Journal of Computer and System Sciences ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Fingerprints in compressed strings ☆

Philip Bille [a,*,1], Inge Li Gørtz [a,1], Patrick Hagge Cording [a,1], Benjamin Sach [b], Hjalte Wedel Vildhøj [a], Søren Vind [a]

[a] Technical University of Denmark, DTU Compute, Richard Petersens Plads, 2800 Kgs. Lyngby, Denmark
[b] Department of Computer Science, University of Bristol, Merchant Venturer's Building, BS81UB, United Kingdom

ARTICLE INFO

Article history:
Received 28 October 2013
Received in revised form 11 January 2017
Accepted 13 January 2017
Available online xxxx

Keywords:
Karp–Rabin fingerprints
Grammar compressed string
Longest common extensions

ABSTRACT

In this paper we show how to construct a data structure for a string S of size N compressed into a context-free grammar of size n that supports efficient Karp–Rabin fingerprint queries to any substring of S. 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. We extend the result 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

Karp–Rabin fingerprinting [1] is arguably the most well-known and widely used method for hashing strings. The key idea is to map substrings to a (large) numeric value by considering an sequence of characters as digits of an integer in base σ, where σ is the number of characters in the alphabet, and then hash this value into a polynomial universe. The hash value, called the *fingerprint* of the string, has constant size (in words) and is collision free with high probability when the hash function is chosen randomly at runtime.

Karp and Rabin [1] introduced fingerprinting to solve the classic *string matching problem*. Here we are given a text string S and a pattern string P and the goal is to identify all substrings of S that match P. To solve the problem the algorithm computes the fingerprint of P and compares it to the fingerprint of all substrings of length $|P|$ in S. To efficiently compute all fingerprints of length $|P|$ in S, the key observation is that the fingerprint for $S[i, i + |P|]$ can be computed from the fingerprint of $S[i - 1, i - 1 + |P|]$ in constant time using simple properties of the specific fingerprint function (discussed in detail in Sec. 2.1). This leads to a linear time (Monte Carlo) algorithm with a small probability of error. By rerunning and

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

* Corresponding author.
E-mail addresses: phbi@dtu.dk (P. Bille), inge@dtu.dk (I.L. Gørtz), phaco@dtu.dk (P.H. Cording), ben@cs.bris.ac.uk (B. Sach), hwvi@dtu.dk (H.W. Vildhøj), sovi@dtu.dk (S. Vind).

[1] Supported by the Danish Research Council (DFF4005-00267).

http://dx.doi.org/10.1016/j.jcss.2017.01.002
0022-0000/

rechecking the matches the algorithm can be converted into a linear expected time (Las Vegas) algorithm that makes no errors [1].

In addition to computing fingerprints of adjacent substrings as above, we can do other useful constant time string manipulations such as concatenation, substitution, and splits directly on the fingerprints without knowing the original strings. This has led to fingerprints being used as a central tool in algorithms for a wide and diverse range of problems, such as string matching, plagiarism detection, encryption, deduplication, bioinformatics, and data compression, see e.g., [2–16].

In all of the above scenarios the fingerprints are computed on *uncompressed strings*. In this paper, we consider the natural question of whether fingerprints can be efficiently computed on *compressed* strings.

Given a string S of size N and a Karp–Rabin fingerprint function ϕ, the answer to the fingerprint query, FINGERPRINT(i, j) 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 into a context-free grammar of size n.

As a motivating example, we first consider a simple but space inefficient approach, which utilises the aforementioned string manipulation properties of Karp–Rabin fingerprints. Specifically, the 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 $i = 1, 2, \ldots, N$, a query can be answered in $O(1)$ time. This data structure uses $\Theta(N)$ space which can be exponential in n. Another approach is to use the data structure of Gąsieniec et al. [17] 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. [18] giving $O(\log N + j - i)$ time for a FINGERPRINT(i, j) query. The query time of this approach is linear in the length of the decompressed substring, which can be large. It should be noted that in previous applications of Karp–Rabin fingerprints such as those cited above, the required fingerprints *are* typically of large substrings. In related work, for the special case of *balanced* grammars (by height or weight), Gagie et al. [19] showed how to efficiently compute fingerprints for Lempel–Ziv compressed strings.

We present the first data structures that answer fingerprint queries on general 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 widely-implemented LZ78 compression scheme [20] 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. [18]. 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. A finger for position i in a Linear SLP is a pointer to the child of the root that produces $S[i]$. To avoid potential confusion we clarify that the term finger is of independent origin to the term fingerprint and the similarity in their names is purely coincidental.

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* 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.