

Technical University of Denmark



## Compressed and Practical Data Structures for Strings

Christiansen, Anders Roy; Bille, Philip; Gørtz, Inge Li

*Publication date:*  
2018

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

[Link back to DTU Orbit](#)

*Citation (APA):*  
Christiansen, A. R., Bille, P., & Gørtz, I. L. (2018). Compressed and Practical Data Structures for Strings. DTU Compute. (DTU Compute PHD-2017, Vol. 464).

**DTU Library**  
Technical Information Center of Denmark

---

### General rights

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

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

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

# Compressed and Practical Data Structures for Strings

Anders Roy Christiansen

DTU



Kongens Lyngby 2017

PHD-2017-464

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](mailto:compute@compute.dtu.dk)  
[www.compute.dtu.dk](http://www.compute.dtu.dk)

PHD-2017-464  
ISSN: 0909-3192

# Abstract

---

In this dissertation, I will cover a number of different topics related to strings in compressed and practical settings. I will first present some fundamental techniques from the area, and then cover 6 different topics within the area. A short introduction to each of these topics is given in the following.

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

Let  $n$  be the size of the grammar, and let  $N$  be the size of the string. For the static variant we give a linear space representation that supports placing the finger in  $O(\log N)$  time and subsequently accessing in  $O(\log D)$  time, where  $D$  is the distance between the finger and the accessed index. For the dynamic variant we give a linear space representation that supports placing the finger in  $O(\log N)$  time and accessing and moving the finger in  $O(\log D + \log \log N)$  time. Compared to the best linear space solution to random access, we improve a  $O(\log N)$  query bound to  $O(\log D)$  for the static variant and to  $O(\log D + \log \log N)$  for

the dynamic variant, while maintaining linear space. As an application of our results we obtain an improved solution to the longest common extension problem in grammar compressed strings. To obtain our results, we introduce several new techniques of independent interest, including a novel van Emde Boas style decomposition of grammars.

**Compressed Indexing with Signature Grammars.** The *compressed indexing problem* is to preprocess a string  $S$  of length  $n$  into a compressed representation that supports pattern matching queries. That is, given a string  $P$  of length  $m$  report all occurrences of  $P$  in  $S$ .

We present a data structure that supports pattern matching queries in  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  time using  $O(z \lg(n/z))$  space where  $z$  is the size of the LZ77 parse of  $S$  and  $\epsilon > 0$ , when the alphabet is small or the compression ratio is at least polynomial. We also present two data structures for the general case; one where the space is increased by  $O(z \lg \lg z)$ , and one where the query time changes from worst-case to expected.

In all cases, the results improve the previously best known solutions. Notably, this is the first data structure that decides if  $P$  occurs in  $S$  in  $O(m)$  time using  $O(z \lg(n/z))$  space.

Our results are mainly obtained by a novel combination of a randomized grammar construction algorithm with well known techniques relating pattern matching to 2D-range reporting.

**Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation.** Given a static reference string  $R$  and a source string  $S$ , a relative compression of  $S$  with respect to  $R$  is an encoding of  $S$  as a sequence of references to substrings of  $R$ . Relative compression schemes are a classic model of compression and have recently proved very successful for compressing highly-repetitive massive data sets such as genomes and web-data. We initiate the study of relative compression in a dynamic setting where the compressed source string  $S$  is subject to edit operations. The goal is to maintain the compressed representation compactly, while supporting edits and allowing efficient random access to the (uncompressed) source string. We present new data structures that achieve optimal time for updates and queries while using space linear in the size of the optimal relative compression, for nearly all combinations of parameters. We also present solutions for restricted and extended sets of updates. To achieve these results, we revisit the dynamic partial sums problem and the substring concatenation problem. We present new optimal or

near optimal bounds for these problems. Plugging in our new results we also immediately obtain new bounds for the string indexing for patterns with wildcards problem and the dynamic text and static pattern matching problem.

**Succinct Partial Sums and Fenwick Trees.** We consider the well-studied *partial sums* problem in succinct space where one is to maintain an array of  $n$   $k$ -bit integers subject to updates such that partial sums queries can be efficiently answered. We present two succinct versions of the Fenwick Tree – which is known for its simplicity and practicality. Our results hold in the encoding model where one is allowed to reuse the space from the input data. Our main result is the first that only requires  $nk + o(n)$  bits of space while still supporting sum/update in  $\mathcal{O}(\log_b n)$  /  $\mathcal{O}(b \log_b n)$  time where  $2 \leq b \leq \log^{\mathcal{O}(1)} n$ . The second result shows how optimal time for sum/update can be achieved while only slightly increasing the space usage to  $nk + o(nk)$  bits. Beyond Fenwick Trees, the results are primarily based on bit-packing and sampling – making them very practical – and they also allow for simple optimal parallelization.

**Fast Dynamic Arrays.** We present a highly optimized implementation of tiered vectors, a data structure for maintaining a sequence of  $n$  elements supporting access in time  $\mathcal{O}(1)$  and insertion and deletion in time  $\mathcal{O}(n^\epsilon)$  for  $\epsilon > 0$  while using  $o(n)$  extra space. We consider several different implementation optimizations in C++ and compare their performance to that of vector and multiset from the standard library on sequences with up to  $10^8$  elements. Our fastest implementation uses much less space than multiset while providing speedups of  $40\times$  for access operations compared to multiset and speedups of  $10.000\times$  compared to vector for insertion and deletion operations while being competitive with both data structures for all other operations.

**Parallel Lookups in String Indexes.** Here we consider the indexing problem on in the parallel random access machine model. Recently, the first PRAM algorithms were presented for looking up a pattern in a suffix tree. We improve the bounds, achieving optimal results for all parameters but the preprocessing. Given a text  $T$  of length  $n$  we create a data structure of size  $\mathcal{O}(n)$  that answers pattern matching queries for a pattern  $P$  of length  $m$  in  $\mathcal{O}(\log m)$  time and  $\mathcal{O}(m)$  work.



# Danish Abstract

---

I denne afhandling vil jeg undersøge en række forskellige emner omkring komprimering af strenge og strenge anvendt i praksis. Jeg vil først præsentere nogle fundamentale teknikker fra området og så beskrive 6 forskellige emner inden for området. En kort introduktion til hvert af de emner er givet nedenfor.

**Fingersøgning i grammatik komprimerede strenge.** Grammatik baseret komprimering, hvor en lang streng erstattes af en lille kontekst-fri grammatik, der genererer strengen, er et simpelt og stærkt værktøj, der omfatter mange populære komprimeringsteknikker. Givet en grammatik, er opslagsproblemet at lave en kompakt repræsentation af grammatikken, der tillader opslag på vilkårlige positioner, hvilket betyder at man skal kunne fortælle hvilket bogstav, der står på en givet position i den originale ukomprimerede streng. Vi undersøger opslagsproblemet med fingersøgnings egenskaben, hvilket betyder at tiden det tager at lave et opslag skal afhænge af afstanden mellem fingerpositionen  $f$  og opslagspositionen  $i$ . Vi ser både på en statisk variant, hvor man først placerer sin finger, og derefter laver opslag nær fingeren effektivt, og en dynamisk variant hvor det også er muligt at flytte fingeren således at tiden det tager at flytte fingeren afhænger af hvor langt den bliver flyttet.

Lad  $n$  være størrelsen på grammatikken, og lad  $N$  være længden af strengen. I det statiske tilfælde præsenterer vi en lineær plads repræsentation, der understøtter at sætte fingeren i  $O(\log N)$  tid og derefter opslag i  $O(\log D)$  tid, hvor  $D$  er afstanden imellem fingeren og opslaget. I det dynamiske tilfælde præsenterer vi en lineær plads repræsentation, der understøtter at sætte fingeren i  $O(\log N)$  tid og både opslag og flytning af fingeren i  $O(\log D + \log \log N)$  tid. Sammenlignet med den bedste lineærplads løsning til vilkårlige opslag forbedrer vi  $O(\log N)$



opslags tid til  $O(\log D)$  i det statiske tilfælde og til  $O(\log D + \log \log N)$  i det dynamiske tilfælde, imens vi fortsat kun bruger lineær plads. Som en anvendelse af vores resultater viser vi en forbedret løsning til det længste-fælles-udvidelsesproblem i grammatik komprimerede strenge. For at opnå vores resultater introducerer vi adskillelige nye teknikker af uafhængig interesse, inklusiv en ny van Emde Boas lignende dekomposition af grammatikker.

**Komprimeret indeksering med signaturgrammatikker.** Komprimeret indeksering er at præbehandle en streng  $S$  af længde  $n$  til en komprimeret repræsentation, der understøtter mønstergenkendelsesforespørgelser.

Vi præsenterer en datastruktur, der understøtter mønstergenkendelse i  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  tid og bruger  $O(z \lg(n/z))$  plads, hvor  $z$  er størrelsen af LZ77-parsen af  $S$  og  $\epsilon > 0$  er en konstant, når alfabetet er småt eller komprimeringen som minimum er polynomisk. Vi præsenterer også to datastrukturer for det generelle tilfælde; en hvor pladsforbruget er forøget med  $O(z \lg \lg z)$ , og en hvor forespørgelsestiden ændres fra at være i værste tilfælde til at være i forventning.

I alle tilfælde, forbedrer vores resultater de tidligere bedst kendte løsninger. Specielt er dette den første datastruktur, der kan afgøre om  $P$  optræder i  $S$  i  $O(m)$  tid ved brug af  $O(z \lg(n/z))$  plads.

Vores resultater er primært opnået ved en ny kombination af en tilfældig grammatik konstruktion og velkendte teknikker, der forbinder mønstergenkendelse med 2D-område rapportering.

**Dynamisk relativ komprimering, dynamiske delsummer, og delstrengs sammensætning.** Givet en statisk referencestreng  $R$  og en kildestreng  $S$  er en relativ komprimering af  $S$  i forhold til  $R$  en indkodning af  $S$  som en sekvens af referencer til delstrengene af  $R$ . Relativ komprimering er en klassisk metode til komprimering og har fornyligt bevist sit værd til komprimering af data med mange repetitioner, som f.eks. gener og web-data. Vi starter undersøgelsen af relativ komprimering i en dynamisk sammenhæng hvor den komprimerede kildestreng  $S$  bliver opdateret/ændret løbende. Målet er at holde den komprimerede repræsentation kompakt samtidigt med effektivt at understøtte ændringer og vilkårlige opslag i den ukomprimerede kildestreng. Vi præsenterer nye datastrukturer, der opnår optimal tid for opdateringer og forespørgelser, og samtidigt kun bruger lineær plads i forhold til den optimale relative komprimering for næsten alle kombinationer af parametre. Vi præsenterer også løsninger for problemet, hvor mængden af tilladte opdateringsoperationer er ændret. For at opnå disse resultater, undersøger vi det dynamiske delsumsproblem og delstrengssam-

mensætningsproblemet. Vi viser nye optimale eller næsten optimale grænser for disse problemer. Ved brug af vores nye resultater opnår vi også bedre grænser for strengindekseringsproblemet for mønstre med jokertegn og mønstergenkendelsesproblemet hvor teksten er dynamisk men mønsteret er statisk.

**Koncise delsummer og Fenwick træer.** Vi undersøger det velstuderede *delsumsproblem* i nær optimal plads, hvor man skal vedligeholde en tabel med  $n$   $k$ -bit heltal, der bliver opdateret løbende, således at der effektivt kan svares på delsumsforespørgelser. Vi præsenterer to koncise versioner af Fenwick træet – som er kendt for dets simplicitet og anvendelighed. Vores resultater er givet i indkodningsmodellen hvor det er muligt at genbruge pladsen fra input dataen. Vores primære resultat er det første, der kun kræver  $nk + o(n)$  bits plads og stadig understøtter sum/opdatering i  $\mathcal{O}(\log_b n)$  /  $\mathcal{O}(b \log_b n)$  tid hvor  $2 \leq b \leq \log^{O(1)} n$ . Det andet resultat viser hvordan man kan opnå optimal tid for sum/opdatering ved kun at øge pladsforbruget minimalt til  $nk + o(nk)$  bits. Udover Fenwick træer, er resultaterne primært baseret på bit-pakning og prøvetagning – hvilket gør dem meget anvendelige og muliggør simpel parallelisering.

**Hurtige dynamiske tabeller.** Vi præsenterer en meget optimeret implementation af lagdelte vektorer, en datastruktur til at vedligeholde en sekvens af  $n$  elementer, der tillader opslag i  $\mathcal{O}(1)$  tid og indsættelse/sletning i  $\mathcal{O}(n^\epsilon)$  når  $\epsilon > 0$  er en konstant, og bruger  $o(n)$  ekstra plads. Vi ser på flere forskellige implementationsoptimeringer i C++ og sammenligner deres ydeevne med vector og multiset fra C++'s standard bibliotek på sekvenser med op til  $10^8$  elementer. Vores hurtigste implementation bruger væsentligt mindre plads end multiset, giver hastighedsforbedringer på en faktor 40 for opslag ift. multiset, og op til 10000 gange hurtigere indsætning/sletning ift. vector, og samtidigt er ydeevnen sammenlignelig med begge datastrukturer for alle andre operationer.

**Parallele opslag i strengindeks.** Her ser vi på indekseringsproblemet på den parallelle vilkårligt-opslagsmaskine (PRAM). Fornyligt blev de første PRAM algoritmer til opslag efter et mønster i et suffikstræ præsenteret. Vi forbedrer grænserne, og opnår optimale resultater for alle parametre udover præbehandlingstiden. Givet en tekst  $T$  af længde  $n$ , konstruerer vi en datastruktur af størrelse  $\mathcal{O}(n)$ , der kan svare på mønstergenkendelsesopslag efter et mønster  $P$  af længde  $m$  i  $\mathcal{O}(\log m)$  tid og  $\mathcal{O}(m)$  arbejde.



# Preface

---

This dissertation is the result of research I have been doing while being a part of the project called Compressed Computation on Highly-Repetitive Data partially funded by the Danish Research Council (DFR – 4005-00267). It was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark. The 3 years of PhD studies started on October 1, 2014 and ended on November 30, 2017, during this period I had one month leave of absence where I worked on teaching materials for DTU. My supervisors are Associate Professor Philip Bille and Associate Professor Inge Li Gørtz.

**Acknowledgements** First of all, I will like to thank my two supervisors Inge and Philip, who have provided great advice and intellectual challenges during my studies, and always been available when needed. A big thank to Martín Farach-Colton who hosted me during my memorable external research stay in New York City. It was great to experience the city and another way to work and live as a researcher. Thanks to all my colleagues – especially to my office mates Mikko, Nicola, Frederik, Patrick, Hjalte, and Søren with whom I have had a great time and (too) many good discussions about all and nothing. Thanks to all the people in the community I have met at conferences and social events. Thanks to my friends, family, and Josefine for their support. Finally, thanks to my mascot who has given me renewed energy whenever going to the mail/printer room in the past years.



Anders Roy Christiansen

November 30, 2017  
Lyngby



# Contents

---

<b>Abstract</b>	<b>i</b>
<b>Danish Abstract</b>	<b>v</b>
<b>Preface</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Overview . . . . .	2
1.2 Preliminaries . . . . .	4
1.3 The Lempel-Ziv Family and Friends . . . . .	5
1.4 Context-Free Grammars and SLPs . . . . .	6
1.5 Heavy Paths . . . . .	12
1.6 On Chapter 2: Finger Search in Grammar-Compressed Strings .	14
1.7 On Chapter 3: Compressed Indexing with Signature Grammars .	16
1.8 On Chapter 4: Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation . . . . .	17
1.9 On Chapter 5: Succinct Partial Sums and Fenwick Trees . . . . .	19
1.10 On Chapter 6: Fast Dynamic Arrays . . . . .	20
1.11 On Chapter 7: Parallel Lookups in String Indexes . . . . .	22
<b>2 Finger Search in Grammar-Compressed Strings</b>	<b>25</b>
2.1 Introduction . . . . .	26
2.1.1 Related Work . . . . .	27
2.1.2 Our results . . . . .	28
2.1.3 Technical Overview . . . . .	29
2.1.4 Longest Common Extensions . . . . .	30
2.2 Preliminaries . . . . .	31
2.3 Fringe Access . . . . .	32
2.3.1 van Emde Boas Decomposition for Grammars . . . . .	33

2.3.2	Data Structure . . . . .	35
2.3.3	Improving the Query Time for Small Indices . . . . .	37
2.4	Static Finger Search . . . . .	39
2.5	Dynamic Finger Search . . . . .	41
2.5.1	Left Heavy Path Decomposition of a Path . . . . .	41
2.5.2	Data Structure . . . . .	42
2.5.3	Moving/Access to the Left of the Finger . . . . .	44
2.6	Finger Search with Fingerprints and Longest Common Extensions . . . . .	45
2.6.1	Fast Fingerprints on the Fringe . . . . .	45
2.6.2	Finger Search with Fingerprints . . . . .	46
2.6.3	Longest Common Extensions . . . . .	46
<b>3</b>	<b>Compressed Indexing with Signature Grammars</b> . . . . .	<b>49</b>
3.1	Introduction . . . . .	50
3.1.1	Our Results . . . . .	50
3.1.2	Technical Overview . . . . .	52
3.2	Preliminaries . . . . .	53
3.3	Signature Grammars . . . . .	54
3.3.1	Signature Grammar Construction . . . . .	55
3.3.2	Properties of the Signature Grammar . . . . .	56
3.4	Long Patterns . . . . .	58
3.4.1	Data Structure . . . . .	58
3.4.2	Searching . . . . .	59
3.4.3	Correctness . . . . .	60
3.4.4	Complexity . . . . .	61
3.5	Short Patterns . . . . .	61
3.6	Semi-Short Patterns . . . . .	62
3.6.1	Data Structure . . . . .	63
3.6.2	Searching . . . . .	63
3.6.3	Analysis . . . . .	64
3.7	Randomized Solution . . . . .	64
<b>4</b>	<b>Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation</b> . . . . .	<b>67</b>
4.1	Introduction . . . . .	68
4.1.1	Dynamic Relative Compression . . . . .	69
4.1.2	Dynamic Partial Sums . . . . .	70
4.1.3	Substring Concatenation . . . . .	72
4.1.4	Extensions . . . . .	73
4.2	Dynamic Relative Compression . . . . .	74
4.2.1	Data Structure . . . . .	75
4.2.2	Answering Queries . . . . .	76
4.3	Dynamic Partial Sums . . . . .	77
4.3.1	Dynamic Partial Sums for Small Sequences . . . . .	77

4.3.2	Dynamic Partial Sums for Large Sequences . . . . .	82
4.4	Substring Concatenation . . . . .	83
4.5	Extensions . . . . .	85
4.5.1	Dynamic Relative Compression with Access and Replace . . . . .	85
4.5.2	Dynamic Relative Compression with Split and Concatenate . . . . .	86
4.6	Conclusion . . . . .	87
<b>5</b>	<b>Succinct Partial Sums and Fenwick Trees</b>	<b>89</b>
5.1	Introduction . . . . .	90
5.2	Data structure . . . . .	91
5.2.1	Layered b-ary structure . . . . .	91
5.2.2	Sampling . . . . .	93
5.3	Optimal-time sum and update . . . . .	94
<b>6</b>	<b>Fast Dynamic Arrays</b>	<b>97</b>
6.1	Introduction . . . . .	97
6.2	Preliminaries . . . . .	99
6.3	Tiered Vectors . . . . .	99
6.4	Improved Tiered Vectors . . . . .	103
6.4.1	Implicit Tiered Vectors . . . . .	103
6.4.2	Lazy Tiered Vectors . . . . .	104
6.5	Implementation . . . . .	104
6.5.1	C++ Templates . . . . .	105
6.6	Experiments . . . . .	107
6.6.1	Comparison to C++ STL Data Structures . . . . .	108
6.6.2	Tiered Vector Variants . . . . .	109
6.6.3	Width Experiments . . . . .	111
6.6.4	Height Experiments . . . . .	112
6.6.5	Configuration Experiments . . . . .	112
6.7	Conclusion . . . . .	114
<b>7</b>	<b>Parallel Lookups in String Indexes</b>	<b>115</b>
7.1	Introduction . . . . .	115
7.2	Preliminaries . . . . .	117
7.3	Simple Fingerprint-Based Pattern Lookup . . . . .	118
7.4	Better Fingerprint-Based Pattern Lookup . . . . .	119
7.5	Parallel Suffix Array Pattern Lookup . . . . .	121
	<b>Bibliography</b>	<b>123</b>





## CHAPTER 1

# Introduction

---

The amount of digital data we store worldwide increases at an incredible pace. In order to use all this data for something meaningful, we need methods to handle it efficiently. Fortunately, hardware improvements over the past decades has enabled us to handle much more data than previously possible. Unfortunately, it is very difficult for hardware improvements to keep up with the current growth in data. Designing clever and efficient ways to handle all this data in software instead of relying on hardware improvements is therefore of utmost importance. This is where the topic of algorithms and data structures comes to the rescue. A key component in handling all this data is compression. All data with regularities can be compressed to use less space than the original data. One recent source of this growth in data is DNA sequencing where researchers try to understand the human genome. DNA sequences are long and thus takes up a lot of space, but the DNA of two people from the same population is approximately 99% similar, ie. only few differences in the DNA make individuals who they are. This means DNA is highly compressible due to the many repetitions in the DNA sequences, and therefore one among other good applications for the methods we develop in this dissertation.

Data compression has been studied in many different settings for a long time. For example a well-known area where data compression has played a significant role is video transmission. It is estimated that more than half of all traffic on the Internet is currently due to video streaming. Without video compression

this traffic would explode. For most video transmission it is acceptable that the received video is not 100% identical to the original video as long as the differences are (almost) invisible for the human eye, ie. we accept some loss of information. In this dissertation, however, we focus on lossless compression where information is not allowed to be lost. Most people also encounter this kind of compression on a daily basis – for instance almost all websites are transferred from the server to the client in compressed form to speedup the loading time. In fact, the compression technique most commonly used to do this, gzip, is based on one of the most fundamental compression schemes which we will encounter in this study, namely the LZ77 compression scheme. This compression scheme is also a key component of zip-files. The compression schemes we will study in this dissertation are all based on exploiting repetitions in the data.

Earlier on - and still to a large extent - when working with compressed data, the strategy was to simply compress it at some point and then when the original data was needed decompress all of the data again - even if only small amounts of the data was needed. In this dissertation we will instead consider compressed representations of the data that allows us to use the data without decompressing it first, we call such representations of data *compressed data structures*.

One of the topics in this dissertation is text indexing. A text index data structure is to a computer what the (word) index in the back of a book is to a human. That is, it is an (additional) representation of the original data that helps us quickly answer certain questions. In the case, it allows us to quickly answer where some given word occurs. This is a fundamental and well studied problem within the field of data structures that already have many variants and solutions. In this dissertation, we will consider this fundamental problem in a compressed setting. Thus we design compressed data structures that allow us to answer where a given word/pattern occurs in a text/string.

## 1.1 Overview

During my PhD studies I have co-authored the following papers:

**Finger Search in Grammar-Compressed Strings.** Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording and Inge Li Gørtz. *Accepted for publication in Theory of Computing Systems. An extended abstract appeared in the proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science.*

**Compressed Indexing with Signature Grammars.** Anders Roy Chris-

tiansen and Mikko Berggren Ettienne. *Submitted to the 13th Latin American Theoretical Informatics Symposium.*

**\* Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation.** Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, Inge Li Gørtz, Frederik Rye Skjoldjensen, Hjalte Wedel Vildhøj and Søren Vind. *In Algorithmica 2017.*

**Succinct Partial Sums and Fenwick Trees.** Philip Bille, Anders Roy Christiansen, Nicola Prezza and Frederik Rye Skjoldjensen. *In the proceedings of the 24th International Symposium on String Processing and Information Retrieval.*

**Fast Dynamic Arrays.** Philip Bille, Anders Roy Christiansen, Mikko Berggren Ettienne and Inge Li Gørtz. *In the proceedings of the 25th Annual European Symposium on Algorithms.*

**Parallel Lookups in String Indexes.** Anders Roy Christiansen and Martín Farach-Colton. *In the proceedings of the 23rd International Symposium on String Processing and Information Retrieval.*

Except from the paper "Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation", I have contributed to the entire process from the initial ideas to the final and published version – se details about my contributions to this last paper in section 1.8.

The first three of the papers in the above list are about compressed data structures and the second three are about data structures related to strings and are in some sense practical – thus the title of this dissertation. Five of the above papers are theoretical whereas the paper Fast Dynamic Arrays is mainly experimental. Five of the papers have already been peer-reviewed and published, but Compressed Indexing with Signature Grammars is in submission.

In the rest of this first chapter, I will give a short introduction to some of the fundamental compression schemes and techniques that I have been working with throughout my studies. Even though only half of my papers are based on compressed data structures, this is the topic I feel I have spent the most time on during my studies. Afterwards in section 1.6-1.11 I will give a short introduction to each of the 6 following chapters. In each of these introductions, I will shortly present a selection of the problem, solutions, main techniques, future directions and a background story on when/why I worked on that problem on my path to handing in this dissertation. The remaining chapters are (with very few exceptions) verbatim copies of the published papers listed above.

## 1.2 Preliminaries

**Models of Computation** With a single exception all papers presented in this dissertation are based on the RAM-model [69]. In Chapter 7 (Parallel Lookups in String Indexes) we consider the strongly related but parallel PRAM-model [77]. As the name suggests, the core difference is that parallel processing is possible in the PRAM model. In the PRAM model, we have multiple processors that can execute different code at the same time. It is normally assumed that an unlimited number of processors are available. This is a reasonable assumption due to Brent’s theorem [20] that shows how time/work is influenced if only a given number of processors are available. All processors can access the same memory space, but this might cause conflicts if multiple processors try to read/write the same memory cell at the same time. Therefore, the PRAM is divided into three subcategories Exclusive-Read-Exclusive-Write (EREW), Concurrent-Read-Exclusive-Write (CREW), and Concurrent-Read-Concurrent-Write (CRCW). Where the former is most restricted and the latter least restricted. On a EREW PRAM one must guarantee that no two processors will try to access the same memory cell at the same time. On a CREW PRAM two (or more) processors may read but not write the same memory cell at the same time. On a CRCW PRAM processors may both read and write the same memory cells at the same time. As writing to the same memory cell may cause the outcome to be ambiguous there exist different conflict resolution techniques – we will not go into the details of these as they are not relevant for this dissertation.

Normally we analyze algorithms in the RAM model by the number of operations performed – which we call the time used (as all operations are assumed to take the same time) – and the amount of space used. In the PRAM model we measure three parameters; time, work and space. Here, time is the number of operations performed on the processor that finishes the latest. The work is the total number of operations performed on all processors. The space usage is the same as normal.

Finally, in Chapter 6 we do experiments on a real modern processor where we see the unit cost assumption for operations in the RAM-model does not fully resemble the real world. In that chapter, we also consider the memory caches of modern processors to fully optimize our implementation.

**Notation** Since each chapter is an (almost) verbatim copy of a paper, there are small differences in the notation used throughout this dissertation, but mostly it will be the same. The notation used in each chapter, will be introduced therein. I will keep the notation to a minimum in this introduction, but do however need the following:

Let  $T$  be a string of characters from an alphabet  $\Sigma$  of length  $|T| = N$ . Let  $T[i]$  be the character on position  $i$  in  $T$  where  $1 \leq i \leq N$ . Let  $T[i, j]$  denote the substring of  $T$  from position  $i$  to position  $j$  (both positions included). Let a run of a character  $\alpha$  be a maximal substring of  $T$  that only consists of the character  $\alpha$ . Let  $G$  be a context-free grammar that produces  $T$  and has size  $n$ . Let  $z$  be the size of the LZ77-parse of a string. Let  $P$  be a pattern string of length  $m$ .

### 1.3 The Lempel-Ziv Family and Friends

In this section I will introduce a number of different compression schemes for highly repetitive data. The main idea of schemes in the Lempel-Ziv family is to avoid storing the same substring multiple times by instead referring to an earlier occurrence of the substring in the text. The LZ77 [139] scheme is the most powerful compression-wise in this family, and hence it is one of the most popular measures used for space analysis of compressed data structures.

**LZ77** The LZ77-parse of a string  $T$  is a sequence of  $z$  *phrases*. A phrase is defined by a source position  $s_i$ , a length  $l_i$  and a character  $\alpha_i$ . A phrase expands to the string  $T[s_i, s_i + l_i - 1]\alpha_i$  (if  $l_i = 0$  then it simply expands to  $\alpha_i$ ). The concatenation of the expansion of all  $z$  phrases is  $T$ . The  $i^{th}$  phrase starts on position  $u_i$  in  $T$  where  $u_1 = 1$  and  $u_i = u_{i-1} + l_{i-1} + 1$ . The substring  $T[s_i, s_i + l_i - 1]$  is called the source of the  $i^{th}$  phrase. The source of a phrase must appear to the left of the phrase itself, ie.  $s_i + l_i < u_i$ .

Text:            abaabaacabaabaac

LZ77-parse:     $(0, 0, a)(0, 0, b)(1, 1, a)(2, 3, c)(1, 8, -)$

The LZ77-parse is obtained by reading  $T$  from left-to-right while constructing one new phrase at a time. The  $i^{th}$  phrase is constructed by finding the largest  $l_i \geq 0$  such that  $T[u_i, u_i + l_i - 1]$  occurs in  $T[1, u_i - 1]$  at some position  $s_i$  and then  $\alpha_i = T[u_i + l_i]$ . This construction yields the smallest number of phrases that fulfill the above requirements.

There exists a so called self-referential version of the LZ77-parse that allows a phrase to overlap with its source, here the constraint  $s_i + l_i < u_i$  is relaxed to  $s_i < u_i$  which still guarantees the phrases expand to a well-defined string.

**Variants** There have been proposed many variants of the LZ77 compression scheme that put further restrictions on the parse. Typically in order to reduce the time it takes to compute the parse, or to make it easier to do computations on the compressed data, at the cost of a (possibly) slightly worse compression ratio.

**Bidirectional Parse** A more powerful compression scheme is the so called bidirectional parse. Again, it is almost the same as LZ77, but without the restriction that the source of a phrase must appear before the phrase itself. When removing this restriction, it is possible to construct a parse with circular references which results in a parse that does not expand to a single well-defined string. The only restriction on the phrases in a bidirectional parse is that the expansion of all phrases must produce a single unique string. Unfortunately, it is NP-hard to construct the smallest possible bidirectional parse of a string.

**Relative Compression** In relative compression, one is given a reference text  $R$  besides the text  $T$  that should be compressed. Both the compressor and decompressor are assumed to know  $R$ . A relative compression of  $T$  is basically the same as the bidirectional parse of  $T$  with two changes; all sources are now arbitrary substrings of  $R$  (instead of substrings of  $T$ ) and the character  $\alpha_i$  is normally omitted from a phrase since it is not needed as it is assumed  $R$  contains at least one occurrence of each character in the alphabet.

Reference:	abac
Text:	abaabaacabaabaac
RC-parse:	$(1, 3)(1, 3)(3, 2)(1, 3)(1, 3)(3, 2)$

As an application of relative compression consider for instance the human DNA. As already stated, the DNA of two different humans are almost identical. Thus it makes sense to store one copy of the human DNA, and then compress all others DNA relative to that.

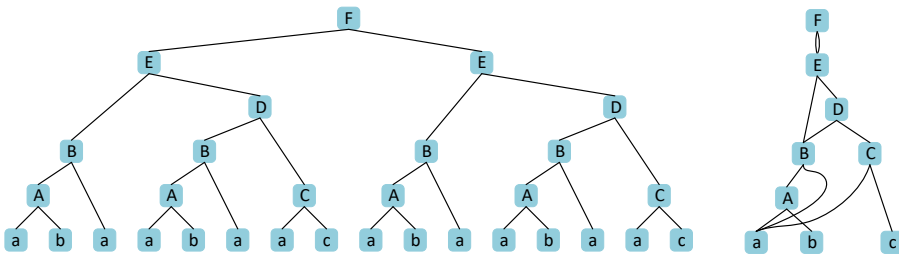
## 1.4 Context-Free Grammars and SLPs

In this dissertation we restrict our focus to context-free grammars. A grammar consists of *productions* and *terminals*. A production maps to a sequence

of productions and/or terminals. A terminal is simply a symbol from the alphabet. An example of a grammar is;  $a, b$  and  $c$  are the terminals and  $A \rightarrow ab, B \rightarrow Aa, C \rightarrow ac, D \rightarrow BC, E \rightarrow BD, F \rightarrow EE$  are the productions (note we often use capital letters for productions and non-capital letters for terminals). A production is said to produce/generate the unique string obtained by repeatedly replacing all productions on the right-side with the string they produce. In our example  $F$  produces  $abaabaacabaabaac$  ( $F \rightarrow EE \rightarrow BDBD \rightarrow AaAaCAaAaC \rightarrow abaabaacabaabaac$ ). In a grammar one production is chosen as the *start production*, in the example it is  $F$ . A grammar is said to generate/produce the string that the start production generates. The size of a grammar is the total number of productions/terminals on the right-hand side of all productions, in the example the size is 12.

**Parse Trees and DAGs** One can view a grammar  $G$  as a (parse-)tree. All nodes in this tree correspond to either a production or a terminal in  $G$ . The root of the tree corresponds to the start production. Nodes corresponding to productions have a child for each production/terminal on the production's right-side. Nodes corresponding to terminals have no children. Thus all internal nodes correspond to productions and all leaves correspond to terminals. A grammar is said to be balanced if its parse tree is balanced.

A directed acyclic graph, DAG, can be constructed from the parse tree of a grammar by replacing all identical subtrees in the tree by a single such subtree that has an incoming-edge from all the places it occurs in the parse tree. The size of this DAG (the number of edges) is exactly the same as the size of the grammar. In the DAG there is a one-to-one correspondence between productions/terminals and internal nodes/leaves. Thus this DAG can be considered as the compressed parse tree.



The parse tree of the above grammar to the left. The corresponding DAG to the right.

In all of the following we will view grammars in all these ways interchangeably.



**Run-Length Grammars** In run-length grammars an additional type of productions is introduced. A production of the form  $P \rightarrow R^n$  produces the string that is the string  $R$  produces repeated  $n$  times. Ie. a run of characters/productions can now be produced by a single production. The size of such a production is said to be  $O(1)$  (since we can easily store a reference to  $R$  and the integer  $n$  in  $O(1)$  words). A run-length grammar can be turned in to a normal grammar with at most logarithmic blowup since a run-length production can be replaced by a logarithmic number of productions that produces the run.

**SLPs** A *straight line program*, SLP, is basically a restricted grammar where productions are only allowed to either have two productions or one terminal on the right-side. All grammars can be translated to an SLP of roughly the same size thus we often consider this restricted variant for simplicity.

**The Smallest Grammar** When grammars are used for compressing a string  $T$ , we are interested in finding the smallest possible grammar that produces  $T$ . Unfortunately, the problem of finding the smallest such grammar is known to be NP-hard, so it is not feasible to construct the smallest possible grammar in practice. Instead we are happy with (good) approximations hereof.

In the following, we will sketch the ideas behind a number of different grammar construction algorithms that we find relevant.

**Simple Pairing** Given a string  $T$  of length  $N$  (assume for simplicity  $N$  is a power of two), look at pairs of neighboring symbols  $T[1, 2], T[3, 4], \dots$ . For each distinct pair of symbols introduce a new production that produces these two symbols, and replace all pairs with the corresponding production in  $T$ . Repeat this until  $T$  only consists of a single production, let this production be the root of the grammar.

Text	Productions
abaabaacabaabaac	$A \rightarrow ab, B \rightarrow aa, C \rightarrow ba, D \rightarrow ac$
ABCDABCD	$E \rightarrow AB, F \rightarrow CD$
EF EF	$G \rightarrow EF$
GG	$H \rightarrow GG$
H	

This process produces a balanced SLP since it takes  $O(\log n)$  rounds before  $T$  is reduced to a single symbol, and the height of the SLP only increases by one in each iteration. In fact, this gives a parse tree that is a complete binary tree where the leaves correspond to the characters of  $T$ .

**RePair** All pairs of symbols  $T[1, 2], T[2, 3], T[3, 4], \dots$  are considered and the most frequent is selected. A production is created for this pair of symbols and then all occurrences of these pairs are replaced by that production in  $T$ . Like before, this is repeated until  $T$  consists of a single production that becomes the root.

Text	Productions
abaabaacabaabaac	$A \rightarrow ab$
AaAaacAaAaac	$B \rightarrow Aa$
BBacBBac	$C \rightarrow ac$
BBCBBC	$D \rightarrow BC$
BDBD	$E \rightarrow BD$
EE	$F \rightarrow EE$
F	

This generates a possibly unbalanced SLP that might have be as high as  $\Theta(\sqrt{N})$ . The creators of this algorithm showed how this construction can be done in linear time [94]. This is not trivial as one has to efficiently find and replace the most frequent pair in the text in each iteration (the trivial way to do this takes  $O(N^2)$  time).

**Signature Grammars** First find all runs of a character  $\alpha$  in  $T$  and replace these by a run-length production. After doing this, no two neighboring characters are the same. Now create a random permutation of the characters present in  $T$ , let  $\phi(\alpha)$  be the index of  $\alpha$  in this permutation. Look at the sequence  $\phi(T[1]), \phi(T[2]), \phi(T[3]), \dots, \phi(T[N])$ . Let  $m_1, m_2, \dots$  denote the positions of the local minimas in this sequence. Consider all the substrings  $T[m_0 + 1, m_1], T[m_1, m_2], T[m_2 + 1, m_3], \dots$ . For each distinct substring in this set, create a production in the grammar that generates that substring, and

replace all occurrences of this substring with this new production. As in the previous algorithms, repeat these two steps until only one production is left in  $T$  and let this be the root of  $T$ .

Text	Productions
abaabaacabaabaac	$A \rightarrow a^2$
<u>a</u> <u>b</u> A <u>b</u> A <u>c</u> <u>a</u> <u>b</u> A <u>c</u>	$B \rightarrow bA, C \rightarrow ca$
aBBCBBc	$D \rightarrow B^2$
<u>a</u> D <u>C</u> D <u>c</u>	$E \rightarrow aD, F \rightarrow CD$
<u>a</u> EF	$G \rightarrow aEF$
G	

The random permutation of the symbols used above is BcCbAaDEF. The local minima are underlined.

First of all, this process produces a context-free run-length grammar and not an SLP. By iterating this process a constant number of times in expectation, we obtain a grammar that has size  $O(z \lg(N/z))$  (where  $z$  is the size of the LZ77-parse of  $T$ ) and where each production has size at most  $O(\lg N)$ . This is not trivial to see, but will be covered in Chapter 3. The length of  $T$  is at least halved in each iteration of this process, thus the height of the resulting grammar is  $O(\lg N)$ .

**Alternative Constructions** There exists a dozen of other ways to construct SLPs/grammars. One such way is to construct the LZ77-parse and then construct an SLP based on this. This was the first way to obtain the  $O(z \lg(N/z))$  size bound for a SLP [117]. Another approach is called locally consistent parsing where the idea is to ensure identical substrings are mostly generated by the same productions. This is also the idea used for our proposed signature grammars.

**Incremental Construction** If the string we want to compress is too big to fit in memory then it would be beneficial if we could incrementally construct the grammar by reading  $T$  from left-to-right character-by-character once. The RePair algorithm relies on global information about the entire string since it needs the frequency of all pairs in the entire string. This makes it unsuitable for incremental construction. On the other hand simple pairing and signature

grammars only use local information to construct the grammar, so they are good candidates for incremental construction. It is indeed possible to make both of these algorithms incremental in a simple way.

Consider the simple pairing algorithm. Let  $B_i$  be buffer strings that are initially empty for  $1 \leq i \leq \log N + 1$ . Read the characters from  $T$  one-by-one by appending them to  $B_1$ . When a buffer string  $B_i$  has length two, see if there exists a production that generates  $B_i$ , otherwise create a new production that does, and then clear  $B_i$  and append the production to  $B_{i+1}$ . In the end the root production will be in  $B_{\log N + 1}$ . The same idea can be applied for signature grammars, except new productions should be created when a buffer either contains a run of characters or a local minima with respect to the  $\phi$ -function.

This means we can construct these grammars in (poly)logarithmic working space in addition to the size of the grammar. This construction technique is also very fast in practice.

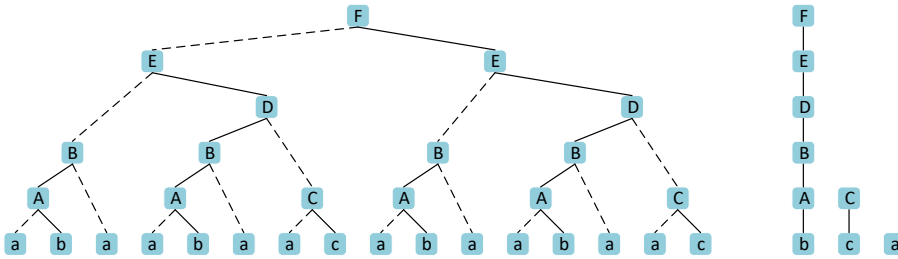
**Comparison of the Construction Algorithms** There are a number of interesting parameters when it comes to picking the best way to construct a grammar. One is the complexity of constructing the grammar. Even though all of the above grammars can be constructed in linear time in  $N$ , RePair is significantly slower to compute in practice. On the other hand, RePair is known to be one of the algorithms that produces the smallest grammars in practice. Intuitively, it makes sense that RePair produces small grammars as it repeatedly picks the production that can be reused the most times. Unfortunately, we do not have practical data for signature grammars, but intuitively it should also produce relatively small grammars as two identical substrings of length  $l$  will be generated by productions with at most  $O(\log l)$  differences. And in this case, we also have a guarantee it is at most a factor  $O(\log(N/z))$  larger than the smallest possible grammar since  $z$  is a lower-bound for the smallest grammar. Simple pairing is mainly included above as a simple introduction, as it may produce really bad grammars. Consider for instance the string `abcdefgabcdefg`, it will generate a grammar where no productions are used more than once, because the first occurrence of `ab` is paired as `ab` but the second occurrence of `ab` is paired as `ga` and `bc`, and similarly for the other pairs. Finally, simple pairing and the signature construction algorithm produce balanced grammars whereas RePair can produce unbalanced grammars (though, in practice they are in many cases not).

## 1.5 Heavy Paths

In this section, we will sketch what a *heavy path decomposition* is and how it can be used to perform random access in an SLP. This decomposition has been used to obtain  $O(\log N)$  random access time in (unbalanced) SLPs in  $O(n)$  space [18]. The idea behind this kind of decomposition of a SLP is a key component in Chapter 2 where we introduce new slightly different decompositions.

In the following, we aim to give the idea behind a simplified version of random access that takes  $O(\log N \log \log N)$  instead of  $O(\log N)$  time. But before that, we will introduce a simple solution that takes  $O(h)$  time where  $h$  is the height of the SLP. The idea is simply to traverse the grammar from the root to the correct leaf/terminal. The only thing needed besides the productions in the SLP is the size of the string each production produces. This takes  $O(n)$  space. Initially, let  $v$  be the root of the SLP, and let  $i$  be the position in  $T$  to access. If  $i \leq |\text{left}(v)|$  then set  $v = \text{left}(v)$  otherwise set  $v = \text{right}(v)$  and  $i = i - |\text{left}(v)|$  (where  $|\text{left}(v)|$  is the size of the string  $v$ 's left child produces, and similarly for  $|\text{right}(v)|$ ). When  $v$  is a terminal, report the character of  $v$ .

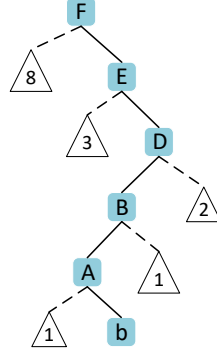
This algorithm is fine for balanced SLPs, but if there are long root-to-leaf paths then access on these will be slow. Thus, we need some way to efficiently traverse long root-to-leaf paths without looking on all nodes on the path. This is what the heavy path decomposition can help us with.



To the left the parse-tree of the grammar with the heavy edges marked by solid lines. To the right the corresponding heavy path forest.

Define the heavy child of a node to be the child producing the longest string (in our example we pick the right child in case of a tie). Define an edge to be heavy if it ends in a heavy child. Consider all the heavy edges in the parse tree. These form a decomposition of the tree into a disjoint set of paths (see illustration above). We call a path in this set a heavy path. A key property of a heavy path decomposition is that any root-to-leaf path in the original tree intersects

at most  $\log N$  distinct heavy paths. Now the idea is to find an efficient way to traverse a heavy path. Also note all heavy paths end in a leaf of the tree.



The heavy path starting in the root with subtrees hanging to the left and right. The sequence of sizes of subtrees hanging to the left is here 8, 3, 1.

Assume we have traversed the parse tree until some node  $v$  (initially  $v$  is the root), and we want to access the character at position  $i$  in the text  $v$  produces. We know there is a heavy path going from  $v$  and down to some leaf  $u$  in the subtree of  $v$ . Let  $i_u$  be the index of the character  $u$  corresponds to in the text  $v$  produces. If  $i < i_u$  then the leaf we are trying to navigate to must be somewhere to the left of the heavy path. If  $i = i_u$  then  $u$  is the leaf we want to navigate to, thus we are done. Finally, the case  $i > i_u$  is symmetric to the first case, namely the leaf we want to navigate must be to the right of the heavy path. Let us now focus on the first case. Consider the heavy paths, a number of subtrees hang to the left of this path. We need to decide which of these subtrees contain the leaf we are looking for. Let  $s_1, s_2, \dots, s_j$  be the size of the subtrees hanging to the left. We want to navigate the  $k^{th}$  subtree hanging to the left such that  $\sum_{l=1}^{k-1} s_l < i \leq \sum_{l=1}^k s_l$ . Furthermore, we want to access the  $i - \sum_{l=1}^{k-1} s_l$  leaf in this subtree. By repeated using this procedure to navigate a heavy path starting from the root, we end up in the leaf we are looking for.

Finding the correct subtree among the subtrees that hang to the left of the heavy path is basically the predecessor problem. We could use a predecessor data structure to answer this question for each of the heavy paths. Such structures can answer predecessor queries in  $O(\log \log N)$  time. Since we have to go through at most  $\log N$  heavy paths on our way from the root to a leaf, the overall time is  $O(\log N \log \log N)$ .

In the above, we did not consider the space usage of the solution. Instead of explicitly storing a predecessor data structure for each of these heavy paths,

we can do a reduction to the weighted ancestor problem in trees. The first observation to do this is that if we only look at the heavy edges in the DAG, they form a forest of trees rooted in the terminals/leaves, ie. they are up-side-down compared to the parse tree. All the predecessor data structures can then be replaced by a weighted ancestor data structure on this forest. We will not give the details of this reduction here, but simply conclude that this weighted ancestor data structure uses  $O(n)$  space and still answers queries in  $O(\log \log N)$  time.

## 1.6 On Chapter 2: Finger Search in Grammar-Compressed Strings

In this chapter, we study a variation of one of the most fundamental problems on compressed data, namely the random access problem. In the compressed random access problem one is to construct a compressed data structure given a string  $T$  of length  $N$  that can answer access queries, which is to return the character of a given position  $i$  in  $T$  efficiently – independent on where  $i$  is. In the finger search problem, one first puts his/her virtual finger at some position  $f$  in the text, and must then answer access queries efficiently as a function of the distance between the finger and the access query, ie.  $|f - i|$ . That is, access queries close to the finger should be answered faster than queries far from the finger.

We study this problem for SLPs. We are the first to study this problem in this context, but obviously solutions to the random access problem are also (inefficient) solutions to the finger search problem. As already discussed, the random access problem has been solved for arbitrary SLPs in  $O(n)$  space and  $O(\log N)$  time where  $n$  is the size of the SLP [18] thus our finger access should be better than  $O(\log N)$  time in order to be relevant. On the other hand, a solution to the finger access problem is also a solution to the random access problem, so a solution better than  $O(\log N)$  when the finger and access position are far apart would imply an improvement of the random access problem.

**Our contributions** Our main result is a solution where the finger can be placed in  $O(\log N)$  time and access queries can be answered in  $O(\log |f - i|)$  time using  $O(n)$  space. We call this the *static* version of the problem. We also present a *dynamic* version where the finger itself can be moved efficiently. In this setting, we can move the finger from  $f_{old}$  to  $f_{new}$  in  $O(\log |f_{old} - f_{new}| + \log \log N)$  time and access now takes  $O(\log |f - i| + \log \log N)$  time. The complexities for

placing the finger and space usage is the same.

As an application of the finger search solution, we show how to use it to perform *longest common extension* queries. In this problem, you are given two positions  $i$  and  $j$  in  $T$  and want to know the largest  $\ell$  such that  $T[i, i + \ell - 1] = T[j, j + \ell - 1]$ . At the time of publication, the best known solution in  $O(n)$  space ran in  $O(\log N \log \ell)$  time which we improved to  $O(\log N + \log^2 \ell)$  time – ie. an improvement for small values of  $\ell$ . Others have improved on this problem since we got our results, but in slightly different settings where the space usage is not directly dependent on the size of the input SLP. For instance in [127] the author gave an  $O(\log N)$  time solution using  $O(z \log(N/z))$  space where  $z$  is the size of the LZ77-parse of  $T$ .

**What are the challenges of this problem?** First of all, the solution should work on arbitrary SLPs and not only on balanced SLPs. We have already seen the random access problem is significantly simpler to solve in the balanced case thus it is likely also to be the case for this problem. Normally, access queries are solved by traversing a root-to-leaf path in the SLP. If we want to access two positions close to each other, one might think we do not have to traverse the entire root-to-leaf path twice. This is indeed often the case, but there exists neighboring positions in  $T$  that does not share any edges on their root-to-leaf paths. As we want to provide a worst-case bound, we have to handle this case in some other way.

The main technique used for this result is new heavy path like decompositions of the SLP. I will not go into details about this here, as I have already introduced the concept of heavy paths.

**A  $\rightsquigarrow$   $\hat{A}$**  This was the first problem I worked on when I started my PhD studies. This was a good path into the world of SLPs (and context-free grammars in general) as it required me to read and understand the details of the solutions to the random access problem. Many of these techniques are widely applicable when working with (unbalanced) SLPs. Originally, when we researched this problem, we also came up with a solution that provided constant time access in the SLP at the cost of space dependent on  $N$ , ie. independent on how well  $T$  compresses. Using this we could also answer LCE queries in constant time. Unfortunately for us, it turned out others had recently found a (slightly different) solution obtaining similar results.



## 1.7 On Chapter 3: Compressed Indexing with Signature Grammars

In this chapter, we study another fundamental string problem on compressed text. Given a string  $T$  of length  $n$  construct a compressed data structure that can efficiently answer if and where a pattern  $P$  of length  $m$  occurs in  $T$ . This problem has many well-known solutions in the uncompressed case for instance suffix trees solve the problem in optimal  $O(|P|)$  time using  $O(n)$  space. Also in the compressed case there exist many solutions to this problem based on different kind of compression techniques. In this chapter, we restrict our focus to solutions where the space usage is bounded by the size  $z$  of the LZ77-parse of  $T$ . In  $O(z \lg(n/z))$  space Gagie et al. [55, 56] showed how to solve queries in  $O(m \lg m + \text{occ} \lg \lg n)$  time. Bille et al. [16] gave a number of trade-offs to this problem where one solution obtained optimal  $O(m)$  search time (disregarding the reporting time per occurrence) but using  $O(z \lg(n/z) \lg \lg z)$ .

**Our contributions** In this chapter, we show how to obtain better trade-offs for this problem. Like the previous results, the trade-offs depend on the size of the alphabet and the compressibility of  $T$ . For small alphabets or sufficiently compressible texts we obtain  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  query time and  $O(z \lg(n/z))$  space which improves the previously best known solutions. Most interestingly, the search time  $O(m)$  is optimal when disregarding the time required for reporting subsequent occurrences.

The main component of our solutions is the signature grammar construction previously described. It ensures multiple occurrences of the same substring in  $T$  are produced by almost the same set of productions. We combine this component with an already known technique for pattern matching in compressed texts, namely a reduction to 2D range reporting.

**Future directions** There are two "obvious" things that would be nice to improve – we already tried hard to do that so I do not currently have any good ideas on how though. First of all, it would be nice to have the  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  time for all alphabet sizes/compressions. Secondly, it would be nice to reduce the time per occurrence – ideally all the way to  $O(1)$  time per occurrence. Both of these bounds have in fact been shown in a different compression scheme for highly repetitive texts; namely compression based on the Burrows–Wheeler transform [59] (not that this guarantees it is possible with our space bounds).

I also think it would be interesting to do an implementation of the basic ideas

in this paper. First of all to determine if the suggested grammar construction algorithm is competitive to other grammar construction algorithms. And if this is the case, it would be interesting to add pattern matching on top of it – most likely with some simplifications of the techniques/data structures used in the chapter.

**0  $\rightsquigarrow$  3** The ideas this chapter is based on actually came while working on another problem. We were trying to solve the (uncompressed) dynamic substring concatenation problem where one is given a text  $T$  and the indices of two substrings  $T[i, j]$  and  $T[i', j']$  of  $T$  and must answer if the concatenation  $T[i, j]T[i', j']$  occurs anywhere in  $T$ . Furthermore, in this dynamic version, the text  $T$  is subject to character replacements. During this, we ran in to a paper by Mehlhorn et al. [101] that introduced us to the idea of signatures. We then realized that several papers using similar techniques existed that solved dynamic pattern matching, compressed pattern matching, and similar. Thus we decided to explore how these ideas could be combined with the techniques from [16] that my co-author had recently been working on.

This problem was the last I worked on during my PhD. For this reason, the paper this chapter is based on has not yet been peer reviewed (as the only paper presented in this dissertation). It is under submission for LATIN17 as of writing this, but notification is unfortunately first shortly after my deadline.

## **1.8 On Chapter 4: Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation**

Before I introduce the contents of this chapter, I should emphasize that I was not a co-author of the first versions of this paper, and thus did not participate in obtaining many of the results this chapter is based on. I was first involved after a journal reviewer pointed out a possible mistake in the substring concatenation part of the paper. My main contribution was to help find a solution to this problem that fortunately turned out to make the original claims of the paper correct. I have chosen to include this chapter in this dissertation despite this since it fits the topic, I have spent quite sometime working on the substring concatenation problem, and I have been co-supervising students implementing some of the basic ideas from this paper so I know it well.

Here we study the problem of efficiently maintaining a relative compression of a

dynamic text. Given a static reference text  $R$  of length  $r$  and a dynamic text  $T$  of length  $N$ , we compress  $T$  by referring to a sequence of  $n$  substrings of  $R$  which concatenation spells out  $T$  as described earlier in the introduction. To do this, we also look at the problems dynamic partial sums and substring concatenation. Since next chapter is on partial sums and substring concatenation was just introduced in the last section, we will skip them here. However, we should note the dynamic partial sums described in this chapter supports more operations than in the next chapter.

**Our contributions** We present two different solutions to the substring concatenation problem that result in two different bounds for the overall compression problem. These bounds are either  $O(n+r)$  space and  $O(\frac{\log n}{\log \log n} + \log \log r)$  time per operation (access/update), or  $O(n + r \log^\epsilon r)$  space and  $O(\frac{\log n}{\log \log n})$  time per operation where  $\epsilon > 0$  is a constant. The first provides optimal space but only near optimal time (given this compression scheme), and the second provides optimal time but not space.

To solve the substring concatenation problem we use a slightly modified version of a 1D range emptiness data structure by Belazzougui et al. [10] that is based on a weak prefix search data structure by Goswami et al. [65]. A combination of this and a heavy path decomposition of the suffix tree of the text gives the second of the two above bounds. Let us take a further look at the first of these data structures. Given a set  $A$  of integers in the range  $[0, r]$  it can report the index of the first integer in  $A$  within a given subrange of  $[0, r]$  (the index of the integer if the set was sorted). However, if no such integer exists it may report an arbitrary index. In other words, it may report false-positives. It was this problem with false-positives that was not addressed in the first versions of this paper. As this data structure uses  $|A| \log^\epsilon r$  bits of space for any constant  $\epsilon > 0$ , it is clear from the information theoretic lower-bound that it must return false-positives. If this was not the case, then  $A$  could be reconstructed from this data structure which contradicts it takes up to  $|A| \log r$  bits to represent this set. The solution to this problem was to do a verification step after querying this data structure.

**S  $\rightsquigarrow$  T** During my external stay, I worked quite some time on the substring concatenation problem (independently of this chapter). We had a hope that the techniques from a recent paper [62] that solves the weighted ancestor problem in suffix trees in constant time could be used to solve the substring concatenation problem in  $O(1)$  time as well. Many ideas came across, but in the end none of them turned out to work. I still believe (or hope maybe?) it is possible somehow, and hopefully somebody will show that sometime or a lower bound.

However, after all, this was probably the main reason I was able to help solving the issue in this paper.

## 1.9 On Chapter 5: Succinct Partial Sums and Fenwick Trees

We study a problem that is not directly related to strings, namely the partial sums problem. In this problem, one is given an array  $A$  of  $n$  integers and must then be able to handle update, sum and search queries. That is, update the integer at some position in  $A$  to something else, return the sum of a prefix of the numbers in  $A$ , find the smallest  $j$  such that the sum of the integers  $A[1, j]$  is at least some given number. Even though this problem is not directly related to strings, it is being used in many data structures for strings (for instance Chapter 4). We study the succinct version of the problem where the space used is supposed to be very close to the theoretical minimum needed to store the data. If  $A$  consists of  $k$ -bit integers this is  $nk$  bits.

**Our contributions** We present two modified versions of the original Fenwick tree that use less space and also have better bounds for some of the operations. The first version is the first data structure to use only  $nk + o(k)$  bits of space while supporting sum, update, and search in  $O(\log_b n)$ ,  $O(\log_b n)$ , and  $O(\log n)$  time respectively where  $2 \leq b \leq \log^{O(1)} n$ . The second version uses slightly more space, namely  $nk + o(nk)$  bits of space, but then it supports faster sum and update queries.

Most of the bounds we obtain in this chapter have in fact been found by others earlier, however the data structures that achieve them are relatively complicated. The idea behind this result was to build on top of the simple Fenwick tree in order to get a relatively simple and practical solution that obtain bounds comparable to those already achieved.

**Future directions** As we have claimed that the results obtained in this chapter are very practical, it would be very nice to actually present an implementation of it to support this claim.

**0  $\rightsquigarrow$  11** I first encountered Fenwick trees long before my PhD studies started, since these are very popular in programming competitions because they are

extremely simple to implement. This paper started after a discussion involving Fenwick trees and their space usage. As the original Fenwick tree is simply implemented as an array of integers of size  $n$ , it is easy to think it does not use more space than the  $n$  input integers. However, in the Fenwick tree array each entry corresponds to the sum of many input integers, thus these numbers may require more bits to represent than the input numbers. In fact, if the input are  $k$ -bit integers, then the Fenwick tree array must be able to store  $(k + \lg n)$ -bit integers. This discussion then made us think about how to optimize the space usage of Fenwick trees which in the end resulted in the results presented in this chapter.

## 1.10 On Chapter 6: Fast Dynamic Arrays

This chapter is the first and only experimental in this dissertation. In this chapter, we study the problem of maintaining a dynamic sequence of elements – this could for instance be a string of characters. We are concerned with the operations; access, insertion, deletion and update. Which respectively reports the character on a given position, inserts a new character at an arbitrary position in the string (and thereby moving all subsequent characters one position), deletes a character on an arbitrary position and change a character on a given position to something else.

A dozen more or less practical solutions exist to this problem, most notably all kinds of balanced trees solve all the operations in  $O(\lg n)$  time. There even exist solutions that match the lower bound of  $\Omega(\lg n / \lg \lg n)$ . This lower bound applies when all operations must be equally fast. In the extreme case, where access is required to run in constant time, a simple solution is to use an array (growing/shrinking using the doubling strategy) and then move all characters in this array when a character is either inserted or deleted. Access and update operations are easily done in  $O(1)$  time this way, but insertion/deletion take  $O(n)$  time when elements must be shifted to make room for the new element/fill out the deleted position. We introduce the data structure tiered vector that solves the dynamic array problem. It was invented by Goodrich and Kloss in [64]. It allows access and updates in  $O(1)$  time and insertion/deletion in  $O(n^\epsilon)$  for any constant  $\epsilon > 0$ . A few groups have implemented this data structure including Goodrich and Kloss and performed experiments on these implementations. However, all these implementations have been significantly restricted such that they only work for  $\epsilon = 1/2$  giving  $O(\sqrt{n})$  insertion/deletion. The special case when  $\epsilon = 1/2$  gives a very nice and simple data structure, but we expected it not to perform well on big but still practical data sets.

**Our contributions** We give the first implementation of the tiered vector data structure that works for an arbitrarily small constant  $\epsilon > 0$ . We compare different versions of the implementation to obtain the best possible and then afterwards to some built-in data structures in C++. We get that for sequences of  $10^8$  elements, the best practical value for  $\epsilon$  is  $1/4$ , ie. it pays off to not only implement the simpler structure for  $\epsilon = 1/2$ .

We describe different memory layouts that reduce the number of memory probes needed by the operations compared to the original tiered vector. Among this, we consider memory layouts that utilizes that cache-lines are bigger than single words and layouts that utilizes the cache hierarchy in modern processors. All considerations that are normally ignored when we work in the RAM-model where we assume unit cost operations.

Besides these more general optimizations, we also spent much time on optimizing the code to minimize the number of instructions needed per operation. This is slightly difficult to achieve when the implementation should still be parameterized in  $\epsilon$ . To do this, we used C++ templates in a slightly unusual way to do what we call *template recursion*. The idea here is that the  $\epsilon$  parameter can be specified at compile-time, and then everything regarding this parameter can be optimized away by the compiler already at compile-time, so the final code is as optimized as if the code was not parameterized.

We show that this implementation of tiered vectors is at least competitive on all operations compared with the built-in vector and multiset implementations and even significantly faster in many cases.

**Future directions** Our implementation is not made with the focus of being production ready. I think our results are so promising, it would make sense to do a production ready version of the code. Ie. one that supports the normal C++ container interfaces, can dynamically grow/shrink, etc.

**A  $\rightsquigarrow$  Z** I have always enjoyed implementation and I also like to see stuff that works well in practice. Thus, when looking for a problem to work on this time the focus was not as much on the topic as on finding something interesting that would make sense to implement. It was interesting to get a practical feeling with some of the differences between real modern processors and the RAM model. For instance that the practical running time often depended more on what memory was accessed than the number of operations.

Even though we work in theoretical computer science, I personally think it is

a good thing to implement and experiment with the stuff we are working on occasionally, so our work/focus does not diverge too far away from what is actually meaningful in practice.

## 1.11 On Chapter 7: Parallel Lookups in String Indexes

In this chapter we look at the same fundamental string problem as in Chapter 3, namely searching for a pattern  $P$  of length  $m$  in a text  $T$  of length  $n$ . However in this chapter, we study the problem in a uncompressed version but in the parallel random access machine model (the PRAM model). We study the variant where one is to construct a data structure based on  $T$  and must then be able to answer pattern matching queries for a pattern  $P$  efficiently.

This problem was only studied recently even though many other similar string problems were already well-studied in the PRAM model back in the end of the last century. This problem was first studied in [79] where they show an index that supports queries in  $O(m)$  work and  $O(\log m)$  time on a CREW PRAM but in  $O(n^2)$  space. These bounds are work-time-optimal due to a lower bound given in [31], but the space usage is unnecessarily big. They also show an index of size  $O(n \log n)$  that answers in optimal  $O(\log m)$  time but  $O(m \log m)$  work. In [46] they improve this latter result. In  $n + o(n)$  space they can answer queries in  $O(\log \log m \log \log n + \log m)$  time and  $O(m + \min(m, \log n)(\log m + \log \log m \log \log n))$  work on a CREW PRAM.

**Our contributions** We show how to solve this problem in  $O(m)$  work,  $O(\log m)$  time and  $O(n)$  space on a EREW PRAM. Ie. our solution is optimal on all these parameters in this model. The only non-optimal part is the preprocessing that is  $O(\log n \log^* n)$  time and  $O(n)$  work w.h.p., ie. the preprocessing work/time is non-deterministic which could possibly be improved. Afterwards, we show a simpler way to solve this problem by parallelizing the normal way to search for a pattern in a suffix array that gives rise to  $O(\log n)$  time and  $O(m + \log n)$  work queries which is however worse. In particular for short patterns.

Our first solution is primarily based on hashing and the structure of the suffix tree of  $T$ . It is only composed of simple data structures that works well in practice and is in this sense practical. The second solution is even simpler in this sense, and does only require the suffix and lcp arrays thus it is practical also space-wise. However, there are discussions on how well the PRAM model

resembles real practical parallel systems. Another reason for not using it in practice is the use cases for this problem. Often the patterns we search for are relatively short, and thus the linear running time of the sequential solution is sufficient. Thus the primary reason for parallelizing this problem would be to handle many pattern lookups simultaneously, but this can easily be achieved by running the sequential algorithm in parallel for each pattern.

**MSc  $\rightsquigarrow$  PhD** These results were made while I was on my external stay in the US where Martín Farach-Colton hosted me. The main motivation for studying this problem was Martin short before my arrival had been presented some of the earlier results mentioned above, and believed we could improve on those. This was the first time I worked in the PRAM model, so I first had to understand the details of this model. It was interesting to learn how changing the model opened the possibility to use techniques that are not applicable in the RAM model (in which I normally work). Especially some of the tricks that are possible on the CRCW PRAM were a bit challenging to my normal assumptions about what is possible (unfortunately many of these tricks are probably not that practical).





## CHAPTER 2

# Finger Search in Grammar-Compressed Strings

---

Philip Bille<sup>†</sup>

Anders Roy Christiansen<sup>†</sup>

Patrick Hagge Cording<sup>†</sup>

Inge Li Gørtz<sup>†</sup>

<sup>†</sup> The Technical University of Denmark

### Abstract

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

Let  $n$  be the size the grammar, and let  $N$  be the size of the string. For the static variant we give a linear space representation that supports placing the finger in  $O(\log N)$  time and subsequently accessing in  $O(\log D)$

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

## 2.1 Introduction

Grammar-based compression, where one replaces a long string by a small context-free grammar that generates the string, is a simple and powerful paradigm that captures many popular compression schemes including the Lempel-Ziv family [134, 139, 140], Sequitur [106], Run-Length Encoding, Re-Pair [94], and many more [5–7, 54, 66, 88, 89, 121, 136]. All of these are or can be transformed into equivalent grammar-based compression schemes with little expansion [25, 116].

Given a grammar  $\mathcal{S}$  representing a string  $S$ , the *random access problem* is to compactly represent  $\mathcal{S}$  while supporting fast **access** queries, that is, given an index  $i$  in  $S$  to report  $S[i]$ . The random access problem is one of the most basic primitives for computation on grammar compressed strings, and solutions to the problem are a key component in a wide range of algorithms and data structures for grammar compressed strings [12, 14, 15, 18, 55–57, 76, 126, 128].

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

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

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

As an application of our results we also give a new solution to the *longest common extension problem* on grammar compressed strings [15, 76, 107]. Here, the goal is to compactly represent  $\mathcal{S}$  while supporting fast **lce** queries, that is, given a pair of indices  $i, j$  to compute the length of the longest common prefix of  $S[i, N]$  and  $S[j, N]$ . We give an  $O(n)$  space representation that answers queries in  $O(\log N + \log^2 \ell)$ , where  $\ell$  is the length of the longest common prefix. The best  $O(n)$  space solution for this problem uses  $O(\log N \log \ell)$  time, and hence our new bound is always at least as good and better whenever  $\ell = o(N^\epsilon)$ .

### 2.1.1 Related Work

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

**Random Access in Grammar Compressed Strings** First note that naively we can store  $S$  explicitly using  $O(N)$  space and report any character in constant time. Alternatively, we can compute and store the sizes of the strings derived by each grammar symbol in  $\mathcal{S}$  and use this to simulate a top-down search on the grammars derivation tree in constant time per node. This leads to an  $O(n)$  space representation using  $O(h)$  time, where  $h$  is the height of the grammar [63]. Improved succinct space representation of this solution are also known [27]. Bille et al. [18] gave a solution using  $O(n)$  and  $O(\log N)$  time, thus achieving a query time independent of the height of the grammar. Verbin and Yu [132] gave a near matching lower bound by showing that any solution using  $O(n \log^{O(1)} N)$  space must use  $\Omega(\log^{1-\epsilon} N)$  time. Hence, we cannot hope to obtain significantly faster query times within  $O(n)$  space. Finally, Belazzougui et al. [12] very recently showed that with superlinear space slightly faster query times are possible. Specifically, they gave a solution using  $O(n\tau \log_\tau N/n)$  space and  $O(\log_\tau N)$  time, where  $\tau$  is a trade-off parameter. For  $\tau = \log^\epsilon N$  this is  $O(n \log^\epsilon N)$  space and  $O(\log N / \log \log N)$  time. Practical solutions to this problem have been considered in [9, 58, 104].

The above solutions all generalize to support decompression of an arbitrary substring of length  $D$  in time  $O(t_{\text{access}} + D)$ , where  $t_{\text{access}}$  is the time for `access` (and even faster for small alphabets [12]). We can extend this to a simple solution to finger search (static and dynamic). The key idea is to implement `setfinger` as a random access and `access` and `movefinger` by decompressing or traversing, respectively, the part of the grammar in-between the two positions. This leads to a solution that uses  $O(t_{\text{access}})$  time for `setfinger` and  $O(D)$  time for `access` and `movefinger`.

Another closely related problem is the *bookmarking problem*, where a set of positions, called *bookmarks*, are given at preprocessing time and the goal is to support fast substring decompression from any bookmark in constant or near-constant time per decompressed character [32, 55]. In other words, bookmarking allows us to decompress a substring of length  $D$  in time  $O(D)$  if the substring crosses a bookmark. Hence, with bookmarking we can improve the  $O(t_{\text{access}} + D)$  time solution for substring decompression to  $O(D)$  whenever we know the positions of the substrings we want to decompress at preprocessing time. A key component in the current solutions to bookmarking is to trade-off the  $\Omega(D)$  time we need to pay to decompress and output the substring. Our goal is to support access without decompressing in  $o(D)$  time and hence this idea does not immediately apply to finger search.

**Finger Search** Finger search is a classic and well-studied concept in data structures, see e.g., [13, 19, 22, 36, 47, 68, 90, 100, 113, 120, 122] and the survey [21]. In this setting, the goal is to maintain a dynamic dictionary data structure such that searches have the finger search property. Classic textbook examples of efficient finger search dictionaries include splay trees, skip lists, and level linked trees. Given a comparison based dictionary with  $n$  elements, we can support optimal searching in  $O(\log n)$  time and finger searching in  $O(\log d)$  time, where  $d$  is the rank distance between the finger and the query [21]. Note the similarity to our compressed results that reduce an  $O(\log N)$  bound to  $O(\log D)$ .

### 2.1.2 Our results

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

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

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

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

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

**THEOREM 2.1** *Let  $S$  be a grammar of size  $n$  representing a string  $S$  of length  $N$ . Let  $f$  be the current position of the finger, and let  $D = |f - i|$  for some  $i$ . Using  $O(n)$  space we can support either:*

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

Compared to the previous best linear space solution, we improve the  $O(\log N)$  bound to  $O(\log D)$  for the static variant and to  $O(\log D + \log \log N)$  for the dynamic variant, while maintaining linear space. These are the first non-trivial solutions to the finger search problems. Moreover, the logarithmic bound in terms of  $D$  may be viewed as a natural grammar compressed analogue of the classic uncompressed finger search solutions. We note that Theorem 2.1 is straightforward to generalize to multiple fingers. Each additional finger can be set in  $O(\log N)$  time, uses  $O(\log N)$  additional space, and given any finger  $f$ , we can support **access( $i$ )** in  $O(\log D_f)$  time, where  $D_f = |f - i|$ .

### 2.1.3 Technical Overview

To obtain Theorem 2.1 we introduce several new techniques of independent interest. First, we consider a variant of the random access problem, which we call the *fringe access problem*. Here, the goal is to support fast access close to the beginning or end (the fringe) of a substring derived by a grammar symbol. We present an  $O(n)$  space representation that supports fringe access from any grammar symbol  $v$  in time  $O(\log D_v + \log \log N)$ , where  $D_v$  is the distance from the fringe in the string  $S(v)$  derived by  $v$  to the queried position. The key challenge is designing a data structure for efficient navigation in unbalanced grammars.

The main component in our solution to this problem is a new recursive decomposition. The decomposition resembles the classic van Emde Boas data structure [129], in the sense that we recursively partition the grammar into a

hierarchy of depth  $O(\log \log N)$  consisting of subgrammars generating strings of lengths  $N^{1/2}, N^{1/4}, N^{1/8}, \dots$ . We then show how to implement fringe access via predecessor queries on special paths produced by the decomposition. We cannot afford to explicitly store a predecessor data structure for each special path, however, using a technique due to Bille et al. [18], we can represent all the special paths compactly in a tree and instead implement the predecessor queries as weighted ancestor queries on the tree. This leads to an  $O(n)$  space solution with  $O(\log D_v + (\log \log N)^2)$  query time. Whenever  $D_v \geq 2^{(\log \log N)^2}$  this matches our desired bound of  $O(\log D_v + \log \log N)$ . To handle the case when  $D_v \leq 2^{(\log \log N)^2}$  we use an additional decomposition of the grammar and further reduce the problem to weighted ancestor queries on trees of small weighted height. Finally, we give an efficient solution to weighted ancestor for this specialized case that leads to our final result for fringe access.

Next, we use our fringe access result to obtain our solution to the static finger search problem. The key idea is to decompose the grammar into heavy paths as done by Bille et al. [18], which has the property that any root-to-leaf path in the directed acyclic graph representing the grammar consists of at most  $O(\log N)$  heavy paths. We then use this to compactly represent the finger as a sequence of the heavy paths. To implement **access**, we binary search the heavy paths in the finger to find an exit point on the finger, which we then use to find an appropriate node to apply our solution to fringe access on. Together with a few additional tricks this gives us Theorem 2.1(i).

Unfortunately, the above approach for the static finger search problem does not extend to the dynamic setting. The key issue is that even a tiny local change in the position of the finger can change  $\Theta(\log N)$  heavy paths in the representation of the finger, hence requiring at least  $\Omega(\log N)$  work to implement **movefinger**. To avoid this we give a new compact representation of the finger based on both heavy path and the special paths obtained from our van Emde Boas decomposition used in our fringe access data structure. We show how to efficiently maintain this representation during local changes of the finger, ultimately leading to Theorem 2.1(ii).

### 2.1.4 Longest Common Extensions

As application of Theorem 2.1, we give an improved solution to longest common extension problem in grammar compressed strings. The first solution to this problem is due to Bille et al. [15]. They showed how to extend random access queries to compute Karp-Rabin fingerprints. Combined with an exponential search this leads to a linear space solution to the longest common extension problem using  $O(\log N \log \ell)$  time, where  $\ell$  is the length of the longest common

extension. We note that we can plug in any of the above mentioned random access solution. More recently, Nishimoto et al. [107] used a completely different approach to get  $O(\log N + \log \ell \log^* N)$  query time while using superlinear  $O(n \log N \log^* N)$  space. We obtain:

**THEOREM 2.2** *Let  $\mathcal{S}$  be a grammar of size  $n$  representing a string  $S$  of length  $N$ . We can solve the longest common extension problem in  $O(\log N + \log^2 \ell)$  time and  $O(n)$  space where  $\ell$  is the length of the longest common extension.*

Note that we need to verify the Karp-Rabin fingerprints during preprocessing in order to obtain a worst-case query time. Using the result from Bille et al. [18] this gives a randomized expected preprocessing time of  $O(N \log N)$ .

Theorem 2.2 improves the  $O(\log N \log \ell)$  solution to  $O(\log N + \log^2 \ell)$ . The new bound is always at least as good and asymptotically better whenever  $\ell = o(N^\epsilon)$  where  $\epsilon$  is a constant. The new result follows by extending Theorem 2.1 to compute Karp-Rabin fingerprints and use these to perform the exponential search from [15].

## 2.2 Preliminaries

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

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

**Grammars and Straight Line Programs** Grammar-based compression replaces a long string by a small context-free grammar (CFG). We assume without loss of generality that the grammars are in fact *straight-line programs* (SLPs). The lefthand side of a grammar rule in an SLP has exactly one variable, and the forrighthand side has either exactly two variables or one terminal symbol. In addition, SLPs are unambiguous and acyclic. We view SLPs as a directed acyclic graph (DAG) where each rule correspond to a node with outgoing ordered edges



to its variables. Let  $\mathcal{S}$  be an SLP. As with trees, we denote the left and right child of an internal node  $v$  by  $left(v)$  and  $right(v)$ . The unique string  $S(v)$  of length  $N_v$  is produced by a depth-first left-to-right traversal of  $v$  in  $\mathcal{S}$  and consist of the characters on the leafs in the order they are visited. The corresponding parse tree for  $v$  is denoted  $T(v)$ . We will use the following results, that provides efficient random access from any node  $v$  in  $\mathcal{S}$ .

**LEMMA 2.3 ([18])** *Let  $S$  be a string of length  $N$  compressed into a SLP  $\mathcal{S}$  of size  $n$ . Given a node  $v \in \mathcal{S}$ , we can support random access in  $S(v)$  in  $O(\log N_v)$  time, and at the same time reporting the sequence of heavy paths and their entry- and exit points in the corresponding depth-first traversal of  $\mathcal{S}(v)$ . The number of heavy paths visited is  $O(\log N_v)$ .*

**Karp-Rabin Fingerprints** For a prime  $p$ ,  $2n^{c+4} < p \leq 4n^{c+4}$  and  $x \in [p]$  the Karp-Rabin fingerprint [83], denoted  $\phi(S[i, j])$ , of the substring  $S[i, j]$  is defined as  $\phi(S[i, j]) = \sum_{i \leq k \leq j} S[k]x^{k-i} \bmod p$ . The key property is that for a random choice of  $x$ , two substrings of  $S$  match iff their fingerprints match (whp.), thus allowing us to compare substrings in constant time. We use the following well-known properties of fingerprints.

**LEMMA 2.4** *The Karp-Rabin fingerprints have the following properties:*

- 1) *Given  $\phi(S[i, j])$ , the fingerprint  $\phi(S[i, j \pm a])$  for some integer  $a$ , can be computed in  $O(a)$  time.*
- 2) *Given fingerprints  $\phi(S[1, i])$  and  $\phi(S[1, j])$ , the fingerprint  $\phi(S[i, j])$  can be computed in  $O(1)$  time.*
- 3) *Given fingerprints  $\phi(S_1)$  and  $\phi(S_2)$ , the fingerprint  $\phi(S_1 \cdot S_2) = \phi(S_1) \oplus \phi(S_2)$  can be computed in  $O(1)$  time.*

## 2.3 Fringe Access

In this section we consider the *fringe access problem*. Here the goal is to compactly represent the SLP, such that for any node  $v$ , we can efficiently access locations in the string  $S(v)$  close to the start or the end of the substring. The fringe access problem is the key component in our finger search data structures. A straightforward solution to the fringe access problem is to apply a solution to the random access problem. For instance if we apply the random access solution from Bille et al. [18] stated in Lemma 2.3 we immediately obtain a linear space

solution with  $O(\log N_v)$  access time, i.e., the access time is independent of the distance to the start or the end of the string. This is an immediate consequence of the central grammar decomposition technique of [18], and does not extend to solve fringe access efficiently. Our main contribution in this section is a new approach that bypasses this obstacle. We show the following result.

**LEMMA 2.5** *Let  $\mathcal{S}$  be an SLP of size  $n$  representing a string of length  $N$ . Using  $O(n)$  space, we can support access to position  $i$  of any node  $v$ , in time  $O(\log(\min(i, N_v - i)) + \log \log N)$ .*

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

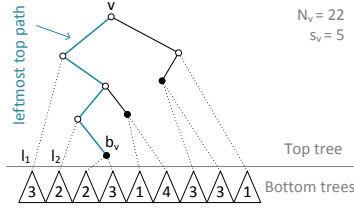
For simplicity and without loss of generality we assume that the access point  $i$  is closest to the start of  $S(v)$ , i.e., the goal is to obtain  $O(\log(i) + \log \log N)$  time. By symmetry we can obtain the corresponding result for access points close to the end of  $S(v)$ .

### 2.3.1 van Emde Boas Decomposition for Grammars

We first define the vEB decomposition on the parse tree  $T$  and then extend it to the SLP  $\mathcal{S}$ . In the decomposition we use the ART decomposition by Alstrup et al. [2].

**ART Decomposition** The ART decomposition introduced by Alstrup et al. [2] decomposes a tree into a single top tree and a number of bottom trees. Each bottom tree is a subtree rooted in a node of minimal depth such that the subtree contains no more than  $x$  leaves and the top tree is all nodes not in a bottom tree. The decomposition has the following key property.

**LEMMA 2.6 ([2])** *The ART decomposition with parameter  $x$  for a rooted tree  $T$  with  $N$  leaves produces a top tree with at most  $\frac{N}{x+1}$  leaves.*



**Figure 2.1:** Example of the ART-decomposition and a leftmost top path. In the top, the nodes forming the top tree are drawn. In the bottom, triangles representing the bottom trees with a number that is the size of the bottom tree.  $v$ 's leftmost top path is shown as well, and the two trees hanging to the left of this path  $l_1$  and  $l_2$ .

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

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

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

Note that except for the nodes on the lowest level—which are not in any top tree—all nodes belong to exactly one top tree. For any node  $v \in T$  not in the last level, let  $T_{\text{top}}(v)$  be the top tree  $v$  belongs to. The *leftmost top path* of  $v$  is the path from  $v$  to the *leftmost leaf* of  $T_{\text{top}}(v)$ . See Figure 2.1.

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

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

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

### 2.3.2 Data Structure

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

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

- The length  $N_v$  of  $S(v)$ .
- The sum of the sizes of nodes hanging to the left of  $v$ 's leftmost top path  $s_v = |l_1| + |l_2| + \dots + |l_k|$ .
- A pointer  $b_v$  to the bottom node on  $v$ 's leftmost top path.
- A predecessor data structure over the sequence  $1, |l_1| + 1, |l_1| + |l_2| + 1, \dots, \sum_{i=1}^{k-1} |l_i| + 1$ . We will later show how to represent this data structure.

In addition we also build the data structure from Lemma 2.3 that given any node  $v$  supports random access to  $S(v)$  in  $O(\log N_v)$  time using  $O(n)$  space.

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

1. If  $N_v = O(1)$ . Decompress  $S(v)$  and return the  $i$ 'th character.
2. If  $i \leq s_v$ . Find the predecessor  $p$  of  $i$  in  $v$ 's predecessor structure and let  $u$  be the corresponding node. Recursively find  $S(u)[i - p]$ .
3. If  $i \leq s_v + N_{\text{left}(b_v)}$ . Recursively find  $S(\text{left}(b_v))[i - s_v]$ .
4. If  $i \leq s_v + N_{b_v}$ . Recursively find  $S(\text{right}(b_v))[i - s_v - N_{\text{left}(b_v)}]$ .
5. In all other cases, perform a random access for  $i$  in  $S(v)$  using Lemma 2.3.

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

**Compact Predecessor Data Structures** We now describe how to represent the predecessor data structure. Simply storing a predecessor structure in every single node would use  $O(n^2)$  space. We can reduce the space to  $O(n)$  using ideas similar to the construction of the "heavy path suffix forest" in [18].

Let  $L$  denote the *leftmost top path forest*. The nodes of  $L$  are the nodes of  $\mathcal{S}$ . A node  $u$  is the parent of  $v$  in  $L$  iff  $u$  is a child of  $v$  in  $\mathcal{S}$  and  $u$  is on  $v$ 's leftmost top path. Thus, a leftmost top path  $v_1, \dots, v_k$  in  $\mathcal{S}$  is a sequence of ancestors from  $v_1$  in  $L$ . The weight of an edge  $(u, v)$  in  $L$  is 0 if  $u$  is a left child of  $v$  in  $\mathcal{S}$  and otherwise  $N_{\text{left}(v)}$ . Several leftmost top paths in  $\mathcal{S}$  can share the same suffix, but the leftmost top path of a node in  $\mathcal{S}$  is uniquely defined and thus  $L$  is a forest. A leftmost path ends in a leaf in the top DAG, and therefore  $L$  consists of  $O(n)$  trees each rooted at a unique leaf of a top dag. A predecessor query on the sequence  $1, |l_1| + 1, |l_1| + |l_2| + 1, \dots, \sum_{i=1}^{k-1} |l_i| + 1$  now corresponds to a weighted ancestor query in  $L$ . We plug in the weighted ancestor data structure from Farach-Colton and Muthukrishnan [39], which supports weighted ancestor queries in a forest in  $O(\log \log n + \log \log U)$  time with  $O(n)$  preprocessing and space, where  $U$  is the maximum weight of a root-to-leaf path and  $n$  the number of leaves. We have  $U = N$  and hence the time for queries becomes  $O(\log \log N)$ .

**Space and Preprocessing Time** For each node in  $\mathcal{S}$  we store a constant number of values, which takes  $O(n)$  space. Both the predecessor data structure and the data structure for supporting random access from Lemma 2.3 take  $O(n)$  space, so the overall space usage is  $O(n)$ . The vEB decomposition can be computed in  $O(n)$  time. The leftmost top paths and the information saved in each node can be computed in linear time. The predecessor data structure uses linear preprocessing time, and thus the total preprocessing time is  $O(n)$ .

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

In case (5) we make a random access query in a node of size  $N_v$ . From Lemma 2.3

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

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

In summary, we have the following result.

**LEMMA 2.7** *Let  $\mathcal{S}$  be an SLP of size  $n$  representing a string of length  $N$ . Using  $O(n)$  space, we can support access to position  $i$  of any node  $v$ , in time  $O(\log i + (\log \log N)^2)$ .*

### 2.3.3 Improving the Query Time for Small Indices

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

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

The new copy of the data structure is implemented as follows. In the first level of the ART-decomposition let  $x = 2^{(\log \log N)^2}$  instead of  $\sqrt{N}$ . For the rest of the levels use  $\sqrt{x}$  as before. Furthermore, we split the resulting new leftmost top path forest  $L$  into two disjoint parts:  $L_1$  consisting of all nodes with level 1 and  $L_{\geq 2}$  consisting of all nodes with level at least 2. For  $L_1$  we use the weighted ancestor data structure by Farach-Colton and Muthukrishnan [39] as in the previous section using  $O(\log \log n + \log \log N) = O(\log \log N)$  time. However, if we apply this solution for  $L_{\geq 2}$  we end up with a query time of  $O(\log \log n + \log \log x)$ , which does not lead to an improved solution. Instead, we present a new data structure that supports queries in  $O(\log \log x)$  time.

**LEMMA 2.8** *Given a tree  $T$  with  $n$  leaves where the sum of edge weights on any root-to-leaf path is at most  $x$  and the height is at most  $x$ , we can support weighted ancestor queries in  $O(\log \log x)$  time using  $O(n)$  space and preprocessing time.*

PROOF. Create an ART-decomposition of  $T$  with parameter  $x$ . For each bottom tree in the decomposition construct the weighted ancestor structure from [39]. For the top tree, construct a predecessor structure over the accumulated edge weights for each root-to-leaf path.

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

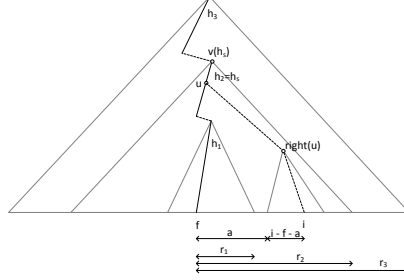
The total space for bottom trees is  $O(n)$ . Since the top tree has  $O(n/x)$  leaves and height at most  $x$ , the total space for all predecessor data structures on root-to-leaf paths in the top tree is  $O(n/x \cdot x) = O(n)$ . Hence, the total space is  $O(n)$ .

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

We reduce the query time for queries with  $i < 2^{(\log \log N)^2}$  using the new data structure. The level of any node in the new structure is at most  $O(1 + \log \log 2^{(\log \log N)^2}) = O(\log \log \log N)$ . A weighted ancestor query in  $L_1$  takes time  $O(\log \log N)$ . For weighted ancestor queries in  $L_{\geq 2}$ , we know any node  $v$  has height at most  $2^{(\log \log N)^2}$  and on any root-to-leaf path the sum of the weights is at most  $2^{(\log \log N)^2}$ . Hence, by Lemma 2.8 we support queries in  $O(\log \log 2^{(\log \log N)^2}) = O(\log \log \log N)$  time for nodes in  $L_{\geq 2}$ .

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

In summary, this completes the proof of Lemma 2.5.



**Figure 2.2:** Illustration of the data structure for a finger pointing at  $f$  and an access query at location  $i$ .  $h_1, h_2, h_3$  are the heavy paths visited when finding the finger.  $u$  corresponds to  $NCA(v_f, v_i)$  in the parse tree and  $h_s$  is the heavy path on which  $u$  lies, which we use to find  $u$ .  $a$  is a value calculated during the access query.

## 2.4 Static Finger Search

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

Similar to the fringe search problem we assume without loss of generality that the access point  $i$  is to the right of the finger.

**Data Structure** We store the random access data structure from [18] used in Lemma 2.3 and the fringe search data structures from above. Also from [18] we store the data structure that for any heavy path  $h$  starting in a node  $v$  and an index  $i$  of a leaf in  $T(v)$  gives the exit-node from  $h$  when searching for  $i$  in  $O(\log \log N)$  time and uses  $O(n)$  space.

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



1. The sequence  $r_1, r_2, \dots, r_j$  (note  $r_1 \leq r_2 \leq \dots \leq r_j$ ).
2. The sequence  $v(h_1), v(h_2), \dots, v(h_j)$ .
3. The string  $F_T = S[f + 1, f + \log N]$ .

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

**Setfinder** We implement `setfinder( $f$ )` as follows. First, we apply Lemma 2.3 to make random access to position  $f$ . This gives us the sequence of visited heavy paths which exactly corresponds to  $h_j, h_{j-1}, \dots, h_1$  including the corresponding  $l_i$  values from which we can calculate the  $r_i$  values. So we update the  $r_i$  sequence accordingly. Finally, decompress and save the string  $F_T = S[f + 1, f + \log N]$ .

The random access to position  $f$  takes  $O(\log N)$  time. In addition to this we perform a constant number of operations for each heavy path  $h_i$ , which in total takes  $O(\log N)$  time. Decompressing a string of  $\log N$  characters can be done in  $O(\log N)$  time (using [18]). In total, we use  $O(\log N)$  time.

**Access** To perform `access( $i$ )` ( $i > f$ ), there are two cases. If  $D = i - f \leq \log N$  we simply return the stored character  $F_T[D]$  in constant time. Otherwise, we compute the node  $u = \text{nca}(v_f, v_i)$  in the parse tree  $T$  as follows. First find the index  $s$  of the successor to  $D$  in the  $r_i$  sequence using binary search. Now we know that  $u$  is on the heavy path  $h_s$ . Find the exit-nodes from  $h_s$  when searching for respectively  $i$  and  $f$  using the data structure from [18] - the topmost of these two is  $u$ . See Fig. 2.2. Finally, we compute  $a$  as the index of  $f$  in  $T(\text{left}(u))$  from the right and use the data structure for fringe search from Lemma 2.5 to compute  $S(\text{right}(u))[i - f - a]$ .

For  $D \leq \log N$ , the operation takes constant time. For  $D > \log N$ , the binary search over a sequence of  $O(\log N)$  elements takes  $O(\log \log N)$  time, finding the exit-nodes takes  $O(\log \log N)$  time, and the fringe search takes  $O(\log(i - f - a)) = O(\log D)$  time. Hence, in total  $O(\log \log N + \log D) = O(\log D)$  time.

This completes the proof of Theorem 2.1(i).

## 2.5 Dynamic Finger Search

In this section we show how to extend the solution from Section 2.4 to handle dynamic finger search. The target is to support the **movefinger** operation that will move the current finger, where the time it takes is dependent on how far the finger is moved. Obviously, it should be faster than simply using the **setfinger** operation. The key difference from the static finger is a new decomposition of a root-to-leaf path into paths. The new decomposition is based on a combination of heavy paths and leftmost top paths, which we will show first. Then we show how to change the data structure to use this decomposition, and how to modify the operations accordingly. Finally, we consider how to generalize the solution to work when **movefinger/access** might both be to the left and right of the current finger, which for this solution is not trivially just by symmetry.

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

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

### 2.5.1 Left Heavy Path Decomposition of a Path

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

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

Define  $v(p_i)$  as the topmost node on the subpath  $p_i$ . Let  $l_j$  be the index of the

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

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

**LEMMA 2.9** *The number of paths in a left heavy path decomposition is  $O(\log N)$ .*

PROOF. There are at most  $O(\log N)$  heavy paths that intersects with a root-to-leaf path (Lemma 2.3). Each of these can at most be used once because of the maximality. So there can at most be  $O(\log N)$  maximal heavy paths. Each time there is a maximal leftmost top path, the level of the following node on  $p$  increases. This can happen at most  $O(\log \log N)$  times. Each non-maximal leftmost top path is followed by a maximal heavy path, and since there are only  $O(\log N)$  of these, this can happen at most  $O(\log N)$  times. Therefore the sequence of paths has length  $O(\log N + \log \log N + \log N) = O(\log N)$ .

## 2.5.2 Data Structure

We use the data structures from [18] as in the static variant and the fringe access data structure with an extension. In the fringe access data structure there is a predecessor data structure for all the nodes hanging to the left of a leftmost top path. To support **access** and **movefinger** we need to find a node hanging to the left or right of a leftmost top path. We can do this by storing an identical predecessor structure for the accumulated sizes of the nodes hanging to the right of each leftmost top path. Again, the space usage for this predecessor structure can be reduced to  $O(n)$  by turning it into a weighted ancestor problem.

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

1. The sequence  $r_1, r_2, \dots, r_j$  ( $r_1 \leq r_2 \leq \dots \leq r_j$ ) on a stack with the last element on top.
2. The sequence  $v(p_1), v(p_2), \dots, v(p_j)$  on a stack with the last element on top.

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

**Analysis** The fringe access data structure takes up  $O(n)$  space. For each path in the left heavy path decomposition we use constant space. Using Lemma 2.9 we have the space usage of this is  $O(\log N) = O(n)$ .

**Setfinger** Use fringe access (Lemma 2.5) to access position  $f$ . This gives us a sequence of leftmost top paths and heavy paths visited during the fringe access which is a valid left heavy path decomposition. Calculate  $r_i$  for each of these and store the three sequences of  $r_i$ ,  $v(p_i)$  and  $t(p_i)$  on stacks.

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

**Access** To implement **access**( $i$ ) for  $i > f$  we have to find  $u = \text{nca}(v_i, v_f)$  in the  $T$ . Find the index  $s$  of the successor to  $D = i - f$  in  $r_1, r_2, \dots, r_j$  using binary search. We know  $\text{nca}(v_i, v_f)$  lies on  $p_s$ , and  $v_i$  is in a subtree that hangs of  $p_s$ . The exit-nodes from  $p_s$  to  $v_f$  and  $v_i$  are now found - the topmost of these two is  $\text{nca}(v_i, v_f)$ . If  $t(p_s) = HP$  then we can use the same data structure as in the static case, otherwise we perform the predecessor query on the extra predecessor data structure for the nodes hanging of the leftmost top path. Finally, we compute  $a$  as the index of  $f$  in  $S(\text{left}(u))$  from the right and use the data structure for fringe access from Lemma 2.5 to compute  $S(\text{right}(u))[i - f - a]$ .

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

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

**Movefinger** To move the finger we combine the **access** and **setfinger** operations. Find the index  $s$  of the successor to  $D = i - f$  in  $r_1, r_2, \dots, r_j$  using binary search. Now we know  $u = \text{nca}(v_i, v_f)$  must lie on  $p_s$ . Find  $u$  in the same way as when performing access. From all of the stacks pop all elements above index  $s$ . Compute  $a$  as the index of  $f$  in  $S(\text{left}(u))$  from the right. The finger

should be moved to index  $i - f - a$  in  $right(u)$ . First look at the heavy path  $right(u)$  lies on and find the proper exit-node  $w$  using the data structure from [18]. Then continue with fringe search from the proper child of  $w$ . This gives a heavy path followed by a sequence of maximal leftmost top paths and heavy paths needed to reach  $v_i$  from  $right(u)$ , push the  $r_j$ ,  $v(p_j)$ , and  $t(p_j)$  values for these on top of the respective stacks.

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

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

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

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

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

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

## 2.6 Finger Search with Fingerprints and Longest Common Extensions

We show how to extend our finger search data structure from Theorem 2.1(i) to support computing fingerprints and then apply the result to compute longest common extensions. First, we will show how to return a fingerprint for  $S(v)[1, i]$  when performing `access` on the fringe of  $v$ .

### 2.6.1 Fast Fingerprints on the Fringe

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

**LEMMA 2.10 ([15])** *Let  $S$  be a string of length  $N$  compressed into a SLP  $\mathcal{S}$  of size  $n$ . Given a node  $v \in \mathcal{S}$ , we can find the fingerprint  $\phi(S(v)[1, i])$  where  $1 \leq i \leq N_v$  in  $O(\log N_v)$  time.*

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

1. From the decompressed  $S(v)$ , calculate the fingerprint for  $S(v)[1, i]$ , now update  $p = p \oplus \phi(S(v)[1, i])$ .
2.  $p = p \oplus (\phi(p_v) \ominus_s \phi(p_u))$ .
3.  $p = p \oplus \phi(p_v)$ .
4.  $p = p \oplus \phi(p_v) \oplus \phi(S(left(b_v)))$ .

5. Use Lemma 2.10 to find the fingerprint for  $S(v)[1, i]$  and then update with  $p = p \oplus \phi(S(v)[1, i])$ .

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

### 2.6.2 Finger Search with Fingerprints

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

When we perform `setfinger( $f$ )` we use the algorithm from [15] to compute fingerprints during the search of  $\mathcal{S}$  from the root to  $f$ . This allows us to subsequently compute for any heavy path  $h_j$  on the root to position  $f$  the fingerprint  $p(h_j)$  of the concatenation of the strings generated by the subtrees hanging to the left of  $h_j$ . In addition, we explicitly compute and store the fingerprints  $\phi(S[1, f + 1]), \phi(S[1, f + 2]), \dots, \phi(S[1, f + \log N + 1])$ . In total, this takes  $O(\log N)$  time.

Suppose that we have now performed a `setfinger( $f$ )` operation. To implement `access( $i$ )`,  $i > f$ , there are two cases. If  $D = i - f \leq \log N$  we return the appropriate precomputed fingerprint. Otherwise, we compute the node  $u = \text{nca}(v_f, v_i)$  in the parse tree  $T$  as before. Let  $h$  be the heavy path containing  $u$ . Using the data structure from [15] we compute the fingerprint  $p_l$  of the nodes hanging to the left of  $h$  above  $u$  in constant time. The fingerprint is now obtained as  $\phi(S[1, i]) = p_{h_j} \oplus p_l \oplus \phi(S(\text{right}(u))[1, (i - f) - a])$ , where the latter is found using fringe access with fingerprints in `right( $u$ )`. None of these additions change the asymptotic complexities of Theorem 2.1(i). Note that with the fingerprint construction in [15] we can guarantee that all fingerprints are collision-free.

### 2.6.3 Longest Common Extensions

Using the fingerprints it is now straightforward to implement `lce` queries as in [15]. Given a `lce( $i, j$ )` query, first set fingers at positions  $i$  and  $j$ . This allows us to get fingerprints of the form  $\phi(S[i, i + a])$  or  $\phi(S[j, j + a])$  efficiently. Then, we find the largest value  $\ell$  such that  $\phi(S[i, i + \ell]) = \phi(S[j, j + \ell])$  using a standard exponential search. Setting the two finger uses  $O(\log N)$  time and by Theorem 2.1(i) the at most  $O(\log \ell)$  searches in the exponential search take at

most  $O(\log \ell)$  time. Hence, in total we use  $O(\log N + \log^2 \ell)$  time, as desired. This completes the proof of Theorem 2.2.





# Compressed Indexing with Signature Grammars

---

Anders Roy Christiansen <sup>†</sup>      Mikko Berggren Ettienne <sup>†</sup>

<sup>†</sup> The Technical University of Denmark

## Abstract

The *compressed indexing problem* is to preprocess a string  $S$  of length  $n$  into a compressed representation that supports pattern matching queries. That is, given a string  $P$  of length  $m$  report all occurrences of  $P$  in  $S$ .

We present a data structure that supports pattern matching queries in  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  time using  $O(z \lg(n/z))$  space where  $z$  is the size of the LZ77 parse of  $S$  and  $\epsilon > 0$ , when the alphabet is small or the compression ratio is at least polynomial. We also present two data structures for the general case; one where the space is increased by  $O(z \lg \lg z)$ , and one where the query time changes from worst-case to expected.

In all cases, the results improve the previously best known solutions. Notably, this is the first data structure that decides if  $P$  occurs in  $S$  in  $O(m)$  time using  $O(z \lg(n/z))$  space.

Our results are mainly obtained by a novel combination of a randomized grammar construction algorithm with well known techniques relating pattern matching to 2D-range reporting.

### 3.1 Introduction

Given a string  $S$  and a pattern  $P$ , the core problem of pattern matching is to report all locations where  $P$  occurs in  $S$ . Pattern matching problems can be divided into two: the algorithmic problem where the text and the pattern are given at the same time, and the data structure problem where one is allowed to preprocess the text (pattern) before a query pattern (text) is given. Many problems within both these categories are well-studied in the history of stringology, and optimal solutions to many variants have been found.

In the last decades, researchers have shown an increasing interest in the compressed version of this problem, where the space used by the index is related to the size of some compressed representation of  $S$  instead of the length of  $S$ . This could be measures such as the size of the LZ77-parse of  $S$ , the smallest grammar representing  $S$ , the number of runs in the BWT of  $S$ , etc. see e.g. [16, 55, 56, 59, 82, 102, 108]. This problem is highly relevant as the amount of highly-repetitive data increases rapidly, and thus it is possible to handle greater amounts of data by compressing it. The increase in such data is due to things like DNA sequencing, version control repositories, etc.

In this paper we consider what we call the *compressed indexing problem*, which is to preprocess a string  $S$  of length  $n$  into a compressed representation that supports fast *pattern matching queries*. That is, given a string  $P$  of length  $m$ , report all  $\text{occ}$  occurrences of substrings in  $S$  that match  $P$ .

Table 3.1 gives an overview of the results on this problem.

#### 3.1.1 Our Results

In this paper we improve previous solutions that are bounded by the size of the LZ77-parse. For constant-sized alphabets we obtain the following result:

**THEOREM 3.1** *Given a string  $S$  of length  $n$  from a constant-sized alphabet with an LZ77 parse of length  $z$ , we can build a compressed-index supporting pattern matching queries in  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  time using  $O(z \lg(n/z))$  space.*

In particular, we are the first to obtain optimal search time using only  $O(z \lg(n/z))$  space. For general alphabets we obtain the following:

**Table 3.1:** Selection of previous results and our new results on compressed indexing. The variables are the text size  $n$ , the LZ77-parse size  $z$ , the pattern length  $m$ ,  $\text{occ}$  is the number of occurrences and  $\sigma$  is the size of the alphabet. (The time complexity marked by  $\dagger$  is expected whereas all others are worst-case)

Index	Space	Locate time	$\sigma$
Gagie et al. [56]	$O(z \lg(n/z))$	$O(m \lg m + \text{occ} \lg \lg n)$	$O(1)$
Nishimoto et al. [108]	$O(z \lg n \lg^* n)$	$O(m \lg \lg n \lg \lg z + \lg z \lg m \lg n (\lg^* n)^2 + \text{occ} \lg n)$	$n^{O(1)}$
Bille et al. [16]	$O(z(\lg(n/z) + \lg^\epsilon z))$	$O(m + \text{occ}(\lg^\epsilon n + \lg \lg n))$	$n^{O(1)}$
Bille et al. [16]	$O(z \lg(n/z) \lg \lg z)$	$O(m + \text{occ} \lg \lg n)$	$O(1)$
Bille et al. [16]	$O(z \lg(n/z))$	$O(m(1 + \frac{\lg^\epsilon z}{\lg(n/z)}) + \text{occ}(\lg^\epsilon n + \lg \lg n))$	$O(1)$
<b>Theorem 1</b>	$O(z \lg(n/z))$	$O(m + \text{occ}(\lg^\epsilon z + \lg \lg n))$	$O(1)$
<b>Theorem 2 (1)</b>	$O(z(\lg(n/z) + \lg \lg z))$	$O(m + \text{occ}(\lg^\epsilon z + \lg \lg n))$	$n^{O(1)}$
<b>Theorem 2 (2)</b>	$O(z(\lg(n/z))$	$O(m + \text{occ}(\lg^\epsilon z + \lg \lg n))^\dagger$	$n^{O(1)}$

**THEOREM 3.2** *Given a string  $S$  of length  $n$  from an integer alphabet polynomially bounded by  $n$  with an LZ77-parse of length  $z$ , we can build a compressed-index supporting pattern matching queries in:*

- (1)  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  time using  $O(z(\lg(n/z) + \lg \lg z))$  space.
- (2)  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  expected time using  $O(z \lg(n/z))$  space.
- (3)  $O(m + \lg^\epsilon z + \text{occ}(\lg \lg n + \lg^\epsilon z))$  time using  $O(z \lg(n/z))$  space.

Note  $\lg \lg z = O(\lg(n/z))$  when either the alphabet size is  $O(2^{\lg^\epsilon n})$  or  $z = o(\frac{n}{\lg^{\epsilon'} n})$  where  $\epsilon$  and  $\epsilon'$  are arbitrarily small positive constants. Theorem 4.1 follows directly from Theorem 3.2 (1) given these observations. Theorem 3.2 is a consequence of Lemma 3.11, 3.13, 3.14 and 3.15.

### 3.1.2 Technical Overview

Our main new contribution is based on a new grammar construction. In [101] Melhorn et al. presented a way to maintain dynamic sequences subject to equality testing using a technique called signatures. They presented two signature construction techniques. One is randomized and leads to complexities that hold in expectation. The other is based on a deterministic coin-tossing technique of Cole and Vishkin [30] and leads to worst-case running times but incurs an iterated logarithmic overhead compared to the randomized solution. This technique has also resembles the string labeling techniques found e.g. in [119]. To the best of our knowledge, we are the first to consider grammar compression based on the randomized solution from [101]. Despite it being randomized we show how to obtain worst-case query bounds for text indexing using this technique.

The main idea in this grammar construction is that similar substrings will be parsed almost identically. This property also holds true for the deterministic construction technique which has been used to solve dynamic string problems with and without compression, see e.g. [1, 108]. In [80] Jeř deviates a different grammar construction algorithm with similar properties to solve the algorithmic pattern matching problem on grammar compressed strings which has later been used for both static and dynamic string problems, see [61, 127]

Our primary solution has an  $\lg^\epsilon z$  term in the query time which is problematic for short query patterns. To handle this, we show different solutions for handling short query patterns. These are based on the techniques from LZ77-based indexing combined with extra data structures to speed up the queries.

## 3.2 Preliminaries

We assume a standard unit-cost RAM model with word size  $\Theta(\lg n)$  and that the input is from an integer alphabet  $\Sigma = \{1, 2, \dots, n^{O(1)}\}$ . We measure space complexity in terms of machine words unless explicitly stated otherwise. A string  $S$  of length  $n = |S|$  is a sequence of  $n$  symbols  $S[1] \dots S[n]$  drawn from an alphabet  $\Sigma$ . The sequence  $S[i, j]$  is the *substring* of  $S$  given by  $S[i] \dots S[j]$  and strings can be concatenated, i.e.  $S = S[1, k]S[k+1, n]$ . The empty string is denoted  $\epsilon$  and  $S[i, i] = S[i]$  while  $S[i, j] = \epsilon$  if  $j < i$ ,  $S[i, j] = S[1, j]$  if  $i < 1$  and  $S[i, n]$  if  $j > n$ . The reverse of  $S$  denoted  $rev(s)$  is the string  $S[n]S[n-1] \dots S[1]$ . A *run* in a string  $S$  is a substring  $S[i, j]$  with identical letters, i.e.  $S[k] = S[k+1]$  for  $k = i, \dots, j-1$ . Let  $S[i, j]$  be a run in  $S$  then it is a *maximal run* if it cannot be extended, i.e.  $S[i-1] \neq S[i]$  and  $S[j] \neq S[j+1]$ . If there are no runs in  $S$  we say that  $S$  is *run-free* and it follows that  $S[i] \neq S[i+1]$  for  $1 \leq i < n$ . Denote by  $[u]$  the set of integers  $\{1, 2, \dots, u\}$ .

Let  $X \subseteq [u]^2$  be a set of points in a 2-dimensional grid. The *2D-orthogonal range reporting problem* is to compactly represent  $Z$  while supporting *range reporting queries*, that is, given a rectangle  $R = [a_1, b_1] \times [a_2, b_2]$  report all points in the set  $R \cap X$ . We use the following:

**LEMMA 3.3 (CHAN ET AL. [24])** *For any set of  $n$  points in  $[u] \times [u]$  and constant  $\epsilon > 0$ , we can solve 2D-orthogonal range reporting with  $O(n \lg n)$  expected preprocessing time using:*

- i  $O(n)$  space and  $(1+k) \cdot O(\lg^\epsilon n \lg \lg u)$  query time
- ii  $O(n \lg \lg n)$  space and  $(1+k) \cdot O(\lg \lg u)$  query time

where  $k$  is the number of occurrences inside the rectangle.

A *Karp-Rabin fingerprinting function* [84] is a randomized hash function for strings. Given a string  $S$  of length  $n$  and a fingerprinting function  $\phi$  we can in  $O(n)$  time and space compute and store  $O(n)$  fingerprints such that the fingerprint of any substring of  $S$  can be computed in constant time. Identical strings have identical fingerprints. The fingerprints of two strings  $S$  and  $S'$  *collide* when  $S \neq S'$  and  $\phi(S) = \phi(S')$ . A fingerprinting function is *collision-free* for a set of strings when there are no collisions between the fingerprints of any two strings in the set. We can find collision-free fingerprinting function for a set of strings with total length  $n$  in  $O(n)$  expected time [112].

Let  $D$  be a lexicographically sorted set of  $k$  strings. The weak prefix search problem is to compactly represent  $D$  while supporting *weak prefix queries*, that is, given a query string  $P$  of length  $m$  report the rank of the lexicographically smallest and largest strings in  $D$  of which  $P$  is a prefix. If no such strings exist, the answer can be arbitrary.

**LEMMA 3.4 (BELAZZOUGUI ET AL. [11], APPENDIX H.3)** *Given a set  $D$  of  $k$  strings with average length  $l$ , from an alphabet of size  $\sigma$ , we can build a data structure using  $O(k(\lg l + \lg \lg \sigma))$  bits of space supporting weak prefix search for a pattern  $P$  of length  $m$  in  $O(m \lg \sigma / w + \lg m)$  time where  $w$  is the word size.*

We will refer to the data structure of Lemma 3.4 as a *z-fast trie* following the notation from [11]. The  $m$  term in the time complexity is due to a linear time preprocessing of the pattern and is not part of the actual search. Therefore it is simple to do weak prefix search for any length  $l$  substring of  $P$  in  $O(\lg l)$  time after preprocessing  $P$  once in  $O(m)$  time.

The *LZ77-parse* [138] of a string  $S$  of length  $n$  is a string  $\mathcal{Z}$  of the form  $(s_1, l_1, \alpha_1) \dots (s_z, l_z, \alpha_z) \in ([n], [n], \Sigma)^z$ . We define  $u_1 = 1$ ,  $u_i = u_{i-1} + l_{i-1} + 1$  for  $i > 1$ . For  $\mathcal{Z}$  to be a valid parse, we require  $l_1 = 0$ ,  $s_i < u_i$ ,  $S[u_i, u_i + l_i - 1] = S[s_i, s_i + l_i - 1]$ , and  $S[u_i + l_i] = \alpha_i$  for  $i \in [z]$ . This guarantees  $\mathcal{Z}$  represents  $S$  and  $S$  is uniquely defined in terms of  $\mathcal{Z}$ . The substring  $S[u_i, u_i + l_i]$  is called the  $i^{\text{th}}$  phrase of the parse and  $S[s_i, s_i + l_i - 1]$  is its source. A minimal LZ77-parse of  $S$  can be found greedily in  $O(n)$  time and stored in  $O(z)$  space [138]. We call the positions  $u_1 + l_1, \dots, u_z + l_z$  the borders of  $S$ .

### 3.3 Signature Grammars

We consider a hierarchical representation of strings given by Melhorn et al. [101] with some slight modifications. Let  $S$  be a run-free string of length  $n$  from an integer alphabet  $\Sigma$  and let  $\pi$  be a uniformly random permutation of  $\Sigma$ . Define a position  $S[i]$  as a local minimum of  $S$  if  $1 < i < n$  and  $\pi(S[i]) < \pi(S[i - 1])$  and  $\pi(S[i]) < \pi(S[i + 1])$ . In the block decomposition of  $S$ , a block starts at position 1 and at every local minimum in  $S$  and ends just before the next block begins (the last block ends at position  $n$ ). The block decomposition of a string  $S$  can be used to construct the signature tree of  $S$  denoted  $\text{sig}(S)$  which is an ordered labeled tree with several useful properties.

**LEMMA 3.5** *Let  $S$  be a run-free string  $S$  of length  $n$  from an alphabet  $\Sigma$  and let  $\pi$  be a uniformly random permutation of  $\Sigma$  such that  $\pi(c)$  is the rank of the*

symbol  $c \in \Sigma$  in this permutation. Then the expected length between two local minima in the sequence  $\pi(S[1]), \pi(S[2]), \dots, \pi(S[n])$  is at most 3 and the longest gap is  $O(\lg n)$  in expectation.

PROOF. First we show the expected length between two local minima is at most 3. Look at a position  $1 \leq i \leq n$  in the sequence  $\pi(S[1]), \pi(S[2]), \dots, \pi(S[n])$ . To determine if  $\pi(S[i])$  is a local minimum, we only need to consider the two neighbouring elements  $\pi(S[i-1])$  and  $\pi(S[i+1])$  thus let us consider the triple  $(\pi(S[i-1]), \pi(S[i]), \pi(S[i+1]))$ . We need to consider the following cases. First assume  $S[i-1] \neq S[i] \neq S[i+1]$ . There exist  $3! = 6$  permutations of a triple with unique elements and in two of these the minimum element is in the middle. Since  $\pi$  is a uniformly random permutation of  $\Sigma$  all 6 permutations are equally likely, and thus there is  $1/3$  chance that the element at position  $i$  is a local minimum. Now instead assume  $S[i-1] = S[i+1] \neq S[i]$  in which case there is  $1/2$  chance that the middle element is the smallest. Finally, in the case where  $i = 1$  or  $i = n$  there is also  $1/2$  chance. As  $S$  is run-free, these cases cover all possible cases. Thus there is at least  $1/3$  chance that any position  $i$  is a local minimum independently of  $S$ . Thus the expected number of local minima in the sequence is therefore at least  $n/3$  and the expected distance between any two local minima is at most 3.

The expected longest distance between two local minima of  $O(\lg n)$  was shown in [101].

### 3.3.1 Signature Grammar Construction

We now give the construction algorithm for the signature tree  $\text{sig}(S)$ . Consider an ordered forest  $F$  of trees. Initially,  $F$  consists of  $n$  trees where the  $i^{\text{th}}$  tree is a single node with label  $S[i]$ . Let the label of a tree  $t$  denoted  $l(t)$  be the label of its root node. Let  $l(F)$  denote the string that is given by the in-order concatenation of the labels of the trees in  $F$ . The construction of  $\text{sig}(S)$  proceeds as follows:

1. Let  $t_i, \dots, t_j$  be a maximal subrange of consecutive trees of  $F$  with identical labels, i.e.  $l(t_i) = \dots = l(t_j)$ . Replace each such subrange in  $F$  by a new tree having as root a new node  $v$  with children  $t_i, \dots, t_j$  and a label that identifies the number of children and their label. We call this kind of node a run node. Now  $l(F)$  is run-free.
2. Consider the block decomposition of  $l(F)$ . Let  $t_i, \dots, t_j$  be consecutive trees in  $F$  such that their labels form a block in  $l(F)$ . Replace all identical



blocks  $t_i, \dots, t_j$  by a new tree having as root a new node with children  $t_i, \dots, t_j$  and a unique label. We call this kind of node a run-free node.

3. Repeat step 1 and 2 until  $F$  contains a single tree, we call this tree  $\text{sig}(S)$ .

In each iteration the size of  $F$  decreases by at least a factor of two and each iteration takes  $O(|F|)$  time, thus it can be constructed in  $O(n)$  time.

Consider the directed acyclic graph (DAG) of the tree  $\text{sig}(S)$  where all identical subtrees are merged. Note we can store run nodes in  $O(1)$  space since all outgoing edges are pointing to the same node, so we store the number of edges along with a single edge instead of explicitly storing each of them. For run-free nodes we use space proportional to their out-degrees. We call this the signature DAG of  $S$  denoted  $\text{dag}(S)$ . There is a one-to-one correspondence between this DAG and an acyclic run-length grammar producing  $S$  where each node corresponds to a production and each leaf to a terminal.

### 3.3.2 Properties of the Signature Grammar

We now show some properties of  $\text{sig}(S)$  and  $\text{dag}(S)$  that we will need later. Let  $\text{str}(v)$  denote the substring of  $S$  given by the labels of the leaves of the subtree of  $\text{sig}(S)$  induced by the node  $v$  in left to right order.

**LEMMA 3.6** *Let  $v$  be a node in the signature tree for a string  $S$  of length  $n$ . If  $v$  has height  $h$  then  $|\text{str}(v)|$  is at least  $2^h$  and thus  $\text{sig}(S)$  (and  $\text{dag}(S)$ ) has height  $O(\lg n)$ .*

PROOF. This follows directly from the out-degree of all nodes being at least 2.

Denote by  $T(i, j)$  the set of nodes in  $\text{sig}(S)$  that are ancestors of the  $i^{\text{th}}$  through  $j^{\text{th}}$  leaf of  $\text{sig}(S)$ . These nodes form a sequence of adjacent nodes at every level of  $\text{sig}(S)$  and we call them *relevant nodes* for the substring  $S[i, j]$ .

**LEMMA 3.7**  *$T(i, j)$  and  $T(i', j')$  have identical nodes except at most the two first and two last nodes on each level whenever  $S[i, j] = S[i', j']$ .*

PROOF. Trivially, the leaves of  $T(i, j)$  and  $T(i', j')$  are identical if  $S[i, j] = S[i', j']$ . Now we show it is true for nodes on level  $l$  assuming it is true for nodes on level  $l - 1$ . We only consider the left part of each level as the argument for the right part is (almost) symmetric. Let  $v_1, v_2, v_3, \dots$  be the nodes on level  $l - 1$

in  $T(i, j)$  and  $u_1, u_2, u_3, \dots$  the nodes on level  $l - 1$  in  $T(i', j')$  in left to right order. From the assumption, we have  $v_a, v_{a+1}, \dots$  are identical with  $u_b, u_{b+1}, \dots$  for some  $1 \leq a, b \leq 3$ . When constructing the  $l^{th}$  level of  $sig(S)$ , these nodes are divided into blocks. Let  $v_{a+k}$  be the first block that starts after  $v_a$  then by the block decomposition, the first block after  $u_b$  starts at  $u_{b+k}$ . The nodes  $v_1, \dots, v_{a+k}$  are spanned by at most two blocks and similarly for  $u_1, \dots, u_{b+k}$ . These blocks become the first one or two nodes on level  $l$  in  $T(i, j)$  and  $T(i', j')$  respectively. The block starting at  $v_{a+k}$  is identical to the block starting at  $u_{b+k}$  and the same holds for the following blocks. These blocks result in identical nodes on level  $l$ . Thus, if we ignore the at most two first (and last) nodes on level  $l$  the remaining nodes are identical.

We call nodes of  $T(i, j)$  consistent in respect to  $T(i, j)$  if they are guaranteed to be in any other  $T(i', j')$  where  $S[i, j] = S[i', j']$ . We denote the remaining nodes of  $T(i, j)$  as inconsistent. From the above lemma, it follows at most the left-most and right-most two nodes on each level of  $T(i, j)$  can be inconsistent.

**LEMMA 3.8** *The expected size of the signature DAG  $dag(S)$  is  $O(z \lg(n/z))$ .*

PROOF. We first bound the number of unique nodes in  $sig(S)$  in terms of the LZ77-parse of  $S$  which has size  $z$ . Consider the decomposition of  $S$  into the  $2z$  substrings  $S[u_1, u_1 + l_1], S[u_1 + l_1 + 1], \dots, S[u_z, u_z + l_z], S[u_z + l_z + 1]$  given by the phrases and borders of the LZ77-parse of  $S$  and the corresponding sets of relevant nodes  $R = \{T(u_1, u_1 + l_1), T(u_1 + l_1 + 1, u_1 + l_1 + 1), \dots\}$ . Clearly, the union of these sets are all the nodes of  $sig(S)$ . Since identical nodes are represented only once in  $dag(S)$  we need only count one of their occurrences in  $sig(S)$ . We first count the nodes at levels lower than  $\lg(n/z)$ . A set  $T(i, i)$  of nodes relevant to a substring of length one has no more than  $O(\lg(n/z))$  such nodes. By Lemma 3.7 only  $O(\lg(n/z))$  of the relevant nodes for a phrase are not guaranteed to also appear in the relevant nodes of its source. Thus we count a total of  $O(z \lg(n/z))$  nodes for the  $O(z)$  sets of relevant nodes. Consider the leftmost appearance of a node appearing one or more times in  $sig(S)$ . By definition, and because every node of  $sig(S)$  is in at least one relevant set, it must already be counted towards one of the sets. Thus there are  $O(z \lg(n/z))$  unique vertices in  $sig(S)$  at levels lower than  $\lg(n/z)$ . Now for the remaining at most  $\lg(z)$  levels, there are no more than  $O(z)$  nodes because the out-degree of every node is at least two. Thus we have proved that there are  $O(z \lg(n/z))$  unique nodes in  $sig(S)$ . By Lemma 3.5 the average block size and thus the expected out-degree of a node is  $O(1)$ . It follows that the expected number of edges and the expected size of  $dag(S)$  is  $O(z \lg(n/z))$ .

**LEMMA 3.9** *A signature grammar of  $S$  using  $O(z \lg(n/z))$  (worst case) space can be constructed in  $O(n)$  expected time.*

PROOF. Construct a signature grammar for  $S$  using the signature grammar construction algorithm. If the average out-degree of the run-free nodes in  $\text{dag}(S)$  is more than some constant greater than 3 then try again. In expectation it only takes a constant number of retries before this is not the case.

**LEMMA 3.10** *Given a node  $v \in \text{dag}(S)$ , the child that produces the character at position  $i$  in  $\text{str}(v)$  can be found in  $O(1)$  time.*

PROOF. First assume  $v$  is a run-free node. If we store  $|\text{str}(u)|$  for each child  $u$  of  $v$  in order, the correct child corresponding to position  $i$  can simply be found by iterating over these. However, this may take  $O(\log n)$  time since this is the maximum out-degree of a node in  $\text{dag}(S)$ . This can be improved to  $O(\log \log n)$  by doing a binary search, but instead we use a Fusion Tree from [50] that allows us to do this in  $O(1)$  time since we have at most  $O(\log n)$  elements. This does not increase the space usage. If  $v$  is a run node then it is easy to calculate the right child by a single division.

## 3.4 Long Patterns

In this section we present how to use the signature grammar to construct a compressed index that we will use for patterns of length  $\Omega(\lg^\epsilon z)$  for constant  $\epsilon > 0$ . We obtain the following lemma:

**LEMMA 3.11** *Given a string  $S$  of length  $n$  with an LZ77-parse of length  $z$  we can build a compressed index supporting pattern matching queries in  $O(m + (1 + \text{occ}) \lg^\epsilon z)$  time using  $O(z \lg(n/z))$  space for any constant  $\epsilon > 0$ .*

### 3.4.1 Data Structure

Consider a vertex  $v$  with children  $u_1, \dots, u_k$  in  $\text{dag}(S)$ . Let  $\text{pre}(v, i)$  denote the prefix of  $\text{str}(v)$  given by concatenating the strings represented by the first  $i$  children of  $v$  and let  $\text{suf}(v, i)$  be the suffix of  $\text{str}(v)$  given by concatenating the strings represented by the last  $k - i$  children of  $x$ .

The data structure is composed of two z-fast tries (see Lemma 3.4)  $T_1$  and  $T_2$  and a 2D-range reporting data structure  $R$ .

For every non-leaf node  $v \in \text{dag}(S)$  we store the following. Let  $k$  be the number of children of  $v$  if  $v$  is a run-free node otherwise let  $k = 2$ :

- The reverse of the strings  $pre(v, i)$  for  $i \in [k - 1]$  in the z-fast trie  $T_1$ .
- The strings  $suf(v, i)$  for  $i \in [k - 1]$  in the z-fast trie  $T_2$ .
- The points  $(a, b)$  where  $a$  is the rank of the reverse of  $pre(v, i)$  in  $T_1$  and  $b$  is the rank of  $suf(v, i)$  in  $T_2$  for  $i \in [k - 1]$  are stored in  $R$ . A point stores the vertex  $v \in dag(S)$  and the length of  $pre(v, i)$  as auxiliary information.

There are  $O(z \lg(n/z))$  vertices in  $dag(S)$  thus  $T_1$  and  $T_2$  take no more than  $O(z \lg(n/z))$  words of space using Lemma 3.4. There  $O(z \lg(n/z))$  points in  $R$  which takes  $O(z \lg(n/z))$  space using Lemma 3.3 (i) thus the total space in words is  $O(z \lg(n/z))$ .

### 3.4.2 Searching

Assume in the following that there are no fingerprint collisions. Compute all the prefix fingerprints of  $P$   $\phi(P[1]), \phi(P[1, 2]), \dots, \phi(P[1, m])$ . Consider the signature tree  $sig(P)$  for  $P$ . Let  $l_i^k$  denote the  $k$ 'th left-most vertex on level  $i$  in  $sig(P)$  and let  $j$  be the last level. Let  $P_L = \{|str(l_1^1)|, |str(l_1^1)| + |str(l_1^2)|, |str(l_2^1)|, |str(l_2^1)| + |str(l_2^2)|, \dots, |str(l_j^1)|, |str(l_j^1)| + |str(l_j^2)|\}$ . Symmetrically, let  $r_i^k$  denote the  $k$ 'th right-most vertex on level  $i$  in  $sig(P)$  and let  $P_R = \{m - |str(r_1^1)|, m - |str(r_1^1)| - |str(r_1^2)|, m - |str(r_2^1)|, m - |str(r_2^1)| - |str(r_2^2)|, \dots, m - |str(r_j^1)|, m - |str(r_j^1)| - |str(r_j^2)|\}$ . Let  $P_S = P_L \cup P_R$ .

For  $p \in P_S$  search for the reverse of  $P[1, p]$  in  $T_1$  and for  $P[p + 1, m]$  in  $T_2$  using the precomputed fingerprints. Let  $[a, b]$  and  $[c, d]$  be the respective ranges returned by the search. Do a range reporting query for the (possibly empty) range  $[a, b] \times [c, d]$  in  $R$ . Each point in the range identifies a node  $v$  and a position  $i$  such that  $P$  occurs at position  $i$  in the string  $str(v)$ . If  $v$  is a run node, there is furthermore an occurrence of  $P$  in  $str(v)$  for all positions  $i + k \cdot |str(child(v))|$  where  $k = 1, \dots, j$  and  $j \cdot |str(child(v))| + m \leq str(v)$ .

To report the actual occurrences of  $P$  in  $S$  we traverse all ancestors of  $v$  in  $dag(S)$ ; for each occurrence of  $P$  in  $str(v)$  found, recursively visit each parent  $u$  of  $v$  and offset the location of the occurrence to match the location in  $str(u)$  instead of  $str(v)$ . When  $u$  is the root, report the occurrence. Observe that the time it takes to traverse the ancestors of  $v$  is linear in the number of occurrences we find.

We now describe how to handle fingerprint collisions. Given a z-fast trie, Gagie et al. [56] show how to perform  $k$  weak prefix queries and identify all false positives using  $O(k \lg m + m)$  extra time by employing bookmarked extraction and

bookmarked fingerprinting. Because we only compute fingerprints and extract prefixes (suffixes) of the strings represented by vertices in  $dag(S)$  we do not need bookmarking to do this. We refer the reader to [56] for the details. Thus, we modify the search algorithm such that all the searches in  $T_1$  and  $T_2$  are carried out first, then we verify the results before progressing to doing range reporting queries only for ranges that were not discarded during verification.

### 3.4.3 Correctness

For any occurrence  $S[l, r]$  of  $P$  in  $S$  there is a node  $v$  in  $sig(S)$  that stabs  $S[l, r]$ , i.e. a suffix of  $pre(v, i)$  equals a prefix  $P[1, j]$  and a prefix of  $suf(v, i)$  equals the remaining suffix  $P[j + 1, m]$  for some  $i$  and  $j$ . Since we put all combinations of  $pre(v, i)$ ,  $suf(v, i)$  into  $T_1, T_2$  and  $R$ , we would be guaranteed to find all nodes  $v$  that contains  $P$  in  $str(v)$  if we searched for all possible split-points  $1, \dots, m - 1$  of  $P$  i.e.  $P[1, i]$  and  $P[i + 1, m]$  for  $i = 1, \dots, m - 1$ .

We now argue that we do not need to search for all possible split-points of  $P$  but only need to consider those in the set  $P_S$ . For a position  $i$ , we say the node  $v$  stabs  $i$  if the nearest common ancestor of the  $i^{th}$  and  $i + 1^{th}$  leaf of  $sig(S)$  denoted  $NCA(l_i, l_{i+1})$  is  $v$ .

Look at any occurrence  $S[l, r]$  of  $P$ . Consider  $T_S = T(l, r)$  and  $T_P = sig(P)$ . Look at a possible split-point  $i \in [1, m - 1]$  and the node  $v$  that stabs position  $i$  in  $T_P$ . Let  $u_l$  and  $u_r$  be adjacent children of  $v$  such that the rightmost leaf descendant of  $u_l$  is the  $i^{th}$  leaf and the leftmost leaf descendant of  $u_r$  is the  $i + 1^{th}$  leaf. We now look at two cases for  $v$  and argue it is irrelevant to consider position  $i$  as split-point for  $P$  in these cases:

1. **Case  $v$  is consistent (in respect to  $T_P$ ).** In this case it is guaranteed that the node that stabs  $l + i$  in  $T_S$  is identical to  $v$ . Since  $v$  is a descendant of the root of  $T_P$  (as the root of  $T_P$  is inconsistent)  $str(v)$  cannot contain  $P$  and thus it is irrelevant to consider  $i$  as a split-point.
2. **Case  $v$  is inconsistent and  $u_l$  and  $u_r$  are both consistent (in respect to  $T_P$ ).** In this case  $u_l$  and  $u_r$  have identical corresponding nodes  $u'_l$  and  $u'_r$  in  $T_S$ . Because  $u_l$  and  $u_r$  are children of the same node it follows that  $u'_l$  and  $u'_r$  must also both be children of some node  $v'$  that stabs  $l + i$  in  $T_S$  (however  $v$  and  $v'$  may not be identical since  $v$  is inconsistent). Consider the node  $u'_{l_l}$  to the left of  $u'_l$  (or symmetrically for the right side if  $v$  is an inconsistent node in the right side of  $T_P$ ). If  $str(v')$  contains  $P$  then  $u'_{l_l}$  is also a child of  $v'$  (otherwise  $u_l$  would be inconsistent). So it suffices

to check the split-point  $i - |u_l|$ . Surely  $i - |u_l|$  stabs an inconsistent node in  $T_P$ , so either we consider that position relevant, or the same argument applies again and a split-point further to the left is eventually considered relevant.

Thus only split-points where  $v$  and at least one of  $u_l$  or  $u_r$  are inconsistent are relevant. These positions are a subset of the position in  $P_S$ , and thus we try all relevant split-points.

### 3.4.4 Complexity

A query on  $T_1$  and  $T_2$  takes  $O(\lg m)$  time by Lemma 3.4 while a query on  $R$  takes  $O(\lg^\epsilon z)$  time using Lemma 3.3 (i) (excluding reporting). We do  $O(\lg m)$  queries as the size of  $P_S$  is  $O(\lg m)$ . Verification of the  $O(\lg m)$  strings we search for takes total time  $O(\lg^2 m + m) = O(m)$ . Constructing the signature DAG for  $P$  takes  $O(m)$  time, thus total time without reporting is  $O(m + \lg m \lg^\epsilon z) = O(m + \lg^{\epsilon'} z)$  for any  $\epsilon' > \epsilon$ . This holds because if  $m \leq \lg^{2\epsilon} z$  then  $\lg m \lg^\epsilon z \leq \lg \lg^{2\epsilon} z \lg^\epsilon z = O(\lg^{\epsilon'} z)$ , otherwise  $m > \lg^{2\epsilon} z \Leftrightarrow \sqrt{m} > \lg^\epsilon z$  and then  $\lg m \lg^\epsilon z = O(\lg m \sqrt{m}) = O(m)$ . For every query on  $R$  we may find multiple points each corresponding to an occurrence of  $P$ . It takes  $O(\lg^\epsilon z)$  time to report each point thus the total time becomes  $O(m + (1 + \text{occ}) \lg^{\epsilon'} z)$ .

## 3.5 Short Patterns

Our solution for short patterns uses properties of the LZ77-parse of  $S$ . A *primary* substring of  $S$  is a substring that contains one or more borders of  $S$ , all other substrings are called *secondary*. A primary substring that matches a query pattern  $P$  is a *primary occurrence* of  $P$  while a secondary substring that matches  $P$  is a *secondary occurrence* of  $P$ . In a seminal paper on LZ77 based indexing [82] Kärkkäinen and Ukkonen use some observations by Farach and Thorup [38] to show how all secondary occurrences of a query pattern  $P$  can be found given a list of the primary occurrences of  $P$  through a reduction to orthogonal range reporting. Employing the range reporting result given in Lemma 3.3 (ii), all secondary occurrences can be reported as stated in the following lemma:

**LEMMA 3.12 (KÄRKKÄINEN AND UKKONEN [82])** *Given the LZ77-parse of a string  $S$  there exists a data structure that uses  $O(z \lg \lg z)$  space that can report all secondary occurrences of a pattern  $P$  given the list of primary occurrences of  $P$  in  $S$  in  $O(\text{occ} \lg \lg n)$  time.*

We now describe a data structure that can report all primary occurrences of a pattern  $P$  of length at most  $k$  in  $O(m + \text{occ})$  time using  $O(zk)$  space.

**LEMMA 3.13** *Given a string  $S$  of length  $n$  and a positive integer  $k \leq n$  we can build a compressed index supporting pattern matching queries for patterns of length  $m$  in  $O(m + \text{occ} \lg \lg n)$  time using  $O(zk + z \lg \lg z)$  space that works for  $m \leq k$ .*

PROOF. Consider the set  $C$  of  $z$  substrings of  $S$  that are defined by  $S[u_i - k, u_i + k - 1]$  for  $i \in [z]$ , ie. the substrings of length  $2k$  surrounding the borders of the LZ77-parse. The total length of these strings is  $\Theta(zk)$ . Construct the generalized suffix tree  $T$  over the set of strings  $C$ . This takes  $\Theta(zk)$  words of space. To ensure no occurrence is reported more than once, if multiple suffixes in this generalized suffix tree correspond to substrings of  $S$  that starts on the same position in  $S$ , only include the longest of these. This happens when the distance between two borders is less than  $2k$ .

To find the primary occurrences of  $P$  of length  $m$ , simply find all occurrences of  $P$  in  $T$ . These occurrences are a super set of the primary occurrences of  $P$  in  $S$ , since  $T$  contains all substrings starting/ending at most  $k$  positions from a border. It is easy to filter out all occurrences that are not primary, simply by calculating if they cross a border or not. This takes  $O(m + \text{occ})$  time (where  $\text{occ}$  includes secondary occurrences). Combined with Lemma 3.12 this gives Lemma 3.13.

## 3.6 Semi-Short Patterns

In this section, we show how to handle patterns of length between  $\lg \lg z$  and  $\lg^\epsilon z$ . It is based on the same reduction to 2D-range reporting as used for long patterns. However, the positions in  $S$  that are inserted in the range reporting structure is now based on the LZ77-parse of  $S$  instead. Furthermore we use Lemma 3.3 (ii) which gives faster range reporting but uses super-linear space, which is fine because we instead put fewer points into the structure. We get the following lemma:

**LEMMA 3.14** *Given a string  $S$  of length  $n$  we solve the compressed indexing problem for a pattern  $P$  of length  $m$  with  $\lg \lg z \leq m \leq \lg^\epsilon z$  for any positive constant  $\epsilon < \frac{1}{2}$  in  $O(m + \text{occ}(\lg \lg n + \lg^\epsilon z))$  time using  $O(z(\lg \lg z + \log(n/z)))$  space.*

### 3.6.1 Data Structure

As in the previous section for short patterns, we only need to worry about primary occurrences of  $P$  in  $S$ . Let  $B$  be the set of all substrings of length at most  $\lg^\epsilon z$  that cross a border in  $S$ . The split positions of such a string are the offsets of the leftmost borders in its occurrences. All primary occurrences of  $P$  in  $S$  are in this set. The size of this set is  $|B| = O(z \lg^{2\epsilon} z)$ . The data structure is composed by the following:

- A dictionary  $H$  mapping each string in  $B$  to its split positions.
- A z-fast trie  $T_1$  on the reverse of the strings  $T[u_i, l_i]$  for  $i \in [z]$ .
- A z-fast trie  $T_2$  on the strings  $T[u_i, n]$  for  $i \in [z]$ .
- A range reporting data structure  $R$  with a point  $(c, d)$  for every pair of strings  $C_i = T[u_i, l_i], D_i = T[u_{i+1}, n]$  for  $i \in [z]$  where  $D_z = \epsilon$  and  $c$  is the lexicographical rank of the reverse of  $C_i$  in the set  $\{C_1, \dots, C_z\}$  and  $d$  is the lexicographical rank of  $D_i$  in the set  $\{D_1, \dots, D_z\}$ . We store the border  $u_i$  along with the point  $(c, d)$ .
- The data structure described in Lemma 3.12 to report secondary occurrences.
- The signature grammar for  $S$ .

Each entry in  $H$  requires  $\lg \lg^\epsilon z = O(\lg \lg z)$  bits to store since a split position can be at most  $\lg^\epsilon z$ . Thus the dictionary can be stored in  $O(|B| \cdot \lg \lg z) = O(z \lg^{2\epsilon} z \lg \lg z)$  bits which for  $\epsilon < \frac{1}{2}$  is  $O(z)$  words. The tries  $T_1$  and  $T_2$  take  $O(z)$  space while  $R$  takes  $O(z \lg \lg z)$  space. The signature grammar takes  $O(z \log(n/z))$ . Thus the total space is  $O(z(\lg \lg z + \log(n/z)))$ .

### 3.6.2 Searching

Assume a lookup for  $P$  in  $H$  does not give false-positives. Given a pattern  $P$  compute all prefix fingerprints of  $P$ . Next do a lookup in  $H$ . If there is no match then  $P$  does not occur in  $S$ . Otherwise, we do the following for each of the split-points  $s$  stored in  $H$ . First split  $P$  into a left part  $P_l = P[0, s-1]$  and a right part  $P_r = P[s, m]$ . Then search for the reverse of  $P_l$  in  $T_1$  and for  $P_r$  in  $T_2$  using the corresponding fingerprints. The search induces a (possibly empty) range for which we do a range reporting query in  $R$ . Each occurrence in



$R$  corresponds to a primary occurrence of  $P$  in  $S$ , so report these. Finally use Lemma 3.12 to report all secondary occurrences.

Unfortunately, we cannot guarantee a lookup for  $P$  in  $H$  does not give a false positive. Instead, we pause the reporting step when the first possible occurrence of  $P$  has been found. At this point, we verify the substring  $P$  matches the found occurrence in  $S$ . We know this occurrence is around an LZ-border in  $S$  such that  $P_l$  is to the left of the border and  $P_r$  is to the right of the border. Thus we can efficiently verify that  $P$  actually occurs at this position using the grammar.

### 3.6.3 Analysis

Computing the prefix fingerprints of  $P$  takes  $O(m)$  time. First, we analyze the running time in the case  $P$  actually exists in  $S$ . The lookup in  $H$  takes  $O(1)$  time using perfect hashing. For each split-point we do two z-fast trie lookups in time  $O(\lg m) = O(\lg \lg z)$ . Since each different split-point corresponds to at least one unique occurrence, this takes at most  $O(\text{occ} \lg \lg z)$  time in total. Similarly each lookup and occurrence in the 2D-range reporting structure takes  $\lg \lg z$  time, which is therefore also bounded by  $O(\text{occ} \lg \lg z)$  time. Finally, we verified one of the found occurrence against  $P$  in  $O(m)$  time. So the total time is  $O(m + \text{occ} \lg \lg z)$  in this case.

In the case  $P$  does not exist, either the lookup in  $H$  tells us that, and we spend  $O(1)$  time, or the lookup in  $H$  is a false-positive. In the latter case, we perform exactly two z-fast trie lookups and one range reporting query. These all take time  $O(\lg \lg z)$ . Since  $m \geq \lg \lg z$  this is  $O(m)$  time. Again, we verified the found occurrence against  $P$  in  $O(m)$  time. The total time in this case is therefore  $O(m)$ .

Note we ensure our fingerprint function is collision free for all substrings in  $B$  during the preprocessing thus there can only be collisions if  $P$  does not occur in  $S$  when  $m \leq \lg^\epsilon z$ .

## 3.7 Randomized Solution

In this section we present a very simple way to turn the  $O(m + (1 + \text{occ}) \lg^\epsilon z)$  worst-case time of Lemma 3.11 into  $O(m + \text{occ} \lg^\epsilon z)$  expected time. First observe, this is already true if the pattern we search for occurs at least once or if  $m \geq \lg^\epsilon z$ .

As in the semi-short patterns section, we consider the set  $B$  of substrings of  $S$  of length at most  $\lg^\epsilon z$  that crosses a border. Create a dictionary  $H$  with  $z \lg^{3\epsilon} z$  entries and insert all the strings from  $B$ . This means only a  $\lg^\epsilon z$  fraction of the entries are used, and thus if we lookup a string  $s$  (where  $|s| \leq \lg^\epsilon z$ ) that is not in  $H$  there is only a  $\frac{1}{\lg^\epsilon z}$  chance of getting a false-positive.

Now to answer a query, we first check if  $m \leq \lg^\epsilon z$  in which case we look it up in  $H$ . If it does not exist, report that. If it does exist in  $H$  or if  $m > \lg^\epsilon z$  use the solution from Lemma 3.11 to answer the query.

In the case  $P$  does not exist, we spend either  $O(m)$  time if  $H$  reports no, or  $O(m + \lg^\epsilon z)$  time if  $H$  reports a false-positive. Since there is only  $\frac{1}{\lg^\epsilon z}$  chance of getting a false positive, the expected time in this case is  $O(m)$ . In all other cases, the running time is  $O(m + \text{occ} \lg^\epsilon z)$  in worst-case, so the total expected running time is  $O(m + \text{occ} \lg^\epsilon z)$ . The space usage of  $H$  is  $O(z \lg^{3\epsilon} z)$  bits since we only need to store one bit for each entry. This is  $O(z)$  words for  $\epsilon \leq 1/3$ . To sum up, we get the following lemma:

**LEMMA 3.15** *Given a signature grammar for a text  $S$  of length  $n$  with an LZ77-parse of length  $z$  we can build a compressed index supporting pattern matching queries in  $O(m + \text{occ} \lg^\epsilon z)$  expected time using  $O(z \lg(n/z))$  space for any constant  $0 < \epsilon \leq 1/3$ .*



# Dynamic Relative Compression, Dynamic Partial Sums, and Substring Concatenation

---

Philip Bille <sup>†</sup>      Anders Roy Christiansen <sup>†</sup>      Patrick Hagge Cording <sup>†</sup>  
 Inge Li Gørtz <sup>†</sup>      Frederik Rye Skjoldjensen <sup>†</sup>      Hjalte Wedel Vildhøj <sup>†</sup>  
 Søren Vind <sup>†</sup>

<sup>†</sup> The Technical University of Denmark

## Abstract

Given a static reference string  $R$  and a source string  $S$ , a relative compression of  $S$  with respect to  $R$  is an encoding of  $S$  as a sequence of references to substrings of  $R$ . Relative compression schemes are a classic model of compression and have recently proved very successful for compressing highly-repetitive massive data sets such as genomes and web-data. We initiate the study of relative compression in a dynamic setting where the compressed source string  $S$  is subject to edit operations. The goal is to maintain the compressed representation compactly, while supporting edits and allowing efficient random access to the (uncompressed) source string. We present new data structures that achieve optimal time for updates and queries while using space linear in the size of the optimal relative compression, for nearly all combinations of parameters. We also present solutions

for restricted and extended sets of updates. To achieve these results, we revisit the dynamic partial sums problem and the substring concatenation problem. We present new optimal or near optimal bounds for these problems. Plugging in our new results we also immediately obtain new bounds for the string indexing for patterns with wildcards problem and the dynamic text and static pattern matching problem.

## 4.1 Introduction

Given a static reference string  $R$  and a source string  $S$ , a *relative compression of  $S$  with respect to  $R$*  is an encoding of  $S$  as a sequence of references to substrings of  $R$ . Relative compression (or *external macro compression*) is a classic model of compression defined by Storer and Szymanski [123, 124] in 1978 and has since been used in a wide range of compression scenarios [26, 37, 73, 91, 92, 96, 97]. To compress massive highly-repetitive data sets, such as biological sequences and web collections, relative compression has been shown to be very practical [73, 91, 92].

Relative compression is often applied to compress multiple similar source strings. In such settings relative compression is superior to compressing the source strings individually. For instance, human genomes are 99% similar and hence relative compression might be used to compress a large collection of sequenced genomes using, e.g., the human reference genome as the static reference string. We focus on the case of compressing a single source string, but our results trivially generalize to compressing multiple source strings.

In this paper we initiate the study of relative compression in a *dynamic setting*, where the compressed source string  $S$  is subject to edit operations (insertions, deletions, and replacements of single characters). The goal is to maintain the compressed representation compactly, while supporting edits and allowing efficient random access to the (uncompressed) source string. Efficient data structures supporting these operations allow us to avoid costly recompression of massive data sets after updates.

We provide the first non-trivial bounds for this problem. We present new data structures that achieve *optimal* time for updates and queries while using space linear in the size of the *optimal* relative compression, for nearly all combinations of parameters. We also present solutions for restricted and extended sets of updates.

To achieve these results, we revisit the *dynamic partial sums problem* and the

*substring concatenation problem*. We present new optimal or near optimal bounds for both of these problems (see detailed discussion below). Furthermore, plugging in our new results immediately leads to new bounds for the *string indexing for patterns with wildcards problem* [17, 95] and the *dynamic text and static pattern matching problem* [4].

### 4.1.1 Dynamic Relative Compression

Given a *reference string*  $R$  and a *source string*  $S$ , a *relative compression* of  $S$  with respect to  $R$  is a sequence  $C = (i_1, j_1), \dots, (i_{|C|}, j_{|C|})$  such that  $S = R[i_1, j_1] \cdots R[i_{|C|}, j_{|C|}]$ . We call  $C$  a *substring cover* for  $S$ . The substring cover is *optimal* if  $|C|$  is minimum over all relative compressions of  $S$  with respect to  $R$ . The *dynamic relative compression problem* is to maintain a relative compression of  $S$  under the following operations. Let  $i$  be a position in  $S$  and  $\alpha$  be a character.

- access( $i$ ):** return the character  $S[i]$ ,
- replace( $i, \alpha$ ):** change  $S[i]$  to character  $\alpha$ ,
- insert( $i, \alpha$ ):** insert character  $\alpha$  before position  $i$  in  $S$ ,
- delete( $i$ ):** delete the character at position  $i$  in  $S$ .

Note that operations **insert** and **delete** change the length of  $S$  by a single character. In all bounds below, the **access( $i$ )** operation extends to decompressing an arbitrary substring of length  $\ell$  using only  $O(\ell)$  additional time.

**Our Results** Throughout the paper, let  $r$  be the length of the reference string  $R$ ,  $N$  be the length of the (uncompressed) string  $S$ , and  $n$  be the size of an optimal relative compression of  $S$  with regards to  $R$ . All of the bounds mentioned below and presented in this paper hold for a standard unit-cost RAM with  $w$ -bit words with standard arithmetic/logical operations on words and where space is measured in words. This means that the algorithms can be implemented directly in standard imperative programming languages such as C [87] or C++ [125]. We assume that an index into  $S$  or  $R$  can be stored in a single word and hence  $w \geq \log(N + r)$ .

**THEOREM 4.1** *Let  $R$  and  $S$  be a reference and source string of lengths  $r$  and  $N$ , respectively, and let  $n$  be the length of the optimal substring cover of  $S$  by  $R$ . Then, we can solve the dynamic relative compression problem supporting access, replace, insert, and delete*

- (i) in  $O(n + r)$  space and  $O\left(\frac{\log n}{\log \log n} + \log \log r\right)$  time per operation, or
- (ii) in  $O(n + r \log^\epsilon r)$  space and  $O\left(\frac{\log n}{\log \log n}\right)$  time per operation, for any constant  $\epsilon > 0$ .

These are the first non-trivial bounds for the problem. Together, the bounds are optimal for most natural parameter combinations. In particular, any data structure for a string of length  $N$  supporting **access**, **insert**, and **delete** must use  $\Omega(\log N / \log \log N)$  time in the worst-case regardless of the space [51] (this is called the *list representation problem*). Since  $n \leq N$ , we can view  $O(\log n / \log \log n)$  as a compressed version of the optimal time bound that is always  $O(\log N / \log \log N)$  and better when  $S$  is compressible. Hence, Theorem 4.1(i) provides a linear-space solution that achieves the compressed time bound except for an  $O(\log \log r)$  additive term. Note that whenever  $n \geq (\log r)^{\log \log \log r}$ , the  $\log n / \log \log n$  term dominates the query time and we match the compressed time bound. Hence, Theorem 4.1(i) is only suboptimal in the special case when  $n$  is almost exponentially smaller than  $r$ . In this case, we can use Theorem 4.1(ii) which always provides a solution achieving the compressed time bound at the cost of increasing the space to  $O(n + r \log^\epsilon r)$ . We could easily generalize Theorem 4.1 to compress multiple source strings with respect to the same reference string. Since every source string can share the same data structure build on  $R$  we would only increase the space bounds with the size of the relative compression of the new source strings.

We note that dynamic compression under different models of compression has been studied extensively [41–43, 67, 78, 103, 118]. However, all of these results require space dependent on the size of the original string and hence cannot take full advantage of highly-repetitive data.

### 4.1.2 Dynamic Partial Sums

The *partial sums problem* is to maintain an array  $Z[1..s]$  under the following operations.

**sum( $i$ ):** return  $\sum_{j=1}^i Z[j]$ .

**update( $i, \Delta$ ):** set  $Z[i] = Z[i] + \Delta$  given that  $Z[i] + \Delta \geq 0$ .

**search( $t$ ):** return  $1 \leq i \leq s$  such that  $\text{sum}(i-1) < t \leq \text{sum}(i)$ . To ensure well-defined answers, we require that  $Z[i] \geq 0$  for all  $i$ .

The partial sums problem is a classic and well-studied problem [34, 40, 51, 71, 74, 75, 109, 114]. In our context, we consider the problem in the word RAM model, where each array entry stores a  $w$ -bit integer and the element of the array can be changed by  $\delta$ -bit integers, i.e., the argument  $\Delta$  can be stored in  $\delta$  bits. To allow  $\Delta$  to take negative values we store a sign bit in addition to the  $\delta$  bits. In this setting, Pătraşcu and Demaine [109] gave a linear-space data structure with  $\Theta(\log s / \log(w/\delta))$  time per operation. They also gave a matching lower bound.

We consider the following generalization supporting dynamic changes to the array. The *dynamic partial sums problem* is to additionally support the following operations.

**insert( $i, \Delta$ ):** insert a new entry in  $Z$  with value  $\Delta \geq 0$  before  $Z[i]$ ,

**delete( $i$ ):** delete the entry  $Z[i]$  of value at most  $\Delta$ .

**merge( $i$ ):** replace entry  $Z[i]$  and  $Z[i + 1]$  with a new entry with value  $Z[i] + Z[i + 1]$ .

**divide( $i, t$ ):** replace entry  $Z[i]$  by two new consecutive entries with value  $t$  and  $Z[i] - t$ , respectively, where  $0 \leq t \leq Z[i]$ .

Hon et al. [71] and Navarro and Sadakane [105] presented optimal solutions for this problem in the case where the entries in  $Z$  are at most polylogarithmic in  $s$  (they did not explicitly consider the **merge** and **divide** operation).

**Our Results** We show the following improved result.

**THEOREM 4.2** *Given an array of length  $s$  storing  $w$ -bit integers and parameter  $\delta$ , such that  $-2^\delta < \Delta < 2^\delta$ , we can solve the dynamic partial sums problem supporting **sum**, **update**, **search**, **insert**, **delete**, **merge**, and **divide** in linear space and  $O(\log s / \log(w/\delta))$  time per operation.*

Note that this bound simultaneously matches the optimal time bound for the standard partial sums problem and supports storing arbitrary  $w$ -bit values in the entries of the array, i.e., the values we can handle in optimal time are exponentially larger than in the previous results.

To achieve our bounds we extend the static solution by Pătraşcu and Demaine [109]. Their solution is based on storing a sampled subset of *representative*



*elements* of the array and difference encode the remaining elements. They pack multiple difference encoded elements in words and then apply word-level parallelism to speedup the operations. To support **insert** and **delete** the main challenge is to maintain the representative elements that now dynamically move within the array. We show how to efficiently do this by combining a new representation of representative elements with a recent result by Pătraşcu and Thorup [111]. Along the way we also slightly simplify the original construction by Pătraşcu and Demaine [109].

### 4.1.3 Substring Concatenation

Let  $R$  be a string of length  $r$ . A *substring concatenation query* on  $R$  takes two pairs of indices  $(i, j)$  and  $(i', j')$  and returns the start position in  $R$  of an occurrence of  $R[i, j]R[i', j']$ , or **NO** if the string is not a substring of  $R$ . The *substring concatenation problem* is to preprocess  $R$  into a data structure that supports substring concatenation queries.

Amir et al. [4] gave a solution using  $O(r\sqrt{\log r})$  space with query time  $O(\log \log r)$ , and recently Gawrychowski et al. [62] showed how to solve the problem in  $O(r \log r)$  space and  $O(1)$  time.

**Our Results** We give the following improved bounds.

**THEOREM 4.3** *Given a string  $R$  of length  $r$ , the substring concatenation problem can be solved in either*

- (i)  $O(r \log^\epsilon r)$  space and  $O(1)$  time, for any constant  $\epsilon > 0$ , or
- (ii)  $O(r)$  space and  $O(\log \log r)$  time.

Hence, Theorem 4.3(i) matches the previous  $O(1)$  time bound while reducing the space from  $O(r \log r)$  to  $O(r \log^\epsilon r)$  and Theorem 4.3(ii) achieves linear space while using  $O(\log \log r)$  time. Plugging in the two solutions into our solution for dynamic relative compression leads to the two branches of Theorem 4.1.

To achieve the bound in (i), the main idea is a new construction that efficiently combines compact data structure for weak prefix serach [10] with the recent constant time weighted level ancestor data structure for suffix trees [62]. The bound in (ii) follows as a simple implication of another recent result for *unrooted*

*LCP queries* [17] by some of the authors. The substring concatenation problem is a key component in several solutions to the *string indexing for patterns with wildcards problem* [17, 28, 95], where the goal is to preprocess a string  $T$  to support pattern matching queries for patterns with wildcards. Plugging in Theorem 4.3(i) we immediately obtain the following new bound for the problem.

**COROLLARY 4.4** *Let  $T$  be a string of length  $t$ . For any pattern string  $P$  of length  $p$  with  $k$  wildcards, we can support pattern matching queries on  $T$  using  $O(t \log^\epsilon t)$  space and  $O(p + \sigma^k)$  time for any constant  $\epsilon > 0$ .*

This improves the running time of fastest linear space solution by a factor  $\log \log t$  at the cost of increasing the space slightly by a factor  $\log^\epsilon t$ . See [95] for detailed overview of the known results.

#### 4.1.4 Extensions

Finally, we present two extensions of the dynamic relative compression problem.

##### 4.1.4.1 Dynamic Relative Compression with Access and Replace

If we restrict the operations to **access** and **replace** we obtain the following improved bound.

**THEOREM 4.5** *Let  $R$  and  $S$  be a reference and source string of lengths  $r$  and  $N$ , respectively, and let  $n$  be the length of the optimal substring cover of  $S$  by  $R$ . Then, we can solve the dynamic relative compression problem supporting **access** and **replace** in  $O(n + r)$  space and  $O(\log \log N)$  expected time.*

This version of dynamic relative compression is a key component in the *dynamic text and static pattern matching problem*, where the goal is to efficiently maintain a set of occurrences of a pattern  $P$  in a text  $T$  that is dynamically updated by changing individual characters. Let  $p$  and  $t$  denote the lengths of  $P$  and  $T$ , respectively. Amir et al. [4] gave a data structure using  $O(t + p\sqrt{\log p})$  space which supports updates in  $O(\log \log p)$  time. The computational bottleneck in the update operation is to update a substring cover of size  $O(p)$ . Plugging in the bounds from Theorem 4.5, we immediately obtain the following improved bound.

**COROLLARY 4.6** *Given a pattern  $P$  and text  $T$  of lengths  $p$  and  $t$ , respectively, we can solve the dynamic text and static pattern matching problem in  $O(t + p)$  space and  $O(\log \log p)$  expected time per update.*

Hence, we match the previous time bound while improving the space to linear.

#### 4.1.4.2 Dynamic Relative Compression with Split and Concatenate

We also consider maintaining a set of compressed strings under split and concatenate operations (as in Alstrup et al. [1]). Let  $R$  be a reference string and let  $\mathcal{S} = \{S_1, \dots, S_k\}$  be a set of strings compressed relative to  $R$ . In addition to access, replace, insert and delete we also define the following operations.

**concat( $i, j$ ):** add string  $S_i \cdot S_j$  to  $\mathcal{S}$  and remove  $S_i$  and  $S_j$ .

**split( $i, j$ ):** remove  $S_i$  from  $\mathcal{S}$  and add  $S_i[1, j - 1]$  and  $S_i[j, |S_i|]$ .

We obtain the following bounds.

**THEOREM 4.7** *Let  $R$  be a reference string of length  $r$ , let  $\mathcal{S} = \{S_1, \dots, S_k\}$  be a set of source strings of total length  $N$ , and let  $n$  be the total length of the optimal substring covers of the strings in  $\mathcal{S}$ . Then, we can solve the dynamic relative compression problem supporting access, replace, insert, delete, split, and concat,*

- (i) *in space  $O(n + r)$  and time  $O(\log n)$  for access and time  $O(\log n + \log \log r)$  for replace, insert, delete, split, and concat, or*
- (ii) *in space  $O(n + r \log^\epsilon r)$  and time  $O(\log n)$  for all operations.*

Hence, compared to the bounds in Theorem 4.1 we only increase the time bounds by an additional  $\log \log n$  factor.

## 4.2 Dynamic Relative Compression

In this section we show how Theorems 4.2 and 4.3 lead to Theorem 4.1. The proofs of Theorems 4.2 and 4.3 appear in Section 4.3 and Section 4.4, respectively.

Let  $C = ((i_1, j_1), \dots, (i_{|C|}, j_{|C|}))$  be the compressed representation of  $S$ . From now on, we refer to  $C$  as the *cover* of  $S$ , and call each element  $(i_l, j_l)$  in  $C$  a *block*. Recall that a block  $(i_l, j_l)$  refers to a substring  $R[i_l, j_l]$  of  $R$ . A cover  $C$  is *minimal* if concatenating any two consecutive blocks  $(i_l, j_l), (i_{l+1}, j_{l+1})$  in  $C$  yields a string that does not occur in  $R$ , i.e., the string  $R[i_l, j_l]R[i_{l+1}, j_{l+1}]$  is not a substring of  $R$ . This definition of covers comes from Amir et al. [4]. We need the following lemma.

**LEMMA 4.8** *If  $C_{\min}$  is a minimal cover and  $C$  is an arbitrary cover of  $S$ , then  $|C_{\min}| \leq 2|C| - 1$ .*

**PROOF.** In each block  $b$  of  $C$  there can start at most two blocks in  $C_{\min}$ , because otherwise two adjacent blocks in  $C_{\min}$  would be entirely contained in the block  $b$ , contradicting the minimality of  $C_{\min}$ . Since the last block of both  $C$  and  $C_{\min}$  end at the last position of  $S$ , a contradiction of the minimality is already obtained when more than one block of  $C_{\min}$  start in the last block of  $C$ . Hence,  $|C_{\min}| \leq 2|C| - 1$ .

Recall that  $n$  is the size of an optimal cover of  $S$  with regards to  $R$ . The lemma implies that we can maintain a compression of size at most  $2n - 1$  by maintaining a minimal cover of  $S$ . The remainder of this section describes our data structure for maintaining and accessing such a cover.

Initially, we can use the suffix tree of  $R$  to construct a minimal cover of  $S$  in  $O(N + r)$  time by greedily matching the maximal prefix of the remaining part of  $S$  with any suffix of  $R$ . This guarantees that the blocks constitute a minimal cover of  $S$ .

### 4.2.1 Data Structure

The high level idea for supporting the operations on  $S$  is to store the sequence of block lengths  $j_1 - i_1 + 1, \dots, j_{|C|} - i_{|C|} + 1$  in a dynamic partial sums data structure. This allows us, for example, to identify the block that encodes the  $k^{\text{th}}$  character in  $S$  by performing a `search( $k$ )` query.

Updates to  $S$  are implemented by splitting a block in  $C$ . This may break the minimality property so we use substring concatenation queries on  $R$  to detect if blocks can be merged. We only need a constant number of substring concatenation queries to restore minimality. To maintain the correct sequence of block lengths we use `update`, `divide` and `merge` operations on the dynamic partial sums data structure.

Our data structure consist of the string  $R$ , a substring concatenation data structure of Theorem 4.3 for  $R$ , a minimal cover  $C$  for  $S$  stored in a doubly linked list, and the dynamic partial sums data structure of Theorem 4.2 storing the block lengths of  $C$ . We also store auxiliary links between a block in the doubly linked list and the corresponding block length in the partial sums data structure, and a list of alphabet symbols in  $R$  with the location of an occurrence for each symbol. By Lemma 4.8 and since  $C$  is minimal we have  $|C| \leq 2n - 1 = O(n)$ . Hence, the total space for  $C$  and the partial sums data structure is  $O(n)$ . The space for  $R$  is  $O(r)$  and the space for substring concatenation data structure is either  $O(r)$  or  $O(r \log^\epsilon r)$  depending on the choice in Theorem 4.3. Hence, in total we use either  $O(n + r)$  or  $O(n + r \log^\epsilon r)$  space.

### 4.2.2 Answering Queries

To answer  $\text{access}(i)$  queries we first compute  $\text{search}(i)$  in the dynamic partial sums structure to identify the block  $b_l = (i_l, j_l)$  containing position  $i$  in  $S$ . The local index in  $R[i_l, j_l]$  of the  $i^{\text{th}}$  character in  $R$  is  $\ell = i - \text{sum}(l - 1)$ , and thus the answer to the query is the character  $R[i_l + \ell - 1]$ .

We perform **replace** and **delete** by first identifying  $b_l = (i_l, j_l)$  and  $\ell$  as above. Then we partition  $b_l$  into three new blocks  $b_l^1 = (i_l, i_l + \ell - 2)$ ,  $b_l^2 = (i_l + \ell - 1, i_l + \ell - 1)$ ,  $b_l^3 = (i_l + \ell, j_l)$  where  $b_l^2$  is the single character block for index  $i$  in  $S$  that we must change. In **replace** we change  $b_l^2$  to an index of an occurrence in  $R$  of the new character (which we can find from the list of alphabet symbols), while we remove  $b_l^2$  in **delete**. The new blocks and their neighbors, that is,  $b_{l-1}$ ,  $b_l^1$ ,  $b_l^2$ ,  $b_l^3$ , and  $b_{l+1}$  may now be non-minimal. To restore minimality we perform substring concatenation queries on each consecutive pair of these 5 blocks, and replace non-minimal blocks with merged minimal blocks. All other blocks are still minimal, since the strings obtained by concatenating  $b_{l'}$  with  $b_{l'+1}$ , for all  $l' < l - 1$  and all  $l' > l$ , were not present in  $R$  before the change and are not present afterwards. A similar idea is used by Amir et al. [4]. We perform **update**, **divide** and **merge** operations to maintain the corresponding lengths in the dynamic partial sums data structure. The **insert** operation is similar, but inserts a new single character block between two parts of  $b_l$  before restoring minimality. Observe that using  $\delta = O(1)$  bits in **update** is sufficient to maintain the correct block lengths.

In total, each operation requires a constant number of substring concatenation queries and dynamic partial sums operations; the latter having time complexity  $O(\log n / \log(w/\delta)) = O(\log n / \log \log n)$  as  $w \geq \log n$  and  $\delta = O(1)$ . Hence, the total time for each **access**, **replace**, **insert**, and **delete** operation is either  $O(\log n / \log \log n + \log \log r)$  or  $O(\log n / \log \log n)$  depending on the substring

concatenation data structure used. In summary, this proves Theorem 4.1.

## 4.3 Dynamic Partial Sums

In this section we prove Theorem 4.2. We support the operations  $\text{insert}(i, \Delta)$  and  $\text{delete}(i)$  on a sequence of  $w$ -bit integer keys by implementing them using  $\text{update}$  and a  $\text{divide}$  or  $\text{merge}$  operation, respectively. This means that we support inserting or deleting keys with value at most  $2^\delta$ .

We first solve the problem for small sequences. The general solution uses a standard reduction, storing  $Z$  at the leaves of a B-tree of large outdegree. We use the solution for small sequences to navigate in the internal nodes of the B-tree.

**Dynamic Integer Sets** We need the following recent result due to Pătraşcu and Thorup [111] on maintaining a set of integer keys  $X$  under insertions and deletions. The queries are as follows, where  $q$  is an integer. The membership query  $\text{member}_X(q)$  returns true if  $q \in X$ , predecessor  $\text{pred}_X(q)$  returns the largest key  $x \in X$  where  $x < q$ , and successor  $\text{succ}_X(q)$  returns the smallest key  $x \in X$  where  $x \geq q$ . The rank  $\text{rank}_X(q)$  returns the number of keys in  $X$  smaller than  $q$ , and  $\text{select}_X(i)$  returns the  $i^{\text{th}}$  smallest key in  $X$ .

**LEMMA 4.9 (PĂTRAŞCU AND THORUP [111])** *There is a data structure for maintaining a dynamic set of  $n \leq w^{O(1)}$   $w$ -bit integers in  $O(n)$  space that supports insert, delete, membership, predecessor, successor, rank and select in constant time per operation.*

### 4.3.1 Dynamic Partial Sums for Small Sequences

Let  $Z$  be a sequence of at most  $B \leq w^{O(1)}$  integer keys. We will show how to store  $Z$  in linear space such that all dynamic partial sums operations can be performed in constant time. We let  $Y$  be the sequence of prefix sums of  $Z$ , defined such that each key  $Y[i]$  is the sum of the first  $i$  keys in  $Z$ , i.e.,  $Y[i] = \sum_{j=1}^i Z[j]$ . Observe that  $\text{sum}(i) = Y[i]$  and  $\text{search}(t)$  is the index of the successor of  $t$  in  $Y$ . Our goal is to store and maintain a representation of  $Y$  subject to the dynamic operations  $\text{update}$ ,  $\text{divide}$  and  $\text{merge}$  in constant time per operation.

#### 4.3.1.1 The Scheme by Pătraşcu and Demaine

We first review the solution to the static partial sums problem by Pătraşcu and Demaine [109], slightly simplified due to Lemma 4.9. Our dynamic solution builds on this.

The entire data structure is rebuilt every  $B$  operations as follows. We first partition  $Y$  greedily into *runs*. Two adjacent elements in  $Y$  are in the same run if their difference is at most  $B \cdot 2^\delta$ , and we call the first element of each run a *representative* for all elements in the run. We use  $\mathcal{R}$  to denote the sequence of representative values in  $Y$  and  $\text{rep}(i)$  to be the index of the representative for element  $Y[i]$  among the elements in  $\mathcal{R}$ .

We store  $Y$  by splitting representatives and other elements into separate data structures:  $\mathcal{I}$  and  $\mathcal{R}$  store the representatives at the time of the last rebuild, while  $\mathcal{U}$  stores each element in  $Y$  as an offset to its representative value as well as updates since the last rebuild. We ensure  $Y[i] = \mathcal{R}[\text{rep}(i)] + \mathcal{U}[i]$  for any  $i$  and can thus reconstruct the values of  $Y$ .

The representatives are stored as follows.  $\mathcal{I}$  is the sequence of indices in  $Y$  of the representatives and  $\mathcal{R}$  is the sequence of representative values in  $Y$ . Both  $\mathcal{I}$  and  $\mathcal{R}$  are stored using the data structure of Lemma 4.9. We can then define  $\text{rep}(i) = \text{rank}_{\mathcal{I}}(\text{pred}_{\mathcal{I}}(i))$  as the index of the representative for  $i$  among all representatives, and use  $\mathcal{R}[\text{rep}(i)] = \text{select}_{\mathcal{R}}(\text{rep}(i))$  to get the value of the representative for  $i$ .

We store in  $\mathcal{U}$  the current difference from each element to its representative,  $\mathcal{U}[i] = Y[i] - \mathcal{R}[\text{rep}(i)]$  (i.e. updates between rebuilds are applied to  $\mathcal{U}$ ). The idea is to pack  $\mathcal{U}$  into a single word of  $B$  elements. Observe that  $\text{update}(i, \Delta)$  adds value  $\Delta$  to all elements in  $Y$  with index at least  $i$ . We can support this operation in constant time by adding to  $\mathcal{U}$  a word that encodes  $\Delta$  for those elements. Since each difference between adjacent elements in a run is at most  $B \cdot 2^\delta$  and  $|Y| = O(B)$ , the maximum value in  $\mathcal{U}$  after a rebuild is  $O(B^2 \cdot 2^\delta)$ . As  $B$  updates of size  $2^\delta$  may be applied before a rebuild, the changed value at each element due to updates is  $O(B \cdot 2^\delta)$ . So each element in  $\mathcal{U}$  requires  $O(\log B + \delta)$  bits (including an overflow and sign bit per element). Thus,  $\mathcal{U}$  requires  $O(B(\log B + \delta))$  bits in total and can be packed in a single word for  $B = O(\min\{w/\log w, w/\delta\})$ .

Between rebuilds the stored representatives are potentially outdated because updates may have changed their values. However, observe that the values of two consecutive representatives differ by more than  $B \cdot 2^\delta$  at the time of a rebuild, so the gap between two representatives cannot be closed by  $B$  updates of  $\delta$

bits each (before the structure is rebuilt again). Hence, an answer to  $\text{search}(t)$  cannot drift much from the values stored by the representatives; it can only be in a constant number of runs, namely those with a representative value  $\text{succ}_{\mathcal{R}}(t)$  and its two neighboring runs. In a run with representative value  $v$ , we find the smallest  $j$  (inside the run) such that  $\mathcal{U}[j] + v - t > 0$ . The smallest  $j$  found in all three runs is the answer to the  $\text{search}(t)$  query. Thus, by rebuilding periodically, we only need to check a constant number of runs when answering a  $\text{search}(t)$  query.

On this structure, Pătraşcu and Demaine [109] show that the operations **sum**, **search** and **update** can be supported in constant time each as follows:

**sum( $i$ )**: return the sum of  $\mathcal{R}[\text{rep}(i)]$  and  $\mathcal{U}[i]$ . This takes constant time as  $\mathcal{U}[i]$  is a field in a word and representatives are stored using Lemma 4.9.

**search( $t$ )**: let  $r_0 = \text{rank}_{\mathcal{R}}(\text{succ}_{\mathcal{R}}(t))$ . We must find the smallest  $j$  such that  $\mathcal{U}[j] + R[r] - t > 0$  for  $r \in \{r_0 - 1, r_0, r_0 + 1\}$ , where  $j$  is in run  $r$ . We do this for each  $r$  using standard word operations in constant time by adding  $R[r] - t$  to all elements in  $\mathcal{U}$ , masking elements not in the run (outside indices  $\text{select}_{\mathcal{I}}(r)$  to  $\text{select}_{\mathcal{I}}(r+1) - 1$ , and counting the number of negative elements).

**update( $i, \Delta$ )**: we do this in constant time by copying  $\Delta$  to all fields  $j \geq i$  by a multiplication and adding the result to  $\mathcal{U}$ .

To count the number of negative elements or find the least significant bit in a word in constant time, we use the technique by Fredman and Willard [53].

Notice that rebuilding the data structure every  $B$  operations takes  $O(B)$  time, resulting in amortized constant time per operation. We de-amortize this to worst case constant time in the following way: After  $B/2$  operations we make a copy of  $\mathcal{U}$  that we call  $\mathcal{U}_{B/2}$  and build a new data structure over the next  $B/2$  operations based on the content of  $\mathcal{U}_{B/2}$ ,  $\mathcal{I}$  and  $\mathcal{R}$ . This gives us a new data structure consisting of  $\mathcal{U}'$ ,  $\mathcal{I}'$  and  $\mathcal{R}'$  but  $\mathcal{U}'$  needs to be updated to reflect the last  $B/2$  operations performed during the build. We do the update by subtracting the values stored in  $\mathcal{U}_{B/2}$  from the current values stored in  $\mathcal{U}$  and adding the differences to  $\mathcal{U}'$ . We then substitute  $\mathcal{U}$ ,  $\mathcal{I}$  and  $\mathcal{R}$  for  $\mathcal{U}'$ ,  $\mathcal{I}'$  and  $\mathcal{R}'$  and rebuild again during the next  $B/2$  operations. We spend  $O(B)$  time building the new data structure over  $B/2$  operations such that we use  $O(1)$  extra time for each operation and lastly we update  $\mathcal{U}'$  in  $O(1)$  time with word-level parallel addition and subtraction.



#### 4.3.1.2 Efficient Support for divide and merge

We now show how to maintain the structure described above while supporting operations `divide( $i, t$ )` and `merge( $i$ )`. An example supporting the following explanation is provided in Figure 4.1.

Observe that the operations are only local: Splitting  $Z[i]$  into two parts or merging  $Z[i]$  and  $Z[i+1]$  does not influence the precomputed values in  $Y$  (besides adding/removing values for the divided/merged elements). We must update  $\mathcal{I}$ ,  $\mathcal{R}$  and  $\mathcal{U}$  to reflect these local changes accordingly. Because a `divide` or `merge` operation may create new representatives between rebuilds with values that do not fit in  $\mathcal{U}$ , we change  $\mathcal{I}$ ,  $\mathcal{R}$  and  $\mathcal{U}$  to reflect these new representatives by rebuilding the data structure locally. This is done as follows.

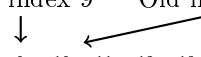
Consider the run representatives. Both `divide( $i, t$ )` and `merge( $i$ )` may require us to create a new run, combine two existing runs or remove a run. In any case, we can find a replacement representative for each run affected. As the operations are only local, the replacement is either a divided or merged element, or one of the neighbors of the replaced representative. Replacing representatives may cause both indices and values for the stored representatives to change. We use insertions and deletions on  $\mathcal{R}$  to update representative values.

Since the new operations change the indices of the elements, these changes must also be reflected in  $\mathcal{I}$ . For example, a `merge( $i$ )` operation decrements the indices of all elements with index larger than  $i$  compared to the indices stored at the time of the last rebuild. We should in principle adjust the  $O(B)$  changed indices stored in  $\mathcal{I}$ . The cost of adjusting the indices accordingly when using Lemma 4.9 to store  $\mathcal{I}$  is  $O(B)$ . Instead, to get our desired constant time bounds, we represent  $\mathcal{I}$  using a resizable data structure with the same number of elements as  $Y$  that supports this kind of update. We must support `select $_{\mathcal{I}}(i)$` , `rank $_{\mathcal{I}}(q)$` , and `pred $_{\mathcal{I}}(q)$`  as well as inserting and deleting elements in constant time. Because  $\mathcal{I}$  has few and small elements, we can support the operations in constant time by representing it using a bitstring  $\mathcal{B}$  and a structure  $\mathcal{C}$  which is the prefix sum over  $\mathcal{B}$  as follows.

Let  $\mathcal{B}$  be a bitstring of length  $|Y| \leq B$ , where  $\mathcal{B}[i] = 1$  iff there is a representative at index  $i$ . Then  $\mathcal{C}$  has  $|Y|$  elements, where  $\mathcal{C}[i]$  is the prefix sum of  $\mathcal{B}$  including element  $i$ . Since  $\mathcal{C}$  requires  $O(B \log B)$  bits in total we can pack it in a single word. We answer queries as follows: `rank $_{\mathcal{I}}(q)$`  equals  $\mathcal{C}[q-1]$ , we answer `select $_{\mathcal{I}}(i)$`  by subtracting  $i$  from all elements in  $\mathcal{C}$  and return one plus the number of elements smaller than 0 (as done in  $\mathcal{U}$  when answering `search`), and we find `pred $_{\mathcal{I}}(q)$`  as the index of the least significant bit in  $\mathcal{B}$  after having masked all indices larger than  $q$ . Updates are performed as follows. Using mask, shift


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$Z$	5	1	4	7	1	1	6	5	1	1	2	2	1	3	5	10	5	10	2
$Y$	5	6	10	17	18	19	25	30	31	32	34	36	37	40	45	55	60	70	72
$\mathcal{R}$	{5, 17, 25, 30, 45, 55, 60, 70}																		
$\mathcal{U}$	0	1	5	0	1	2	0	0	1	2	4	6	7	10	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	1	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	4	4	4	4	4	4	4	5	6	7	8	8

a) The initial data structure constructed from  $Z$ .

New index 9      Old index 9  


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$Z$	5	1	4	7	1	1	6	3	2	1	1	2	2	1	3	5	10	5	10	2
$Y$	5	6	10	17	18	19	25	28	30	31	32	34	36	37	40	45	55	60	70	72
$\mathcal{R}$	{5, 17, 25, 45, 55, 60, 70}																			
$\mathcal{U}$	0	1	5	0	1	2	0	3	5	6	7	9	11	12	15	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	3	3	3	3	3	3	3	3	4	5	6	7	7

b) The result of  $\text{divide}(8, 3)$  on the structure of a). Representative value 30 was removed from  $\mathcal{R}$ . We shifted and updated  $\mathcal{U}$ ,  $\mathcal{B}$  and  $\mathcal{C}$  to remove the old representative and accommodate for a new element with value 2.

Index containing the sum of the merged indices.  


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$Z$	5	1	4	7	1	1	6	3	2	1	1	4	1	3	5	10	5	10	2
$Y$	5	6	10	17	18	19	25	28	30	31	32	36	37	40	45	55	60	70	72
$\mathcal{R}$	{5, 17, 25, 45, 55, 60, 70}																		
$\mathcal{U}$	0	1	5	0	1	2	0	3	5	6	7	11	12	15	0	0	0	0	2
$\mathcal{B}$	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	0
$\mathcal{C}$	1	1	1	2	2	2	3	3	3	3	3	3	3	3	4	5	6	7	7

c) The result of  $\text{merge}(12)$  on the structure of b).

**Figure 4.1:** Illustrating operations on the data structure with  $B \cdot 2^\delta = 4$ . a) shows the data structure immediately after a rebuild, b) shows the result of performing  $\text{divide}(8, 3)$  on the structure of a), and c) shows the result of performing  $\text{merge}(12)$  on the structure of b).

and concatenate operations, we can ensure that  $\mathcal{B}$  and  $\mathcal{C}$  have the same size as  $Y$  at all times (we extend and shrink them when performing `divide` and `merge` operations). Inserting or deleting a representative is to set a bit in  $\mathcal{B}$ , and to keep  $\mathcal{C}$  up to date, we employ the same word-level parallel update scheme as used for  $\mathcal{U}$ .

We finally need to adjust the relative offsets of all elements with a changed representative in  $\mathcal{U}$  (since they now belong to a representative with a different value). In particular, if the representative for  $\mathcal{U}[j]$  changed value from  $v$  to  $v'$ , we must subtract  $v' - v$  from  $\mathcal{U}[j]$ . This can be done for all affected elements belonging to a single representative simultaneously in  $\mathcal{U}$  by a single addition with an appropriate bitmask (`update` a range of  $\mathcal{U}$ ). Note that we know the range of elements to update from the representative indices. Finally, we may need to insert or delete an element in  $\mathcal{U}$ , which can be done easily by mask, shift and concatenate operations on the word  $\mathcal{U}$ .

We still need to make sure that each index of  $\mathcal{U}$  does not overflow due to `update` operations. We do this by checking the value of each index in a round-robin fashion such that each index is checked once every  $B$  operations. We add and remove representatives based on whether the difference between neighboring indices exceeds or is at most  $B \cdot 2^\delta$ . We do this exactly as when adding or removing representatives due to `divide` and `merge` operations. This leads to Theorem 4.10.

**THEOREM 4.10** *There is a linear space data structure for dynamic partial sums supporting each operation `search`, `sum`, `update`, `insert`, `delete`, `divide`, and `merge` on a sequence of length  $O(\min\{w/\log w, w/\delta\})$  in worst-case constant time.*

### 4.3.2 Dynamic Partial Sums for Large Sequences

Willard [135] (and implicitly Dietz [35]) showed that a leaf-oriented B-tree with out-degree  $B$  of height  $h$  can be maintained in  $O(h)$  worst-case time if: 1) searches, insertions and deletions take  $O(1)$  time per node when no splits or merges occur, and 2) merging or splitting a node of size  $B$  requires  $O(B)$  time.

We use this as follows, where  $Z$  is our integer sequence of length  $s$ . Create a leaf-oriented B-tree of degree  $B = \Theta(\min\{w/\log w, w/\delta\})$  storing  $Z$  in the leaves, with height  $h = O(\log_B n) = O(\log n / \log(w/\delta))$ . Each node  $v$  uses Theorem 4.10 to store the  $O(B)$  sums of leaves in each of the subtrees of its children. A `sum( $i$ )` operation corresponds to traversing the B-tree from root to leaf  $i$  while constructing  $Y[i]$ . An `update( $i, \Delta$ )` operation corresponds to

traversing the B-tree from root to leaf  $i$  while performing local **update** operations on the nodes encountered. A **search**( $t$ ) operation corresponds to traversing the B-tree from root to leaf where each node is navigated by performing a local **search** operation. The **merge**( $i$ ) and **divide**( $i, t$ ) corresponds to performing a local **merge** or **divide** in the leaf node containing index  $i$ . This concludes the proof of Theorem 4.2.

## 4.4 Substring Concatenation

In this section we prove Theorem 4.3. Recall that we must store a string  $R$  subject to substring concatenation queries: given the location of two substrings  $x$  and  $y$  of  $R$  return the location of an occurrence of  $xy$  in  $R$  or NO if no such occurrence exists.

Before we start the proof of Theorem 4.3, we first need to introduce a data structure. Goswami et al. [65] showed how to use the weak prefix search data structure from Belazzougui et al. [10] for a 1D range emptiness data structure. The following data structure follows from these results<sup>1</sup>:

**LEMMA 4.11 ([65] AND [10])** *Given an ordered set  $A \subseteq [1, r]$  we can construct a data structure that uses  $O(|A| \log^\epsilon r)$  bits of space, for any constant  $\epsilon > 0$ , and supports the following query in constant time. Given a range  $[a, b] \subseteq [1, r]$  report the first index of an element within the range  $[a, b]$ , or if  $A \cap [a, b] = \emptyset$  an arbitrary index (i.e. it may report false-positives).*

To prove Theorem 4.3(i) we need the following definitions. For a substring  $x$  of  $R$ , let  $S(x)$  denote the suffixes of  $R$  that have  $x$  as a prefix, and let  $S'(x) = \{i + |x| \mid i \in S(x)\}$ , i.e.,  $S'(x)$  are the suffixes of  $R$  that are immediately preceded by  $x$ . Hence for two substrings  $x$  and  $y$ , the suffixes that have  $xy$  as a prefix are exactly  $S'(x) \cap S(y)$  shifted left by  $|x|$ . We can reduce this intersection problem to a query on the data structure from Lemma 4.11 as follows.

Let  $\text{rank}(i)$  be the position of suffix  $R[i..r]$  in the lexicographic ordering of all suffixes of  $R$ , and let  $\text{rank}(A)$  be the ordered set  $\{\text{rank}(i) \mid i \in A\}$  for  $A \subseteq \{1, 2, \dots, r\}$ , and let  $\text{pos}(i)$  be the inverse of  $\text{rank}(i)$ , i.e., it returns the position in  $R$  of the  $i^{\text{th}}$  suffix in the lexicographical ordering. Then  $xy$  is a substring of  $R$  if and only if  $\text{rank}(S'(x)) \cap \text{rank}(S(y)) \neq \emptyset$ . In particular  $xy$  then

---

<sup>1</sup>The technique is described in the section "Range Emptiness Data Structure" in [65]. Note since we allow false-positives, we do not need the sorted list of points and thus avoid the  $n \lg U$  bits term in the space usage.

occurs at the positions  $\text{pos}(\text{rank}(S'(x)) \cap \text{rank}(S(y)))$  shifted  $|x|$  to the left in  $R$ . Note that  $\text{rank}(S(y))$  is a range  $[a, b] \subseteq [1, r]$ , and we can determine this range in constant time for any substring  $y$  using a constant-time weighted ancestor query on the suffix tree of  $R$  [62]. Consequently, if we have built the data structure from Lemma 4.11 for  $\text{rank}(S'(x))$  we can query for the range  $[a, b]$  and thereby determine the intersection.

Unfortunately, this data structure only gives the index of an element within the range, and not the element itself<sup>2</sup>. Thus the position of  $xy$  in  $R$  cannot be computed as indicated before. Instead we use the followings properties. The  $i^{\text{th}}$  suffix in  $\text{rank}(S(x))$  is the  $i^{\text{th}}$  suffix in  $\text{rank}(S'(x))$  with  $x$  prepended. The  $i^{\text{th}}$  suffix in  $\text{rank}(S(x))$  has rank  $\text{rank}(x) + i$  where  $\text{rank}(x)$  is the number of suffixes of  $R$  that are lexicographically smaller than  $x$ . Thus if a query on  $\text{rank}(S'(x))$  for  $[a, b]$  returns the  $i^{\text{th}}$  element then  $xy$  occurs at position  $\text{pos}(\text{rank}(x) + i)$  in  $R$ . Also, this data structure might answer with false-positives, so we have to verify  $xy$  actually occurs at the found position. The found position certainly starts with  $x$ , but is not guaranteed to be followed by  $y$ , but this can simply be verified using a constant time longest common prefix query to compare the found position shifted  $|x|$  to the right with the known occurrence of  $y$ .

To arrive at the space bound of  $O(r \log^\epsilon r)$  (words), we employ a heavy path decomposition [70] on the suffix tree of  $R$ , and only build the data structure from Lemma 4.11 for substrings of  $R$  that correspond to the top of a heavy path. In this way, each suffix will appear in at most  $\log r$  such data structures, leading to the claimed  $O(r \log^\epsilon r)$  space bound (in words). In addition, we build an  $O(r)$ -space nearest common ancestor data structure [70] for the suffix tree of  $R$ . Constant-time nearest common ancestor queries will allow us to also answer longest common prefix queries on  $R$  in constant time.

To answer a substring concatenation query with substrings  $x$  and  $y$ , we first determine how far  $y$  follows the heavy path in the suffix tree from the location where  $x$  stops. This can be done in  $O(1)$  time by a constant-time longest common prefix query between two suffixes of  $R$ . We then proceed to the top of the next heavy path, where we query the data structure with the range  $\text{rank}(S(y'))$  where  $y'$  is the remaining unmatched suffix of  $y$ . Finally, we compute the actual position of  $xy$  from the answer and verify it is a real occurrence as previously described. This completes the query, and the proof of (i).

The second solution (ii) is an implication of a result by Bille et al. [17]. Given the suffix tree  $ST_R$  of  $R$ , an *unrooted longest common prefix query* [28] takes a suffix  $y$  and a location  $\ell$  in  $ST_R$  (either a node or a position on an edge) and returns the location in  $ST_S$  that is reached after matching  $y$  starting from

<sup>2</sup>Storing the element itself would take up too much space.

location  $\ell$ . A substring concatenation query is straightforward to implement using two unrooted longest common prefix queries, the first one starting at the root, and the second starting from the location returned by the first query. It follows from Bille et al. [17] that we can build a linear space data structure that supports unrooted longest common prefix queries in time  $O(\log \log r)$  thus completing the proof of (ii).

## 4.5 Extensions

In this section we show how to solve two other variants of the dynamic relative compression problem. We first prove Theorem 4.5, showing how to improve the query time if only supporting operations **access** and **replace**. We then show Theorem 4.7, generalising the problem to support multiple strings. These data structures use the same substring concatenation data structure of Theorem 4.3 as before but replaces the dynamic partial sums data structure.

### 4.5.1 Dynamic Relative Compression with Access and Replace

In this setting we constrain the operations on  $S$  to **access**( $i$ ) and **replace**( $i, \alpha$ ). Then, instead of maintaining a dynamic partial sums data structure over the lengths of the substrings in  $C$ , we only need a dynamic predecessor data structure over the prefix sums. The operations are implemented as before, except that for **access**( $i$ ) we obtain block  $b_j$  by computing the predecessor of  $i$  in the predecessor data structure, which also immediately gives us access to the local index in  $b_j$ . For **replace**( $i, \alpha$ ), a constant number of updates to the predecessor data structure is needed to reflect the changes. We use substring concatenation queries to restore minimality as described in Section 4.2. The prefix sums of the subsequent blocks in  $C$  are preserved since  $|b_j| = |b_j^1| + |b_j^2| + |b_j^3|$ .

With a linear space implementation of the van Emde Boas data structure [99, 130, 131] we can support the predecessor queries and updates in  $O(\log \log N)$  expected time. For substring concatenation we apply Theorem 4.3(ii) using  $O(r)$  space and  $O(\log \log r)$  time. Since the length of source string does not change, we can always assume that  $r < N$ , and the total time becomes  $O(\log \log N + \log \log r) = O(\log \log N)$ . Note that it is actually possible to get  $O(\log \log N)$  worst case time for **access** operations. In summary, this proves Theorem 4.5.

### 4.5.2 Dynamic Relative Compression with Split and Concatenate

Consider the variant of the dynamic relative compression problem where we want to maintain a relative compression of a set of strings  $S_1, \dots, S_k$ . Each string  $S_i$  has a cover  $C_i$  and all strings are compressed relative to the same string  $R$ . In this setting  $n = \sum_{i=1}^k |C_i|$ . In addition to the operations **access**, **replace**, **insert**, and **delete**, we also want to support **split** and **concatenate** of strings. Note that the semantics of the operations change to indicate the string(s) to perform a given operation on.

We build a leaf-oriented height-balanced binary tree  $T_i$  (e.g. an 2-3 tree) over the blocks  $C_i[1], \dots, C_i[|C_i|]$  for each string  $S_i$ . In each internal node  $v$ , we store the sum of the block sizes represented by its leaves. Since the total number of blocks is  $n$ , the trees use  $O(n)$  space. All operations rely on the standard procedures for searching, inserting, deleting, splitting and joining height-balanced binary trees. All of these run in  $O(\log n)$  time for a tree of size  $n$ .

The answer to an **access**( $i, j$ ) query is found by doing a top-down search in  $T_i$  using the sums of block sizes to navigate. Since the tree is balanced and the size of the cover is at most  $n$ , this takes  $O(\log n)$  time. The operations **replace**( $i, j, \alpha$ ), **insert**( $i, j, \alpha$ ), and **delete**( $i, j$ ) all initially require that we use **access**( $i, j$ ) to locate the block containing the  $j$ -th character of  $S_i$ . To reflect possible changes to the blocks of the cover, we need to modify the corresponding tree to contain more leaves and restore the balancing property. Since the number of nodes added to the tree is constant these operations each take  $O(\log n)$  time. The **concat**( $i, j$ ) operation requires that we join two trees in the standard way and restore the balancing property of the resulting tree. For the **split**( $i, j$ ) operation we first split the block that contains position  $j$  such that the  $j$ -th character is the trailing character of a block. We then split the tree into two trees separated by the new block. This takes  $O(\log n)$  time for a height-balanced tree.

To finalize the implementation of the operations, we must restore the minimality property of the affected covers as described in Section 4.2. At most a constant number of blocks are non-minimal as a result of any of the operations. If two blocks can be combined to one, we delete the leaf that represents the rightmost block, update the leftmost block to reflect the change, and restore the property that the tree is balanced. If the tree subsequently contains an internal node with only one child, we delete it and restore the balancing. Again, this takes  $O(\log n)$  time for balanced trees, which concludes the proof of Theorem 4.7.

## 4.6 Conclusion

We have shown how to compress a text relatively to a reference string while supporting access to the text and a range of dynamic operations under some strong guarantees for the space usage and the query times. There are, however, room for improvement.

Our solution to DRC is built on data structures for the partial sums problem and the substring concatenation problem. Our partial sums-solution is optimal, but in order to get the desired constant query time for substring concatenation, our data structure uses  $O(r \log^\epsilon r)$  space. As opposed to this, our linear space solution leads to  $O(\log \log r)$  query time. We leave as an open problem if it is possible to get  $O(1)$  time substring concatenation queries using  $O(r)$  space, which will also carry over to a stronger result for the DRC problem.

Moreover, the size of the cover that is maintained by our DRC data structure is also an interesting parameter. Currently we maintain a 2-approximation of the optimal cover. It would be interesting to know if a better approximation ratio can be maintained under the same (or better) time and space bounds that we give. A first step could be to investigate whether some of the ideas by Fischer et al. [45] could be applied to our problem.

**Acknowledgments** We thank Pawel Gawrychowski for helpful discussions.





## CHAPTER 5

# Succinct Partial Sums and Fenwick Trees

---

Philip Bille <sup>†</sup>      Anders Roy Christiansen <sup>†</sup>      Nicola Prezza <sup>†</sup>  
Frederik Rye Skjoldjensen <sup>†</sup>

<sup>†</sup> The Technical University of Denmark

### Abstract

We consider the well-studied *partial sums* problem in succinct space where one is to maintain an array of  $n$   $k$ -bit integers subject to updates such that partial sums queries can be efficiently answered. We present two succinct versions of the Fenwick Tree – which is known for its simplicity and practicality. Our results hold in the encoding model where one is allowed to reuse the space from the input data. Our main result is the first that only requires  $nk + o(n)$  bits of space while still supporting sum/update in  $\mathcal{O}(\log_b n)$  /  $\mathcal{O}(b \log_b n)$  time where  $2 \leq b \leq \log^{\mathcal{O}(1)} n$ . The second result shows how optimal time for sum/update can be achieved while only slightly increasing the space usage to  $nk + o(nk)$  bits. Beyond Fenwick Trees, the results are primarily based on bit-packing and sampling – making them very practical – and they also allow for simple optimal parallelization.

## 5.1 Introduction

Let  $A$  be an array of  $k$ -bits integers, with  $|A| = n$ . The *partial sums* problem is to build a data structure maintaining  $A$  under the following operations.

- **sum**( $i$ ): return the value  $\sum_{t=1}^i A[t]$ .
- **search**( $j$ ): return the smallest  $i$  such that **sum**( $i$ )  $\geq j$ .
- **update**( $i, \Delta$ ): set  $A[i] \leftarrow A[i] + \Delta$ , for some  $\Delta$  such that  $0 \leq A[i] + \Delta < 2^k$ .
- **access**( $i$ ): return  $A[i]$ .

Note that **access**( $i$ ) can be implemented as **sum**( $i$ ) – **sum**( $i - 1$ ) and we therefore often do not mention it explicitly.

The partial sums problem is one of the most well-studied data structure problems [34, 40, 51, 52, 72, 110, 114, 137]. In this paper, we consider solutions to the partial sums problem that are *succinct*, that is, we are interested in data structures that use space close to the information-theoretic lower bound of  $nk$  bits. We distinguish between *encoding data structures* and *indexing data structures*. Indexing data structures are required to store the input array  $A$  verbatim along with additional information to support the queries, whereas encoding data structures have to support operations without consulting the input array.

In the indexing model Raman et al. [115] gave a data structure that supports **sum**, **update**, and **search** in  $\mathcal{O}(\log n / \log \log n)$  time while using  $nk + o(nk)$  bits of space. This was improved and generalized by Hon et al. [72]. Both of these papers have the constraint  $\Delta \leq \log^{\mathcal{O}(1)} n$ . The above time complexity is nearly optimal by a lower bound of Patrascu and Demaine [110] who showed that **sum**, **search**, and **update** operations take  $\Theta(\log_{w/\delta} n)$  time per operation, where  $w \geq \log n$  is the word size and  $\delta$  is the number of bits needed to represent  $\Delta$ . In particular, whenever  $\Delta = \log^{\mathcal{O}(1)} n$  this bound matches the  $\mathcal{O}(\log n / \log \log n)$  bound of Raman et al. [115].

Fenwick [40] presented a simple, elegant, and very practical encoding data structure. The idea is to replace entries in the input array  $A$  with partial sums that cover  $A$  in an implicit complete binary tree structure. The operations are then implemented by accessing at most  $\log n$  entries in the array. The Fenwick tree uses  $nk + n \log n$  bits and supports all operations in  $\mathcal{O}(\log n)$  time. In this paper we show two succinct  $b$ -ary versions of the Fenwick tree. In the first version we reduce the size of the Fenwick tree while improving the **sum** and **update** time.

In the second version we obtain optimal times for **sum** and **update** without using more space than the previous best succinct solutions [72,114]. All results in this paper are in the RAM model.

**Our results** We show two encoding data structures that gives the following results.

**THEOREM 5.1** *We can replace  $A$  with a succinct Fenwick tree of  $nk + o(n)$  bits supporting **sum**, **update**, and **search** queries in  $\mathcal{O}(\log_b n)$ ,  $\mathcal{O}(b \log_b n)$ , and  $\mathcal{O}(\log n)$  time, respectively, for any  $2 \leq b \leq \log^{\mathcal{O}(1)} n$ .*

**THEOREM 5.2** *We can replace  $A$  with a succinct Fenwick tree of  $nk + o(nk)$  bits supporting **sum** and **update** queries in optimal  $\mathcal{O}(\log_{w/\delta} n)$  time and **search** queries in  $\mathcal{O}(\log n)$  time.*

## 5.2 Data structure

For simplicity, assume that  $n$  is a power of 2. The Fenwick tree is an implicit data structure replacing a word-array  $A[1, \dots, n]$  as follows:

**DEFINITION 5.3** Fenwick tree of  $A$  [40]. If  $n = 1$ , then leave  $A$  unchanged. Otherwise, divide  $A$  in consecutive non-overlapping blocks of two elements each and replace the second element  $A[2i]$  of each block with  $A[2i - 1] + A[2i]$ , for  $i = 1, \dots, n/2$ . Then, recurse on the sub-array  $A[2, 4, \dots, 2i, \dots, n]$ .

To answer  $\text{sum}(i)$ , the idea is to write  $i$  in binary as  $i = 2^{j_1} + 2^{j_2} + \dots + 2^{j_k}$  for some  $j_1 > j_2 > \dots > j_k$ . Then there are  $k \leq \log n$  entries in the Fenwick tree, that can be easily computed from  $i$ , whose values added together yield  $\text{sum}(i)$ . In Section 5.2.1 we describe in detail how to perform such accesses. As per the above definition, the Fenwick tree is an array with  $n$  indices. If represented compactly, this array can be stored in  $nk + n \log n$  bits. In this section we present a generalization of Fenwick trees taking only succinct space.

### 5.2.1 Layered b-ary structure

We first observe that it is easy to generalize Fenwick trees to be  $b$ -ary, for  $b \geq 2$ : we divide  $A$  in blocks of  $b$  integers each, replace the first  $b - 1$  elements in each

block with their partial sum, and fill the remaining  $n/b$  entries of  $A$  by recursing on the array  $A'$  of size  $n/b$  that stores the sums of each block. This generalization gives an array of  $n$  indices supporting **sum**, **update**, and **search** queries on the original array in  $\mathcal{O}(\log_b n)$ ,  $\mathcal{O}(b \log_b n)$ , and  $\mathcal{O}(\log n)$  time, respectively. We now show how to reduce the space of this array.

Let  $\ell = \log_b n$ . We represent our  $b$ -ary Fenwick tree  $T_b(A)$  using  $\ell + 1$  arrays (layers)  $T_b^1(A), \dots, T_b^{\ell+1}(A)$ . For simplicity, we assume that  $n = b^e$  for some  $e \geq 0$  (the general case is then straightforward to derive). To improve readability, we define our layered structure for the special case  $b = 2$ , and then sketch how to extend it to the general case  $b \geq 2$ . Our layered structure is defined as follows. If  $n = 1$ , then  $T_2^1(A) = A$ . Otherwise:

- $T_2^{\ell+1}(A)[i] = A[(i-1) \cdot 2 + 1]$ , for all  $i = 1, \dots, n/2$ . Note that  $T_2^{\ell+1}(A)$  contains  $n/2$  elements.
- Divide  $A$  in blocks of 2 elements each, and build an array  $A'[j]$  containing the  $n/2$  sums of each block, i.e.  $A'[j] = A[(j-1) \cdot 2 + 1] + A[(j-1) \cdot 2 + 2]$ , for  $j = 1, \dots, n/2$ . Then, the next layers are recursively defined as  $T_2^\ell(A) \leftarrow T_2^\ell(A'), \dots, T_2^1(A) \leftarrow T_2^1(A')$ .

For general  $b \geq 2$ ,  $T_b^{\ell+1}(A)$  is an array of  $\frac{n(b-1)}{b}$  elements that stores the  $b-1$  partial sums of each block of  $b$  consecutive elements in  $A$ , while  $A'$  is an array of size  $n/b$  containing the complete sums of each block. In Figure 5.1 we report an example of our layered structure with  $b = 3$ . It follows that elements of  $T_b^i(A)$ , for  $i > 1$ , take at most  $k + (\ell - i + 2) \log b$  bits each. Note that arrays  $T_b^1(A), \dots, T_b^{\ell+1}(A)$  can easily be packed contiguously in a word array while preserving constant-time access to each of them. This saves us  $\mathcal{O}(\ell)$  words that would otherwise be needed to store pointers to the arrays. Let  $S_b(n, k)$  be the space (in bits) taken by our layered structure. This function satisfies the recurrence

$$\begin{aligned} S_b(1, k) &= k \\ S_b(n, k) &= \frac{n(b-1)}{b} \cdot (k + \log b) + S_b(n/b, k + \log b) \end{aligned}$$

Which unfolds to  $S_b(n, k) = \sum_{i=1}^{\log_b n+1} \frac{n(b-1)}{b^i} \cdot (k + i \log b)$ . Using the identities  $\sum_{i=1}^{\infty} 1/b^i = 1/(b-1)$  and  $\sum_{i=1}^{\infty} i/b^i = b/(b-1)^2$ , one can easily derive that  $S_b(n, k) \leq nk + 2n \log b$ .

We now show how to obtain the time bounds stated in Theorem 5.1. In the next section, we reduce the space of the structure without affecting query times.

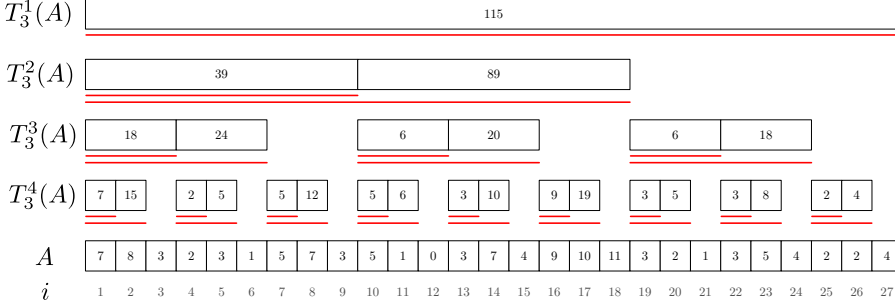
**Answering sum** Let the notation  $(x_1x_2\dots x_t)_b$ , with  $0 \leq x_i < b$  for  $i = 1, \dots, t$ , represent the number  $\sum_{i=1}^t b^{t-i}x_i$  in base  $b$ .  $\text{sum}(i)$  queries on our structure are a generalization (in base  $b$ ) of  $\text{sum}(i)$  queries on standard Fenwick trees. Consider the base- $b$  representation  $x_1x_2\dots x_{\ell+1}$  of  $i$ , i.e.  $i = (x_1x_2\dots x_{\ell+1})_b$  (note that we have at most  $\ell + 1$  digits since we enumerate indexes starting from 1). Consider now all the positions  $1 \leq i_1 < i_2 < \dots < i_t \leq \ell + 1$  such that  $x_j \neq 0$ , for  $j = i_1, \dots, i_t$ . The idea is that each of these positions  $j = i_1, \dots, i_t$  can be used to compute an offset  $o_j$  in  $T_b^j(A)$ . Then,  $\text{sum}(i) = \sum_{j=i_1, \dots, i_t} T_b^j(A)[o_j]$ . The offset  $o_j$  relative to the  $j$ -th most significant (nonzero) digit of  $i$  is defined as follows. If  $j = 1$ , then  $o_j = x_1$ . Otherwise,  $o_j = (b-1) \cdot (x_1\dots x_{j-1})_b + x_j$ . Note that we scale by a factor of  $b-1$  (and not  $b$ ) as the first term in this formula as each level  $T_b^j(A)$  stores only  $b-1$  out of  $b$  partial sums (the remaining sums are passed to level  $j-1$ ). Note moreover that each  $o_j$  can be easily computed in constant time and *independently from the other offsets* with the aid of modular arithmetic. It follows that **sum** queries are answered in  $\mathcal{O}(\log_b n)$  time. See Figure 5.1 for a concrete example of **sum**.

**Answering update** The idea for performing  $\text{update}(i, \Delta)$  is analogous to that of  $\text{sum}(i)$ . We access all levels that contain a partial sum covering position  $i$  and update at most  $b-1$  sums per level. Using the same notation as above, for each  $j = i_1, \dots, i_t$  such that  $x_j \neq 0$ , we update  $T_b^j(A)[o_j + l] \leftarrow T_b^j(A)[o_j + l] + \Delta$  for  $l = 0, \dots, b - x_j - 1$ . This procedure takes  $\mathcal{O}(b \log_b n)$  time.

**Answering search** To answer  $\text{search}(j)$  we start from  $T_b^1(A)$  and simply perform a top-down traversal of the implicit B-tree of degree  $b$  defined by the layered structure. At each level, we perform  $\mathcal{O}(\log b)$  steps of binary search to find the new offset in the next level. There are  $\log_b n$  levels, so **search** takes overall  $\mathcal{O}(\log n)$  time.

## 5.2.2 Sampling

Let  $0 < d \leq n$  be a sample rate, where for simplicity we assume that  $d$  divides  $n$ . Given our input array  $A$ , we derive an array  $A'$  of  $n/d$  elements containing the sums of groups of  $d$  adjacent elements in  $A$ , i.e.  $A'[i] = \sum_{j=1}^d A[(i-1) \cdot d + j]$ ,  $i = 1, \dots, n/d$ . We then compact  $A$  by removing  $A[j \cdot d]$  for  $j = 1, \dots, n/d$ , and by packing the remaining integers in at most  $nk(1 - 1/d)$  bits. We build our layered  $b$ -ary Fenwick tree  $T_b(A')$  over  $A'$ . It is clear that queries on  $A$  can be solved with a query on  $T_b(A')$  followed by at most  $d$  accesses on (the compacted)  $A$ . The space of the resulting data structure is  $nk(1 - 1/d) + S_b(n/d, k + \log d) \leq$



**Figure 5.1:** Example of our layered structure with  $n = 27$  and  $b = 3$ . Horizontal red lines show the portion of  $A$  covered by each element in  $T_3^j(A)$ , for  $j = 1, \dots, \log_b n + 1$ . To access the  $i$ -th partial sum, we proceed as follows. Let, for example,  $i = 19 = (0201)_3$ . The only nonzero digits in  $i$  are the 2-nd and 4-th most significant. This gives us  $o_2 = 2 \cdot (0)_3 + 2 = 2$  and  $o_4 = 2 \cdot (020)_3 + 1 = 13$ . Then,  $\text{sum}(19) = T_3^2(A)[2] + T_3^4(A)[13] = 89 + 3 = 92$ .

$nk + \frac{n \log d}{d} + \frac{2n \log b}{d}$  bits. In order to retain the same query times of our basic layered structure, we choose  $d = (1/\epsilon) \log_b n$  for any constant  $\epsilon > 0$  and obtain a space occupancy of  $nk + \epsilon \left( \frac{n \log \log_b n}{\log_b n} + \frac{2n \log b}{\log_b n} \right)$  bits. For  $b \leq \log^{\mathcal{O}(1)} n$ , this space is  $nk + o(n)$  bits. Note that—as opposed to existing succinct solutions—the low-order term does not depend on  $k$ .

### 5.3 Optimal-time sum and update

In this section we show how to obtain optimal running times for **sum** and **update** queries in the RAM model. We can directly apply the word-packing techniques described in [110] to speed-up queries; here we only sketch this strategy, see [110] for full details. Let us describe the idea on the structure of Section 5.2.1, and then plug in sampling to reduce space usage. We divide arrays  $T_b^j(A)$  in blocks of  $b - 1$  entries, and store one word ( $w$  bits) for each such block. We can pack  $b - 1$  integers of at most  $w/(b - 1)$  bits each (for an opportune  $b$ , read below) in the word associated with each block. Since blocks of  $b - 1$  integers fit in a single word, we can easily answer **sum** and **update** queries on them in constant time. **sum** queries on our overall structure can be answered as described in Section 5.2.1, except that now we also need to access one of the packed integers at each level  $j$  to correct the value read from  $T_b^j(A)$ . To answer **update** queries, the idea

is to perform **update** operations on the packed blocks of integers in constant time exploiting bit-parallelism instead of updating at most  $b - 1$  values of  $T_b^j(A)$ . At each **update** operation, we transfer one of these integers on  $T_b^j(A)$  (in a cyclic fashion) to avoid overflowing and to achieve worst-case performance. Note that each packed integer is increased by at most  $\Delta$  for at most  $b - 1$  times before being transferred to  $T_b^j(A)$ , so we get the constraint  $(b - 1) \log((b - 1)\Delta) \leq w$ . We choose  $(b - 1) = \frac{w}{\log w + \delta}$ . Then, it is easy to show that the above constraint is satisfied. The number of levels becomes  $\log_b n = \mathcal{O}(\log_{w/\delta} n)$ . Since we spend constant time per level, this is also the worst-case time needed to answer **sum** and **update** queries on our structure. To analyze space usage we use the corrected formula

$$S_b(1, k) = k$$

$$S_b(n, k) = \frac{n(b-1)}{b} \cdot (k + \log b) + \frac{nw}{b} + S_b(n/b, k + \log b)$$

yielding  $S_b(1, k) \leq nk + 2n \log b + \frac{nw}{b-1}$ . Replacing  $b - 1 = \frac{w}{\log w + \delta}$  we achieve  $nk + \mathcal{O}(n\delta + n \log w)$  bits of space.

We now apply the sampling technique of Section 5.2.2 with a slight variation. In order to get the claimed space/time bounds, we need to further apply bit-parallelism techniques on the packed integers stored in  $A$ : using techniques from [69], we can answer **sum**, **search**, and **update** queries in  $\mathcal{O}(1)$  time on blocks of  $w/k$  integers. It follows that we can now use sample rate  $d = \frac{w \log n}{k \log(w/\delta)}$  without affecting query times. After sampling  $A$  and building the Fenwick tree above described over the sums of size- $d$  blocks of  $A$ , the overall space is  $nk(1 - 1/d) + S_b(n/d, k + \log d) = nk + \frac{n \log d}{d} + \mathcal{O}(\frac{n\delta}{d} + \frac{n \log w}{d})$ . Note that  $d \leq \frac{w^2}{k \log(w/\delta)} \leq w^2$ , so  $\log d \in \mathcal{O}(\log w)$  and space simplifies to  $nk + \mathcal{O}(\frac{n\delta}{d} + \frac{n \log w}{d})$ . The term  $\frac{n\delta}{d}$  equals  $\frac{n\delta k \log(w/\delta)}{w \log n}$ . Since  $\delta \leq w$ , then  $\delta \log(w/\delta) \leq w$ , and this term therefore simplifies to  $\frac{nk}{\log n} \in o(nk)$ . Finally, the term  $\frac{n \log w}{d}$  equals  $\frac{n \log w \cdot k \log(w/\delta)}{w \log n} \leq \frac{nk}{(w \log n)/(\log w)^2} \in o(nk)$ . The bounds of Theorem 5.2 follow.

**Parallelism** Note that **sum** and **update** queries on our succinct Fenwick trees can be naturally parallelized as all accesses/updates on the levels can be performed independently from each other. For **sum**, we need  $\mathcal{O}(\log \log_b n)$  further time to perform a parallel sum of the  $\log_b n$  partial results. It is not hard to show that—on architectures with  $\log_b n$  processors—this reduces **sum/update** times to  $\mathcal{O}(\log \log_b n)/\mathcal{O}(b)$  and  $\mathcal{O}(\log \log_{w/\delta} n)/\mathcal{O}(1)$  in Theorems 5.1 and 5.2, respectively.





## CHAPTER 6

# Fast Dynamic Arrays

---

Philip Bille<sup>†</sup>

Anders Roy Christiansen<sup>†</sup>

Mikko Berggren Ettienne<sup>†</sup>

Inge Li Gørtz<sup>†</sup>

<sup>†</sup> The Technical University of Denmark

### Abstract

We present a highly optimized implementation of tiered vectors, a data structure for maintaining a sequence of  $n$  elements supporting access in time  $O(1)$  and insertion and deletion in time  $O(n^\epsilon)$  for  $\epsilon > 0$  while using  $o(n)$  extra space. We consider several different implementation optimizations in C++ and compare their performance to that of vector and multiset from the standard library on sequences with up to  $10^8$  elements. Our fastest implementation uses much less space than multiset while providing speedups of  $40\times$  for access operations compared to multiset and speedups of  $10.000\times$  compared to vector for insertion and deletion operations while being competitive with both data structures for all other operations.

## 6.1 Introduction

We present a highly optimized implementation of a data structure solving the *dynamic array problem*, that is, maintain a sequence of elements subject to the following operations:

**access( $i$ ):** return the  $i^{th}$  element in the sequence.

**access( $i, m$ ):** return the  $i^{th}$  through  $(i + m - 1)^{th}$  elements in the sequence.

**insert( $i, x$ ):** insert element  $x$  immediately after the  $i^{th}$  element.

**delete( $i$ ):** remove the  $i^{th}$  element from the sequence.

**update( $i, x$ ):** exchange the  $i^{th}$  element with  $x$ .

This is a fundamental and well studied data structure problem [23, 35, 48, 49, 64, 85, 86, 115] solved by textbook data structures like arrays and binary trees. Many dynamic trees provide all the operations in  $O(\lg n)$  time including 2-3-4 trees, AVL trees, splay trees, etc. and Dietz [35] gives a data structure that matches the lower bound of  $\Omega(\lg n / \lg \lg n)$  showed by Fredman and Saks [49]. The lower bound only holds when identical complexities are required for all operations. In this paper we focus on the variation where **access** must run in  $O(1)$  time. Goodrich and Kloss present what they call *tiered vectors* [64] with a time complexity of  $O(1)$  for **access** and **update** and  $O(n^{1/l})$  for **insert** and **delete** for any constant integer  $l \geq 2$ , using ideas similar to Frederickson's in [48]. The data structure uses only  $o(n)$  extra space beyond that required to store the actual elements. At the core, the data structure is a tree with out degree  $n^{1/l}$  and *constant* height  $l - 1$ .

Goodrich and Kloss compare the performance of an implementation with  $l = 2$  to that of *vector* from the standard library of Java and show that the structure is competitive for access operations while being significantly faster for insertions and deletions. Tiered vectors provide a performance trade-off between standard arrays and balanced binary trees for the dynamic array problem.

**Our Contribution** In this paper, we present what we believe is the first implementation of tiered vectors that supports more than 2 tiers. Our C++ implementation supports **access** and **update** in times that are competitive with the vector data structure from C++'s standard library while **insert** and **delete** run more than  $10,000\times$  faster. It performs **access** and **update** more than  $40\times$  faster than the multiset data structure from the standard library while **insert** and **delete** is only a few percent slower. Furthermore multiset uses more than  $10\times$  more space than our implementation. All of this when working on large sequences of  $10^8$  32-bit integers.

To obtain these results, we significantly decrease the number of memory probes per operation compared to the original tiered vector. Our best variant requires only half as many memory probes as the original tiered vector for **access** and

update operations which is critical for the practical performance. Our implementation is cache efficient which makes all operations run fast in practice even on tiered vectors with several tiers.

We experimentally compare the different variants of tiered vectors. Besides the comparison to the two commonly used C++ data structures, vector and multiset, we compare the different variants of tiered vectors to find the best one. We show that the number of tiers have a significant impact on the performance which underlines the importance of tiered vectors supporting more than 2 tiers.

Our implementations are parameterized and thus support any number of tiers  $\geq 2$ . We use techniques like *template recursion* to keep the code rather simple while enabling the compiler to generate highly optimized code.

The source code can be found at <https://github.com/mettienne/tiered-vector>.

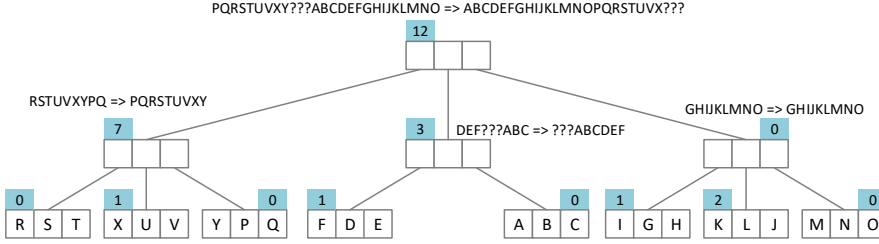
## 6.2 Preliminaries

The first and  $i^{th}$  element of a sequence  $A$  are denoted  $A[0]$  and  $A[i - 1]$  respectively and the  $i^{th}$  through  $j^{th}$  elements are denoted  $A[i - 1, j - 1]$ . Let  $A_1 \cdot A_2$  denote the concatenation of the sequences  $A_1$  and  $A_2$ .  $|A|$  denotes the number of elements in the sequence  $A$ . A circular shift of a sequence  $A$  by  $x$  is the sequence  $A[|A| - x, |A| - 1] \cdot A[0, |A| - x - 1]$ . Define the remainder of division of  $a$  by  $b$  as  $a \bmod b = a - qb$  where  $q$  is the largest integer such that  $q \cdot b \leq a$ . Define  $A[i, j] \bmod w$  to be the elements  $A[i \bmod w], A[(i+1) \bmod w], \dots, A[j \bmod w]$ , i.e.  $A[4, 7] \bmod 5 = A[4], A[0], A[1], A[2]$ . Let  $\lfloor x \rfloor$  denote the largest integer smaller than  $x$ .

## 6.3 Tiered Vectors

In this section we will describe how the tiered vector data structure from [64] works.

**Data Structure** An  $l$ -tiered vector can be seen as a tree  $T$  with root  $r$ , fixed height  $l - 1$  and out-degree  $w$  for any  $l \geq 2$ . A node  $v \in T$  represents a sequence of elements  $A(v)$  thus  $A(r)$  is the sequence represented by the tiered vector.



**Figure 6.1:** An illustration of a tiered vector with  $l = w = 3$ . The elements are letters, and the tiered vector represents the sequence ABCDEFGHIJKLMNOPQRSTUVWXYZ. The elements in the leaves are the elements that are actually stored. The number above each node is its offset. The strings above an internal node  $v$  with children  $c_1, c_2, c_3$  is respectively  $A(c_1) \cdot A(c_2) \cdot A(c_3)$  and  $A(v)$ , i.e. the elements  $v$  represents before and after the circular shift. ? specifies an empty element.

The capacity  $\text{cap}(v)$  of a node  $v$  is  $w^{\text{height}(v)+1}$ . For a node  $v$  with children  $c_1, c_2, \dots, c_w$ ,  $A(v)$  is a circular shift of the concatenation of the elements represented by its children,  $A(c_1) \cdot A(c_2) \cdot \dots \cdot A(c_w)$ . The circular shift is determined by an integer  $\text{off}(v) \in [\text{cap}(v)]$  that is explicitly stored for all nodes. Thus the sequence of elements  $A(v)$  of an internal node  $v$  can be reconstructed by recursively reconstructing the sequence for each of its children, concatenating these and then circular shifting the sequence by  $\text{off}(v)$ . See Figure 6.1 for an illustration. A leaf  $v$  of  $T$  explicitly stores the sequence  $A(v)$  in a circular array  $\text{elems}(v)$  with size  $w$  whereas internal nodes only store their respective offset. Call a node  $v$  full if  $|A(v)| = \text{cap}(v)$  and empty if  $|A(v)| = 0$ . In order to support fast **access**, for all nodes  $v$  the elements of  $A(v)$  are located in consecutive children of  $v$  that are all full, except the children containing the first and last element of  $A(v)$  which may be only partly full.

**Access & Update** To access an element  $A(r)[i]$  at a given index  $i$ ; one traverses a path from the root down to a leaf in the tree. In each node the offset of the node is added to the index to compensate for the cyclic shift, and the traversing is continued in the child corresponding to the newly calculated index. Finally when reaching a leaf, the desired element is returned from the elements array of that leaf. The operation  $\text{access}(v, i)$  returns the element  $A(v)[i]$  and is recursively computed as follows:

**v is internal:** Compute  $i' = (i + \text{off}(v)) \bmod \text{cap}(v)$ , let  $v'$  be the  $\lfloor i'/w \rfloor^{\text{th}}$

child of  $v$  and return the element  $\text{access}(v', i' \bmod \text{cap}(v'))$ .

**v is leaf:** Compute  $i' = (i + \text{off}(v)) \bmod w$  and return the element  $\text{elems}(v)[i']$ .

The time complexity is  $\Theta(l)$  as we visit all nodes on a root-to-leaf path in  $T$ . To navigate this path we must follow  $l - 1$  child pointers, lookup  $l$  offsets, and access the element itself. Therefore this requires  $l - 1 + l + 1 = 2l$  memory probes.

The update operation is entirely similar to access, except the element found is not returned but substituted with the new element. The running time is therefore  $\Theta(l)$  as well. For future use, let  $\text{update}(v, i, e)$  be the operation that sets  $A(v)[i] = e$  and returns the element that was substituted.

**Range Access** Accessing a range of elements, can obviously be done by using the **access**-operation multiple times, but this results in redundant traversing of the tree, since consecutive elements of a leaf often – but not always due to circular shifts – corresponds to consecutive elements of  $A(r)$ . Let  $\text{access}(v, i, m)$  report the elements  $A(v)[i \dots i + m - 1]$  in order. The operation can recursively be defined as:

**v is internal:** Let  $i_l = (i + \text{off}(v)) \bmod \text{cap}(v)$ , and let  $i_r = (i_l + m) \bmod \text{cap}(v)$ . The children of  $v$  that contains the elements to be reported are in the range  $[[i_l \cdot w / \text{cap}(v)], [i_r \cdot w / \text{cap}(v)]] \bmod w$ , call these  $c_l, c_{l+1}, \dots, c_r$ . In order, call  $\text{access}(c_l, i_l, \min(m, \text{cap}(c_l) - i_l))$ ,  $\text{access}(c_i, 0, \text{cap}(c_i))$  for  $c_i = c_{l+1}, \dots, c_{r-1}$ , and  $\text{access}(c_r, i_r - 1, i_r \bmod \text{cap}(c_r))$ .

**v is leaf:** Report the elements  $\text{elems}(v)[i, i + m - 1] \bmod w$ .

The running time of this strategy is  $O(lm)$ , but saves a constant factor over the naive solution.

**Insert & Delete** Inserting an element in the end (or beginning) of the array can simply be achieved using the **update**-operation. Thus the interesting part is fast insertion at an arbitrary position; this is where we utilize the offsets.

Consider a node  $v$ , the key challenge is to shift a big chunk of elements  $A(v)[i, i + m - 1]$  one index right (or left) to  $A(v)[i + 1, i + m]$  to make room for a new element (without actually moving each element in the range). Look at the range of children  $c_l, c_{l+1}, \dots, c_r$  that covers the range of elements  $A(v)[i, i + m - 1]$  to

be shifted. All elements in  $c_{l+1}, \dots, c_{r-1}$  must be shifted. These children are guaranteed to be full, so make a circular shift by decrementing each of their offsets by one. Afterwards take the element  $A(c_{i-1})[0]$  and move it to  $A(c_i)[0]$  using the **update** operation for  $l < i \leq r$ . In  $c_l$  and  $c_r$  only a subrange of the elements might need shifting, which we do recursively. In the base case of this recursion, namely when  $v$  is a leaf, shift the elements by actually moving the elements one-by-one in  $\text{elems}(v)$ .

Formally we define the **shift**( $v, e, i, m$ ) operation that (logically) shifts all elements  $A(v)[i, i + m - 1]$  one place right to  $A[i + 1, i + m]$ , sets  $A[i] = e$  and returns the value that was previously on position  $A[i + m]$  as:

**v is internal:** Let  $i_l = (i + \text{off}(v)) \bmod \text{cap}(v)$ , and let  $i_r = (i_l + m) \bmod \text{cap}(v)$ . The children of  $v$  that must be updated are in the range  $[[i_l \cdot w / \text{cap}(v)], [i_r \cdot w / \text{cap}(v)]] \bmod w$  call these  $c_l, c_{l+1}, \dots, c_r$ . Let  $e_l = \text{shift}(c_l, e, i_l, \min(m, \text{cap}(c_l) - i_l))$ . Let  $e_i = \text{update}(c_i, \text{size}(c) - 1, e_{i-1})$  and set  $\text{off}(c_i) = (\text{off}(c_i) - 1) \bmod \text{cap}(c)$  for  $c_i = c_{l+1}, \dots, c_{r-1}$ . Finally call **shift**( $c_r, e_{r-1}, 0, i_r \bmod \text{cap}(c_r)$ ).

**v is leaf:** Let  $e_o = \text{elems}(v)[(i+m) \bmod w]$ . Move the elements  $\text{elems}(v)[i, (i + m - 1) \bmod w]$  to  $\text{elems}(v)[i + 1, (i + m) \bmod w]$ , and set  $\text{elems}(v)[i] = e$ . Return  $e_o$ .

An insertion **insert**( $i, e$ ) can then be performed as **shift**( $\text{root}, e, i, \text{size}(\text{root}) - i - 1$ ). The running time of an insertion is  $T(l) = 2T(l-1) + w \cdot l \Rightarrow T(l) = O(2^l w)$ .

A deletion of an element can basically be done as an inverted insertion, thus deletion can be implemented using the **shift**-operation from before. A **delete**( $i$ ) can be performed as **shift**( $r, \perp, 0, i$ ) followed by an update of the root's offset to  $(\text{off}(r) + 1) \bmod \text{cap}(r)$ .

**Space** There are at most  $O(w^{l-1})$  nodes in the tree and each takes up constant space, thus the total space of the tree is  $O(w^{l-1})$ . All leaves are either empty or full except the two leaves storing the first and last element of the sequence which might contain less than  $w$  elements. Because the arrays of empty leaves are not allocated the space overhead of the arrays is  $O(w)$ . Thus beyond the space required to store the  $n$  elements themselves, tiered vectors have a space overhead of  $O(w^{l-1})$ .

To obtain the desired bounds  $w$  is maintained such that  $w = \Theta(n^\epsilon)$  where  $\epsilon = 1/l$  and  $n$  is the number of elements in the tiered vector. This can be

achieved by using global rebuilding to gradually increase/decrease the value of  $w$  when elements are inserted/deleted without asymptotically changing the running times. We will not provide the details here. We sum up the original tiered vector data structure in the following theorem:

**THEOREM 6.1 ([64])** *The original  $l$ -tiered vector solves the dynamic array problem for  $l \geq 2$  using  $\Theta(n^{1-1/l})$  extra space while supporting access and update in  $\Theta(l)$  time and  $2l$  memory probes. The operations insert and delete take  $O(2^l n^{1/l})$  time.*

## 6.4 Improved Tiered Vectors

In this paper, we consider several new variants of the tiered vector. This section considers the theoretical properties of these approaches. In particular we are interested in the number of memory accesses that are required for the different memory layouts, since this turns out to have an effect on the experimental running time. In Section 6.5.1 we analyze the actual impact in practice through experiments.

### 6.4.1 Implicit Tiered Vectors

As the degree of all nodes is always fixed at some constant value  $w$  (it may be changed for all nodes when the tree is rebuilt due to a full root), it is possible to layout the offsets and elements such that no pointers are necessary to navigate the tree. Simply number all nodes from left-to-right level-by-level starting in the root with number 0. Using this numbering scheme, we can store all offsets of the nodes in a single array and similarly all the elements of the leaves in another array.

To access an element, we only have to lookup the offset for each node on the root-to-leaf path which requires  $l - 1$  memory probes plus the final element lookup, i.e. in total  $l$  which is half as many as the original tiered vector. The downside with this representation is that it must allocate the two arrays in their entirety at the point of initialization (or when rebuilding). This results in a  $\Theta(n)$  space overhead which is worse than the  $\Theta(n^{1-\epsilon})$  space overhead from the original tiered vector.

**THEOREM 6.2** *The implicit  $l$ -tiered vector solves the dynamic array problem for  $l \geq 2$  using  $O(n)$  extra space while supporting access and update in  $O(l)$*



time requiring  $l$  memory probes. The operations insert and delete take  $O(2^l n^{1/l})$  time.

### 6.4.2 Lazy Tiered Vectors

We now combine the original and the implicit representation, to get both few memory probes and little space overhead. Instead of having a single array storing all the elements of the leaves, we store for each leaf a pointer to a location with an array containing the leaf's elements. The array is lazily allocated in memory when elements are actually inserted into it.

The total size of the offset-array and the element pointers in the leaves is  $O(n^{1-\epsilon})$ . At most two leaves are only partially full, therefore the total space is now again reduced to  $O(n^{1-\epsilon})$ . To navigate a root-to-leaf path, we now need to look at  $l - 1$  offsets, follow a pointer from a leaf to its array and access the element in the array, giving a total of  $l + 1$  memory accesses.

**THEOREM 6.3** *The lazy  $l$ -tiered vector solves the dynamic array problem for  $l \geq 2$  using  $\Theta(n^{1-1/l})$  extra space while supporting access and update in  $\Theta(l)$  time requiring  $l + 1$  memory probes. The operations insert and delete take  $O(2^l n^{1/l})$  time.*

## 6.5 Implementation

We have implemented a generic version of the tiered vector data structure such that the number of tiers and the size of each tier can be specified at compile time. To the best of our knowledge, all prior implementations of the tiered vector are limited to the considerably simpler 2-tier version. Also, most of the performance optimizations applied in the 2-tier implementations do not easily generalize. We have implemented the following variants of tiered vectors:

- *Original* The data structure described in Theorem 6.1.
- *Optimized Original* As described in Theorem 6.1 but with the offset of a node  $v$  located in the parent of  $v$ , adjacent in memory to the pointer to  $v$ . Leaves only consist of an array of elements (since their parent store their offset) and the root's offset is maintained separately as there is no parent to store it in.

- *Implicit* This is the data structure described in Theorem 6.2 where the tree is represented implicitly in an array storing the offsets and the elements of the leaves are located in a single array.
- *Packed Implicit* This is the data structure described in Theorem 6.2 with the following optimization; The offsets stored in the offset array are packed together and stored in as little space as possible. The maximum offset of a node  $v$  in the tree is  $n^{\epsilon(\text{height}(v)+1)}$  and the number of bits needed to store all the offsets is therefore  $\sum_{i=0}^l n^{1-i\epsilon} \log(n^{i\epsilon}) = \log(n) \sum_{i=0}^l i\epsilon n^{1-i\epsilon} \approx \epsilon n^{1-\epsilon} \log(n)$  (for sufficiently large  $n$ ). Thus the  $n^{1-\epsilon}$  offsets can be stored in approximately  $\epsilon n^{1-\epsilon}$  words giving a space reduction of a constant factor  $\epsilon$ . The smaller memory footprint could lead to better cache performance.
- *Lazy* This is the data structure described in Theorem 6.3 where the tree is represented implicitly in an array storing the offsets and every leaf stores a pointer to an array storing only the elements of that leaf.
- *Packed Lazy* This is the data structure described in Theorem 6.3 with the following optimization; The offset and the pointer stored in a leaf is packed together and stored at the same memory location. On most modern 64-bit systems – including the one we are testing on – a memory pointer is only allowed to address 48 bits. This means we have room to pack a 16 bit offset in the same memory location as the elements pointer, which results in one less memory probe during an access operation.
- *Non-Templated* The implementations described above all use C++ templating for recursive functions in order to let the compiler do significant code optimizations. This implementation is template free and serves as a baseline to compare the performance gains given by templating.

In Section 6.6 we compare the performance of these implementations.

### 6.5.1 C++ Templates

We use templates to support storing different types of data in our tiered vector similar to what most other general purpose data structures in C++ do. This is a well-known technique which we will not describe in detail.

However, we have also used *template recursion* which is basically like a normal recursion except that the recursion parameter must be a compile-time constant. This allows the compiler to unfold the recursion at compile-time eliminating all (recursive) function calls by inlining code, and allows better local code optimizations. In our case, we exploit that the height of a tiered vector is constant.

To show the rather simple code resulting from this approach (disregarding the template stuff itself), we have included a snippet of the internals of our access operation:

```
template <class T, class Layer>
struct helper {
    static T& get(size_t node, size_t idx) {
        idx = (idx + get_offset(node)) % Layer::capacity;
        auto child = get_child(node, idx / Layer::child::capacity);
        return helper<T, typename Layer::child>::get(child, idx);
    }
}

template <class T, size_t W>
struct helper<T, Layer<W, LayerEnd> > {
    static T& get(size_t node, size_t idx) {
        idx = (idx + get_offset(node)) % L::capacity;
        return get_elem(node, idx);
    }
}
```

We also briefly show how to use the data structure. To specify the desired height of the tree, and the width of the nodes on each tier, we also use templating:

```
Tiered<int, Layer<8, Layer<16, Layer<32>>>> tiered;
```

This will define a tiered vector containing integers with three tiers. The height of the underlying tree is therefore 3 where the root has 8 children, each of which has 16 children each of which contains 32 elements. We call this configuration 8-16-32.

In this implementation of tiered vectors we have decided to let the number of children on each level be a fixed number as described above. This imposes a maximum on the number of elements that can be inserted. However, in a production ready implementation, it would be simple to make it grow-able by maintaining a single growth factor that should be multiplied on the number of children on each level. This can be combined with the templated solution since the growing is only on the number of children and not the height of the tree (per definition of tiered vectors the height is constant). This will obviously increase the running time for operations when growing/shrinking is required, but will

only have minimal impact on all other operations (they will be slightly slower because computations now must take the growth factor into account).

In practice one could also, for many uses, simply pick the number of children on each level sufficiently large to ensure the number of elements that will be inserted is less than the maximum capacity. This would result in a memory overhead when the tiered vector is almost empty, but by choosing the right variant of tiered vectors and the right parameters this overhead would in many cases be insignificant.

## 6.6 Experiments

In this section we compare the tiered vector to some widely used C++ standard library containers. We also compare different variants of the tiered vector. We consider how the different representations of the data structure listed in Section 6.5, and also how the height of tree and the capacity of the leaves affects the running time. The following describes the test setup:

**Environment** All experiments have been performed on a Intel Core i7-4770 CPU @ 3.40GHz with 32 GB RAM. The code has been compiled with GNU GCC version 5.4.0 with flags “-O3”. The reported times are an average over 10 test runs.

**Procedure** In all tests  $10^8$  32-bit integers are inserted in the data structure as a preliminary step to simulate that it has already been used<sup>1</sup>. For all the access and successor operations  $10^9$  elements have been accessed and the time reported is the average time per element. For range access, 10.000 consecutive elements are accessed. For insertion/deletion  $10^6$  elements have been (semi-)randomly<sup>2</sup> added/deleted, though in the case of “vector” only 10.000 elements were inserted/deleted to make the experiments terminate in reasonable time.

---

<sup>1</sup>In order to minimize the overall running time of the experiments, the elements were not added randomly, but we show this does not give our data structure any benefits

<sup>2</sup>In order to not impact timing, a simple access pattern has been used instead of a normal pseudo-random generator.

### 6.6.1 Comparison to C++ STL Data Structures

In the following we have compared our best performing tiered vector (see the next sections) to the vector and the multiset class from the C++ standard library. The vector data structure directly supports the operations of a dynamic array. The multiset class is implemented as a red-black tree and is therefore interesting to compare with our data structure. Unfortunately, multiset does not directly support the operations of a dynamic array (in particular it has no notion of positions of elements). To simulate an access operation we instead find the successor of an element in the multiset. This requires a root-to-leaf traversal of the red-black tree, just as an access operation in a dynamic array implemented as a red-black tree would. Insertion is simulated as an insertion into the multiset, which again requires the same computations as a dynamic array implemented as a red-black tree would.

Besides the random access, range access and insertion, we have also tested the operations *data dependent access*, insertion in the end, deletion, and *successor* queries. In the *data dependent access* tests, the next index to lookup depends on the values of the prior lookups. This ensures that the CPU cannot successfully pipeline consecutive lookups, but must perform them in sequence. We test insertion in the end, since this is a very common use case. Deletion is performed by deleting elements at random positions. The *successor* queries returns the successor of an element and is not actually part of the dynamic array problem, but is included since it is a commonly used operation on a multiset in C++. It is simply implemented as a binary search over the elements in both the vector and tiered vector tests where the elements are now inserted in sorted order.

The results are summarized in Table 6.1 which shows that the vector performs slightly better than the tiered vector on all access and successor tests. As expected from the  $\Theta(n)$  running time, it performs extremely poor on random insertion and deletion. For insertion in the end of the sequence, vector is also slightly faster than the tiered vector. The interesting part is that even though the tiered vector requires several extra memory lookups and computations, we have managed to get the running time down to less than the double of the vector for access, even less for data dependent access and only a few percent slowdown for range access. As discussed earlier, this is most likely because the entire tree structure (without the elements) fits within the CPU cache, and because the computations required has been minimized.

Comparing our tiered vector to multiset, we would expect access operations to be faster since they run in  $O(1)$  time compared to  $O(\log n)$ . On the other hand, we would expect insertion/deletion to be significantly slower since it runs in  $O(n^{1/l})$  time compared to  $O(\log n)$  (where  $l = 4$  in these tests). We see our

	<i>tiered vector</i>	<i>set</i>	<i>s / t</i>	<i>vector</i>	<i>v / t</i>
access	34.07 ns	1432.05 ns	42.03	21.63 ns	0.63
dd-access	99.09 ns	1436.67 ns	14.50	79.37 ns	0.80
range access	0.24 ns	13.02 ns	53.53	0.23 ns	0.93
insert	1.79 $\mu$ s	1.65 $\mu$ s	0.92	21675.49 $\mu$ s	12082.33
insertion in end	7.28 ns	242.90 ns	33.38	2.93 ns	0.40
successor	0.55 $\mu$ s	1.53 $\mu$ s	2.75	0.36 $\mu$ s	0.65
delete	1.92 $\mu$ s	1.78 $\mu$ s	0.93	21295.25 $\mu$ s	11070.04
memory	408 MB	4802 MB	11.77	405 MB	0.99

**Table 6.1:** The table summarizes the performance of the implicit tiered vector compared to the performance of multiset and vector from the C++ standard library. dd-access refers to data dependent access.

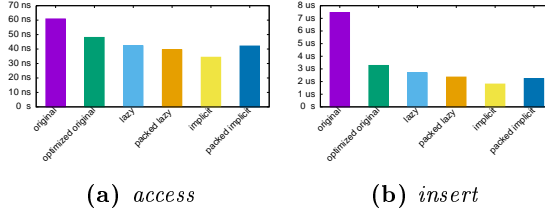
expectations hold for the access operations where the tiered vector is faster by more than an order of magnitude. In random insertions however, the tiered vector is only 8% slower – even when operating on 100,000,000 elements. Both the tiered vector and set requires  $O(\log n)$  time for the successor operation. In our experiments the tiered vector is 3 times faster for the successor operation.

Finally, we see that the memory usage of vector and tiered vector is almost identical. This is expected since in both cases the space usage is dominated by the space taken by the actual elements. The multiset uses more than 10 times as much space, so this is also a considerable drawback of the red-black tree behind this structure.

To sum up, the tiered vectors performs better than multiset on all tests but insertion, where it performs only slightly worse.

### 6.6.2 Tiered Vector Variants

In this test we compare the performance of the implementations listed in Section 6.5 to that of the original data structure as described in 6.1.



**Figure 6.2:** Figures (a) and (b) show the performance of the *original* (—), *optimized original* (—), *lazy* (—), *packed lazy* (—), *implicit* (—) and *packed implicit* (—) layouts.

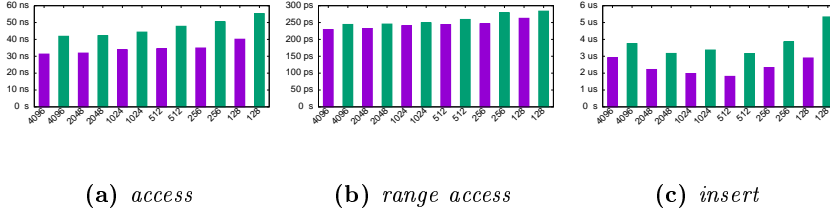
**Optimized Original** By co-locating the child offset and child pointer, the two memory lookups are at adjacent memory locations. Due to the cache lines in modern processors, the second memory lookup will then often be answered directly by the fast L1-cache. As can be seen on Figure 6.2, this small change in the memory layout results in a significant improvement in performance for both access and insertion. In the latter case, the running time is more than halved.

**Lazy and Packed Lazy** Figure 6.2 shows how the fewer memory probes required by the *lazy* implementation in comparison to the original and optimized original results in better performance. Packing the offset and pointer in the leaves results in even better performance for both access and insertion even though it requires a few extra instructions to do the actual packing and unpacking.

**Implicit** From Figure 6.2, we see the implicit data structure is the fastest. This is as expected because it requires fewer memory accesses than the other structures except for the packed lazy which instead has a slight computational overhead due to the packing and unpacking.

As shown in Theorem 6.2 the implicit data structure has a bigger memory overhead than the lazy data structure. Therefore the packed lazy representation might be beneficial in some settings.

**Packed Implicit** Packing the offsets array could lead to better cache performance due to the smaller memory footprint and therefore yield better overall performance. As can be seen on Figure 6.2, the smaller memory footprint did not improve the performance in practice. The simple reason for this, is that the strategy we used for packing the offsets required extra computation. This clearly dominated the possible gain from the hypothesized better cache perfor-



**Figure 6.3:** Figures (a), (b) and (c) show the performance of the *implicit* (—) and the *optimized original* tiered vector (—) for different tree widths.

mance. We tried a few strategies to minimize the extra computations needed at the expense of slightly worse memory usage, but none of these led to better results than when not packing the offsets at all.

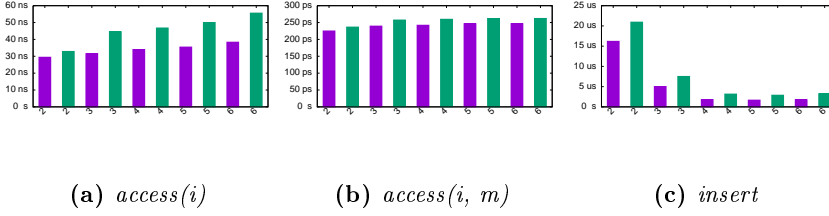
### 6.6.3 Width Experiments

This experiment was performed to determine the best capacity ratio between the leaf nodes and the internal nodes. The six different width configurations we have tested are: 32-32-32-4096, 32-32-64-2048, 32-64-64-1024, 64-64-64-512, 64-64-128-256, and 64-128-128-128. All configurations have a constant height 4 and a capacity of approximately 130 mio.

We expect the performance of access operations to remain unchanged, since the amount of work required only depends on the height of the tree, and not the widths. We expect range access to perform better when the leaf size is increased, since more elements will be located in consecutive memory locations. For *insertion* there is not a clearly expected behavior as the time used to physically move elements in a leaf will increase with leaf size, but then less operations on the internal nodes of the tree has to be performed.

On Figure 6.3 we see access times are actually decreasing slightly when leaves get bigger. This was not expected, but is most likely due to small changes in the memory layout that results in slightly better cache performance. The same is the case for range access, but this was expected. For insertion, we see there is a tipping point. For our particular instance, the best performance is achieved when the leaves have size 512.





**Figure 6.4:** Figures (a),(b) and (c) show the performance of the *implicit* (—) and the *optimized original* tiered vector (—) for different tree heights.

### 6.6.4 Height Experiments

In these tests we have studied how different heights affect the performance of access and insertion operations. We have tested the configurations 8196-16384, 512-512-512, 64-64-64-512, 16-16-32-32-512, 8-8-16-16-16-512. All resulting in the same capacity, but with heights in the range 2-6.

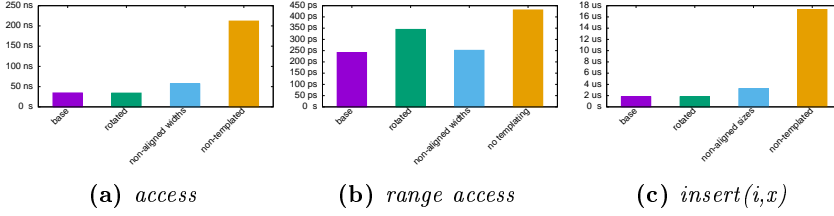
We expect the access operations to perform better for lower trees, since the number of operations that must be performed is linear in the height. On the other hand we expect insertion to perform significantly better with higher trees, since its running time is  $O(n^{1/l})$  where  $l$  is the height plus one.

On Figure 6.4 we see the results follow our expectations. However, the access operations only perform slightly worse on higher trees. This is most likely because all internal nodes fit within the L3-cache. Therefore the running time is dominated by the lookup of the element itself. (It is highly unlikely that the element requested by an access to a random position would be among the small fraction of elements that fit in the L3-cache).

Regarding insertion, we see significant improvements up until a height of 4. After that, increasing the height does not change the running time noticeably. This is most likely due to the hidden constant in  $O(n^{1/l})$  increasing rapidly with the height.

### 6.6.5 Configuration Experiments

In these experiments, we test a few hypotheses about how different changes impact the running time. The results are shown on Figure 6.5, the leftmost result (base) is the implicit 64-64-64-512 configuration of the tiered vector to



**Figure 6.5:** Figures (a) and (b) show the performance of the *base* (—), *rotated* (—), *non-aligned sizes* (—), *non-templated* (—) layouts.

which we compare our hypotheses.

*Rotated:* As already mentioned, the insertions performed as a preliminary step to the tests are not done at random positions. This means that all offsets are zero when our real operations start. The purpose of this test is to ensure that there are no significant performance gains in starting from such a configuration which could otherwise lead to misleading results. To this end, we have randomized all offsets (in a way such that the data structure is still valid, but the order of elements change) after doing the preliminary insertions but before timing the operations. As can be seen on Figure 6.5, the difference between this and the normal procedure is insignificant, thus we find our approach gives a fair picture.

*Non-Aligned Sizes:* In all our previous tests, we have ensured all nodes had an out-degree that was a power of 2. This was chosen in order to let the compiler simplify some calculations, i.e. replacing multiplication/division instructions by shift/and instructions. As Figure 6.5 shows, using sizes that are not powers of 2 results in significantly worse performance. Besides showing that powers of 2 should always be used, this also indicates that not only the number of memory accesses during an operation is critical for our performance, but also the amount of computation we make.

*Non-Templated* The non-templated results in Figure 6.2 show that the change to templated recursion has had a major impact on the running time. It should be noted that some improvements have not been implemented in the non-templated version, but it gives a good indication that this has been quite useful.

## 6.7 Conclusion

This paper presents the first implementation of a generic tiered vector supporting any constant number of tiers. We have shown a number of modified versions of the tiered vector, and employed several optimizations to the implementation. These implementations have been compared to vector and multiset from the C++ standard library. The benchmarks show that our implementation stays on par with vector for access and on update operations while providing a considerable speedup of more than  $40\times$  compared to multiset. At the same time the asymptotic difference between the logarithmic complexity of multiset and the polynomial complexity of tiered vector for insertion and deletion operations only has little effect in practice. For these operations, our fastest version of the tiered vector suffers less than 10% slowdown. Arguably, our tiered array provides a better trade-off than the balanced binary tree data structures used in the standard library for most applications that involves big instances of the dynamic array problem.

## CHAPTER 7

# Parallel Lookups in String Indexes

---

Anders Roy Christiansen<sup>†</sup>      Martín Farach-Colton<sup>\*</sup>

<sup>†</sup> The Technical University of Denmark

<sup>\*</sup> Rutgers University

### Abstract

Recently, the first PRAM algorithms were presented for looking up a pattern in a suffix tree. We improve the bounds, achieving optimal results.

## 7.1 Introduction

Looking up a pattern string in an index is one of the most basic primitives in stringology, and the suffix tree (and its suffix array representation) is among the most basic indexes. It is therefore surprising that, until recently, there were no known PRAM algorithms for looking up an  $m$ -character pattern  $P$  in a suffix tree of an  $n$ -character text  $T$ . This contrasts sharply with the rich PRAM literature for the problem of finding all occurrences of  $P$  in  $T$  in the case where  $P$  can be preprocessed, optimal solutions of which are known for the full range of PRAM models [33, 60, 133]

Recently Jekovec and Brodnik [79] considered the problem of parallel lookups in an index, specifically suffix trees and quadratic-space suffix tries. They achieved work-time optimal  $O(m)$  work and  $O(\log m)$  time for suffix trie lookups in the CREW PRAM, although the preprocessing involves quadratic work and space. For suffix tree lookups, they achieve  $O(m \log m)$  work and  $O(\log m)$  time by augmenting the  $O(n)$ -size suffix tree with further data structures<sup>1</sup> that increase the size to  $O(n \log n)$ . These bounds are time-optimal due to the  $\Omega(\log n)$  time lower bound for computing the OR of  $n$ -bits [31] in the CREW PRAM.

Fischer et al. [46] gave an CREW PRAM algorithm using the suffix array and some additional compact data structures requiring a total of  $n \log n + O(n)$  bits (ie.  $n + o(n)$  words), thus improving the space. Their algorithm uses  $O(\log \log m \log \log n + \log m)$  time and  $O(m + \min(m, \log n)(\log m + \log \log m \log \log n))$  work. Additionally they considered the approximate pattern lookup problem and lookups in compressed suffix arrays.

In this paper, we improve the bounds for looking up a pattern in an index in several ways. First, we provide an algorithm that matches the time-work optimal bounds of  $O(\log m)$  time,  $O(m)$  work while achieving  $O(n)$  space. Also, our algorithm runs on the EREW PRAM, thus improving on the earlier CREW PRAM algorithms. As in the previous algorithms, we use randomization, but only in the preprocessing, whereas the pattern matching phase is deterministic<sup>2</sup>.

We consider two variants of the pattern lookup problem: exact matching and prefix matching. In exact matching, we find the place in the suffix tree where the complete pattern matches. In prefix matching, we find the location in the suffix tree which matches the longest possible prefix of the pattern.

Our main result is:

**THEOREM 7.1** *Given a suffix tree of a string  $T$  of length  $n$  and a pattern  $P$  of length  $m$ , then parallel prefix pattern lookup in the suffix tree takes worst-case  $O(\log m)$  time and  $O(m)$  work, after  $O(\log n \log^* n)$  time and  $O(n)$  work preprocessing w.h.p. requiring  $O(n)$  additional space. All bounds are on the EREW PRAM model.*

In order to present this result, we first present a simpler but similar method that does more work during preprocessing and does not support prefix pattern lookups. Both results augment suffix trees with Karp-Rabin fingerprints [83] and perfect hashing [8]. The final result is obtained by reducing the number

<sup>1</sup>Suffix trees of subsets of characters, hash tables, etc.

<sup>2</sup>Both earlier results involve hashing, as does ours. We give our bounds using fast, randomized perfect hashing, rather than slow, deterministic perfect hashing.

of strings that must be guaranteed to have collision-free KR fingerprints by discarding possible false-positives during a query. We note that the technique of combining indexes with Karp-Rabin fingerprints for efficient pattern lookups was introduced in [3], but in that case it was to improve sequential dictionary pattern matching.

Furthermore we include a simple algorithm for parallel prefix pattern lookup in a suffix array because it is deterministic in both the query and preprocessing phases and works on general alphabets whereas the first works on integer alphabets, at the cost of some running time. The result is summarized below:

**THEOREM 7.2** *Prefix pattern matching in a suffix array with LCP-values can be performed in  $O(\log n)$  time and  $O(m + \log n)$  work on the CRCW PRAM model with no other preprocessing than computing the suffix array and the LCP array.*

## 7.2 Preliminaries

Denote by  $T$  a text of length  $n$  of letters from an alphabet  $\Sigma$ . Call the corresponding suffix tree  $S$ . For an edge  $e \in S$  denote by  $T(e)$  the string of letters on the path from the root to  $e$  and including the letters on  $e$ . Similarly let  $\text{pre}(e)$  denote the string of letters on the path from the root to  $e$  including only the first letter on  $e$ . Let  $\text{parent}(e)$  be the edge that shares a node with  $e$  and is on the  $e$ -root path. Let  $T[i]$  be the  $i$ th character of  $T$  and  $T[i, j]$  be the substring of  $T$  from the  $i$ th character to the  $j$ th character, both inclusive.

In this paper we will be working in the PRAM model see [77] for details. We present all results based on the work-time presentation framework described in [77] (ie. without having the number of processors as a parameter).

We will be using the following lemmas throughout our solutions:

**LEMMA 7.3 (FOLLOWS FROM LIST RANK IN [29])** *Given a set of linked lists represented by a table of length  $n$  of next-pointers and the index of a head element one can compute which elements are in the linked list that contains the head. This can be done in  $O(\log n)$  time and  $O(n)$  work in the EREW PRAM model.*

**LEMMA 7.4 ([44])** *Given a table  $B$  of  $n$  bits, one can find the leftmost 1-bit in  $B$  in  $O(1)$  time and  $O(n)$  work in the CRCW PRAM model.*

**LEMMA 7.5 (FOLLOWS FROM PREFIX SUM [93])** *Given a string  $T$  of length  $n$ , all prefix Karp-Rabin fingerprints [83]  $\phi(T[1,1]), \phi(T[1,2]), \dots$  can be computed in  $O(\log n)$  time and  $O(n)$  work in the EREW PRAM model.*

**LEMMA 7.6 (FROM [8], ADAPTED TO EREW)** *Given a (multi-)set of  $n$  integers a perfect hash table of size  $n$  can be computed in time  $O(\log n \log^* n)$  using  $O(n)$  work and space w.h.p. in the EREW PRAM model.*

### 7.3 Simple Fingerprint-Based Pattern Lookup

The main idea in this solution is to use a combination of Karp-Rabin fingerprints and perfect hash tables to avoid doing an actual traversal of the suffix tree from the root. We first show a simplified version of this solution, and then extend it to reduce preprocessing time and to support prefix lookups.

**Data Structure.** Let  $\phi$  be a Karp-Rabin based fingerprint function that is collision free for all substrings in  $T$ . We store the string  $T$ , the suffix tree  $S$  for  $T$ , and a perfect hash table  $H_d$  for each  $d = 1 \dots n$  mapping  $H_{|\text{pre}(e)|}[\phi(\text{pre}(e))] \rightarrow e$  for each edge  $e$  in  $S$ . These structures use  $O(n)$  space in total.

**Query.** Given a pattern  $P$ , first compute the prefix fingerprints of  $P$  using Lemma 7.5. In parallel, look up a fingerprint  $\phi(P[1,d])$  in hash table  $H_d$ , for all  $d = 1 \dots m$ . If there is a match, let  $M[d] = H_d[\phi(P[1,d])]$ , and otherwise let  $M[d] = \perp$ . Since all lookups are in different hash tables there are no read conflicts. Find the rightmost non- $\perp$  value in  $M$  and call it  $e_c$ . If  $P$  occurs in  $T$  then this match must be on  $e_c$  in the suffix tree. Match  $P$  character-by-character to  $T(e_c)[1,m]$ . If there are no differences, report that  $P$  exists on  $e_c$  in  $T$ ; otherwise, report that  $P$  does not occur.

Since all characters of  $P$  are compared to a substring of  $T$  before reporting an occurrence, no false positives are reported. We need this verification part because our fingerprints are only guaranteed to be collision-free on  $T$ , not on  $P$ .

If  $P$  does occur in  $T$ , then the fingerprint function is guaranteed to be collision-free in both  $P$  and  $T$ , and so we will find a single maximal  $e_c$  so that a prefix of  $P$  matches with  $\text{pre}(e_c)$ . The brute-force matching phase then extends the match length down the edge  $e_c$ .

The bottleneck of the query is the  $O(\log m)$ -time,  $O(m)$ -work of computing the fingerprints (Lemma 7.5), and the same time and work to verify a match. We conclude that these are overall work-time bounds.

**Preprocessing.** We assume the suffix tree is given<sup>3</sup>. In  $O(\log n)$  time and  $O(n)$  work we can compute all prefix fingerprints of  $T$  using Lemma 7.5. From these prefix fingerprints the fingerprint of an arbitrary substring of  $T$  can be computed in constant time and work.

Validate that  $\phi$  is collision-free for the substrings of  $T$  by computing all possible fingerprints. Since there are  $\Theta(n^2)$  different substrings this takes  $O(n^2)$  work. They can all be calculated independently, but  $O(n)$  fingerprints might depend on the same fingerprint prefix which means the algorithm might need to read the same memory cell at the same time. Since a CREW algorithm can be simulated as an EREW algorithm with  $O(\log n)$  time overhead per step [77], this takes  $O(\log n)$  time. Construct a hash table over all the fingerprints using Lemma 7.6 to check for duplicates - if there are any duplicates, start over with a new random Karp-Rabin fingerprint function. In total this takes  $O(\log n \log^* n)$  time and  $O(n^2)$  work w.h.p.

Finally constructing the  $n$  different hash tables with a total of  $O(n)$  elements can be done in  $O(\log n \log^* n)$  time and  $O(n)$  work w.h.p. using Lemma 7.6.

Overall preprocessing takes  $O(\log n \log^* n)$  time and  $O(n^2)$  work, both w.h.p.

## 7.4 Better Fingerprint-Based Pattern Lookup

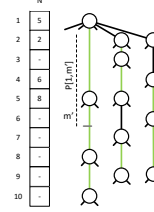
We now show how to improve the above solution such that the preprocessing work will be  $O(n)$  w.h.p. instead of  $O(n^2)$ . Furthermore, this method will support general prefix pattern lookups. These improvements are achieved by reducing the number of substrings of  $T$  that must be guaranteed to have collision-free fingerprints from  $O(n^2)$  to  $O(n)$ , and instead taking care of possible false positives during the query.

**Data Structure.** The data structure used is the same as above with the difference that the fingerprint function  $\phi$  is only guaranteed to be collision free for the substrings  $\text{pre}(e)$  for all  $e \in S$ , of which there are  $O(n)$ .

<sup>3</sup>Though, in fact, the suffix tree of  $T$  can be constructed in  $O(\log^2 n)$  time,  $O(n \log n)$  work and  $O(n)$  space [81] for general alphabets.



**Figure 7.1:** An illustration of a small part of the suffix tree  $S$ . Green edges represent the edges in  $M$ . As illustrated they all form disjoint monotone paths. A prefix of the pattern  $P$  of length  $m'$  occurs on the left-most path in the illustration. An example of the  $N$ -array is included. The y-position of a node represents the string depth.



**Query.** Given a pattern  $P$ , first compute its prefix fingerprints. In parallel, look up the fingerprint  $\phi(P[1, d])$  in the respective hash table  $H_d$  for all  $d = 1 \dots m$ . If there is a match set  $M[d] = H_d[\phi(P[1, d])]$  otherwise  $M[d] = \perp$ . If  $M[1] = \perp$  then  $P$  does not occur in  $T$ , so in this case stop and report no match. The edges contained in  $M$  form a set of disjoint paths in  $S$  (see proof below). Consider each of these paths to be a linked list of edges. Let  $N[i]$  be a table describing the next-pointers ie. which edge  $M[N[i]]$  follows  $M[i]$ . Define  $e = M[d]$  and  $e' = \text{parent}(e)$ , and set  $N[|\text{pre}(e')|] = d$  if  $e \neq \perp$  and  $M[|\text{pre}(e')|] = e'$ . Let  $N[d] = \perp$  denote unset entries. Use Lemma 7.3 to compute which edges are in the same linked list as  $M[1]$ , let  $d$  be the index of the right-most of these. Now  $e_c = M[d]$  is our candidate edge. In parallel find the longest prefix of the strings  $P$  and  $T(e_c)[1, m]$  that matches. Report the result.

Before reporting any results we verify by comparing  $P$  to a substring of  $T$ , so that no false-positives are reported.

We focus on proving that we always find a (prefix) match of  $P$  in  $T$  if it exists. So assume a non-empty prefix of  $P$  exists somewhere in  $T$ . In this case there is a path  $\hat{P}P$  from the root spelling out this prefix of  $P$ . We now need to show  $\hat{P}P$  is a prefix of the path  $PP$  from the root to the edge  $e_c$  our algorithm picks as the candidate edge for verification.

Consider the set of edges the algorithm finds in  $M$ . All edges  $e \in \hat{P}P$  are in this set as  $P[1, |\text{pre}(e)|] = \text{pre}(e) \Rightarrow \phi(P[1, |\text{pre}(e)|]) = \phi(\text{pre}(e))$ . If the fingerprint function were collision-free (even with  $P$ ), then this set of edges would be exactly the edges on  $\hat{P}P$ . Unfortunately, this is not the case for the restricted-collision-free fingerprint function we are using. In our case the set of edges form a disjoint set of monotone paths in  $S$  as illustrated in Figure 7.1. To prove this, we show that at most one outgoing edge of a node can be in  $M$ . Assume to the contrary that  $e_1$  and  $e_2$  are both outgoing edges of a node with string depth  $d$  and they are both in  $M$ . Then  $\phi(P[1, d+1]) = \phi(\text{pre}(e_1))$  and  $\phi(P[1, d+1]) = \phi(\text{pre}(e_2))$ , which implies that  $\phi(\text{pre}(e_1)) = \phi(\text{pre}(e_2))$ . This

contradicts that the fingerprint function is collision free for strings  $\text{pre}(e)$  where  $e \in S$ .

Since all edges in  $\text{PP}$  are in  $M$  and any node can have at most one outgoing edge, the path we are interested in is the one containing the root of  $S$ . All other paths can safely be discarded. Therefore we use Lemma 7.3 to remove all edges of  $M$  not connected to the root. Since all the edges on  $\hat{\text{P}}$  are on this path and we pick the deepest,  $\hat{\text{P}}$  is a prefix of  $\text{PP}$ . This completes the proof.

**Preprocessing.** All steps of the preprocessing are similar to the steps of the preprocessing before with the only exception we only need to verify our Karp-Rabin fingerprint function is collision free on a set of  $O(n)$  strings. As this was the bottleneck on the work before, the work is now reduced to  $O(n)$  w.h.p.

## 7.5 Parallel Suffix Array Pattern Lookup

Here we describe a parallelization of [98], which has the advantage of working for any alphabet and of being deterministic in both query and preprocessing. The query run time is slower.

Manber's algorithm performs a binary search over the suffix array. It maintains an interval  $[L, R] \subseteq [1, n]$  of the suffix array wherein potential matches lie. In each round the middle element  $M$  in  $[L, R]$  is found, and it is determined if the search should continue in the interval  $[L, M]$  or  $[M, R]$ . This is accomplished by matching  $P$  to  $T[\text{SA}[M], \text{SA}[M] + m]$ . Finding the leftmost mismatch between the two strings in parallel takes  $O(1)$  time and  $O(m)$  work using Lemma 7.4. There are  $O(\log n)$  rounds, so the overall time is  $O(\log n)$  and the work is  $O(m \log n)$ .

This method can be generalized to the algorithm that uses the LCP-array as well. If we just keep comparing the current suffix with the entire part of  $P$  that has not yet been matched we will obtain the same time and work bounds as above. By a small modification, the work can be reduced to  $O(m)$  as follows. Instead of comparing all of the pattern to the current suffix the algorithm should perform the comparison in chunks of size  $\frac{m}{\log n}$ .

In rounds where no more than  $\frac{m}{\log n}$  characters match, the total work is  $O(m + \log n)$ . In the remaining rounds, the total work is  $O(m)$ . Thus the overall time is still  $O(\log n)$  but the work is reduced to  $O(m + \log n)$ .



# Bibliography

---

- [1] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *Proceedings of the 11th Annual Symposium on Discrete Algorithms*. Citeseer, 2000.
- [2] Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *Proc. 39th FOCS*, pages 534–543, 1998.
- [3] Amihood Amir, Martin Farach, and Yossi Matias. Efficient randomized dictionary matching algorithms. In *Proc. 3rd CPM*, pages 262–275, 1992.
- [4] Amihood Amir, Gad M Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM TALG*, 3(2):19, 2007.
- [5] A. Apostolico and S. Lonardi. Some theory and practice of greedy off-line textual substitution. In *Proc. DCC*, pages 119–128, 1998.
- [6] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *Proc. DCC*, pages 143–152, 2000.
- [7] Alberto Apostolico and Stefano Lonardi. Off-line compression by greedy textual substitution. *Proceedings of the IEEE*, 88(11):1733–1744, 2000.
- [8] Holger Bast and Torben Hagerup. Fast and reliable parallel hashing. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 50–61. ACM, 1991.
- [9] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Karkkainen, A. Ordonez, S.J. Puglisi, and Y. Tabei. Queries on lz-bounded encodings. In *Proc. DCC*, pages 83–92, April 2015.

- [10] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *Proc. 18th ESA*, pages 427–438, 2010.
- [11] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *Proc. 18th ESA*, pages 427–438, 2010.
- [12] Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Proc. 23rd ESA*, 2015.
- [13] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Inform. Process. Lett.*, 5(3):82 – 87, 1976.
- [14] Philip Bille, Patrick Hagge Cording, and Inge Li Gørtz. Compressed subsequence matching and packed tree coloring. *Algorithmica*, pages 1–13, 2015.
- [15] Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. In *Proc. 13th SWAT*, 2013.
- [16] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. Time-space trade-offs for lempel-ziv compressed indexing. In *28th Annual Symposium on Combinatorial Pattern Matching*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017.
- [17] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014.
- [18] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2014. Announced at SODA 2011.
- [19] Guy E. Blelloch, Bruce M. Maggs, and Shan Leung Maverick Woo. Space-efficient finger search on degree-balanced search trees. In *Proc. 14th SODA*, pages 374–383, 2003.
- [20] Richard P Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974.
- [21] Gerth Stølting Brodal. Finger search trees. In *Handbook of Data Structures and Applications*. Chapman and Hall/CRC, 2004.

- [22] Gerth Stølting Brodal, George Lagogiannis, Christos Makris, Athanasios K. Tsakalidis, and Kostas Tsichlas. Optimal finger search trees in the pointer machine. *J. Comput. Syst. Sci.*, 67(2):381–418, 2003.
- [23] Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures, WADS '99*, pages 37–48, London, UK, UK, 1999. Springer-Verlag.
- [24] Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range searching on the ram, revisited. In *Proc. 27th SOCG*, pages 1–10, 2011.
- [25] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005. Announced at STOC 2002 and SODA 2002.
- [26] BG Chern, Idoia Ochoa, Alexandros Manolakos, Albert No, Kartik Venkat, and Tsachy Weissman. Reference based genome compression. In *IEEE ITW*, pages 427–431, 2012.
- [27] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fund. Inform.*, 111(3):313–337, 2011.
- [28] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don’t cares. In *Proc. 36th STOC*, pages 91–100, 2004.
- [29] Richard Cole and Uzi Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 478–491. IEEE, 1986.
- [30] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32 – 53, 1986.
- [31] Stephen Cook, Cynthia Dwork, and Rüdiger Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.
- [32] Patrick Hagge Cording, Paweł Gawrychowski, and Oren Weimann. Bookmarks in grammar-compressed strings. In *Proc. 23rd SPIRE*, pages x–y, 2016.
- [33] Artur Czumaj, Zvi Galil, Leszek Gąsieniec, Kunsoo Park, and Wojciech Plandowski. Work-time-optimal parallel algorithms for string problems. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 713–722. ACM, 1995.

- [34] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. 1st WADS*, pages 39–46, 1989.
- [35] Paul F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proceedings of the Workshop on Algorithms and Data Structures, WADS '89*, pages 39–46, London, UK, UK, 1989. Springer-Verlag.
- [36] Paul F. Dietz and Rajeev Raman. A constant update time finger search tree. *Inf. Process. Lett.*, 52(3):147–154, 1994.
- [37] Huy Hoang Do, Jesper Jansson, Kunihiro Sadakane, and Wing-Kin Sung. Fast relative Lempel–Ziv self-index for similar sequences. *TCS*, 532:14–30, 2014.
- [38] M. Farach and M. Thorup. String Matching in Lempel–Ziv Compressed Strings. *Algorithmica*, 20(4):388–404, 1998.
- [39] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proc. 7th CPM*, pages 130–140. Springer, 1996.
- [40] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [41] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
- [42] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Succinct representation of sequences. Technical report, 2004.
- [43] Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. *TCS*, 372(1):115 – 121, 2007.
- [44] Faith E Fich, Prabhakar Ragde, and Avi Wigderson. Relations between concurrent-write models of parallel computation. *SIAM Journal on Computing*, 17(3):606–627, 1988.
- [45] Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating lz77 via small-space multiple-pattern matching. In *Algorithms-ESA 2015*, pages 533–544. Springer, 2015.
- [46] Johannes Fischer, Dominik Köppl, and Florian Kurpicz. On the benefit of merging suffix array intervals for parallel pattern matching. In *Proc. 27th CPM*, 2016.
- [47] Rudolf Fleischer. A simple balanced search tree with  $O(1)$  worst-case update time. *Int. J. Found. Comput. Sci.*, 7(2):137–150, 1996.

- [48] Greg N. Frederickson. Implicit data structures for the dictionary problem. *J. ACM*, 30(1):80–94, January 1983.
- [49] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 345–354, New York, NY, USA, 1989. ACM.
- [50] M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 1–7, New York, NY, USA, 1990. ACM.
- [51] Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.
- [52] Michael L Fredman. The complexity of maintaining an array and computing its partial sums. *Journal of the ACM (JACM)*, 29(1):250–260, 1982.
- [53] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47(3):424–436, 1993.
- [54] P. Gage. A new algorithm for data compression. *The C Users J.*, 12(2):23 – 38, 1994.
- [55] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. A faster grammar-based self-index. In *Proc. 6th LATA*, pages 240–251, 2012.
- [56] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th LATIN*, pages 731–742. Springer, 2014.
- [57] Travis Gagie, Paweł Gawrychowski, and Simon J. Puglisi. Approximate pattern matching in lz77-compressed texts. *J. Discrete Algorithms*, 32:64–68, 2015.
- [58] Travis Gagie, Christopher Hoobin, and Simon J. Puglisi. Block graphs in practice. In *Proc. ICABD*, pages 30–36, 2014.
- [59] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. 06 2017.
- [60] Zvi Galil. A constant-time optimal parallel string-matching algorithm. *Journal of the ACM (JACM)*, 42(4):908–918, 1995.



- [61] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Łącki, and Piotr Sankowski. Optimal dynamic strings. *arXiv preprint arXiv:1511.02612*, 2015.
- [62] Paweł Gawrychowski, Moshe Lewenstein, and Patrick K Nicholson. Weighted ancestors in suffix trees. In *Proc. 22nd ESA*, pages 455–466. 2014.
- [63] Leszek Gąsieniec, Roman Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th DCC*, page 458, 2005.
- [64] Michael T. Goodrich and John G. Kloss. *Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences*, pages 205–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.
- [65] Mayank Goswami, Allan Grønlund, Kasper Green Larsen, and Rasmus Pagh. Approximate range emptiness in constant time and optimal space. In *Proc. 26th SODA*, pages 769–775, 2015.
- [66] Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *Proc. 26th CPM*, pages 219–230. Springer, 2015.
- [67] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850, 2003.
- [68] Leonidas J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In *Proc. 9th STOC*, pages 49–60, 1977.
- [69] Torben Hagerup. Sorting and searching on the word ram. In *STACS 98*, pages 366–398. Springer, 1998.
- [70] Dov Harel and Robert E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
- [71] Wing-Kai Hon, Kunihiro Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *TCS*, 412(39):5176–5186, 2011.
- [72] Wing-Kai Hon, Kunihiro Sadakane, and Wing-Kin Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *Theoretical Computer Science*, 412(39):5176–5186, 2011.

- [73] Christopher Hoobin, Simon J Puglisi, and Justin Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB*, 5(3):265–273, 2011.
- [74] Thore Husfeldt and Theis Rauhe. New lower bound techniques for dynamic partial sums and related problems. *SIAM J. Comput.*, 32(3):736–753, 2003.
- [75] Thore Husfeldt, Theis Rauhe, and Søren Skyum. Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. In *Proc. 5th SWAT*, pages 198–211, 1996.
- [76] Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Inform. Comput.*, 240:74–89, 2015.
- [77] Joseph JáJá. *An introduction to parallel algorithms*. addison Wesley, 1992.
- [78] Jesper Jansson, Kunihiro Sadakane, and Wing-Kin Sung. CRAM: Compressed random access memory. In *Proc. 39th ICALP*, pages 510–521. 2012.
- [79] Matevž Jekovec and Andrej Brodnik. Parallel query in the suffix tree. *arXiv preprint arXiv:1509.06167*, 2015.
- [80] Artur Jež. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms (TALG)*, 11(3):20, 2015.
- [81] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.
- [82] Juha Kärkkäinen and Esko Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. *Proceedings of the 3rd South American Workshop on String Processing (WSP’96)*, 26(Teollisuuskatu 23):141–155, 1996.
- [83] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
- [84] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
- [85] Jyrki Katajainen. *Worst-Case-Efficient Dynamic Arrays in Practice*, pages 167–183. Springer International Publishing, Cham, 2016.
- [86] Jyrki Katajainen and Bjarke Buur Mortensen. *Experiences with the Design and Implementation of Space-Efficient Deques*, pages 39–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

- [87] Brian Kernighan and Dennis Ritchie. *The C Programming Language (1st Ed.)*. Prentice-Hall, 1978.
- [88] J. C. Kieffer and E. H. Yang. Grammar based codes: A new class of universal lossless source codes. *IEEE Trans. Inf. Theory*, 46(3):737–754, 2000.
- [89] J. C. Kieffer, E. H. Yang, G. J. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Trans. Inf. Theory*, 46(5):1227 – 1245, 2000.
- [90] S. Rao Kosaraju. Localized search in sorted lists. In *Proc. 13th STOC*, pages 62–69, New York, NY, USA, 1981.
- [91] Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proc. 17th SPIRE*, pages 201–206, 2010.
- [92] Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proc. 34th ACSC*, pages 91–98, 2011.
- [93] Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM (JACM)*, 27(4):831–838, 1980.
- [94] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.
- [95] Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. In *Proc. 31st STACS*, pages 506–517, 2014.
- [96] Stan Y. Liao, Srinivas Devadas, and Kurt Keutzer. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. Design Autom. Electr. Syst.*, 4(1):12–38, 1999.
- [97] Stan Y. Liao, Srinivas Devadas, Kurt Keutzer, Steven W. K. Tjiang, and Albert Wang. Code optimization techniques in embedded DSP microprocessors. *Design Autom. for Emb. Sys.*, 3(1):59–73, 1998.
- [98] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.
- [99] K. Mehlhorn and S. Nähler. Bounded ordered dictionaries in  $O(\log \log N)$  time and  $O(n)$  space. *Inform. Process. Lett.*, 35(4):183–189, 1990.
- [100] Kurt Mehlhorn. A new data structure for representing sorted lists. In *Proc. WG*, pages 90–112, 1981.

- [101] Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.
- [102] Gonzalo Navarro and Veli Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007.
- [103] Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. In *Proc. 24th SODA*, pages 865–876, 2013.
- [104] Gonzalo Navarro and Alberto Ordóñez. Grammar compressed sequences with rank/select support. In *21st SPIRE*, pages 31–44. Springer, 2014.
- [105] Gonzalo Navarro and Kunihiro Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Alg.*, 10(3):16, 2014.
- [106] Craig G Nevill-Manning and Ian H Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *J. Artificial Intelligence Res.*, 7:67–82, 1997.
- [107] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *Proc. 41st MFCS*, pages 72:1–72:15, 2016.
- [108] Takaaki Nishimoto, I Tomohiro, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index, LZ factorization, and LCE queries in compressed space, apr 2015.
- [109] Mihai Pătraşcu and Erik D Demaine. Tight bounds for the partial-sums problem. In *Proc. 15th SODA*, pages 20–29, 2004.
- [110] Mihai Patrascu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
- [111] Mihai Pătraşcu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *Proc. 55th FOCS*, pages 166–175, 2014.
- [112] Benny Porat and Ely Porat. Exact and approximate pattern matching in the streaming model. In *Proc. 50th FOCS*, pages 315–323, 2009.
- [113] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
- [114] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct dynamic data structures. In *Proc. 7th WADS*, pages 426–437. 2001.

- [115] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. *Succinct Dynamic Data Structures*, pages 426–437. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.
- [116] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.
- [117] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.*, 302(1):211–222, 2003.
- [118] Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. 17th SODA*, pages 1230–1239, 2006.
- [119] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proceedings of 37th Conference on Foundations of Computer Science*, Oct 1996.
- [120] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.
- [121] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Byte Pair encoding: A text compression scheme that accelerates pattern matching. *Technical Report DOI-TR-161, Dept. of Informatics, Kyushu University*, 1999.
- [122] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.
- [123] James A. Storer and Thomas G. Szymanski. The macro model for data compression. In *Proc. 10th STOC*, pages 30–39, 1978.
- [124] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.
- [125] Bjarne Stroustrup. *The C++ Programming Language: Special Edition (3rd Edition)*. Addison-Wesley, 2000. First edition from 1985.
- [126] Toshiya Tanaka, I Tomohiro, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing convolution on grammar-compressed text. In *Proc. 23rd DCC*, pages 451–460, 2013.
- [127] I Tomohiro. Longest common extensions with recompression. *CoRR*, abs/1611.05359, 2016.
- [128] I Tomohiro, Takaaki Nishimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Compressed automata for dictionary matching. *Theor. Comput. Sci.*, 578:30–41, 2015.

- [129] 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.
- [130] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6(3):80–82, 1977.
- [131] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.
- [132] Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th CPM*, pages 247–258, 2013.
- [133] Uzi Vishkin. Optimal parallel pattern matching in strings. *Information and control*, 67(1-3):91–113, 1985.
- [134] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.
- [135] Dan E Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM J. Comput.*, 29(3):1030–1049, 2000.
- [136] E. H. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – part one: Without context models. *IEEE Trans. Inf. Theory*, 46(3):755–754, 2000.
- [137] Andrew C Yao. On the complexity of maintaining partial sums. *SIAM Journal on Computing*, 14(2):277–288, 1985.
- [138] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, (3), 1977.
- [139] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.
- [140] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory*, 24(5):530–536, 1978.