

# Algorithms and data structures for grammar-compressed strings

**Cording, Patrick Hagge**

*Publication date:*
2015

*Document Version*
Publisher's PDF, also known as Version of record

Technical University of Denmark

# Algorithms and Data Structures for Grammar-Compressed Strings

Patrick Hagge Cording

DTU Compute
Department of Applied Mathematics and Computer Science

PHD-2014-348

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, Building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

PHD-2014-357
ISSN: 0909-3192

# PREFACE

This doctoral dissertation was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfilment of the requirements for acquiring a doctoral degree. The results presented in the dissertation were obtained from December 2011 to December 2014 under supervision of Associate Professor Philip Bille and Associate Professor Inge Li Gørtz. The dissertation is comprised of three joint peer-reviewed publications and two unpublished results.

**Acknowledgements.** I first of all feel obliged to have become a part of the warm and welcoming community revolving around the topic of combinatorial pattern matching. My deepest gratitude goes out to Philip and Inge for providing me with academic and personal advice and for completely ripping apart my mental picture of what a professor looks like. During my three years I sometimes forgot if I was at work or hanging out with good friends. I owe this to my office comrades Hjalte Wedel Vildhøj, Søren Vind, Anders Roy Christiansen, Frederik Rye Skjoldjensen, and the rest the ALGOLOG section. I feel very fortunate to have spent five months at Haifa University hosted by Gad Landau and Oren Weimann. In this regard, thanks to Gad, Oren, Ofer, Cyril, Liat, Mika, Rogers, and Deepak for great spirit. I am also grateful for the patience of Oren Weimann, Benjamin Sach, and Pawel Gawrychowski whom I have had the pleasure of collaborating with. Last but not least, thanks to the algorithmic community in and around Copenhagen.

Patrick Hagge Cording
Copenhagen, December 2014

# ABSTRACT

This thesis presents new algorithms and data structures for handling data represented as grammar-compressed strings. The compression scheme we focus on is the Straight Line Program (SLP). In the following, $\mathcal{S}$ is an SLP of size $n$ compressing a string $S$ of size $N$. We consider the following problems.

**The q-gram profile of a compressed string.** We present an algorithm for computing the q-gram profile from a grammar-compressed string that runs in $O(N - \alpha)$ expected time and uses $O(n + q + k_{S,q})$ space. Here, $N - \alpha \leq qn$ is the exact number of characters decompressed by the algorithm and $k_{S,q} \leq N - \alpha$ is the number of distinct q-grams in $S$. This simultaneously matches the current best known time bound and improves the best known space bound. Our space bound is asymptotically optimal in the sense that any algorithm storing the grammar and the q-gram profile must use $\Omega(n + q + k_{S,q})$ space. To achieve this we introduce the q-gram graph that space-efficiently captures the structure of a string with respect to its q-grams, and show how to construct it from a grammar.

**Fingerprints and Longest Common Extensions in compressed strings.** 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 grammar-compressed string 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 SLPs 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, $\ell$ denotes the length of the LCE.

**Subsequence matching in compressed strings and the tree color problem.** We present a new algorithm for subsequence matching in grammar-compressed strings that uses $O(n + \frac{n\sigma}{w})$ space and $O(n + \frac{n\sigma}{w} + m \log N \log w \cdot occ)$ or $O(n + \frac{n\sigma}{w} \log w + m \log N \cdot occ)$ time. Here $w$ is the word size and $occ$ is the number of occurrences of the pattern. Our algorithm uses less space than previous algorithms and is also faster for $occ = o(\frac{n}{\log N})$ occurrences. The algorithm uses a new data structure that allows us to efficiently find the next occurrence of a given character after a given position in a compressed string. This data structure in turn is based on a new data structure for the tree color problem, where the node colors are packed in bit strings.

**Random access in compressed strings.** The random access problem for compressed strings is to build a data structure that efficiently supports accessing the character in

position $i$ of a string given in compressed form. We present a data structure using $O(n\Delta \log_\Delta \frac{N}{n} \log N)$ bits of space that supports accessing position $i$ of a grammar-compressed string in $O(\log_\Delta N)$ time for $\Delta \leq \log^{O(1)} N$. The query time is optimal for polynomially compressible strings, i.e., when $n = O(N^{1-\varepsilon})$.

**Bookmarking of compressed strings.** We consider the problem of storing an SLP and a set of positions $\{i_1, \ldots, i_b\}$ such that any substring of length $l$ starting from one of the positions can be decompressed in $O(l)$ time. Our solution requires space

$$O\left(\min_{1\leq\tau\leq\log^* N}\left\{\tau(n+b) + \min\{n,b\}\log^{(\tau)} N\right\}\right).$$

# DANISH ABSTRACT

Denne afhandling præsenterer nye algoritmer og datastrukturer til at håndtere data repræsenteret som en grammatik-komprimerede streng (tekst). Vi fokuserer på kompressionsmetoden kaldet Straight Line Program (SLP). I det resterende af dette afsnit er $\mathcal{S}$ en SLP af størrelse $n$ der komprimerer en streng $S$ af størrelse $N$. Vi studerer følgende problemer.

**Q-gram–profilen af komprimerede strenge.** Vi præsenterer en algoritme til at beregne q-gram profilen af en grammatik-komprimeret streng der kører i $O(N - \alpha)$ forventet tid og bruger $O(n + q + k_{S,q})$ plads. Her er $N - \alpha \leq qn$ det præcise antal karakterer som algoritmen dekomprimerer og $k_{S,q} \leq N - \alpha$ er antallet af unikke q-gram i $S$. Dette svarer til tidsgrænsen for den nuværende hurtigste algoritme og forbedrer pladsforbruget. Vores datastruktur er asymptotisk optimal idet enhver algoritme der både skal lagre grammatikken og q-gram–profilen skal bruge $\Omega(n + q + k_{S,q})$ plads. For at opnå dette introducerer vi q-gram–grafen, som beskriver strukturen i en text i forhold til dens q-gram på en pladseffektiv måde, og vi viser hvordan denne kan konstrueres fra en grammatik.

**Fingeraftryk og længste fælles præfiks i komprimerede strenge.** Et Karp-Rabin fingeraftryk af en streng er en slags hash-værdi som pga. sine stærke egenskaber anvendes i mange strengalgoritmer. I denne artikel viser vi hvordan man konstruerer en datastruktur til at svare på fingeraftrykforespørgsler for komprimerede strenge. Dvs., givet to positioner $i$ og $j$, så er en svaret på en forespørgsel fingeraftrykket for delstrengen $S[i, j]$. Vi præsenterer den første datastruktur der bruger $O(n)$ plads og kan svare på fingeraftrykforespørgsler uden at dekomprimere tegn fra strengen. For SLP'er tager en forespørgsel $O(\log N)$ tid, og for Lineære SLP'er (en variant af en SLP som modellerer LZ78 kompression og dens varianter) tager det $O(\log \log N)$ tid. Disse forespørgselstider svarer til tiden for random access i SLP'er. Vi udnytter fingerafstrykdatastrukturen til at løse længeste fælles præfiks-problemet i $O(\log N \log \ell)$ tid og $O(\log \ell \log \log \ell + \log \log N)$ tid for henholdsvis SLP'er and Lineære SLP'er. Her er $\ell$ længden af svaret på en længste fælles præfiks-forespørgsel.

**Søgning efter delsekvenser i komprimerede strenge og træfarveproblemet.** Vi præsenterer en ny algoritme til søgning efter delsekvenser i grammatik-komprimerede strenge som bruger $O(n+\frac{n\sigma}{w})$ plads og $O(n+\frac{n\sigma}{w}+m \log N \log w \cdot occ)$ eller $O(n+\frac{n\sigma}{w} \log w+ m \log N \cdot occ)$ tid. Her er $w$ ordstørrelsen og $occ$ er antallet af gange delsekvensen forekommer. Algoritmen bruger mindre plads end tidligere algoritmer og er hurtigere end den hidtil hurtigste når $occ = o(\frac{n}{\log N})$. Algoritmen er baseret på en ny datastruktur som effektivt kan svare på en forespørgsel om på hvilken position et bestemt tegn forekommer næste gang i en komprimeret streng. Denne datastruktur anvender en ny datastruktur til en variant af træfarveproblemet hvor knudernes farver er pakket i bitstrenge.

**Random access i komprimerede strenge.** Random access problemet i komprimerede strenge handler om at konstruere en datastruktur som understøtter tilgang til tegnet der står på position $i$ af den komprimerede streng. Vi præsenterer en datastruktur som bruger $O(n\Delta \log_\Delta \frac{N}{n} \log N)$ bits plads og understøtter tilgang til position $i$ i $O(\log_\Delta N)$ tid, hvor $\Delta \leq \log^{O(1)} N$. Dette er optimalt for strenge der komprimerer polynomielt godt, altså når $n = O(N^{1-\varepsilon})$.

**Bogmærker i komprimerede strenge.** Vi ønsker at gemme en SLP og et sæt af positioner (bogmærker) $\{i_1, \ldots, i_b\}$ således at en delstreng af længde $l$ startende i en af disse positioner kan decomprimeres i $O(l)$ tid. Vores datastruktur skal bruge følgende plads

$$O\left(\min_{1 \leq \tau \leq \log^* N} \left\{\tau(n+b) + \min\{n, b\} \log^{(\tau)} N\right\}\right).$$

# CONTENTS

# CHAPTER 1

## INTRODUCTION

Textual databases for e.g. biological or web-data are growing rapidly, and it is often only feasible to store the data in compressed form. However, compressing the data comes at a price. Traditional algorithms for e.g. pattern matching requires all data to be decompressed – a computationally demanding task. In this thesis we design data structures for accessing and searching compressed data efficiently. The goal is to augment a compressed file to enable answering queries about the data by adding as little data as possible.

We consider some fundamental problems in string pattern matching and some new problems arising from wanting to store the data in compressed form while supporting fast access to certain parts of the data. We focus on grammar-compressed strings because this provides a very general model that captures many other compression schemes.

The work in this thesis appears in the following papers.

- **Compact q-Gram Profiling of Compressed Strings**
  *Philip Bille, Patrick Hagge Cording, and Inge Li Gørtz.*
  In Theoretical Computer Science, volume 550, 2014. An extended abstract appeared in Proceedings of the 24th Symposium on Combinatorial Pattern Matching, 2013.

- **Fingerprints in Compressed Strings**
  *Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind.*
  In Proceedings of the 13th Algorithms and Data Structures Symposium, 2013.

- **Compressed Subsequence Matching and Packed Tree Coloring**
  *Philip Bille, Patrick Hagge Cording, and Inge Li Gørtz.*
  In Proceedings of the 25th Symposium on Combinatorial Pattern Matching, 2014.

- **Optimal Time Random Access to Grammar-Compressed Strings in Small Space**
  *Patrick Hagge Cording.*
  Unpublished. Will be merged with Belazzougui et al. [15] and submitted for publication in 2015.

- **Bookmarks in Grammar-Compressed Strings**
  *Patrick Hagge Cording, Pawel Gawrychowski, and Oren Weimann.*
  Unpublished.

The rest of this chapter will largely cover the basic prerequites for the results presented in this thesis. In Section 1.5 we introduce the problems studied along with an outline of the solutions. Chapter 2 through to 6 present our solutions independently and in a form that is unaltered from the way they appear when originally published.

## 1.1 Model of Computation

We base our work on the word RAM model of computation. The most meticulous description of this model is found in [56]. The word RAM is a unit-cost random-access machine with a word size of $w$ bits. We assume that we have access to inifinitely many memory cells that are not initialized in any way. Each memory cell holds an integer value in the range $0, \ldots, 2^w - 1$.

We adopt the common assumption that $w \geq \log N$ where $N$ is the input size (in this work, $N$ is the size of the uncompressed string), i.e., a word can hold the address to any element of the input.

The word RAM adopts basic operations from the RAM model such as load, store, comparison, and jump statements. In addition we assume that we can perform basic arithmetic operations on words such as addition, subtraction, multiplication, division, left and right shifts, and standard bitwise operations (AND, OR, XOR, and NOT). We also assume that the machine has access to an operation that can produce a random number in some specified range.

All of the operations mentioned above are performed in unit time. The running time of an algorithm is the number of unit-time operations it performs. Unless otherwise noted, the space usage of an algorithm is measured in the number of distinct $w$-bit memory cells the algorithm writes to.

## 1.2 Strings and Trees

The following is the general terminology for strings and trees used throughout the thesis.

A string $S$ is a sequence of $N$ characters drawn from an alphabet $\Sigma$ of size $\sigma$. $S[i, j]$ is the substring starting in position $i$ and ending in $j$, both positions inclusive. $S$ is indexed from 1 to $N$. We assume that the $i$-th character $S[i]$ can be accessed in $O(1)$ time, i.e., $w = \Omega(\log N)$ and that $\sigma = O(2^w)$.

Trees are rooted and ordered unless otherwise noted. For a node $v$, we refer to the edge adjacent to the parent of $v$ as the ingoing edge of $v$ and the edges going to $v$'s children as outgoing edges of $v$. For binary trees, we denote the children of a node $v$ by $\mathit{left}(v)$ and $\mathit{right}(v)$. Both nodes and edges may be labelled.

## 1.3 Compression Schemes

The work presented in this thesis is largely focused on a compression scheme named after its representation, namely, the Straight Line Program (SLP). The SLP is a widely studied compression scheme because of its simplicity and it is known to model many other compression schemes. We also describe the LZ77 [109] and LZ78 [110] schemes, named after its inventors Lempel and Ziv in the years 1977 and 1978.

These three compression schemes are closely related. For instance, an LZ77 compressed file can be converted to an SLP with logarithmic overhead and an LZ78 file can be converted with only constant overhead. The conversion can be done in time nearly linear in the compressed size of the files. Consequently, if we design a data structure for the SLP of size $f(n)$ we also have a data structure for LZ77 of size $O(f(n \log \frac{N}{n}))$ and a data structure for LZ78 of size $O(f(n))$. An overview of algorithms to convert between files from different compression schemes is given in [54].

We now give the details of SLPs, LZ77, and LZ78 along with terminology used throughout the thesis.

### *1.3.1 Straight Line Programs*

A Straight Line Program (SLP) is a context-free grammar in Chomsky normal form that generates one string only. A production rule either has two other rules on its righthand

side if it is non-terminal or a character from $\Sigma$ if it is terminal. Rules are unambiguous and non-recursive.

For convenience we assume that the SLP is represented as a directed acyclic graph (DAG). Nodes in the DAG are labeled with the names of production rules. Non-terminals have out-degree 2 and a terminal has one outgoing edge to a node labelled by one character. An example of an SLP is shown in Figure 1.1.

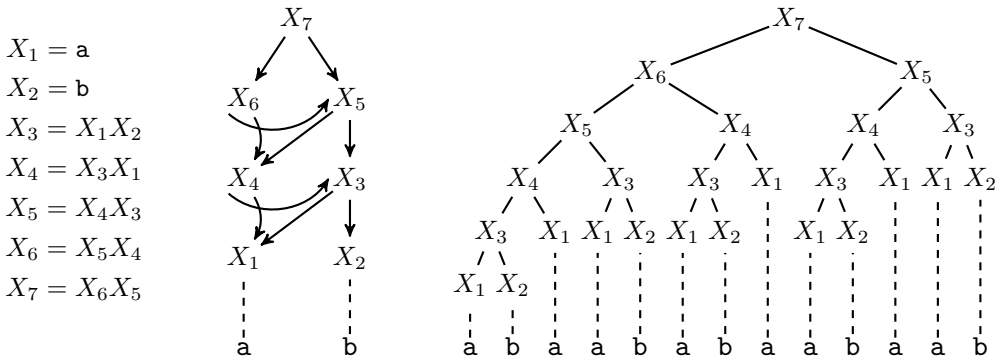



**Figure 1.1:** An SLP compressing the string `abaababaabaab`, its corresponding DAG, and its parse tree.

Let $\mathcal{S}$ denote an SLP with $n$ nodes compressing a string $S$ of size $N$. The size of the SLP will lie in the range $\log N \leq n \leq N$. To decompress a string we start a top-down left-to-right traversal starting from the node representing the start rule of the grammar. The tree emerging from this is the parse tree of the SLP and the leaves on the tree (read from left to right) is the uncompressed string. The height $h$ of $\mathcal{S}$ is the longest path from its root to a leaf in the parse tree of $\mathcal{S}$.

If we want to support decompression of substrings without having to decompress the entire string, then we need to add auxiliary data to the SLP. The simplest approach is to store the size of the string generated by each node. This requires $O(n)$ space and we may decompress an arbitrary substring of length $l$ in $O(h + l)$ time. Using a much more elaborate data structure, this can be improved to $O(\log N + l)$ time while retaining $O(n)$ space [21]. Using non-linear space, decompression can be done in $O(\log N / \log \log N + l)$ time [15]. In some cases, the "kick-off" time can be avoided. We may decompress the prefix or suffix of length $l$ of a string generated by some node in $O(l)$ time, provided that we have already found said node [50].

Finding the smallest SLP representing a string is NP-hard [26]. However, several approximation algorithms exist [26, 61, 62, 88] and efficient practical algorithms have been developed [73,85]. We do not assume that the SLP given as input for our algorithms is optimal.

### *1.3.2 LZ77*

The LZ77 factorization of a string $S$ of size $N$ is a sequence of factors $f_1, \ldots, f_z$ such that these generate $S$ when expanding them left to right. A factor has the form $f_k = (i, j, \alpha)$, where the string $S[i, j]$ is a substring of the string obtained from expanding $f_1, \ldots, f_{k-1}$ and $\alpha$ is a character. Some factors may expand to just $\alpha$ and will then have null values instead of $i$ and $j$. The LZ77 factorization defined here is not self-referential. In the self-referential version of LZ77, a factor $f_k$ is allowed to refer to a substring that is a concatenation of a suffix of $f_1, \ldots, f_{k-1}$ and a prefix of itself. Since we at most double the size of the string for every factor we add, the achievable compression for LZ77 is $\log N \leq z \leq N$. The following is an example of an LZ77 factorization of a string.

$$(-,-,\mathtt{a})(-,-,\mathtt{b})(1,1,\mathtt{a})(2,3,\mathtt{b})(3,6,\mathtt{a})(-,-,\mathtt{b})$$

**Figure 1.2:** The LZ77 factorization of the string abaababaabaab.

To decompress a string we process the factors from $f_1$ to $f_z$. When processing $f_k = (i, j, \alpha)$ we append $S[i, j]$ to the string that we have already decompressed from $f_1, \ldots, f_{k-1}$ and then append $\alpha$.

The greedy LZ77 parse maximizes the substring of $f_1, \ldots, f_{k-1}$ referred to from $f_k$, and is known to produce the optimal LZ77 factorization [28, 32, 92] when measuring the number of factors produced[1]. The optimal factorization may be found in $O(N)$ time given the suffix tree/array[2] of $S$.

Given the LZ77 factorization of $S$ we may build an SLP of size $O(z \log \frac{N}{z})$ for $S$ [88]. Hence, a data structure using $f(n)$ space for SLPs is also a data structure using $O(f(z \log \frac{N}{z}))$ space for an LZ77 compressed file of size $z$. If we want to convert an SLP to the LZ77 factorization, this can be done in $O(\mathrm{poly}(n))$ time [54].

Unlike the SLP, LZ77 is used widely in data compression software in practice (often combined with other methods and/or in derivative variants). For example, it is the basis of the gzip, png, rar, and zip file formats.

### *1.3.3 LZ78*

Much like LZ77, the LZ78 factorization is a sequence of factors $f_1, \ldots, f_z$ that generate $S$ when expanded left to right. However, in this scheme a factor has the form $f_k = (f_i, \alpha)$, where $1 \leq i < k$ and $\alpha$ is a character. As with LZ77, factors can have a null value instead of $f_i$ and in this case only expand to one character. With LZ78 compression we may achieve a compression in the range $\sqrt{N} \leq z \leq N$. The following shows the LZ78 factorization of a string.

$$(-,\mathtt{a})(-,\mathtt{b})(1,\mathtt{a})(2,\mathtt{a})(4,\mathtt{a})(5,\mathtt{b})$$

**Figure 1.3:** The LZ78 factorization of the string abaababaabaab.

LZ78 can be seen as a restricted version of LZ77 where the strings that are referred to from factors are restricted to start and end in substrings that correspond to the expansion of a single preceding factor. Contrary to LZ77, the LZ78 parse may be seen as a grammar, and it converts to an SLP with less overhead. For each unique character we create a terminal node with an edge to that character. For each factor $f_k = (f_i, \alpha)$ we create a node $v_k$ with $v_i$ as its left child and the terminal node representing $\alpha$ as its right child. Finally, we build a binary tree with the sequence $v_1, \ldots, v_z$ as leaves. The resulting SLP has size $O(z)$. Recent advances also show that we can convert a string compressed by an SLP into its LZ78 factorization in almost linear time [12, 13].

LZ78 is used in e.g. the gif file format among others.

## 1.4 Techniques

In this section we cover some of the core techniques that we rely on to design the algorithms and data structures in this thesis.

[1] The greedy parsing strategy is no longer optimal when measuring the output in number of bits required [39].

[2] It is assumed for this chapter that the reader is familiar with the suffix tree or array. Otherwise, see [36, 55, 80, 103, 106] or [67, 77].

### 1.4.1 Karp-Rabin Fingerprints

Karp-Rabin fingerprints [69] is a popular tool when designing string algorithms because they can be stored in little space and can be used to quickly compare strings with a small probability of error. A fingerprint function maps substrings of arbitrary length into integer hash values. The idea originates from a pattern matching algorithm by Karp and Rabin in 1987 and has since then been used in the design of many randomized algorithms (see e.g., [7, 10, 29–31, 37, 49, 64, 87]). They play a crucial role in the first two results of this thesis, described in Chapter 2 and Chapter 3.

Let $S$ be a string, $p$ a prime, and $b$ a randomly chosen number. Then the Karp-Rabin fingerprint $\phi(S)$ is

$$\phi(S) = \sum_{k=1}^{|S|} s[k] \cdot b^k \mod p$$

This type of fingerprint function has a range of properties that makes it ideal for use in string algorithms. In particular:

- For two strings $S_1$ and $S_2$, if $S_1 = S_2$ then $\phi(S_1) = \phi(S_2)$ with high probability. If $S_1 \neq S_2$ then $\phi(S_1) \neq \phi(S_2)$.
- $\phi$ is a rolling function. That is, given $\phi(S[i, j])$ we can compute $\phi(S[i + 1, j + 1])$ in constant time.
- Given $\phi(S_1)$ and $\phi(S_2)$, we can compute $\phi(S_1S_2)$ in constant time. Similar operations are also possible in constant time.

Further details on how to chose the parameters and how the above operations are implemented is given in the respective chapters where needed.

### 1.4.2 Decomposition of SLPs

A heavily used technique in the design of algorithms and data structures for trees is the decomposition of trees into paths. Perhaps most notable is the heavy-path decomposition [57, 90] that first appeared in the early 80s and since has been used in many results. Here we will describe the generalization of tree decompositions to SLPs.

A path decomposition of a tree will typically select one outgoing edge for each node based on some rule. A simple rule is to always select the leftmost outgoing edge. We could also select the path going to the child spanning the largest subtree among the children – the is known as the heavy-path decomposition. The selected edges will form a set of disjoint paths. When doing the same thing for SLPs, the selected edges will form a forest where the trees are rooted in the terminals of the SLP and grow upwards. Now the problem of traversing the SLP is reduced to visiting and solving some problem on a number of trees.

The left-path decomposition of SLPs was used implicitly in [50] to support backtracking on a path that is traversed by following a pointer from a node to its leftmost child. Heavy-path decomposition of SLPs was first used in [21] as a core technique in a data structure for random access in SLPs. We fix a heavy-path decomposition of an SLP as follows. For each node with children $u$ and $w$, select the edge to $u$ as heavy if $u$ generates a string that is longer than the string generated by $w$, and the edge to $w$ otherwise. On any path from the root of the SLP to a leaf we visit at most $\log N$ of the trees emerging from the heavy-path decomposition.

The heavy-path decomposition of SLPs will play a role in the data structures described in Chapter 3, 4, and 6.

### *1.4.3 Balancing SLPs*

Alike trees, SLPs can be balanced in many ways. We say that an SLP is balanced if its parse tree is balanced according to any definition that applies to trees. For example, an SLP is substring-balanced if for every node with children $v$ and $u$, the strings generated by $v$ and $u$ have the same length (within a constant factor). It corresponds to the parse tree being leaf-balanced. A balanced SLP typically provides a guarantee on its height which is an advantage in algorithms that employ top-down traversals of the SLP. When the SLP is balanced there are often very simply algorithms with bounds matching the best known bounds for some problem. Balancing of the SLP will play an important role in the result presented in Chapter 5.

Given an arbitrary, and possibly unbalanced SLP, the two most widely methods ([26, 88]) for balancing the SLP are actually approximation algorithms for the smallest grammar problem exerting the side effect that the resulting SLP is balanced. Common to the two algorithms is that they produce an SLP of size $O(z \log \frac{N}{z})$, where $z$ is the size of the LZ77 parse of the input string[3]. The number of nodes in the smallest SLP of a string is always greater than or equal to $z$, so if we use the approximation algorithms on a string that is already compressed to an SLP of size $n$, the resulting balanced SLP has size $O(n \log \frac{N}{n})$. The two algorithms differ in the balancing scheme that the resulting SLPs adhere to. One produces a substring-balanced SLP and the other a height-balanced SLP.

We do not assume that the SLPs given as input to our algorithms are balanced.

### *1.4.4 Pattern Matching in SLPs*

In string pattern matching the goal is to find and report all occurrences of a pattern string of size $m$ in a much larger string, often called the text. In compressed pattern matching the text, or both the text and the pattern, are compressed. When both the text and the pattern are compressed we call it fully-compressed pattern matching and when only the text is compressed it is semi-compressed pattern matching.

Two heavily studied variations of pattern matching are approximate pattern matching and the k-mismatch problem, where we allow up to $k$ errors in an occurrence of the pattern (see e.g. [8, 84] for details on these problems).

We will now describe the basic algorithm for semi-compressed pattern matching in an SLP. The method is interesting because it reduces the problem to pattern matching in uncompressed strings and therefore also works for approximate pattern matching and the k-mismatch problem if we plug in the known algorithms for these problems. With a few exceptions, better algorithms do not exist for these problems on SLPs.

The source of the algorithm is uncertain to us. It might originate from [70] albeit in a different form than presented here.

Consider a node $v$ with children $u$ and $w$ in an SLP $\mathcal{S}$. Let the relevant substring of $v$ be the suffix of length $m-1$ of the string generated by $u$ concatenated with the prefix of length $m-1$ of the string generated by $w$. It can be shown that the pattern occurs in the string generated by $\mathcal{S}$ if and only if it occurs in at least one of the relevant substrings[4]. Let $S(v)$ be the string generated by $v$. We may compute the prefixes and suffixes of length at most $m-1$ as follows. For each terminal $v$ generating the character $c$, let $\text{Pref}(v) = c$ and $\text{Suf}(v) = c$. For each internal node $v = uw$, the prefixes and suffixes are found as follows.

[3] In a recent survey on SLPs by Lohrey [76], it is claimed that the algorithm by Rytter [88] also can be used to produce a balanced SLP without approximating the smallest grammar and avoid producing the LZ77 factorization at that.

[4] Variations of this theorem is used in this thesis; see Lemma 2.3 on page 16 and Lemma 6.2 on page 55.

$$\text{Pref}(v) = \begin{cases} \text{Pref}(u) & \text{if } |u| \geq m-1 \\ S(u)\text{Pref}(w)[1, m-|u|-1] & \text{otherwise} \end{cases}$$

$$\text{Suf}(v) = \begin{cases} \text{Suf}(w) & \text{if } |w| \geq m-1 \\ \text{Suf}(u)[|w|+1, m]S(w) & \text{otherwise} \end{cases}$$

Computing tables with the prefixes and suffixes can be done in $O(nm)$ time and space and from these we may construct all relevant substrings. Then we run a linear time pattern matching algorithm, such as the one by Knuth, Morris and Pratt [71], and all occurrences of a pattern is found in $O(nm)$ time.

Alternatively, the suffix and prefix of $u$ and $w$ for some node $v = uw$ can be decompressed in $O(m)$ time using the $O(n)$-space data structure of Gasieniec et al. [50]. Pattern matching still takes $O(nm)$ time but the space is reduced to $O(n+m)$.

To report the occurrences we need to add some bookkeeping and visit the nodes in a certain order. The running time and space usage becomes $O(nm + \text{occ})$, where occ is the number of occurrences of the pattern.

To extend the method to approximate pattern matching or the k-mismatch problem we modify the algorithm to decompress the prefixes and suffixes of length $m-1+k$ and apply the known algorithms accordingly. For approximate pattern matching we get $O(nmk)$ time if we for instance plug in the algorithm of Landau and Vishkin [72]. For k-mismatch we get $O(nm\sqrt{k \log k})$ or $O(n(m+k^3)\log k))$ time by applying the algorithms by Amir et al. [8].

## 1.5 Our Contributions

In this section we will introduce the problems and results of this thesis.

### *1.5.1 Chapter 2: Compact q-Gram Profiling of Compressed Strings*

A *q-gram* is popular term in bioinformatics used to denote a substring of length $q$. The q-gram profile of a string is a mapping from every q-gram to the number of occurrences in the string. We show how to construct a data structure representing this mapping from an SLP. The data structure supports queries for the number of occurrences in $O(q)$ time (and also support listing all q-gram occurring once or more). Methods for computing the q-gram profile from an uncompressed string can be used if we apply the same reduction as in Section 1.4.4, but the challenge is to use less space than what this approach offers.

Goto et al. [52, 53] were the first to present algorithms for computing the q-gram profile of an SLP-compressed string. Their first algorithm uses $O(qn)$ time and space which was later improved to $O(N-\alpha)$ time and space. They show that $N-\alpha$ is an asymptotic improvement over $qn$. Specifically, it is the number of characters sufficient to decompress to capture every distinct q-gram in $S$. The expression depends on the amount of redundancy captured by the SLP. It seems inevitable that we have to visit every node once (and decompress some characters) to ensure that we have read all distinct q-grams, so the drawback of Goto et al.'s algorithms is that the space usage also depends on the quality of the compression.

We present a randomized algorithm for computing the q-gram profile. It runs in $O(N-\alpha)$ time and uses $O(n+q+k_{S,q})$ space, where $k_{S,q}$ is the number of distinct q-grams in $S$. This improves the space usage over the algorithms by Goto et al. In our solution, a specific q-gram does not appear more than once in the q-gram profile, even if it occurs many times in $S$ and this redundancy not is reflected by the SLP.

The algorithm works by first computing a subgraph of the De Bruijn graph containing every distinct q-gram of $S$ exactly once. This graph is then transformed to a set of strings

represented by a trie and we compute the suffix tree of the set of strings. Augmented with a little extra data, the suffix tree serves as the q-gram profile.

To efficiently compute the De Bruijn graph we use Karp-Rabin fingerprints. Consequently, our algorithm has a positive probability of computing an incorrect q-gram profile. We show how to transform the algorithm from a Monte-Carlo to a Las-Vegas type randomized algorithm by verifying that the chosen fingerprint function is collision free for all q-grams in $O(N - \alpha)$ time. We are to the best of our knowledge the first to show how to do this using the suffix tree. To summarize, our algorithm employs fresh uses of the old and well-known concepts of the De Bruijn graph, the suffix tree and Karp-Rabin fingerprints to build the q-gram profile in optimal space.

**Related work**

Our work demonstrates how to utilize the SLP to extract relevant information from the string without having to decompress more than the neccessary characters. Mining characteristics of strings is useful in many settings. Related to this, it has been shown how to compute all palindromes [79], and how to find the longest repeating substrings and the most frequent substring [59] in SLPs. The closely related problem of finding the frequencies of non-overlapping q-grams in an SLP-compressed string has also been studied [51].

**Further work and open problems**

We have shown how to verify that a fingerprint function is collision-free for all substrings of length $q$. In the next section we need to verify that it is collision-free for $q = 2^i$ for $i = 1, \ldots, \log N$. Verifying that a function is collision-free seems to be an important element in the design of randomized algorithms based on fingerprints, and remaining question is how to do this fast in compressed space. We will resume this discussion in the next section.

### *1.5.2 Chapter 3: Fingerprints in Compressed Strings*

In this work we consider two problems. The first is to construct a data structure that efficiently supports finding the Karp-Rabin fingerprint of any substring of a grammar-compressed string. The second is to construct a data structure for the longest common extension (LCE) problem on a compressed string. The LCE of two indices in $S$ is the length of the longest substring starting in these two indices. Or formally, given $i$ and $j$, the answer to a query is the length $\ell$ of the maximum substring such that $S[i, i + \ell] = S[j, j + \ell]$.

To tackle the latter problem we take the approach of [91] where the LCE is computed by doing an exponential search on $\ell$ using Karp-Rabin fingerprints to compare substrings. To implement this, we develop a data structure for finding fingerprints in SLPs.

The LCE problem is fundamental in string matching, and efficient theoretical and practical solutions have been developed for uncompressed strings[19, 91]. We are the first to present a dedicated solution for grammar-compressed strings. Other solutions depend on data structures for fast decompression of substrings and require $\Omega(\ell)$ time to compute the LCE. Our data structure supports finding the Karp-Rabin fingerprint of any substring in $O(\log N)$ time. Using this data structure, we get a data structure that anwers LCE queries in $O(\log N \log \ell)$ time.

Since the data structure uses Karp-Rabin fingerprints there is a probability that the answer is incorrect. However, we show how to verify that the selected fingerprint function is collision-free for all the fingerprints that we may possibly use in our algorithm. We may do this in $O(N \log N)$ time and $O(N)$ space or alternatively $O(N^2/n \log N)$ time and $O(n)$ space.

In addition, we also show how SLPs model LZ78 and how to efficiently utilize the structure of the resulting SLP to obtain faster algorithms for finding fingerprints and the LCE problem.

The fingerprint data structure is an augmentation of the random access data structure due to Bille et al. [21]. In addition to the data we need to store to facilitate random access queries, we also store fingerprints that we can stitch together as we do a top-down traversal.

**Further work and open problems**

The query time and space bounds of our fingerprint data structure matches the bounds of the random access data structure of Bille et al. [21]. It seems unlikely that computing the fingerprint of an arbitrary substring should be faster than accessing some character of $S$. We refer to Section 1.5.4 where the random access problem is discussed.

With the addition of our fingerprint data structure, we now know how to support finding fingerprints and random access queries in $O(\log N)$ time and $O(n)$ space. For uncompressed strings, both of these primitives are supported $O(1)$ time and linear space. It is also known how to perform LCE queries in uncompressed strings in $O(1)$ time using $O(N)$ space, while in SLPs it requires $O(\log^2 N)$ time. Perhaps this indicates that there is a faster algorithm for the LCE problem on compressed strings.

We saw in this and the previous section that the Karp-Rabin fingerprint is a powerful tool for designing randomized algorithms. The caveat is that in order to make the algorithm deterministic, we need to verify that the function is collision-free for some set of substrings of $S$. In the previous section we saw how to efficiently verify the function in compressed space, but in the present work it added an undesireable $O(N^2/n \log N)$ term to the construction time of the data structure. We leave it as an interesting open question how to verify that fingerprint functions are collision-free in compressed space and in general.

### *1.5.3 Chapter 4: Compressed Subsequence Matching and Packed Tree Coloring*

The subsequence matching problem is to find and report all the minimal substrings of a string that contains a given pattern as a subsequence. A match is minimal when it is no longer a match if the substring containing it is shortened.

There is no upper limit on how big the window that contains a minimal occurrence may be, so the reduction given in Section 1.4.4 does not apply. The size of the relevant substrings simply cannot be bounded.

Previous results for subsequence matching in SLPs are summarized in Table 4.1 on page 38. They are based on dynamic programming and the fastest of these, due to Yamamoto et al. [108], runs in $O(nm + \mathrm{occ})$ time and space. The pattern of dependencies in the dynamic program excludes the application of standard techniques to speed up the algorithm or reduce the space usage.

Our algorithm for compressed subsequence matching uses $O(n + \frac{n\sigma}{w})$ space and runs in

- $O(n + \frac{n\sigma}{w} + m \log N \log w \cdot \mathrm{occ})$ time, or
- $O(n + \frac{n\sigma}{w} \log w + m \log N \cdot \mathrm{occ})$ time.

With a straightforward trick we may limit the alphabet size to $\sigma = m$ and are therefore guaranteed that we asymptotically never use more space than the previously best known algorithm. For $occ = o(\frac{n}{\log N})$ our algorithm also runs faster than the $O(nm)$-time algorithm.

To overcome the challenges posed by the known methods we have based our algorithm on the classic algorithm for subsequence matching in uncompressed strings, where the

text is scanned left to right for occurrences of the pattern. Instead of scanning the full uncompressed string we develop a data structure for SLPs that allow fast queries for the next occurrence of a specific character. This data structure is able to answer queries of the form "what is the index of the first occurrence of character $c$ after position $i$?" It is based on a heavy-path decomposition of the SLP that links the problem to finding the first colored ancestor (FCA) in trees.

In the FCA problem the nodes of a tree has zero or more colors and the goal is to construct a data structure that may answer queries of the form "what is the deepest node with color $c$ that is an ancestor of node $v$?".

Muthukrishnan and Müller [83] invented a data structure for this problem that uses $O(t + D)$ space and has query time $O(\log w)$, where $t$ is the size of the tree and $D$ is the accumulated number of colors given to nodes. Using this in our setting would yield poor space bounds since the best bound on the number of colors is $D = O(t\sigma)$ which sums to $O(n\sigma)$ when applied to all the trees in the heavy-path decomposition.

In this paper we present a FCA data structure that places itself between the state-of-the-art solution and the trivial solution in terms of space usage. It is constructed in a way that allows data to be packed into machine words, and thus use only $O(\frac{t\sigma}{w})$ space. The query time is either $O(1)$ or $O(\log w)$ depending on whether we want to spend $O(\frac{t\sigma}{w} \log w)$ or $O(\frac{t\sigma}{w})$ time on preprocessing, respectively.

**Related work**

Subsequence matching is just one of many variations of string pattern matching that has attracted attention. Exact semi-compressed pattern matching in LZ77 and LZ78 was first studied by Farach and Thorup [37] and Amir et al. [6], and later improved by Gawrychowski [46, 48]. In SLPs, the problem can be solved in $O(nm)$ time as described in Section 1.4.4. Moreover, pattern matching is known to reduce to convolution; see e.g. [55]. Tanaka et al. [96] show how to compute the convolution of a compressed string and an uncompressed pattern in time $O(nm \log m)$.

Semi-compressed approximate pattern matching has also been studied. Kärkkäinen et al. [66] and Bille et al. [18] give algorithms for approximate pattern matching in LZ78 compressed strings with a running time of $O(zmk + \mathrm{occ})$. By plugging in Landau and Vishkins algorithm [72] to the reduction in Section 1.4.4 we get an algorithm with running time $O(nmk)$ for SLPs. Another approach is taken by Tiskin [100] who presents an $O(nm \log m)$ time algorithm for approximate pattern matching.

Fully-compressed pattern matching seems related to the previously mentioned problems, but the solutions employ a very different fan of techniques; see e.g. [44, 49, 58, 60, 70, 75, 82].

Claude and Navarro [27] and Gagie et al. [41] studied the problem of indexing a string compressed by an SLP. The latter uses nearly linear space and supports finding all occurrences of a pattern in $O(m^2 + \mathrm{occ} \cdot \log \log n)$ time.

**Further work and open problems**

At first it might seem that our algorithm can be modified to also work for approximate pattern matching. However, it has been based specifically on the premise that all characters of the pattern must occur in a match, which is rather limiting.

A glaringly open problem is to determine if the $\log w$-factor that is shifted from query time to preprocessing time in the FCA data structure can be removed. Preliminiary studies indicate that it might be possible to trade it with a $\log^* w$-factor on both time bounds [45].

Moreover, it is not unreasonable to believe that the $\log N$ of the query time can be reduced. Our algorithms repeatedly queries the SLP about the first character $c$ after

position $i$ in $S$, and this is clearly at least as hard as random access in the compressed string (see the next section for a discussion on this). However, since it is in fact multiple successive queries, we believe that a more clever solution exists.

### *1.5.4 Chapter 5: Optimal Time Random Access to Grammar-Compressed Strings in Small Space*

Accessing a specific position in a string is a fundamental primitive in algorithm design, and we assume that this takes $O(1)$ time in the RAM model when the string is uncompressed. In a grammar-compressed string it is more complicated. We develop a data structure that can answer the question "which character is in position $i$ of the compressed string?", also known as random access in compressed strings.

Bille et al. [21] present a data structure using $O(n)$ space that supports random access in $O(\log N)$ time. Until this, random access was done by storing the lengths of the strings generated by each node and use these for a top-down traversal in $O(h)$ time. Because of the the clean bounds given by Bille et al. (and possibly also due to the high complexity of their data structure), it was widely believed that this was optimal until Verbin and Yu [105] proved lower bounds saying otherwise. Verbin and Yu show two results. First, if we allow $O(\text{poly}(n))$ space, then $\Omega(n^{1/2-\varepsilon})$ time is a lower bound for random access in a grammar-compressed string with any compression ratio. Second, if the size of the string in question is polynomial in the size of the grammar compressing it, i.e., when $n = O(N^{1-\varepsilon})$, then $\Omega(\log n/\log\log n)$ time is a lower bound for random access using $O(n \cdot \text{polylog}(N))$ space. This discovery inspired the search for a better random access data structure.

We describe a data structure using $O(n\tau \log_\tau \frac{N}{n} \log N)$ bits of space that supports random access in $O(\log_\tau N)$. If we set $\tau = \log^{O(1)} N$, the query time is optimal within a constant factor for polynomially compressible strings as shown by Verbin and Yu.

Our data structure is constructed in two steps. First we balance the input grammar and then we increase the degree of each node. A nearly identical data structure has been presented independently by Belazzougui et al. [15]. We improve the space usage of their data structure by a $\log \tau$ factor. This improvement comes from employing a specific algorithm to balance the SLP and using this in the analysis of the space usage.

#### Further work and open problems

For SLPs with arbitrary compression ratio there still is a gap between the lower bounds and the upper bound achieved by our data structure. Specifically, we highlight the following interesting open questions:

- Is there a data structure with optimal $O(n^{1/2-\varepsilon})$ query time?
- Is there a linear space data structure with optimal query time $O(\log N/\log\log N)$ for polynomially compressible strings?
- Can we construct a data structure with a time-space product of $n \log N$? That is, a data structure that offers the same space and time bounds as Bille et al. [21] as one extreme and optimal query time, i.e., $O(\log N/\log\log N)$ time and $O(n \log\log N)$ words of space, as the other.
- Are there different lower bounds if we are limited to $O(n)$ space?

As mentioned, our data structure relies on having to balance the input SLP as a first step. It also remains an open question whether the SLP can be balanced (for any suitable definition of balancing) while only adding $o(\log \frac{N}{n})$ nodes.

### *1.5.5 Chapter 6: Bookmarks in Grammar-Compressed Strings*

A bookmark in a compressed string is a position from which any substring of length $l$ starting in that position can be decompressed in $O(l)$ time. For the bookmarking problem, we are given an SLP and a set of $b$ bookmarks (positions in the string) and the goal is to build a data structure that supports decompression of substrings starting in a bookmark in linear time.

Using existing techniques, it requires some time to iniate the decompression of an arbitrary substring. If we use the data structure by Gasieniec et al. [50] the initiation time is $O(h)$. The random access data structures by Bille et al. [21] or Belazzougui et al. [15] also support $O(l)$-time substring decompression after a $O(\log N)$- or $O(\log N/\log\log N)$-time random access query, respectively. For short substrings the "kick-off" time is dominant.

Gagie et al. [42] present a bookmarking data structure that uses $O(n + b\log^* N)$ space for balanced SLPs. If the SLP is not balanced the space usage of their data structure becomes $O(n\log\frac{N}{n} + b\log^* N)$, cf. Section 1.4.3.

We present a bookmarking data structure that uses space

$$O\left(\min_{1\leq\tau\leq\log^* N}\left\{\tau(n+b) + \min\{n,b\}\log^{(\tau)} N\right\}\right).$$

The optimal value for $\tau$ depends on the relationship between the size of the SLP and the number of bookmarks. If $b = \Theta(n)$ then the two terms balance when $\tau = \log^* N$ and the space becomes $O(n\log^* N)$. For other values of $b$, i.e., when $b \leq \frac{n}{\log^{(c)} N}$ or $n\log^{(c)} N \leq b$, for some constant $c$, the terms balance for $\tau = c$.

Our data structure is based on the observation that since we can decompress an arbitrary substring of length $l$ in $O(\log N + l)$ time, we only have to facilitate decompression of substrings of length less than $\log N$ from the bookmarks. It consists of $\tau$ copies of the SLP. The $i$-th copy is restructured such that for any substring of length at most $\log^{(i)} N$ there exists two nodes that generate strings whose concatenation contains the substring. Depending on the size of the substring we want to decompress, we start the decompression from a certain bookmark from one of these $\tau$ nodes.

#### Further work and open problems

The space bound of our data structure may be stated as $O((n+b)\log^* N)$. It is relevant to investigate if the $\log^* N$-factor is in fact necessary.

# CHAPTER 2

# COMPACT Q-GRAM PROFILING OF COMPRESSED STRINGS

Philip Bille Patrick Hagge Cording Inge Li Gørtz

Technical University of Denmark, DTU Compute

**Abstract**

We consider the problem of computing the q-gram profile of a string $S$ of size $N$ compressed by a context-free grammar with $n$ production rules. We present an algorithm that runs in $O(N-\alpha)$ expected time and uses $O(n+q+k_{S,q})$ space, where $N-\alpha \leq qn$ is the exact number of characters decompressed by the algorithm and $k_{S,q} \leq N-\alpha$ is the number of distinct q-grams in $S$. This simultaneously matches the current best known time bound and improves the best known space bound. Our space bound is asymptotically optimal in the sense that any algorithm storing the grammar and the q-gram profile must use $\Omega(n+q+k_{S,q})$ space. To achieve this we introduce the q-gram graph that space-efficiently captures the structure of a string with respect to its q-grams, and show how to construct it from a grammar.

## 2.1 Introduction

Given a string $S$, the q-gram profile of $S$ is a data structure that can answer substring frequency queries for substrings of length $q$ (q-grams) in $O(q)$ time. We study the problem of computing the q-gram profile from a string $S$ of size $N$ compressed by a context-free grammar with $n$ production rules. We assume that the model of computation is the standard $w$-bit word RAM where each word is capable of storing a character of $T$, i.e., the characters of $T$ are drawn from an alphabet $\{1,\ldots,2^w\}$, and hence $w \geq \log N$ [56]. The space complexities are measured by the number of words used.

The generalization of string algorithms to grammar-based compressed text is currently an active area of research. Grammar-based compression is studied because it offers a simple and strict setting and is capable of modelling many commonly used compression schemes, such as those in the Lempel-Ziv family [109,110], with little expansion [26,88]. The problem of computing the q-gram profile has its applications in bioinformatics, data mining, and machine learning [43,74,86]. All are fields where handling large amount of data effectively is crucial. Also, the q-gram distance can be computed from the q-gram profiles of two strings and used for filtering in string matching [23,63,93–95,102].

Recently the first dedicated solution to computing the q-gram profile from a grammar-based compressed string was proposed by Goto et al. [53]. Their algorithm runs in $O(qn)$

expected time[1] and uses $O(qn)$ space. This was later improved by the same authors [52] to an algorithm that takes $O(N - \alpha)$ expected time and uses $O(N - \alpha)$ space, where $N$ is the size of the uncompressed string, and $\alpha$ is a parameter depending on how well $S$ is compressed with respect to its q-grams. $N - \alpha \leq \min(qn, N)$ is in fact the exact number of characters decompressed by the algorithm in order to compute the q-gram profile, meaning that the latter algorithm excels in avoiding decompressing the same character more than once.

We present a Las Vegas-type randomized algorithm that gives Theorem 1.

**Theorem 2.1** *Let $S$ be a string of size $N$ compressed by a grammar of size $n$. The q-gram profile can be computed in $O(N - \alpha)$ expected time and $O(n + q + k_{S,q})$ space, where $k_{S,q} \leq N - \alpha$ is the number of distinct q-grams in $S$.*

Hence, our algorithm simultaneously matches the current best known time bound and improves the best known space bound. Our space bound is asymptotically optimal in the sense that any algorithm storing the grammar and the q-gram profile must use $\Omega(n + q + k_{S,q})$ space.

A straightforward approach to computing the q-gram profile is to first decompress the string and then use an algorithm for computing the profile from a string. For instance, we could construct a compact trie of the q-grams using an algorithm similar to a suffix tree construction algorithm as mentioned in [68], or use Rabin-Karp fingerprints to obtain a randomized algorithm [102]. However, both approaches are impractical because the time and space usage associated with a complete decompression of $S$ is linear in its size $N = O(2^n)$. To achieve our bounds we introduce the q-gram graph, a data structure that space efficiently captures the structure of a string in terms of its q-grams, and show how to compute the graph from a grammar. We then transform the graph to a suffix tree containing the q-grams of $S$. Because our algorithm uses randomization to construct the q-gram graph, the answer to a query may be incorrect. However, as a final step of our algorithm, we show how to use the suffix tree to verify that the fingerprint function is collision free and thereby obtain Theorem 2.1.

## 2.2 Preliminaries and Notation

### *2.2.1 Strings and Suffix Trees*

Let $S$ be a string of length $|T|$ consisting of characters from the alphabet $\Sigma$. We use $S[i, j]$, $0 \leq i \leq j < |S|$, to denote the substring starting in position $i$ of $S$ and ending in position $j$ of $S$. We define $socc(s, S)$ to be the number of occurrences of the string $s$ in $S$.

The suffix tree of $S$ is a compact trie containing all suffixes of $S$. That is, it is a trie containing the strings $S[i, |T| - 1]$ for $i = 0..|T| - 1$. The suffix tree of $S$ can be constructed in $O(|T|)$ time and uses $O(|T|)$ space [36]. The generalized suffix tree is the suffix tree for a set of strings. It can be constructed using time and space linear in the sum of the lengths of the strings in the set. The set of strings may be compactly represented as a common suffix tree (CS-tree). The CS-tree has the characters of the strings on its edges, and the strings start in the leaves and end in the root. If two strings have some suffix in common, the suffixes are merged to one path. In other words, the CS-tree is a trie of the reversed strings, and is not to be confused with the suffix tree. For CS-trees, the following is known.

**Lemma 2.1 (Shibuya [89])** *Given a set of strings represented by a CS-tree of size $n$ and comprised of characters from an alphabet of size $O(n^c)$, where $c$ is a constant, the*

[1]The bound in [53] is stated as worst-case since they assume alphabets of size $O(N^c)$ for fast suffix sorting, where $c$ is a constant. We make no such assumptions and without it hashing can be used to obtain the same bound in expectation.

*generalized suffix tree of the set of strings can be constructed in $O(n)$ time using $O(n)$ space.*

For a node $v$ in a suffix tree, the string depth $sd(v)$ is the sum of the lengths of the labels on the edges from the root to $v$. We use $parent(v)$ to get the parent of $v$, and $nca(v, u)$ is the nearest common ancestor of the nodes $v$ and $u$.

### 2.2.2 Straight Line Programs

A Straight Line Program (SLP) is a context-free grammar in Chomsky normal form that derives a single string $S$ of length $N$ over the alphabet $\Sigma$. In other words, an SLP $\mathcal{S}$ is a set of $n$ production rules of the form $X_i = X_l X_r$ or $X_i = a$, where $a$ is a character from the alphabet $\Sigma$, and each rule is reachable from the start symbol $X_n$. Our algorithm assumes without loss of generality that the compressed string given as input is compressed by an SLP.

It is convenient to view an SLP as a directed acyclic graph (DAG) in which each node represents a production rule. Consequently, nodes in the DAG have exactly two outgoing edges. An example of an SLP is seen in Figure 2.2(a). When a string is decompressed we get a derivation tree which corresponds to the depth-first traversal of the DAG.

We denote by $t_{X_i}$ the string derived from production rule $X_i$, so $S = t_{X_n}$. For convenience we say that $|X_i|$ is the length of the string derived from $X_i$, and these values can be computed in linear time in a bottom-up fashion using the following recursion. For each $X_i = X_l X_r$ in $\mathcal{S}$,

$$|X_i| = \begin{cases} |X_l| + |X_r| & \text{if } X_i \text{ is a nonterminal,} \\ 1 & \text{otherwise.} \end{cases}$$

Finally, we denote by $occ(X_i)$ the number of times the production rule $X_i$ occurs in the derivation tree. We can compute the occurrences using the following linear time and space algorithm due to Goto et al. [53]. Set $occ(X_n) = 1$ and $occ(X_i) = 0$ for $i = 1..n-1$. For each production rule of the form $X_i = X_l X_r$, in decreasing order of $i$, we set $occ(X_l) = occ(X_l) + occ(X_i)$ and similarly for $occ(X_r)$.

### 2.2.3 Fingerprints

A Rabin-Karp fingerprint function $\phi$ takes a string as input and produces a value small enough to let us determine with high probability whether two strings match in constant time. Let $s$ be a substring of $S$, $c$ be some constant, $2N^{c+4} < p \leq 4N^{c+4}$ be a prime, and choose $b \in \mathbb{Z}_p$ uniformly at random. Then,

$$\phi(s) = \sum_{k=1}^{|s|} s[k] \cdot b^k \mod p.$$

**Lemma 2.2 (Rabin and Karp [69])** *Let $\phi$ be defined as above. Then, for all $0 \leq i, j \leq |S| - q$,*

$$\phi(S[i, i+q]) = \phi(S[j, j+q]) \quad \textit{iff} \quad S[i, i+q] = T[j, j+q] \quad \textit{w.h.p.}$$

We denote the case when $S[i, i+q] \neq S[j, j+q]$ and $\phi(T[i, i+q]) = \phi(T[j, j+q])$ for some $i$ and $j$ a collision, and say that $\phi$ is collision free on substrings of length $q$ in $S$ if $\phi(S[i, i+q]) = \phi(S[j, j+q])$ iff $S[i, i+q] = S[j, j+q]$ for all $i$ and $j$, $0 \leq i, j < |S| - q$.

Besides Lemma 2.2, fingerprints exhibit the useful property that once we have computed $\phi(S[i, i+q])$ we can compute the fingerprint $\phi(S[i+1, i+q+1])$ in constant time using the update function,

$$\phi(S[i+1, i+q+1]) = \phi(S[i, i+q])/b - S[i] + S[i+q+1] \cdot b^q \mod p.$$

## 2.3 Key Concepts

### *2.3.1 Relevant Substrings*

Consider a production rule $X_i = X_l X_r$ that derives the string $t_{X_i} = t_{X_l} t_{X_r}$. Assume that we have counted the number of occurrences of q-grams in $t_{X_l}$ and $t_{X_r}$ separately. Then the relevant substring $r_{X_i}$ is the smallest substring of $t_{X_i}$ that is necessary and sufficient to process in order to detect and count q-grams that have not already been counted. In other words, $r_{X_i}$ is the substring that contains q-grams that start in $t_{X_l}$ and end in $t_{X_r}$ as shown in Figure 2.1. Formally, for a production rule $X_i = X_l X_r$, the relevant substring is $r_{X_i} = t_{X_i}[\max(0, |X_l| - q + 1) : \min(|X_l| + q - 2, |X_i| - 1)]$. We want the relevant substrings to contain at least one q-gram, so we say that a production rule $X_i$ only has a relevant substring if $|X_i| \geq q$. The size of a relevant substring is $q \leq |r_{X_i}| \leq 2(q-1)$.

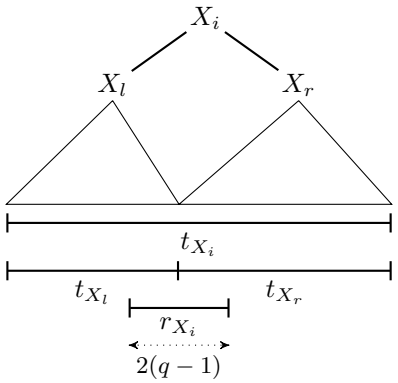



**Figure 2.1:** The derivation tree for $X_i = X_l X_r$ and the relevant susbtring $r_{X_i}$ of $X_i$.

The concept of relevant substrings is the backbone of our algorithm because of the following. If $X_i$ occurs $occ(X_i)$ times in the derivation tree for $\mathcal{S}$, then the substring $t_{X_i}$ occurs at least $occ(X_i)$ times in $S$. It follows that if a q-gram $s$ occurs $socc(s, t_{X_i})$ times in some substring $t_{X_i}$ then we know that it occurs at least $socc(s, t_{X_i}) \cdot occ(X_i)$ times in $S$. Using our description of relevant substrings we can rewrite the latter statement to $socc(s, t_{X_i}) \cdot occ(X_i) = socc(s, t_{X_l}) \cdot occ(X_l) + socc(s, t_{X_r}) \cdot occ(X_r) + socc(s, r_{X_i}) \cdot occ(X_i)$ for the production rule $X_i = X_l X_r$. By applying this recursively to the root $X_n$ of the SLP we get the following lemma.

**Lemma 2.3 (Goto et al. [52])** *Let $\mathcal{S}_q = \{X_i \mid X_i \in \mathcal{S}$ and $|X_i| \geq q\}$ be the set of production rules that have a relevant substring, and let $s$ be some q-gram. Then,*

$$socc(s, T) = \sum_{X_i \in \mathcal{S}_q} socc(s, r_{X_i}) \cdot occ(X_i).$$

### *2.3.2 Prefix and Suffix Decompression*

The following Lemma states a result that is crucial to the algorithm presented in this paper.

**Lemma 2.4 (Gąsieniec et al. [50])** *An SLP $\mathcal{S}$ of size $n$ can be preprocessed in $O(n)$ time using $O(n)$ extra space such that, given a pointer to a variable $X_i$ in $\mathcal{S}$, the prefix and suffix of $t_{X_i}$ of length $j$ can be decompressed in $O(j)$ time.*

Gąsieniec et al. give a data structure that supports linear time decompression of prefixes, but it is easy to extend the result to also hold for suffixes. Let $s$ be some string and $s^R$ the reversed string. If we reverse the prefix of length $j$ of $s^R$ this corresponds to the suffix of length $j$ of $s$. To obtain an SLP for the reversed string we swap the two variables on the right-hand side of each nonterminal production rule. The reversed SLP $\mathcal{S}'$ contains $n$ production rules and the transformation ensures that $t_{X_{i'}} = t^R_{X_i}$ for each production rule $X_{i'}$ in $\mathcal{S}'$. A proof of this can be found in [79]. Producing the reversed SLP takes linear time and in the process we create pointers from each variable to its corresponding variable in the reversed SLP. After both SLP's are preprocessed for linear time prefix decompression, a query for the length-$j$ suffix of $t_{X_i}$ is handled by following the pointer from $X_i$ to its counterpart in the reversed SLP, decompressing the prefix of length $j$ of this, and reversing the prefix.

### *2.3.3 The q-gram Graph*

We now describe a data structure that we call the q-gram graph. It too will play an important role in our algorithm. The q-gram graph $G_q(S)$ captures the structure of a string $S$ in terms of its q-grams. In fact, it is a subgraph of the De Bruijn graph over $\Sigma^q$ with a few augmentations to give it some useful properties. We will show that its size is linear in the number of distinct q-grams in $S$, and we give a randomized algorithm to construct the graph in linear time in $N$.

A node in the graph represents a distinct $(q-1)$-gram, and the label on the node is the fingerprint of the respective $(q-1)$-gram. The graph has a special node that represents the first $(q-1)$-gram of $S$ and which we will denote the start node. Let $x$ and $y$ be characters and $\alpha$ a string such that $|\alpha| = q-2$. There is an edge between two nodes with labels $\phi(x\alpha)$ and $\phi(\alpha y)$ if $x\alpha y$ is a substring of $S$. The graph may contain self-loops. Each edge has a label and a counter. The label of the edge $\{\phi(x\alpha), \phi(\alpha y)\}$ is $y$, and its counter indicates the number of times the substring $x\alpha y$ occurs in $S$. Since $|x\alpha y| = q$ this data structure contains information about the frequencies of q-grams in $S$.

**Lemma 2.5** *The q-gram graph of $S$, $G_q(S)$, has $O(k_{S,q})$ nodes and $O(k_{S,q})$ edges.*

**Proof** Each node represents a distinct $(q-1)$-gram and its outgoing edges have unique labels. The combination of a node and an outgoing edge thus represents a distinct q-gram, and therefore there can be at most $k_{S,q}$ edges in the graph. For every node with label $\phi(S[i, i+q-1])$, $i = 1..|S|-q-1$, the graph contains a node with label $\phi(S[i+1, i+q])$ with an edge between the two. The graph is therefore connected and has at most has at most $k_{S,q}+1$ nodes.

The graph can be constructed using the following online algorithm which takes a string $S$, an integer $q \geq 2$, and a fingerprint function $\phi$ as input. Let the start node of the graph have the fingerprint $\phi(S[0, (q-1)-1])$. Assume that we have built the graph $G_q(S[0, k+(q-1)-1])$ and that we keep its nodes and edges in two dictionaries implemented using hashing. We then compute the fingerprint $\phi(S[k+1, k+(q-1)])$ for the $(q-1)$-gram starting in position $k+1$ in $S$. Recall that since this is the next successive q-gram, this computation takes constant time. If a node with label $\phi(S[k+1, k+(q-1)])$ already exists we check if there is an edge from $\phi(S[k, k+(q-1)-1])$ to $\phi(S[k+1, k+(q-1)])$. If such an edge exists we increment its counter by one. If it does not exist we create it and set its counter to 1. If a node with label $\phi(S[k+1, k+(q-1)])$ does not exist we create it along with an edge from $\phi(S[k, k+(q-1)-1])$ to it.

**Lemma 2.6** *For a string $S$ of length $N$, the algorithm is a Monte Carlo-type randomized algorithm that builds the q-gram graph $G_q(S)$ in $O(N)$ expected time.*

## 2.4 Algorithm

Our main algorithm is comprised of four steps: preparing the SLP, constructing the q-gram graph from the SLP, turning it into a CS-tree, and computing the suffix tree of the CS-tree. Ultimately the algorithm produces a suffix tree containing the reversed q-grams of $S$, so to answer a query for a q-gram $s$ we will have to lookup $s^R$ in the suffix tree. Below we will describe the algorithm and we will show that it runs in $O(qn)$ expected time while using $O(n + q + k_{S,q})$ space; an improvement over the best known algorithm in terms of space usage. The catch is that a frequency query to the resulting data structure may yield incorrect results due to randomization. However, we show how to turn the algorithm from a Monte Carlo to a Las Vegas-type randomized algorithm with constant overhead. Finally, we show that by decompressing substrings of $S$ in a specific order, we can construct the q-gram graph by decompressing exactly the same number of characters as decompressed by the best known algorithm.

The algorithm is as follows. Figure 2.2 shows an example of the data structures after each step of the algorithm.

**Preprocessing.** As the first step of our algorithm we preprocess the SLP such that we know the size of the string derived from a production rule, $|X_i|$, and the number of occurrences in the derivation tree, $occ(X_i)$. We also prepare the SLP for linear time prefix and suffix decompressions using Lemma 2.4.

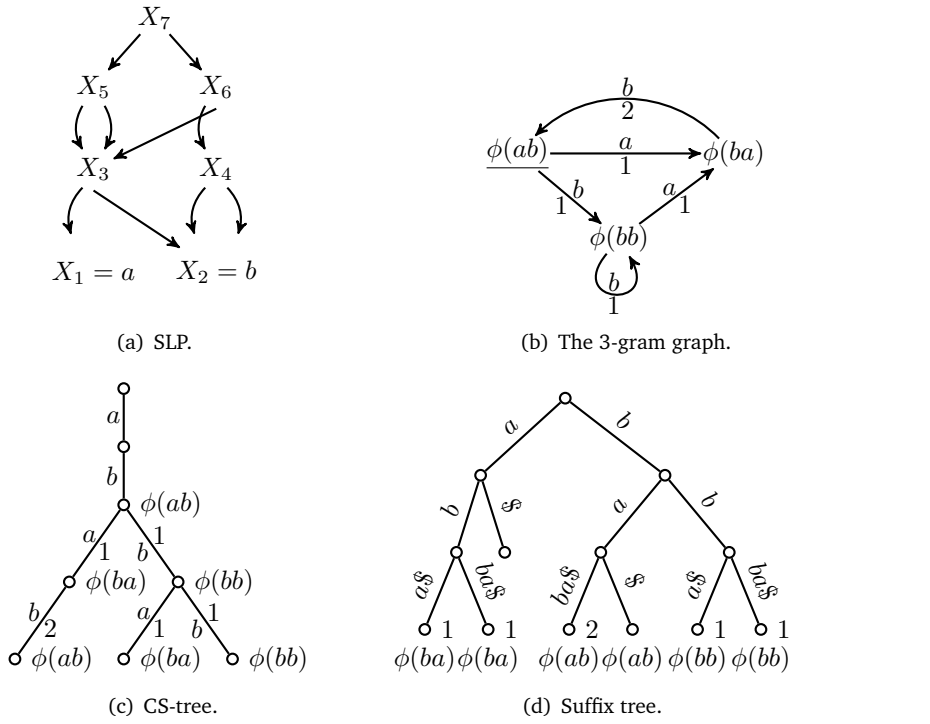
**Computing the q-gram graph.** In this step we construct the q-gram graph $G_q(S)$ from the SLP $\mathcal{S}$. Initially we choose a suitable fingerprint function for the q-gram graph construction algorithm and proceed as follows. For each production rule $X_i = X_l X_r$ in $\mathcal{S}$, such that $|X_i| \geq q$, we decompress its relevant substring $r_{X_i}$. Recall from the definition of relevant substrings that $r_{X_i}$ is the concatenation of the $q-1$ length suffix of $t_{X_l}$ and the $q-1$ length prefix of $t_{X_r}$. If $|X_l| \leq q-1$ we decompress the entire string $t_{X_l}$, and similarly for $t_{X_r}$. Given $r_{X_i}$ we compute the fingerprint of the first $(q-1)$-gram, $\phi(r_{X_i}[0,(q-1)-1])$, and find the node in $G_q(S)$ with this fingerprint as its label. The node is created if it does not exist. Now the construction of $G_q(S)$ can continue from this node, albeit with the following change to the construction algorithm. When incrementing the counter of an edge we increment it by $occ(X_i)$ instead of 1.

The q-gram graph now contains the information needed for the q-gram profile; namely the frequencies of the q-grams in $S$. The purpose of the next two steps is to restructure the graph to a data structure that supports frequency queries in $O(q)$ time.

**Transforming the q-gram graph to a CS-tree.** The CS-tree that we want to create is basically the depth-first tree of $G_q(S)$ with the extension that all edges in $G_q(S)$ are also in the tree. We create it as follows. Let the start node of $G_q(S)$ be the node whose label match the fingerprint of the first $q-1$ characters of $S$. Do a depth-first traversal of $G_q(S)$ starting from the start node. For a previously unvisited node, create a node in the CS-tree with an incoming edge from its predecessor. When reaching a previously visited node, create a new leaf in the CS-tree with an incoming edge from its predecessor. Labels on nodes and edges are copied from their corresponding labels in $G_q(S)$. We now create a path of length $q-1$ with the first $q-1$ characters of $S$ as labels on its edges. We set the last node on this path to be the root of the depth-first tree. The first node on the path is the root of the final CS-tree.

**Computing the suffix tree of the CS-tree.** Recall that a suffix in the CS-tree starts in a node and ends in the root of the tree. Usually we store a pointer from a leaf in the suffix tree to the node in the CS-tree from which the particular suffix starts. However,

when we construct the suffix tree, we store the value of the counter of the first edge in the suffix as well as the label of the first node on the path of the suffix.




(a) SLP.

(b) The 3-gram graph.

(c) CS-tree.

(d) Suffix tree.

**Figure 2.2:** An SLP compressing the string `ababbbab`, and the data structures after each step of the algorithm executed with $q = 3$.

Our algorithm resembles the one by Goto et al. [52]. The main difference between the algorithms is that Goto et al. use the so-called neighbour graph to capture the q-grams of $S$ where we use the q-gram graph. This is also the key to the improvement in space usage. If a q-gram occurs in several relevant substrings it will occur several times in the neighbour graph but only once in the q-gram graph.

### *2.4.1 Correctness*

Before showing that our algorithm is correct, we will prove some crucial properties of the q-gram graph, the CS-tree, and the suffix tree of the CS-tree subsequent to their construction in the algorithm.

**Lemma 2.7** *The q-gram graph $G_q(S)$ constructed from the SLP is connected.*

**Proof** Consider a production rule $X_i = X_l X_r$. If $|X_i| \leq 2(q-1)$ we decompress the entire string $t_{X_i}$ and insert it into the q-gram graph, and we know that $G_q(t_{X_i})$ is connected. Assume that $G_q(t_{X_l})$ and $G_q(t_{X_r})$ are both connected. We know from Lemma 2.3 that if we insert all the relevant substrings of the nodes reachable from $X_l$ (including $X_l$) into the graph, then it will contain all $(q-1)$-grams of $t_{X_l}$. Since the first $q-1$ characters of $r_{X_i}$ is a suffix of $t_{X_l}$, the subgraphs $G_q(t_{X_l})$ and $G_q(r_{X_i})$ will have at least one node in common, and similarly for $G_q(t_{X_r})$ and $G_q(r_{X_i})$. Therefore, $G_q(X_i)$ is connected.

**Lemma 2.8** *Assuming that we are given a fingerprint function $\phi$ that is collision free for substrings of length $q-1$ in $S$, then the CS-tree built by the algorithm contains each distinct q-gram in $S$ exactly once.*

**Proof** Let $v$ be a node with an outgoing edge $e$ in $G_q(S)$. The combination of the label of $v$ followed by the character on $e$ is a distinct q-gram and occurs only once in $G_q(S)$ due to the way we construct it. There may be several paths of length $q-1$ ending in $v$ spelling the same string $s$, and because the fingerprint function is deterministic, there can not be a path spelling $s$ ending in some other node. Since the depth-first traversal of $G_q(S)$ only visits $e$ once, the resulting CS-tree will only contain the combination of the labels on $v$ and $e$ once.

**Lemma 2.9** *Assuming that we are given a fingerprint function $\phi$ that is collision free for substrings of length $q-1$ in $S$, then any node $v$ in the suffix tree of the CS-tree with $sd(v) \geq q$ is a leaf.*

**Proof** Each suffix of length $\geq q$ in the CS-tree has a distinct $q$ length prefix (Lemma 2.8), so therefore each node in the suffix tree with string depth $\geq q$ is a leaf.

We have now established the necessary properties to prove that our algorithm is correct.

**Lemma 2.10** *Assuming that we are given a fingerprint function $\phi$ that is collision free on all substrings of length $q-1$ of $S$, our algorithm correctly computes a q-gram profile for $S$.*

**Proof** Our algorithm inserts each relevant substring $r_{X_i}$ exactly once, and if a q-gram $s$ occurs $socc(s, r_{X_i})$ times in $r_{X_i}$, the counter on the edge representing $s$ is incremented by exactly $socc(s, r_{X_i}) \cdot occ(X_i)$. From Lemma 2.3 we then know that when $G_q(S)$ is fully constructed, the counters on its edges correspond to the frequencies of the q-grams in $S$. Since $G_q(S)$ is connected (Lemma 2.7) the tree created by the algorithm is a CS-tree that contains each q-gram from $G_q(S)$ exactly once (Lemma 2.8). Finally, we know from Lemma 2.9 that a node $v$ with $sd(v) \geq q$ in the suffix tree is a leaf, so searching for a string of length $q$ in the suffix tree will yield a unique result and can be done in $O(q)$ time.

### *2.4.2 Analysis*

**Theorem 2.2** *The algorithm runs in $O(qn)$ expected time and uses $O(n+q+k_{S,q})$ space.*

**Proof** Let $\mathcal{S}_q = \{X_i \mid X_i \in \mathcal{S} \text{ and } |X_i| \geq q\}$ be the set of production rules that have a relevant substring. For each production rule $X_i = X_l X_r \in \mathcal{S}_q$ we decompress its relevant substring of size $|r_{X_i}|$ and insert it into the q-gram graph. Since $r_{X_i}$ is comprised of the suffix of $t_{X_l}$ and the prefix of $t_{X_r}$ we know from Lemma 2.4 that $r_{X_i}$ can be decompressed in $O(|r_{X_i}|)$ time. Inserting $r_{X_i}$ into the q-gram graph can be done in $O(|r_{X_i}|)$ expected time (Lemma 2.6). Since $|\mathcal{S}_q| = O(n)$ and $q \leq |r_{X_i}| \leq 2(q-1)$ this step of the algorithm takes $O(qn)$ time. When transforming the q-gram graph to a CS-tree we do one traversal of the graph and add $q-1$ nodes, so this step takes $O(q + k_{S,q})$ time. Constructing the suffix tree takes expected linear time in the size of the CS-tree if we hash the characters of the alphabet to a polynomial range first (Lemma 2.1). Finally, observe that since our algorithm is correct, it detects all q-grams in $S$ and therefore there can be at most $k_{S,q} \leq \sum_{X_i \in \mathcal{S}_q} |r_{X_i}| = O(qn)$ distinct q-grams in $S$. Thus, the expected running time of our algorithm is $O(qn)$.

In the preprocessing step of our algorithm we use $O(n)$ space to store the size of the derived substrings and the number of occurrences in the derivation tree as well as the data structure needed for linear time prefix and suffix decompressions (Lemma 2.4). The space used by the q-gram graph is $O(k_{S,q})$, and when transforming it to a CS-tree we add at most one new node per edge in the graph and extend it by $q-1$ nodes and edges. Thus, its size is $O(q+k_{S,q})$. The CS-tree contains $O(q+k_{S,q})$ suffixes, so the size of the suffix tree is $O(q+k_{S,q})$. In total our algorithm uses $O(n+q+k_{S,q})$ space.

### *2.4.3 Verifying the fingerprint function*

Until now we have assumed that the fingerprints used as labels for the nodes in the q-gram graph are collision free. In this section we describe an algorithm that verifies if the chosen fingerprint function is collision free using the suffix tree resultant from our algorithm.

If there is a collision among fingerprints, the q-gram graph construction algorithm will add an edge such that there are two paths of length $q-1$ ending in the same node while spelling two different strings. This observation is formalized in the next lemma.

**Lemma 2.11** *For each node $v$ in $G_q(S)$, if every path of length $q-1$ ending in $v$ spell the same string, then the fingerprint function used to construct $G_q(S)$ is collision free for all $(q-1)$-grams in $S$.*

**Proof** From the q-gram graph construction algorithm we know that we create a path of characters in the same order as we read them from $S$. This means that every path of length $q-1$ ending in a node $v$ represents the $q-1$ characters generating the fingerprint stored in $v$, regardless of what comes before those $q-1$ characters. If all the paths of length $q-1$ ending in $v$ spell the same string $s$, then we know that there is no other substring $s' \neq s$ of length $q-1$ in $S$ that yields the fingerprint $\phi(s)$.

It is not straightforward to check Lemma 2.11 directly on the q-gram graph without using too much time. However, the error introduced by a collision naturally propagates to the CS-tree and the suffix tree of the CS-tree, and as we shall now see, the suffix tree offers a clever way to check for collisions. First, recall that in a leaf $v$ in the suffix tree, we store the fingerprint of the reversed prefix of length $q-1$ of the suffix ending in $v$. Now consider the following property of the suffix tree.

**Lemma 2.12** *Let $v_\phi$ be the fingerprint stored in a leaf $v$ in the suffix tree. The fingerprint function $\phi$ is collision free for $(q-1)$-grams in $S$ if $v_\phi \neq u_\phi$ or $sd(nca(v,u)) \geq q-1$ for all pairs $v, u$ of leaves in the suffix tree.*

**Proof** Consider the contrapositive statement: If $\phi$ is not collision free on $S$ then there exists some pair $v, u$ for which $v_\phi = u_\phi$ and $sd(nca(v,u)) < q-1$. Assume that there is a collision. Then at least two paths of length $q-1$ spelling the same string end in the same node in $G_q(S)$. Regardless of the order of the nodes in the depth-first traversal of $G_q(S)$, the CS-tree will have two paths of length $q-1$ spelling different strings and yet starting in nodes storing the same fingerprint. Therefore, the suffix tree contains two suffixes that differ by at least one character in their $q-1$ length prefix while ending in leaves storing the same fingerprint, which is what we want to show.

Checking if there exists a pair of leaves where $v_\phi = u_\phi$ and $sd(nca(v,u)) < q-1$ is straightforward. For each leaf we store a pointer to its ancestor $w$ that satisfies $sd(w) \geq q-1$ and $sd(parent(w)) < q-1$. Then we visit each leaf $v$ again and store $v_\phi$ in a dictionary along with the ancestor pointer just defined. If the dictionary already

contains $v_\phi$ and the ancestor pointer points to a different node, then it means that $v_\phi = u_\phi$ and $sd(nca(v,u)) < q-1$ for some two leaves.

The algorithm does two passes of the suffix tree which has size $O(q + k_{S,q})$. Using a hashing scheme for the dictionary we obtain an algorithm that runs in $O(q + k_{S,q})$ expected time.

### *2.4.4 Eliminating redundant decompressions*

We now present an alternative approach to constructing the q-gram graph from the SLP. The resulting algorithm decompresses fewer characters.

In our first algorithm for constructing the q-gram graph we did not specify in which order to insert the relevant substrings into the graph. For that reason we do not know from which node to resume construction of the graph when inserting a new relevant substring. So to determine the node to continue from, we need to compute the fingerprint of the first $(q-1)$-gram of each relevant substring. In other words, the relevant substrings are overlapping, and consequently some characters are decompressed more than once. Our improved algorithm is based on the following observation. Consider a production rule $X_i = X_l X_r$. If all relevant substrings of production rules reachable from $X_l$ (including $r_{X_l}$) have been inserted into the graph, then we know that all q-grams in $t_{X_l}$ are in the graph. Since the $q-1$ length prefix of $r_{X_i}$ is also a suffix of $t_{X_l}$, then we know that a node with the label $\phi(r_{X_i}[0,(q-1)-1])$ is already in the graph. Hence, after inserting all relevant substrings of production rules reachable from $X_l$ we can proceed to insert $r_{X_i}$ without having to decompress $r_{X_i}[0,(q-1)-1]$.

**Algorithm.** First we compute and store the size of the relevant substring $|r_{X_i}| = \min(q-1,|X_l|) + \min(q-1,|X_r|)$ for each production rule $X_i = X_l X_r$ in the subset $\mathcal{S}_q = \{X_i \mid X_i \in \mathcal{S} \text{ and } X_i \geq q\}$ of the production rules in the SLP. We maintain a linked list $L$ with a pointer to its head and tail, denoted by $head(L)$ and $tail(L)$. The list is initially empty.

We now start decompressing $S$ by traversing the SLP depth-first, left-to-right. When following a pointer from $X_i$ to a right child, and $X_i \in \mathcal{S}_q$, we add $X_i$ and the sentinel value $|r_{X_i}| - (q-1)$ to the back of $L$. As characters are decompressed they are fed to the q-gram graph construction algorithm, and when a counter on an edge in $G_q(S)$ is incremented, we increment it by $occ(head(L))$. For each character we decompress, we decrement the sentinel value for $head(L)$, and if this value becomes $0$ we remove the head of the list and set $head(L)$ to be the next production rule in the list. Note that when $L$ is empty in the beginning of the execution of the algorithm we do not alter any values.

When leaving a node $X_i \in \mathcal{S}_q$ we mark it as visited and store a pointer from $X_i$ to the node with label $\phi(t_{X_i}[|X_i|-(q-1),|X_i|-1])$ in $G_q(S)$, i.e., the node labelled with the suffix of length $q-1$ of $t_{X_i}$. To do this we need to consider two cases. Let $X_i = X_l X_r$. If $X_r \in \mathcal{S}_q$ then we copy the pointer from $X_r$. If $X_r \notin \mathcal{S}_q$ then $\phi(t_{X_i}[|X_i|-(q-1),|X_i|-1])$ is the most recently visited node in $G_q(S)$.

If we encounter a node that has been marked as visited, we decompress its prefix of length $q-1$ using the data structure of Lemma 2.4, set the node with label $\phi(t_{X_i}[|X_i| - (q-1),|X_i|-1])$ to be the node from where construction of the q-gram graph should continue, and do not proceed to visit its children nor add it to $L$.

**Analysis.** Assume without loss of generality that the algorithm is at a production rule deriving the string $t_{X_i} = t_{X_l} t_{X_r}$ and all q-grams in $t_{X_l}$ are in $G_q(S)$. There is always such a rule, since we start by decompressing the string derived by the left child of the leftmost rule in $\mathcal{S}_q$. For each variable $X_i$ added to $L$ we decompress $|r_{X_i}| - (q-1)$ characters before $X_i$ is removed from the list. We only add a variable once to the list, so the total number of characters decompressed is at most $(q-1) + \sum_{X_i \in \mathcal{S}_q} |r_{X_i}| - (q-1) = O(N-\alpha)$,

and we hereby obtain our result from Theorem 2.1. This is fewer characters than our first algorithm that require $\sum_{X_i \in \mathcal{S}_q} |r_{X_i}|$ characters to be decompressed. Furthermore, it is exactly the same number of characters decompressed by the fastest known algorithm due to Goto et al. [52].

# CHAPTER 3

## FINGERPRINTS IN COMPRESSED STRINGS

Philip Bille* Patrick Hagge Cording* Inge Li Gørtz*
Benjamin Sach† Hjalte Wedel Vildhøj* Søren Vind*,‡

* Technical University of Denmark, DTU Compute
† University of Warwick, Department of Computer Science

**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, $\ell$ denotes the length of the LCE.

### 3.1 Introduction

Given a string $S$ of size $N$ and a Karp-Rabin fingerprint function $\phi$, 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 [69] 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., [7, 10, 29–31, 37, 49, 64, 87]).

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

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

answered in $O(1)$ time. However, this data structure uses $O(N)$ space which can be exponential in $n$. Another approach is to use the data structure of Gąsieniec et al. [50] 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. [21] giving $O(\log N + j - i)$ time for a FINGERPRINT$(i, j)$ query. Note that the query time depends on the length of the decompressed string which can be large. For the case of *balanced* grammars (by height or weight) Gagie et al. [41] showed how to efficiently compute fingerprints for indexing 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 LZ78 compression scheme [110] and its variations. Our data structures give the following theorem.

**Theorem 3.1** *Let $S$ be a string of length $N$ compressed into an SLP $\mathcal{S}$ 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 $\mathcal{S}$ 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 3.1(i), we combine this with the fast random access data structure by Bille et al. [21]. 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 3.2** *Let $S$ be a string of length $N$ compressed into an SLP $\mathcal{S}$ 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.

### 3.1.1 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 $\ell$ 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 $S$ [57]. Other trade-offs are obtained by doing an exponential search over the fingerprints of strings starting in $i$ and $j$ [20]. 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 3.3.

**Theorem 3.3** *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 $\ell$ 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.*

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

The following corollary follows immediately because an LZ77 compression [109] consisting of $n$ phrases can be transformed to an SLP with $O(n \log \frac{N}{n})$ production rules [26, 88].

**Corollary 3.1** *We can solve the* LCE *problem in $O(n \log \frac{N}{n})$ space and $O(\log \ell \log N)$ query time for LZ77 compression.*

Finally, the LZ78 compression can be modelled by a Linear SLP $\mathcal{S}_L$ with constant overhead. Consider an LZ78 compression with $n$ phrases, denoted $r_1, \ldots, r_n$. A terminal phrase corresponds to a leaf in $\mathcal{S}_L$, and each phrase $r_j = (r_i, a)$, $i < j$, corresponds to a node $v \in \mathcal{S}_L$ 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 3.2** *We can solve the* LCE *problem in $O(n)$ space and $O(\log \ell \log \log \ell + \log \log N)$ query time for LZ78 compression.*

## 3.2 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) $\mathcal{S}$ is a context-free grammar in Chomsky normal form that we represent as a node-labeled and ordered directed acyclic graph. Each leaf in $\mathcal{S}$ is labelled with a character, and corresponds to a terminal grammar production rule. Each internal node in $\mathcal{S}$ is labeled with a nonterminal rule from the grammar. The unique

string $S(v)$ of length $size(v) = |S(v)|$ 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. We let $root(\mathcal{S})$ denote the root of $\mathcal{S}$, and $left(v)$ and $right(v)$ denote the left and right child of an internal node $v \in \mathcal{S}$, respectively.

A Linear SLP $\mathcal{S}_L$ is an SLP where we allow $root(\mathcal{S}_L)$ to have more than two children. All other internal nodes $v \in \mathcal{S}_L$ have a leaf as $right(v)$. Although similar, this is not the same definition as given for the Relaxed SLP by Claude and Navarro [27]. 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 [57] to SLPs as in [21]. For each node $v \in \mathcal{S}$, 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 $size(u) \leq size(v)/2$ 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 $leaf(H(v))$, is well-defined for each node $v \in \mathcal{S}$.

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* $\text{PRED}(q)$ is the largest integer $r^- \in R$ such that $r^- \leq q$, while the answer to a *successor query* $\text{SUCC}(q)$ 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 $O(n)$ [81, 104, 107].

Given a rooted tree $T$ with $n$ vertices, we let $depth(v)$ 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* $\text{LA}(v, i)$, asking for the ancestor $u$ of $v \in T$ such that $depth(u) = depth(v) - i$. There is a level ancestor data structure answering queries in $O(1)$ time using $O(n)$ space [35] (see also [2, 16, 17]).

### 3.2.1 Fingerprinting

The Karp-Rabin fingerprint [69] of a string $x$ is defined as $\phi(x) = \sum_{i=1}^{|x|} x[i] \cdot c^i \bmod p$, where $c$ is a randomly chosen positive integer, and $2N^{c+4} \leq p \leq 4N^{c+4}$ is a prime. Karp-Rabin fingerprints guarantee that given two strings $x$ and $y$, if $x = y$ then $\phi(x) = \phi(y)$. Furthermore, if $x \neq y$, then with high probability $\phi(x) \neq \phi(y)$. Fingerprints can be composed and subtracted as follows.

**Lemma 3.1** *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 3.1 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.

### 3.3 Basic fingerprint queries in SLPs

We now describe a simple data structure for answering FINGERPRINT$(1, i)$ queries for a string $S$ compressed into a SLP $\mathcal{S}$ 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 $\mathcal{S}$.

The data structure stores $size(v)$ and the fingerprint $\phi(S(v))$ of the string produced by each node $v \in \mathcal{S}$. To compose the fingerprint $f = \phi(S[1, i])$ we start from the root of $\mathcal{S}$ 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 $\mathcal{S}$, so the space usage is $O(n)$.

### 3.4 Faster fingerprints in SLPs

Using the data structure of Bille et al. [21] to perform random access queries allows for a faster way to answer FINGERPRINT$(1, i)$ queries.

**Lemma 3.2 ([21])** *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 $O(\log(size(v)))$ time, at the same time reporting the sequence of heavy paths and their entry- and exit points in the corresponding depth-first traversal of $\mathcal{S}(v)$.*

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 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 3.1 for the relationship between $u$, $v$ and the defined strings.

Let each node $v \in \mathcal{S}$ 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 [21], 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 3.2. 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 $O(\log N)$.

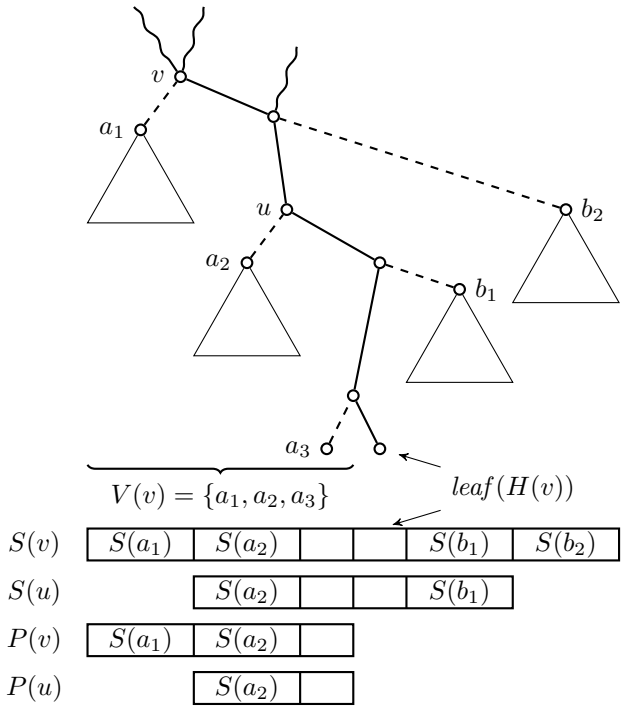



**Figure 3.1:** Figure showing how $S(v)$ and its prefix $P(v)$ is composed of substrings generated by the left children $a_1, a_2, a_3$ and right children $b_1, b_2$ of the heavy path $H(v)$. Also illustrates how this relates to $S(u)$ and $P(u)$ for a node $u \in H(v)$.

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 $f = f \oplus \phi(S(\mathit{left}(u)))$. If $u$ is a leaf, set $f = f \oplus \phi(S(u))$ to include the fingerprint of the terminal character.

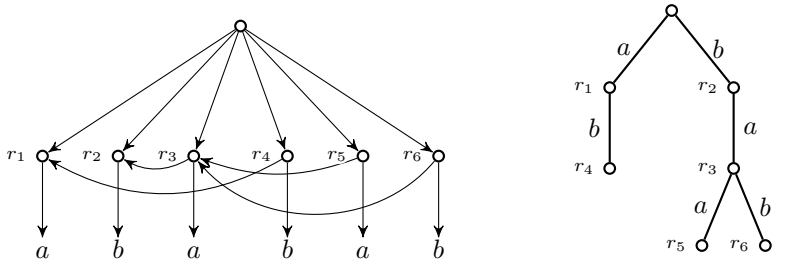
Remember that $P(v)$ is exactly the string generated from $v$ along $H$, produced by the left children of nodes on $H$ where the heavy path was extended to the right child. Thus, this method corresponds exactly to adding the fingerprint for the substrings generated by all left children of nodes on $H$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

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 $\mathcal{S}$, so the space usage is $O(n)$. This concludes the proof of Theorem 3.1(i).

## 3.5 Faster fingerprints in Linear SLPs

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

We also define the dictionary tree $F$ over $\mathcal{S}_L$ as follows. Each node $v \in \mathcal{S}_L$ corresponds to a single vertex $v^F \in F$. There is an edge $(u^F, v^F)$ labeled $c$ if $u = \mathit{left}(v)$ and




(a) Linear SLP. (b) Dictionary tree.

**Figure 3.2:** A Linear SLP compressing the string `abbaabbaabab` and the dictionary tree obtained from the Linear SLP.

$c = S(\mathit{right}(v))$. If $v$ is a leaf, there is an edge $(\mathit{root}(F), v^F)$ labeled $S(v)$. That is, a left child edge of $v \in \mathcal{S}_L$ is converted to a parent edge of $v^F \in F$ labeled like the right child leaf of $v$. Note that for any node $v \in \mathcal{S}_L$ except the root, producing $S(v)$ is equivalent to following edges and reporting edge labels on the path from $\mathit{root}(F)$ to $v^F$. Thus, the prefix of length $a$ of $S(v)$ may be produced by reporting the edge labels on the path from $\mathit{root}(F)$ 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 $\mathcal{S}_L$. We store the dictionary tree $F$ over $\mathcal{S}_L$, augment it with a level ancestor data structure, and add bidirectional pointers between $v \in \mathcal{S}_L$ and $v^F \in F$. Finally, for each node $v \in \mathcal{S}_L$, we store the fingerprint of the string it produces, $\phi(S(v))$.

A query $\textsc{Fingerprint}(1, i)$ is answered as follows. Let $R(m)$ be the predecessor of $i$ among $R(0), R(1), \ldots, R(k)$. Compose the answer to $\textsc{Fingerprint}(1, i)$ 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 $\mathit{root}(\mathcal{S}_L)$) from a universe of size $N$. The data structure uses $O(n)$ space, as there is a bijection between nodes in $\mathcal{S}_L$ and vertices in $F$, and we only spend constant additional space per node in $\mathcal{S}_L$ and vertex in $F$. This concludes the proof of Theorem 3.1(ii).

## 3.6 Finger fingerprints in Linear SLPs

The $O(\log \log N)$ running time of a $\textsc{Fingerprint}(1, i)$ 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 $\textsc{Fingerprint}(1, i)$ 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.

### 3.6.1 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. [14] present a succinct solution for solving the finger predecessor problem relying on a modification of z-fast tries. Other previous work present dynamic finger search trees on the word RAM [9, 65]. Here, we use a simple reduction for solving the finger predecessor problem using any predecessor data structure as a black box.

**Lemma 3.3** *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 worst case 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 $\text{SUCC}(q)$ and $\text{PRED}(q)$ 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 $\text{SUCC}(q)$ 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 $\text{SUCC}(q)$ and its left neighbour in $T$ as $\text{PRED}(q)$.

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 [40, 56] 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 $\text{PRED}(q)$ and $\text{SUCC}(q)$ 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

[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 $2^j$.

proceed as before using $g_l$ as the finger into $T$ and query point $q$. This gives $p = \text{PRED}(q)$ and $s = \text{SUCC}(q)$ 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 $O(s(N, \frac{n}{\log N}) \log N)$.

Using the van Emde Boas predecessor data structure [81, 104, 107] with $t(N, n) = O(\log \log N)$ query time using $s(N, n) = O(n)$ space, we obtain the following corollary.

**Corollary 3.3** *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 worst case time $O(\log \log |f - q|)$ and space $O(n)$.*

### *3.6.2 Finger Fingerprints*

We can now prove Theorem 3.2. 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 3.3 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 $O(\log \log |i - j|)$.

## 3.7 Longest Common Extensions in Compressed Strings

Given an SLP $\mathcal{S}$, the longest common extension (LCE) problem is to build a data structure for $\mathcal{S}$ 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 3.3.

### *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 3.4** *For any string $S$ and any partition $S = s_1 s_2 \cdots s_t$ of $S$ into $k$ non-empty substrings called phrases, $\ell = \text{LCE}(i, j)$ 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 $\ell = \text{LCE}(i, j)$ 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 $\ell$ to two phrase substrings, and we can find the exact value using $O(\log \ell)$ type 1 comparisons.

Otherwise, $\text{LCE}(i, j) = k + \text{LCE}(i + k + 1, j + k + 1)$ and either $i + k + 1$ or $j + k + 1$ is a start position. This leaves us with the task of describing how to answer $\text{LCE}(i, j)$, 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 3.3 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 3.2 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.

### *3.7.2 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 $\phi$ is *good*, i.e., collision-free on all substrings compared in the computation of $\text{LCE}(i, j)$. 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).

#### 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 3.4). We thus only have to check that $\phi$ 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 $\phi$ is not good. If all rounds are successfully verified, we report that $\phi$ 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 $\phi$ 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, $\phi$ can be verified in $O(N^2/n \log N)$ time and $O(n)$ space.

**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 3.4 with the partition of $S$ being the root substrings. We verify that $\phi$ 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 $\phi$ 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 $\phi$ 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 $\phi$ 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.

# CHAPTER 4

# COMPRESSED SUBSEQUENCE MATCHING AND PACKED TREE COLORING

Philip Bille Patrick Hagge Cording Inge Li Gørtz

Technical University of Denmark, DTU Compute

**Abstract**

We present a new algorithm for subsequence matching in grammar compressed strings. Given a grammar of size $n$ compressing a string of size $N$ and a pattern string of size $m$ over an alphabet of size $\sigma$, our algorithm uses $O(n + \frac{n\sigma}{w})$ space and $O(n + \frac{n\sigma}{w} + m \log N \log w \cdot occ)$ or $O(n + \frac{n\sigma}{w} \log w + m \log N \cdot occ)$ time. Here $w$ is the word size and $occ$ is the number of minimal occurrences of the pattern. Our algorithm uses less space than previous algorithms and is also faster for $occ = o(\frac{n}{\log N})$ occurrences. The algorithm uses a new data structure that allows us to efficiently find the next occurrence of a given character after a given position in a compressed string. This data structure in turn is based on a new data structure for the tree color problem, where the node colors are packed in bit strings.

## 4.1 Introduction

Subsequence matching is a variant of string pattern matching where an occurrence of the pattern in the text must contain all the characters of the pattern, but not necessarily contigously. If there is such an occurrence we say that the pattern is a subsequence of a substring of the text. A pattern string $P$ is a subsequence of another string $S$ (the text) if $P$ can be obtained by deleting characters in $S$. Just answering if $P$ is a subsequence of $S$ is easily solved in linear time by scanning $S$ from left to right looking for the characters of $P$. In the *subsequence matching problem* the goal is to find and report the positions of all minimal substrings of $S$ that contain $P$ as a subsequence. A substring is said to be minimal if shortening it implies that $P$ is no longer a subsequence of that substring. More formally, if $P$ is a subsequence of the substring $S[i, j]$, then $i, j$ is a minimal occurrence if $P$ is not a subsequence of $S[i + 1, j]$ or $S[i, j - 1]$.

If the text is large and sufficiently repetetive, it might be useful to compress it. In this paper we consider the *compressed subsequence matching problem* where we are given a grammar $\mathcal{S}$ of size $n$ compressing a string $S$ of size $N$ and a pattern string $P$ of size $m$ over an alphabet of size $\sigma$. We present a new algorithm for compressed subsequence matching which is space efficient and is faster than the previously fastest algorithm for a bounded number of occurrences. Our algorithm relies on a method that is different from the ones used by previous algorithms.

Subsequence matching is useful when searching sequential log data for a sequence of events that may be separated by other events. Say for instance that we are running a webserver and we want to know how often a visitor has found her way to subpage $C$ through page $A$ and then $B$. We then set $P = ABC$ and apply a subsequence matching algorithm to the contents of the log file. Many applications will automatically compress log data to save space, and so the bottleneck of the procedure becomes decompression of the data. In this case, processing the data without fully decompressing it, is crucial. Subsequence matching was also considered in relation to knowledge discovery and data mining [78].

Several algorithms have been presented for uncompressed strings [11, 22, 25, 33, 34, 78, 101]. The fastest of these is due to Das et al. [34]. Since it is an online algorithm we may apply it to the compressed version without having to store the entire decompressed string, and we get an algorithm with running time $O(\frac{Nm}{\log m})$ that uses $O(n+m)$ space. The first algorithm with time complexity independent of the size of the string was presented by Cegielski et al. [24] in 2006. Its runnning time is $O(nm^2 \log m + occ)$ time and it uses $O(nm^2)$ space. Using a different approach, Tiskin improved the running time to $O(nm^{1.5}+occ)$ [98] and later even further to $O(nm \log m+occ)$ [99]. The space usage of his algorithms is $O(nm)$. The most recent improvement is due to Yamamoto et al. [108] who present an algorithm based on the ideas of Cegielski et al. that runs in $O(nm + occ)$ time and $O(nm)$ space. All results are summarized in Table 1.

| Time complexity | Space complexity | Author(s) |
|---|---|---|
| $O(\frac{Nm}{\log m})$ | $O(n+m)$ | Das et al. [34] |
| $O(nm^2 \log m + occ)$ | $O(nm^2)$ | Cegielski et al. [24] |
| $O(nm^{1.5} + occ)$ | $O(nm)$ | Tiskin [98] |
| $O(nm \log m + occ)$ | $O(nm)$ | Tiskin [99] |
| $O(nm + occ)$ | $O(nm)$ | Yamamoto et al. [108] |
| $O(n + \frac{n\sigma}{w} + m \log N \log w \cdot \mathsf{occ})$<br>$O(n + \frac{n\sigma}{w} \log w + m \log N \cdot \mathsf{occ})$ | $O(n + \frac{n\sigma}{w})$ | This paper |

**Table 4.1:** Time and space complexities of algorithms for compressed subsequence matching.

Our algorithm assumes that the input grammar is a Straight Line Program (SLP). An SLP is an acyclic grammar in Chomsky normal form, i.e., each nonterminal production rule expands to two other rules and generates one string only. Any grammar producing a single string can be transformed to an SLP with linear overhead so our results hold for grammar-compressed strings in general. Moreover, SLPs are widely studied because they model many well-known compression schemes, such as LZ77 [109], LZ78 [110], and Re-Pair [73] with little overhead [26, 88]. The following theorem is the main result of this work.

**Theorem 4.1** *Given an SLP $\mathcal{S}$ of size $n$ compressing a string $S$ of size $N$ and a pattern $P$ of size $m$ over an alphabet of size $\sigma$, compressed subsequence matching can be solved in $O(n + \frac{n\sigma}{w})$ words of space and time*

*(i)* $O(n + \frac{n\sigma}{w} + m \log N \log w \cdot \mathsf{occ})$*, or*

*(ii)* $O(n + \frac{n\sigma}{w} \log w + m \log N \cdot \mathsf{occ})$

*in the word RAM model with word size $w \geq \log N$, and where occ is the number of minimal occurrences of $P$ in $S$.*

Our new algorithm uses less space (linear in $n$ if $\sigma \leq w$) and is also faster than the previously fastest algorithm for few occurrences when $\sigma \leq m$. Particularly, solution

$(ii)$ is faster if the number of occurrences is bounded by $o(\frac{n}{\log N})$. Note that we in $O(n+m)$ expected time or $O(n \log m + m)$ time and using $O(m)$ additional extra space can guarantee that $\sigma \leq m$ always holds[1].

The algorithm is based on the idea of a simple algorithm for subsequence matching in uncompressed strings which basically scans the string for occurrences of the pattern. We speed up the scanning on compressed strings by introducing the first data structure for SLPs that supports labelled successor queries. The answer to a labelled succesor query $\textsc{ls}(i, c)$ on a string is the index of the first character $c$ occurring after position $i$ in the string. An essential part of this data structure is a new data structure for the tree color problem. This problem is to preprocess a tree where each node is colored by zero or more colors, such that given a node $v$ and a color $c$, we may efficiently answer a first colored ancestor query, i.e., compute the lowest ancestor of $v$ with color $c$. Additionally, this data structure also supports a new type of query we call the last colored ancestor. Here the query is two nodes $u$ and $v$ and a color $c$, and the answer is the highest node on the path from $u$ to $v$ with color $c$. These results may be of independent interest.

This paper is organized such that we start by describing our new result for the tree color problem (after a section of preliminaries), then move on to the labelled successor data structure, and finally describe the algorithm for subsequence matching.

## 4.2 Preliminaries

**Bit Strings.** We will use bit strings to represent sets. In a bit string $B = b_1 b_2 \dots b_u$ representing a set $\mathcal{B}$ of elements from a universe of size $u$, $b_i = 1$ iff element $i$ is in $\mathcal{B}$. $B = [0]^u$ denotes the empty set. The operators $\wedge$, $\vee$, and $\oplus$ denote the bitwise AND, OR, and exclusive OR (XOR) of two bit strings. The negation of a bit string $B$ is $\overline{B}$. A *summary* $B_s$ of $k$ bit strings $B_1, B_2, \dots, B_k$ of equal length is $B_s = B_1 \vee B_2 \vee \dots \vee B_k$. For a bit string of length $w$ we assume that the mask of any constant can be computed in $O(1)$ time. Given a bit string $B = b_1 b_2 \dots b_w$, $b_1$ is the most significant bit. The index of the least significant set bit can be found in $O(1)$ time from $\log_2((\overline{B}+1) \wedge B)$. Finding the most significant set bit is more elaborate, but can also be done $O(1)$ time [**?**]. An $n \times m$ bit matrix may be transposed in $O(w \log w)$ time if $n \leq w$ and $m \leq w$ [97].

**Trees.** In this paper all trees are rooted, ordered, and have labels on the nodes. The number of nodes in a tree $T$ is $t$. We denote by $T(v)$ the subtree rooted at $v$ containing all descendants of $v$. The size $|T(v)|$ is the number of nodes in the subtree $T(v)$ including $v$. If $u$ is a node in the subtree $T(v)$ we write $u \in T(v)$. If $T$ is a binary tree we denote the left and right child of a node $v$ by *left*$(v)$ and *right*$(v)$.

A heavy path decomposition [90] decomposes $T$ into disjoint paths. Nodes are classified as either heavy or light and the decomposition is defined as follows. The root is light. For each internal node $v$, its heavy child $w$ is the node for which $T(w)$ is of maximum size among the subtrees rooted at children of $v$. The other children of $v$ are light. Edges are also classified as heavy or light. An edge going into a heavy node is heavy and likewise for light nodes. The heavy path decomposition ensures the property that $\frac{1}{2}|T(v)| > |T(u)|$ for any light child $u$ of $v$. This means that there are $O(\log t)$ light edges on any path from the root to a leaf. The heavy path decomposition can be computed in $O(t)$ time and space.

Given a binary tree $T$ rooted at a node $r$, $t > 1$, and a parameter $1 \leq x \leq t$, we may partition $T$ into at most $t/x$ clusters such that for a fixed constant $c$, the size of any cluster is at most $cx$ [3, 5] (see also [1] for a full proof). Two clusters overlap in at most

[1] The trick is to replace every character with the rank of its first occurrence in the pattern. To do this in $O(n+m)$ expected time we use hashing and to get $O(n \log m + m)$ time the ranks are stored in a binary search tree.

one node, and a node is called a boundary node if it is part of more than one cluster. Any cluster has at most two boundary nodes, and a boundary node is either a leaf or the root in the subtree that is the cluster. The sum of nodes in all clusters is $O(t)$. The tree obtained by repeatedly contracting edges between two nodes if one of them is not a boundary node is called the macro tree. In other words, the macro tree is the tree consisting only of boundary nodes. A cluster partition can be found in $O(t)$ time.

The answer to a level ancestor query $\text{LA}(v, d)$ on $T$ is the ancestor of $v$ with depth $d$. A linear space data structure that answers an LA query in $O(1)$ time can be computed for $T$ in $O(t)$ time [35] (see also [2, 16, 17]).

**Straight Line Programs.** A Straight Line Program $\mathcal{S}$ is a context-free grammar in Chomsky normal form with $n$ production rules that produce a single string $S$ of length $N$. We represent the SLP as a rooted, ordered, and node-labelled directed acyclic graph (DAG) with outdegree 2 and we will refer to production rules as nodes in the DAG. A depth-first left-to-right traversal starting from a node $v$ in the DAG produces the string $S(v)$ of length $|S(v)|$. The tree that emerges from the traversal we call the derivation tree. We denote the left and right children of $v$ for *left*$(v)$ and *right*$(v)$, respectively. Furthermore, the height of the SLP is the length of the longest path going from the root to a terminal node and is denoted by $h$.

We may access a character $S[i]$ in $O(h)$ time by storing $|S(v)|$ for each node $v$ in the SLP, and simulate a top-down search of the derivation tree. Doing so yields a unique path from the root of $\mathcal{S}$ to the terminal node labelled $S[i]$. There is also a linear space data structure that supports random access in SLPs in $O(\log N)$ time [21]. A key technique used in this data structure is the extension of the heavy path decomposition of trees to SLPs which we will also use in our data structure. For each node $v \in \mathcal{S}$, we select the child of $v$ that derives the longest string to be a heavy node. The other child is light. Heavy and light edges are defined as in the decomposition of trees. Whereas applying this technique to a tree results in a decomposition into disjoint paths, it will result in a decomposition into disjoint trees when applied to an SLP (see Figure 4.1). We denote this set of trees by the heavy forest $\mathcal{H}$ of the SLP. This decomposition ensures that the number of light edges on any path from the root to a terminal node is $O(\log N)$. Hence, on any path from the root of the SLP to a terminal node, we visit at most $\log N$ trees from $\mathcal{H}$. When accessing a character using the data structure of [21] we may also report the entry and exit nodes for each tree visited on the unique root-to-terminal path that emerges from the query.

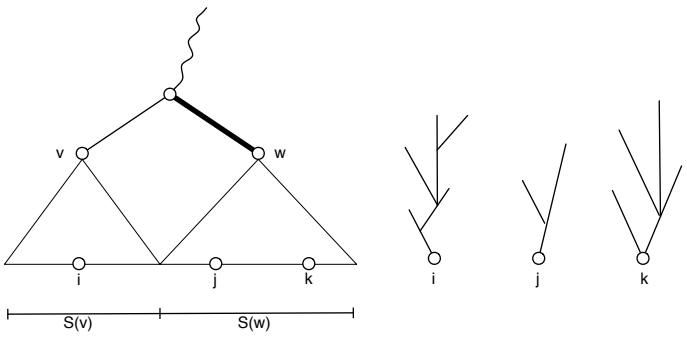



**Figure 4.1:** An example of the heavy path decomposition of SLPs. *(Left)* A node in $\mathcal{S}$ with children $v$ and $w$ where the edge to $w$ is selected as heavy because $S(w) \geq S(v)$. *(Right)* The heavy forest obtained by removing light edges from $\mathcal{S}$. The trees are rooted in terminals of $\mathcal{S}$ and can therefore be seen as growing upwards.

## 4.3 Packed Tree Color Problems

In a colored tree, each node is colored by zero or more colors from the set $\{1, \ldots, \sigma\}$. A packed colored tree is a colored tree where the colors of each node $v$ are given as a bit string $C(v)$ where $C(v)[c] = 1$ iff $v$ is colored $c$. In this section we consider the *packed tree color problem* which is to preprocess a packed colored tree $T$ to support first and last colored ancestor queries. The answer to a first colored ancestor query FIRSTCOLOR$(v, c)$ is the lowest ancestor of $v$ with color $c$, and the answer to a last colored ancestor query LASTCOLOR$(u, v, c)$ is the highest node with color $c$ on the path from $u$ to $v$, where we always assume that $u$ is an ancestor of $v$. Throughout this section we will use the following notation to distinguish results. If a data structure requires $p(t)$ time to build, uses $s(t)$ space, and supports FIRSTCOLOR and LASTCOLOR queries in $q(t)$ time, then the the triple $\langle p(t), s(t), q(t)\rangle$ refers to the solution.

Solutions to the tree color problem for trees that are not packed may be applied to packed trees. All known solutions focus entirely on supporting FIRSTCOLOR queries [4,35,38,83]. A simple solution that supports FIRSTCOLOR queries in $O(1)$ time is to store the answer for every color in every node. This yields a $\langle O(t\sigma), O(t\sigma), O(1)\rangle$ solution. The currently best known trade-off for the tree color problem is $\langle O(t+D), O(t+D), O(\log w)\rangle$ [83], where $D = \sum_{v \in T} \sum_{i=1}^{\sigma} C(v)[i]$ is the accumulated number of colors used.

Our motivation for revisiting this problem is twofold. First we have that $D = O(t\sigma)$ in our application and we are striving for a space bound that is in $o(t\sigma)$. Second we want to support LASTCOLOR queries.

In this section we present three solutions to the packed tree coloring problem and combine them to a data structure with a new and desireable time-space trade-off.

### 4.3.1 A $\langle O(t\sigma), O(t\sigma), O(1)\rangle$ Solution

We store the result of a FIRSTCOLOR$(v, c)$ query for every node and color. For each color, let the induced $c$-colored subtree be the tree obtained by deleting all nodes that are not colored by color $c$ except the root. Build a levelled ancestor data structure for each induced colored subtree.

The result of a FIRSTCOLOR query is precomputed. A LASTCOLOR$(u, v, c)$ query is answered as follows. If FIRSTCOLOR$(v, c) =$ FIRSTCOLOR$(u, c)$ then there is not a node with color $c$ on the path from $u$ to $v$. If FIRSTCOLOR$(v, c) \neq$ FIRSTCOLOR$(u, c)$ then let $v'$ and $u'$ be the nodes corresponding to FIRSTCOLOR$(v, c)$ and FIRSTCOLOR$(u, c)$ in the induced $c$-colored subtree. The answer to LASTCOLOR$(u, v, c)$ is then the answer to LA$(v', depth(u') - 1)$ in the induced $c$-colored subtree.

The results of FIRSTCOLOR queries can be found and stored using $O(t\sigma)$ time and space. The induced colored subtrees can be computed in $O(t\sigma)$ time and use $O(D) = O(t\sigma)$ space. A FIRSTCOLOR query clearly takes $O(1)$ time. For a LASTCOLOR query, we perform two FIRSTCOLOR queries and one LA query, each of which takes constant time.

**Lemma 4.1** *The packed tree color problem can be solved using $O(t\sigma)$ preprocessing time and space, and $O(1)$ query time.*

### 4.3.2 A $\langle O(t + \frac{t\sigma}{w}), O(t + \frac{t\sigma}{w}), O(\log t)\rangle$ Solution

We fix a heavy path decomposition of $T$. For each path $p$ in the heavy path decomposition of $T$ we build a balanced binary tree $T_p$ having the nodes of $p$ as leaves. For each node $v$ in $T_p$ we store a summary $B(v)$ of the colors of its children. For each heavy path $p = v_1, v_2, \ldots, v_k$, where $v_1$ is the highest node on the path, we store a summary $P(v_i)$ of colors on the path prefix $v_1 \ldots v_i$ for every $v_i$ on $p$.

For answering a FIRSTCOLOR$(v, c)$ query, let $p = v_1, v_2, \ldots, v_k$ be the heavy path containing $v$ and let $v_i = v$ for some $1 \leq i \leq k$. If $P(v_i)[c] = 1$ we find the lowest

ancestor $x$ of $v_i$ in $T_p$ for which $B(\mathit{left}(x))[c] = \mathtt{1}$ and $v_i \notin T_p(\mathit{left}(x))$. The answer to the query is then the rightmost leaf in $T_p(\mathit{left}(v_a))$ with color $c$. If $P(v_i)[c] = \mathtt{0}$ we repeat the procedure with $v_i = parent(v_1)$, i.e., we jump to the previous heavy path, until we find the first colored ancestor or we reach the root of $T$.

A LASTCOLOR$(u, v, c)$ query is handled in a similar way. We first find the highest light node $w$ on the path from $u$ to $v$ for which $P(parent(w))[c] = \mathtt{1}$. Let $p$ be the heavy path containing $parent(w)$. Now there are three cases. If $u$ is not on $p$, the answer to the query is the leftmost leaf in $T_p$ that has color $c$. If $p$ contains $u$, the answer is the leftmost leaf with color $c$ to the right of $u$ in $T_p$, if such a node exists. If it does not exist, we repeat the first step for the second highest light node $w'$ between $u$ and $v$ for which $P(parent(w'))[c] = \mathtt{1}$.

The heavy path decomposition of $T$ can be found and stored in $O(t)$ time and space. Since the paths of the heavy path decomposition are disjoint, the total number of leaves in the binary summary trees is $t$, so the total number of nodes in the trees is $O(t)$. We store $O(t)$ summary bit vectors of size $O(\frac{\sigma}{w})$ using a total of $O(\frac{t\sigma}{w})$ space. We use $O(\frac{t\sigma}{w})$ bitwise OR operations to create the summaries in a bottom up fashion. In total, preprocessing time and space usage is $O(t + \frac{t\sigma}{w})$.

For both queries we visit at most $\log t$ heavy paths. When the path with the answer has been found we walk up the binary tree and then down again. Since the tree is balanced and has at most $t$ leaves, this takes $O(\log t)$ time. For LASTCOLOR queries we do this at most twice. The query time for FIRSTCOLOR and LASTCOLOR queries is therefore $O(\log t)$ time.

**Lemma 4.2** *The packed tree color problem can be solved using $O(t + \frac{t\sigma}{w})$ preprocessing time and space, and $O(\log t)$ query time.*

### 4.3.3 A $\langle O(t + \frac{t\sigma \log w}{w} + \frac{t^2}{w}), O(t + \frac{t\sigma}{w} + \frac{t^2}{w}), O(\frac{t}{w})\rangle$ Solution

Let $v_1, \ldots, v_t$ be the nodes of $T$ in pre-order. We will represent $T$ as a $\sigma \times t$ bit matrix $M$. Let $c$ be a color from the set of colors $\{1, \ldots, \sigma\}$. In row $c$ of $M$ we store a bit string where bit $i$ is 1 iff $v_i$ has color $c$. For each node $v_i$ we also store a bit string $A(v_i)$ where bit $j$ is 1 iff $v_j$ is an ancestor of $v_i$.

We construct this data structure from a packed colored tree as follows. Assume that the bit strings representing the node colorings form a $t \times \sigma$ matrix where row $i$ is the colorings of node $v_i$. We transpose this matrix to get $M$. To do this we partition the matrix into a $\frac{t}{w} \times \frac{\sigma}{w}$ matrix (assume w.l.o.g. that $w$ divides $t$ and $\sigma$), transpose each $w \times w$ submatrix as described in [97], and transpose the $\frac{t}{w} \times \frac{\sigma}{w}$ matrix to get $M$. To compute the ancestor bit strings first set $A(root(T)) = [\mathtt{0}]^t$. For all other nodes $v_i$, where $v_j$ is the parent of $v_i$, set $A(v_i) = A(v_j) \vee 2^j$.

We answer a FIRSTCOLOR$(v, c)$ as follows. Let $R = M[c] \wedge A(v)$. Now $R$ is a bit string representing the set of ancestors of $v$ with color $c$. Since the nodes have pre-order indices, the answer to the query is $v_i$, where $i$ is the index of the least significant set bit in $R$.

To answer a LASTCOLOR$(v, u, c)$ query we start by computing $R$ the same way as above. We then set the first $i - 1$ bits of $R$ to 0, where $i$ is the index of $u$. The answer to the query is the most significant set bit of $R$.

The $\sigma \times t$ bit matrix $M$ can be packed in words and therefore uses $O(\frac{t\sigma}{w})$ space. Similarly, the ancestor bit strings use $O(\frac{t^2}{w})$ space. Transposing a $w \times w$ matrix takes $O(w \log w)$ time, and since there are $\frac{t\sigma}{w^2}$ submatrices of this size in the color bit matrix, the total time spent for all submatrices is $O(\frac{t\sigma \log w}{w})$. Transposing the $\frac{t}{w} \times \frac{\sigma}{w}$ matrix takes $O(\frac{t\sigma}{w})$ time. Computing the ancestor bit strings takes $O(\frac{t^2}{w})$ time.

The size of $R$ is $O(\frac{t}{w})$, so finding the first non-zero word takes $O(\frac{t}{w})$ time. Determining the least or most significant set bit of a word is done in $O(1)$ time. Thus, the query time for both a FIRSTCOLOR and a LASTCOLOR query is $O(\frac{t}{w})$.

**Lemma 4.3** *The packed tree color problem can be solved using* $O(t + \frac{t\sigma \log w}{w} + \frac{t^2}{w})$ *preprocessing time,* $O(t + \frac{t\sigma}{w} + \frac{t^2}{w})$ *space, and* $O(\frac{t}{w})$ *query time.*

#### 4.3.4 Combining the Solutions

We now show how to combine the previously described solutions to get $\langle O(t + \frac{n\sigma}{w}), O(t + \frac{n\sigma}{w}), O(\log w)\rangle$ and $\langle O(t + \frac{t\sigma \log w}{w}), O(t + \frac{t\sigma}{w}), O(1)\rangle$ trade-offs. This is achieved by doing a cluster partioning of the tree.

First we convert $T$ to a binary tree $T'$. Then we partition $T'$ into $O(\frac{t}{w})$ clusters, i.e., each cluster has size $O(w)$. For each cluster $C$, where one boundary node is a leaf in the cluster and the other is the root of the cluster, we make a summary of the colors of the nodes on the path from the root to the leaf. The summary is stored in the macro tree node that corresponds to the leaf boundary node of $C$. Apply the $\langle O(t\sigma), O(t\sigma), O(1)\rangle$ solution to the macro tree, and apply either the $\langle O(t + \frac{t\sigma}{w}), O(t + \frac{t\sigma}{w}), O(\log t)\rangle$ solution or the $\langle O(t + \frac{t\sigma \log w}{w} + \frac{t^2}{w}), O(t + \frac{t\sigma}{w} + \frac{t^2}{w}), O(\frac{t}{w})\rangle$ solution to each cluster using the original colors.

Here is how we answer a FIRSTCOLOR$(v, c)$ query. Let $C_v$ be the cluster containing $v$. First we ask for FIRSTCOLOR$(v, c)$ in $C_v$. If the answer is a node in $C_v$, we are done. If it is undefined, we find the node $r$ in the macro tree corresponding to the root of $C_v$. We check if $r$ has color $c$ in the macro tree and otherwise ask for $w =$ FIRSTCOLOR$(r, c)$ in the macro tree. In the cluster $C_w$ having $w$ as a leaf boundary node we then check if $w$ has color $c$ and otherwise ask for FIRSTCOLOR$(w, c)$ in $C_w$.

We answer a LASTCOLOR$(u, v, c)$ query as follows. Assume that $u \neq v$ and let $C_u$ and $C_v$ be the clusters containing $u$ and $v$. If $C_u = C_v$ then the answer is LASTCOLOR$(u, v, c)$ in the cluster containing $u$ and $v$. If $C_u \neq C_v$, let $w$ be the leaf boundary node of $C_u$ where $v \in T(w)$. We now proceed in three steps. First, we ask for LASTCOLOR$(u, w, c)$ in $C_u$. If the query returns a node, this is also the answer to the LASTCOLOR$(u, v, c)$ query. If the answer in the first step is undefined we ask for $z =$ LASTCOLOR$(w, root(C_v), c)$ in the macro tree to locate the highest cluster with a node with color $c$ between $u$ and $v$. The answer to the query is then LASTCOLOR$(root(C_z), z, c)$ on $C_z$. If the first two steps fail, the answer to a query is LASTCOLOR$(root(C_v), v, c)$.

The cluster partition can be computed in linear time, and the cluster path summaries are computed in $O(\frac{t\sigma}{w})$ time. Since the macro tree has $O(\frac{t}{w})$ nodes the preprocessing time and space to apply the $\langle O(t\sigma), O(t\sigma), O(1)\rangle$ solution becomes $O(\frac{t\sigma}{w})$. To answer a query we perform a constant number of FIRSTCOLOR and LASTCOLOR queries on the macro tree and clusters. Therefore the total time to perform queries on the macro tree is $O(1)$ time. To get $(i)$ we apply the $\langle O(t + \frac{t\sigma}{w}), O(t + \frac{t\sigma}{w}), O(\log t)\rangle$ solution to clusters. Since a cluster has size $O(w)$ we use a total of $O(\log w)$ time performing queries on clusters. To get $(ii)$ we apply the $\langle O(t + \frac{t\sigma \log w}{w} + \frac{t^2}{w}), O(t + \frac{t\sigma}{w} + \frac{t^2}{w}), O(\frac{t}{w})\rangle$ solution to clusters. Again, since clusters have size $O(w)$ we use a total of $O(1)$ time performing queries on clusters. Preprocessing time and space for the cluster data structures follow because the sum of the sizes of clusters is $O(t)$.

**Theorem 4.2** *The packed tree color problem can be solved using* $O(t + \frac{t\sigma}{w})$ *space,*

*(i)* $O(t + \frac{t\sigma}{w})$ *preprocessing time, and* $O(\log w)$ *query time, or*

*(ii)* $O(t + \frac{t\sigma}{w} \log w)$ *preprocessing time, and* $O(1)$ *query time.*

### 4.4 Labelled Successor Data Structure for SLPs

The answer to a labelled successor LS$(i, c)$ query on a string $S$ is the index of the first occurrence of the character $c$ after position $i$ in $S$. More formally, the answer to LS$(i, c)$ is an index $j$ such that $S[j] = c$, $j > i$, and $S[k] \neq c$ for $k = i + 1, \ldots, j - 1$.

In this section we present a data structure that supports $\text{LS}(i, c)$ queries on an SLP. This is the first data structure dedicated to solving this problem on SLPs. Alternatively, we may build the random access data structure of [21] and then answer an $\text{LS}(i, c)$ query by doing a random access query for position $i$ followed by a linear scan to find the first occurrence of $c$. This yields a query time of $O(\log N + j - i)$ while using $O(n)$ space for the data structure.

Our data structure combines the random access data structure of [21] with a new way of navigating the SLP based on the characters of substrings. For the latter we will utilize our result for the packed tree color problem described in the previous section.

The basic idea is to store a bit string for each node $v \in \mathcal{S}$ that summarizes which characters are generated by $S(v)$. We first seach for position $i$ in $S$ and let $p$ be the unique path in $\mathcal{S}$ defining $S[i]$. We then walk up $p$ until reaching a node $u$ where *right*$(u)$ generates a string that contains $c$ and *right*$(u)$ is not on $p$. Then we walk down from *right*$(u)$ using the summaries to locate the leftmost terminal descending from *right*$(u)$ that generates $c$. This algorithm requires $O(n + \frac{n\sigma}{w})$ space and $O(h)$ time to find $\text{LS}(i, c)$.

To speed things up we fix a heavy path decomposition of the SLP to get a heavy forest and build the random access data structure of [21]. Now $p$ is a sequence of entry and exit points in the trees of the heavy forest. When we walk up $p$ we enter a tree in an exit node and have to walk away from the root to the first node whose right child generates a string that contains $c$ before reaching the entry node. This is equivalent to a LASTCOLOR query. When we walk down to find $\text{LS}(i, c)$ we enter a tree and have to walk towards the root to find either the first ancestor whose left child generates a string that contains $c$ or the highest ancestor whose right child generates $c$. This is equivalent to a FIRSTCOLOR and a LASTCOLOR query, respectively.

In the remainder of this section we give the details of the data structure.

**Theorem 4.3** *There is a data structure supporting labelled successor (and predecessor) queries on a string of size $N$ over an alphabet of size $\sigma$ compressed by an SLP of size $n$ in the word RAM model with word size $w \geq \log N$ using $O(n + \frac{n\sigma}{w})$ space and*

*(i) $O(n + \frac{n\sigma}{w})$ preprocessing time, and $O(\log N \log w)$ query time, or*

*(ii) $O(n + \frac{n\sigma}{w} \log w)$ preprocessing time, and $O(\log N)$ query time.*

**Proof** We first apply the construction of [21], and let $\mathcal{H}$ be the heavy forest obtained from the heavy path decomposition of $\mathcal{S}$. For each node $v$ in $\mathcal{S}$ with children *left*$(v)$ and *right*$(v)$ we store two bit strings $L(v)$ and $R(v)$ summarizing the characters in $S($*left*$(v))$ and $S($*right*$(v))$. Specifically, $L(v)$ summarizes the characters in $S($*left*$(v))$ if the edge from $v$ to *left*$(v)$ is light. If the edge is heavy then $L(v) = [0]^\sigma$. $R(v)$ is defined analogously. These summaries are used for determining where to exit a heavy tree when searching for some character. When following a heavy edge, say from $v$ to *left*$(v)$, we do not exit a tree, so therefore $L(v)$ is set to $[0]^\sigma$. For each tree in $\mathcal{H}$ we build two data structures for the packed tree color problem. One where the $L$ bit strings serve as colors and one where the $R$ bit strings serve as colors.

We answer an $\text{LS}(i, c)$ query as follows. First we access the character $S[i]$ using the random access data structure. We now have the entry and exit points of the heavy trees in $\mathcal{H}$ on the unique path $p$ describing $S[i]$. Let $T_1, \ldots, T_k \in \mathcal{H}$ be a sequence of trees on $p$ in the order they are visited when starting from the root and ending in the terminal generating $S[i]$, and let $(v_1, u_1), \ldots, (v_k, u_k)$ be the entry and exit nodes for each tree in the sequence. Using the packed tree color data structure for the $R$ colors, we repeat $\text{LASTCOLOR}(u_i, v_i, c)$ for $i = k$ down to some $j$ until $\text{LASTCOLOR}(u_j, v_j, c)$ is not undefined. Let $w =$ *right*$(\text{LASTCOLOR}(u_j, v_j, c))$. We now search for the first occurrence of $c$ in $S(w)$. Let $T_i$ be the tree in $\mathcal{H}$ that contains the node $w$, then the search proceeds in three steps. First, we ask for $v = \text{FIRSTCOLOR}(w, c)$ in $T_i$ in the data structure for $L$

colors and restart the search from *left*$(v)$. If the query FIRSTCOLOR$(w, c)$ is undefined we continue to the next step. In the second step we check if $root(T_i)$ generates $c$. If it does, we now have a unique set of entry and exit nodes in the trees of $\mathcal{H}$ that constitutes a path to a terminal that generates the first $c$ after position $i$. The answer to the LS$(i, c)$ query is the index of this $c$ which we retrieve using the random access data structure. Finally, if $root(T_i)$ does not generate $c$ we ask for $v =$ LASTCOLOR$(w, root(T_i), c)$ in $T_i$ in the data structure for $R$ colors, and restart the search from *right*$(v)$.

The data structure uses $O(n + \frac{n\sigma}{w})$ space because the random access data structure uses linear space and the bit strings $L$ and $R$ use $O(\frac{n\sigma}{w})$ space. The random access data structure, including the heavy path decomposition, takes $O(n)$ time to compute and the $L$ and $R$ values are computed using $O(\frac{n\sigma}{w})$ OR operations in a bottom up fashion. Therefore, this part of the data structure is computed in $O(n + \frac{n\sigma}{w})$ time.

To get Theorem 4.3 (i) we use the packed tree color data structure of Theorem 4.2 (i) for the trees in $\mathcal{H}$ and likewise for (ii). Since the trees are disjoint, the preprocessing time and space becomes as in the Theorem 4.3.

For the query, we first do one random access query that takes $O(\log N)$ time, then we perform at most $\log N$ LASTCOLOR queries walking up the SLP and at most $2 \log N$ FIRSTCOLOR and LASTCOLOR queries locating the labelled successor. Finally, retrieving the index also takes $O(\log N)$ time using the random access data structure.

## 4.5 Subsequence Matching

We will now use the labelled successor data structure to obtain a subsequence matching algorithm for SLPs. Our algorithm is based on the folklore algorithm for subsequence matching which works as follows (see also [34, 78]). First we find the minimal prefix $S[1..j]$ that contains $P$ as a subsequence. This is done by reading $S$ left to right while searching for the characters of $P$ one at a time. We then find the minimal suffix $S[i..j]$ of the prefix $S[1..j]$ that contains $P$. Similarly, this is done by scanning the prefix right to left. Now $S[i..j]$ is the first minimal occurrence of $P$. To find the next minimal occurrence we repeat this process for the suffix $S[i + 1..N]$. It can be shown that this algorithm finds all minimal occurrences of $P$ in $O(Nm)$ time.

By using our labelled successor data structure described in the previous section we speed up the procedure of finding some specific character of $P$. Assume we have matched $P[1..k]$ to $S[1..j]$ such that $P[k] = S[j]$. Instead of doing a linear scan of $S[j + 1..N]$ to find $P[k + 1]$ we ask for the next occurrence of $P[k + 1]$ using LS$(j, P[k + 1])$.

For each occurrence of $P$ we perform $O(m)$ labelled successor (and labelled predecessor) queries, and we also have to construct the data structures to support these. By applying the results of Theorem 4.3 we get Theorem 4.1.

# CHAPTER 5

# OPTIMAL TIME RANDOM ACCESS TO GRAMMAR-COMPRESSED STRINGS IN SMALL SPACE

Patrick Hagge Cording

Technical University of Denmark, DTU Compute

**Abstract**

The random access problem for compressed strings is to build a data structure that efficiently supports accessing the character in position $i$ of a string given in compressed form. Given a grammar of size $n$ compressing a string of size $N$, we present a data structure using $O(n\tau \log_\tau \frac{N}{n} \log N)$ bits of space that supports accessing position $i$ in $O(\log_\tau N)$ time for $\tau \leq \log^{O(1)} N$. The query time is optimal for polynomially compressible strings, i.e., when $n = \Theta(N^{1-\varepsilon})$.

## 5.1 Introduction

Let $S$ be a string of length $N$ compressed into a context-free grammar $\mathcal{S}$ of size $n$. The *random access problem* on grammar-compressed strings is to build a data structure that efficiently supports accessing $S[i]$. In this paper we present a new random access data structure for grammar-compressed strings with a new trade-off between the time and space complexity. The data structure also facilitates decompression of a substring $S[i, j]$.

Assume w.l.o.g. that $\mathcal{S}$ is a Straight Line Program (SLP), i.e., a context-free grammar where a rule has either two variables or a terminal on its right-hand side. We regard an SLP as a rooted directed acyclic graph (DAG) with outdegree 2 and we refer to rules as nodes and terminals as leafs when appropriate.

If we store the size of the string generated by each node in the SLP we may access the $i$-th character in $O(h)$ time by doing a top-down search. Here, $h$ is the height of the grammar and the data structure requires $O(n)$ space. The concept of balance generalizes from binary trees to SLPs. A balanced SLP has height $h = O(\log N)$. We may apply the algorithm by Rytter [88] or Charikar et al. [26] to balance the SLP and then access $S[i]$ in $O(\log N)$ time. The culprit is that the number of nodes in the SLP resulting from applying either algorithm is $O(n \log \frac{N}{n})$. A major breakthrough was done when Bille et al. [21] published a data structure using $O(n)$ space and supporting access in $O(\log N)$ time and decompression of the substring $S[i, j]$ in $O(\log N + j - i)$ time. Their work introduced many new techniques and did not rely on the algorithms by Rytter and Charikar et al. Later, Verbin and Yu [105] proved that if $n = \Omega(N^{1-\varepsilon})$ any data

structure of size $O(n \log N)$ must use $\Omega(\log n / \log \log n)$ time to access $S[i]$. For other compression ratios, they also show a lower bound of $\Omega(n^{1/2-\varepsilon})$ query time for any $\varepsilon > 0$ and any data structure of polynomial size. In the wake of this, Belazzougui, Puglisi, and Tabei [15] presented a data structure where $O(\log N / \log \log N)$ access time is achieved using $O(n \log_2 \frac{N}{n} \log^{1+\varepsilon} N)$ bits of space which is matching the lower bound by Verbin and Yu for polynomially compressible strings.

Note that for the most recent result by Belazzougui et al., the space bound is in bits and not words of memory. In the remainder of this paper all space bounds will be in bits to comply with their work. Furthermore, all results assume the RAM model of computation with a word size $w \geq \log N$. We present a data structure using $O(n\tau \log_\tau \frac{N}{n} \log N)$ bits of space that supports access to $S[i]$ in $O(\log_\tau N)$ time and decompression of $S[i, j]$ in $O(\log_\tau N + j - i)$ time. The time bound optimal for polynomially compressible strings, i.e., $n = \Theta(N^{1-\varepsilon})$, when we set $\tau = \log^\varepsilon N$.

The key idea behind our data structure is to increase the out-degree of a balanced SLP, exactly as done in the work by Belazzougui et al. This decreases the height of the grammar, and we use the basic top-down search to perform access queries. To get fast navigation in each node we use fusion trees. Pivotal to our improvement over Belazzougui et al. in terms of space usage is that we exploit some properties resulting from applying Rytter's [88] algorithm to balance the SLP.

## 5.2 Preliminaries

A string $S = S[1, |S|]$ is a sequence of $|S| = N$ characters. $S[j]$ is a single character.

An SLP $\mathcal{S}$ is a context-free grammar in Chomsky normal form that we represent as a node-labeled, directed acyclic graph. Each leaf in $\mathcal{S}$ is labelled with a character and corresponds to a terminal grammar production rule. Each internal node in $\mathcal{S}$ is labeled with a non-terminal rule from the grammar. We call a node that is reachable from $v$ in one step a child of $v$.

$\mathcal{S}(v)$ is the subgraph rooted in $v$ producing the string $S(v)$. $S(v)$ is produced by a depth-first left-to-right traversal from $v$ and consists of the characters on the leafs in the order they are visited. The tree that emerges from this we call the parse tree for $S(v)$. Let $root(\mathcal{S})$ denote the start rule of $\mathcal{S}$, and *left*$(v)$ and *right*$(v)$ denote the left and right child of an internal node $v \in \mathcal{S}$, respectively. The height of a node $v$ is the length of the longest path from $v$ to a leaf reachable from $v$. We denote this by *height*$(v)$. We sometimes use $\mathcal{S}$ as shorthand for $root(\mathcal{S})$ when used as parameter in any of the definitions. As a shorthand for $|S(v)|$ we use $|v|$.

An SLP is AVL-balanced if $|$*height*$($*left*$(v)) -$ *height*$($*right*$(v))| \leq 1$ for all $v \in \mathcal{S}$. In [88], Rytter presents an approximation algorithm for the smallest grammar problem. The SLP produced by his algorithm is AVL-balanced and has $O(z \log \frac{N}{z})$ nodes, where $z$ is the number of factors in the LZ77 [109] parse of $S$. Let $n'$ be the size of the smallest grammar. We will use his algorithm to balance an SLP, and since $z \leq n' \leq n$ for any given SLP of size $n$, the number of nodes is $O(n \log \frac{N}{n})$ after applying it.

In addition, to accomplish our goal, we employ Fredman and Willard's fusion tree [40], where predecessor queries can be performed in $O(\log t / \log w)$ time for $t$ $w$-bit integers.

## 5.3 Access data structure for balanced grammars

In this section, the setting is that we are given a balanced grammar producing one string only, and we will show how to construct a random access data structure with fast query time. The general idea is to increase the outdegree of all non-terminals of the grammar.

For every non-terminal $v$ we generate a new right-hand side that contains at most $\log^\varepsilon N$ variables by iteratively expanding the right-hand side of $v$ down $\varepsilon \log \log N$ levels

(or less if we reach a terminal). We then store in a fusion tree the prefix-sum of the lengths of the strings generated by each variable in the right-hand side. More precisely, assuming that the expanded right-hand side of $v$ is $v_1 v_2 \dots v_t$ with $t \leq 2^{\varepsilon \log\log n}$, the prefix sums are $s_1, s_2 \dots s_t$, where $s_i = \sum_{j=1}^{i} |v_j|$. The height of the resulting grammar is $O(\log N / \log\log N)$ since the original grammar is balanced and thus have height $O(\log N)$. Every fusion tree uses $O(\log^{1+\varepsilon} N)$ bits of space and the expanded grammar has $O(n)$ nodes, so we use $O(n \log^{1+\varepsilon} N)$ bits of space.

To access a specific character $S[i]$ we do a top-down traversel of the grammar and use the prefix-sums for navigation. Suppose we have reached a rule $v$ producing the string $S[i', j']$ (where $i' \leq i \leq j'$). To find the child of $v$ to continue the search from, we ask for the predecessor of $i - i'$ among the prefix-sums stored in $v$. The values for $i'$ and $j'$ are not unique to a node, so we must keep a counter indicating what $i'$ is at every step. The counter is initially zero and when the search exits a node $v$ in child $v_k$, we add $s_{k-1}$ to the counter (or 0 if $k = 1$).

The fusion tree allows predecessor searches on a set of $t$ integers of $w$ bits in $O(\log t / \log w)$. Since in our case, we have $t = \log^{\varepsilon} N$ and $w \geq \log N$, the query time is constant. The traversal therefore takes $O(\log N / \log\log N)$ time.

This data structure also supports fast decompression of the string $S[i, j]$. When accessing $S[i]$ we obtain a path from the root of the grammar to the terminal $v_i$ generating $S[i]$. Let $u$ be the lowest common ancestor on the paths from the root to $v_i$ and $v_j$ obtained from accessing $S[i]$ and $S[j]$. To decompress $S[i, j]$ we expand the part of the parse tree that corresponds to all nodes to the right of nodes on the path from $v_i$ to $u$ and to the left on the path from $u$ to $v_j$. The part of the parse tree that we expand has $m = j - i$ leaves, and therefore we visit $O(m)$ nodes in the parse tree in the process, so decompression of $S[i, j]$ takes $O(\log N / \log\log N + m)$ time.

**Theorem 5.1** *Given a* balanced *grammar of size $n$ generating a string $S$ of length $N$, we can build a data structure of $O(n \log^{1+\varepsilon} N)$ bits (for any constant $\varepsilon$), that supports random access to any character of $S$ in $O(\log N / \log\log N)$ time, and access to $m$ consecutive characters in $O(\log N / \log\log N + m / \log_\sigma N)$ time.*

Our upper bound for access matches the lower bound of Verbin and Yu [105] who have shown that for "not-so-compressible" strings — those that have a grammar of length $n$ such that $N \leq n^{1+\varepsilon}$ for some constant $\varepsilon$ — the query time cannot be better that $O(\log n / \log\log n) = O(\log N / \log\log N)$ if the used space is not more than $O(n \log^c n)$ for some constant $c$.

## 5.4 Fast queries for unbalanced grammars

For unbalanced grammars we may use the results of Charikar et al. [26] or Rytter [88] to generate a balanced grammar that produces the same string as the unbalanced, but is of size larger by a factor $O(\log(N/n))$. This immediately gives a data structure that uses $O(n \log(N/n) \log^{1+\varepsilon} N)$ bits (for any constant $\varepsilon$) supporting access to $m$ consecutive characters in $O(\log N / \log\log N + m / \log_\sigma N)$ time, and a data structure using $O(n\sigma \log(N/n) \log^{1+\varepsilon} N)$ bits of space that supports rank and select in $O(\log N / \log\log N)$ time. In this section we show the following theorem.

**Theorem 5.2** *Given any grammar of size $n$ generating a string $S$ of length $N$, we can build a data structure of $O(n\tau \log_\tau (N/n) \log N)$ bits that supports random access to any character of $S$ in $O(\log_\tau N)$ time, and access to $m$ consecutive characters in $O(\log_\tau N + m)$ time.*

Our data structure relies on Rytter's algorithm to generate a balanced grammar and the data structure described in the previous section. Rytter's algorithm generates the

LZ77 parse of the string and builds an AVL-balanced grammar by processing the factors of the parse left to right. It maintains an AVL-balanced grammar of the string generated by all factors preceeding some factor $f_k$. Then it builds an AVL-balanced grammar of the string generated by $f_k$ by selecting a logarithmic number of non-terminals and joining them. By carefully choosing the order in which these are joined, this step can be done in time linear in the height of the tallest grammar, which is bounded by $O(\log N)$. The grammar for $f_k$ is then joined with the existing grammar. The algorithm produces a grammar of size $O(z \log \frac{N}{z})$ where $z$ is the size of the greedy LZ77 parse. Rytter then shows that $z \leq n'$ where $n'$ is the size of the smallest grammar generating $S$, so if the input is a grammar of size $n$, the resulting balanced grammar has size $O(n \log \frac{N}{n})$. One of the key observations in the analysis of Rytter's algorithm is that when joining two AVL-balanced grammars we add only a number of new rules proportional to the difference in their heights, as stated below.

**Lemma 5.1** *([88]) Given two AVL-balanced grammars $\mathcal{S}_1$ and $\mathcal{S}_2$, we can create an AVL-balanced grammar $\mathcal{S}$ generating the concatenation of the strings produced by $\mathcal{S}_1$ and $\mathcal{S}_2$ by adding $O(|\mathit{height}(\mathcal{S}_1) - \mathit{height}(\mathcal{S}_2)|)$ new non-terminals. When doing so, we add a constant number of non-terminals with height $h$ for $\min(height(\mathcal{S}_1), height(\mathcal{S}_2)) \leq h \leq \max(height(\mathcal{S}_1), height(\mathcal{S}_2))$.*

A closer look at the algorithm will reveal the following useful property.

**Lemma 5.2** *Let $\mathcal{S}$ be an arbitrary grammar of size $n$ and $\mathcal{S}'$ an AVL-balanced grammar generated by Rytter's algorithm producing the same string as $\mathcal{S}$. The number of non-terminals with height $h$ in $\mathcal{S}'$ is $O(n)$ for any $h$.*

**Proof** The key observation needed for this proof is Lemma 5.1. First, recall that Rytter's algorithm processes the LZ77 factorization $f_1, \ldots, f_z$ of $S$ from left to right, such that, before processing $f_i$, we have an AVL-balanced grammar $\mathcal{G}_{i-1}$ for the string produced by $f_1, \ldots, f_{i-1}$. We refer to the processing of $f_i$ as round $i$ of the algorithm.

We will prove that the number of rules added with height $h$ in each round is constant. Rytter's algorithm picks rules $v_1, \ldots, v_j$, $j = O(\log N)$, from $\mathcal{G}_{i-1}$ such that the concatenation of the strings produced by these is the string represented by $f_i$. For some $k$, it holds that $\mathit{height}(v_1) \leq \mathit{height}(v_2) \leq \ldots \leq \mathit{height}(v_k)$ and $\mathit{height}(v_k) \geq \mathit{height}(v_{k+1}) \geq \ldots \geq \mathit{height}(v_j)$. In addition, it holds that if $\mathit{height}(v_l) = \mathit{height}(v_{l+1})$ then $\mathit{height}(v_{l+1}) \neq \mathit{height}(v_{l+2})$. The algorithm joins the grammars from lowest to tallest, i.e., we first join the grammar rooted in $v_1$ with the grammar rooted in $v_2$ up to $v_k$ and then $v_j$ with $v_{j-1}$ down to $v_k$.

We will prove by induction that joining the grammars from $v_1$ to $v_k$ only adds a constant number of rules with height $h$ for any $h$. For the base case we have that $\mathit{height}(v_1) \leq \mathit{height}(v_2)$. If $\mathit{height}(v_1) < \mathit{height}(v_2)$ we know from Lemma 5.1 that the number of rules added at any depth is $O(1)$. If $\mathit{height}(v_1) = \mathit{height}(v_2)$ the two are joined by adding at new rule with two children, and clearly only one new rule is added.

For the inductive step, let $\mathcal{L}$ be the result of joining $v_1, \ldots, v_{l-1}$. It follows from the aforementioned properties that $\mathit{height}(\mathcal{L}) \leq \mathit{height}(v_l) + 1$. If $\mathit{height}(\mathcal{L}) < \mathit{height}(v_l)$ then we will add a constant number of rules with height $h$ for $\mathit{height}(\mathcal{L}) \leq h \leq \mathit{height}(v_l)$. No new rules were previously added to $v_l$ in this round and the induction hypothesis says that only $O(1)$ rules with height $1 \leq h' \leq \mathit{height}(\mathcal{L})$ were added to $\mathcal{L}$, so only $O(1)$ rules are added at any height in the resulting grammar. If $\mathit{height}(\mathcal{L}) \geq \mathit{height}(v_l)$ then we add one new node as a root.

We know from Lemma 5.1 that when joining $\mathcal{G}_{i-1}$ with the grammar obtained from joining the grammars rooted in $v_1, \ldots, v_j$ we add only a constant number of new rules at each height. Since there are $z \leq n$ factors we conclude that the number of rules added at each depth of the grammar produced by Rytter's algorithm is $O(n)$.

Suppose we are given a grammar balanced by Rytter's algorithm. We want to expand the right-hand sides of rules to be of size $O(\tau)$. Because the grammar is AVL-balanced, we may find a set of rules where each rule has height $h-1$ or $h$ for some $h$ such that $S$ can be partitioned into substrings of size $\frac{1.6^{h-1}}{\sqrt{5}} < \tau \leq 2^h$ produced by these rules. We expand the right-hand sides of these rules and proceed to a higher level. We then have the following corollary.

**Corollary 5.1** *Given a string of size $N$ compressed by a grammar of size $n$. After applying Rytter's algorithm to balance the grammar and expanding the right-hand sides of rules to size $\frac{1.6^{h-1}}{\sqrt{5}} < \tau \leq 2^h$ for some $h$, the resulting grammar has $O(n \log_\tau N)$ rules.*

**Proof** The grammar is balanced by Rytter's algorithm so after it is expanded its height is $O(\log N/(h-1))$ which in terms of $\tau$ is $O(\log_\tau N)$. We select rules with a height difference of at most one in every iteration, so from Lemma 5.2 we know that $O(n)$ rules is selected at each level. Therefore the resulting grammar has $O(n \log_\tau N)$ rules.

We now have seen how to obtain an expanded, balanced grammar with $O(n \log_\tau N)$ rules and we proceed to describe the data structure for access, rank, and select queries.

First we balance the grammar using Rytter's algorithm. Due to the way the balanced grammar is constructed we can select $k \leq n$ rules $v_1, \ldots, v_k$ such that $S$ can be written as the concatenation of the string produced by $v_1, \ldots, v_k$, where for each $v_i$, $|v_i| \leq \frac{N}{n}$ and $height(v_i) = O(\log \frac{N}{n})$[1]. We store these in a linked list. For the access data structure we store the prefix sums of the number of characters $|v_1|, \ldots, |v_k|$ and a predecessor data structure containing these values (using the y-fast trie due to Willard [107]). We then expand the grammar and build the data structures described in the previous section.

Queries are handled the same way as before when we reach the part of the grammar that has been expanded. All we have to do is to specify where to start the query among the rules $v_1, \ldots, v_k$. For an access query we first ask for the predecessor among the prefix sums of the length of the strings produced by $v_1, \ldots, v_k$.

Querying the predecessor data structure takes $O(\log \log N)$ time which is dominated by the $O(\log_\tau N)$ time it takes to search the grammar using the data structure described in the previous section.

The y-fast trie requires $O(n \log N)$ bits of space. For the expanded grammar we pick $O(n \log_\tau (N/n))$ rules with $\Theta(\tau)$ variables on each right-hand side. We use $O(\tau \log N)$ bits of space per rule to store the fusion trees, so the total space usage becomes $O(n\tau \log_\tau (N/n) \log N)$ bits. This concludes the proof of Theorem 5.2.

*Acknowledgements*

The author thanks Noy Rotbart, Pawel Gawrychowski, Philip Bille, and Søren Vind for helpful discussions, and Simon Puglisi, Djamal Belazzougui, and Yasuo Tabei for agreeing to merge their work ([15]) with this.

[1] This is implicit in section 5 of Rytter's paper.

# CHAPTER 6

# BOOKMARKS IN GRAMMAR-COMPRESSED STRINGS

Patrick Hagge Cording* Pawel Gawrychowski† Oren Weimann‡

* Technical University of Denmark, DTU Compute
† Max-Planck-Institut fur Informatik
‡ University of Haifa

**Abstract**

We consider the problem of storing a Straight Line Program $\mathcal{S}$ of size $n$ compressing a string of size $N$, and a set of positions $\{i_1, \ldots, i_b\}$ such that any substring of length $l$ starting from one of the positions can be decompressed in $O(l)$ time. Our solution requires space

$$O\left(\min_{1\leq\tau\leq\log^* N}\left\{\tau(n+b)+\min\{n,b\}\log^{(\tau)} N\right\}\right).$$

## 6.1 Introduction

A bookmark in a compressed string $S$ is a position $i$ from which the substring $S[i, i+l]$ can be decompressed in $O(l)$ time. For the *bookmark problem*, we are given $S$ of length $N$ compressed into a Straight Line Program $\mathcal{S}$ of size $n$ and a set of positions $\{i_1, \ldots, i_b\}$, and we want to construct a data structure that supports linear time decompression from any of the $b$ positions. We present a bookmarking data structure giving the following theorem.

**Theorem 6.1** *Given an SLP for $S[1, N]$ with $n$ rules and positions $i_1, \ldots, i_b$ in $S$, we can store $S$ in space*

$$O\left(\min_{1\leq\tau\leq\log^* N}\left\{\tau(n+b)+\min\{n,b\}\log^{(\tau)} N\right\}\right)$$

*such that later, given $i \in \{i_1, \ldots, i_b\}$ we can extract $S[i, i+l]$ in $O(l)$ time.*

In the case where $b = \Theta(n)$ the best trade-off is obtained by setting $\tau = \log^* N$ and the space becomes $O(n \log^* N)$.

Bookmarking of compressed strings is useful when we for instance want to compress a collection of documents in to one file. For this scenario, we may store a list of document titles or IDs on the side, and define a bookmark in the beginning of each document. Now we can decompress a document without any time overhead.

Gagie et al. [42] presented a bookmarking data structure that uses $O(n + b \log^* N)$ space for balanced SLPs[1]. When the input is unbalanced, we may use an algorithm to balance it at the cost of adding nodes [26, 88], and as a result the space usage of their data structure becomes $O(n \log \frac{N}{n} + b \log^* N)$.

If we allow a "kick-off" time of $O(\log N)$ when decompressing a string, we may use the $O(n)$-space data structure of Bille et al. [21] to decompress substrings. This does not require any bookmarks to be predefined. If the string that we want to decompress is longer than $l$ then this solution is sufficient to get $O(l)$ decompression time. We will use the data structure of Bille et al. for $\tau$ slightly modified copies of $\mathcal{S}$ to exploit the same idea for values of $l$ smaller than $\log N$.

## 6.2 Preliminaries

Let $S$ be a string of length $|S|$ consisting of characters from an alphabet of size $\sigma$. We use $S[i, j]$, $1 \leq i \leq j \leq |S|$, to denote the substring starting in position $i$ and ending in position $j$ of $S$.

A Straight Line Program (SLP) is a context-free grammar in Chomsky normal form with $n$ production rules that derives a single string $S$ of size $N$. We represent the SLP as a rooted, ordered, and node-labelled directed acyclic graph (DAG) with outdegree 2 and we will refer to production rules as nodes where it is appropriate. A depth-first left-to-right traversal starting from a node $v$ in the DAG produces the string $S(v)$. As a shorthand we sometimes use $|v|$ instead of $|T(v)|$. The tree that emerges from the traversal we call the parse tree. We denote the left and right children of $v$ by *left*$(v)$ and *right*$(v)$, respectively, in both the SLP and the parse tree.

All logarithms in this paper are base 2. As a shorthand to denote the logarithm applied $i$ times to a number $n$ we write $\log^{(i)} n$, e.g., $\log^{(3)} n = \log \log \log n$. The iterated logarithm $\log^* n$ is equal to the number of times the logarithm can be applied to $n$ before the result is less than 1, i.e., $\log^* n = \arg\min_i \{\log^{(i)} n \leq 1\}$.

## 6.3 A Simple Solution

As backbone in our data structure we use a data structure by Bille et al. [21] summarized in the next lemma.

**Lemma 6.1 ([21])** *Let $S$ be a string of length $N$ compressed by an SLP of size $n$. There is data structure of size $O(n)$ that supports decompression of a substring $S[i, i + l]$ in $O(\log N + l)$ time.*

The key observation given this data structure is that when $l \geq \log N$ the decompression time is dominated by the $l$ term, and we therefore only need to focus on the case where $l < \log N$.

To obtain a data structure using $O(n + b \log N)$ space for the bookmarking problem we store the substring $S[i, i + \log N]$ for each bookmark $i \in \{i_1, \ldots, i_b\}$ along with the data structure of Lemma 6.1. When we need to decompress a string we use the data structure of Lemma 6.1 if $l \geq \log N$ and otherwise we just read form the stored substring.

In the case where $n < b$ it is sufficient to store $n$ substrings of length $O(\log N)$ to get $O(l)$-time decompression from bookmarks. For this data structure we need the property given in the following lemma. The observation was first used for compressed pattern matching [82] and later it appeared in other variations for other purposes [52, 75]. However, we will give a proof using our terminology for the sake of completeness.

[1]The bound is in fact $O(z + b \log^* N)$, where $z$ is the size of the LZ77 parse of $S$. Since it is known that $z \leq n' \leq n$ [88], where $n'$ is the size of the smallest SLP generating $S$, we replace $z$ by $n$ for clarity.

**Lemma 6.2** *Let $S$ be a string of length $N$ compressed by an SLP of size $n$. Let $r(v) = S(u)[\max\{1, |u| - k\}, |u|] \circ S(w)[1, \min\{1, k-1\}]$ be the relevant substring with respect to $k$ of a node $v = uw$ in $\mathcal{S}$. Then any substring of $S$ of length at most $k$ is also a substring of at some string in the set $\{r(v) \mid v \in \mathcal{S} \wedge |v| \geq k\}$.*

**Proof** The proof is by induction. For the base case, consider a node $v = uw$ where $|v| \leq 2k - 2$ and $|u| < k$ and $|w| < k$. Since $r(v) = S(v)$ this obviously contains every substring of length $k$. For the inductive step we again consider some node $v = uw$ and we know that $S(v) = S(u) \circ S(w)$. Assume that $|u| \geq k$ and $|w| \geq k$, then by the induction hypothesis it holds that the set of strings $\{r(u') \mid u' \in \mathcal{S}(u) \wedge |u'| \geq k\} \cup \{r(w') \mid w' \in \mathcal{S}(w) \wedge |w'| \geq k\}$ contains all substrings of length $k$ in $S(u)$ and $S(w)$. The substrings of length $k$ starting in $S(u)$ and $S(w)$ are not guaranteed to be in this set, but since $r(v)$ contains exactly all these, they will be after adding $r(v)$ to the set. For the cases when $|u| < k$ or $|w| < k$ the same argument holds. ∎

For our data structure set $k = \log N$ and store the strings $r(v)$ for all $v \in \mathcal{S}$. For each bookmark $i$ we store the node that generates the string $S[i, i + \log N]$ and store the "local" index in $r(v)$. Furthermore, we build the data structure of Lemma 6.1 for use for the case where $l \geq \log N$.

Since $k \leq |r(v)| \leq 2 \log N - 2$ and we store a constant number of words (pointers) for each bookmark, and the data structure of Lemma 6.1 uses $O(n)$ space, our data structure uses $O(n \log N + b)$ space in total. To summarize, we have shown the following theorem.

**Theorem 6.2** *Given an SLP for $S[1, N]$ with $n$ rules and positions $i_1, \ldots, i_b$ in $S$, we can store $S$ in $O(n + b + \min\{n, b\} \log N)$ space such that later, given $i \in \{i_1, \ldots, i_b\}$ we can extract $S[i, i + l]$ in $O(l)$ time.*

## 6.4 Generalized Solution

We now describe a data structure that seeks to reduce the $\log N$ factor of the space usage in Theorem 6.2. The time to decompress a string $S[i, i + l]$ for some bookmark $i$ is still $O(l)$. Key to the solution is a technique due to Gawrychowski [47] captured by the following lemma.

**Lemma 6.3 ([47])** *Let $S$ be a string of length $N$ compressed into an SLP $\mathcal{S}$ of size $n$. We can choose an arbitrary $\ell$ and modify $\mathcal{S}$ in $O(N)$ time adding $O(n)$ new variables such that we can write $S$ as $S = S(v_1) \circ \ldots \circ S(v_m)$ with $m = O(N/\ell)$ and $|S(v_i)| \leq 2\ell - 2$. Among the nodes $v_1, \ldots, v_m$ there are $O(n)$ distinct nodes.*

A proof of the lemma is given in the appendix. The lemma tells that we can restructure the given SLP such that for any substring $S[i, i + \ell]$ we can find $O(1)$ nodes whose concatenation has length $O(\ell)$ and contains $S[i, i + \ell]$. We now describe how to apply this result to get a bookmarking data structure using almost linear space. In the description we use the parameter $\tau$ which is later to be minimized subject to $n$ and $b$.

First we make $\tau$ copies of $\mathcal{S}$, denoted by $\mathcal{S}_1, \ldots, \mathcal{S}_\tau$. Apply the restructuring algorithm for $\ell = \log N, \log^{(2)} N, \ldots, \log^{(\tau)} N$ to the $\tau$ copies of $\mathcal{S}$ to get $\mathcal{S}'_1, \ldots, \mathcal{S}'_\tau$. Build the data structure of Lemma 6.1 for each SLP $\mathcal{S}'_1, \ldots, \mathcal{S}'_\tau$. For each copy of the SLP $\mathcal{S}'_i$, let a *block node* be a node $v$ for which $|S(v)| = \Theta(\log^{(i)} N)$. For each bookmark $i$ we store the $O(1)$ block nodes generating the string containing $S[i, i + \ell]$ for each SLP $\mathcal{S}'_1, \ldots, \mathcal{S}'_\tau$. We also store the relative index of position $i$ in the string generated by the first block node. On the lowest levest, i.e., for $\mathcal{S}'_\tau$, we apply the technique from the previous section that requires the least space. Specifically, if $b < n$ we store the substrings of length $\log^{(\tau)} N$ starting in each bookmark, and if $n < b$ we set $k = \log^{(\tau)} N$ in Lemma 6.2 and store the relevant substrings subject to this $k$-value.

To decompress a substring of length $l$ from a bookmark position $i \in \{i_1, \ldots, i_b\}$ we do the following. Suppose that $\log^{(j+1)} N < l \leq \log^{(j)} N$ for $1 \leq j < \tau$. We locate the block node that contains $i$ in $\mathcal{S}'_j$ and starts decompressing the string starting in the relative position stored for the current bookmark using the data structure of Lemma 6.1. If we reach the end of the string generated by the current block node, we move on to the next one stored and repeat the process from relative position 1.

When we decompress from a block node $v$ in $\mathcal{S}'_j$, the query time of Lemma 6.1 becomes $O(\log\log^{(j)} N + l) = O(l)$ since $\log^{(j+1)} N < l$. We visit $O(1)$ block nodes so the totalt time to decompress $S[i, i+l]$ becomes $O(l)$.

If $l < \log^{(\tau)} N$ we use the solution chosen for the bottom level, which according to Theorem 6.2 yields a decompression time of $O(l)$.

Our data structure creates $\tau$ copies of $\mathcal{S}$. Each have size $O(n)$ after the restructuring of Lemma 6.3 and the application of Lemma 6.1, i.e, this requires $O(\tau n)$ space. For each bookmark we store references to $O(1)$ nodes in each copy totalling $O(\tau b)$ space. For $\mathcal{S}'_\tau$ we need $O(\min\{n, b\} \log^{(\tau)} N)$ space as stated in Theorem 6.2. In conclusion we get Theorem 6.1.

Notice that this is a generalization of Theorem 6.2. If we set $\tau = 1$ the two are equivalent.

### *6.4.1 Getting Linear Space*

There are certain values of $n$ and $b$ for which our data structure uses linear space. Obviously, if $n = O(N)$ or $b = O(N)$ we can afford to store the entire uncompressed string, and decompression becomes equivalent to reading contiguous memory. Moreover, if $b = O(1)$ the space usage of our simple solution (Theorem 6.2) becomes $O(n+\log N) = O(n)$ because $\log N \leq n$ for SLPs. In some cases, we may get $O(n + b)$ space for non-constant values of $b$, as seen here.

**Corollary 6.1** *Given an SLP for $T[1, N]$ with $n$ rules and positions $i_1, \ldots, i_b$ in $T$. If $b \leq \frac{n}{\log^{(c)} N}$ or $n \log^{(c)} N \leq b$ for some constant $c$, we can store $T$ in space $O(n + b)$, such that later, given $i \in \{i_1, \ldots, i_b\}$ we can extract $T[i, i + l]$ in $O(l)$ time.*

**Proof** Set $\tau = c$ in Theorem 6.1 and it follows. ∎

Our solution also yields linear space if the alphabet is sufficiently small.

**Corollary 6.2** *Given an SLP for $T[1, N]$ with $n$ rules, where the characters are drawn from an alphabet of size $\sigma$, and positions $i_1, \ldots, i_b$ in $T$. If $\sigma \leq 2^{1+\max\{n/b, b/n\}}$, we can store $T$ in space $O(n + b)$, such that later, given $i \in \{i_1, \ldots, i_b\}$ we can extract $T[i, i + l]$ in $O(l)$ time.*

**Proof** Consider Theorem 6.2. To store the set of strings we need $O(n+b+\min\{n, b\} \log N \frac{\log \sigma}{w})$ space, where $w$ is the word size. Since $w \leq \log N$ in our model we set $w = \log N$. By manipulation of the expression we see that the space becomes linear when $\sigma \leq 2^{(n+b)/\min\{n,b\}} = 2^{1+\max\{n/b, b/n\}}$. ∎

# BIBLIOGRAPHY


[1] S. Abiteboul, S. Alstrup, H. Kaplan, T. Milo, and T. Rauhe. Compact labeling scheme for ancestor queries. *SIAM J. Comput*, 35(6):1295–1309, 2006.

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

[3] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th ICALP*, pages 270–280, 1997.

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

[5] S. Alstrup, J. P. Secher, and M. Spork. Optimal on-line decremental connectivity in trees. *Inform. Process. Lett.*, 64(4):161–164, 1997.

[6] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Comput. System Sci.*, 52(2):299–307, 1996.

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

[8] A. Amir, M. Lewenstein, and E. Porat. Faster algorithms for string matching with k mismatches. *J. of Algorithms*, 50(2):257–275, 2004.

[9] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.

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

[11] R. A. Baeza-Yates. Searching subsequences. *Theoret. Comput. Sci.*, 78(2):363–376, 1991.

[12] H. Bannai, P. Gawrychowski, S. Inenaga, and M. Takeda. Converting SLP to LZ78 in almost linear time. In *Proc. 24th CPM*, pages 38–49, 2013.

[13] H. Bannai, S. Inenaga, and M. Takeda. Efficient LZ78 factorization of grammar compressed text. In *Proc. 19th SPIRE*, pages 86–98, 2012.

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

[15] D. Belazzougui, S. J. Puglisi, and Y. Tabei. Rank, select and access in grammar-compressed strings. *arXiv preprint arXiv:1408.3093*, 2014.

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

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

[18] P. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts. *ACM Trans. Algorithms*, 6(1):3, 2009.

[19] P. Bille, I. L. Gørtz, and J. Kristensen. Longest common extensions via fingerprinting. In *Proc. 6th LATA*, pages 119–130. 2012.

[20] 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.

[21] 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.

[22] L. Boasson, P. Cegielski, I. Guessarian, and Y. Matiyasevich. Window-accumulated subsequence matching problem is linear. In *Proc. 18th PODS*, pages 327–336, 1999.

[23] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (QUASAR). In *Proc. 3rd RECOMB*, pages 77–83, 1999.

[24] P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich. Window subsequence problems for compressed texts. In *Proc. 1st CSR*, pages 127–136, 2006.

[25] P. Cégielski, I. Guessarian, and Y. Matiyasevich. Multiple serial episodes matching. *Inform. Process. Lett.*, 98(6):211–218, 2006.

[26] 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.

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

[28] M. Cohn and R. Khazan. Parsing with suffix and prefix dictionaries. In *Proc. 6th DCC*, pages 180–180, 1996.

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

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

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

[32] M. Crochemore, A. Langiua, and F. Mignosi. Note on the greedy parsing optimality for dictionary-based text compression. *Theoret. Comp. Sci.*, 525(0):55 – 59, 2014.

[33] M. Crochemore, B. Melichar, and Z. Troníček. Directed acyclic subsequence graph—-overview. *J. Discrete Algorithms*, 1(3):255–280, 2003.

[34] G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proc. 8th CPM*, pages 12–27, 1997.

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

[36] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th FOCS*, pages 137–143, 1997.

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

[38] P. Ferragina and S. Muthukrishnan. Efficient dynamic method-lookup for object oriented languages. In *Proc. 4th ESA*, pages 107–120, 1996.

[39] P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel–Ziv compression. *SIAM J. Comput.*, 42(4):1521–1541, 2013.

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

[41] 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.

[42] 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. 2014.

[43] T. Gärtner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58, 2003.

[44] L. Gasieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 5th SWAT*, pages 392–403. Springer, 1996.

[45] P. Gawrychowski. Personal communication, July 2014.

[46] P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In *Proc. 19th ESA*, pages 421–432. Springer, 2011.

[47] P. Gawrychowski. Faster algorithm for computing the edit distance between slp-compressed strings. In *Proc. 19th SPIRE*, pages 229–236. Springer, 2012.

[48] P. Gawrychowski. Optimal pattern matching in LZW compressed strings. *ACM Trans. Algorithms*, 9(3):25, 2013.

[49] 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.

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

[51] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. Computing q-gram non-overlapping frequencies on SLP compressed texts. In *Proc. 38th SOFSEM*, pages 301–312. 2012.

[52] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. Speeding up q-gram mining on grammar-based compressed texts. In *Proc. 23rd CPM*, pages 220–231, 2012.

[53] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. Fast q-gram mining on SLP compressed strings. *J. Discrete Algorithms*, 18(0):89–99, 2013.

[54] K. Goto, S. Maruyama, S. Inenaga, H. Bannai, H. Sakamoto, and M. Takeda. Restructuring compressed texts without explicit decompression. *arXiv preprint arXiv:1107.2729*, 2011.

[55] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

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

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

[58] M. Hirao, A. Shinohara, M. Takeda, and S. Arikawa. Fully compressed pattern matching algorithm for balanced straight-line programs. In *Proc. 7th SPIRE*, pages 132–138, 2000.

[59] S. Inenaga and H. Bannai. Finding characteristic substrings from compressed texts. *Internat. J. Found. Comput. Sci.*, 23(02):261–280, 2012.

[60] A. Jeż. Faster fully compressed pattern matching by recompression. In *Proc. 39th ICALP*, pages 533–544. 2012.

[61] A. Jeż. Approximation of grammar-based compression via recompression. In *Proc. 24th CPM*, pages 165–176, 2013.

[62] A. Jeż. A really simple approximation of smallest grammar. In *Proc. 25th CPM*, 2014.

[63] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 16th MFCS*, pages 240–248, 1991.

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

[65] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, and C. Zaroliagis. Improved bounds for finger search on a ram. *Algorithmica*, pages 1–38, 2013.

[66] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching on Ziv–Lempel compressed text. *J. Discrete Algorithms*, 1(3):313–338, 2003.

[67] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[68] J. Kärkkäinen and E. Sutinen. Lempel–Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998.

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

[70] M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In *Proc. 6th CPM*, pages 205–214. Springer, 1995.

[71] D. Knuth, J. Morris, and V. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[72] G. M. Landau and U. Vishkin. Fast Parallel and Serial Approximate String Matching. *J. Algorithms*, 10:157–169, 1989.

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

[74] C. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: A string kernel for SVM protein classification. In *Proc. PSB*, volume 7, pages 566–575, 2002.

[75] Y. Lifshits. Processing compressed texts: A tractability border. In *Proc. 18th CPM*, pages 228–240. Springer, 2007.

[76] M. Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups Complexity Cryptology*, 4(2):241–299, 2012.

[77] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[78] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Min. Knowl. Discov.*, 1(3):259–289, 1997.

[79] W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoret. Comput. Sci.*, 410(8):900–913, 2009.

[80] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[81] 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.

[82] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. 8th CPM*, pages 1–11. Springer, 1997.

[83] S. Muthukrishnan and M. Müller. Time and space efficient method-lookup for object-oriented programs. In *Proc. 7th SODA*, pages 42–51, 1996.

[84] G. Navarro. A guided tour to approximate string matching. *ACM CSUR*, 33(1):31–88, 2001.

[85] 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.

[86] G. Paaß, E. Leopold, M. Larson, J. Kindermann, and S. Eickeler. SVM classification using sequences of phonemes and syllables. In *Proc. 6th PKDD*, pages 373–384, 2002.

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

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

[89] T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. *IEICE Trans. Fundamentals*, 86(5):1061–1066, 2003.

[90] D. D. Sleator and R. Endre Tarjan. A data structure for dynamic trees. *J. Comput. System Sci.*, 26(3):362–391, 1983.

[91] T. Starikovskaya and H. W. Vildhøj. Time-Space Trade-Offs for the Longest Common Substring Problem. In *Proc. 24th CPM*, pages 223–234, 2013.

[92] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[93] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. 3rd ESA*, pages 327–340, 1995.

[94] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. 7th CPM*, pages 50–63, 1996.

[95] T. Takaoka. Approximate pattern matching with samples. In *Proc. 5th ISAAC*, pages 234–242, 1994.

[96] 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.

[97] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Algorithms*, 42(2):205–230, 2002.

[98] A. Tiskin. Faster subsequence recognition in compressed strings. *J. Math. Sci.*, 158(5):759–769, 2009.

[99] A. Tiskin. Towards approximate matching in compressed strings: Local subsequence recognition. In *Proc. 6th CSR*, pages 401–414, 2011.

[100] A. Tiskin. Threshold approximate matching in grammar-compressed strings. In *Proc. 18th PSC*, page 124, 2014.

[101] Z. Troníček. Episode matching. In *Proc. 12th CPM*, pages 143–146, 2001.

[102] E. Ukkonen. Approximate string-matching with $q$-grams and maximal matches. *Theoret. Comput. Sci.*, 92(1):191–211, 1992.

[103] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[104] 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.

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

[106] P. Weiner. Linear Pattern Matching Algorithms. In *Proc. 14th FOCS (SWAT)*, pages 1–11, 1973.

[107] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta$(N). *Inf. Proc. Let.*, 17(2):81–84, 1983.

[108] T. Yamamoto, H. Bannai, S. Inenaga, and M. Takeda. Faster subsequence and don't-care pattern matching on compressed texts. In *Proc. 22nd CPM*, pages 309–322, 2011.

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

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

# Appendices

## Proof of Lemma 6.3

For completeness of this work, we give a proof of Lemma 6.3 (alternative to the one given in [47]).

Before the proof we introduce some notation. We consider rooted, ordered, binary trees and forests with labels on both nodes and edges. As with SLPs, we denote the left and right child of a node $v$ by *left*$(v)$ and *right*$(v)$. The depth of a node $v$ is the number of edges on the path from the root to $v$. The nearest common ancestor $nca(v, u)$ of two nodes $v$ and $u$ is the deepest node that is an ancestor of both $v$ and $u$. Let $p = v_1, v_2, \ldots, v_k$ be a path in a forest $F$ such that $v_1$ is an ancestor of $v_k$. The *left forest* of $p$ is composed of the subtrees $F($*left*$(v_i))$, where $1 \leq i < k$ and *left*$(v_i)$ is not on $p$. The *right forest* of $p$ is defined similarly.

For the proof we will describe an algorithm that transforms the SLP and show that the resulting SLP obeys the specifics of the lemma. On a high level, the algorithm selects all the nodes that generate strings of size $\Theta(\ell)$. This partitions $S$ into a substrings where some are covered by the selected nodes and some are not. Suppose for simplicity that a substring $S[i, j]$ that is not covered by a selected node has a substring of size $\Theta(\ell)$ on its left and on its right. We then use the path going from the selected node on the left to the selected node on the right to construct new grammars generating strings of length $O(\ell)$ that generate $S[i, j]$.

**Proof** Let $\mathcal{B}$ be the set of nodes that generate strings of length greater than or equal to $\ell$ and whose children generate strings of length less than $\ell$. The nodes in $\mathcal{B}$ generate strings of length at most $2\ell - 2$. Let $u', v'$ be an occurrence of $u, v \in \mathcal{B}$ in the parse tree of $\mathcal{S}$ such that $u'$ is to the left of $v'$ and no other node from $\mathcal{B}$ occurs in the right forest of the path from $v'$ to $nca(v', u')$ or in the left forest of the path from $u'$ to $nca(v', u')$. For all such pairs $u', v'$, the forest $F_l$ contains the path from $u'$ to $nca(u', v')$ and the forest $F_r$ contains the path from $v'$ to $nca(u', v')$.

The two forests are labelled as follows. In $F_l$, if there is an edge $\{$*left*$(v), v\}$ then we assign it the primary label $p(\{$*left*$(v), v\}) =$ *right*$(v)$ and secondary label $s(\{$*left*$(v), v\}) = \varepsilon$. Labels for $F_r$ are defined conversely: If $\{$*left*$(v), v\}$ is an edge then $p(\{$*left*$(v), v\}) = \varepsilon$ and $s(\{$*left*$(v), v\}) =$ *right*$(v)$, if $\{$*right*$(v), v\}$ is an edge then $p(\{$*right*$(v), v\}) =$ *left*$(v)$ and $s(\{$*right*$(v), v\}) = \varepsilon$.

The size of $F_l$ and $F_r$ is $O(n)$. We prove this by arguing that a node only occurs X times in $F_l$ (and $F_r$). Let $v$ be a node in $F_l$. There can be at most two paths ending in $v$, because $v$ can only be the nearest common ancestor of one specific pair of nodes in $\mathcal{B}$, so $v$ occurs at most twice as a leaf in $F_l$. The node $v$ also occurs at most twice as an internal node in $F_l$. Let $w$ be a node in $\mathcal{S}(v)$ that is also in $\mathcal{B}$. Let $u \in \mathcal{B}$ be any node occuring to the left of $w$ in the parse tree of $\mathcal{S}$ without any other node from $\mathcal{B}$ occurring between them. If $u \in \mathcal{S}(v)$ the the path from $w$ to $nca(w, u)$ does not contain $v$. If $u \notin \mathcal{S}(v)$, the path from $w$ to $v$ is the same for any $u$. Hence, $v$ occurs only once in $F_l$ for this case. Similarly, $v$ only occurs once in $F_l$ for the case where $u$ occurs to the right of $w$ in the parse tree without any other nodes from $\mathcal{B}$ between them.

Finding $\mathcal{B}$ takes $O(n)$ time. Finding the ancestors of each pair can be done in one pass of the parse tree by marking the highest node visited when going from one node in $\mathcal{B}$ to the next. One pass takes $O(N)$ time. The paths between nodes in $\mathcal{B}$ can be found in $O(N)$ time, so the construction of $F_l$ and $F_r$ can also be done in $O(N)$ time.

In the following we show how to use $F_l$ and $F_r$ to restructure the SLP. The algorithm proceeds in two rounds. First we compute $F_l$ and use it to restructure the SLP. Then we compute $F_r$ from the new SLP and use it to restructure the SLP once again.

Starting from the roots of $F_l$ (or $F_r$ in the second round of the algorithm), we partition the trees greedily into edge-disjoint clusters s.t. the length of the string generated by the concatenation of primary edge labels is at most $\ell$ on any root to leaf path in a cluster.

Consider a cluster rooted in $v \in F_l$ with leaves $v_1, \ldots, v_k$. For every root-to-leaf path in the cluster we build a tree over the primary and secondary labels on the path[2]. Let $v_i$ be the current leaf and $t_p$ the root of the tree over primary labels and $t_s$ the root of tree over secondary labels. Replace $v_i$ by $v_i = t_s v t_p$ in $\mathcal{S}$. In the second round, when we consider clusters in $F_r$, the node is replaced by $v_i = t_p v t_s$. The nodes of the trees rooted in $t_p$ and $t_s$ are the new nodes of the SLP.

When a node in $\mathcal{S}$ is replaced it generates the same string, so the resulting SLP also generates $S$. This concludes the proof of Lemma 6.3. ∎

[2]To efficiently reuse subtrees when building the trees, they should be constructed from the root of $F_l$ and down. This ensures that there is a tree representing the edges of any path from the root to an internal node in $F_l$.