**DTU Library**

# Finger Search in Grammar-Compressed Strings

**Bille, Philip; Christiansen, Anders Roy; Cording, Patrick Hagge; Gørtz, Inge Li**

Link back to DTU Orbit

# Finger Search in Grammar-Compressed Strings

Philip Bille
phbi@dtu.dk

Anders Roy Christiansen
aroy@dtu.dk

Patrick Hagge Cording
phaco@dtu.dk

Inge Li Gørtz
inge@dtu.dk

## Abstract

Grammar-based compression, where one replaces a long string by a small context-free grammar that generates the string, is a simple and powerful paradigm that captures many popular compression schemes. Given a grammar, the *random access problem* is to compactly represent the grammar while supporting random access, that is, given a position in the original uncompressed string report the character at that position. In this paper we study the random access problem with the finger search property, that is, the time for a random access query should depend on the distance between a specified index $f$, called the *finger*, and the query index $i$. We consider both a static variant, where we first place a finger and subsequently access indices near the finger efficiently, and a dynamic variant where also moving the finger such that the time depends on the distance moved is supported.

Let $n$ be the size the grammar, and let $N$ be the size of the string. For the static variant we give a linear space representation that supports placing the finger in $O(\log N)$ time and subsequently accessing in $O(\log D)$ time, where $D$ is the distance between the finger and the accessed index. For the dynamic variant we give a linear space representation that supports placing the finger in $O(\log N)$ time and accessing and moving the finger in $O(\log D + \log \log N)$ time. Compared to the best linear space solution to random access, we improve a $O(\log N)$ query bound to $O(\log D)$ for the static variant and to $O(\log D + \log \log N)$ for the dynamic variant, while maintaining linear space. As an application of our results we obtain an improved solution to the longest common extension problem in grammar compressed strings. To obtain our results, we introduce several new techniques of independent interest, including a novel van Emde Boas style decomposition of grammars.

## 1 Introduction

Grammar-based compression, where one replaces a long string by a small context-free grammar that generates the string, is a simple and powerful paradigm that captures many popular compression schemes including the Lempel-Ziv family [47, 49, 50], Sequitur [36], Run-Length Encoding, Re-Pair [33], and many more [2–4, 20, 26, 30, 31, 41, 48]. All of these are or can be transformed into equivalent grammar-based compression schemes with little expansion [14, 39].

Given a grammar $\mathcal{S}$ representing a string $S$, the *random access problem* is to compactly represent $\mathcal{S}$ while supporting fast access queries, that is, given an index $i$ in $S$ to report $S[i]$. The random access problem is one of the most basic primitives for computation on grammar compressed strings, and solutions to the problem are a key component in a wide range of algorithms and data structures for grammar compressed strings [5, 8–10, 21–23, 28, 43, 44].

In this paper we study the random access problem with the *finger search property*, that is, the time for a random access query should depend on the distance between a specified index $f$, called the *finger*, and the query index $i$. We consider two variants of the problem. The first variant is *static finger search*, where we can place a finger with a setfinger operation and subsequently access positions near the finger efficiently. The finger can only be moved by a new setfinger operation, and the time for setfinger is independent of the distance to the previous position of the finger. The second variant is *dynamic finger search*, where we also support a movefinger operation that updates the finger such that the update time depends on the distance the finger is moved.

Our main result is efficient solutions to both finger search problems. To state the bounds, let $n$ be the size the grammar $\mathcal{S}$, and let $N$ be the size of the string $S$. For the static finger search problem, we give an

$O(n)$ space representation that supports setfinger in $O(\log N)$ time and access in $O(\log D)$ time, where $D$ is the distance between the finger and the accessed index. For the dynamic finger search problem, we give an $O(n)$ space representation that supports setfinger in $O(\log N)$ time and movefinger and access in $O(\log D + \log \log N)$ time. The best linear space solution for the random access problem uses $O(\log N)$ time for access. Hence, compared to our result we improve the $O(\log N)$ bound to $O(\log D)$ for the static version and to $O(\log D + \log \log N)$ for the dynamic version, while maintaining linear space. These are the first non-trivial bounds for the finger search problems.

As an application of our results we also give a new solution to the *longest common extension problem* on grammar compressed strings [9, 28, 37]. Here, the goal is to compactly represent $\mathcal{S}$ while supporting fast lce queries, that is, given a pair of indices $i, j$ to compute the length of the longest common prefix of $S[i, N]$ and $S[j, N]$. We give an $O(n)$ space representation that answers queries in $O(\log N + \log^2 \ell)$, where $\ell$ is the length of the longest common prefix. The best $O(n)$ space solution for this problem uses $O(\log N \log \ell)$ time, and hence our new bound is always at least as good and better whenever $\ell = o(N^\varepsilon)$.

## 1.1 Related Work

We briefly review the related work on the random access problem and finger search.

**Random Access in Grammar Compressed Strings** First note that naively we can store $S$ explicitly using $O(N)$ space and report any character in constant time. Alternatively, we can compute and store the sizes of the strings derived by each grammar symbol in $\mathcal{S}$ and use this to simulate a top-down search on the grammars derivation tree in constant time per node. This leads to an $O(n)$ space representation using $O(h)$ time, where $h$ is the height of the grammar [25]. Improved succinct space representation of this solution are also known [15]. Bille et al. [10] gave a solution using $O(n)$ and $O(\log N)$ time, thus achieving a query time independent of the height of the grammar. Verbin and Yu [46] gave a near matching lower bound by showing that any solution using $O(n \log^{O(1)} N)$ space must use $\Omega(\log^{1-\epsilon} N)$ time. Hence, we cannot hope to obtain significantly faster query times within $O(n)$ space. Finally, Belazzougui et al. [5] very recently showed that with superlinear space slightly faster query times are possible. Specifically, they gave a solution using $O(n\tau \log_\tau N/n)$ space and $O(\log_\tau N)$ time, where $\tau$ is a trade-off parameter. For $\tau = \log^\epsilon N$ this is $O(n \log^\epsilon N)$ space and $O(\log N/\log \log N)$ time. Practical solutions to this problem have been considered in [6, 24, 35].

The above solutions all generalize to support decompression of an arbitrary substring of length $D$ in time $O(t_{\mathsf{access}} + D)$, where $t_{\mathsf{access}}$ is the time for access (and even faster for small alphabets [5]). We can extend this to a simple solution to finger search (static and dynamic). The key idea is to implement setfinger as a random access and access and movefinger by decompressing or traversing, respectively, the part of the grammar in-between the two positions. This leads to a solution that uses $O(t_{\mathsf{access}})$ time for setfinger and $O(D)$ time for access and movefinger.

Another closely related problem is the *bookmarking problem*, where a set of positions, called *bookmarks*, are given at preprocessing time and the goal is to support fast substring decompression from any bookmark in constant or near-constant time per decompressed character [16, 21]. In other words, bookmarking allows us to decompress a substring of length $D$ in time $O(D)$ if the substring crosses a bookmark. Hence, with bookmarking we can improve the $O(t_{\mathsf{access}} + D)$ time solution for substring decompression to $O(D)$ whenever we know the positions of the substrings we want to decompress at preprocessing time. A key component in the current solutions to bookmarking is to trade-off the $\Omega(D)$ time we need to pay to decompress and output the substring. Our goal is to support access without decompressing in $o(D)$ time and hence this idea does not immediately apply to finger search.

**Finger Search** Finger search is a classic and well-studied concept in data structures, see e.g., [7,11,13, 17,19,27,32,34,38,40,42] and the survey [12]. In this setting, the goal is to maintain a dynamic dictionary data structure such that searches have the finger search property. Classic textbook examples of efficient finger search dictionaries include splay trees, skip lists, and level linked trees. Given a comparison based dictionary with $n$ elements, we can support optimal searching in $O(\log n)$ time and finger searching in $O(\log d)$ time, where $d$ is the rank distance between the finger and the query [12]. Note the similarity to our compressed results that reduce an $O(\log N)$ bound to $O(\log D)$.

## 1.2 Our results

We now formally state our results. Let $S$ be a string of length $N$ compressed into a grammar $\mathcal{S}$ of length $n$. Our goal is to support the following operations on $\mathcal{S}$.

access($i$): return the character $S[i]$

setfinger($f$): set the finger at position $f$ in $S$.

movefinger($f$): move the finger to position $f$ in $S$.

The static finger problem is to support access and setfinger, and the dynamic finger search problem is to support all three operations. We obtain the following bounds for the finger search problems.

**Theorem 1** *Let $\mathcal{S}$ be a grammar of size $n$ representing a string $S$ of length $N$. Let $f$ be the current position of the finger, and let $D = |f - i|$ for some $i$. Using $O(n)$ space we can support either:*

*(i)* setfinger($f$) *in $O(\log N)$ time and* access($i$) *in $O(\log D)$ time.*

*(ii)* setfinger($f$) *in $O(\log N)$ time,* movefinger($i$) *and* access($i$) *both in $O(\log D + \log \log N)$ time.*

Compared to the previous best linear space solution, we improve the $O(\log N)$ bound to $O(\log D)$ for the static variant and to $O(\log D + \log \log N)$ for the dynamic variant, while maintaining linear space. These are the first non-trivial solutions to the finger search problems. Moreover, the logarithmic bound in terms of $D$ may be viewed as a natural grammar compressed analogue of the classic uncompressed finger search solutions. We note that Theorem 1 is straightforward to generalize to multiple fingers. Each additional finger can be set in $O(\log N)$ time, uses $O(\log N)$ additional space, and given any finger $f$, we can support access($i$) in $O(\log D_f)$ time, where $D_f = |f - i|$.

## 1.3 Technical Overview

To obtain Theorem 1 we introduce several new techniques of independent interest. First, we consider a variant of the random access problem, which we call the *fringe access problem*. Here, the goal is to support fast access close to the beginning or end (the fringe) of a substring derived by a grammar symbol. We present an $O(n)$ space representation that supports fringe access from any grammar symbol $v$ in time $O(\log D_v + \log \log N)$, where $D_v$ is the distance from the fringe in the string $S(v)$ derived by $v$ to the queried position. The key challenge is designing a data structure for efficient navigation in unbalanced grammars.

The main component in our solution to this problem is a new recursive decomposition. The decomposition resembles the classic van Emde Boas data structure [45], in the sense that we recursively partition the grammar into a hierarchy of depth $O(\log \log N)$ consisting of subgrammars generating strings of lengths $N^{1/2}, N^{1/4}, N^{1/8}, \ldots$. We then show how to implement fringe access via predecessor queries on special paths produced by the decomposition. We cannot afford to explicitly store a predecessor data structure for each special path, however, using a technique due to Bille et al. [10], we can represent all the special paths compactly in a tree and instead implement the predecessor queries as weighted ancestor queries on the tree. This leads to an $O(n)$ space solution with $O(\log D_v + (\log \log N)^2)$ query time. Whenever $D_v \geq 2^{(\log \log N)^2}$ this matches our desired bound of $O(\log D_v + \log \log N)$. To handle the case when $D_v \leq 2^{(\log \log N)^2}$ we use an additional decomposition of the grammar and further reduce the problem to weighted ancestor queries on trees of small weighted height. Finally, we give an efficient solution to weighted ancestor for this specialized case that leads to our final result for fringe access.

Next, we use our fringe access result to obtain our solution to the static finger search problem. The key idea is to decompose the grammar into heavy paths as done by Bille et al. [10], which has the property that any root-to-leaf path in the directed acyclic graph representing the grammar consists of at most $O(\log N)$ heavy paths. We then use this to compactly represent the finger as a sequence of the heavy paths. To implement access, we binary search the heavy paths in the finger to find an exit point on the finger, which we then use to find an appropriate node to apply our solution to fringe access on. Together with a few additional tricks this gives us Theorem 1(i).

Unfortunately, the above approach for the static finger search problem does not extend to the dynamic setting. The key issue is that even a tiny local change in the position of the finger can change $\Theta(\log N)$

heavy paths in the representation of the finger, hence requiring at least $\Omega(\log N)$ work to implement movefinger. To avoid this we give a new compact representation of the finger based on both heavy path and the special paths obtained from our van Emde Boas decomposition used in our fringe access data structure. We show how to efficiently maintain this representation during local changes of the finger, ultimately leading to Theorem 1(ii).

## 1.4 Longest Common Extensions

As application of Theorem 1, we give an improved solution to longest common extension problem in grammar compressed strings. The first solution to this problem is due to Bille et al. [9]. They showed how to extend random access queries to compute Karp-Rabin fingerprints. Combined with an exponential search this leads to a linear space solution to the longest common extension problem using $O(\log N \log \ell)$ time, where $\ell$ is the length of the longest common extension. We note that we can plug in any of the above mentioned random access solution. More recently, Nishimoto et al. [37] used a completely different approach to get $O(\log N + \log \ell \log^* N)$ query time while using superlinear $O(n \log N \log^* N)$ space. We obtain:

**Theorem 2** *Let $\mathcal{S}$ be a grammar of size $n$ representing a string $S$ of length $N$. We can solve the longest common extension problem in $O(\log N + \log^2 \ell)$ time and $O(n)$ space where $\ell$ is the length of the longest common extension.*

Note that we need to verify the Karp-Rabin fingerprints during preprocessing in order to obtain a worst-case query time. Using the result from Bille et al. [10] this gives a randomized expected preprocessing time of $O(N \log N)$.

Theorem 2 improves the $O(\log N \log \ell)$ solution to $O(\log N + \log^2 \ell)$. The new bound is always at least as good and asymptotically better whenever $\ell = o(N^\epsilon)$ where $\epsilon$ is a constant. The new result follows by extending Theorem 1 to compute Karp-Rabin fingerprints and use these to perform the exponential search from [9].

## 2 Preliminaries

**Strings and Trees**   Let $S = S[1, |S|]$ be a string of length $|S|$. Denote by $S[i]$ the character in $S$ at index $i$ and let $S[i, j]$ be the substring of $S$ of length $j - i + 1$ from index $i \geq 1$ to $|S| \geq j \geq i$, both indices included.

Given a rooted tree $T$, we denote by $T(v)$ the subtree rooted in a node $v$ and the left and right child of a node $v$ by $left(v)$ and $right(v)$ if the tree is binary. The *nearest common ancestor* $\mathsf{nca}(v, u)$ of two nodes $v$ and $u$ is the deepest node that is an ancestor of both $v$ and $u$. A weighted tree has weights on its edges. A *weighted ancestor* query for node $v$ and weight $d$ returns the highest node $w$ such that the sum of weights on the path from the root to $w$ is at least $d$.

**Grammars and Straight Line Programs**   Grammar-based compression replaces a long string by a small context-free grammar (CFG). We assume without loss of generality that the grammars are in fact *straight-line programs* (SLPs). The lefthand side of a grammar rule in an SLP has exactly one variable, and the forighthand side has either exactly two variables or one terminal symbol. In addition, SLPs are unambigous and acyclic. We view SLPs as a directed acyclic graph (DAG) where each rule correspond to a node with outgoing ordered edges to its variables. Let $\mathcal{S}$ be an SLP. As with trees, we denote the left and right child of an internal node $v$ by $left(v)$ and $right(v)$. The unique string $S(v)$ of length $N_v$ is produced by a depth-first left-to-right traversal of $v$ in $\mathcal{S}$ and consist of the characters on the leafs in the order they are visited. The corresponding parse tree for $v$ is denoted $T(v)$. We will use the following results, that provides efficient random access from any node $v$ in $\mathcal{S}$.

**Lemma 1 ([10])** *Let $S$ be a string of length $N$ compressed into a SLP $\mathcal{S}$ of size $n$. Given a node $v \in \mathcal{S}$, we can support random access in $S(v)$ in $O(\log N_v)$ time, and at the same time reporting the sequence of heavy paths and their entry- and exit points in the corresponding depth-first traversal of $\mathcal{S}(v)$. The number of heavy paths visited is $O(\log N_v)$.*

**Karp-Rabin Fingerprints** For a prime $p$, $2n^{c+4} < p \leq 4n^{c+4}$ and $x \in [p]$ the Karp-Rabin fingerprint [29], denoted $\phi(S[i,j])$, of the substring $S[i,j]$ is defined as $\phi(S[i,j]) = \sum_{i \leq k \leq j} S[k] x^{k-i} \bmod p$. The key property is that for a random choice of $x$, two substrings of $S$ match iff their fingerprints match (whp.), thus allowing us to compare substrings in constant time. We use the following well-known properties of fingerprints.

**Lemma 2** *The Karp-Rabin fingerprints have the following properties:*

1) *Given $\phi(S[i,j])$, the fingerprint $\phi(S[i, j \pm a])$ for some integer $a$, can be computed in $O(a)$ time.*

2) *Given fingerprints $\phi(S[1,i])$ and $\phi(S[1,j])$, the fingerprint $\phi(S[i,j])$ can be computed in $O(1)$ time.*

3) *Given fingerprints $\phi(S_1)$ and $\phi(S_2)$, the fingerprint $\phi(S_1 \cdot S_2) = \phi(S_1) \oplus \phi(S_2)$ can be computed in $O(1)$ time.*

# 3 Fringe Access

In this section we consider the *fringe access problem*. Here the goal is to compactly represent the SLP, such that for any node $v$, we can efficiently access locations in the string $S(v)$ close to the start or the end of the substring. The fringe access problem is the key component in our finger search data structures. A straightforward solution to the fringe access problem is to apply a solution to the random access problem. For instance if we apply the random access solution from Bille et al. [10] stated in Lemma 1 we immediately obtain a linear space solution with $O(\log N_v)$ access time, i.e., the access time is independent of the distance to the start or the end of the string. This is an immediate consequence of the central grammar decomposition technique of [10], and does not extend to solve fringe access efficiently. Our main contribution in this section is a new approach that bypasses this obstacle. We show the following result.

**Lemma 3** *Let $\mathcal{S}$ be an SLP of size $n$ representing a string of length $N$. Using $O(n)$ space, we can support access to position $i$ of any node $v$, in time $O(\log(\min(i, N_v - i)) + \log \log N)$.*

The key idea in this result is a van Emde Boas style decomposition of $\mathcal{S}$ combined with a predecessor data structure on selected paths in the decomposition. To achieve linear space we reduce the predecessor queries on these paths to a weighted ancestor query. We first give a data structure with query time $O((\log \log N)^2 + \log(\min(i, N_v - i)))$. We then show how to reduce the query time to $O(\log \log N + \log(\min(i, N_v - i)))$ by reducing the query time for small $i$. To do so we introduce an additional decomposition and give a new data structure that supports fast weighted ancestor queries on trees of small weighted height.

For simplicity and without loss of generality we assume that the access point $i$ is closest to the start of $S(v)$, i.e., the goal is to obtain $O(\log(i) + \log \log N)$ time. By symmetry we can obtain the corresponding result for access points close to the end of $S(v)$.

## 3.1 van Emde Boas Decomposition for Grammars

We first define the vEB decomposition on the parse tree $T$ and then extend it to the SLP $\mathcal{S}$. In the decomposition we use the ART decompostion by Alstrup et al. [1].

**ART Decomposition** The ART decomposition introduced by Alstrup et al. [1] decomposes a tree into a single top tree and a number of bottom trees. Each bottom tree is a subtree rooted in a node of minimal depth such that the subtree contains no more than $x$ leaves and the top tree is all nodes not in a bottom tree. The decomposition has the following key property.

**Lemma 4 ([1])** *The ART decomposition with parameter $x$ for a rooted tree $T$ with $N$ leaves produces a top tree with at most $\frac{N}{x+1}$ leaves.*

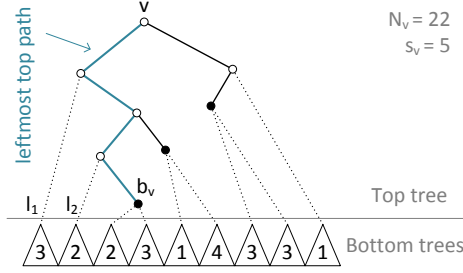We are now ready to define the *van Emde Boas (vEB) decomposition*.

Figure 1: Example of the ART-decomposition and a leftmost top path. In the top, the nodes forming the top tree are drawn. In the bottom, triangles representing the bottom trees with a number that is the size of the bottom tree. $v$'s leftmost top path is shown as well, and the two trees hanging to the left of this path $l_1$ and $l_2$.

**The van Emde Boas Decomposition** We define the van Emde Boas Decomposition of a tree $T$ as follows. The *van Emde Boas (vEB) decomposition* of $T$ is obtained by recursively applying an ART decomposition: Let $v = root(T)$ and $x = \sqrt{N}$. If $N = O(1)$, stop. Otherwise, construct an ART decomposition of $T(v)$ with parameter $x$. For each bottom tree $T(u)$ recursively construct a vEB decomposition with $v = u$ and $x = \sqrt{x}$.

Define the *level* of a node $v$ in $T$ as $\text{level}(v) = \lfloor \log \log N - \log \log N_v \rfloor$ (this corresponds to the depth of the recursion when $v$ is included in its top tree).

Note that except for the nodes on the lowest level—which are not in any top tree—all nodes belong to exactly one top tree. For any node $v \in T$ not in the last level, let $T_{top}(v)$ be the top tree $v$ belongs to. The *leftmost top path* of $v$ is the path from $v$ to the *leftmost leaf* of $T_{top}(v)$. See Figure 1.

Intuitively, the vEB decomposition of $T$ defines a nested hierarchy of subtrees that decrease by at least the square root of the size at each step.

**The van Emde Boas Decomposition of Grammars** Our definition of the vEB decomposition of trees can be extended to SLPs as follows. Since the vEB decomposition is based only on the length of the string $N_v$ generated by each node $v$, the definition of the vEB decomposition is also well-defined on SLPs. As in the tree, all nodes belong to at most one top DAG. We can therefore reuse the terminology from the definition for trees on SLPs as well.

To compute the vEB decomposition first determine the level of each node and then remove all edges between nodes on different levels. This can be done in $O(n)$ time.

## 3.2 Data Structure

We first present a data structure that achieves $O((\log \log N)^2 + \log(i))$ time. In the next section we then show how to improve the running time to the desired $O(\log \log(N) + \log(i))$ bound.

Our data structure contains the following information for each node $v \in \mathcal{S}$. Let $l_1, l_2, \ldots, l_k$ be the nodes hanging to the left of $v$'s leftmost top path (excluding nodes hanging from the bottom node).

- The length $N_v$ of $S(v)$.

- The sum of the sizes of nodes hanging to the left of $v$'s leftmost top path $s_v = |l_1| + |l_2| + \ldots + |l_k|$.

- A pointer $b_v$ to the bottom node on $v$'s leftmost top path.

- A predecessor data structure over the sequence $1, |l_1| + 1, |l_1| + |l_2| + 1, \ldots, \sum_{i=1}^{k-1} |l_i| + 1$. We will later show how to represent this data structure.

In addition we also build the data structure from Lemma 1 that given any node $v$ supports random access to $S(v)$ in $O(\log N_v)$ time using $O(n)$ space.

To perform an access query we proceed as follows. Suppose that we have reached some node $v$ and we want to compute $S(v)[i]$. We consider the following five cases (when multiple cases apply take the first):

1. If $N_v = O(1)$. Decompress $S(v)$ and return the $i$'th character.

2. If $i \leq s_v$. Find the predecessor $p$ of $i$ in $v$'s predecessor structure and let $u$ be the corresponding node. Recursively find $S(u)[i - p]$.

3. If $i \leq s_v + N_{left(b_v)}$. Recursively find $S(left(b_v))[i - s_v]$.

4. If $i \leq s_v + N_{b_v}$. Recursively find $S(right(b_v))[i - s_v - N_{left(b_v)}]$.

5. In all other cases, perform a random access for $i$ in $\mathcal{S}(v)$ using Lemma 1.

To see correctness, first note that case (1) and (5) are correct by definition. Case (2) is correct since when $i \leq s_v$ we know the $i$'th leaf must be in one of the trees hanging to the left of the leftmost top path, and the predecessor query ensures we recurse into the correct one of these bottom trees. In case (3) and (4) we check if the $i$'th leaf is either in the left or right subtree of $b_v$ and if it is, we recurse into the correct one of these.

**Compact Predecessor Data Structures**  We now describe how to represent the predecessor data structure. Simply storing a predecessor structure in every single node would use $O(n^2)$ space. We can reduce the space to $O(n)$ using ideas similar to the construction of the "heavy path suffix forest" in [10].

Let $L$ denote the *leftmost top path forest*. The nodes of $L$ are the nodes of $\mathcal{S}$. A node $u$ is the parent of $v$ in $L$ iff $u$ is a child of $v$ in $\mathcal{S}$ and $u$ is on $v$'s leftmost top path. Thus, a leftmost top path $v_1, \ldots, v_k$ in $\mathcal{S}$ is a sequence of ancestors from $v_1$ in $L$. The weight of an edge $(u, v)$ in $L$ is 0 if $u$ is a left child of $v$ in $\mathcal{S}$ and otherwise $N_{left(v)}$. Several leftmost top paths in $\mathcal{S}$ can share the same suffix, but the leftmost top path of a node in $\mathcal{S}$ is uniquely defined and thus $L$ is a forest. A leftmost path ends in a leaf in the top DAG, and therefore $L$ consists of $O(n)$ trees each rooted at a unique leaf of a top dag. A predecessor query on the sequence $1, |l_1| + 1, |l_1| + |l_2| + 1, \ldots, \sum_{i=1}^{k-1} |l_i| + 1$ now corresponds to a weighted ancestor query in $L$. We plug in the weighted ancestor data structure from Farach-Colton and Muthukrishnan [18], which supports weighted ancestor queries in a forest in $O(\log \log n + \log \log U))$ time with $O(n)$ preprocessing and space, where $U$ is the maximum weight of a root-to-leaf path and $n$ the number of leaves. We have $U = N$ and hence the time for queries becomes $O(\log \log N)$.

**Space and Preprocessing Time**  For each node in $\mathcal{S}$ we store a constant number of values, which takes $O(n)$ space. Both the predecessor data structure and the data structure for supporting random access from Lemma 1 take $O(n)$ space, so the overall space usage is $O(n)$. The vEB decomposition can be computed in $O(n)$ time. The leftmost top paths and the information saved in each node can be computed in linear time. The predecessor data structure uses linear preprocessing time, and thus the total preprocessing time is $O(n)$.

**Query Time**  Consider each case of the recursion. The time for case (1), (3) and (4) is trivially $O(1)$. Case (2) is $O(\log \log N)$ since we perform exactly one predececssor query in the predecessor data structure.

In case (5) we make a random access query in a node of size $N_v$. From Lemma 1 we have that the query time is $O(\log N_v)$. We know $level(v) = level(b_v)$ since they are on the same leftmost top path. From the definition of the level it follows for any pair of nodes $u$ and $w$ with the same level that $N_u \geq \sqrt{N_w}$ and thus $N_{b_v} \geq \sqrt{N_v}$. From the conditions we have $i > s_v + N_{b_v} \geq N_{b_v} \geq \sqrt{N_v}$. Since $\sqrt{N_v} < i \Leftrightarrow \log N_v < 2 \log i$ we have $\log N_v = O(\log i)$ and thus the running time for case (5) is $O(\log N_v) = O(\log i)$.

Case (1) and (5) terminate the algorithm and can thus not happen more than once. Case (2), (3) and (4) are repeated at most $O(\log \log N)$ times since the level of the node we recurse on increments by at least one in each recursive call, and the level of a node is at most $O(\log \log N)$. The overall running time is therefore $O((\log \log N)^2 + \log i)$.

In summary, we have the following result.

**Lemma 5** *Let $\mathcal{S}$ be an SLP of size $n$ representing a string of length $N$. Using $O(n)$ space, we can support access to position $i$ of any node $v$, in time $O(\log i + (\log \log N)^2)$.*

### 3.3 Improving the Query Time for Small Indices

The above algorithm obtains the running time $O(\log i)$ for $i \geq 2^{(\log \log N)^2}$. We will now improve the running time to $O(\log \log N + \log i)$ by improving the running time in the case when $i < 2^{(\log \log N)^2}$.

In addition to the data structure from above, we add another copy of the data structure with a few changes. When answering a query, we first check if $i \geq 2^{(\log \log N)^2}$. If $i \geq 2^{(\log \log N)^2}$ we use the original data structure, otherwise we use the new copy.

The new copy of the data structure is implemented as follows. In the first level of the ART-decomposition let $x = 2^{(\log \log N)^2}$ instead of $\sqrt{N}$. For the rest of the levels use $\sqrt{x}$ as before. Furthermore, we split the resulting new leftmost top path forest $L$ into two disjoint parts: $L_1$ consisting of all nodes with level 1 and $L_{\geq 2}$ consisting of all nodes with level at least 2. For $L_1$ we use the weighted ancestor data structure by Farach-Colton and Muthukrishnan [18] as in the previous section using $O(\log \log n + \log \log N) = O(\log \log N)$ time. However, if we apply this solution for $L_{\geq 2}$ we end up with a query time of $O(\log \log n + \log \log x)$, which does not lead to an improved solution. Instead, we present a new data structure that supports queries in $O(\log \log x)$ time.

**Lemma 6** *Given a tree $T$ with $n$ leaves where the sum of edge weights on any root-to-leaf path is at most $x$ and the height is at most $x$, we can support weighted ancestor queries in $O(\log \log x)$ time using $O(n)$ space and preprocessing time.*

*Proof.* Create an ART-decomposition of $T$ with parameter $x$. For each bottom tree in the decomposition construct the weighted ancestor structure from [18]. For the top tree, construct a predecessor structure over the accumulated edge weights for each root-to-leaf path.

To perform a weighted ancestor query on a node in a bottom tree, we first perform a weighted ancestor query using the data structure for the bottom tree. In case we end up in the root of the bottom tree, we continue with a predecessor search in the top tree from the leaf corresponding to the bottom tree.

The total space for bottom trees is $O(n)$. Since the top tree has $O(n/x)$ leaves and height at most $x$, the total space for all predecessor data structures on root-to-leaf paths in the top tree is $O(n/x \cdot x) = O(n)$. Hence, the total space is $O(n)$.

A predecessor query in the top tree takes $O(\log \log x)$ time. The number of nodes in each bottom tree is at most $x^2$ since it has at most $x$ leaves and height $x$ and the maximum weight of a root-to-leaf path is $x$ giving weighted ancestor queries in $O(\log \log x^2 + \log \log x) = O(\log \log x)$ time. Hence, the total query time is $O(\log \log x)$. □

We reduce the query time for queries with $i < 2^{(\log \log N)^2}$ using the new data structure. The level of any node in the new structure is at most $O(1 + \log \log 2^{(\log \log N)^2}) = O(\log \log \log N)$. A weighted ancestor query in $L_1$ takes time $O(\log \log N)$. For weighted ancestor queries in $L_{\geq 2}$, we know any node $v$ has height at most $2^{(\log \log N)^2}$ and on any root-to-leaf path the sum of the weights is at most $2^{(\log \log N)^2}$. Hence, by Lemma 6 we support queries in $O(\log \log 2^{(\log \log N)^2}) = O(\log \log \log N)$ time for nodes in $L_{\geq 2}$.

We make at most one weighted ancestor query in $L_1$, the remaining ones are made in $L_{\geq 2}$, and thus the overall running time is $O(\log \log N + (\log \log \log N)^2 + \log i) = O(\log \log N + \log i)$.

In summary, this completes the proof of Lemma 3.

## 4 Static Finger Search

We now show how to apply our solution to the fringe access to a obtain a simple data structure for the static finger search problem. This solution will be the starting point for solving the dynamic case in the next section, and we will use it as a key component in our result for longest common extension problem.

Similar to the fringe search problem we assume without loss of generality that the access point $i$ is to the right of the finger.
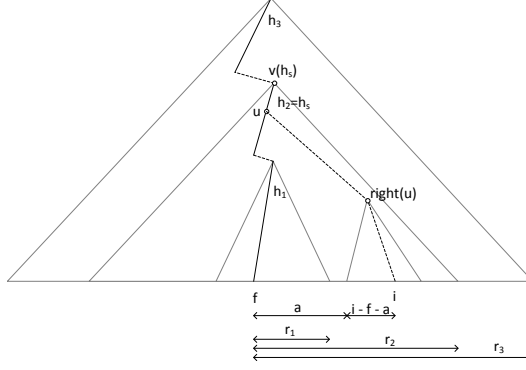
Figure 2: Illustration of the data structure for a finger pointing at $f$ and an access query at location $i$. $h_1, h_2, h_3$ are the heavy paths visited when finding the finger. $u$ corresponds to $NCA(v_f, v_i)$ in the parse tree and $h_s$ is the heavy path on which $u$ lies, which we use to find $u$. $a$ is a value calculated during the access query.

**Data Structure**   We store the random access data structure from [10] used in Lemma 1 and the fringe search data structures from above. Also from [10] we store the data structure that for any heavy path $h$ starting in a node $v$ and an index $i$ of a leaf in $T(v)$ gives the exit-node from $h$ when searching for $i$ in $O(\log\log N)$ time and uses $O(n)$ space.

   To represent a finger the key idea is store a compact data structure for the corresponding root-to-leaf path in the grammar that allows us to navigate it efficiently. Specifically, let $f$ be the position of the current finger and let $p = v_1 \ldots v_k$ denote the path in $\mathcal{S}$ from the root to $v_f$ ($v_1 = root$ and $v_k = v_f$). Decompose $p$ into the $O(\log N)$ heavy paths it intersects, and call these $h_j = v_1 \ldots v_{i_1}, h_{j-1} = v_{i_1+1} \ldots v_{i_2}, \cdots, h_1 = v_{i_{j-1}+1} \ldots v_k$. Let $v(h_i)$ be the topmost node on $h_i$ ($v(h_j) = v_1, v(h_{j-1}) = v_{i_1}, \ldots$). Let $l_j$ be the index of $f$ in $\mathcal{S}(v(h_j))$ and $r_j = N_{v(h_j)} - l_j$. For the finger we store:

1. The sequence $r_1, r_2, \ldots, r_j$ (note $r_1 \leq r_2 \leq \cdots \leq r_j$).

2. The sequence $v(h_1), v(h_2), \ldots, v(h_j)$.

3. The string $F_T = S[f+1, f+\log N]$.

**Analysis**   The random access and fringe search data structures both require $O(n)$ space. Each of the 3 bullets above require $O(\log N)$ space and thus the finger takes up $O(\log N)$ space. The total space usage is $O(n)$.

**Setfinger**   We implement setfinger($f$) as follows. First, we apply Lemma 1 to make random access to position $f$. This gives us the sequence of visited heavy paths which exactly corresponds to $h_j, h_{j-1}, \ldots, h_1$ including the corresponding $l_i$ values from which we can calculate the $r_i$ values. So we update the $r_i$ sequence accordingly. Finally, decompress and save the string $F_T = S[f+1, f+\log N]$.

   The random access to position $f$ takes $O(\log N)$ time. In addition to this we perform a constant number of operations for each heavy path $h_i$, which in total takes $O(\log N)$ time. Decompressing a string of $\log N$ characters can be done in $O(\log N)$ time (using [10]). In total, we use $O(\log N)$ time.

**Access**   To perform access($i$) ($i > f$), there are two cases. If $D = i - f \leq \log N$ we simply return the stored character $F_T[D]$ in constant time. Otherwise, we compute the node $u = \mathsf{nca}(v_f, v_i)$ in the parse tree $T$ as follows. First find the index $s$ of the successor to $D$ in the $r_i$ sequence using binary search. Now we know that $u$ is on the heavy path $h_s$. Find the exit-nodes from $h_s$ when searching for respectively $i$ and $f$ using the data structure from [10] - the topmost of these two is $u$. See Fig. 2. Finally, we compute $a$ as the index of $f$ in $T(left(u))$ from the right and use the data structure for fringe search from Lemma 3 to compute $S(right(u))[i - f - a]$.

   For $D \leq \log N$, the operation takes constant time. For $D > \log N$, the binary search over a sequence of $O(\log N)$ elements takes $O(\log\log N)$ time, finding the exit-nodes takes $O(\log\log N)$ time, and the

fringe search takes $O(\log(i - f - a)) = O(\log D)$ time. Hence, in total $O(\log \log N + \log D) = O(\log D)$ time.

This completes the proof of Theorem 1(i).

# 5 Dynamic Finger Search

In this section we show how to extend the solution from Section 4 to handle dynamic finger search. The target is to support the movefinger operation that will move the current finger, where the time it takes is dependent on how far the finger is moved. Obviously, it should be faster than simply using the setfinger operation. The key difference from the static finger is a new decomposition of a root-to-leaf path into paths. The new decomposition is based on a combination of heavy paths and leftmost top paths, which we will show first. Then we show how to change the data structure to use this decomposition, and how to modify the operations accordingly. Finally, we consider how to generalize the solution to work when movefinger/access might both be to the left and right of the current finger, which for this solution is not trivially just by symmetry.

Before we start, let us see why the data structure for the static finger cannot directly be used for dynamic finger. Suppose we have a finger pointing at $f$ described by $\Theta(\log N)$ heavy paths. It might be the case that after a movefinger($f + 1$) operation, it is $\Theta(\log N)$ completely different heavy paths that describes the finger. In this case we must do $\Theta(\log N)$ work to keep our finger data structure updated. This can for instance happen when the current finger is pointing at the right-most leaf in the left subtree of the root.

Furthermore, in the solution to the static problem, we store the substring $S[f + 1, f + \log N]$ decompressed in our data structure. If we perform a movefinger($f + \log N$) operation nothing of this substring can be reused. To decompress $\log N$ characters takes $\Omega(\log N)$ time, thus we cannot do this in the movefinger operation and still get something faster than $\Theta(\log N)$.

## 5.1 Left Heavy Path Decomposition of a Path

Let $p = v_1 \ldots v_k$ be a root-to-leaf path in $\mathcal{S}$. A subpath $p_i = v_a \ldots v_b$ of $p$ is a *maximal heavy subpath* if $v_a \ldots v_b$ is part of a heavy path and $v_{b+1}$ is not on the same heavy path. Similarly, a subpath $p_i = v_a \ldots v_b$ of $p$ is a *maximal leftmost top subpath* if $v_a \ldots v_b$ is part of a leftmost top path and $level(v_b) \neq level(v_{b+1})$.

A *left heavy path decomposition* is a decomposition of a root-to-leaf path $p$ into an arbitrary sequence $p_1 \ldots p_j$ of maximal heavy subpaths, maximal leftmost top subpaths and (non-maximal) leftmost top subpaths immediately followed by maximal heavy subpaths.

Define $v(p_i)$ as the topmost node on the subpath $p_i$. Let $l_j$ be the index of the finger $f$ in $\mathcal{S}(v(p_j))$ and $r_j = N_{v(p_j)} - l_j$. Let $t(p_i)$ be the type of $p_i$; either heavy subpath ($HP$) or leftmost top subpath ($LTP$).

A left heavy path decomposition of a root-to-leaf path $p$ is not unique. The heavy path decomposition of $p$ is always a valid left heavy path decomposition as well. The visited heavy paths and leftmost top paths during fringe search are always maximal and thus is always a valid left heavy path decomposition.

**Lemma 7** *The number of paths in a left heavy path decomposition is $O(\log N)$.*

*Proof.* There are at most $O(\log N)$ heavy paths that intersects with a root-to-leaf path (Lemma 1). Each of these can at most be used once because of the maximality. So there can at most be $O(\log N)$ maximal heavy paths. Each time there is a maximal leftmost top path, the level of the following node on $p$ increases. This can happen at most $O(\log \log N)$ times. Each non-maximal leftmost top path is followed by a maximal heavy path, and since there are only $O(\log N)$ of these, this can happen at most $O(\log N)$ times. Therefore the sequence of paths has length $O(\log N + \log \log N + \log N) = O(\log N)$.

## 5.2 Data Structure

We use the data structures from [10] as in the static variant and the fringe access data structure with an extension. In the fringe access data structure there is a predecessor data structure for all the nodes hanging to the left of a leftmost top path. To support access and movefinger we need to find a node

hanging to the left or right of a leftmost top path. We can do this by storing an identical predecessor structure for the accumulated sizes of the nodes hanging to the right of each leftmost top path. Again, the space usage for this predecessor structure can be reduced to $O(n)$ by turning it into a weighted ancestor problem.

To represent a finger the idea is again to have a compact data structure representing the root-to-leaf path corresponding to the finger. This time we will base it on a left heavy path decomposition instead of a heavy path decomposition. Let $f$ be the current position of the finger. For the root-to-leaf path to $v_f$ we maintain a left heavy path decomposition, and store the following for a finger:

1. The sequence $r_1, r_2, \ldots, r_j$ ($r_1 \leq r_2 \leq \cdots \leq r_j$) on a stack with the last element on top.

2. The sequence $v(p_1), v(p_2), \ldots, v(p_j)$ on a stack with the last element on top.

3. The sequence $t(p_1), t(p_2), \ldots, t(p_j)$ on a stack with the last element on top.

**Analysis**  The fringe access data structure takes up $O(n)$ space. For each path in the left heavy path decomposition we use constant space. Using Lemma 7 we have the space usage of this is $O(\log N) = O(n)$.

**Setfinger**  Use fringe access (Lemma 3) to access position $f$. This gives us a sequence of leftmost top paths and heavy paths visited during the fringe access which is a valid left heavy path decomposition. Calculate $r_i$ for each of these and store the three sequences of $r_i$, $v(p_i)$ and $t(p_i)$ on stacks.

The fringe access takes $O(\log f + \log \log N)$ time. The number of subpaths visited during the fringe access cannot be more than $O(\log f + \log \log N)$ and we only perform constant extra work for each of these.

**Access**  To implement access($i$) for $i > f$ we have to find $u = \mathsf{nca}(v_i, v_f)$ in the $T$. Find the index $s$ of the successor to $D = i - f$ in $r_1, r_2, \ldots, r_j$ using binary search. We know $\mathsf{nca}(v_i, v_f)$ lies on $p_s$, and $v_i$ is in a subtree that hangs of $p_s$. The exit-nodes from $p_s$ to $v_f$ and $v_i$ are now found - the topmost of these two is $\mathsf{nca}(v_i, v_f)$. If $t(p_s) = HP$ then we can use the same data structure as in the static case, otherwise we perform the predecessor query on the extra predecessor data structure for the nodes hanging of the leftmost top path. Finally, we compute $a$ as the index of $f$ in $S(left(u))$ from the right and use the data structure for fringe access from Lemma 3 to compute $S(right(u))[i - f - a]$.

The binary search on $r_1, r_2, \ldots, r_j$ takes $O(\log \log N)$ time. Finding the exit-nodes from $p_s$ takes $O(\log \log N)$ in either case. Finally the fringe access takes $O(\log(i - f - a) + \log \log N) = O(\log D + \log \log N)$. Overall it takes $O(\log D + \log \log N)$.

Note the extra $O(\log \log N)$ time usage because we have not decompressed the first $\log N$ characters following the finger.

**Movefinger**  To move the finger we combine the access and setfinger operations. Find the index $s$ of the successor to $D = i - f$ in $r_1, r_2, \ldots, r_j$ using binary search. Now we know $u = \mathsf{nca}(v_i, v_f)$ must lie on $p_s$. Find $u$ in the same way as when performing access. From all of the stacks pop all elements above index $s$. Compute $a$ as the index of $f$ in $S(left(u))$ from the right. The finger should be moved to index $i - f - a$ in $right(u)$. First look at the heavy path $right(u)$ lies on and find the proper exit-node $w$ using the data structure from [10]. Then continue with fringe searh from the proper child of $w$. This gives a heavy path followed by a sequence of maximal leftmost top paths and heavy paths needed to reach $v_i$ from $right(u)$, push the $r_j$, $v(p_j)$, and $t(p_j)$ values for these on top of the respective stacks.

We now verify the sequence of paths we maintain is still a valid left heavy path decomposition. Since fringe search gives a sequence of paths that is a valid left heavy path decomposition, the only problem might be $p_s$ is no longer maximal. If $p_s$ is a heavy path it will still be maximal, but if $p_s$ is a leftmost top path then $level(u)$ and $level(right(u))$ might be equal. But this possibly non-maximal leftmost top path is always followed by a heavy path. Thus the overall sequence of paths remains a left heavy path decomposition.

The successor query in $r_1, r_2, \ldots, r_j$ takes $O(\log \log N)$ time. Finding $u$ on $p_i$ takes $O(\log \log N)$ time, and so does finding the exit-node on the following heavy path. Popping a number of elements from the top of the stacks can be done in $O(1)$ time. Finally the fringe access takes $O(\log(i - f - a) + \log \log N) = O(\log D + \log \log n)$ including pushing the right elements on the stacks. Overall the running time is therefore $O(\log D + \log \log n)$.

## 5.3 Moving/Access to the Left of the Finger

In the above we have assumed $i > f$, we will now show how this assumption can be removed. It is easy to see we can mirror all data structures and we will have a solution that works for $i < f$ instead. Unfortunately, we cannot just use a copy of each independently, since one of them only supports moving the finger to the left and the other only supports moving to the right. We would like to support moving the finger left and right arbitrarily. This was not a problem with the static finger since we could just make setfinger in both the mirrored and non-mirrored data structures in $O(\log N)$ time.

Instead we extend our finger data structure. First we extend the left heavy path decomposition to a *left right heavy path decomposition* by adding another type of paths to it, namely *rightmost top paths* (the mirrored version of leftmost top paths). Thus a *left right heavy path decomposition* is a decomposition of a root-to-leaf path $p$ into an arbitrary sequence $p_1 \ldots p_j$ of maximal heavy subpaths, maximal leftmost/rightmost top subpaths and (non-maximal) leftmost/rightmost top subpaths immediately followed by maximal heavy subpaths. Now $t(p_i) = HP|LTP|RTP$. Furthermore, we save the sequence $l_1, l_2, \ldots, l_j$ ($l_j$ being the left index of $f$ in $T(v(p_j))$) on a stack like the $r_1, r_2, \ldots, r_j$ values, etc.

When we do access and movefinger where $i < f$, the subpath $p_s$ where $nca(v_f, v_i)$ lies can be found by binary search on the $l_j$ values instead of the $r_j$ values. Note the $l_j$ values are sorted on the stack, just like the $r_j$ values. The following heavy path lookup/fringe access should now be performed on $left(u)$ instead of $right(u)$. The remaining operations can just be performed in the same way as before.

# 6 Finger Search with Fingerprints and Longest Common Extensions

We show how to extend our finger search data structure from Theorem 1(i) to support computing fingerprints and then apply the result to compute longest common extensions. First, we will show how to return a fingerprint for $S(v)[1, i]$ when performing access on the fringe of $v$.

## 6.1 Fast Fingerprints on the Fringe

To do this, we need to store some additional data for each node $v \in \mathcal{S}$. We store the fingerprint $\phi(S(v))$ and the concatenation of the fingerprints of the nodes hanging to the left of the leftmost top path $p_v = \phi(S(l_1)) \oplus \phi(S(l_2)) \oplus \ldots \oplus \phi(S(l_k))$. We also need the following lemma:

**Lemma 8 ([9])** *Let $S$ be a string of length $N$ compressed into a SLP $\mathcal{S}$ of size $n$. Given a node $v \in \mathcal{S}$, we can find the fingerprint $\phi(S(v)[1, i])$ where $1 \leq i \leq N_v$ in $O(\log N_v)$ time.*

Suppose we are in a node $v$ and we want to calculate the fingerprint $\phi(S(v)[1, i])$. We perform an access query as before, but also maintain a fingerprint $p$, initially $p = \phi(\epsilon)$, computed thus far. We follow the same five cases as before, but add the following to update $p$:

1. From the decompressed $S(v)$, calculate the fingerprint for $S(v)[1, i]$, now update $p = p \oplus \phi(S(v)[1, i])$.

2. $p = p \oplus (\phi(p_v) \ominus_s \phi(p_u))$.

3. $p = p \oplus \phi(p_v)$.

4. $p = p \oplus \phi(p_v) \oplus \phi(S(left(b_v)))$.

5. Use Lemma 8 to find the fingerprint for $S(v)[1, i]$ and then update with $p = p \oplus \phi(S(v)[1, i])$.

These extra operations do not change the running time of the algorithm, so we can now find the fingerprint $\phi(S(v)[1, i])$ in time $O(\log \log N + \log(\min(i, N_v - i)))$.

## 6.2 Finger Search with Fingerprints

Next we show how to do finger search while computing fingerprints between the finger $f$ and the access point $i$.

When we perform $\mathsf{setfinger}(f)$ we use the algorithm from [9] to compute fingerprints during the search of $\mathcal{S}$ from the root to $f$. This allows us to subsequently compute for any heavy path $h_j$ on the root to position $f$ the fingerprint $p(h_j)$ of the concatenation of the strings generated by the subtrees hanging to the left of $h_j$. In addition, we explicitly compute and store the fingerprints $\phi(S[1, f+1]), \phi(S[1, f+2]), \ldots, \phi(S[1, f + \log N + 1])$. In total, this takes $O(\log N)$ time.

Suppose that we have now performed a $\mathsf{setfinger}(f)$ operation. To implement $\mathsf{access}(i)$, $i > f$, there are two cases. If $D = i - f \leq \log N$ we return the appropriate precomputed fingerprint. Otherwise, we compute the node $u = \mathsf{nca}(v_f, v_i)$ in the parse tree $T$ as before. Let $h$ be the heavy path containing $u$. Using the data structure from [9] we compute the fingerprint $p_l$ of the nodes hanging to the left of $h$ above $u$ in constant time. The fingerprint is now obtained as $\phi(S[1, i]) = p_{h_j} \oplus p_l \oplus \phi(S(right(u))[1, (i-f)-a])$, where the latter is found using fringe access with fingerprints in $right(u)$. None of these additions change the asymptotic complexities of Theorem 1(i). Note that with the fingerprint construction in [9] we can guarantee that all fingerprints are collision-free.

## 6.3 Longest Common Extensions

Using the fingerprints it is now straightforward to implement $\mathsf{lce}$ queries as in [9]. Given a $\mathsf{lce}(i, j)$ query, first set fingers at positions $i$ and $j$. This allows us to get fingerprints of the form $\phi(S[i, i+a])$ or $\phi(S[j, j+a])$ efficiently. Then, we find the largest value $\ell$ such that $\phi(S[i, i+\ell]) = \phi(S[j, j+\ell])$ using a standard exponential search. Setting the two finger uses $O(\log N)$ time and by Theorem 1(i) the at most $O(\log \ell)$ searches in the exponential search take at most $O(\log \ell)$ time. Hence, in total we use $O(\log N + \log^2 \ell)$ time, as desired. This completes the proof of Theorem 2.

# References

[1] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. 39th FOCS*, pages 534–543, 1998.

[2] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In *Proc. DCC*, pages 119–128, 1998.

[3] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *Proc. DCC*, pages 143–152, 2000.

[4] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proceedings of the IEEE*, 88(11):1733–1744, 2000.

[5] D. Belazzougui, P. H. Cording, S. J. Puglisi, and Y. Tabei. Access, rank, and select in grammar-compressed strings. In *Proc. 23rd ESA*, 2015.

[6] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Karkkainen, A. Ordonez, S. Puglisi, and Y. Tabei. Queries on lz-bounded encodings. In *Proc. DCC*, pages 83–92, April 2015.

[7] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Inform. Process. Lett.*, 5(3):82 – 87, 1976.

[8] P. Bille, P. H. Cording, and I. L. Gørtz. Compressed subsequence matching and packed tree coloring. *Algorithmica*, pages 1–13, 2015.

[9] P. Bille, P. H. Cording, I. L. Gørtz, B. Sach, H. W. Vildhøj, and S. Vind. Fingerprints in compressed strings. In *Proc. 13th SWAT*, 2013.

[10] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput*, 44(3):513–539, 2014. Announced at SODA 2011.

[11] G. E. Blelloch, B. M. Maggs, and S. L. M. Woo. Space-efficient finger search on degree-balanced search trees. In *Proc. 14th SODA*, pages 374–383, 2003.

[12] G. S. Brodal. Finger search trees. In *Handbook of Data Structures and Applications.* Chapman and Hall/CRC, 2004.

[13] G. S. Brodal, G. Lagogiannis, C. Makris, A. K. Tsakalidis, and K. Tsichlas. Optimal finger search trees in the pointer machine. *J. Comput. Syst. Sci.*, 67(2):381–418, 2003.

[14] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005. Announced at STOC 2002 and SODA 2002.

[15] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fund. Inform.*, 111(3):313–337, 2011.

[16] P. H. Cording, P. Gawrychowski, and O. Weimann. Bookmarks in grammar-compressed strings. In *Proc. 23rd SPIRE*, pages x–y, 2016.

[17] P. F. Dietz and R. Raman. A constant update time finger search tree. *Inf. Process. Lett.*, 52(3):147–154, 1994.

[18] M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proc. 7th CPM*, pages 130–140. Springer, 1996.

[19] R. Fleischer. A simple balanced search tree with O(1) worst-case update time. *Int. J. Found. Comput. Sci.*, 7(2):137–150, 1996.

[20] P. Gage. A new algorithm for data compression. *The C Users J.*, 12(2):23 – 38, 1994.

[21] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. 6th LATA*, pages 240–251, 2012.

[22] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th LATIN*, pages 731–742. Springer, 2014.

[23] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Approximate pattern matching in lz77-compressed texts. *J. Discrete Algorithms*, 32:64–68, 2015.

[24] T. Gagie, C. Hoobin, and S. J. Puglisi. Block graphs in practice. In *Proc. ICABD*, pages 30–36, 2014.

[25] L. Gąsieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th DCC*, page 458, 2005.

[26] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *Proc. 26th CPM*, pages 219–230. Springer, 2015.

[27] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9th STOC*, pages 49–60, 1977.

[28] T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, and A. Shinohara. Detecting regularities on grammar-compressed strings. *Inform. Comput.*, 240:74–89, 2015.

[29] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[30] J. C. Kieffer and E. H. Yang. Grammar based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.

[31] J. C. Kieffer, E. H. Yang, G. J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Trans. Inf. Theory*, 46(5):1227 – 1245, 2000.

[32] S. R. Kosaraju. Localized search in sorted lists. In *Proc. 13th STOC*, pages 62–69, New York, NY, USA, 1981.

[33] N. J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[34] K. Mehlhorn. A new data structure for representing sorted lists. In *Proc. WG*, pages 90–112, 1981.

[35] G. Navarro and A. Ordóñez. Grammar compressed sequences with rank/select support. In *21st SPIRE*, pages 31–44. Springer, 2014.

[36] C. G. Nevill-Manning and I. H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *J. Artificial Intelligence Res.*, 7:67–82, 1997.

[37] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proc. 41st MFCS*, pages 72:1–72:15, 2016.

[38] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[39] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[40] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[41] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte Pair encoding: A text compression scheme that accelerates pattern matching. *Technical Report DOI-TR-161, Dept. of Informatics, Kyushu University*, 1999.

[42] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[43] T. Tanaka, I. Tomohiro, S. Inenaga, H. Bannai, and M. Takeda. Computing convolution on grammar-compressed text. In *Proc. 23rd DCC*, pages 451–460, 2013.

[44] I. Tomohiro, T. Nishimoto, S. Inenaga, H. Bannai, and M. Takeda. Compressed automata for dictionary matching. *Theor. Comput. Sci.*, 578:30–41, 2015.

[45] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory Comput. Syst.*, 10(1):99–127, 1976.

[46] E. Verbin and W. Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th CPM*, pages 247–258, 2013.

[47] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[48] E. H. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – part one: Without context models. *IEEE Trans. Inf. Theory*, 46(3):755–754, 2000.

[49] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

[50] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, 24(5):530–536, 1978.