Downloaded from orbit.dtu.dk on: Dec 20, 2017

Technical University of Denmark

Fingerprints in Compressed Strings

Bille, Philip; Cording, Patrick Hagge; Gørtz, Inge Li; Sach, Benjamin; Vildhøj, Hjalte Wedel; Vind, Søren Juhl

Published in:
Algorithms and Data Structures

Link to article, DOI:
10.1007/978-3-642-40104-6_13

Publication date:
2013

Link back to DTU Orbit

Citation (APA):
Bille, P., Cording, P. H., Gørtz, I. L., Sach, B., Vildhøj, H. W., & Vind, S. J. (2013). Fingerprints in Compressed Strings. In Algorithms and Data Structures: 13th International Symposium, WADS 2013, London, ON, Canada, August 12-14, 2013. Proceedings (pp. 146-157). Springer. (Lecture Notes in Computer Science, Vol. 8037). DOI: 10.1007/978-3-642-40104-6_13

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

arXiv:1305.2777v2 [cs.DS] 16 May 2013

Fingerprints in Compressed Strings*

Philip Bille
phbi@dtu.dk

Patrick Hagge Cording
phaco@dtu.dk

Inge Li Gørtz†
inge@dtu.dk

Benjamin Sach
sach@dcs.warwick.ac.uk

Hjalte Wedel Vildhøj
hwvi@dtu.dk

Søren Vind‡
sovi@dtu.dk

May 17, 2013

Abstract

The Karp-Rabin fingerprint of a string is a type of hash value that due to its strong properties has been used in many string algorithms. In this paper we show how to construct a data structure for a string S of size N compressed by a context-free grammar of size n that answers fingerprint queries. That is, given indices i and j, the answer to a query is the fingerprint of the substring $S[i, j]$. We present the first $O(n)$ space data structures that answer fingerprint queries without decompressing any characters. For Straight Line Programs (SLP) we get $O(\log N)$ query time, and for Linear SLPs (an SLP derivative that captures LZ78 compression and its variations) we get $O(\log \log N)$ query time. Hence, our data structures has the same time and space complexity as for random access in SLPs. We utilize the fingerprint data structures to solve the longest common extension problem in query time $O(\log N \log \ell)$ and $O(\log \ell \log \log \ell + \log \log N)$ for SLPs and Linear SLPs, respectively. Here, ℓ denotes the length of the LCE.

1 Introduction

Given a string S of size N and a Karp-Rabin fingerprint function ϕ, the answer to a FINGERPRINT(i, j) query is the fingerprint $\phi(S[i, j])$ of the substring $S[i, j]$. We consider the problem of constructing a data structure that efficiently answers fingerprint queries when the string is compressed by a context-free grammar of size n.

The fingerprint of a string is an alternative representation that is much shorter than the string itself. By choosing the fingerprint function randomly at runtime it exhibits strong guarantees for the probability of two different strings having different fingerprints. Fingerprints were introduced by Karp and Rabin [22] and used to design a randomized string matching algorithm. Since then, they have been used as a central tool to design algorithms for a wide range of problems (see e.g., [2, 3, 11–13, 15, 17, 21, 24]).

A fingerprint requires constant space and it has the useful property that given the fingerprints $\phi(S[1, i-1])$ and $\phi(S[1, j])$, the fingerprint $\phi(S[i, j])$ can be computed in constant time. By storing the fingerprints $\phi(S[1, i])$ for $i = 1 \dots N$ a query can be answered in $O(1)$ time. However, this data

*An extended abstract of this paper appeared at the 13th Algorithms and Data Structures Symposium.

†Supported by a grant from the Danish Council for Independent Research | Natural Sciences.

‡Supported by a grant from the Danish National Advanced Technology Foundation.

structure uses $O(N)$ space which can be exponential in n. Another approach is to use the data structure of Gąsieniec et al. [18] which supports linear time decompression of a prefix or suffix of the string generated by a node. To answer a query we find the deepest node that generates a string containing $S[i]$ and $S[j]$ and decompress the appropriate suffix of its left child and prefix of its right child. Consequently, the space usage is $O(n)$ and the query time is $O(h + j - i)$, where h is the height of the grammar. The $O(h)$ time to find the correct node can be improved to $O(\log N)$ using the data structure by Bille et al. [8] giving $O(\log N + j - i)$ time for a $\textsc{Fingerprint}(i, j)$ query. Note that the query time depends on the length of the decompressed string which can be large.

We present the first data structures that answers fingerprint queries on grammar compressed strings without decompressing any characters, and improve all of the above time-space trade-offs. Assume without loss of generality that the compressed string is given as a Straight Line Program (SLP). An SLP is a grammar in Chomsky normal form, i.e., each nonterminal has exactly two children. A Linear SLP is an SLP where the root is allowed to have more than two children, and for all other internal nodes, the right child must be a leaf. Linear SLPs capture the LZ78 compression scheme [29] and its variations. Our data structures give the following theorem.

Theorem 1 *Let S be a string of length N compressed into an SLP G of size n. We can construct data structures that support* Fingerprint *queries in:*

(i) $O(n)$ space and query time $O(\log N)$

(ii) $O(n)$ space and query time $O(\log \log N)$ if G is a Linear SLP

Hence, we show a data structure for fingerprint queries that has the same time and space complexity as for random access in SLPs.

Our fingerprint data structures are based on the idea that a random access query for i produces a path from the root to a leaf labelled $S[i]$. The concatenation of the substrings produced by the left children of the nodes on this path produce the prefix $S[1, i]$. We store the fingerprints of the strings produced by each node and concatenate these to get the fingerprint of the prefix instead. For Theorem 1(i), we combine this with the fast random access data structure by Bille et al. [8]. For Linear SLPs we use the fact that the production rules form a tree to do large jumps in the SLP in constant time using a level ancestor data structure. Then a random access query is dominated by finding the node that produces $S[i]$ among the children of the root, which can be modelled as the predecessor problem.

Furthermore, we show how to obtain faster query time in Linear SLPs using finger searching techniques. Specifically, a finger for position i in a Linear SLP is a pointer to the child of the root that produces $S[i]$.

Theorem 2 *Let S be a string of length N compressed into an SLP G of size n. We can construct an $O(n)$ space data structure such that given a finger f for position i or j, we can answer a* $\textsc{Fingerprint}(i, j)$ *query in time $O(\log \log D)$ where $D = |i - j|$.*

Along the way we give a new and simple reduction for solving the finger predecessor problem on integers using any predecessor data structure as a black box.

In compliance with all related work on grammar compressed strings, we assume that the model of computation is the RAM model with a word size of $\log N$ bits.

1.1 Longest common extension in compressed strings

As an application we show how to efficiently solve the longest common extension problem (LCE). Given two indices i, j in a string S, the answer to the LCE(i, j) query is the length ℓ of the maximum substring such that $S[i, i+\ell] = S[j, j+\ell]$. The compressed LCE problem is to preprocess a compressed string to support LCE queries. On uncompressed strings this is solvable in $O(N)$ preprocessing time, $O(N)$ space, and $O(1)$ query time with a nearest common ancestor data structure on the suffix tree for S [20]. Other trade-offs are obtained by doing an exponential search over the fingerprints of strings starting in i and j [7]. Using the exponential search in combination with the previously mentioned methods for obtaining fingerprints without decompressing the entire string we get $O((h+\ell)\log \ell)$ or $O((\log N + \ell)\log \ell)$ time using $O(n)$ space for an LCE query. Using our new (finger) fingerprint data structures and the exponential search we obtain Theorem 3.

Theorem 3 *Let G be an SLP of size n that produces a string S of length N. The SLP G can be preprocessed in $O(N)$ time into a Monte Carlo data structure of size $O(n)$ that supports LCE queries on S in*

(i) $O(\log \ell \log N)$ *time*

(ii) $O(\log \ell \log\log \ell + \log\log N)$ *time if G is a Linear SLP.*

Here ℓ denotes the LCE value and queries are answered correctly with high probability. Moreover, a Las Vegas version of both data structures that always answers queries correctly can be obtained with $O(N^2/n \log N)$ preprocessing time with high probability.

We furthermore show how to reduce the Las Vegas preprocessing time to $O(N \log N \log\log N)$ when all the internal nodes in the Linear SLP are children of the root (which is the case in LZ78).

The following corollary follows immediately because an LZ77 compression [28] consisting of n phrases can be transformed to an SLP with $O(n \log \frac{N}{n})$ production rules [9, 25].

Corollary 1 *We can solve the* LCE *problem in $O(n \log \frac{N}{n})$ space and query time $O(\log \ell \log N)$ for LZ77 compression.*

Finally, the LZ78 compression can be modelled by a Linear SLP G_L with constant overhead. Consider an LZ78 compression with n phrases, denoted $r_1, \ldots, r_n$. A terminal phrase corresponds to a leaf in G_L, and each phrase $r_j = (r_i, a)$, $i < j$, corresponds to a node $v \in G_L$ with r_i corresponding to the left child of v and the right child of v being the leaf corresponding to a. Therefore, we get the following corollary.

Corollary 2 *We can solve the* LCE *problem in $O(n)$ space and query time $O(\log \ell \log\log \ell + \log\log N)$ for LZ78 compression.*

2 Preliminaries

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.

A Straight Line Program (SLP) G is a context-free grammar in Chomsky normal form that we represent as a node-labeled and ordered directed acyclic graph. Each leaf in G is labelled with

a character, and corresponds to a terminal grammar production rule. Each internal node in G is labeled with a nonterminal rule from the grammar. The unique string $S(v)$ of length $\mathit{size}(v) = |S(v)|$ is *produced* by a depth-first left-to-right traversal of $v \in G$ and consist of the characters on the leafs in the order they are visited. We let $\mathit{root}(G)$ denote the root of G, and $\mathit{left}(v)$ and $\mathit{right}(v)$ denote the left and right child of an internal node $v \in G$, respectively.

A Linear SLP G_L is an SLP where we allow $\mathit{root}(G_L)$ to have more than two children. All other internal nodes $v \in G_L$ have a leaf as $\mathit{right}(v)$. Although similar, this is not the same definition as given for the Relaxed SLP by Claude and Navarro [10]. The Linear SLP is more restricted since the right child of any node (except the root) must be a leaf. Any Linear SLP can be transformed into an SLP of at most double size by adding a new rule for each child of the root.

We extend the classic *heavy path decomposition* of Harel and Tarjan [20] to SLPs as in [8]. For each node $v \in G$, we select one edge from v to a child with maximum size and call it the *heavy edge*. The remaining edges are *light edges*. Observe that $\mathit{size}(u) \leq \mathit{size}(v)/2$ if v is a parent of u and the edge connecting them is light. Thus, the number of light edges on any path from the root to a leaf is at most $O(\log N)$. A *heavy path* is a path where all edges are heavy. The heavy path of a node v, denoted $H(v)$, is the unique path of heavy edges starting at v. Since all nodes only have a single outgoing heavy edge, the heavy path $H(v)$ and its leaf $\mathit{leaf}(H(v))$, is well-defined for each node $v \in G$.

A *predecessor data structure* supports predecessor and successor queries on a set $R \subseteq U = \{0, \ldots, N-1\}$ of n integers from a universe U of size N. The answer to a *predecessor query* $\textsc{pred}(q)$ is the largest integer $r^- \in R$ such that $r^- \leq q$, while the answer to a *successor query* $\textsc{succ}(q)$ is the smallest integer $r^+ \in R$ such that $r^+ \geq q$. There exist predecessor data structures achieving a query time of $O(\log \log N)$ using space $O(n)$ [23, 26, 27].

Given a rooted tree T with n vertices, we let $\mathit{depth}(v)$ denote the length of the path from the root of T to a node $v \in T$. A *level ancestor data structure* on T supports *level ancestor queries* $\mathrm{LA}(v, i)$, asking for the ancestor u of $v \in T$ such that $\mathit{depth}(u) = \mathit{depth}(v) - i$. There is a level ancestor data structure answering queries in $O(1)$ time using $O(n)$ space [14] (see also [1, 5, 6]).

2.1 Fingerprinting

The Karp-Rabin fingerprint [22] 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 1 *Let $x = yz$ be a string decomposable into a prefix y and suffix z. Let N be the maximum length of x, c be a random integer and $2N^{c+4} \leq p \leq 4N^{c+4}$ be a prime. 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 as follows:*

$$\phi(x) = \phi(y) \oplus \phi(z) = \phi(y) + c^{|y|} \cdot \phi(z) \bmod p$$

$$\phi(y) = \phi(x) \ominus_s \phi(z) = \phi(x) - \frac{c^{|x|}}{c^{|z|}} \cdot \phi(z) \bmod p$$

$$\phi(z) = \phi(x) \ominus_p \phi(y) = \frac{\phi(x) - \phi(y)}{c^{|y|}} \bmod p$$

In order to calculate the fingerprints of Lemma 1 in constant time, each fingerprint for a string x must also store the associated exponent $c^{|x|} \bmod p$, and we will assume this is always the case. Observe that a fingerprint for any substring $\phi(S[i, j])$ of a string can be calculated by subtracting the two fingerprints for the prefixes $\phi(S[1, i-1])$ and $\phi(S[1, j])$. Hence, we will only show how to find fingerprints for prefixes in this paper.

3 Basic fingerprint queries in SLPs

We now describe a simple data structure for answering $\textsc{Fingerprint}(1, i)$ queries for a string S compressed into a SLP G in time $O(h)$, where h is the height of the parse tree for S. This method does not unpack the string to obtain the fingerprint, instead the fingerprint is generated by traversing G.

The data structure stores $size(v)$ and the fingerprint $\phi(S(v))$ of the string produced by each node $v \in G$. To compose the fingerprint $f = \phi(S[1, i])$ we start from the root of G and do the following. Let v' denote the currently visited node, and let $p = 0$ be a variable denoting the size the concatenation of strings produced by left children of visited nodes. We follow an edge to the right child of v' if $p + size(left(v')) < i$, and follow a left edge otherwise. If following a right edge, update $f = f \oplus \phi(S(left(v')))$ such that the fingerprint of the full string generated by the left child of v' is added to f, and set $p = p + size(left(v'))$. When following a left edge, f and p remains unchanged. When a leaf is reached, let $f = f \oplus \phi(S(v'))$ to include the fingerprint of the terminal character. Aside from the concatenation of fingerprints for substrings, this procedure resembles a random access query for the character in position i of S.

The procedure correctly composes $f = \phi(S[1, i])$ because the order in which the fingerprints for the substrings are added to f is identical to the order in which the substrings are decompressed when decompressing $S[1, i]$.

Since the fingerprint composition takes constant time per addition, the time spent generating a fingerprint using this method is bounded by the height of the parse tree for $S[i]$, denoted $O(h)$. Only constant additional space is spent for each node in G, so the space usage is $O(n)$.

4 Faster fingerprints in SLPs

Using the data structure of Bille et al. [8] to perform random access queries allows for a faster way to answer $\textsc{Fingerprint}(1, i)$ queries.

Lemma 2 ([8]) *Let S be a string of length N compressed into a SLP G of size n. Given a node $v \in G$, we can support random access in $S(v)$ in $O(\log(size(v)))$ time, at the same time reporting the sequence of heavy paths and their entry- and exit points in the corresponding depth-first traversal of $G(v)$.*

The main idea is to compose the final fingerprint from substring fingerprints by performing a constant number of fingerprint additions per heavy path visited.

In order to describe the data structure, we will use the following notation. Let $V(v)$ be the left children of the nodes in $H(v)$ where the heavy path was extended to the right child, ordered by increasing depth. The order of nodes in $V(v)$ is equal to the sequence in which they occur when decompressing $S(v)$, so the concatenation of the strings produced by nodes in $V(v)$ yields the

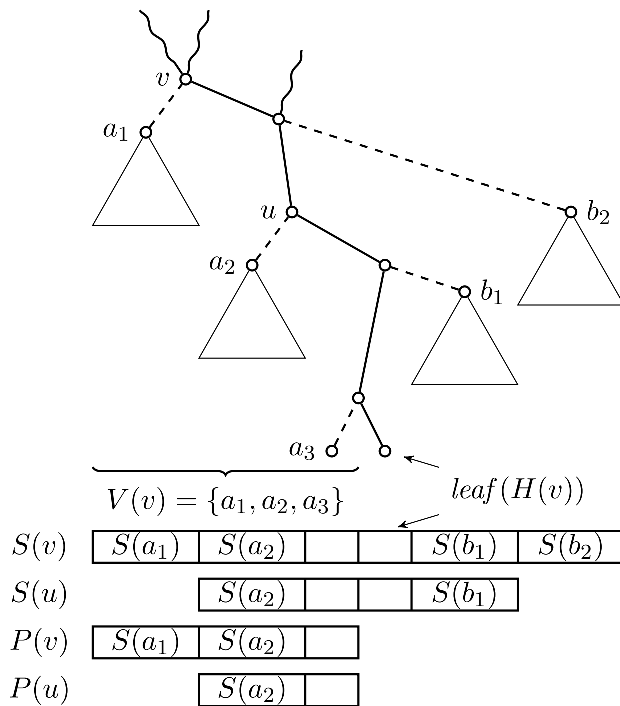

Figure 1: Figure showing how $S(v)$ and its prefix $P(v)$ is composed of substrings generated by the left children a_1, a_2, a_3 and right children b_1, b_2 of the heavy path $H(v)$. Also illustrates how this relates to $S(u)$ and $P(u)$ for a node $u \in H(v)$.

prefix $P(v) = S(v)[1, L(v)]$, where $L(v) = \sum_{u \in V(v)} size(u)$. Observe that $P(u)$ is a suffix of $P(v)$ if $u \in H(v)$. See Figure 1 for the relationship between u, v and the defined strings.

Let each node $v \in G$ store its unique outgoing heavy path $H(v)$, the length $L(v)$, $size(v)$, and the fingerprints $\phi(P(v))$ and $\phi(S(v))$. By forming heavy path trees of total size $O(n)$ as in [8], we can store $H(v)$ as a pointer to a node in a heavy path tree (instead of each node storing the full sequence).

The fingerprint $f = \phi(S[1, i])$ is composed from the sequence of heavy paths visited when performing a single random access query for $S[i]$ using Lemma 2. Instead of adding all left-children of the path towards $S[i]$ to f individually, we show how to add all left-children hanging from each visited heavy path in constant time per heavy path. Thus, the time taken to compose f is $O(\log N)$.

More precisely, for the pair of entry- and exit-nodes v, u on each heavy path H traversed from the root to $S[i]$, we set $f = f \oplus (\phi(P(v)) \ominus_s \phi(P(u))$ (which is allowed because $P(u)$ is a suffix of $P(v)$). If we leave u by following a right-pointer, we additionally set $f = f \oplus \phi(S(left(u)))$. If u is a leaf, set $f = f \oplus \phi(S(u))$ to include the fingerprint of the terminal character.

Remember that $P(v)$ is exactly the string generated from v along H, produced by the left children of nodes on H where the heavy path was extended to the right child. Thus, this method corresponds exactly to adding the fingerprint for the substrings generated by all left children of nodes on H between the entry- and exit-nodes in depth-first order, and the argument for correctness from the slower fingerprint generation also applies here.

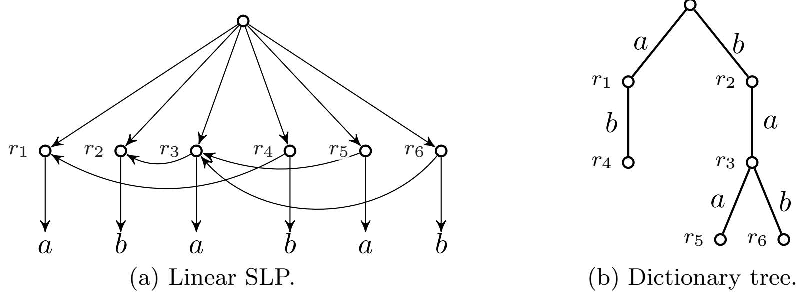

Figure 2: A Linear SLP compressing the string `abbaabbaabab` and the dictionary tree obtained from the Linear SLP.

Since the fingerprint composition takes constant time per addition, the time spent generating a fingerprint using this method is bounded by the number of heavy paths traversed, which is $O(\log N)$. Only constant additional space is spent for each node in G, so the space usage is $O(n)$. This concludes the proof of Theorem 1(i).

5 Faster fingerprints in Linear SLPs

In this section we show how to quickly answer FINGERPRINT$(1, i)$ queries on a Linear SLP G_L. In the following we denote the sequence of k children of $root(G_L)$ from left to right by $r_1, \ldots, r_k$. Also, let $R(j) = \sum_{m=1}^{j} size(r_m)$ for $j = 0, \ldots, k$. That is, $R(j)$ is the length of the prefix of S produced by G_L including r_j (and $R(0)$ is the empty prefix).

We also define the dictionary tree F over G_L as follows. Each node $v \in G_L$ corresponds to a single vertex $v^F \in F$. There is an edge (u^F, v^F) labeled c if $u = left(v)$ and $c = S(right(v))$. If v is a leaf, there is an edge $(root(F), v^F)$ labeled $S(v)$. That is, a left child edge of $v \in G_L$ is converted to a parent edge of $v^F \in F$ labeled like the right child leaf of v. Note that for any node $v \in G_L$ except the root, producing $S(v)$ is equivalent to following edges and reporting edge labels on the path from $root(F)$ to v^F. Thus, the prefix of length a of $S(v)$ may be produced by reporting the edge labels on the path from $root(F)$ until reaching the ancestor of v^F at depth a.

The data structure stores a predecessor data structure over the prefix lengths $R(j)$ and the associated node r_j and fingerprint $\phi(S[1, R(j)])$ for $j = 0, \ldots, k$. We also have a doubly linked list of all r_j's with bidirectional pointers to the predecessor data structure and G_L. We store the dictionary tree F over G_L, augment it with a level ancestor data structure, and add bidirectional pointers between $v \in G_L$ and $v^F \in F$. Finally, for each node $v \in G_L$, we store the fingerprint of the string it produces, $\phi(S(v))$.

A query FINGERPRINT$(1, i)$ is answered as follows. Let $R(m)$ be the predecessor of i among $R(0), R(1), \ldots, R(k)$. Compose the answer to FINGERPRINT$(1, i)$ from the two fingerprints $\phi(S[1, R(m)]) \oplus \phi(S[R(m)+1, i])$. The first fingerprint $\phi(S[1, R(m)])$ is stored in the data structure and the second fingerprint $\phi(S[R(m)+1, i])$ can be found as follows. Observe that $S[R(m)+1, i]$ is fully generated by r_{m+1} and hence a prefix of $S(r_{m+1})$ of length $i - R(m)$. We can get r_{m+1} in constant time from r_m using the doubly linked list. We use a level ancestor query $u^F = \mathrm{LA}(r^F_{m+1}, i - R(m))$ to determine the ancestor of r^F_{m+1} at depth $i - R(m)$, corresponding to a prefix of r_{m+1} of the correct length. From u_F we can find $\phi(S(u)) = \phi(S[R(m) + 1, i])$.

It takes constant time to find $\phi(S[R(m)+1,i])$ using a single level ancestor query and following pointers. Thus, the time to answer a query is bounded by the time spent determining $\phi(S[1,R(m)])$, which requires a predecessor query among k elements (i.e. the number of children of $root(G_L)$) from a universe of size N. The data structure uses $O(n)$ space, as there is a bijection between nodes in G_L and vertices in F, and we only spend constant additional space per node in G_L and vertex in F. This concludes the proof of Theorem 1(ii).

6 Finger fingerprints in Linear SLPs

The $O(\log\log N)$ running time of a Fingerprint$(1,i)$ query is dominated by having to find the predecessor $R(m)$ of i among $R(0), R(1), \ldots, R(k)$. Given $R(m)$ the rest of the query takes constant time. In the following, we show how to improve the running time of a Fingerprint$(1,i)$ query to $O(\log\log|j-i|)$ given a finger for position j. Recall that a finger f for a position j is a pointer to the node r_m producing $S[j]$. To achieve this, we present a simple linear space finger predecessor data structure that is interchangeable with any other predecessor data structure.

6.1 Finger Predecessor

Let $R \subseteq U = \{0,\ldots,N-1\}$ be a set of n integers from a universe U of size N. Given a finger $f \in R$ and a query point $q \in U$, the *finger predecessor problem* is to answer finger predecessor or successor queries in time depending on the universe distance $D = |f-q|$ from the finger to the query point. Belazzougui et al. [4] present a succinct solution for solving the finger predecessor problem relying on a modification of z-fast tries. Here, we use a simple reduction for solving the finger predecessor problem using any predecessor data structure as a black box.

Lemma 3 *Let $R \subseteq U = \{0,\ldots,N-1\}$ be a set of n integers from a universe U of size N. Given a predecessor data structure with query time $t(N,n)$ using $s(N,n)$ space, we can solve the finger predecessor problem in time $O(t(D,n))$ using space $O(s(N,\frac{n}{\log N})\log N)$.*

Proof. Construct a complete balanced binary search tree T over the universe U. The leaves of T represent the integers in U, and we say that a vertex *span* the range of U represented by the leaves in its subtree. Mark the leaves of T representing the integers in R. We remove all vertices in T where the subtree contains no marked vertices. Observe that a vertex at height j span a universe range of size $O(2^j)$. We augment T with a level ancestor data structure answering queries in constant time. Finally, left- and right-neighbour pointers are added for all nodes in T.

Each internal node $v \in T$ at height j store an instance of the given predecessor data structure for the set of marked leaves in the subtree of v. The size of the universe for the predecessor data structure equals the span of the vertex and is $O(2^j)$[1].

Given a finger $f \in R$ and a query point $q \in U$, we will now describe how to find both succ(q) and pred(q) when $q < f$. The case $q > f$ is symmetric. Observe that f corresponds to a leaf in T, denoted f_l. We answer a query by determining the ancestor v of f_l at height $h = \lceil\log(|f-q|)\rceil$ and its left neighbour v_L (if it exists). We query for succ(q) in the predecessor data structures of both v and v_L, finding at least one leaf in T (since v spans f and $q < f$). We return the leaf representing the smallest result as succ(q) and its left neighbour in T as pred(q).

[1] The integers stored by the data structure may be shifted by some constant $k \cdot 2^j$ for a vertex at height j, but we can shift all queries by the same constant and thus the size of the universe is 2^j.

Observe that the predecessor data structures in v and v_L each span a universe of size $O(2^h) = O(|f-q|) = O(D)$. All other operations performed take constant time. Thus, for a predecessor data structure with query time $t(N,n)$, we can answer finger predecessor queries in time $O(t(D,n))$.

The height of T is $O(\log N)$, and there are $O(n \log N)$ vertices in T (since vertices spanning no elements from R are removed). Each element from R is stored in $O(\log N)$ predecessor data structures. Hence, given a predecessor data structure with space usage $s(N,n)$, the total space usage of the data structure is $O(s(N,n)\log N)$.

We reduce the size of the data structure by reducing the number of elements it stores to $O(\frac{n}{\log N})$. This is done by partitioning R into $O(\frac{n}{\log N})$ sets of consecutive elements R_i of size $O(\log N)$. We choose the largest integer in each R_i set as the representative g_i for that set, and store that in the data structure described above. We store the integers in set R_i in an atomic heap [16,19] capable of answering predecessor queries in $O(1)$ time and linear space for a set of size $O(\log N)$. Each element in R keep a pointer to the set R_i it belongs to, and each set left- and right-neighbour pointers.

Given a finger $f \in R$ and a query point $q \in U$, we describe how to determine $\textsc{pred}(q)$ and $\textsc{succ}(q)$ when $q < f$. The case $q > f$ is symmetric. We first determine the closest representatives g_l and g_r on the left and right of f, respectively. Assuming $q < g_l$, we proceed as before using g_l as the finger into T and query point q. This gives $p = \textsc{pred}(q)$ and $s = \textsc{succ}(q)$ among the representatives. If g_l is undefined or $g_l < q < f \leq g_r$, we select $p = g_l$ and $s = g_r$. To produce the final answers, we perform at most 4 queries in the atomic heaps that p and s are representatives for.

All queries in the atomic heaps take constant time, and we can find g_l and g_r in constant time by following pointers. If we query a predecessor data structure, we know that the range it spans is $O(|g_l - q|) = O(|f-q|) = O(D)$ since $q < g_l < f$. Thus, given a predecessor data structure with query time $t(N,n)$, we can solve the finger predecessor problem in time $O(t(D,n))$.

The total space spent on the atomic heaps is $O(n)$ since they partition R. The number of representatives is $O(\frac{n}{\log N})$. Thus, given a predecessor data structure with space usage $s(N,n)$, we can solve the finger predecessor problem in space $O(s(N, \frac{n}{\log N})\log N)$. $\square$

Using the van Emde Boas predecessor data structure [23,26,27] with $t(N,n) = O(\log\log N)$ query time using $s(N,n) = O(n)$ space, we obtain the following corollary.

Corollary 3 *Let $R \subseteq U = \{0,\ldots,N-1\}$ be a set of n integers from a universe U of size N. Given a finger $f \in R$ and a query point $q \in U$, we can solve the finger predecessor problem in time $O(\log\log|f-q|)$ and space $O(n)$.*

6.2 Finger Fingerprints

We can now prove Theorem 2. Assume wlog that we have a finger for i, i.e., we are given a finger f to the node r_m generating $S[i]$. From this we can in constant time get a pointer to r_{m+1} in the doubly linked list and from this a pointer to $R(m+1)$ in the predecessor data structure. If $R(m+1) > j$ then $R(m)$ is the predecessor of j. Otherwise, using Corollary 3 we can in time $O(\log\log|R(m+1)-j|)$ find the predecessor of j. Since $R(m+1) \geq i$ and the rest of the query takes constant time, the total time for the query is $O(\log\log|i-j|)$.

7 Longest Common Extensions in Compressed Strings

Given an SLP G, the longest common extension (LCE) problem is to build a data structure for G that supports longest common extension queries $LCE(i, j)$. In this section we show how to use our fingerprint data structures as a tool for doing LCE queries and hereby obtain Theorem 3.

7.1 Computing Longest Common Extensions with Fingerprints

We start by showing the following general lemma that establishes the connection between LCE and fingerprint queries.

Lemma 4 *For any string S and any partition $S = s_1 s_2 \cdots s_t$ of S into k non-empty substrings called phrases, $\ell = \mathrm{LCE}(i, j)$ can be found by comparing $O(\log \ell)$ pairs of substrings of S for equality. Furthermore, all substring comparisons $x = y$ are of one of the following two types:*

Type 1 *Both x and y are fully contained in (possibly different) phrase substrings.*

Type 2 $|x| = |y| = 2^p$ *for some $p = 0, \ldots, \log(\ell) + 1$ and for x or y it holds that*

- *(a) The start position is also the start position of a phrase substring, or*
- *(b) The end position is also the end position of a phrase substring.*

Proof. Let a position of S be a *start* (*end*) position if a phrase starts (ends) at that position. Moreover, let a comparison of two substrings be of *type 1* (*type 2*) if it satisfies the first (second) property in the lemma. We now describe how to find $\ell = \mathrm{LCE}(i, j)$ by using $O(\log \ell)$ type 1 or 2 comparisons.

If i or j is not a start position, we first check if $S[i, i + k] = S[j, j + k]$ (type 1), where $k \geq 0$ is the minimum integer such that $i + k$ or $j + k$ is an end position. If the comparison fails, we have restricted the search for ℓ to two phrase substrings, and we can find the exact value using $O(\log \ell)$ type 1 comparisons.

Otherwise, $\mathrm{LCE}(i, j) = k + \mathrm{LCE}(i + k + 1, j + k + 1)$ and either $i + k + 1$ or $j + k + 1$ is a start position. This leaves us with the task of describing how to answer $\mathrm{LCE}(i, j)$, assuming that either i or j is a start position.

We first use $p = O(\log \ell)$ type 2 comparisons to determine the biggest integer p such that $S[i, i + 2^p] = S[j, j + 2^p]$. It follows that $\ell \in [2^p, 2^{p+1}]$. Now let $q < 2^p$ denote the length of the longest common prefix of the substrings $x = S[i + 2^p + 1, i + 2^{p+1}]$ and $y = S[j + 2^p + 1, j + 2^{p+1}]$, both of length 2^p. Clearly, $\ell = 2^p + q$. By comparing the first half x' of x to the first half y' of y, we can determine if $q \in [0, 2^{p-1}]$ or $q \in [2^{p-1} + 1, 2^p - 1]$. By recursing we obtain the exact value of q after $\log 2^p = O(\log \ell)$ comparisons.

However, comparing $x' = S[a_1, b_1]$ and $y' = S[a_2, b_2]$ directly is not guaranteed to be of type 1 or 2. To fix this, we compare them indirectly using a type 1 and type 2 comparison as follows. Let $k < 2^p$ be the minimum integer such that $b_1 - k$ or $b_2 - k$ is a start position. If there is no such k then we can compare x' and y' directly as a type 1 comparison. Otherwise, it holds that $x' = y'$ if and only if $S[b_1 - k, b_1] = S[b_2 - k, b_2]$ (type 1) and $S[a_1 - k - 1, b_1 - k - 1] = S[a_2 - k - 1, b_2 - k - 1]$ (type 2). □

Theorem 3 follows by using fingerprints to perform the substring comparisons. In particular, we obtain a Monte Carlo data structure that can answer a LCE query in $O(\log \ell \log N)$ time for SLPs and in $O(\log \ell \log\log N)$ time for Linear SLPs. In the latter case, we can use Theorem 2 to reduce the query time to $O(\log \ell \log\log \ell + \log\log N)$ by observing that for all but the first fingerprint query, we have a finger into the data structure.

7.2 Verifying the Fingerprint Function

Since the data structure is Monte Carlo, there may be collisions among the fingerprints used to determine the LCE, and consequently the answer to a query may be incorrect. We now describe how to obtain a Las Vegas data structure that always answers LCE queries correctly. We do so by showing how to efficiently verify that the fingerprint function ϕ is *good*, i.e., collision-free on all substrings compared in the computation of $\mathrm{LCE}(i, j)$. We give two verification algorithms. One that works for LCE queries in SLPs, and a faster one that works for Linear SLPs where all internal nodes are children of the root (e.g. LZ78).

7.2.1 SLPs

If we let the phrases of S be its individual characters, we can assume that all fingerprint comparisons are of type 2 (see Lemma 4). We thus only have to check that ϕ is collision-free among all substrings of length $2^p, p = 0, \ldots, \log N$. We verify this in $\log N$ rounds. In round p we maintain the fingerprint of a sliding window of length 2^p over S. For each substring x we insert $\phi(x)$ into a dictionary. If the dictionary already contains a fingerprint $\phi(y) = \phi(x)$, we verify that $x = y$ in constant time by checking if $\phi(x[1, 2^{p-1}]) = \phi(y[1, 2^{p-1}])$ and $\phi(x[2^{p-1} + 1, 2^p]) = \phi(y[2^{p-1} + 1, 2^p])$. This works because we have already verified that the fingerprinting function is collision-free for substrings of length 2^{p-1}. Note that we can assume that for any fingerprint $\phi(x)$ the fingerprints of the first and last half of x are available in constant time, since we can store and maintain these at no extra cost. In the first round $p = 0$, we check that $x = y$ by comparing the two characters explicitly. If $x \neq y$ we have found a collision and we abort and report that ϕ is not good. If all rounds are successfully verified, we report that ϕ is good.

For the analysis, observe that computing all fingerprints of length 2^p in the sliding window can be implemented by a single traversal of the SLP parse tree in $O(N)$ time. Thus, the algorithm correctly decides whether ϕ is good in $O(N \log N)$ time and $O(N)$ space. We can easily reduce the space to $O(n)$ by carrying out each round in $O(N/n)$ iterations, where no more than n fingerprints are stored in the dictionary in each iteration. So, alternatively, ϕ can be verified in $O(N^2/n \log N)$ time and $O(n)$ space.

7.2.2 Linear SLPs

In Linear SLPs where all internal nodes are children of the root, we can reduce the verification time to $O(N \log N \log\log N)$, while still using $O(n)$ space. To do so, we use Lemma 4 with the partition of S being the root substrings. We verify that ϕ is collision-free for type 1 and type 2 comparisons separately.

Type 1 Comparisons. We carry out the verification in rounds. In round p we check that no collisions occur among the p-length substrings of the root substrings as follows: We traverse the

SLP maintaining the fingerprint of all p-length substrings. For each substring x of length p, we insert $\phi(x)$ into a dictionary. If the dictionary already contains a fingerprint $\phi(y) = \phi(x)$ we verify that $x = y$ in constant time by checking if $x[1] = y[1]$ and $\phi(x[2, |x|]) = \phi(y[2, |y|])$ (type 1).

Every substring of a root substring ends in a leaf in the SLP and is thus a suffix of a root substring. Consequently, they can be generated by a bottom up traversal of the SLP. The substrings of length 1 are exactly the leaves. Having generated the substrings of length p, the substrings of length $p+1$ are obtained by following the parents left child to another root node and prepending its right child. In each round the p length substrings correspond to a subset of the root nodes, so the dictionary never holds more than n fingerprints. Furthermore, since each substring is a suffix of a root substring, and the root substrings have at most N suffixes in total, the algorithm will terminate in $O(N)$ time.

Type 2 Comparisons. We adopt an approach similar to that for SLPs and verify ϕ in $O(\log N)$ rounds. In round p we store the fingerprints of the substrings of length 2^p that start or end at a phrase boundary in a dictionary. We then slide a window of length 2^p over S to find the substrings whose fingerprint equals one of those in the dictionary. Suppose the dictionary in round p contains the fingerprint $\phi(y)$, and we detect a substring x such that $\phi(x) = \phi(y)$. To verify that $x = y$, assume that y starts at a phrase boundary (the case when it ends in a phrase boundary is symmetric). As before, we first check that the first half of x is equal to the first half of y using fingerprints of length 2^{p-1}, which we know are collision-free. Let $x' = S[a_1, b_1]$ and $y' = S[a_2, b_2]$ be the second half of x and y. Contrary to before, we can not directly compare $\phi(x') = \phi(y')$, since neither x' nor y' is guaranteed to start or end at a phrase boundary. Instead, we compare them indirectly using a type 1 and type 2 comparison as follows: Let $k < 2^{p-1}$ be the minimum integer such that $b_1 - k$ or $b_2 - k$ is a start position. If there is no such k then we can compare x' and y' directly as a type 1 comparison. Otherwise, it holds that $x' = y'$ if and only if $\phi(S[b_1 - k, b_1]) = \phi(S[b_2 - k, b_2])$ (type 1) and $\phi(S[a_1 - k - 1, b_1 - k - 1]) = \phi(S[a_2 - k - 1, b_2 - k - 1])$ (type 2), since we have already verified that ϕ is collision-free for type 1 comparisions and type 2 comparisions of length 2^{p-1}.

The analysis is similar to that for SLPs. The sliding window can be implemented in $O(N)$ time, but for each window position we now need $O(\log \log N)$ time to retrieve the fingerprints, so the total time to verify ϕ for type 2 collisions becomes $O(N \log N \log \log N)$. The space is $O(n)$ since in each round the dictionary stores at most $O(n)$ fingerprints.

References

[1] S. Alstrup and J. Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Proc. 27th ICALP*, pages 73–84, 2000.

[2] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. In *Proc. 3rd CPM*, pages 262–275, 1992.

[3] A. Andoni and P. Indyk. Efficient algorithms for substring near neighbor problem. In *Proc. 17th SODA*, pages 1203–1212, 2006.

[4] D. Belazzougui, P. Boldi, and S. Vigna. Predecessor search with distance-sensitive query time. *arXiv:1209.5441*, 2012.

[5] M. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoret. Comput. Sci.*, 321:5–12, 2004.

[6] O. Berkman and U. Vishkin. Finding level-ancestors in trees. *J. Comput. System Sci.*, 48(2):214–230, 1994.

[7] P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space trade-offs for longest common extensions. In *Proc. 23rd CPM*, pages 293–305, 2012.

[8] P. Bille, G. Landau, R. Raman, K. Sadakane, S. Satti, and O. Weimann. Random access to grammar-compressed strings. In *Proc. 22nd SODA*, pages 373–389, 2011.

[9] 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.

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

[11] R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. *SIAM J. Comput.*, 33(1):26–42, 2003.

[12] G. Cormode and S. Muthukrishnan. Substring compression problems. In *Proc. 16th SODA*, pages 321–330, 2005.

[13] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2, 2007.

[14] P. F. Dietz. Finding level-ancestors in dynamic trees. In *Proc. 2nd WADS*, pages 32–40, 1991.

[15] M. Farach and M. Thorup. String matching in Lempel–Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[16] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47(3):424–436, 1993.

[17] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter. Randomized efficient algorithms for compressed strings: The finger-print approach. In *Proc. 7th CPM*, pages 39–49, 1996.

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

[19] T. Hagerup. Sorting and searching on the Word RAM. In *Proc. 15th STACS*, pages 366–398, 1998.

[20] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[21] A. Kalai. Efficient pattern-matching with don't cares. In *Proc. 13th SODA*, pages 655–656, 2002.

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

[23] K. Mehlhorn and S. Näher. Bounded ordered dictionaries in $O(\log\log N)$ time and $O(n)$ space. *Inform. Process. Lett.*, 35(4):183–189, 1990.

[24] B. Porat and E. Porat. Exact and approximate pattern matching in the streaming model. In *Proc. 50th FOCS*, pages 315–323, 2009.

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

[26] 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.

[27] D. Willard. Log-logarithmic worst-case range queries are possible in space Θ(N). *Inform. Process. Lett.*, 17(2):81–84, 1983.

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

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