O(\log |S(C)|) = O(\log \sigma)$. If this edge exists, we can also find hexit(C, α) in the same time using similar ideas. More precisely, we have the following result.

▶ **Lemma 18.** *Given a cluster C in $\mathcal{H}$ and a character $\alpha \in \Sigma$ we can solve the horizontal acces problem in $O(n_{\mathcal{HD}})$ space and $O(\log \sigma)$ time.*

8 An $O(m \log \sigma)$ Solution

We can now plug in the spine extraction from Section 6.2 and the horizontal access from Section 7 into the simple algorithm from Section 3. Define the *vertical entry cluster* for a horizontal cluster C, denoted ventry(C), to be the highest vertical cluster or leaf cluster in $\mathcal{T}(C)$ that contains the first edge on spine(C).

Our data structure consists of the data structure from Section 6.2 for spine path extraction and the data structure from Section 7.3 for horizontal access. Furthermore, we store for each vertical cluster in $\mathcal{TD}$ a pointer to its horizontal entry cluster and for each horizontal cluster a pointer to its vertical entry cluster. In total this uses $O(n_{\mathcal{TD}})$ space.

To search we alternate between horizontal accesses using Lemma 18 and spine path extractions using Lemma 12. Instead of traversals to find entry clusters we jump directly using the new pointers. Specifically, we have the following modified algorithm:

Initially, we search for $P[1,m]$ starting at the root of $\mathcal{TD}$. Suppose we have reached cluster C and have matched $P[1,i]$. If $i = m$ we return m. Otherwise ($i < m$) there are three cases:

Case 1: C is a leaf cluster. Let e be the edge stored in C. We compare $P[i+1]$ with the label of e. We return $i+1$ if they match and otherwise i.

Case 2: C is a horizontal cluster. Compute $E = \mathrm{hexit}(C, P[i+1])$. If $P[i+1]$ does not match return i. Otherwise, continue the search for $P[i+1,m]$ from $\mathrm{ventry}(E)$.

Case 3: C is vertical cluster. We check if the first character on $\mathrm{spine}(C)$ matches $P[i+1]$. If it does not we continue the algorithm from $\mathrm{hentry}(C)$. Otherwise, we extract characters from $\mathrm{spine}(C)$ in order to compute the length ℓ of the longest common prefix of $\mathrm{spine}(C)$ and $P[i+1,m]$ and the corresponding vertical exit cluster $E = \mathrm{vexit}(C, \ell+1)$. Continue the search for $P[\ell+1,m]$ from $\mathrm{hentry}(E)$.

▶ **Lemma 19.** *The algorithm correctly computes the longest matching prefix of P in T.*

Consider the alternating sequence of horizontal accesses and spine extractions. Each time we go from a horizontal access to a spine extraction the current character of P must match the first character on the spine. Hence, each horizontal access is on a distinct character of P and the total number of horizontal accesses is at most m. By Lemma 18 it follows that the total time for horizontal accesses is $O(m \log \sigma)$. Since the sequence is alternating the number of spine extractions is at most $m+1$. Hence, by Lemma 12 the total time for spine extractions is at most $O(m)$. This concludes the proof of the $O(m \log \sigma)$ query time in Theorem 1.

References

1 Peyman Afshani, Lars Arge, and Kasper Green Larsen. Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model. In *Proc. 28th SoCG*, pages 323–332, 2012.

2 Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Proc. 27th ICALP*, pages 73–84, 2000.

3 Stephen Alstrup, Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Maintaining Information in Fully Dynamic Trees with Top Trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005.

4 J-I Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Trans. Soft. Eng.*, 15(9):1066–1077, 1989.

5 Julian Arz and Johannes Fischer. LZ-compressed string dictionaries. In *Proc. 24th DCC*, pages 322–331, 2014.

6 Julian Arz and Johannes Fischer. Lempel–Ziv-78 Compressed String Dictionaries. *Algorithmica*, pages 1–36, 2018.

7 Nikolas Askitis and Ranjan Sinha. Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal*, 19(5):633–660, 2010.

8 Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In *Proc. 17th SPIRE*, pages 159–172, 2010.

9 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Proc. 26th CPM*, pages 26–39, 2015.

10 Djamal Belazzougui, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Alberto Ordónez, Simon J Puglisi, and Yasuo Tabei. Queries on LZ-bounded encodings. In *Proc. 25th DCC*, pages 83–92, 2015.

11 Djamal Belazzougui, Travis Gagie, Simon Gog, Giovanni Manzini, and Jouni Sirén. Relative FM-indexes. In *Proc. 21st SPIRE*, pages 52–64, 2014.

12 Michael A. Bender and Martin Farach-Colton. The Level Ancestor Problem simplified. *Theoret. Comput. Sci.*, 321(1):5–12, 2004.

13 David Benoit, Erik D Demaine, J Ian Munro, Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

14 Samuel W. Bent, Daniel D. Sleator, and Robert E. Tarjan. Biased Search Trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

15 Philip Bille, Mikko B. Ettienne, Inge Li Gørtz, and Hjalte W. Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theor. Comput. Sci.*, 713:66–77, 2018.

16 Philip Bille, Finn Fernstrøm, and Inge Li Gørtz. Tight Bounds for Top Tree Compression. In *Proc. 24th SPIRE*, pages 97–102, 2017.

17 Philip Bille, Inge Li Gørtz, and Frederik Rye Skjoldjensen. Deterministic Indexing for Packed Strings. In *Proc. 28th CPM*, 2017.

18 Philip Bille, Inge Li Gørtz, Oren Weimann, and Gad M. Landau. Tree compression with top trees. *Inf. Comput.*, 243:166–177, 2015. Announced at ICALP 2013.

19 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random Access to Grammar-Compressed Strings and Trees. *SIAM J. Comput.*, 44(3):513–539, 2015. Announced at SODA 2011.

20 Barnard Chazelle. Lower bounds for orthogonal range searching: I. The reporting case. *J. ACM*, 37(2):200–212, 1990.

21 Bernard Chazelle and Burton Rosenberg. Simplex range reporting on a pointer machine. *Comput. Geom.*, 5(5):237–247, 1996.

22 Anders R. Christiansen and Mikko B. Ettienne. Compressed Indexing with Signature Grammars. In *Proc. 13th LATIN*, pages 331–345, 2018.

23 Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.

24 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *Proc. 19th SPIRE*, pages 180–192, 2012.

25 John J Darragh, John G Cleary, and Ian H Witten. Bonsai: a compact representation of trees. *Softw. Pract. Exper.*, 23(3):277–291, 1993.

26 Paul F. Dietz. Finding level-ancestors in dynamic trees. In *Proc. 2nd WADS*, pages 32–40, 1991.

27 Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.

28 Bartlomiej Dudek and Paweł Gawrychowski. Slowing Down Top Trees for Better Worst-Case Compression. In *Proc. 29th CPM*, pages 16:1–16:8, 2018.

29 Andrea Farruggia, Travis Gagie, Gonzalo Navarro, Simon J Puglisi, and Jouni Sirén. Relative suffix trees. *Comput. J.*, 61(5):773–788, 2017.

30 Edward Fredkin. Trie Memory. *Commun. ACM*, 3(9):490–499, 1960.

31 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A Faster Grammar-Based Self-index. In *Proc. 6th LATA*, pages 240–251, 2012.

32 Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th LATIN*, pages 731–742, 2014.

33 Roberto Grossi and Giuseppe Ottaviano. Fast compressed tries through path decompositions. *ACM J. Exp. Alg.*, 19:3–4, 2015.

34 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

35 Torben Hagerup. Sorting and Searching on the Word RAM. In *Proc. 15th STACS*, pages 366–398, 1998.

36 Meng He, J. Ian Munro, and Gelin Zhou. Data Structures for Path Queries. *ACM Trans. Algorithms*, 12(4):53:1–53:32, 2016.

37 Robert Hood and Robert Melville. Real-Time Queue Operation in Pure LISP. *Inf. Process. Lett.*, 13(2):50–54, 1981.

38 Lorenz Hübschle-Schneider and Rajeev Raman. Tree compression with top trees revisited. In *Proc. 14th SEA*, pages 15–27, 2015.

39 Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Compressed double-array tries for string dictionaries supporting fast lookup. *Knowl. Inf. Syst.*, 51(3):1023–1042, 2017.

40 Shunsuke Kanda, Kazuhiro Morita, and Masao Fuketa. Practical implementation of space-efficient dynamic keyword dictionaries. In *Proc. 24th SPIRE*, pages 221–233, 2017.

41 Juha Kärkkäinen and Esko Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd WSP*, pages 141–155, 1996.

42 Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

43 Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

44 Donald Erwin Knuth. *The Art of Computer Programming, Volume 1.* Addison Wesley, 1969.

45 Veli Mäkinen. Compact suffix array—a space-efficient full-text index. *Fundamenta Informaticae*, 56(1-2):191–210, 2003.

46 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. Comput.*, 12(1):40–66, 2005.

47 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of individual genomes. In *Proc. 13th RECOMB*, pages 121–137, 2009.

48 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comp.Bio.*, 17(3):281–308, 2010.

49 Gonzalo Navarro and Veli Mäkinen. Compressed Full-text Indexes. *ACM Comput. Surv.*, 39(1), 2007.

50 Gonzalo Navarro and Nicola Prezza. Universal compressed text indexing. *Theor. Comput. Sci.*, 762:41–50, 2019.

51 Takaaki Nishimoto, I Tomohiro, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. *Disc. App. Math.*, 2019.

52 Andreas Poyias and Rajeev Raman. Improved practical compact dynamic tries. In *Proc. 22nd SPIRE*, pages 324–336, 2015.

53 Franco P Preparata and Se June Hong. Convex hulls of finite sets of points in two and three dimensions. *Communications of the ACM*, 20(2):87–93, 1977.

54 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007.

55 Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proc. 11th ISAAC*, pages 410–421, 2000.

56 Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th SPIRE*, pages 164–175, 2008.

57 Takuya Takagi, Keisuke Goto, Yuta Fujishige, Shunsuke Inenaga, and Hiroki Arimura. Linear-size CDAWG: new repetition-aware indexing and grammar compression. In *Proc. 24th SPIRE*, pages 304–316, 2017.

58 Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. Packed compact tries: A fast and efficient data structure for online string processing. *IEICE Trans. on Fund. Elect., Comm. and Comp. Sci.*, 100(9):1785–1793, 2017.

59 Robert Endre Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of Computer and System Sciences*, 18(2):110–127, April 1979.

60 Kazuya Tsuruta, Dominik Köppl, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic Packed Compact Tries Revisited. arXiv preprint, 2019. `arXiv:1904.07467`.

61 Susumu Yata. Dictionary compression by nesting prefix/patricia tries. In *Proc. 17th Meeting of the Association for Natural Language*, 2011.

62 Naoki Yoshinaga and Masaru Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In *Proc. 25th COLING*, pages 1091–1102, 2014.