Compressed and efficient algorithms and data structures for strings

Ettienne, Mikko Berggren

Publication date:
2018

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

Citation (APA):
Ettienne, M. B. (2018). *Compressed and efficient algorithms and data structures for strings*. DTU Compute. DTU Compute PHD-2018 Vol. 490

Technical University of Denmark

Compressed and Efficient Algorithms and Data Structures for Strings

Mikko Berggren Ettienne

DTU Compute
Department of Applied Mathematics and Computer Science
PHD-2018-490

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

PHD-2018-490
ISSN: 0909-3192

Preface

This dissertation was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfillment of the requirements for acquiring a PhD degree. It is comprised by four joint peer-reviewed publications and one unpublished paper which is the result of my research as part of the project Compressed Computation on Highly-Repetitive Data partially funded by the Danish Research Council (DFF – 4005-00267). The results were obtained under the supervision of Associate Professor Philip Bille and Associate Professor Inge Li Gørtz from May 2015 to August 2018 including 3 months of paternity leave.

Acknowledgements. First and foremost I want extend my sincere gratitude to my supervisors Inge and Philip. They have been a great inspiration and have been available whenever needed for advice, discussions and support during the inevitable academic and personal ups and downs of my PhD. A big thanks to professor John Iacono from New York University who was kind enough to invite my to New York City and set aside time for academic discussion and research. I also want to thank my colleagues and the people in the community for their inspiring dedication and their warm and welcoming attitude. Thanks to the people in the AlgoloG group at The Technical University of Denmark for contributing to the pleasant working environment. In particular, thank you Anders, Patrick, Steen, Thomas, Nicola, Eva, Frederik, Søren and Hjalte. I have truly enjoyed the multitude of discussions we have had around everything and nothing at all and I believe you all qualify as my friends. Last but not least, I thank my friends, family and in particular Natasja for her love, patience and support.

Mikko Berggren Ettienne
Copenhagen, August 2018

Abstract

In this dissertation we study the design of efficient algorithms, data structures, and protocols for computation on compressed data and manipulation of long sequences. We consider the following topics:

Time-Space Trade-Offs for Lempel–Ziv Compressed Indexing Given a string S, the *compressed indexing problem* is to preprocess S into a compressed representation that supports fast *pattern matching queries*. That is, given a string P, report all occurrences of P in S. The goal is to use little space relative to the compressed size of S while supporting fast queries. We present a compressed index based on the Lempel–Ziv 1977 compression scheme. We obtain the following time-space trade-offs: For constant-sized alphabets

(i) $O(m + \mathrm{occ} \lg\lg n)$ time using $O(z \lg(n/z) \lg\lg z)$ space, or

(ii) $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ time using $O(z \lg(n/z))$ space,

For integer alphabets polynomially bounded by n

(iii) $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ time using $O(z(\lg(n/z) + \lg\lg z))$ space, or

(iv) $O(m + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ time using $O(z(\lg(n/z) + \lg^{\varepsilon} z))$ space,

where n and m are the length of the input string and query string respectively, z is the number of phrases in the LZ77 parse of the input string, occ is the number of occurrences of the query in the input and $\varepsilon > 0$ is an arbitrarily small constant. In particular, (i) improves the leading term in the query time of the previous best solution from $O(m \lg m)$ to $O(m)$ at the cost of increasing the space by a factor $\lg\lg z$. Alternatively, (ii) matches the previous best space bound, but has a leading term in the query time of $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}))$. However, for any polynomial compression ratio, i.e., $z = O(n^{1-\delta})$, for constant $\delta > 0$, this becomes $O(m)$. Our index also supports extraction of any substring of length ℓ in $O(\ell + \lg(n/z))$ time. Technically, our results are obtained by novel extensions and combinations of existing data structures of independent interest, including a new batched variant of weak prefix search.

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 $O(m + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ time using $O(z \lg(n/z))$ space where z is the size of the LZ77 parse of S and $\varepsilon > 0$ is an arbitrarily small constant, when the alphabet is small or $z = O(n^{1-\delta})$ for any constant $\delta > 0$. 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. These 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.

Decompressing Lempel-Ziv Compressed Text We consider the problem of decompressing the Lempel–Ziv 77 representation of a string S of length n using a working space as close as possible to the size z of the input. The folklore solution for the problem runs in $O(n)$ time but requires random access to the whole decompressed text. A better solution is to convert LZ77 into a grammar of size $O(z \lg(n/z))$ and then stream S in linear time. In this paper, we show that $O(n)$ time and $O(z)$ working space can be achieved for constant-size alphabets. On larger alphabets, we describe (i) a trade-off achieving $O(n \lg^{\delta} \sigma)$ time and $O(z \lg^{1-\delta} \sigma)$ space for any $0 \leq \delta \leq 1$ where σ is the size of the alphabet, and (ii) a solution achieving $O(n)$ time and $O(z \lg \lg n)$ space. Our solutions can, more generally, extract any specified subsequence of S with little overheads on top of the linear running time and working space. As an immediate corollary, we show that our techniques yield improved results for pattern matching problems on LZ77-compressed text.

Compressed Communication Complexity of Longest Common Prefixes We consider the communication complexity of fundamental longest common prefix (LCP) problems. In the simplest version, two parties, Alice and Bob, each hold a string, A and B, and we want to determine the length of their longest common prefix $\ell = \text{LCP}(A, B)$ using as few rounds and bits of communication as possible. We show that if the longest common prefix of A and B is compressible, then we can significantly reduce the number of rounds compared to the optimal uncompressed protocol, while achieving the same (or fewer) bits of communication. Namely, if the longest common prefix has an LZ77 parse of z phrases, only $O(\lg z)$ rounds and $O(\lg \ell)$ total communication is necessary. We extend the result to the natural case when Bob holds a set of strings $B_1, \ldots, B_k$, and the goal is to find the length of the maximal longest prefix shared by A and any of $B_1, \ldots, B_k$. Here, we give a protocol with $O(\lg z)$ rounds and $O(\lg z \lg k + \lg \ell)$ total communication. We present our result in the public-coin model of computation but by a standard technique our results generalize to the private-coin model. Furthermore, if we view the input strings as integers the problems are the greater-than problem and the predecessor problem.

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 $O(1)$ and insertion and deletion in time $O(n^{\varepsilon})$ for $\varepsilon > 0$ 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.

DANISH ABSTRACT

I denne afhandling studeres design af effektive algoritmer, datastrukturer og protokoller til beregninger på komprimeret data og manipulation af lange sekvenser. Vi behandler følgende emner:

Tidspladsafvejninger for Lempel–Ziv Komprimeret Indeksering Komprimeret indeksering er at præprocessere en given streng S til en komprimeret repræsentation, som understøtter hurtige *mønstergenkendelsesforespørgsler*. Det vil sige, givet en streng P, kaldet mønsteret, rapporter alle forekomster af P i S. Målet er at bruge lidt plads relativt til den komprimerede størrelse af S og samtidig understøtte hurtige forespørgsler. Vi præsenterer et komprimeret indeks baseret på Lempel–Ziv 1977 komprimeringsteknikken. Vi opnår de følgende tidspladsafvejninger:

For alfabeter af konstant størrelse:

(i) $O(m + \mathrm{occ} \lg\lg n)$ tid og $O(z \lg(n/z) \lg\lg z)$ plads, eller

(ii) $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ tid og $O(z \lg(n/z))$ plads,

For heltalsalfabeter polynomielt begrænset af n

(iii) $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ tid og $O(z(\lg(n/z) + \lg\lg z))$ plads, eller

(iv) $O(m + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ tid og $O(z(\lg(n/z) + \lg^{\varepsilon} z))$ plads,

hvor n er længden af strengen S, m er længden af forespørgslen, z er antallet af fraser i LZ77-fortolkningen af S, occ er antallet af forekomster af forespørgslen i S og $\varepsilon > 0$ er en arbitrært lille konstant. I særdeleshed forbedrer (i) den ledende term i forespørgseltiden af den hidtil bedste løsning fra $O(m \lg m)$ til $O(m)$ på bekostning af en faktor $\lg\lg z$ forøgelse af pladsforbruget. Alternativt har (ii) et pladsforbrug tilsvarende det hidtil bedste, men har $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}))$ som ledende term i forespørgselstiden. Dette reducerer imidlertid til $O(m)$ når $z = O(n^{1-\delta})$ hvor $\delta > 0$ er konstant. Vores indeks understøtter også dekomprimering af enhver delstreng af længde ℓ i $O(\ell + \lg(n/z))$ tid. Vores resultat er opnået ved originale udvidelser og kombinationer af eksisterende datastrukturer som kan være relevante i andre sammenhænge, herunder en ny variant af svag præfiks søgning.

Komprimeret Indeksering med Signaturgrammatikker Komprimeret indeksering er at præprocessere en given streng S til en komprimeret repræsentation, som understøtter mønstergenkendelsesforespørgsler. Det vil sige, givet en streng P af længde m, kaldet mønsteret, rapporter alle forekomster af P i S.

Vi præsenterer en datastruktur, der understøtter mønstergenkendelsesforespørglser i $O(m + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ tid og bruger $O(z \lg(n/z))$ plads hvor z er antallet af fraser i LZ77-fortolkningen af S og $\varepsilon > 0$ er en arbitrært lille konstant, når alfabetet er

lille eller $z = O(n^{1-\delta})$ hvor $\delta > 0$ er konstant. Vi præsenterer også to datastrukturer for det generelle tilfælde; en hvor pladsforbruget forøges med $O(z \lg \lg z)$, og en hvor forespørgselstiden ændres fra at være i værste tilfælde til at være i forventning. Dette forbedrer de hidtil bedste resultater. Det er desuden bemærkelsesværdigt at dette er den første datastruktur, som kan afgøre om P forekommer i S i $O(m)$ tid og $O(z \lg(n/z))$ plads.

Vores resultater er primært opnået gennem en original kombination af en randomiseret grammatikkonstruktionsalgoritme og velkendte teknikker inden for mønstergenkendelse og 2D-område rapportering.

Dekomprimering af Lempel–Ziv-Komprimeret Tekst Vi undersøger hvorledes Lempel–Ziv 77 repræsentationen af en streng S af længde n kan dekomprimeres ved brug af arbejdshukommelse så tæt som muligt på z, som er størrelsen på input. Den velkendte teknik tager $O(n)$ tid men kræver løbende adgang til den dekomprimerede tekst. En alternativ og bedre løsning er at omdanne LZ77 repræsentationen til en grammatik af størrelse $O(z \lg(n/z))$ og derefter streame S i lineær tid. I denne artikel viser vi at $O(n)$ tid og $O(z)$ arbejdshukommelse kan opnås for alfabeter af konstant størrelse. For større alfabeter beskriver vi to afvejninger mellem tid og plads. Det første kræver $O(n \lg^{\delta} \sigma)$ tid og $O(z \lg^{1-\delta} \sigma)$ plads hvor $0 \leq \delta \leq 1$ er konstant og σ er størrelsen på alfabetet. Det andet kræver $O(n)$ tid og $O(z \lg \lg n)$ plads. Vores løsninger kan også dekomprimere et vilkårlig specificeret sæt af delstrenge af S med små forøgelser i køretid og arbejdshukommelse. Vi viser desuden at vores teknikker forbedrer eksisterende resultater for mønstergenkendelse på LZ77-komprimeret tekst som en naturlig følge.

Komprimeret Kommunkationskompleksitet af Længste Fælles Præfiks Vi betragter kommunikationskompleksiteten af fundamentale længste fælles præfiks (LCP) problemer. I den simpleste version har Alice og Bob hver en streng, A og B og vi ønsker at bestemme længden af strengenes længste fælles præfiks $\ell = \text{LCP}(A, B)$ med minimal kommunikation målt i antallet af bits kommunikeret og antallet af runder. Vi viser at hvis det længste fælles præfiks af A og B kan komprimeres, så kan vi reducere antallet af runder signifikant og fastholde (eller reducere) antallet af bits sammenlignet med den optimale protokol for ikke-komprimerede strenge. Helt præcist er $O(\lg z)$ runder og $O(\lg \ell)$ total kommunikation tilstrækkeligt, hvis LZ77-fortolkningen af det længste fælles præfiks har z fraser.

Vi udvider resultatet to til den generelle version hvor Bob har et sæt af strenge $B_1, \ldots, B_k$, og målet er at finde længden af det maksimale af de længste præfikser mellem A og en enhver af $B_1, \ldots, B_k$. For dette problem giver vi en protokol med $O(\lg z)$ runder og $O(\lg z \lg k + \lg \ell)$ total kommunikation.

Vi præsenterer vores resultater i public-coin modellen, men de kan ved brug af standardteknikker generaliseres til private-coin modellen. Vi bemærker at de nævnte problemer er ækvivalente med større-end problemet og prædecessor problemet, hvis man betragter strengene som heltal.

Hurtige Dynamiske Tabeller Vi præsenterer en højt optimeret implementering a lagdelte vektorer, en datastruktur som vedligeholder en sekvens af n elementer og understøtter tilgang i $O(1)$ tid og indsættelse og sletning i $O(n^{\varepsilon})$ tid, for $\varepsilon > 0$, og bruger $o(n)$ ekstra plads. Vi undersøger flere forskellige implementeringsoptimeringer i C++ og sammenligner ydeevnen med vector og multiset fra standard biblioteket på sekvenser med op til 10^8 elementer. Vores hurtigste implementering bruger væsentligt mindre plads end multiset og forøger hastigheden af tilgangsoperationer med en faktor 40 sammenlignet med multiset og forøger hastigheden af indsættelse og sletning med en faktor 10.000 sammenlignet med vector, imens hastigheden for de resterende operationer ikke forværres nævneværdigt.

CONTENTS

CHAPTER 1

INTRODUCTION

The amount of data being produced and consumed has been growing with an accelerating pace throughout the last couple of decades and there are no obvious signs of a slowdown of this trend.

As a consequence, vast amounts of data needs to be stored, transmitted and processed in order to facilitate extraction of meaningful information. An obvious and increasingly relevant component in handling huge data sets is compression. Repetitive data can be compressed and stored in less space than the original, and for highly repetitive data the difference can be significant.

In this dissertation we design algorithms, data structures and protocols for computations on compressed data and manipulation of huge data sets. The results presented appear in the following papers.

Chapter 2 **Time-Space Trade-Offs for Lempel–Ziv Compressed Indexing**
Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj.
In Theoretical Computer Science, volume 713, 2018. An extended abstract appeared in the Proceedings of the 28th Symposium on Combinatorial Pattern Matching, 2017.

Chapter 3 **Compressed Indexing with Signature Grammars**
Anders Roy Christiansen, and Mikko Berggren Ettienne.
In Proceedings of the 13th Latin American Symposium on Theoretical Informatics, 2018.

Chapter 4 **Fast Lempel-Ziv Decompression in Linear Space**
Philip Bille, Mikko Berggren Ettienne, Travis Gagie, Inge Li Gørtz, and Nicola Prezza
Unpublished. Will be submitted for publication in 2018.

Chapter 5 **Compressed Communication Complexity of Longest Common Prefixes**
Philip Bille, Mikko Berggreen Ettienne, Roberto Grossi, Inge Li Gørtz, and Eva Rotenberg
In the proceedings of the 25th International Symposium on String Processing and Information Retrieval, 2018.

Chapter 6 **Fast Dynamic Arrays**
Philip Bille, Anders Roy Christiansen, Mikko Berggren Ettienne, and Inge Li Gørtz
In Proceedings of the 25th Annual European Symposium on Algorithms, 2017.

1.1 Overview

Chapters 2 through 6 are verbatim copies of the papers listed above in the given order. The first four papers present theoretical results whereas the last is mainly experimental. Chapters 2 through 5 are concerned with compressed computation. In chapters 2 and 3 we consider the problem of designing compressed indexing data structures supporting efficient queries. Chapter 4 deals with the algorithmic problem of how to efficiently decompress specified parts of a data set that is given in a compressed representation. In Chapter 5 we consider how to exploit compression to obtain a low communication complexity for a fundamental string problem. Lastly, Chapter 6 describes an implementation of a practical dynamic array data structure and experimentally show that it offers significant speedups over library implementations of related data structures when dealing with many elements.

Section 1.2 and 1.3 of this chapter gives some basic terminology and an introduction to some of the fundamental techniques, concepts and compression schemes that are relevant to the rest of the dissertation. The sections 1.4 through 1.7 give an overview of the problems we consider in the following chapters by outlining the problem, related work and the results. In Section 1.8 we consider ideas for future work.

We note in accordance with the plagiarism checking process recently implemented by The Technical University of Denmark that the contents of this chapter is based on the papers that appear in chapters 2 through 6. Some definitions and presentations of results might coincide with what is found in later chapters.

1.2 Preliminaries

There will be minor variations in notation style between chapters because each paper appears verbatim, but the papers are self-contained and introduce relevant models, notation and preliminaries whenever necessary. We here give some basic notation relevant to the rest of this chapter:

A string S of length $n = |S|$ is a sequence of n symbols $S[1] \cdots S[n]$ drawn from an alphabet Σ of size $\sigma = |\Sigma|$. The sequence $S[i, j]$ is the *substring* of S given by $S[i] \cdots S[j]$. We use $[n]$ as shorthand notation for $\{1, 2, \ldots, n\}$.

1.2.1 Model of Computation

The results in chapters 2 through 4 are based on the word-RAM model [50]. The word-RAM is an abstract random-access machine where each memory cell stores a w-bit integer which we refer to as a *word* where w is a positive integer called the *word size*. The model assumes that $w = \Omega(\lg n)$ where n is the number of cells required to specify the input. It follows that a word can hold the address of any of the input cells.

The model allows basic operations such as load, store, jump and comparison and also bitwise and arithmetic operations such as addition, multiplication, division, logical shifts, etc. to be carried out on words in constant time. Similarly reading and writing any cell is a constant-time operation.

The time used by an algorithm is the number of such constant-time operations it performs and the space usage is the number of distinct memory cells it writes to during operation. We note that we do not count the input cells toward the space usage and we assume that the input is available in read-only memory.

Most of our algorithms take as input a sequence of symbols from some alphabet. Unless otherwise noted, we always assume that this alphabet consists of integers and that the size of it is polynomially bounded by the length of the input sequence. It follows that any symbol can be stored in $O(1)$ words.

1.2.2 LZ77

The LZ77 algorithm is a seminal lossless compression algorithm by Abraham Lempel and Jacob Ziv [91]. The algorithm provides theoretical guarantees as well as great efficiency in practice and has therefore formed the basis of widespread compression techniques such as 7zip, gzip and PNG to name a few.

The *LZ77 parse* of a string S of length n from an alphabet Σ divides S into z substrings $f_1 f_2 \dots f_z$, called *phrases*, in a greedy left-to-right order. Let $u_1 = 1$, and $u_i = \sum_1^{i-1} |f_i| + 1$ for $i > 1$. Then the i^{th} phrase f_i is the longest substring starting at position u_i that has at least one occurrence starting to the left of u_i plus the following symbol. To compress S (assume for simplicity that S is terminated with the special symbol \$) we represent each phrase as a tuple $(s_i, l_i, \alpha_i) \in ([n] \times [n] \times \Sigma \cup \$)$, such that s_i is the position of a previous occurrence, l_i is the length of the occurrence, and α_i is the symbol at position $u_i + l_i$. We define $s_1 = 0$ and it follows that $l_1 = 0$ and $\alpha_1 = S[1]$. We call the substring $S[s_i, s_i + l_i - 1]$ the *source* and $S[u_i + l_i]$ the *border* of the i^{th} phrase $f_i = S[u_i, u_i + l_i]$. If $s_i + l_i > u_i$ for some $i \in [z]$ such that the source of the i^{th} phrase overlaps the phrase, we say that the parse is self-referential. Sometimes, we restrict ourselves to parses that are not self-referential and thus require that $s_i + l_i \leq u_i$ for $i \in [z]$.

Text: `dissertation_dissemination$`

LZ77-representation:$(0,0,d)(0,0,i)(0,0,s)(3,1,e)(0,0,r)(0,0,t)(0,0,a)$
$(7,1,i)(0,0,o)(0,0,n)(0,0,\_)(1,5,m)(2,1,n)(8,5,\$)$

The string `dissertation_dissemination$` contains 27 symbols while its LZ77 parse consists of 14 phrases.

Text: `(abc)`^{n}`$`

LZ77-representation :$(0,0,a)(0,0,b)(0,0,c)(1,3n-3,\$)$

The string formed by repeating the string `abc` n times has length $3n$ and is compressed into a self-referential parse with $O(1)$ phrases.

Bounds The LZ77 parse of a string contains $O(n/\log_\sigma n)$ phrases and as every phrase can be represented in $O(1)$ words this space usage is $O(n \lg \sigma)$ bits, which is asymptotically optimal in the worst case. The asymptotic optimality also carries over to statistical measures of compression such as empirical entropy (see Section 1.3 for more details on this). Furthermore the LZ77 algorithm performs particularly well when considering highly repetitive text. This follows naturally from its definition due to the fact that any repeated occurrence of a substring is represented by a single phrase which requires $O(1)$ words in the compressed representation.

On top of both the practical and theoretical efficiency of the LZ77 algorithm, it also offers linear compression and decompression time. The LZ77 parse of a string S can be found greedily from the suffix tree of S. Decompression is done incrementally phrase by phrase. A phrase $f_i = (s_i, l_i, \alpha_i)$ is decompressed simply by appending the symbol α_i to the source of the phrase $S[s_i, s_i + l_i - 1]$ which can be read from the already decompressed prefix $f_1 f_2 \cdots f_{i-1}$ of S.

Variations Due to its widespread academic and practical influence, there are several variations of the LZ77 algorithm. Limiting the search for identical substrings to a sliding window that is maintained during compression is a well-studied technique both in theory and practice [64]. It offers increased compression speed and less working space at the cost of (potentially) worse compression. Other variations like LZ78 [92] and LZ-end [59] make it easier to do computations on the compressed data but are also provably inferior to LZ77 when it comes to compression.

Left vs Right Observe that the LZ77 parse of a string is not unique according to the definition given above because it does not specify which substring to choose as a source for a phrase when there are multiple possible choices. One way to resolve this ambiguity, is to choose either the leftmost or the rightmost substring. The rightmost parse is interesting when one is interested in the number of bits required to represent the LZ77 parse and not only the number of phrases. By specifying the position of a source s_i relative to the position of the corresponding phrase u_i, we can hope to reduce the magnitude of the number. This makes a difference if we further compress the parse by using variable-length integer encoders.

1.2.3 Grammar Compression

A grammar is a set of symbols called *non-terminals,* a disjoint set of symbols called *terminals*, a set of *production rules* and a special non-terminal symbol called the *starting symbol*. The production rules describe how to map one string of terminals and at least one non-terminal into a possibly empty string of terminals and non-terminals. A grammar describes a formal language, namely the set of strings over the alphabet of the terminals that can be generated by repeated symbol replacements according to the production rules starting from the starting symbol. We use upper-case letters for the non-terminals and lower-case letters for the terminals and write production rules as $aAb \to acb$ meaning that the string aAb can be replaced by the string acb. A grammar is *context-free* if the left-hand side of all production rules is a single non-terminal.

Grammars have many applications in computer science, but a grammar can also be seen as a compact way to represent the language it generates. When compressing a single string, we can restrict our focus to *context-free* grammars that produce exactly one string. This implies that:

- Every non-terminal appears as the left-hand side in exactly one production rule.
- The grammar is acyclic, i.e. if A appears in a string that can be generated from B then B does not appear in a string that can be generated from A.

For simplicity, we sometime restrict our focus even further and consider context-free grammars in Chomsky normal form that produce only one string. A context-free grammar is in Chomsky normal form if the right-hand side of all rules is either a pair of non-terminals or a single terminal and the starting symbol does not appear in the right-hand side of any rule. A context-free grammar in Chomsky normal form that produces exactly one string is in fact a Straight-Line Program (SLP) and has the form

$$X_1 \to expr_1, X_2 \to expr_2, \ldots, X_n \to expr_n$$

where X_i is a non-terminal and $expr_i$ is either a terminal or a pair of non-terminals $X_j X_k$ where $j, k < i$ and X_n is the starting symbol.

Text: $(\texttt{abc})^n$ (assume n is a power of 2)

Grammar representation:

$$\begin{aligned} X_1 &\rightarrow a \\ X_2 &\rightarrow b \\ X_3 &\rightarrow c \\ X_4 &\rightarrow X_1 X_2 \\ X_5 &\rightarrow X_4 X_3 \\ X_6 &\rightarrow X_5 X_5 \\ X_7 &\rightarrow X_6 X_6 \\ &\vdots \\ X_{\lg n+5} &\rightarrow X_{\lg n+4} X_{\lg n+4} \end{aligned}$$

The string formed by repeating the string `abc` n times has length $3n$ and can be represented by a grammar with $O(\lg n)$ production rules.

An SLP can also be viewed as an directed acyclic graph (DAG) in which the vertices are the terminal and non-terminal symbols of the grammar. The starting symbol is the root of the DAG. If $X_i \rightarrow X_j X_k$ is a production rule then the vertex X_i has two outgoing edges, one to each of the vertices X_j and X_k marked as a left edge and a right edge respectively. If $X_l \rightarrow \alpha$ is a production rule (where α is a terminal) then X_l has a single outgoing edge to the vertex α. There is a one-to-one correspondence between the vertices of the DAG and the symbols of the grammar end every rule induces one or two edges in the DAG. Therefore a grammar has the same size as its corresponding DAG and we can easily go from one representation to the other in linear time.

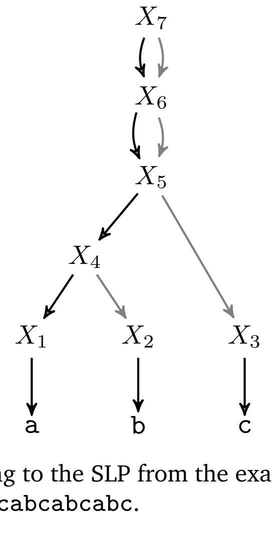


Figure 1.1: The DAG corresponding to the SLP from the example above where $n = 4$. The DAG produces the string `abcabcabcabc`.

The *derivation tree* sometimes called the *parse tree* is a tree induced by the grammar. It is a rooted ordered tree in which every vertex corresponds to a terminal or non-terminal. The starting symbol is the root of the tree. If $X_i \rightarrow X_j X_k$ is a production rule then X_j is

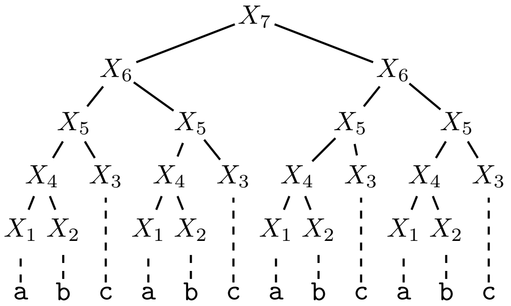


Figure 1.2: The derivation tree of the DAG from the examples above producing the string abcabcabcabc

the left child of X_i and X_k is the right child of X_i. If $X_l \to \alpha$ is a production rule (where α is a terminal) then α is the sole child of X_l.

The sequence formed by the left-to-right order of the leaves of the derivation tree is the string produced by the grammar. We sometimes leave out all rules of the form $X_i \to \alpha$ and replace all occurrences of X_i with α. This implies that all rules have two symbols in the right-hand side. This gives a more uniform representation and the derivation tree becomes a binary tree. The change does not affect the string generated by the grammar or its asymptotic size, but it is no longer in Chomsky normal form. A grammar is *balanced* when its derivation tree is balanced according to some balance criteria, e.g. if the length of any path is asymptotically bounded by the logarithm to the number of vertices in the tree.

Because all internal vertices are branching (except the parents of the leaves), the size of the derivation tree is linear in the length of the string produced by the grammar. The DAG is in fact a compact representation of this tree where identical subtrees are only explicitly represented once. It is easy to see from the derivation tree that every non-terminal symbol generates a substring of the string generated by the grammar. This gives some intuition to why grammars can compress repetitive strings: If a string has many repetitions of a certain substring, then this can be captured by a grammar because the subtree(s) generating this substring only have to be represented once.

Bounds and Construction Computing the exact size of the smallest grammar producing a string is NP-complete and even $o(\lg n / \lg \lg n)$ approximations would require progress on a well-studied algebraic problem [63]. Grammar compression is also not as good as LZ77 compression. In fact the smallest grammar for a string S of length n always has at least as many rules as there are phrases in the LZ77 parse of S and there are families of strings where the number of rules in the smallest grammar is $\Theta(z \lg n)$. Grammars remain interesting for compression because they are easier to do computations on compared to LZ77 representations. For example, decompressing any length ℓ substring of S can be done in $O(\ell + \lg n)$ time by a data structure using linear space in the grammar [13]. In comparison, there are no data structures achieving $o(z \lg \lg n)$ time while using linear space in the LZ77 representation of S.

There are several algorithms that produce a grammar from a given string each offering different trade-offs with respect to measures such as compression ratio, construction speed, balance, practical efficiency, etc.

Chariker et al. and Rytter [18, 84] have shown how to construct a balanced grammar in $O(z \lg(n/z))$ time and space given the LZ77 parse of a string S of length n . Because the size of the smallest grammar is bounded from below by the number of phrases in the LZ77 parse, these construction techniques produce a grammar that is a logarithmic approximation of the smallest grammar. No better approximation algorithms are known.

A different family of construction algorithms provide grammars that are *locally consistent*. A grammar is locally consistent when identical substrings of S are generated by almost identical subtrees of the derivation tree of the grammar. This property is useful when comparing substrings of strings that are represented as grammars, because we can compare the non-terminals that produce these strings instead of the actual strings. In Chapter 3 we consider the *signature grammar* which is produced by a randomized construction technique. It is locally consistent, balanced and has size $O(z \lg(n/z))$ in expectation and its properties form the basis of the compressed index we design.

1.3 Compression, Compact Representations and Succinct Data Structures

The information-theoretic lower bound of representing a length j sequence of k-bit integers is jk bits. In general, assume that n is the information-theoretical minimum number of bits needed to store some data and let D be some representation of this data. If D uses $n + O(1)$ bits of space it is called *implicit*, if it uses $n + o(n)$ bits of space it is called *succinct* and if it uses $O(n)$ bits it is called *compact*.

When it comes to compression, there are some subtleties with respect to lower bounds. For instance, consider a bit string of length n. There are 2^n distinct bit strings of this length, thus $\lg 2^n = n$ bits are required to represent each bit string simply in order to distinguish it from the others (assuming that we require that all representations are of similar length). However, consider the subset of strings where all 1-bits appear consecutively. We can easily represent each such string in $2 \lg n$ bits by storing how many 1-bits it contains and the position of the first 1-bit.

This example serves to illustrate that the lower bounds are relative to the data we consider. If we consider only bit strings with some specified set of regularities, then the $\Omega(n)$ lower bound does not apply. One important source for compression is statistical regularities. Assume that we assign a *code* to every symbol in the alphabet of a sequence and represent the sequence by replacing every symbol by its code. If we allow the codes to be of variable length, then we can choose a short code for symbols that occur with high frequency and use longer codes for symbols that occur less frequently. The 0^{th} order empirical entropy is a lower bound on the average number of bits required per symbol using this technique. For a bit string S it is defined as $\mathcal{H}_0(S) = p \lg \frac{1}{p} + (1-p) \lg \frac{1}{1-p}$ where p is the probability of having a 1-bit and $1 - p$ is the probability of having a 0-bit at any position of the text. This concept can be extended to k^{th} order empirical entropy which is the average number of bits required per symbol by a statistical compressor that can use a length k context when deciding a code for a symbol. This means that the code for each symbol can be a function of itself and the k preceding symbols. A huge body of important and relevant research considers this field of compression. For instance, entropy compressed text indexes aim at using $O(n\mathcal{H}_k(S)) + o(n)$ bits of space to represent a string S of length n while supporting fast queries.

It holds that $\lg \sigma \geq \mathcal{H}_0 \geq \mathcal{H}_1 \geq \cdots \geq \mathcal{H}_k$ however this measure is only sensible when k is at most logarithmic in the length of the string because the number of codes we have to store in order to be able to decompress grows exponentially with k.

For practical applications of text compression, the k^{th} order empirical entropy model captures many of the regularities present in text with one important exception. Highly repetitive texts formed by sets of very similar strings are very compressible. The intuition is that we can represent a set of very similar strings $S_1, S_2, \ldots, S_k$ compactly by storing S_1 and then only the differences between S_1 and S_i for $i > 1$. If the differences are few and the strings are long and many, this yields a very effective compression. However, this form of regularity is not captured by k^{th} order empirical entropy. In fact, the k^{th} order empirical entropy is virtually unchanged between an original text and a twice as long text constructed by concatenating the original with itself.

Conversely, the LZ77 compression technique captures repetitive regularities very well. For example, the parse of the concatenation of two identical texts has only $O(1)$ more phrases than the parse of one of the texts. Furthermore, for collections characterized by a high degree of repetitiveness, e.g. DNA strings, the Wikipedia database, version-controlled document collections, etc. the notions of implicit, succinct and compact data structures also becomes less relevant. This is because the different overheads in space usage are constant factors which can be significantly outweighed by the savings obtained by compressing repetitive regularities. The magnitude of such constant factors may still have an impact in practice, but we argue that it is much less relevant for repetitive texts. As already mentioned in Section 1.2.2, the LZ77 compression technique also performs well with respect to emperical entropy and in fact it holds that $z \lg n \leq n\mathcal{H}_k(S) + o(n \lg \sigma)$ when $k = o(\lg_\sigma n)$ [73]. Thus LZ77 converges to the k^{th} order entropy and is therefore asymptotically optimal with respect to this measure.

The algorithms and data structures designed in chapters 2 through 5 of this dissertation are aimed at highly repetitive data and therefore measure the space usage (and time whenever relevant) relative to the number of LZ77 phrases in the parse of the text in question.

1.4 Chapters 2 & 3: Compressed Indexing

The *indexing problem* is to preprocess a string S of length n into a data structure that supports efficient pattern matching queries. The *pattern matching problem* is to determine the location of all occurrences of a pattern P of length m in a string S of length n often called the text. Sometimes, we are only interested in whether P occurs in S and do not care about the number of occurrences and their locations. We call this variation the *pattern existence problem*. The suffix tree is a well-known solution that solves the indexing problem using $O(n)$ space and $O(m + \mathrm{occ})$ time to report all occ occurrences of a pattern P in S.

The *compressed indexing* problem is to preprocess a string S into a compressed representation that supports efficient pattern matching queries. The goal is to use little space relative to the compressed size of S while supporting fast queries. In this dissertation we measure the space usage of the index relative to the LZ77 representation of the string S, but in general, the compressed representation can be any measure or compression scheme.

Over the last decades, the compressed indexing problem has received significant attention as shown by the multitude of both theoretical and practical solutions, see e.g. [8, 21, 29–32, 40, 47–49, 54, 55, 60, 66, 67, 74] and the surveys [43, 72–74]. The problem is very relevant as the amount of highly repetitive data increases rapidly and compressed indexing makes it possible to query large compressed data sets without decompressing them first. The rapid increase of repetitive data has several sources such as DNA sequencing, version control repositories, regular backups, etc.

When considering compressed indexing based on LZ77 compression relatively few results are known. Let n, m, and z denote the length of the input string, the length of the pattern string, and the number of phrases in the LZ77 parse of the string respectively. Kärkkäinen and Ukkonen introduced the problem in 1996 [55] and gave an initial solution that required read-only access to the uncompressed text. Interestingly, this work is among the first results in compressed indexing [74]. More recently, Gagie et al. [39,40] and Nishimoto et al. [77] revisited the problem. See Table 1.1 for an overview of recent results.

Results and Techniques in Chapter 2 We show that if the LZ77 parse of S consists of z phrases we can solve the compressed indexing problem in $O(m + \mathrm{occ} \lg\lg n)$ time using

Table 1.1: Selection of previous results and our results on compressed indexing. The variables are the text size n, z is the number of phrases in the LZ77 parse, m is the pattern length, occ is the number of occurrences and σ is the size of the alphabet. (The time complexity marked by † is expected whereas all others are worst-case)

Index	Space	Locate time	σ
Gagie et al. [40]	$O(z\lg(n/z))$	$O(m\lg m + \mathrm{occ}\lg\lg n)$	$O(1)$
Nishimoto et al. [77]	$O(z\lg n\lg^* n)$	$O(m\lg\lg n\lg\lg z + \lg z\lg m\lg n(\lg^* n)^2 + \mathrm{occ}\lg n)$	$n^{O(1)}$
Chapter 2	$O(z(\lg(n/z) + \lg^\varepsilon z))$	$O(m + \mathrm{occ}(\lg^\varepsilon n + \lg\lg n))$	$n^{O(1)}$
Chapter 2	$O(z\lg(n/z)\lg\lg z)$	$O(m + \mathrm{occ}\lg\lg n)$	$O(1)$
Chapter 2	$O(z\lg(n/z))$	$O(m(1 + \frac{\lg^\varepsilon z}{\lg(n/z)}) + \mathrm{occ}(\lg^\varepsilon n + \lg\lg n))$	$O(1)$
Chapter 3	$O(z\lg(n/z))$	$O(m + \mathrm{occ}(\lg^\varepsilon z + \lg\lg n))$	$O(1)$
Chapter 3	$O(z(\lg(n/z) + \lg\lg z))$	$O(m + \mathrm{occ}(\lg^\varepsilon z + \lg\lg n))$	$n^{O(1)}$
Chapter 3	$O(z(\lg(n/z))$	$O(m + \mathrm{occ}(\lg^\varepsilon z + \lg\lg n))^\dagger$	$n^{O(1)}$

$O(z\lg(n/z)\lg\lg z)$ space for constant-sized alphabets and $O(m+\mathrm{occ}(\lg\lg n+\lg^\varepsilon z))$ time using $O(z(\lg(n/z) + \lg^\varepsilon z))$ space for integer alphabets of size $n^{O(1)}$.

An important property of the LZ77 parse observed by Farach and Thorup [27] is that any occurrence of a pattern P in S either crosses a phrase border in the LZ77 parse of S, or there is an earlier occurrence that does. This means that the occurrences can be split into *primary occurrences* which are the ones that cross a phrase border and *secondary occurrences* which are the ones that do not. Given a list of the primary occurrences, each of the secondary can be found in $O(\lg\lg n)$ time by a data structure using $O(z\lg\lg n)$ space [55]. So far, finding the primary occurrences is therefore the harder problem. At its core, the technique of Kärkkäinen and Ukkonen [55] is to search for the pattern at all locations that could possibly be a primary occurrence. This can be done efficiently by searching in two tries storing $O(z)$ relevant substrings of S located around phrase borders combined with a 2D-range reporting data structure that effectively combines the results from the trie queries. Techniques similar to this are used for several string problems [65].

Gagie et al. avoid storing the text in read-only memory by storing the text as an SLP and employ several other ideas to further speed up the query time. They keep the space low by using a compact trie with fast query time that uses only linear space in the $O(z)$ strings it stores [6].

We show how to improve the query time by exploiting combinatorial properties of the LZ77 parse that allow us to increase the number of strings stored in the tries. This results in a decrease in the number of trie searches we need to make. The details are numerous, but along the way we give a generalization of the compact trie data structure by Belazzougui et al. [6] supporting batched queries and show that random access can be done in $O(\lg(n/z))$ for a balanced SLP. Finally, we show how to combine this efficiently with range reporting and fast random-access in a balanced grammar leading to the results.

Results and Techniques in Chapter 3 We show that if the LZ77 parse of S consists of z phrases we can solve the compressed indexing problem in $O(m + \mathrm{occ}(\lg^\varepsilon z + \lg\lg n))$ time using $O(z\lg(n/z))$ space for constant-sized alphabets and $O(m+\mathrm{occ}(\lg^\varepsilon z+\lg\lg n))$ time using $O(z(\lg(n/z) + \lg\lg z))$ space for integer alphabets of size $n^{O(1)}$. Notably, we are the first to obtain a leading term of order m while using only $O(\lg(n/z))$ space.

The main component in our index is a new locally consistent grammar construction algorithm. It originates from Mehlhorn et al. [68] who did not use it for compression but instead used it to maintain a dynamic set of strings subject to equality testing. It comes in a deterministic and a randomized version with slightly different complexity profiles. We use the randomized construction algorithm but show that the non-determinism can be restricted to the preprocessing and we obtain new worst-case case complexities for the compressed indexing problem.

1.5 Chapter 4: Fast Lempel–Ziv Decompression in Linear Space

Given the LZ77 parse of a string S we consider the problem of decompressing substrings of S using as little working space as possible. This problem is important in settings with limited storage resources necessitating on-the-fly analysis of data.

The string S is trivially decompressed in optimal $O(n)$ time given its LZ77 parse but this technique requires read-only access to the entire string and therefore $O(n)$ working space. Alternatively, the parse can be converted into an SLP of size $O(z \lg(n/z))$ using $O(z \lg(n/z))$ time and space [18, 84]. With no additional working space, the SLP supports decompressing S in $O(n)$ time and also allows any length l substring of S to be decompressed in time $O(l + \lg(n/z))$ [10]. To the best of our knowledge, no attempts to solve this problem has been described in the literature with only one remotely relevant exception of Belazzougui et al. [9] who consider LZ77 decompression in the external memory model. They aim at minimizing the total number of I/Os required to decompress S and are not concerned with working space making the results incomparable.

Results and Techniques We provide two smooth time-space trade-offs that give several new complexities for decompressing S. For instance, we can extract S in $O(n)$ time and $O(z)$ space for constant-sized alphabets and for integer alphabets of size $n^{O(1)}$ we obtain the following:

- $O(n)$ time using $O(z \lg \sigma)$ space or,
- $O(n \lg \sigma)$ time using $O(z)$ space or,
- $O(n)$ time using $O(z \lg \lg n)$ space.

Both trade-offs also work for decompressing any specified set of s substrings of S with total length l which we solve in $O(l + (s+z) \lg n)$ time using $O(z)$ space for constant-sized alphabets or $O(z \lg \lg n)$ space for general alphabets.

Our results are obtained by exploiting combinatorial properties of the LZ77 parse along with novel combinations and extensions of techniques for grammar construction and the mergeable dictionary data structure by Iacono and Özkan [51]. We show how one can generally focus the decompression of any set of substrings to an essential set of small substrings that are all located around the phrase borders of the LZ77 parse. By breaking the query into carefully chosen subqueries that are handled in batches, we can efficiently map back and forth between the essential set of strings and the originally queried strings. We give different techniques for how to handle decompression of this essential set of strings, for instance, we show how to efficiently construct the SLP generating exactly these strings.

1.6 Chapter 5: Compressed Communication Complexity of Longest Common Prefixes

In Chapter 5 we consider communication protocols for the *Longest Common Prefix problem* and generalizations hereof.

The *longest common prefix problem* is to decide the longest common prefix (LCP) between two strings A and B, that is, the length of the maximal prefix shared by A

and B, i.e. an integer k such that $A[1, k] = B[1, k]$ while $A[1, k+1] \neq B[1, k+1]$. This problem is fundamental and has important applications both in practice (for instance in IP routing) and theory (for instance in compressed indexing or Lempel–Ziv compression).

We consider the *communication complexity* of the LCP problem in the setting where two parties Alice and Bob each hold a string A and B. Their goal is to determine the length of the longest common prefix of A and B using as few rounds and as few bits of communication as possible. The communication complexity of this problems is $\Theta(n)$ bits and $O(1)$ rounds for deterministic protocols and $\Theta(\lg n)$ bits and rounds for randomized protocols. We also consider the communication complexity of the related problem where Alice holds a single string A and Bob holds a set of k strings $B_1, \ldots, B_k$, and the goal it to compute the maximal longest common prefix between A and any of the strings $B_1, \ldots, B_k$.

The communication complexity model is often used to prove lower bounds and generally assumes that Alice and Bob has unbounded computational power. Upper bounds in this model usually serve to prove that some lower bound cannot be improved. Another relevant motivation for upper bounds in the communication complexity model is when the primary concern in solving a distributed computational problem is the amount of communication rather than the amount of computation done by the entities. This could apply to any client server scenario, but also applications such as cell-phone data transfer or long range radio communication. Such scenarios are typically characterized by limited bandwidth and availability on the communication channel due to contention and other factors or limited transmitting capacity due to factors such as power consumption.

Results and Techniques While the worst-case communication complexity for randomized protocols solving the LCP problem is $\Theta(\lg n)$ we show how to significantly reduce the number of rounds required when the longest common prefix of A and B is compressible without increasing the total communication. Our randomized protocol requires $O(\lg \ell)$ bits of communication and $O(\lg z)$ rounds where ℓ is the length of the longest common prefix. When Bob holds a set of k strings, we give a randomized protocol that finds the maximal longest common prefix between Alice's string and any of Bob's strings using $O(\lg z \lg k + \lg \ell)$ bits of communication and $O(\lg z)$ rounds.

These results hold for LZ77 parses without self-references but we also design protocols for the self-referential case. We obtain the results by exploiting the fact that the common prefix between two strings is represented by an almost identical prefix of the phrases in the LZ77 parse of the respective strings. This allows Alice and Bob to limit their search to prefixes that align with phrase borders rather than arbitrary prefixes. In order to keep the number of rounds low, we adapt a noisy binary search technique by Feige et al. [28] to our problem and generalise it to exponential search. Using this technique we can maintain the logarithmic complexity of binary and exponential search when comparisons may give a false answer with some fixed probability.

1.7 Chapter 6: Fast Dynamic Arrays

In Chapter 6 we consider a practical dynamic array data structure. The *dynamic array problem* is to maintain a sequence of elements subject to the 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 problem is one of the most fundamental dynamic data structure problems and is solved by textbook data structures such as arrays, linked lists and dynamic trees. These data structures are important primitives in the design of algorithms and data structures but the problem is also easily motivated by its direct applications. Consider for instance how important the problem of maintaining a dynamic sequence is when designing text editors or file systems.

Arrays provide an extreme in the trade-off between access/update time and insertion/deletion time namely constant time for access/update and linear time for insertion/deletion. Linked lists provide another alternative, and support constant time insertion and deletions given a pointer to the (neighboring) element. Without such a pointer, all operations run in linear time. Many dynamic trees provide a balanced complexity and support all operations in $O(\lg n)$ time, e.g. 2-3-4 trees, AVL trees, red-black trees.

The dynamic array problem is sometimes also referred to as *List Indexing* and is well studied both in terms of upper and lower bounds for both space and time complexities, see e.g. [15,25,35,36,46,57,58,82]. Fredman and Saks [36] show that $\Omega(\lg n/ \lg\lg n)$ is a lower bound when identical complexities are required for all operations. This lower bound is matched by Dietz [25] by a data structure that is essentially weight balanced B-tree with a $\lg^{\varepsilon} n$ fan-out.

An alternative data structure called *tiered vectors* is described by Goodrich and Kloos [46] based on similar ideas as Frederickson's [35] and provides an alternative complexity trade-off. It is a succinct data structure that supports access/update queries in $O(1)$ time and insertions/deletions in $O(n^{1/l})$ time for any constant integer $l \geq 2$.

A tiered vector is a tree with constant height $l - 1$ and a fan-out of $n^{1/l}$ where each leaf is a cyclic array storing some subsequence of the elements and internal vertices stores bookkeeping information.

Goodrich and Kloos [46] carry out experiments on an Java implementation of this data structure with $l = 2$ and compares it to the *vector* data structure from the standard library. Their results show that the tiered vector is competitive for access operations and significantly faster for insertions and deletions.

Results and Techniques We provide a general implementation of tiered vectors. Our implementation is in C++ and the height of the tree l is a compile time parameter. We believe our implementation is the first that supports more than 2 tiers. We carry out an extensive experimental performance comparison between relevant data structures from the standard library of C++ and our data structure with different choices for l along with several other optimizations. The optimizations provides trade-offs with respect to the speed of the different queries and the memory consumption of the data structure. Our comparisons are done on large sequences of 10^8 32-bit integers and our data structure offers significant speedups for access and update operations compared to the multiset data structure from the standard library and is only a few percent slower for insertions and deletions. In comparison with vector the speedups for insertions and deletions are several orders of magnitude while the access and update operations require less than double that of vector. Furthermore, the memory usage of our structure is a tenth of the memory used by multiset and only 1% more than the memory used by vector.

The results are obtained by carefully chosen optimizations, for instance, we manage to half the number of memory probes for access and update operations compared to the original tiered vector through a non-trivial memory layout of the tree structure.

1.8 Future Work

The time frame for the PhD studies is rigorously regulated to 3 years under Danish law. This means that any inconclusive research will have to be either postponed or adapted to some presentable format before the submission of the dissertation. Luckily, most of the work that I have done during my studies is now condensed to the papers that this dissertation is based on. However, there is still room for further work on the topics presented and luckily I've been given the opportunity to continue my research in a postdoctoral position. In the following, I list some of what I consider to be a natural continuation of the work presented here.

Compressed Indexing The compressed indexing problem is highly relevant in practice and the theoretical solutions consist of combinations of fundamental data structures for problems such as 2D-range reporting, prefix search along with insights in combinatorial properties of grammar-compression and LZ77-compression. I think there are several obvious directions to consider.

- Achieving $O(m)$ worst-case query time in $O(z \lg(n/z))$ space for alphabets of size $n^{O(1)}$ is an open problem. As seen in Table 1.1, recent research is approaching this complexity, but we are not quite there yet. Our progress in Chapter 2 is partly due to observations that some of the data structures by Gagie et al. [40] was using less space than the total space of their index. We then show how to exploit this and achieve a speed-up in query time, and therefore, it is not likely that this path can lead to further progress. In Chapter 3 we further improve these results and the main contribution is how to apply properties of the signature grammar to indexing. The use of the signature grammar for compression is relatively new [77]. Therefore, it is not unlikely that it could give rise to even further progress in relation to indexing.

- Indexing techniques that use $o(z \lg(n/z))$ space and have non-trivial time complexities are yet to be seen. Existing techniques usually rely on probabilistic data structures and then use grammars to do verification. Thus it seems like fundamentally different ideas are required. In Chapter 4 we solve the algorithmic compressed pattern existence problem in $O(z \lg m + m)$ space using $O(z \lg n + m)$ time and interestingly the $\lg m$ factor is logarithmic in the length of the pattern and independent of z and n. Here, the input is the pattern of length m and the LZ77 representation of a text of length n and the goal is to determine if the pattern occurs in the text. This problem is very different from the indexing problem where the goal is to build a data structure from the text that supports fast queries. However, it might be possible to adapt some of the techniques from the algorithmic problem to obtain a new trade-off.

LZ77 Decompression The mergeable dictionary by Iacono and Özkan [51] plays an important role in the techniques used in Chapter 4 where we consider how to decompress substrings of a string S given its LZ77 representation. A mergeable dictionary is a dynamic data structure that maintains a set of disjoint sets and supports merging and splitting of such sets. Each of these sets is maintained as biased skip lists and Iacono and Özkan show how to perform finger versions of the normal skip list operations such as join, split and search. Through an involved analysis, it is shown that the amortized time complexity for all operations is logarithmic in the size of the universe the elements are drawn from. Our application of the mergeable dictionary is limited to certain operations and there are clear bounds on the universe of each of the sets. Therefore, it would be interesting to consider this restricted version of the problem to see if it is possible to improve the complexity and also give a simpler analysis. Another obvious question is whether our results can be generalized to the self-referential LZ77 algorithm. I am currently working on this and is so far optimistic. It turns out that this problem also boils down to the analysis of the mergable dictionary which further supports the idea of designing a mergeable dictionary tailored for computations on LZ77 compressed text.

Signature Grammars Besides having several interesting properties relevant to indexing, the signature grammar can also be efficiently constructed online. Preliminary work indicates that we might be able to combine the indexing techniques of Chapter 3 based on the signature grammar with relevant dynamic data structures and thereby develop a technique providing new trade-offs for online construction of the LZ77 representation of a text.

Chapter 2

Time-Space Trade-Offs for Lempel–Ziv Compressed Indexing

Philip Bille Mikko Berggreen Ettienne Inge Li Gørtz Hjalte Wedel Vildhøj

The Technical University of Denmark

Abstract

Given a string S, the *compressed indexing problem* is to preprocess S into a compressed representation that supports fast *substring queries*. The goal is to use little space relative to the compressed size of S while supporting fast queries. We present a compressed index based on the Lempel–Ziv 1977 compression scheme. We obtain the following time-space trade-offs: For constant-sized alphabets

(i) $O(m + \mathrm{occ} \lg\lg n)$ time using $O(z \lg(n/z) \lg\lg z)$ space, or

(ii) $O(m(1 + \frac{\lg^\varepsilon z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^\varepsilon z))$ time using $O(z \lg(n/z))$ space,

For integer alphabets polynomially bounded by n

(iii) $O(m(1+\frac{\lg^\varepsilon z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^\varepsilon z))$ time using $O(z(\lg(n/z) + \lg\lg z))$ space, or

(iv) $O(m + \mathrm{occ}(\lg\lg n + \lg^\varepsilon z))$ time using $O(z(\lg(n/z) + \lg^\varepsilon z))$ space,

where n and m are the length of the input string and query string respectively, z is the number of phrases in the LZ77 parse of the input string, occ is the number of occurrences of the query in the input and $\varepsilon > 0$ is an arbitrarily small constant. In particular, (i) improves the leading term in the query time of the previous best solution from $O(m \lg m)$ to $O(m)$ at the cost of increasing the space by a factor $\lg\lg z$. Alternatively, (ii) matches the previous best space bound, but has a leading term in the query time of $O(m(1 + \frac{\lg^\varepsilon z}{\lg(n/z)}))$. However, for any polynomial compression ratio, i.e., $z = O(n^{1-\delta})$, for constant $\delta > 0$, this becomes $O(m)$. Our index also supports extraction of any substring of length ℓ in $O(\ell + \lg(n/z))$ time. Technically, our results are obtained by novel extensions and combinations of existing data structures of independent interest, including a new batched variant of weak prefix search.

2.1 Introduction

Given a string S, the *compressed indexing problem* is to preprocess S into a compressed representation that supports fast *substring queries*, that is, given a string P, report all occurrences of substrings in S that match P. Here the compressed representation can be any compression scheme or measure (kth order entropy, smallest grammar, Lempel–Ziv, etc.). The goal is to use little space relative to the compressed size of S while supporting fast queries. Compressed indexing is a key computational primitive for querying massive data sets and the area has received significant attention over the last decades with numerous theoretical and practical solutions, see e.g. [8, 21, 29–32, 40, 47–49, 54, 55, 60, 66, 67, 74] and the surveys [43, 72–74].

The Lempel–Ziv 1977 compression scheme (LZ77) [91] is a classic compression scheme based on replacing repetitions by references in a greedy left-to-right order. Numerous variants of LZ77 have been developed and several widely used implementations are available (such as `gzip` [1]). Recently, LZ77 has been shown to be particularly effective at handling highly-repetitive data sets [7, 20, 60, 67, 72] and LZ77 compression is always at least as powerful as any grammar representation [18, 84].

In this paper, we consider compressed indexing based on LZ77 compression. Relatively few results are known for this version of the problem. Let n, m, and z denote the length of the input string, the length of the pattern string, and the number of phrases in the LZ77 parse of the string (definition follows), respectively. Kärkkäinen and Ukkonen introduced the problem in 1996 [55] and gave an initial solution that required read-only access to the uncompressed text. Interestingly, this work is among the first results in compressed indexing [74]. More recently, Gagie et al. [39, 40] revisited the problem and gave a solution using space $O(z \lg(n/z))$ and query time $O(m \lg m + \mathrm{occ} \lg \lg n)$, where occ is the number of occurrences of P in S. Note that these bounds assume a constant-sized alphabet.

2.1.1 Our Results

We show the following main result.

Theorem 2.1 *We can build a compressed-index supporting substring queries in:*
For constant-sized alphabets

(i) $O(m + \mathrm{occ} \lg \lg n)$ *time using* $O(z \lg(n/z) \lg \lg z)$ *space, or*

(ii) $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}) + \mathrm{occ}(\lg \lg n + \lg^{\varepsilon} z))$ *time using* $O(z \lg(n/z))$ *space,*

For integer alphabets polynomially bounded by n

(iii) $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}) + \mathrm{occ}(\lg \lg n + \lg^{\varepsilon} z))$ *time using* $O(z(\lg(n/z) + \lg \lg z))$ *space, or*

(iv) $O(m + \mathrm{occ}(\lg \lg n + \lg^{\varepsilon} z))$ *time using* $O(z(\lg(n/z) + \lg^{\varepsilon} z))$ *space,*

where n *and* m *are the length of the input string and query string respectively,* z *is the number of phrases in the LZ77 parse of the input string,* occ *is the number of occurrences of the query in the input and* $\varepsilon > 0$ *is an arbitrarily small constant.*

Compared to the previous bounds Thm. 2.1 obtains new interesting trade-offs. In particular, Thm. 2.1 (i) improves the leading term in the query time of the previous best solution from $O(m \lg m)$ to $O(m)$ at the cost of increasing the space by only a factor $\lg \lg z$. Alternatively, Thm. 2.1 (ii) matches the previous best space bound, but has a leading term in the query time of $O(m(1 + \frac{\lg^{\varepsilon} z}{\lg(n/z)}))$. However, for any polynomial compression ratio, i.e., $z = O(n^{1-\delta})$, for constant $\delta > 0$, this becomes $O(m)$.

Gagie et al. [40] also showed how to extract an arbitrary substring of S of length ℓ in time $O(\ell + \lg n)$. We show how to support the same extraction operation and slightly improve the time to $O(\ell + \lg(n/z))$.

Technically, our results are obtained by new variants and extensions of existing data structures in novel combinations. In particular, we consider a batched variant of the *weak prefix search problem* and give the first non-trivial solution to it. We also generalize the well-known bidirectional compact trie search technique [65] to reduce the number of queries at the cost of increasing space. Finally, we show how to combine this efficiently with range reporting and fast random-access in a balanced grammar leading to the result.

We note that none of our data structures assume constant-sized alphabet and therefore, Thm. 2.1 is an instance of a full time-space trade-off for general alphabets. We discuss the details in Sec. 2.8.

2.2 Preliminaries

We assume a standard unit-cost RAM model with word size $w = \Theta(\lg n)$ and that the input is from an integer alphabet $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$ and measure space complexity in words unless otherwise specified.

A string S of length $n = |S|$ is a sequence $S[1] \cdots S[n]$ of n characters drawn from Σ. The string $S[i] \cdots S[j]$ denoted $S[i, j]$ is called a *substring* of S. ε is the empty string and $S[i, i] = S[i]$ while $S[i, j] = \varepsilon$ when $i > j$. The substrings $S[1, i]$ and $S[j, n]$ are the i^{th} *prefix* and the j^{th} *suffix* of S respectively. The reverse of the string S is denoted rev(S) $= S[n]S[n-1] \cdots S[1]$. We use the results from Fredman et al. [37] when referring to perfect hashing allowing us to build a dictionary on $O(k)$ integer keys in $O(k)$ expected time supporting constant time lookups.

2.2.1 Compact Tries

A *trie* for a set D of k strings is a rooted tree where the vertices correspond to the prefixes of the strings in D. str(v) denotes the prefix corresponding to the vertex v. str(v) $= \varepsilon$ if v is the root while v is the parent of u if str(v) is equal to str(u) without the last character. We may use v in place of str(v) when it is clear from the context that we talk about the str(v). This character is then the *label* of the edge from u to v. The *depth* of vertex v is the number of edges on the path from v to the root.

We assume each string in D is terminated by a special character $\$ \notin \Sigma$ such that each string in D corresponds to a leaf. The children of each vertex are sorted from left to right in increasing lexicographical order, and therefore the left to right order of the leaves corresponds to the lexicographical order of the strings in D. Let rank(s) denote the rank of the string $s \in D$ in this order.

A *compact trie* for D denoted T_D is obtained from the trie by removing all vertices v with exactly one child excluding the root and replacing the two edges incident to v with a single edge from its parent to its child. This edge is then labeled with the concatenation of the edge labels it replaces, thus the edges of a compact trie may be labeled by strings. The *skip interval* of a vertex $v \in \mathrm{T}_D$ with parent u is $(|\text{str}(u)|, |\text{str}(v)|]$ denoted skip(v) and skip(v) $= \emptyset$ if v is the root. The *locus* of a string s in T_D, denoted locus(s), is the minimum depth vertex v such that s is a prefix of str(v). If there is no such vertex, then locus(s) $= \perp$.

In order to reduce the space used by T_D we only store the first character of every edge and in every vertex v we store $|\text{str}(v)|$ (This variation is also known as a PATRICIA tree [70]). We navigate T_D by storing a dictionary in every internal vertex mapping the first character of the label of an edge to the respective child. The size of T_D is $O(k)$.

2.2.2 Karp–Rabin Fingerprints

A *Karp–Rabin fingerprinting function* [56] is a randomized hash function for strings. The fingerprint for a string S of length n is defined as:

$$\phi(S) = \sum_{i=1}^{n} S[i] \cdot r^{i-1} \bmod p$$

where p is a prime and r is a random integer in $\mathbb{Z}_p$ (the field of integers modulo p). Storing the values n, r^n mod p and r^{-n} mod p along with a fingerprint allows for efficient composition and subtraction of fingerprints:

Lemma 2.1 *Let x, y, z be strings such that $x = yz$. Given two of the three fingerprints $\phi(x), \phi(y)$ and $\phi(z)$, the third can be computed in constant time.*

It follows that we can compute and store the fingerprints of each of the prefixes of a string S of length n in $O(n)$ time and space such that we afterwards can compute the fingerprint of any substring $S[i, j]$ in constant time. We say that the fingerprints of the strings x and y *collide* when $\phi(x) = \phi(y)$ and $x \neq y$. A fingerprinting function ϕ is *collision-free* for a set of strings if there are no fingerprint collisions between any of the strings.

Lemma 2.2 *Let x and y be different strings of length at most n and let $p = \Theta(n^{2+\alpha})$ for some $\alpha > 0$. The probability that $\phi(x) = \phi(y)$ is $O(1/n^{1+\alpha})$.*

See e.g. [14, 80] for proofs of Lemma 2.1 and 2.2.

2.2.3 Range Reporting

Let $X \subseteq \{0, \ldots, u\}^d$ be a set of points in a d-dimensional grid. The *orthogonal range reporting problem* in d-dimensions is to compactly represent X while supporting *range reporting queries*, that is, given a rectangle $R = [a_1, b_2] \times \cdots \times [a_d, b_d]$ report all points in the set $R \cap X$. We use the following results for 2-dimensional range reporting:

Lemma 2.3 (Chan et al. [17]) *For any set of n points in $[0, u] \times [0, u]$ and $2 \leq B \leq \lg^{\varepsilon} n, 0 < \varepsilon < 1$ we can solve 2-d orthogonal range reporting with $O(n \lg n)$ expected preprocessing time, $O(n \lg_B \lg n)$ space and $(1 + k) \cdot O(B \lg \lg u)$ query time where k is the number of occurrences inside the rectangle.*

2.2.4 LZ77

The Ziv–Lempel algorithm from 1977 [91] provides a simple and natural way to compress strings.

The *LZ77 parse* of a string S of length n is a sequence Z of z subsequent substrings of S called *phrases* such that $S = Z[1]Z[2] \cdots Z[z]$. Z is constructed in a left to right pass of S: Assume that we have found the sequence $Z[1, i]$ producing the string $S[1, j-1]$ and let $S[j, j'-1]$ be the longest prefix of $S[j, n-1]$ that is also a substring of $S[1, j'-2]$. Then $Z[i+1] = S[j, j']$. The occurrence of $S[j, j'-1]$ in $S[1, j'-2]$ is called the *source* of the phrase $Z[i]$. Thus a phrase is composed by the contents of its possibly empty source and a trailing character which we call the *phrase border* and is typically represented as a triple $Z[i] = (start, len, c)$ where *start* is the starting position of the source, *len* is the length of the source and $c \in \Sigma$ is the border. For a phrase $Z[i] = S[j, j']$ we denote the position of its border by border($Z[i]$) $= j'$ and its source by source($Z[i]$) $= S[j, j'-1]$. For example, the string $abcabcabc \cdots abc$ of length n has the LZ77 parse $|a|b|c|abcabcabc \cdots abc|$ of length 4 which is represented as $Z = (0, 0, a)(0, 0, b)(0, 0, c)(1, n-4, c)$.

2.3 Prefix Search

The *prefix search* problem is to preprocess a set of strings such that later, we can find all the strings in the set that are prefixed by some query string. Belazzougui et al. [6] consider the *weak prefix search* problem, which is a relaxation of the prefix search problem. In this variant, we report only the ranks (in lexicographic order) of the strings that are prefixed by the query pattern and we are only required to answer correctly when at least one of the strings is prefixed by the pattern. Thus we may answer arbitrarily when no strings are prefixed by the query pattern.

Lemma 2.4 (Belazzougui et al. [6, appendix H.3]) *Given a set D of k strings with average length l, from an alphabet of size σ, 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.*

The term $m\lg\sigma/w$ stems from preprocessing P with an incremental hash function such that the hash of any substring $P[i,j]$ can be obtained in constant time afterwards. Therefore we can do weak prefix search for h substrings of P in $O(m\lg\sigma/w + h\lg m)$ time. We now describe a data structure that builds on the ideas from Lemma 2.4 but obtains the following:

Lemma 2.5 *Given a set D of k strings, we can build a data structure taking $O(k)$ space supporting weak prefix search for h substrings of a pattern P of length m in time $O(m + h(m/x + \lg x))$ where x is a positive integer.*

If we know h when building our data structure, we set x to h and obtain a query time of $O(m + h\lg h)$ with Lemma 2.5.

Before describing our data structure we need the following definition: The *2-fattest* number in a nonempty interval of strictly positive integers is the unique number in the interval whose binary representation has the highest number of trailing zeroes.

2.3.1 Data Structure

Let T_D be the compact trie representing the set D of k strings and let x be a positive integer. Denote by fat(v) the 2-fattest number in the skip interval of a vertex $v \in \mathrm{T}_D$. The *fat prefix* of v is the length fat(v) prefix of str(v). Denote by D^{fat} the set of fat prefixes induced by the vertices of T_D. The x-prefix of v is the shortest prefix of str(v) whose length is a multiple of x and is in the interval skip(v). If v's skip interval does not span a multiple of x, then v has no x-prefix. Let D^x be the set of x-prefixes induced by the vertices of T_D. The data structure is the compact trie T_D augmented with:

- A fingerprinting function ϕ.
- A dictionary $\mathcal{G}$ mapping the fingerprints of the strings in D^{fat} to their associated vertex.
- A dictionary $\mathcal{H}$ mapping the fingerprints of the strings in D^x to their associated vertex.
- For every vertex $v \in \mathrm{T}_D$ we store the rank in D of the string represented by the leftmost and rightmost leaf in the subtree of v, denoted l_v and r_v respectively.

The data structure is similar to the one by Belazzougui et al. [6] except for the dictionary $\mathcal{H}$, which we use in the first step of our search.

There are $O(k)$ strings in each of D^{fat} and D^x thus the total space of the data structure is $O(k)$.

Let i be the start of the skip interval of some vertex $v \in \mathrm{T}_D$ and define the *pseudo-fat* numbers of v to be the set of 2-fattest numbers in the intervals $[i,p]$ where $i \le p <$ fat(v). We use Lemma 2.2 to find a fingerprinting function ϕ that is collision-free for the strings in D^{fat} and all length l prefixes of the strings in D where l is either a pseudo-fat number in the skip interval of some vertex $v \in \mathrm{T}_D$ or a multiple of x.

Observe that the range of strings in D that are prefixed by some pattern P of length m is exactly $[l_v, r_v]$ where $v =$ locus(P). Answering a weak prefix search query for P is comprised by two independent steps. First step is to find a vertex $v \in \mathrm{T}_D$ such that str(v) is a prefix of P and $m - |\text{str}(v)| \leq x$. We say that v is in *x-range of P*. Next step is to apply a slightly modified version of the search technique from Belazzougui et al. [6] to find the *exit vertex* for P, that is, the deepest vertex $v_e \in \mathrm{T}_D$ such that str(v_e) is a prefix of P. Having found the exit vertex we can find the locus in constant time as it is either the exit vertex itself or one of its children.

2.3.2 Finding an x-range Vertex

We now describe how to find a vertex in x-range of P. If $m < x$ we simply report that the root of T_D is in x-range of P. Otherwise, let v be the root of T_D and for $i = 1, 2, \ldots \lfloor m/x \rfloor$ we check if $ix > |\text{str}(v)|$ and $\phi(P[1, ix])$ is in $\mathcal{H}$ in which case we update v to be the corresponding vertex. Finally, if $|\text{str}(v)| \geq m$ we report that v is locus(P) and otherwise we report that v is in x-range of P. In the former case, we report $[l_v, r_v]$ as the range of strings in D prefixed by P. In the latter case we pass on v to the next step of the algorithm.

We now show that the algorithm is correct when P prefixes a string in D. It is easy to verify that the x-prefix of v prefixes P at all time during the execution of the algorithm. Assume that $|\text{str}(v)| \geq m$ by the end of the algorithm. We will show that in that case $v = $ locus(P), i.e., that v is the highest vertex prefixed by P. Since P prefixes a string in D, the x-prefix of v prefixes P, and $|\text{str}(v)| \geq m$, then P prefixes v. Since the x-prefix of v prefixes P, P does not prefix the parent of v and thus v is the highest vertex prefixed by P.

Assume now that $|\text{str}(v)| < m$. We will show that v is in x-range of P. Since P prefixes a string in D and the x-prefix of v prefixes P, then str(v) prefixes P. Let $P[1, ix]$ be the x-prefix of v. Since v is returned, either $\phi(P[1, jx]) \notin \mathcal{H}$ or $jx \leq |\text{str}(v)|$ for all $i < j \leq \lfloor m/x \rfloor$. If $\phi(P[1, jx]) \notin \mathcal{H}$ then $P[1, jx]$ is not a x-prefix of any vertex in T_D. Since P prefixes a string in D this implies that jx is in the skip interval of v, i.e., $jx \leq |\text{str}(v)|$. This means that $jx \leq |\text{str}(v)|$ for all $i < j \leq \lfloor m/x \rfloor$. Therefore $\lfloor m/x \rfloor x \leq |\text{str}(v)| < m$ and it follows that $m - |\text{str}(v)| < x$. We already proved that str(v) prefixes P and therefore v is in x-range of P.

In case P does not prefix any string in D we either report that $v = $ locus(P) even though locus$(P) = \perp$ or report that v is in x-range of P because $m - |\text{str}(v)| \leq x$ even though str(v) is not a prefix of P due to fingerprint collisions. This may lead to a false positive. However, false positives are allowed in the weak prefix search problem.

Given that we can compute the fingerprint of substrings of P in constant time the algorithm uses $O(m/x)$ time.

2.3.3 From x-range to Exit Vertex

We now consider how to find the exit vertex of P hereafter denoted v_e. The algorithm is similar to the one presented in Belazzougui et al. [6] except that we support starting the search from not only the root, but from any ancestor of v_e.

Let v be any ancestor of v_e, let y be the smallest power of two greater than $m - |\text{str}(v)|$ and let z be the largest multiple of y no greater than $|\text{str}(v)|$. The search progresses by iteratively halving the search interval while using $\mathcal{G}$ to maintain a candidate for the exit vertex and to decide in which of the two halves to continue the search.

Let v_c be the candidate for the exit vertex and let l and r be the left and right boundary for our search interval. Initially $v_c = v$, $l = z$ and $r = z + 2y$. When $r - l = 1$, the search terminates and reports v_c. In each iteration, we consider the mid $b = (l + r)/2$ of the interval $[l, r]$ and update the interval to either $[b, r]$ or $[l, b]$. There are three cases:

1. b is out of bounds

   a) If $b > m$ set r to b.
   b) If $b \leq |\text{str}(v_c)|$ set l to b.

2. $P[1, b] \in D^{\text{fat}}$, let u be the corresponding vertex, i.e. $\mathcal{G}(\phi(P[1, b])) = u$.
   a) If $|\text{str}(u)| < m$, set v_c to u and l to b.
   b) If $|\text{str}(u)| \geq m$, report $u = \text{locus}(P)$ and terminate.

3. $P[1, b] \notin D^{\text{fat}}$ and thus $\phi(P[1, b])$ is not in $\mathcal{G}$, set r to b.

Observe that we are guaranteed that all fingerprint comparisons are collision-free in case P prefixes a string in D. This is because the length of the prefix fingerprints we consider are all either 2-fattest or pseudo-fat in the skip interval of locus(P) or one of its ancestors and we use a fingerprinting function that is collision-free for these strings.

Correctness We now show that the invariant $l \leq |\text{str}(v_c)| \leq |\text{str}(v_e)| < r$ is satisfied and that str(v_c) is a prefix of P before and after each iteration. After $O(\lg x)$ iterations $r - l = 1$ and thus $l = |\text{str}(v_e)| = |\text{str}(v_c)|$ and therefore $v_c = v_e$. Initially v_c is an ancestor of v_e and thus str(v_c) is a prefix of P, $l = z \leq |\text{str}(v_c)|$ and $r = z + 2y > m > |\text{str}(v_e)|$ so the invariant is true. Now assume that the invariant is true at the beginning of some iteration and consider the possible cases:

1. b is out of bounds
   a) $b > m$ then because $|\text{str}(v_e)| \leq m$, setting r to b preserves the invariant.
   b) $b \leq |\text{str}(v_c)|$ then setting l to b preserves the invariant.

2. $P[1, b] \in D^{\text{fat}}$, let $u = \mathcal{G}(\phi(P[1, b]))$.
   a) $|\text{str}(u)| < m$ then str(u) is a prefix of P and thus $b = \text{fat}(u) \leq |\text{str}(u)| \leq |\text{str}(v_e)|$ so setting l to b and v_c to u preserves the invariant.
   b) $|\text{str}(u)| \geq m$ yet $u = \mathcal{G}(\phi(P[1, b]))$. Then u is the locus of P.

3. $P[1, b] \notin D^{\text{fat}}$, and thus $\phi(P[1, b])$ is not in $\mathcal{G}$. As we are not in any of the out of bounds cases we have $|\text{str}(v_c)| < b < m$. Thus, either $b > |\text{str}(v_e)|$ and setting r to b preserves the invariant. Otherwise $b \leq |\text{str}(v_e)|$ and thus b must be in the skip interval of some vertex u on the path from v_c to v_e excluding v_c. But skip(u) is entirely included in (l, r) and because b is 2-fattest in (l, r)[1] it is also 2-fattest in skip(u). It follows that fat(u) $= b$ which contradicts $P[1, b] \notin D^{\text{fat}}$ and thus the invariant is preserved.

Thus if P prefixes a string in D we find either the exit vertex v_e or the locus of P. In the former case the locus of P is the child of v_e identified by the character $P[|\text{str}(v_e)| + 1]$. Having found the vertex $u = \text{locus}(P)$ we report $[l_u, r_u]$ as the range of strings in D prefixed by P. In case P does not prefix any strings in D, the fact that the fingerprint of a prefix of P match the fingerprint of some fat prefix in D^x does not guarantee equality of the strings. There are two possible consequences of this. Either the search successfully finds what it believes to be the locus of P even though locus(P) $= \perp$ in which case we report a false positive. Otherwise, there is no child identified by $P[|\text{str}(v_e)| + 1]$ in which case we can correctly report that no strings in D are prefixed by S, a true negative. Recall that false positives are allowed as we are considering the weak prefix search problem.

[1] If $b - a = 2^i$, $i > 0$ and a is a multiple of 2^{i-1} then the mid of the interval $(a + b)/2$ is 2-fattest in (a, b).

Complexity The size of the interval $[l, r]$ is halved in each iteration, thus we do at most $O(\lg(m - |\text{str}(v)|))$ iterations, where v is the vertex from which we start the search. If we use the technique from the previous section to find a starting vertex in x-range of P, we do $O(\lg x)$ iterations. Each iteration takes constant time. Note that if P does not prefix a string in D we may have fingerprint collisions and we may be given a starting vertex v such that str(v) does not prefix P. This can lead to a false positive, but we still have $m - |\text{str}(v)| \leq x$ and therefore the time complexity remains $O(\lg x)$.

2.3.4 Multiple Substrings

In order to answer weak prefix search queries for h substrings of a pattern P of length m, we first preprocess P in $O(m)$ time such that we can compute the fingerprint of any substring of P in constant time using Lemma 2.1. We can then answer a weak prefix search query for any substring of P in total time $O(m/x + \lg x)$ using the techniques described in the previous sections. The total time is therefore $O(m + h(m/x + \lg x))$.

2.4 Distinguishing Occurrences

The following sections describe our compressed-index consisting of three independent data structures. One that finds long primary occurrences, one that finds short primary occurrences and one that finds secondary occurrences.

Let Z be the LZ77 parse of length z representing the string S of length n. If $S[i, j]$ is a phrase of Z then any substring of $S[i, j - 1]$ is a *secondary substring* of S. These are the substrings of S that do not contain any phrase borders. On the other hand, a substring $S[i', j']$ is a *primary substring* of S when there is some phrase $S[i, j]$ where $i \leq i' \leq j \leq j'$, these are the substrings that contain one or more phrase borders. Any substring of S is either primary or 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* [55].

2.5 Long Primary Occurrences

For simplicity, we assume that the data structure given in Lemma 2.5 not only solves the weak prefix problem, but also answers correctly when the query pattern does not prefix any of the indexed strings. Later in Section 2.5.3 we will see how to lift this assumption. The following data structure and search algorithm is a variation of the classical bidirectional search technique for finding primary occurrences [55].

2.5.1 Data Structure

Let τ be a fixed positive integer parameter (its value will be determined later). For every phrase $S[i, j]$, we consider the strings $S[i, j + k], 0 \leq k < \tau$ *relevant substrings* of S unless there is some longer relevant substring ending at position $j + k$. If $S[i', j']$ is a relevant substring then the string $S[j' + 1, n]$ is the *associated suffix*. There are at most $z\tau$ relevant substrings of S and equally many associated suffixes. The primary index is comprised by the following:

- A prefix search data structure T_D on the set of reversed relevant substrings.
- A prefix search data structure $\mathrm{T}_{D'}$ on the set of associated suffixes.
- An orthogonal range reporting data structure R on the $z\tau \times z\tau$ grid. Consider a relevant substring $S[i, j]$. Let x denote the rank of rev$(S[i, j])$ in the lexicographical order of the reversed relevant substrings, let y denote the rank of its associated

suffix $S[j+1,n]$ in the lexicographical order of the associated suffixes. Then (x,y) is a point in R and along with it we store the pair (j,b), where b is the position of the rightmost phrase border contained in $S[i,j]$.

Note that every point (x,y) in R is induced by some relevant substring $S[i,j]$ and its associated suffix $S[j+1,n]$. If some prefix $P[1,k]$ is a suffix of $S[i,j]$ and the suffix $P[k+1,m]$ is a prefix of $S[j+1,n]$ then $S[j-k+1,j-k+m]$ is an occurrence of P and we can compute its exact location from k and j.

2.5.2 Searching

The data structure can be used to find the primary occurrences of a pattern P of length m when $m > \tau$. Consider the $O(m/\tau)$ prefix-suffix pairs $(P[1,i\tau], P[i\tau+1,m])$ for $i = 1,\dots,\lfloor m/\tau \rfloor$ and the pair $(P[1,m],\varepsilon)$ in case m is not a multiple of τ. For each such pair, we do a prefix search for rev($P[1,i\tau]$) and $P[i\tau+1,m]$ in T_D and $\mathrm{T}_{D'}$, respectively. If either of these two searches report no matches, we move on to the next pair. Otherwise, let $[l,r]$, $[l',r']$ be the ranges reported from the search in T_D and $\mathrm{T}_{D'}$ respectively. Now we do a range reporting query on R for the rectangle $[l,r]\times[l',r']$. For each point reported, let (j,b) be the pair stored with the point. We report $j-i\tau+1$ as the starting position of a primary occurrence of P in S.

Finally, in case m is not a multiple of τ, we need to also check the pair $(P[1,m],\varepsilon)$. We search for rev($P[1,m]$) in in T_D and ε in $\mathrm{T}_{D'}$. If the search for rev($P[1,m]$) reports no match we stop. Otherwise, we do a range reporting query as before. For each point reported, let (j,b) be the pair stored with the point. To check that the occurrence has not been reported before we do as follows. Let k be the smallest positive integer such that $j-m+k\tau > b$. Only if $k\tau > m$ we report $j-m+1$ as the starting position of a primary occurrence.

Correctness We claim that the reported occurrences are exactly the primary occurrences of P. We first prove that all primary occurrences are reported correctly. Let $P = S[i',j']$ be a primary occurrence. As it is a primary occurrence, there must be some phrase $S[i^*,j^*]$ such that $i^* \le i' \le j^* \le j'$. Let k be the smallest positive integer such that $i'+k\tau-1 \ge j^*$. There are two cases: $k\tau \le m$ and $k\tau > m$. If $k\tau \le m$ then $P[1,k\tau]$ is a suffix of the relevant substring ending at $i'+k\tau-1$. Such a relevant substring exists since $i'+k\tau-1 < j^*+\tau$. Thus its reverse rev($P[1,k\tau]$) prefixes a string s in D, while $P[k\tau+1,m]$ is a prefix of the associated suffix $S[i'+k\tau,n] \in D'$. Therefore, the respective ranks of s and $S[i'+k\tau,n]$ in D and D' are plotted as a point in R which stores the pair $(i'+k\tau-1,b)$. We will find this point when considering the prefix-suffix pair $(P[1,k\tau]$, $P[k\tau+1,m])$, and correctly report $(i'+k\tau-1)-k\tau+1 = i'$ as the starting position of a primary occurrence. If $k\tau > m$ then $P[1,m]$ is a suffix of the relevant substring ending in $i'+m-1$. Such a relevant substring exists since $i'+m-1 < i'+k\tau-1 < j^*+\tau$. Thus its reverse prefixes a string in D and trivially ε is a prefix of the associated suffix. It follows as before that the ranks are plotted as a point in R storing the pair $(i'+m-1,b)$ and that we find this point when considering the pair $(P[1,m],\varepsilon)$. When considering $(P[1,m],\varepsilon)$ we report $(i'+m-1)-m+1 = i'$ as the starting position of a primary occurrence if $k\tau > m$, and thus i' is correctly reported.

We now prove that all reported occurrences are in fact primary occurrences. Assume that we report $j-i\tau+1$ for some i and j as the starting position of a primary occurrence in the first part of the procedure. Then there exist strings $\mathrm{rev}(S[i',j])$ and $S[j+1,n]$ in D and D' respectively such that $S[i',j]$ is suffixed by $P[1,i\tau]$ and $S[j+1,n]$ is prefixed by $P[i\tau+1,m]$. Therefore $j-i\tau+1$ is the starting position of an occurrence of P. The string $S[i',j]$ is a relevant suffix and therefore there exists a border b in the interval $[j-\tau+1,j]$. Since $i \ge 1$ the occurrence contains the border b and it is therefore a primary occurrence.

If we report $j - m + 1$ for some j as the starting position of a primary occurrence in the second part of the procedure, then $\mathrm{rev}(P[1, m])$ is a prefix of a string rev($S[i', j]$) in D. It follows immediately that $j - m + 1$ is the starting point of an occurrence. Since $m > \tau$ we have $j - m + 1 < j - \tau + 1$, and by the definition of relevant substring there is a border in the interval $[j - \tau + 1, j]$. Therefore the occurrence contains the border and is primary.

Complexity We now consider the time complexity of the algorithm described. First we will argue that any primary occurrence is reported at most once and that the search finds at most two points in R identifying it. Let $S[i', j']$ be a primary occurrence reported when we considered the prefix-suffix pair $(P[1, k\tau],\ P[k\tau + 1, m])$ as in the proof of correctness. Recall that there is some phrase $S[i^*, j^*]$ such that $i^* \leq i' \leq j^* \leq j'$ and again let k be the smallest positive integer such that $i' + k\tau - 1 \geq j^*$. None of the pairs $(P[1, h\tau], P[h\tau + 1, m])$, where $1 \leq h < k$ will identify this occurrence as the reverse of $P[1, h\tau]$ does not prefix the reverse of any relevant substring when $i^* \leq i' \leq i' + h\tau - 1 < j^*$ which is true when $h < k$. None of the pairs $(P[1, h\tau], P[h\tau + 1, m])$, where $h > k$, will identify this occurrence. This is the case since $i' + h\tau - 1 > j^* + \tau - 1$ whenever $h > k$, and from the definition of relevant substrings it follows that if $S[i^*, j^*]$ is a phrase, $S[a, b]$ is a relevant substring and $a < i^*$, then $b < i^* + \tau - 1$. Thus there are no relevant substrings that end after $j^* + \tau - 1$ and start before $i' < j^*$. Therefore, only one of the pairs $(P[1, h\tau], P[h\tau + 1, m])$ for $h = 1, \dots \lfloor m/\tau \rfloor$ identifies the occurrence. If $(k + 1)\tau > m$ then we might also find the occurrence when considering the pair $(P[1, m], \varepsilon)$, but we do not report i' as $k\tau \leq m$.

After preprocessing P in $O(m)$ time, we can do the $O(m/\tau)$ prefix searches in total time $O(m + m/\tau(m/x + \lg x))$ where x is a positive integer by Lemma 2.5. Using the range reporting data structure by Chan et al. [17] each range reporting query takes $(1+k) \cdot O(B \lg\lg(z\tau))$ time where $2 \leq B \leq \lg^{\varepsilon}(z\tau)$ and k is the number of points reported. As each such point in one range reporting query corresponds to the identification of a unique primary occurrence of P, which happens at most twice for every occurrence we charge $O(kB \lg\lg(z\tau))$ to reporting the occurrences. The total time to find all primary occurrences is thus $O(m + \frac{m}{\tau}(\frac{m}{x} + \lg x + B \lg\lg(z\tau)) + \mathrm{occ}\ B \lg\lg(z\tau))$ where occ is the number of primary and secondary occurrences of P.

2.5.3 *Prefix Search Verification*

The prefix data structure from Lemma 2.5 gives no guarantees of correct answers when the query pattern does not prefix any of the indexed strings. If the prefix search gives false-positives, we may end up reporting occurrences of P that are not actually there. We show how to solve this problem after introducing a series of tools that we will need.

Straight Line Programs A *straight line program* (SLP) for a string S is a context-free grammar generating the single string S.

Lemma 2.6 (Rytter [84], Charikar et al. [18]) *Given an LZ77 parse Z of length z producing a string S of length n we can construct a SLP for S of size $O(z \lg(n/z))$ in time $O(z \lg(n/z))$.*

The construction from Rytter [84] produces a balanced grammar for every consecutive substring of length n/z of S after a preprocessing step transforms Z such that no compression element is longer than n/z. These grammars are then connected to form a single balanced grammar of height $O(\lg n)$ which immediately yields extraction of any substring $S[i, j]$ in time $O(\lg(n) + j - i)$. We give a simple solution to reduce this to $O(\lg(n/z) + j - i)$, that also supports computation of the fingerprint of a substring in $O(\lg(n/z))$ time.

Lemma 2.7 *Given an LZ77 parse Z of length z producing a string S of length n we can build a data structure that for any substring $S[i,j]$ can extract $S[i,j]$ in $O(\lg(n/z)+j-i)$ time and compute the fingerprint $\phi(S[i,j])$ in $O(\lg(n/z))$ time. The data structure uses $O(z\lg(n/z))$ space and $O(n)$ construction time.*

Proof Assume for simplicity that n is a multiple of z. We construct the SLP producing S from Z. Along with every non-terminal of the SLP we store the size and fingerprint of its expansion. Let $s_1, s_2, \dots s_z$ be consecutive length n/z substrings of S. We store the balanced grammar producing s_i along with the fingerprint $\phi(S[1,(i-1)n/z])$ at index i in a table A.

Now we can extract s_i in $O(n/z)$ time and any substring $s_i[j,k]$ in time $O(\lg(n/z)+k-j)$. Also, we can compute the fingerprint $\phi(s_i[j,k])$ in $O(\lg(n/z))$ time. We can easily do a constant time mapping from a position in S to the grammar in A producing the substring covering that position and the corresponding position inside the substring. But then any fingerprint $\phi(S[1,j])$ can be computed in time $O(\lg(n/z))$. Now consider a substring $S[i,j]$ that starts in s_k and ends in $s_l, k<l$. We extract $S[i,j]$ in $O(\lg(n/z)+j-i)$ time by extracting the appropriate suffix of s_k, all of s_m for $k<m<l$ and the appropriate prefix of s_l. Each of the fingerprints stored by the data structure can be computed in $O(1)$ time after preprocessing S in $O(n)$ time. Thus table A is filled in $O(z)$ time and by Lemma 2.6 the SLPs stored in A use a total of $O(z\lg(n/z))$ space and construction time.■

Verification of Fingerprints We need the following lemma for the verification.

Lemma 2.8 (Bille et al. [12]) *Given a string S of length n, we can find a fingerprinting function ϕ in $O(n\lg n)$ expected time such that*

$$\phi(S[i,i+2^l])=\phi(S[j,j+2^l]) \textit{ iff } S[i,i+2^l]=S[j,j+2^l] \textit{ for all } (i,j,l).$$

Verification Technique Our verification technique is identical to the one given by Gagie et al. [40] and involves a simple modification of the search for long primary occurrences. By using Lemma 2.7 instead of bookmarking [40] for extraction and fingerprinting and because we only need to verify $O(m/\tau)$ strings, the verification procedure takes $O(m+(m/\tau)\lg(n/z))$ time and uses $O(z\lg(n/z))$ space.

Consider the string S of length n that we wish to index and let Z be the $LZ77$ parse of S. The verification data structure is given by Lemma 2.7. Consider the prefix search data structure $\mathrm{T}_{D'}$ as given in Section 2.5.1 and let ϕ be the fingerprinting function used by the prefix search, the case for T_D is symmetric. We alter the search for primary occurrences such that it first does the $O(m/\tau)$ prefix searches, then verifies the results and discards false-positives before moving on to do the $O(m/\tau)$ range reporting queries on the verified results. We also modify ϕ using Lemma 2.8 to be collision-free for all substrings of the indexed strings which length is a power of two.

Let $Q_1, Q_2, \dots Q_j$ be all the suffixes of P for which the prefix search found a locus candidate, let the candidates be $v_1, v_2, \dots v_j \in \mathrm{T}_{D'}$ and let p_i be $\mathrm{str}(v_i)[1,|Q_i|]$. Assume that $|Q_i|<|Q_{i+1}|$, and let 2-suf(Q) and 2-pre(Q) denote the fingerprints using ϕ of the suffix and prefix respectively of length $2^{\lfloor\lg|Q|\rfloor}$ of some string Q. The verification progresses in iterations. Initially, let $a=1$, $b=2$ and for each iteration do as follows:

1. 2-suf(Q_a) $\neq$ 2-suf(p_a) or 2-pre(Q_a) $\neq$ 2-pre(p_a): Discard v_a and set $a=a+1$ and $b=b+1$.

2. 2-suf(Q_a) $=$ 2-suf(p_a) and 2-pre(Q_a) $=$ 2-pre(p_a), let $R=p_b[|p_b|-|p_a|+1,|p_b|]$.

   a) 2-suf(R) $=$ 2-suf(Q_a) and 2-pre(R) $=$ 2-pre(Q_a): set $a=a+1$ and $b=b+1$.

   b) 2-suf(R) $\neq$ 2-suf(Q_a) or 2-pre(R) $\neq$ 2-pre(Q_a): discard v_b and set $b=b+1$.

3. $b = j + 1$: If all vertices have been discarded, report no matches. Otherwise, let v_f be the last vertex in the sequence $v_1, \ldots, v_j$ that was not discarded. Report all non-discarded vertices v_i where $|p_i|$ is no longer than the longest common suffix of p_f and Q_f as verified and discard the rest.

Consider the correctness and complexity of the algorithm. In case 1, clearly, p_a does not match Q_a and thus v_a must be a false-positive. Now observe that because Q_i is a suffix of P, it is also a suffix of $Q_{i'}$ for any $i < i'$. Thus in case 2 (b), if R does not match Q_a then v_b must be a false-positive. In case 2 (a), both v_a and v_b may still be false-positives, yet by Lemma 2.8, p_a is a suffix of p_b because 2-suf$(p_a) =$ 2-suf(R) and 2-pre$(p_a) =$ 2-pre(R). Finally, in case 3, v_f is a true positive if and only if $p_f = Q_f$. But any other non-discarded vertex $v_i \neq v_f$ is also only a true positive if p_f and Q_f share a length $|p_i|$ suffix because p_i is a suffix of p_f and Q_i is a suffix of Q_f.

The algorithm does j iterations and fingerprints of substrings of P can be computed in constant time after $O(m)$ preprocessing. Every vertex $v \in \mathrm{T}_{D'}$ represents one or more substrings of S. If we store the starting index in S of one of these substrings in v when constructing $\mathrm{T}_{D'}$ we can compute the fingerprint of any substring str$(v)[i, j]$ by computing the fingerprint of $S[i'+i-1, i'+j-1]$ where i' is the starting index of one of the substrings of S that v represents. By Lemma 2.7, the fingerprint computations take $O(\lg(n/z))$ time, the longest common suffix of p_f and Q_f can be found in $O(m + \lg(n/z))$ time and because $j \leq m/\tau$ the total time complexity of the algorithm is $O(m + (m/\tau)\lg(n/z))$.

2.6 Short Primary Occurrences

We now describe a simple data structure that can find primary occurrences of P in time $O(m + \mathrm{occ})$ using space $O(z\tau)$ whenever $m \leq \tau$ where τ is a positive integer.

Let Z be the LZ77 parse of the string S of length n. Let $Z[i] = S[s_i, e_i]$ and define F to be the union of the strings $S[k, \min\{e_i + \tau - 1, n\}]$ where $\max\{1, s_i, e_i - \tau + 1\} \leq k \leq e_i$ for $i = 1, 2, \ldots z$. There are $O(z\tau)$ such strings, each of length $O(\tau)$ and they are all suffixes of the z length $2\tau - 1$ substrings of S starting $\tau - 1$ positions before each border position. We store these substrings along with the compact trie T_F over the strings in F. The edge labels of T_F are compactly represented by storing references into one of the substrings. Every leaf stores the starting positions in S of all the string it represents and the positions of the leftmost borders these strings contain.

The combined size of T_F and the substrings we store is $O(z\tau)$ and we simply search for P by navigating vertices using perfect hashing [37] and matching edge labels character by character. Now either locus$(P) = \perp$ in which case there are no primary occurrences of P in S; otherwise, locus$(P) = v$ for some vertex $v \in \mathrm{T}_F$ and thus every leaf in the subtree of v represents a substring of S that is prefixed by P. By using the indices stored with the leaves, we can determine the starting position for each occurrence and if it is primary or secondary. Because each of the strings in F start at different positions in S, we will only find an occurrence once. Also, it is easy to see that we will find all primary occurrences because of how the strings in F are chosen. It follows that the time complexity is $O(m + \mathrm{occ})$ where occ is the number of primary and secondary occurrences.

2.7 The Secondary Index

Let Z be the LZ77 parse of length z representing the string S of length n. We find the secondary occurrences by applying the most recent range reporting data structure by Chan et al. [17] to the technique described by Kärkkäinen and Ukkonen [55] which is inspired by the ideas of Farach and Thorup [27].

Let $o_1, \ldots o_{\mathrm{occ}}$ be the starting positions of the occurrences of P in S ordered increasingly. Assume that o_h is a secondary occurrence such that $P = S[o_h, o_h + m - 1]$. Then by definition, $S[o_h, o_h + m - 1]$ is a substring the prefix $S[i, j - 1]$ of some phrase $S[i, j]$ and there must be an occurrence of P in the source of that phrase. More precise, let $S[k, l] = S[i, j - 1]$ be the source of the phrase $S[i, j]$ then $o_{h'} = k + o_h - i$ is an occurrence of P for some $h' < h$. We say that $o_{h'}$, which may be primary or secondary, is the source occurrence of the secondary occurrence o_h given the LZ77 parse of S. Thus every secondary occurrence has a source occurrence. Note that it follows from the definition that no primary occurrence has a source occurrence.

We find the secondary occurrences as follows: Build a range reporting data structure Q on the $n \times n$ grid and if $S[i, j]$ is a phrase with source $S[i', j']$ we plot a point (i', j') and along with it we store the phrase start i.

Now for each primary occurrence o found by the primary index, we query Q for the rectangle $[0, o] \times [o + m - 1, n]$. The points returned are exactly the occurrences having o as source. For each point (x, y) and phrase start i reported, we report an occurrence $o' = i + o - x$ and recurse on o' to find all the occurrences having o' as source.

Because no primary occurrence have a source, while all secondary occurrences have a source, we will find exactly the secondary occurrences.

The range reporting structure Q is built using Lemma 2.3 with $B = 2$ and uses space $O(z \lg \lg z)$. Exactly one range reporting query is done for each primary and secondary occurrence each taking $O((1 + k) \lg \lg n)$ where k is the number of points reported. Each reported point identifies a secondary occurrence, so the total time is $O(\mathrm{occ} \lg \lg n)$.

2.8 The Compressed Index

We obtain our final index by combining the primary index, the verification data structure and the secondary index. We use a standard technique to guarantee that no phrase in the LZ77 parse is longer than n/z when building our primary index, see e.g. [18, 84]. Therefore any primary occurrence of P will have a prefix $P[1, k]$ where $k \leq n/z$ that is a suffix of some phrase. It then follows that we need only consider the multiples $(P[1, i\tau], P[i\tau + 1, m])$ for $i < \lfloor \frac{n/z}{\tau} \rfloor$ when searching for long primary occurrences. This yields the following complexities:

- $O(m + \frac{\min\{m, n/z\}}{\tau}(\frac{m}{x} + \lg x + B \lg \lg(z\tau)) + \mathrm{occ}\, B \lg \lg(z\tau))$ time and $O(z\tau \lg_B \lg(z\tau))$ space for the index finding long primary occurrences where x and τ are positive integers and $2 \leq B \leq \lg^{\varepsilon}(z\tau)$.
- $O(m + \mathrm{occ})$ time and $O(z\tau)$ space for the index finding short primary occurrences.
- $O(m + (m/\tau) \lg(n/z))$ time and $O(z \lg(n/z))$ space for the verification data structure.
- $O(\mathrm{occ} \lg \lg n)$ time and $O(z \lg \lg z)$ space for the secondary index.

If we fix x at n/z we have $\frac{\min\{m, n/z\}}{\tau} \frac{m}{x} \leq m$ in which case we obtain the following trade-off simply by combining the above complexities.

Theorem 2.2 *Given a string S of length n from an alphabet of size σ, we can build a compressed-index supporting substring queries in $O(m + \frac{m}{\tau}(\lg(n/z) + B \lg \lg(z\tau)) + \mathrm{occ}(B \lg \lg(z\tau) + \lg \lg n))$ time using $O(z(\lg(n/z) + \tau \lg_B \lg(z\tau) + \lg \lg z))$ space for any query pattern P of length m where $2 \leq B \leq \lg^{\varepsilon}(z\tau)$ and $0 < \varepsilon < 1$ are constants, τ is a positive integer, z is the number of phrases in the LZ77 parse of S and occ is the number of occurrences of P in S.*

We note that none of our data structures assume constant sized alphabet and thus Thm. 2.2 holds for any alphabet size.

2.8.1 Trade-offs

Thm. 2.2 gives rise to a series of interesting time-space trade-offs.

Corollary 2.1 *Given a string S of length n from an alphabet of size σ we can build a compressed-index supporting substring queries in*

(i) $O(m(1+\frac{\lg\lg z}{\lg(n/z)}) + \mathrm{occ}\lg\lg n)$ *time using* $O(z\lg(n/z)\lg\lg z)$ *space, or*

(ii) $O(m(1+\frac{\lg^{\varepsilon} z}{\lg(n/z)}) + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ *time using* $O(z(\lg(n/z) + \lg\lg z))$ *space, or*

(iii) $O(m + \mathrm{occ}\lg\lg n)$ *time using* $O(z(\lg(n/z)\lg\lg z + \lg^2\lg z))$ *space, or*

(iv) $O(m + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ *time using* $O(z(\lg(n/z) + \lg^{\varepsilon} z))$ *space.*

where $\varepsilon > 0$ *is an arbitrarily small constant.*

Proof For (i) set $B = 2$ and $\tau = \lg(n/z)$, for (ii) set $B = \lg^{\varepsilon} z$ and $\tau = \lg(n/z)$, for (iii) set $B = 2$ and $\tau = \lg(n/z) + \lg\lg z$, for (iv) set $B = \lg^{\varepsilon'} z$ and $\tau = \lg(n/z) + \lg^{\varepsilon} z$ where $\varepsilon' < \varepsilon$. ■

The leading term in the time complexity of Cor. 2.1 (i) is $O(m)$ whenever $\lg\lg(z) = O(\lg(n/z))$ which is true when $z = O(n/\lg n)$, i.e. for all strings that are compressible by at least a logarithmic fraction. For $\sigma = O(1)$ we have $z = O(n/\lg n)$ for all strings [74] and thus Thm. 2.1 (i) follows immediately. Cor. 2.1 (ii) matches previous best space bounds but obtains a leading term of $O(m)$ for any polynomial compression rate. Thm. 2.1 (ii) assumes constant-sized alphabet and therefore follows from (ii). Cor. 2.1 (iii) and (iv) show how to guarantee the fast query times with leading term $O(m)$ without the assumptions on compression ratio that (i) and (ii) require to match this, but at the cost of increased space. Thm. 2.1 (iii) is Cor. 2.1 (ii) and thm. 2.1 (iv) is Cor. 2.1 (iv).

2.8.2 Preprocessing

We now consider the preprocessing time of the data structure. Let Z be the LZ77 parse of the string S of length n let T_D and $\mathrm{T}_{D'}$ be the compact tries used in the index for long primary occurrences. The compact trie T_D indexes $O(z\tau)$ substrings of S with overall length $O(n\tau)$. Thus we can construct the trie in $O(n\tau)$ time by sorting the strings and successively inserting them in their sorted order [3]. The compact tries $\mathrm{T}_{D'}$ indexes $z\tau < n$ suffixes of S and can be built in $O(n)$ time using $O(n)$ space [26]. The index for short primary occurrences is a generalized suffix tree over z strings of length $O(\tau)$ with total length $z\tau < n$ and is therefore also built in $O(n)$ time. The dictionaries used by the prefix search data structures and for trie navigation contain $O(z\tau)$ keys and are built in expected linear time using perfect hashing [37]. The range reporting data structures used by the primary and secondary index over $O(z\tau)$ points are built in $O(z\tau\lg(z\tau))$ expected time using Lemma 2.3.

Building the SLP for our verification data structure takes $O(z\lg(n/z))$ time using Lemma 2.6 and finding an appropriate fingerprinting function ϕ takes $O(n\lg n)$ expected time using Lemma 2.8. The prefix search data structures T_D and $\mathrm{T}_{D'}$ also require that ϕ is collision-free for all prefixes whose length are either pseudo fat or multiples of x. There are at most $O(z\tau\lg n + n\tau/x)$ such prefixes [6]. If we compute these fingerprints incrementally while doing a traversal of the tries, we expect all the fingerprints to be unique. We simply check this by sorting the fingerprints in linear time and checking for duplicates by doing a linear scan. If we choose a prime $p = \Theta(n^5)$ with Lemma 2.2 then the probability of a collision between any two strings is $O(1/n^4)$ and by a union bound over the $O((n\lg n)^2)$ possible collisions the probability that ϕ is collision-free is at least $1 - 1/n$. Thus the expected time to find our required fingerprinting function is $O(n + n \lg n)$.

All in all, the preprocessing time for our combined index is therefore expected $O(n \lg n + n\tau)$.

Chapter 3

Compressed Indexing with Signature Grammars

Anders Roy Christiansen Mikko Berggreen Ettienne

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 $O(m + \mathrm{occ}(\lg\lg n + \lg^{\varepsilon} z))$ time using $O(z \lg(n/z))$ space where z is the size of the LZ77 parse of S and $\varepsilon > 0$ is an arbitrarily small constant, when the alphabet is small or $z = O(n^{1-\delta})$ for any constant $\delta > 0$. 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. These 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. [10, 39, 40, 42, 55, 74, 77]. This problem is highly relevant as the amount of highly-repetitive data increases rapidly, and thus it is possible

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, occ is the number of occurrences and σ is the size of the alphabet. (The time complexity marked by † is expected whereas all others are worst-case)

Index	Space	Locate time	σ
Gagie et al. [40]	$O(z \lg(n/z))$	$O(m \lg m + \mathrm{occ} \lg\lg n)$	$O(1)$
Nishimoto et al. [77]	$O(z \lg n \lg^* n)$	$O(m \lg\lg n \lg\lg z + \lg z \lg m \lg n (\lg^* n)^2 + \mathrm{occ} \lg n)$	$n^{O(1)}$
Bille et al. [10]	$O(z(\lg(n/z) + \lg^\varepsilon z))$	$O(m + \mathrm{occ}(\lg^\varepsilon n + \lg\lg n))$	$n^{O(1)}$
Bille et al. [10]	$O(z \lg(n/z) \lg\lg z)$	$O(m + \mathrm{occ} \lg\lg n)$	$O(1)$
Bille et al. [10]	$O(z \lg(n/z))$	$O(m(1 + \frac{\lg^\varepsilon z}{\lg(n/z)}) + \mathrm{occ}(\lg^\varepsilon n + \lg\lg n))$	$O(1)$
Theorem 1	$O(z \lg(n/z))$	$O(m + \mathrm{occ}(\lg^\varepsilon z + \lg\lg n))$	$O(1)$
Theorem 2 (1)	$O(z(\lg(n/z) + \lg\lg z))$	$O(m + \mathrm{occ}(\lg^\varepsilon z + \lg\lg n))$	$n^{O(1)}$
Theorem 2 (2)	$O(z(\lg(n/z))$	$O(m + \mathrm{occ}(\lg^\varepsilon z + \lg\lg n))^\dagger$	$n^{O(1)}$

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 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 $O(m + \mathrm{occ}(\lg\lg n + \lg^\varepsilon z))$ 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:

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) $O(m + \mathrm{occ}(\lg\lg n + \lg^\varepsilon z))$ time using $O(z(\lg(n/z) + \lg\lg z))$ space.

(2) $O(m + \mathrm{occ}(\lg\lg n + \lg^\varepsilon z))$ expected time using $O(z \lg(n/z))$ space.

(3) $O(m + \lg^\varepsilon z + \mathrm{occ}(\lg\lg n + \lg^\varepsilon z))$ time using $O(z \lg(n/z))$ space.

Note $\lg\lg z = O(\lg(n/z))$ when either the alphabet size is $O(2^{\lg^\varepsilon n})$ or $z = o(\frac{n}{\lg^{\varepsilon'} n})$ where ε and ε' are arbitrarily small positive constants. Theorem 3.1 follows directly from Theorem 3.2 (1) given these observations. Theorem 3.2 is a consequence of Lemma 3.9, 3.11, 3.12 and 3.13.

3.1.2 Technical Overview

Our main new contribution is based on a new grammar construction. In [68] 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 [23] 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 [85]. To the best of our knowledge, we are the first to consider grammar compression based on the randomized solution from [68]. 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. [2, 77]. In [52] Jeż devices 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 [45, 89]

Our primary solution has an $\lg^{\varepsilon} z$ 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 $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$. 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 $S[1] \ldots S[n]$ drawn from an alphabet Σ. The sequence $S[i, j]$ is the *substring* of S given by $S[i] \ldots S[j]$ and strings can be concatenated, i.e. $S = S[1, k]S[k + 1, n]$. The empty string is denoted ε and $S[i, i] = S[i]$ while $S[i, j] = \varepsilon$ 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 $S[n]S[n - 1] \ldots S[1]$. A *run* in a string S is a substring $S[i, j]$ with identical letters, i.e. $S[k] = S[k + 1]$ for $k = i, \ldots, j - 1$. 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 $\{1, 2, \ldots, u\}$.

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 3.1 (Chan et al. [17]) *For any set of n points in $[u] \times [u]$ and constant $\varepsilon > 0$, we can solve 2D-orthogonal range reporting with $O(n \lg n)$ expected preprocessing time using:*

i $O(n)$ space and $(1 + k) \cdot O(\lg^{\varepsilon} n \lg \lg u)$ 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* [56] is a randomized hash function for strings. Given a string S of length n and a fingerprinting function ϕ 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 [80].

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 3.2 (Belazzougui et al. [6], appendix H.3) *Given a set D of k strings with average length l, from an alphabet of size σ, 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 3.2 as a *z-fast trie* following the notation from [6]. 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* [91] 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^{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 [91]. 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. [68] with some slight modifications. Let S be a run-free string of length n from an integer alphabet Σ and let π be a uniformly random permutation of Σ. 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 $sig(S)$ which is an ordered labeled tree with several useful properties.

Lemma 3.3 *Let S be a run-free string S of length n from an alphabet Σ and let π be a uniformly random permutation of Σ 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 π is a uniformly random permutation of Σ 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 [68]. ∎

3.3.1 Signature Grammar Construction

We now give the construction algorithm for the signature tree $sig(S)$. Consider an ordered forest F of trees. Initially, F consists of n trees where the i^{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 $sig(S)$ proceeds as follows:

1. Let $t_i, \ldots, t_j$ be a maximal subrange of consecutive trees of F with identical labels, i.e. $l(t_i) = \ldots = l(t_j)$. Replace each such subrange in F by a new tree having as root a new node v with children $t_i, \ldots, t_j$ 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 $t_i, \ldots, t_j$ be consecutive trees in F such that their labels form a block in $l(F)$. Replace all identical blocks $t_i, \ldots, t_j$ by a new tree having as root a new node with children $t_i, \ldots, t_j$ 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 $sig(S)$.

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 $sig(S)$ where all identical subtrees are merged. Note we can store run nodes in $O(1)$ space since all out-going 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 $dag(S)$. 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 $sig(S)$ and $dag(S)$ that we will need later. Let $str(v)$ denote the substring of S given by the labels of the leaves of the subtree of $sig(S)$ induced by the node v in left to right order.

Lemma 3.4 *Let v be a node in the signature tree for a string S of length n. If v has height h then $|str(v)|$ is at least 2^h and thus $sig(S)$ (and $dag(S)$) 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 $sig(S)$ that are ancestors of the i^{th} through j^{th} leaf of $sig(S)$. These nodes form a sequence of adjacent nodes at every level of $sig(S)$ and we call them *relevant nodes* for the substring $S[i,j]$.

Lemma 3.5 *$T(i,j)$ 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 $v_1, v_2, v_3, \ldots$ be the nodes on level $l-1$ in $T(i,j)$ and $u_1, u_2, u_3, \ldots$ the nodes on level $l-1$ in $T(i',j')$ in left to right order. From the assumption, we have $v_a, v_{a+1}, \ldots$ are identical with $u_b, u_{b+1}, \ldots$ 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 $v_1, \ldots, v_{a+k}$ are spanned by at most two blocks and similarly for $u_1, \ldots, u_{b+k}$. 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 3.6 *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 $S[u_1, u_1 + l_1], S[u_1 + l_1 + 1], \ldots, S[u_z, u_z + l_z], S[u_z + l_z + 1]$ given by the phrases and borders of the LZ77-parse of S and the corresponding sets of relevant nodes $R = \{T(u_1, u_1 + l_1), T(u_1 + l_1 + 1, u_1 + l_1 + 1), \ldots\}$. 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 3.5 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 3.3 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 3.7 *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 $dag(S)$ 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 3.8 *Given a node $v \in dag(S)$, the child that produces the character at position i in $str(v)$ can be found in $O(1)$ time.*

Proof First assume v is a run-free node. If we store $|str(u)|$ 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 $O(\lg n)$ time since this is the maximum out-degree of a node in $dag(S)$. This can be improved to $O(\lg\lg n)$ by doing a binary search, but instead we use a Fusion Tree from [38] that allows us to do this in $O(1)$ time since we have at most $O(\lg n)$ 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 $\Omega(\lg^{\varepsilon} z)$ for constant $\varepsilon > 0$. We obtain the following lemma:

Lemma 3.9 *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 $O(m + (1 + \mathrm{occ})\lg^{\varepsilon} z)$ time using $O(z\lg(n/z))$ space for any constant $\varepsilon > 0$.*

3.4.1 Data Structure

Consider a vertex v with children $u_1, \ldots u_k$ in $dag(S)$. Let $pre(v,i)$ denote the prefix of $str(v)$ given by concatenating the strings represented by the first i children of v and let $suf(v,i)$ be the suffix of $str(v)$ 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 3.2) T_1 and T_2 and a 2D-range reporting data structure R.

For every non-leaf node $v \in dag(S)$ 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 3.2. There $O(z\lg(n/z))$ points in R which takes $O(z\lg(n/z))$ space using Lemma 3.1 (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 $\phi(P[1]), \phi(P[1,2]), \ldots, \phi(P[1,m])$. 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 $P_L = \{|str(l_1^1)|, |str(l_1^1)| + |str(l_1^2)|, |str(l_2^1)|, |str(l_2^1)| + |str(l_2^2)|, \ldots, |str(l_j^1)|, |str(l_j^1)| + |str(l_j^2)|\}$. Symmetrically, let r_i^k denote the k'th right-most vertex on level i in $sig(P)$ and let $P_R = \{m - |str(r_1^1)|, m - |str(r_1^1)| - |str(r_1^2)|, m - |str(r_2^1)|, m - |str(r_2^1)| - |str(r_2^2)|, \ldots, m - |str(r_j^1)|, m - |str(r_j^1)| - |str(r_j^2)|\}$. 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 $k = 1, \ldots, j$ 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. [40] 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 [40] 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 $S[l, r]$, ie. 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 $1, \ldots, m - 1$ of P i.e. $P[1, i]$ and $P[i + 1, m]$ for $i = 1, \ldots, m - 1$.

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'_{ll} 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'_{ll} 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 3.2 while a query on R takes $O(\lg^{\varepsilon} z)$ time using Lemma 3.1 (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 $O(m + \lg m \lg^{\varepsilon} z) = O(m + \lg^{\varepsilon'} z)$ for any $\varepsilon' > \varepsilon$. This holds because if $m \leq \lg^{2\varepsilon} z$ then $\lg m \lg^{\varepsilon} z \leq \lg\lg^{2\varepsilon} z \lg^{\varepsilon} z = O(\lg^{\varepsilon'} z)$, otherwise $m > \lg^{2\varepsilon} z \Leftrightarrow \sqrt{m} > \lg^{\varepsilon} z$ and then $\lg m \lg^{\varepsilon} z = O(\lg m \sqrt{m}) = O(m)$. For every query on R we may find multiple points each corresponding to an occurrence of P. It takes $O(\lg^{\varepsilon} z)$ time to report each point thus the total time becomes $O(m + (1 + \mathrm{occ}) \lg^{\varepsilon'} z)$.

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 [55] Kärkkäinen and Ukkonen use some observations by Farach and Thorup [27] 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 3.1 (ii), all secondary occurrences can be reported as stated in the following lemma:

Lemma 3.10 (Kärkkäinen and Ukkonen [55]) *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 $O(\mathrm{occ} \lg\lg n)$ time.*

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

Lemma 3.11 *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 $O(m + \mathrm{occ} \lg\lg n)$ 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 $O(m + \mathrm{occ})$ time (where occ includes secondary occurrences). Combined with Lemma 3.10 this gives Lemma 3.11. ■

3.6 Semi-Short Patterns

In this section, we show how to handle patterns of length between $\lg\lg z$ and $\lg^{\varepsilon} z$. 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 3.1 (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 3.12 *Given a string S of length n we solve the compressed indexing problem for a pattern P of length m with $\lg\lg z \leq m \leq \lg^{\varepsilon} z$ for any positive constant $\varepsilon < \frac{1}{2}$ in $O(m + \text{occ}(\lg\lg n + \lg^{\varepsilon} z))$ time using $O(z(\lg\lg z + \lg(n/z)))$ 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 $\lg^{\varepsilon} z$ 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 $|B| = O(z \lg^{2\varepsilon} z)$. 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 $D_z = \varepsilon$ and c is the lexicographical rank of the reverse of C_i in the set $\{C_1, \ldots, C_z\}$ and d is the lexicographical rank of D_i in the set $\{D_1, \ldots D_z\}$. We store the border u_i along with the point (c, d).
- The data structure described in Lemma 3.10 to report secondary occurrences.
- The signature grammar for S.

Each entry in H requires $\lg\lg^{\varepsilon} z = O(\lg\lg z)$ bits to store since a split position can be at most $\lg^{\varepsilon} z$. Thus the dictionary can be stored in $O(|B| \cdot \lg\lg z) = O(z \lg^{2\varepsilon} z \lg\lg z)$ bits which for $\varepsilon < \frac{1}{2}$ 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 $O(z \lg(n/z))$. Thus the total space is $O(z(\lg\lg z + \lg(n/z)))$.

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 3.10 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 $O(\mathrm{occ} \lg\lg z)$ 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 $O(\mathrm{occ} \lg\lg z)$ time. Finally, we verified one of the found occurrence against P in $O(m)$ time. So the total time is $O(m + \mathrm{occ} \lg\lg z)$ in this case.

In the case P does not exists, 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 $m \leq \lg^\varepsilon z$.

3.7 Randomized Solution

In this section we present a very simple way to turn the $O(m + (1 + \mathrm{occ}) \lg^\varepsilon z)$ worst-case time of Lemma 3.9 into $O(m + \mathrm{occ} \lg^\varepsilon z)$ expected time. First observe, this is already true if the pattern we search for occurs at least once or if $m \geq \lg^\varepsilon z$.

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

Now to answer a query, we first check if $m \leq \lg^\varepsilon z$ in which case we look it up in H. If it does not exist, report that. If it does exist in H or if $m > \lg^\varepsilon z$ use the solution from Lemma 3.9 to answer the query.

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

Lemma 3.13 *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 $O(m + \mathrm{occ} \lg^\varepsilon z)$ expected time using $O(z \lg(n/z))$ space for any constant $0 < \varepsilon \leq 1/3$.*

Chapter 4

Fast Lempel-Ziv Decompression in Linear Space

Philip Bille* Mikko Berggreen Ettienne* Travis Gagie† Inge Li Gørtz* Nicola Prezza*

* The Technical University of Denmark
† EIT, Diego Portales University, Chile

Abstract

We consider the problem of decompressing the Lempel–Ziv 77 representation of a string S of length n using a working space as close as possible to the size z of the input. The folklore solution for the problem runs in $O(n)$ time but requires random access to the whole decompressed text. A better solution is to convert LZ77 into a grammar of size $O(z\lg(n/z))$ and then stream S in linear time. In this paper, we show that $O(n)$ time and $O(z)$ working space can be achieved for constant-size alphabets. On larger alphabets, we describe (i) a trade-off achieving $O(n\lg^{\delta}\sigma)$ time and $O(z\lg^{1-\delta}\sigma)$ space for any $0\leq\delta\leq 1$ where σ is the size of the alphabet, and (ii) a solution achieving $O(n)$ time and $O(z\lg\lg n)$ space. Our solutions can, more generally, extract any specified subsequence of S with little overheads on top of the linear running time and working space. As an immediate corollary, we show that our techniques yield improved results for pattern matching problems on LZ77-compressed text.

4.1 Introduction

In this paper we consider the following problem: given an LZ77 representation of a string S of length n, decompress S and output it as a stream in left-to-right order (without storing it explicitly). Our goal is to solve this problem in as little space as possible, i.e. close to the size z of the compressed input string, and as fast as possible. In this respect, note that at least $\Omega(z)$ time is needed to read the input. This problem is fundamental and of great relevance in domains characterized by the production of huge amounts of repetitive data, where information has to be analyzed on-the-fly due to limitations in storage resources.

The folklore solution for the Lempel-Ziv decompression problem achieves linear time, but requires random access to the whole string. A better solution is to convert LZ77 into a straight-line program (i.e. a context-free grammar generating the text) of size $O(z\lg(n/z))$. This conversion can be performed in $O(z\lg(n/z))$ space and time [18, 84]. Then, the entire text can be decompressed and streamed in linear time using just the space of the grammar. The problem has also been recently considered in [9] in the context of external-memory algorithms. To the best of our knowledge, no other attempts to solve the problem have been described in the literature. In particular, no solutions using $O(z)$ space are known.

4.1.1 Our contributions

The main contribution of this paper is to show that LZ77 decompression can be performed in linear space (in the compressed input's size) and near-optimal time (i.e. almost linear in the length of the extracted string). We provide two smooth space-time trade-offs which enable us to achieve either linear time *or* linear space *or* both if the alphabet's size is constant. The first trade-off is particularly appealing on small alphabets, while the second dominates the first on large alphabets.

Our solution even works for decompressing any specified subsequence of S with little overheads on top of the linear running time and working space. As an application, we show that our techniques yield improved results for pattern matching problems on LZ77-compressed text.

We formalize the LZ77 decompression problem as follows. The input consists of an LZ77 representation of a text and a list of text substrings encoded as pairs: $(i_1, j_1), \ldots, (i_s, j_s)$. We decompress these substrings and output them (e.g. to a stream or to disk) character-by-character in the order $S[i_1, j_1], \ldots, S[i_s, j_s]$. Since both the input strings and the output can be streamed (for example, from/to disk) we only count the working space used on top of the input and the output. Let the quantity $l = \sum_{k=1}^{s}(j_k - i_k + 1)$ denote the total number of characters to be extracted. Our main results are summarized in the following two theorems:

Theorem 4.1 *Let S be a string of length n from an alphabet of size σ compressed into an LZ77 representation with z phrases. For any parameter $0 \le \delta \le 1$, we can decompress any s substrings of S with total length l in $O(l \lg^{\delta} \sigma + (s+z) \lg n)$ time using $O(z \lg^{1-\delta} \sigma)$ space.*

Theorem 4.2 *Let S be a string of length n compressed into an LZ77 representation with z phrases. For any parameter $1 \le \tau \le \lg n$, we can decompress any s substrings of S with total length l in $O\left(l + \frac{l \lg n}{\tau} + (s+z)\lg n\right)$ time using $O(z \lg \tau)$ space.*

Theorems 4.1 and 4.2 lead to a series of new and non-trivial bounds on different algorithmic problems on LZ77. For instance, we provide a smooth time-space trade-off for decompressing S in $O(n \lg^{\delta} \sigma)$ time using $O(z \lg^{1-\delta} \sigma)$ space for any constant $0 \le \delta \le 1$. This gives the linear time and $O(z)$ space for constant-sized alphabets. By combining Theorem 4.2 with $\tau = \lg n$ with the technique based on grammars, we show how to decompress S in $O(n)$ time using $O(z \lg \lg n)$ space. Both bounds are strict improvements over the previous best complexity of $O(n)$ time and $O(z \lg(n/z))$ space. See Section 4.4 and Corollaries 4.2 and 4.3 for details.

Our results also imply new trade-offs for the pattern matching and approximate pattern matching problems on LZ77-compressed texts. By showing how our techniques can be combined with existing pattern matching results, we obtain the following:

Theorem 4.3 *Let S be a string of length n compressed into an LZ77 representation $\mathcal{Z}$ with z phrases, let P be a pattern of length m and let $\mathcal{A}$ be an algorithm that can detect an (approximate) occurrence of P in S (with at most k errors) given P and $\mathcal{Z}$ in $t(z,n,m,k)$ time and $s(z,n,m,k)$ space. Then, we can solve the same task in $O(t(z,zm,m,k) + z \lg n)$ time and $O(s(z,zm,m,k) + z)$ space. If $\mathcal{A}$ reports all occ occurrences using $t(z,n,m,k)$ time and $s(z,n,m,k)$ space, then we can report all occurrences in $O(t(z,zm,m,k) + z \lg n + \mathrm{occ})$ time and $O(s(z,zm,m,k) + z + \mathrm{occ})$ space.*

Theorem 4.4 *Let $\mathcal{A}$ be a streaming algorithm that reports all occ (approximate) occurrence of a pattern $P \in [\sigma]^m$ (with at most k errors) in a stream of length n in $t(n, m, k)$ time and $s(n, m, k)$ space. Then, we can report all occurrences of P in the LZ77 representation of a string $S \in [\sigma]^n$ in either:*

- *$O(t(zm, m, k) + z \lg n)$ time and $O(s(zm, m, k) + z \lg \lg n + \mathrm{occ})$ space or*
- *$O(t(zm, m, k) + z \lg n + zm \lg^\delta \sigma)$ time and $O(s(zm, m, k) + z \lg^{1-\delta} \sigma + \mathrm{occ})$ space.*

The best known algorithm for detecting if pattern P occurs in a string S given P and $\mathcal{Z}$ uses $O(z \lg(n/z) + m)$ time and $O(z \lg n + m)$ space [44]. If we plug this into Theorem 4.3 we obtain $O(z \lg n + m)$ time and $O(z \lg m + m)$ space thereby reducing the $\lg n$ factor in the space to $\lg m$ at the cost of slightly increasing the time.

We also obtain new trade-offs for reporting all approximate occurrences of P with at most k errors. For example if we plug in the Landau–Vishkin and Cole–Hariharan [22, 62] algorithms, we can solve the problem in $O(z \lg n + z \min\{mk, k^4 + m\} + \mathrm{occ})$ time using $O(z + m + \mathrm{occ})$ space for constant-sized alphabets or $O(z \lg \lg n + m + \mathrm{occ})$ space for general alphabets. The previous best solution has the same time complexity but uses $O(z \lg n + m + \mathrm{occ})$ space [41]. We refer to Section 4.5 for more details.

To obtain our results we need mergeable dictionaries with shift operations. We show how to extend the mergeable dictionary by Iacono & Özkan [51] to support shifts. (Iacono & Özkan [51] write that their data structure can be extended to support the shift operation but do not provide any details.)

4.1.2 Related work

While the LZ77 decompression problem has not been studied much in the literature, the problem of fast LZ77 *compression* in small working space has lately attracted a lot of research in the field of compressed computation [33, 34, 77–79].

A closely related problem is the *random access problem*, where the aim is to build a data structure taking space as close as possible to $O(z)$ words and supporting efficient access queries to single characters. Existing solutions for the random access problem [11, 18, 84] need $\Omega(z \lg(n/z))$ space to achieve $O(\lg(n/z))$ access time.

Because these data structures can be built efficiently they also solve the LZ77 decompression problem considered in this paper. In particular they can decompress the entire string S given its LZ77 representation in $O(n)$ time using $O(z \lg(n/z))$ working space. Our results improve this working space bound as we show how to decompress S in $O(n)$ time using only $O(z \lg \lg n)$ working space for general alphabets and $O(z)$ working space for constant-sized alphabets.

The random access data structures can also decompress any set of s substrings with total length l in $O(l + s \lg n)$ time. We provide several new trade-offs for this problem; for instance we can solve it using $O(l + (z + s) \lg n)$ time using $O(z)$ space for constant-sized alphabets or $O(z \lg \lg n)$ space for general alphabets.

4.2 Preliminaries

We assume a standard unit-cost RAM model with word size $w = \Theta(\lg n)$ and that the input is from an integer alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$ where $\sigma \leq n^{O(1)}$, and we measure space complexity in words unless otherwise specified. A string S of length $n = |S|$ is a sequence $S[1] \ldots S[n]$ of n symbols from an alphabet Σ of size $|\Sigma| = \sigma$. The string $S[i] \ldots S[j]$ denoted $S[i, j]$ is called a *substring* of S. Let ε denote the empty string and let $S[i, j] = \varepsilon$ when $i > j$. To ease the notation, let $S[i, j] = S[1, j]$ if $i < 1$ and $S[i, n]$ if $j > n$. Let $[u]$ be shorthand for the interval $[1; u] = \{1, 2, \ldots, u\}$ and let \$ be a special symbol that never occurs in the input text. A straight-line program (SLP) is an acyclic grammar in Chomsky normal form where each non-terminal T has exactly one production rule with T as its left-hand side i.e., a grammar where each non-terminal production rule expands to two other rules and generates one string only.

4.2.1 Lempel-Ziv 77 Algorithm

For simplicity of exposition we use the scheme given by Farach & Thorup [27]. Map Σ into $[\sigma]$ and assume that S is prefixed by Σ in the negative positions, i.e. $S[-c] = c$ for $c \in \Sigma$ and $S[0] = \$ \notin \Sigma$.

An *LZ77 representation* [64, 91] of S is a string $\mathcal{Z}$ of the form $(s_1, l_1) \dots (s_z, l_z) \in ([-\sigma; n] \times [n])^z$. Let $u_1 = 1$ and $u_i = u_{i-1} + l_{i-1}$, for $i > 1$. For $\mathcal{Z}$ to be a valid LZ77 representation of S, we require that $s_i + l_i \leq u_i$ and that $S[u_i, u_i + l_i - 1] = S[s_i, s_i + l_i - 1]$ for $i \in [z]$. This guarantees that $\mathcal{Z}$ *represents* S and clearly S is uniquely defined in terms of $\mathcal{Z}$.

We refer to the substring $S[u_i, u_i + l_i - 1]$ as the i^{th} *phrase* of the representation, the substring $S[s_i, s_i + l_i - 1]$ as the *source of the* i^{th} *phrase* and (s_i, l_i) as the i^{th} *member* of $\mathcal{Z}$. We note that the restriction $s_i + l_i \leq u_i$ for all i implies that a source and a phrase cannot overlap and thus we do not handle representations that are self-referential.

By the given definition, the LZ77 representation of a string is not unique, however a minimal $LZ77$ representing a text S can be found greedily in $O(n)$ time [24, 53].

4.2.2 Mergeable Dictionary

The Mergeable Dictionary problem is to maintain a dynamic collection $\mathcal{G}$ of disjoint sets $\{G_1, G_2, \dots\}$ of n elements from an ordered universe $\{1, 2, \dots, \mathcal{U}\}$ starting from n singleton sets under the operations:

1. $C \leftarrow \texttt{merge}(A, B)$: Creates $C = A \cup B$. C is inserted into $\mathcal{G}$ while A and B are removed.

2. $(A, B) \leftarrow \texttt{split}(G, x)$: Splits G into two sets $A = \{y \in G \mid y \leq x\}$ and $B = \{y \in G \mid y > x\}$. G is removed from $\mathcal{G}$ while A and B are inserted.

Iacono & Özkan [51] show how to solve the mergeable dictionary problem. The initial collection of singleton sets is created in linear time [51]. We need an extended version of the mergeable dictionary that also supports the following operation.

3. $G' \leftarrow \texttt{shift}(G, x)$ for some x such that $y + x \in [\mathcal{U}]$ for each $y \in G$: Creates the set $G' = \{y + x \mid y \in G\}$. G is removed from $\mathcal{G}$ while G' is inserted.

Now, there is no guarantee that the sets in $\mathcal{G}$ remain disjoint. Iacono & Özkan [51] write that their data structure can be extended to support the shift operation but do not provide any details. We show how shifts are done in Appendix A.1 and thus we obtain the following:

Theorem 4.5 *There exists a data structure for the Mergeable Dictionary Problem supporting any sequence of m split, shift and merge operations in worst-case $O(m \lg \mathcal{U})$ time using $O(n)$ space.*

4.3 LZ77 Induced Context

In this section we present the centerpiece of our algorithm. It builds on the fundamental property of LZ77 compression that any substring of a phrase also occurs in the source of that phrase. Our technique is to store a short substring, which we call *context*, around the start and end of every phrase. The contexts are stored in a compressed form that allows faster substring extraction than that of LZ77. We then take advantage of this property when extracting a substring of S by splitting it into short chunks which in turn are extracted by repeatedly mapping them to the source of the phrase they are part of. Eventually, they will end up as a substrings of the contexts from where they can be efficiently extracted.

The technique resembles what Farach & Thorup refer to as *winding* in [27]. We show new applications of the technique and obtain better time complexity by using the *mergeable dictionaries* by Iacono & Özkan [51] presented in Section 4.2.2.

In Section 4.3.1 we show how to obtain an LZ77 parse for the subsequence of S that includes only the context of every phrase. This representation can then efficiently be transformed into an SLP as shown in Section 4.3.2 using the online construction algorithm by Charikar et al. [18] or Rytter [84].

Recall that we assume S is prefixed by the alphabet in the negative positions, that $S[0] = \$$, and that u_k is the starting position in S of the k^{th} phrase.

Definition 4.1 *Let τ be a positive integer. The* τ-context *of a string S (induced by an LZ77 representation $\mathcal{Z}$ of S) is the set of positions j where either $j \leq 0$ or there is some k such that $u_k - \tau < j < u_k + \tau$. If positions i through j are in the τ-context of S, then we simply say "$S[i, j]$ is in the τ-context of S".*

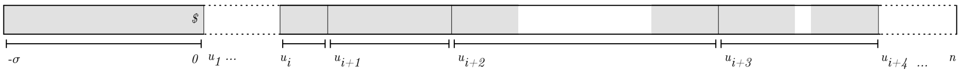


Figure 4.1: Example of the τ-context of a string. Dashed parts are truncated parts of the string not shown by the figure, grey parts represent substrings in the τ-context and white parts represent substrings not in the τ-context. The first substring in the negative positions $-\sigma$ through 0 is always in the τ-context. Recall that l_i is the length of the i^{th} phrase. In this example, $l_i < \tau, l_{i+1} \leq 2\tau$ and $l_{i+2}, l_{i+3} > 2\tau$.

Definition 4.2 *Let τ be a positive integer. The* τ-context string *of S, denoted S^τ, is the subsequence of S that includes $S[j]$ if and only if j is in the τ-context of S. We denote with $\pi^\tau(j)$ the unique position in S^τ where such a position j is mapped to (i.e. $S^\tau[\pi^\tau(j)] = S[j]$).*

We show how to map positions from S to S^τ.

Lemma 4.1 *Let $\mathcal{Z}$ be an LZ77 representation of a string S of length n with z phrases and let τ be a positive integer. Given $t = O(z)$ sorted positions, $p_1 \leq \ldots \leq p_t \in [n]$ in the τ-context of S we can compute $\pi^\tau(p_1), \ldots, \pi^\tau(p_t)$ in $O(z)$ time and space.*

Proof Let $\text{gap}_k = \max\{0, l_k - 2\tau + 1\}$ be the number of positions inside the k-th LZ77 phrase that are not in the τ-context of S.

Let i, k, L be three integers initialized as follows: $i = 0$, $k = 1$, and $L = 0$. We keep the following two invariants:

(i) if $i > 0$, then k is the smallest integer such that $p_i < u_k + l_k$, i.e. position p_i is in the k-th phrase, and

(ii) L is the number of positions $j < u_k$ such that j is in the τ-context of S (i.e. L is the length of the prefix of S^τ containing characters from $S[1..u_k - 1]$).

It is clear that (i) and (ii) hold in the beginning of our procedure. We now show how to iterate through the LZ77 phrases and compute the desired output in one pass.

Assume that we already computed $\pi^\tau(p_1), \ldots, \pi^\tau(p_i)$ (or none of them if $i = 0$). To compute $\pi^\tau(p_{i+1})$, we check whether $u_k \leq p_{i+1} < u_k + l_k$, i.e. whether p_{i+1} is in the k-th phrase. If not, we find the phrase containing p_{i+1} as follows. We set $L \leftarrow L + l_k - \text{gap}_k$, $k \leftarrow k + 1$ and repeat until we find a value of k that satisfies $u_k \leq p_{i+1} < u_k + l_k$. It is clear that, at each step, L is still the length of the prefix of S^τ containing characters from $S[1..u_k - 1]$ (i.e. invariant (ii) is maintained).

Once such a k is found, we compute $\pi^\tau(p_{i+1})$ simply adding L to the relative position of p_{i+1} inside its phrase, and subtract gap_k from this quantity if p_{i+1} is within τ characters from the end of the phrase. More in detail, if $p_{i+1} < u_k + \tau$, then $\pi^\tau(p_{i+1}) \leftarrow L + 1 + (p_{i+1} - u_k)$. Otherwise, $\pi^\tau(p_{i+1}) \leftarrow L + 1 + (p_{i+1} - u_k) - \text{gap}_k$. The correctness of this computation is guaranteed by the way we defined L in property (ii).

Note that k is again the smallest integer such that $p_{i+1} < u_k + l_k$ (invariant (i)), so we can proceed with the same strategy to compute $\pi^\tau(p_{i+2}), \ldots, \pi^\tau(p_t)$.

Overall, the algorithm runs in $O(z)$ time and space.

We use π as shorthand for π^τ whenever τ is clear from context. The following properties follow from the definitions and Lemma 4.1 but will come in handy later on:

Property 4.1 *If a, a' are positions in the τ-context of S and $a < a'$ then $\pi(a) < \pi(a')$.*

Property 4.2 *If $S[a, b]$ is in the τ-context of S then $S^\tau[\pi(a), \pi(b)] = S[a, b]$*

We now consider the following problem: given a substring $S[i, j]$ of length at most τ, find a pair of integers (i', j') such that $i' \leq i$, $S[i, j] = S[i', j']$ and $S[i', j']$ is in the τ-context of S.

We first give an informal overview of how the algorithm works. Recall that if a substring of S is contained within a phrase in the LZ77 parse of S, then the substring also occurs in the source of that phrase. The idea is to repeat this process of finding an identical substring in the source until the found string is in the τ-context of S, which happens after at most z steps. To do this efficiently for multiple strings, we use the mergeable dictionary structure to maintain the relevant positions. This allows us to process all strings inside a phrase simultaneously because they all need to be moved to the same source. By processing the phrases in right-to-left order we can bound the number of dictionary operations by the number of phrases.

The following algorithm gives the details of how to solve the problem for a set of z substrings using $O(z)$ space and $O(z \lg n)$ time.

Algorithm 4.1 *Let $\mathcal{Z}$ be an LZ77 representation of a string S of length n with z phrases and let τ be a positive integer. The input is $t = O(z)$ substrings of S given as pairs of integers denoting start and end position: $(a_1, b_1), \ldots, (a_t, b_t)$ where $b_i - a_i < \tau$ for all $i \in [t]$. Let $\mathcal{G}$ be a mergeable dictionary as given by Lemma 4.5. For each of the pairs (a_i, b_i) create a singleton set G_i with element x_i at position a_i and finally merge all these elements into a single set G. Each element x_i has associated its rank i (i.e. its rank among the input pairs) as satellite information.*

We now consider the members of $\mathcal{Z}$ one by one in reverse order. Member (s_i, l_i) is processed as follows:

1. *If $l_i \leq \tau$ skip to the next member.*

2. *Otherwise let*

a) $(A, B) \leftarrow \texttt{split}(G, u_i + l_i - \tau)$

b) $(A', B') \leftarrow \texttt{split}(A, u_i - 1)$,

c) $B'' \leftarrow \texttt{shift}(B', s_i - u_i)$

d) $G \leftarrow \texttt{merge}(A', B'')$.

In step 1, we skip a phrase if it is no longer than τ because any string of length τ or shorter starting in that phrase is already in the τ-context of S. In step 2a, we split the set such that all strings that start in the last τ positions of the phrase are not shifted, because these already are in the τ-context of S. In 2a-d we split the set to obtain the set of strings B'' that starts in the i^{th} phrase excluding those starting in the last τ positions, as they are already in the τ-context of S. These strings are then shifted to the source and will be considered again in later iterations.

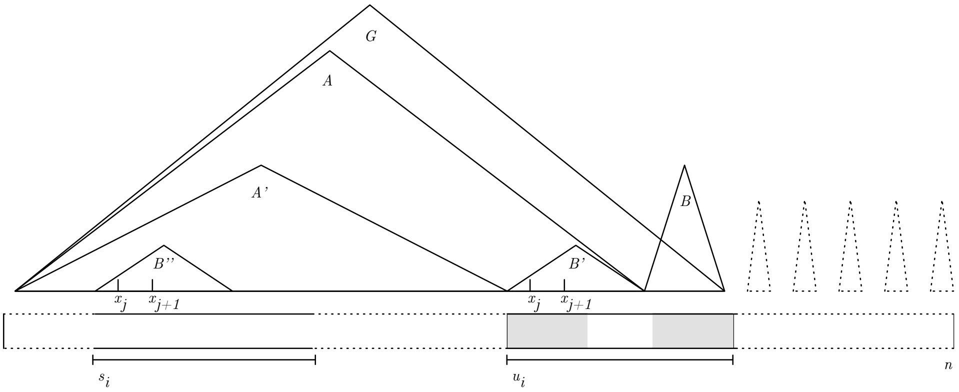


Figure 4.2: Example of the dictionaries created during an iteration. The dashed parts of the string are truncated parts not relevant to the example. The dotted triangles represent the B-sets from earlier iterations. The grey parts show the τ-context of the string inside the i^{th} and $i - 1^{th}$ phrase. Note that the set B'' is the set B' after the shift operation. As exemplified by the elements x_j and x_{j+1}, the relative order and position inside the set is unaffected by the shift. Let p and p' be the position of x_j before and after the shift, respectively. Observe also that $S[p, p + (b_j - a_j) - 1] = S[p', p' + (b_j - a_j) - 1]$, so the shift does not affect the substring represented by x_j. An iteration starts from the set G, obtain A by cutting off B. The new G (not shown in the figure) is then obtained by shifting all elements in the range of B', which are all contained in the i^{th} phrase to the same relative position in the source of the phrase.

After processing all members, scan each set in $\mathcal{G}$ to retrieve all the elements. Let $p(x_i)$ denote the new position in $\mathcal{G}$ of element x_i. We then output the pairs $(p(x_1), e_1), \ldots, (p(x_t), e_t)$ in order of their rank i where $e_i = p(x_i) + b_i - a_i$.

Correctness *Let $p(x_i)$ denote the position in $\mathcal{G}$ of element x_i at any point of the algorithm. We now show that for any element x_i, we have $S[a_i, b_i] = S[p(x_i), e_i]$ both before and after considering the j^{th} member of $\mathcal{Z}$. Initially, $p(x_i) = a_i$ so this is trivially true before the first iteration.*

Assume by induction that this is true before considering member j. If $p(x_i) > u_j + l_j - \tau$ or $p(x_i) < u_j$, then $p(x_i)$ will not be changed when considering member i. Otherwise, $u_j \leq p(x_i) \leq u_j + l_j - \tau$, and thus $S[p(x_i), e_i]$ is a substring of $S[u_j, u_j + l_j - 1]$ which also occurs at the same relative position in $S[s_j, s_j + l_j - 1]$. Now x_i is shifted such that $p(x_i) \leftarrow p(x_i) - u_j + s_j$ thereby maintaining the relative position inside the two *identical strings and it follows that x_i still represents $S[a_i, b_i]$ after considering member j and thus also before considering member $j-1$.*

We now show that, for any element x_i, the string $S[p(x_i), e_i]$ is in the τ-context of S after considering the last member. Observe that when considering member j, every element positioned in $S[u_j, u_j + l_j - \tau]$ is shifted to a position less than u_j, because $s_j + l_j \leq u_j$ by definition. As we are considering the members in reverse order, this means that every element x_i must end in a position such that either there is some k such that $u_k + l_k - \tau < p(x_i) < u_k + l_k$ or $p(x_i) < 0$ which concludes the proof of correctness.

Complexity *Creating the z singleton elements with positions in the range $[n]$ and merging them to G takes $O(z \lg n)$ time. For every member of $\mathcal{Z}$ we do $O(1)$ dictionary operations. All positions remain in the range $[-\sigma; n]$ thus this also takes total time $O(z \lg n)$. We can easily compute and store $u_1, \ldots, u_z$ in $O(z)$ time and space. Outputting the elements x_i in order of their rank i takes linear time as the ranks are consecutive integers thus the total time is $O(z \lg n)$. We never store more than the $O(z)$ elements, thus the total space is $O(z)$.*

This proves the following lemma:

Lemma 4.2 *Let $\mathcal{Z}$ be an LZ77 representation of a string S of length n with z phrases and let τ be a positive integer. Given $t \in O(z)$ substrings of S as pairs of integers $(a_1, b_1), \ldots, (a_t, b_t)$ where $b_i - a_i < \tau$ we can find t pairs of integers $(a'_1, b'_1), \ldots, (a'_t, b'_t)$ such that $S[a'_i, b'_i] = S[a_i, b_i]$, $a'_i \leq a_i$ and $S[a'_i, b'_i]$ is in the τ-context of S using $O(z)$ space and $O(z \lg n)$ time.*

4.3.1 LZ77 Compressed Context

It is possible to obtain an LZ77 representation $\mathcal{Z}^\tau$ of the string S^τ directly from an LZ77 representation $\mathcal{Z}$ of S. Informally, the idea is to split every phrase of $\mathcal{Z}$ into two new phrases consisting of respectively the first and last $O(\tau)$ characters of the phrase. In order to find a source for these phrases, we use Algorithm 4.1 which finds an identical string that also occurs in S^τ.

We now describe the algorithm sketched above that constructs an LZ77 representation $\mathcal{Z}^\tau$ of S^τ given the LZ77 parse $\mathcal{Z}$ of S.

Algorithm 4.2 *First we construct $O(z)$* relevant pairs *of integers representing substrings of S by considering the members of $\mathcal{Z}$ one by one in order. Member (s_i, l_i) is processed as follows:*

1. *If $l_i \leq \tau$: Let $(u_i, u_i + l_i - 1)$ be a relevant pair.*
2. *If $\tau < l_i < 2\tau$: Let $(u_i, u_i + \tau - 1)$ and $(u_i + \tau, u_i + l_i - 1)$ be relevant pairs.*
3. *Otherwise $l_i \geq 2\tau$: Let $(u_i, u_i + \tau - 1)$ and $(u_i + l_i - \tau + 1, u_i + l_i - 1)$ be relevant pairs.*

Each of the relevant pairs represents a prefix or a suffix of a phrase. The concatenation of these phrase prefixes and suffixes in left-to-right order is exactly the string S^τ. Let (a, b) be a relevant pair created when considering the i^{th} member of $\mathcal{Z}$. Then we say that $(a', b') = (a - u_i + s_i, b - u_i + s_i)$ is the related source pair *pair and clearly $S[a, b] = S[a', b']$.*

Note that the related source pairs might not be in the τ-context. We now use Algorithm 4.1 to find a pair of integers (a'', b'') for each related source pair (a', b') such that $S[a'', b''] = S[a', b']$, $a'' \leq a'$ and $S[a'', b'']$ is in the τ-context of S. We give the pairs in order of creation and this order is preserved by Algorithm 4.1. If (a'', b'') is the i^{th} output of Algorithm 4.1 then $(\pi^\tau(a''), l)$ is the i^{th} member of Z^τ where $l = b - a + 1$ and $\pi^\tau(a'')$ is computed using Lemma 4.1.

Correctness *Let $(a_1, b_1), \ldots, (a_t, b_t)$ be the relevant pairs in order of creation, $(a'_1, b'_1) \ldots (a'_t, b'_t)$ be the related source pairs, $(a''_1, b''_1) \ldots (a''_t, b''_t)$ be the output of Algorithm 4.1, and let $l_i = b_i - a_i + 1$.*

Our goal is to show that $\mathcal{Z}^\tau = (\pi(a''_1), l_1), \ldots, (\pi(a''_t), l_t)$ is a valid LZ77 representation of S^τ, that is: (i) the concatenation of the phrases of $\mathcal{Z}^\tau$ yields S^τ, (ii) phrases of $\mathcal{Z}^\tau$ are equal to their sources, and (iii) phrases of $\mathcal{Z}^\tau$ do not overlap their sources. Note that $\mathcal{Z}^\tau$ consists of at most $2z$ phrases

(i-ii) It follows directly from Definition 4.2 that the concatenation of the strings represented by the relevant pairs in order of creation is $S[a_1, a_1] \cdots S[a_t, b_t] = S^\tau$. Since $S[a_i, b_i] = S[a'_i, b'_i]$ and, by Lemma 4.2, $S[a''_i, b''_i] = S[a'_i, b'_i]$ then we also have that $S^\tau = S[a''_1, a''_1] \cdots S[a''_t, b''_t]$. Now, observe that since $S[a_i, b_i]$ and $S[a''_i, b''_i]$ are in the τ-context of S then by Property 4.2 we have $S[a_i, b_i] = S^\tau[\pi(a_i), \pi(b_i)]$ and $S[a''_i, b''_i] = S^\tau[\pi(a''_i), \pi(b''_i)]$. This proves properties (i) and (ii).

(iii) By definition of the LZ77 representation of S and since the substring represented by the pair (a_i, b_i) is entirely contained in a phrase we must have $a'_i + l_i \leq a_i$ and therefore, by Lemma 4.2, $a''_i \leq a'_i$. But this means that $a''_i + l_i \leq a_i$ and therefore, by Property 4.1, $\pi(a''_i + l_i) \leq \pi(a_i)$, i.e. property (iii) holds.

Complexity *For every member of $\mathcal{Z}$ we create at most two relevant substrings taking total $O(z)$ time and space. Applying Algorithm 4.1 takes time $O(z \lg n)$ and $O(z)$ space. We can easily compute and store $u_1, \ldots, u_z$ time and space and computing $\pi^\tau(a')$ for every substring reported by Algorithm 4.1 takes total $O(z)$ time and space using Lemma 4.1 thus the total time is $O(z \lg n)$ and the total space is $O(z)$.*

In summary, we have the following lemma:

Lemma 4.3 *Let $\mathcal{Z}$ be an LZ77 representation of a string S of length n with z phrases. We can construct an LZ77 representation $\mathcal{Z}^\tau$ of S^τ with $O(z)$ phrases in $O(z \lg n)$ time and $O(z)$ space.*

4.3.2 SLP and Word Compressed Context

In this section we consider how to store in compressed form a τ-context string of S.

Our first solution is to decompress the context string efficiently and store it using word packing. First, we construct the LZ77 representation of S^τ using Algorithm 4.2 and decompress it naively. Constructing the representation takes time $O(z \lg n)$ while decompressing it takes linear time in its length $O(z\tau)$. A string of length $z\tau$ can be stored in $O(z\tau \lg \sigma / \lg n)$ words using word packing. We obtain:

Lemma 4.4 *Let S be a string S of length n from an alphabet of size σ compressed into an LZ77 representation with z phrases, and let τ be a positive integer. We can construct and store the τ-context of S in $O(z(\lg n + \tau))$ time and $O(z\tau \lg \sigma / \lg n)$ space.*

As an alternative solution, we show how to store the context string as an SLP. First we need the following theorem:

Theorem 4.6 (Charikar et al. [18], Rytter [84]) *Let z be the number of phrases in the LZ77 representation of a string S of length n. We can build a balanced SLP for S^τ with height $O(\lg n)$ in $O(z \lg(n/z))$ space and time.*

At this point, we build the LZ77 representation of S^τ using Lemma 4.3 and then convert it into an SLP using Theorem 4.6. Note that $n' = |S^\tau| \leq z\tau$ and the SLP's size is $O(z \lg(n'/z)) = O(z \lg(z\tau/z)) = O(z \lg \tau)$. We obtain:

Corollary 4.1 *Let S be a string of length n compressed into an LZ77 representation with z phrases, and let τ be a positive integer. We can build a balanced SLP for S^τ with height $O(\lg n)$ in $O(z \lg \tau)$ time and space.*

4.4 LZ77 Decompression

We now describe how to apply the techniques described in the previous section to extract arbitrary substrings of S.

We first show how to extract a substring of length l. Let S be a string of length n compressed into an LZ77 representation with z phrases and let τ be a positive integer that we will fix later.

Split the string into consecutive blocks of length τ and process a batch of z blocks at a time in left-to-right order. There are $O(1 + l/(\tau z))$ batches each containing z blocks. A batch is processed in $O(z \lg n)$ time using Lemma 4.2 thereby finding a substring s' in the τ-context of S for every block s in the batch.

Using Corollary 4.1 these z substrings can be extracted in $O(z \lg n + z\tau)$ time. Thus the time to extract and output all batches is $O(l + (1 + \frac{l}{\tau z})z \lg n) = O(l + \frac{l \lg n}{\tau} + z \lg n)$ while the time to construct the SLP is $O(z \lg \tau)$. The total space is $O(z \lg \tau)$. If we instead use Lemma 4.4 the time is unchanged while the space becomes $O(z\tau \lg \sigma / \lg n)$

Generalizing to s substrings of total length l the procedure is similar. We split each string into blocks of length τ (or less, if the string's length is not a multiple of τ) and process a batch of z such blocks at a time. In total, there are no more than $O(s + l/\tau)$ blocks of length at most τ. Now there are $O(1+s/z+l/(\tau z))$ batches of $O(z)$ blocks thus the total time becomes $O(l + (s + z) \lg n + \frac{l \lg n}{\tau})$ using $O(z \lg \tau)$ space with Corollary 4.1 which proves Theorem 4.2. Using Lemma 4.4 the time remains the same while the space again becomes $O(z\tau \lg \sigma / \lg n)$. Fixing $\tau = \lg n / \lg^{\delta} \sigma$ for any constant $0 \leq \delta \leq 1$ the time is $O\left(l + (s + z) \lg n + \frac{l \lg n}{\lg n / \lg^{\delta} \sigma}\right) = O((s + z) \lg n + l \lg^{\delta} \sigma)$ while the space is $O(z \frac{\lg n \lg \sigma}{\lg^{\delta} \sigma \lg n}) = O(z \lg^{1-\delta} \sigma)$ which proves Theorem 4.1.

Theorems 4.1 and 4.2 give rise to the following two corollaries on the complexity of decompressing the entire string S.

Corollary 4.2 *For any parameter $0 \leq \delta \leq 1$, we can decompress S in $O(n \lg^{\delta} \sigma)$ time using $O(z \lg^{1-\delta} \sigma)$ space.*

Proof When decompressing the entire text, the number of substrings is $s = 1$ and the total length is $l = n$. First, we compute n and σ with a simple scan of the LZ77 representation of S. If $z \lg n \leq n \lg^{\delta} \sigma$, then we use Theorem 4.1. This solution runs in time $O(n \lg^{\delta} \sigma + z \lg n) = O(n \lg^{\delta} \sigma)$ and uses $O(z \lg^{1-\delta} \sigma)$ space.

Otherwise $z \lg n > n \lg^{\delta} \sigma$, but then it must be the case that $z \lg^{1-\delta} \sigma = \frac{z \lg n \lg^{1-\delta} \sigma}{\lg n} > \frac{n \lg^{\delta} \sigma \lg^{1-\delta} \sigma}{\lg n} = \frac{n \lg \sigma}{\lg n}$ (inequality follows from replacing $z \lg n$ with $n \lg^{\delta} \sigma$). This means that the entire string S can be packed in $O(\frac{n \lg \sigma}{\lg n}) \subseteq O(z \lg^{1-\delta} \sigma)$ words, so we can decompress it naively in $O(n)$ time and $O(z \lg^{1-\delta} \sigma)$ space.

Note that, in particular, Corollary 4.2 achieves $O(n)$ time and $O(z)$ space when σ is constant. On large alphabets, we can further improve upon this result:

Corollary 4.3 *We can decompress S in $O(n)$ time using $O(z \lg \lg n)$ space.*

Proof We compute n and σ with a scan of the LZ77 representation of S. If $z \lg n \leq n$, then we use Theorem 4.2 with $\tau = \lg n$. This solution runs in time $O(n + z \lg n) = O(n)$ and uses space $O(z \lg \tau) = O(z \lg \lg n)$.

Otherwise $z \lg n > n$. In this case, we simply build a grammar for S of size $O(z \lg(n/z)) \subseteq O(z \lg \lg n)$ using [84]. The time to build the grammar is $O(z \lg(n/z)) \subseteq O(n)$. We use this grammar to stream the text in $O(n)$ time.

Note that, when aiming at linear running time, Corollary 4.3 is at least as good as Corollary 4.2 whenever $\sigma \in \Omega(\lg n)$ (and asymptotically better for $\lg \sigma \in \omega(\lg \lg n)$).

4.5 Applications in Pattern Matching

In this section we show how our techniques can be applied as a black box in combination with existing pattern matching results.

Let S be a string of length n and let P be a pattern of length m. The classical *pattern matching problem* is to report all starting positions of occurrences of P in S. In the *approximate pattern matching problem* we are given an error threshold k in addition to P and S. The goal is to find all starting positions of substrings of S that are within distance k of P under some metric, e.g. *edit distance* where the distance is the number of edit operations required to convert the substring to P. When considering *the compressed pattern matching problem*, the string S is given in some compressed form. Sometimes, we are only interested in whether or not P occurs in S.

Pattern matching on LZ77 compressed texts usually takes advantage of the property that any substring of a phrase also occurs in the source of the phrase. This means that if an occurrence of P is contained in single phrase, then there must also be an occurrence in the source of that phrase. The implication is that the occurrences of P can be split into two categories: the ones that overlap two or more phrases and the ones that are contained inside a single phrase — usually referred to as primary and secondary occurrences, respectively [55]. The secondary occurrences can be found from the primary in $O(z + \mathrm{occ})$ time and space [41] where z is the number of phrases in the LZ77 representation of S and occ is the total number of occurrence of P in S.

Approximate pattern matching in small space Consider the \$-padded m-context string of S denoted $S^{\$m}$ obtained by replacing each of the maximal substrings of S that are not in the m-context by a single copy of the symbol \$. This string has length $O(zm)$. Observe that all the primary occurrences of P in S are in this string, that any occurrence of P in this string corresponds to a unique primary or secondary occurrence of P in S and that we can map these occurrences to their position in S in $O(\mathrm{occ} + z \lg z)$ time and $O(z)$ space using the same technique as Lemma 4.1 but by adding the gap lengths instead of subtracting them.

We now prove Theorem 4.3. Let $\mathcal{A}$ be the (approximate) pattern matching algorithm from the theorem using $t(z, n, m, k)$ time and $s(z, n, m, k)$ space. The idea is to run algorithm $\mathcal{A}$ on an LZ77 representation of $S^{\$m}$ to find all the primary occurrences.

Our algorithm works as follows. First create an LZ77 representation $\mathcal{Z}^{\$m}$ of $S^{\$m}$ using Algorithm 4.2. If two consecutive phrases of $\mathcal{Z}^{\$m}$ are both induced by the same phrase of $\mathcal{Z}$ of length $2m$ or more we add a phrase between them representing only the symbol \$. This is easy to do as part of Algorithm 4.2 without changing its complexity and the result is exactly an LZ77 representation of $S^{\$m}$ with $O(z)$ phrases. The pattern P occurs in S if and only if it occurs in $S^{\$m}$. Thus we can run algorithm $\mathcal{A}$ on $\mathcal{Z}^{\$m}$ to detect an (approximate) occurrence of P in S. Constructing $\mathcal{Z}^{\$m}$ takes $O(z \lg n)$ time and $O(z)$ space thus the total time becomes $O(z \lg n + t(z, zm, m, k))$ time and $O(z + s(z, zm, m, k))$ space.

All primary occurrences are found by finding all occurrences of P in $S^{\$m}$, mapping them to their positions in S and filtering out the secondary occurrences. Hereafter, all the secondary occurrences can be found in $O(z + \mathrm{occ})$ time and space [41]. Mapping and filtering also takes $O(\mathrm{occ} + z)$ time and space. Thus, if algorithm $\mathcal{A}$ reports all (approximate) occurrences of P in S in $t(z, n, m, k)$ time and $s(z, n, m, k)$ space we can report all (approximate) occurrences of P in S in $O(z \lg n + t(z, zm, m, k) + \mathrm{occ})$ time and $O(z + s(z, zm, m, k) + \mathrm{occ})$ space.

We now prove Theorem 4.4. Let $\mathcal{A}$ be the streaming algorithm from the theorem that reports all (approximate) occurrences of P in a stream of length n using $t(n, m, k)$ time and $s(n, m, k)$ space. The idea is to run algorithm $\mathcal{A}$ on the string $S^{\$m}$, which we will stream in chunks to find all the primary occurrences. We use the same technique as above to first filter the primary occurrences and then find all the secondary occurrences in $O(z + \mathrm{occ})$ time and space.

We stream the string $S^{\$m}$ consisting of $O(z)$ substrings of total length $O(zm)$. We can easily compute when to output a \$ during the substring extraction. Thus using Theorems 4.1 or 4.2 we can stream $S^{\$m}$ in either $O(zm)$ time using $O(z \lg\lg n)$ space or $O(zm \lg^{\delta} \sigma + z \lg n)$ time using $O(z \lg^{1-\delta} \sigma)$ space. The total time is then either:

- $O(t(zm, m, k) + z \lg n)$ time and $O(s(zm, m, k) + z \lg\lg n + \mathrm{occ})$ space or
- $O(t(zm, m, k) + z \lg n + zm \lg^{\delta} \sigma)$ time and $O(s(zm, m, k) + z \lg^{1-\delta} \sigma + \mathrm{occ})$ space.

Compressed Existence Gawrychowski [44] shows how to decide if P occurs in S given P and the LZ77 representation of S using $O(z \lg(n/z) + m)$ time and space. Applying Theorem 4.3 we get the following:

Corollary 4.4 *We can detect an occurrence of a pattern P of length m given the LZ77 representation of S in $O(z \lg n + m)$ time using $O(z \lg m + m)$ space.*

Approximate Pattern Matching

By combining the Landau-Vishkin and Cole-Hariharan [22, 62] algorithms all approximate occurrences with at most k errors on a stream of length n can be found in $O(\min\{nk, nk^4/m + n\})$ time using $O(m)$ space. Gagie et al. [41] shows how this algorithm can be used to solve the same problem given an LZ77 representation of S in $O(z \lg n + z \cdot \min\{mk, k^4 + m\} + \mathrm{occ})$ time and $O(z \lg n + m + \mathrm{occ})$ space. Applying Theorem 4.4 to the combined Landau-Vishkin and Cole-Hariharan algorithm we get the following new trade-offs:

Corollary 4.5 *We can report all approximate occurrences of a pattern P of length m with k errors given the LZ77 representation with z phrases of a string S of length n in:*

- $O(z \lg n + z \cdot \min\{mk, k^4 + m\} + \mathrm{occ})$ *time using* $O(z \lg\lg n + m + \mathrm{occ})$ *space or*
- $O(z \lg n + z \cdot \min\{mk, k^4 + m\} + zm \lg^{\delta} \sigma + \mathrm{occ})$ *time using* $O(z \lg^{1-\delta} \sigma + m + \mathrm{occ})$ *space.*

4.6 Conclusions

In this paper we described the first solution for decompressing Lempel-Ziv 77 in linear time using a space proportional to the input's size on constant alphabets. On general alphabets, we presented a trade-off that allows getting either linear time or linear space. Our solutions can, in general, decompress any subsequence of the text. Our work leaves several open problems. First of all, our solutions for general alphabets cannot achieve both linear time and space. We also note that our running times could be improved by fully exploiting packed computation; while it is definitely possible to slightly improve running times of Theorems 4.1 and 4.2 in this sense (at the price of a higher space usage), the (opportunely adjusted) case analysis of Corollaries 4.2 and 4.3 would not yield optimal packed extraction times. We therefore suspect that a different technique is needed in order to achieve optimality. Finally, we did not a straight forward way to generalize our results to self-referential LZ77. However, we do believe that this is possible through a closer analysis of our application of the mergeable dictionary.

Chapter 5

Compressed Communication Complexity of Longest Common Prefixes

Philip Bille* Mikko Berggreen Ettienne* Roberto Grossi† Inge Li Gørtz* Eva Rotenberg*

* The Technical University of Denmark
† Università di Pisa, Italy

Abstract

We consider the communication complexity of fundamental longest common prefix (Lcp) problems. In the simplest version, two parties, Alice and Bob, each hold a string, A and B, and we want to determine the length of their longest common prefix $\ell = \text{Lcp}(A, B)$ using as few rounds and bits of communication as possible. We show that if the longest common prefix of A and B is compressible, then we can significantly reduce the number of rounds compared to the optimal uncompressed protocol, while achieving the same (or fewer) bits of communication. Namely, if the longest common prefix has an LZ77 parse of z phrases, only $O(\lg z)$ rounds and $O(\lg \ell)$ total communication is necessary. We extend the result to the natural case when Bob holds a set of strings $B_1, \ldots, B_k$, and the goal is to find the length of the maximal longest prefix shared by A and any of $B_1, \ldots, B_k$. Here, we give a protocol with $O(\lg z)$ rounds and $O(\lg z \lg k + \lg \ell)$ total communication. We present our result in the public-coin model of computation but by a standard technique our results generalize to the private-coin model. Furthermore, if we view the input strings as integers the problems are the greater-than problem and the predecessor problem.

5.1 Introduction

Communication complexity is a basic, useful model, introduced by Yao [90], which quantifies the total number of bits of communication and rounds of communication required between two or more players to compute a function, where each player holds only part of the function's input. A detailed description of the model can be found, for example, in the book by Kushilevitz and Nisam [61].

Communication complexity is widely studied and has found application in many areas, including problems such as equality, membership, greater-than, and predecessor (see the recent book by Rao and Yehudayoff [83]). For the approximate string matching problem, the paper by Starikovskaya [87] studies its deterministic one-way communication complexity, with application to streaming algorithms, and provides the first sublinear-space algorithm. Apart from these results, little work seems to have been done in general for the communication complexity of string problems [88].

In this paper, we study the fundamental *longest common prefix problem*, denoted LCP, where Alice and Bob each hold a string, A and B, and want to determine the length of the longest common prefix of A and B, that is, the maximum $\ell \geq 0$, such that $A[1..\ell] = B[1..\ell]$ (where $\ell = 0$ indicates the empty prefix). This problem is also called the *greater than problem*, since if we view both A and B as integers, the position immediately after their longest common prefix determines which is larger and smaller. The complexity is measured using the number of rounds required and the total amount of bits exchanged in the communication. An optimal randomized protocol for this problem uses $O(\lg n)$ communication and $O(\lg n)$ rounds [76] where n is the length of the strings. Other trade-offs between communication and rounds are also possible [86]. Buhrman et al. [16] describe how to compute LCP in $O(1)$ rounds and $O(n^{\varepsilon})$ communication.

We show that if A and B are compressible we can significantly reduce the number of needed rounds while simultaneously matching the $O(\lg n)$ bound on the number of bits of communication. With the classic and widely used Lempel-Ziv 77 (LZ77) compression scheme [91] we obtain the following bound.

Theorem 5.1 *The* LCP *problem has a randomized public-coin $O(\lg z)$-round protocol with $O(\lg \ell)$ communication complexity, where $\ell \leq n$ is the length of the longest common prefix of A and B and $z \leq \ell$ is the number of phrases in the LZ77 parse of this prefix.*

Compared to the optimal uncompressed bound we reduce the number of rounds from $O(\lg n)$ to $O(\lg z)$ (where typically z is much smaller than n). At the same time we achieve $O(\lg \ell) = O(\lg n)$ communication complexity and thus match or improve the $O(\lg n)$ uncompressed bound. Note that the number of rounds is both compressed and output sensitive and the communication is output sensitive.

As far as we know, this is the first result studying the communication complexity problems in LZ77 compressed strings. A previous result by Bar-Yossef et al. [5] gives some impossibility results on compressing the text for (approximate) string matching in the sketching model, where a sketching algorithm can be seen as a public-coin one-way communication complexity protocol.

Here we exploit the fact that the common prefixes have the same parsing into phrases up to a certain point, and that the "mismatching" phrase has a back pointer to the portion of the text represented by the previous phrases: Alice and Bob can thus identify the mismatching symbol inside that phrase without further communication (see the "techniques" paragraph).

We extend the result stated in Theorem 5.1 so as to compute longest common prefixes when Bob holds a set of k strings $B_1, \ldots, B_k$, and the goal is to compute the maximal longest common prefix between A and any of the strings $B_1, \ldots, B_k$. This problem, denoted LCP^k, naturally captures the distributed scenario, where clients need to search for query strings in a text database stored at a server. To efficiently handle many queries we want to reduce both communication and rounds for each search. If we again view the strings as integers this is the *predecessor problem*. We generalize Theorem 5.1 to this scenario.

Theorem 5.2 *The* LCP^k *problem has a randomized public-coin $O(\lg z)$ round communication protocol with $O(\lg z \lg k + \lg \ell)$ communication complexity, where ℓ is the maximal common prefix between A and any one of $B_1, \ldots, B_k$, and z is the number of phrases in the LZ77 parse of this prefix.*

Compared to Theorem 5.1 we obtain the same number of rounds and only increase the total communication by an additive $O(\lg z \lg k)$ term. As $z \leq \ell$ the total communication increases by at most a factor $\lg k$.

The mentioned results hold only for LZ77 parses without self-references (see Sec. 5.2). We also show how to handle self-referential LZ77 parses and obtain the following bounds, where we add either extra $O(\lg \lg \ell)$ rounds or extra $O(\lg \lg \lg |A|)$ communication.

Theorem 5.3 *The* LCP *problem has a randomized public-coin protocol with*

1. $O(\lg z + \lg\lg \ell)$ *rounds and* $O(\lg \ell)$ *communication complexity,*
2. $O(\lg z)$ *rounds and* $O(\lg \ell + \lg\lg\lg |A|)$ *communication complexity*

where ℓ *is the length of the longest common prefix of* A *and* B*, and* z *is the number of phrases in the* self-referential *LZ77 parse of this prefix.*

Theorem 5.4 *The* LCP^k *problem has a randomized public-coin protocol with*

1. $O(\lg z + \lg\lg \ell)$ *rounds and* $O(\lg z \lg k + \lg \ell)$ *communication complexity,*
2. $O(\lg z)$ *rounds and* $O(\lg z \lg k + \lg \ell + \lg\lg\lg |A|)$ *communication complexity*

where ℓ *is the length of the maximal common prefix between* A *and any one of* $B_1, \ldots, B_k$*, and* z *is the number of phrases in the* self-referential *LZ77 parse of this prefix.*

Turning again to LZ77 parses without self-references we also show the following trade-offs between rounds and communication.

Theorem 5.5 *For any constant* $\varepsilon > 0$ *the* LCP *problem has a randomized public-coin protocol with*

1. $O(1)$ *rounds and* $O(z_A^{\varepsilon})$ *total communication where* z_A *is the number of phrases in the LZ77 parse of* A*,*
2. $O(\lg\lg \ell)$ *rounds and* $O(z^{\varepsilon})$ *total communication where* z *is the number of phrases in the LZ77 parse of the longest common prefix between* A *and* B*.*

We note that all the given bounds are in expectation. Using the standard transformation technique by Newman [75] all of the above results can be converted into private-coin results for bounded length strings: If the sum of the lengths of the strings is $\leq n$, then, Newman's construction adds an $O(\lg n)$ term in communication complexity, and only gives rise to 1 additional round.

Techniques

Our results rely on the following key idea. First, we want to perform an exponential search followed by a binary search over the LZ77-parses of the strings, to find the first phrase where Alice and Bob disagree. Then, the longest common prefix must end somewhere in the next phrase (see Figure 5.1). So Alice needs only to send the offset and length of her next phrase, and Bob can determine the longest common prefix with his string or strings (as proven in Lemma 5.6).

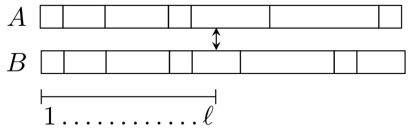


Figure 5.1: If the longest common prefix L of A and B has z phrases, then the first $z-1$ phrases of A, B, and L are identical.

To implement the idea efficiently, we use standard techniques that allow Alice and Bob to check if a specific prefix of their strings match using $O(1)$ communication, with only constant probability of error (we call this the EQUALITY problem). Similarly, if Bob holds k strings, they can check whether any of the k strings matches Alice's string with only $O(\lg k)$ communication, with constant error probability (we call this the MEMBERSHIP problem). This leads to the following $O(\lg z)$ round communication protocol.

1. Alice and Bob do an exponential search, comparing the first, two first, four first, etc, phrases of their strings using EQUALITY or MEMBERSHIP, until they find a mismatch.

2. Alice and Bob do a binary search on the last interval of phrases from Step 1, again, using EQUALITY or MEMBERSHIP, until they find their longest common prefix up to a phrase border.

3. Alice sends the offset and length of her next phrase, and Bob uses this to determine the longest common prefix.

To efficiently cope with errors in each step (which can potentially accumulate), we show how to extend techniques for *noisy binary search* [28] to an exponential search. Our new *noisy exponential search* only increases the number of rounds by a constant factor.

Paper outline

In Section 5.2, we review protocols for EQUALITY and MEMBERSHIP. Section 5.2 also contains a formal definition of the LZ77-parse of a string. In Section 5.3, we recall efficient techniques to handle errors using noisy binary search, and extend them to exponential search. In Section 5.4 we go on to prove Theorem 5.1 and Theorem 5.2. In Section 5.5, we show how to extend our results to self-referencing LZ77 (Theorems 5.3 and 5.4). Finally, in Section 5.6, we give the constant-round and near-constant round protocols promised in Theorem 5.5.

5.2 Definition and Preliminaries

A string S of length $n = |S|$ is a sequence of n symbols $S[1] \cdots S[n]$ drawn from an alphabet Σ. The sequence $S[i, j]$ is the *substring* of S given by $S[i] \cdots S[j]$ and, if $i = 1$, this substring is a *prefix* of S. Strings can be concatenated, i.e. $S = S[1, k]S[k + 1, n]$. Let $\textsc{Lcp}(A, B)$ denote the length of the longest common prefix between strings A and B. Also, denote by $[u]$ the set of integers $\{1, 2, \ldots, u\}$.

Communication Complexity Primitives

We consider the public-coin and private-coin randomized communication complexity models. In the public-coin model the parties share an infinite string of independent unbiased coin tosses and the parties are otherwise deterministic. The requirement is that for every pair of inputs the output is correct with probability at least $1 - \varepsilon$ for some specified $1/2 > \varepsilon > 0$, where the probability is on the shared random string. We note that any constant probability of success can be amplified to an arbitrarily small constant at the cost of a constant factor overhead in communication. In the private-coin model, the parties do not share a random string, but are instead allowed to be randomized using private randomness. Newman [75] showed that any result in the public-coin model can be transformed into private-coin model result at the cost of an additive $O(\lg \lg T)$ bits of communication, where T is the number of different inputs to the players. In our results this leads to an $O(\lg n)$ additive overhead, if we restrict our input to bounded length strings where the sum of the lengths of the strings is $\leq n$.

In the MEMBERSHIP problem, Alice holds a string A of length $|A| \leq n$, and Bob holds a set $\mathcal{B}$ of k strings. The goal is to determine whether $A \in \mathcal{B}$ (we assume that n and k are known to both parties) [83].

Lemma 5.1 *The* MEMBERSHIP *problem has a public-coin randomized* 1*-round communication protocol with* m *communication complexity and error probability* $k2^{-m}$*, for any integer* $m > 0$*.*

Proof sketch. Let $F : \{0,1\}^n \rightarrow \{0,1\}^m$ be a random linear function over $GF(2)$ where the coefficients of F are read from the shared random source (public coin). Alice applies F to A and sends the resulting m bits to Bob, i.e., she computes the product between a random $m \times n$ matrix and her string as a vector. Bob applies the same function to each of his strings, i.e., he computes the product between the same random matrix and each of his strings. If one of these products is the same as the one he received from Alice he sends a "1" to Alice indicating a match. This protocol has no false-negatives and by union bound the probability of a false-positive is at most $k2^{-m}$. For further details see e.g. [16,69].

In the EQUALITY problem, Alice holds a string A of size $|A| \leq n$, and Bob holds a string B. The goal is to determine whether $A = B$ (we assume that n is known to both parties). Lemma 5.1 implies the following corollary.

Corollary 5.1 *The* EQUALITY *problem has a public-coin randomized* 1*-round communication protocol with* m *communication complexity and error probability* 2^{-m}*, for any integer* $m > 0$*.*

Lempel-Ziv Compression

The *LZ77 parse* [91] of a string S of length n divides S into z substrings $f_1 f_2 \dots f_z$, called *phrases*, in a greedy left-to-right order. The i^{th} phrase f_i starting at position u_i is the longest substring having at least one occurrence starting to the left of u_i plus the following symbol. To compress S, we represent each phrase as a tuple $(s_i, l_i, \alpha_i) \in ([n] \times [n] \times \Sigma)$, such that s_i is the position of the previous occurrence, l_i is the length of the previous occurrence, and α_i is the symbol at position $u_i + l_i$. It follows that $s_1 = l_1 = 0, u_1 = 1, \alpha_1 = S[1]$ and we define $e_i = u_i + l_i$ for $i \in z$. That is, the i^{th} *phrase* of S ends at position e_i. We call the positions $e_1, \dots, e_z$ the *borders* of S and the substring $S[s_i, s_i + l_i - 1]$ is the *source* of the i^{th} phrase $f_i = S[u_i, u_i + l_i]$.

When a phrase is allowed to overlap with its source, the parse is *self-referential*. A more restricted version does not allow self-references and thus requires that $s_i + l_i \leq u_i$ for $i \in [z]$. We consider LZ77 parse without self-references unless explicitly stated. An LZ77 parse of S can be found greedily in $O(n \lg |\Sigma|)$ time from the suffix tree of S. It is easy to see that $z = \Omega(\lg n)$ if self-references are not allowed, while $z = \Omega(1)$ for self-referential parses.

5.3 Noisy Search

The *noisy binary search* problem is to find an element x_t among a sequence of elements $x_1, \dots, x_n$ where $x_i \leq x_{i+1}$ using only comparisons in a binary search. Each comparison may fail with a constant probability less than 1/2 and faults are independent.

Lemma 5.2 (Feige et al. [28, Theorem 3.2]) *For every constant* $Q < 1/2$*, we can solve the noisy binary search problem on* n *elements with probability at least* $1 - Q$ *in* $O(\lg(n/Q))$ *steps.*

We now show how to generalize the algorithm by Feige et al. to solve the *noisy exponential search* problem. That is, given a sequence $x_1, x_2, \dots$ where $x_i \leq x_{i+1}$ and an element x_ℓ find an element x_r such that $\ell \leq r \leq 2\ell$ using exponential search.

Lemma 5.3 *For every constant* $Q < 1/2$*, we can solve the noisy exponential search problem searching for* x_ℓ *with probability at least* $1 - Q$ *in* $O(\lg(\ell/Q))$ *steps.*

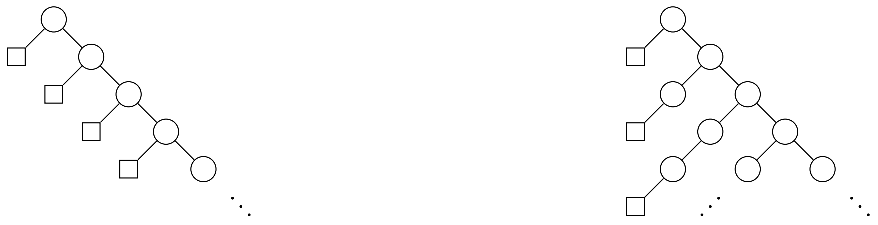


Figure 5.2: In the proof of Lemma 5.3, the decision tree for exponential search (left) is transformed to a fault-tolerant decision tree (right).

Proof In case of no errors we can find x_r on $O(\lg \ell)$ steps comparing x_ℓ and x_i for $i = 1, 2, 4, 8\dots$ until $x_i \geq x_\ell$. At this point we have $\ell \leq i \leq 2\ell$.

Consider the decision tree given by this algorithm. (See Figure 5.2). This tree is simply a path $v_0, v_1, v_2, \dots$ and when reaching vertex v_i the algorithm compares elements x_ℓ and x_{2^i}. In order to handle failing comparisons we transform this tree by adding a path with length l_i (to be specified later) as a child of vertex v_i. Denote such a path with p_i. The search now performs a walk in this tree starting in the root and progresses as follows: Reaching vertex v_i we first check if $x_\ell \geq x_{2^{i-1}}$ (i.e. repeat the previous comparison). If not, this reveals an earlier faulty comparison and we backtrack by moving to the parent. Otherwise, we check if $x_\ell \geq x_{2^i}$. If so we move to vertex v_{i+1}. Otherwise, we move to the first vertex on the path p_i. Reaching a vertex u on a path p_i we test if $x_\ell \geq x_{2^{i-1}}$ and if $x_\ell < x_{2^i}$. If both tests are positive, we move to the only child of u. Otherwise, this reveals an earlier faulty comparison and we backtrack by moving to the parent of u. When reaching a leaf on path p_i we terminate and report the element corresponding to v_i.

The search can be modeled as a Markov process. Assume that $\lceil \lg \ell \rceil = j$ and thus $j = O(\lg \ell)$ and direct all edges towards the leaf u on the path p_j. For every vertex $v \neq u$, exactly one adjacent edge is directed away from v and the remaining edges are directed towards v. Without loss of generality we can assume that the transition probability along an outgoing edge of a vertex is greater than $1/2$ and the transition probability along the remaining edges is less than $1/2$ (this probability can be achieved by taking the majority of $O(1)$ comparisons). Let b be the number of backward transitions and f the number of forward transitions. We need to show that $f - b \geq j + l_j$ with probability at least $1 - Q$ for $Q < 1/2$ implying that the search terminates in the leaf u. Setting $l_i = ic_1$ this follows after $c_2(\lg(2^j/Q)) = O(\lg(\ell/Q))$ rounds from Chernoff's bound [19] with suitable chosen constants c_1 and c_2.

5.4 Communication Protocol for LCP

We now present our protocol for the LCP problem without self-references. We consider the case with self-references in the next section. First, we give an efficient uncompressed output sensitive protocol that works for an arbitrary alphabet (Lemma 5.4). Secondly, we show how to encode LZ77 strings as strings from a small alphabet (Lemma 5.5) which allows us to efficiently determine the first phrase where Alice and Bob disagree. Thirdly, we show that given this phrase Alice and Bob can directly solve LCP (Lemma 5.6). Combining these results leads to Theorem 5.1. Finally, we generalize the results to the LCP^k case.

First we show how to solve the LCP problem with output-sensitive complexity for both the number of rounds and the amount of bits of communication.

Lemma 5.4 *Let A and B be strings over an alphabet Σ known to the parties. The* Lcp *problem has a public-coin randomized $O(\lg \ell)$-round communication protocol with $O(\lg \ell)$ communication complexity, where ℓ is the length of the longest common prefix between A and B.*

Proof Alice and Bob compare prefixes of exponentially increasing length using equality, and stop after the first mismatch. Let t be the length of the prefixes that do not match and observe that $t \leq 2\ell$. They now do a binary search on the interval $[0, t]$, using equality to decide if the left or right end of the interval should be updated to the midpoint in each iteration. The parties use Corollary 5.1 with $m = 2$, and new random bits from the shared random source for every equality check. Thus, the probability of a false-positive is at most $1/4$, and the faults are independent. Using Lemma 5.3 and Lemma 5.2 we get that we can solve the problem in $O(\lg(\ell/Q))$ rounds of communication with probability at least $1 - Q$ for any constant $Q < 1/2$.

Note that the size of the alphabet Σ does not affect the complexity of this protocol. Alice and Bob do however need to agree on how many bits to use per symbol in order to use the same number of random bits for the equality checks. Because Σ is known to the parties, they sort the alphabet and use $\lg |\Sigma|$ bits per symbol.

We move on to consider how to handle LZ77 compressed strings. Recall that the i^{th} phrase in the LZ77 parse of a string S is represented as a tuple (s_i, l_i, α_i) consisting of the source s_i, the length l_i of the source, and a symbol $\alpha_i \in \Sigma$. Observe that the LZ77 parse can be seen as a string where each tuple describing a phrase corresponds to a symbol in this string. Because we consider LZ77 without self-references a phrase is never longer than the sum of the lengths of the previous phrases and we can thus bound the number of bits required to write a phrase.

Lemma 5.5 *Let $Z_i = (s_1, l_1, \alpha_1), \ldots, (s_i, l_i, \alpha_i)$ be the first i elements in the LZ77 parse of a string S. Then, s_i and l_i can be written in binary with i bits.*

Proof Recall that e_j is the position in S of the last symbol in the j^{th} phrase. Since we have no self-references s_i and l_i are both no larger than e_{i-1} they can be written with $\lg e_{i-1}$ bits. By definition $u_j = e_{j-1}+1$. Therefore, $e_j = u_j + l_j = e_{j-1}+1+l_j \leq 2e_{j-1}+1$, and it follows that $e_{i-1} \leq 2e_{i-2} + 1 \leq \cdots \leq 2^i - 1$ since $e_1 = 1$.

We show that $\ell = \text{Lcp}(A, B)$ can be determined from $\text{Lcp}(Z_A, Z_B)$ with only one round and $O(\lg \ell)$ communication, where Z_A and Z_B are the respective LZ77 parses of A and B.

While a LZ77 parse of a string is not necessarily unique, in this case, we can assume that the parties as part of the protocol agree deterministically upon their same decisions on LZ77-compression algorithm (e.g. taking always the leftmost source when there are multiple possibilities). This ensures that we obtain the same parsing for equal strings, independently and without any communication.

Lemma 5.6 *Let A and B be strings and let Z_A and Z_B be their respective LZ77 parses. If Alice knows A and Bob knows B and the length of the longest common prefix $\text{Lcp}(Z_A, Z_B)$, then they can determine the length $\ell = \text{Lcp}(A, B)$ of the longest common prefix of A and B in $O(1)$ rounds and $O(\lg \ell)$ communication.*

Proof First, Z_A and Z_B themselves can be seen as strings over the special alphabet $\Sigma' \equiv ([n] \times [n] \times \Sigma)$ of tuples. Letting $z = \text{Lcp}(Z_A, Z_B)$, these LZ77 parses of A and B are identical up until but no longer than their z^{th} tuple. Now, let $\ell = \text{Lcp}(A, B)$. Let a_i and b_i denote the i^{th} phrase border in the LZ77 parse of A and B respectively. Observe that $A[1, a_z] = B[1, b_z]$ but $A[1, a_{z+1}] \neq B[1, b_{z+1}]$ because of how we choose

z and, thus, $a_z = b_z \leq \ell < a_{z+1}, b_{z+1}$. Let s_{z+1}, l_{z+1} be the source and length of the $(z+1)^{th}$ phrase in Z_A. Alice sends s_{z+1}, l_{z+1} to Bob in one round with $O(\lg a_z) = O(\lg \ell)$ bits of communication since $s_{z+1}, l_{z+1} \leq a_z$. At this point, it is crucial to observe that Bob can recover $A[1, a_{z+1}]$ by definition of LZ77 parsing: he deduces that $A[1, a_{z+1}] = B[1, b_z]B[s_{z+1}, s_{z+1} + l_{z+1}]$, from which he can compute $\textsc{Lcp}(A[1, a_{z+1}], B[1, b_{z+1}]) = \textsc{Lcp}(A, B)$.

We can now combine Lemmas 5.4, 5.5, and 5.6 to prove Theorem 5.1. Alice and Bob construct the LZ77 parse of their respective strings and interpret the parse as a string. Denote these strings by Z_A and Z_B. They first use Lemma 5.4 to determine $\textsc{Lcp}(Z_A, Z_B)$, where the parties decide to use $2i + \lg|\Sigma|$ random bits for the equality check of the i^{th} symbols (from Σ'), which suffices by Lemma 5.5. Then they apply Lemma 5.6 to determine $\textsc{Lcp}(A, B)$. In conclusion this proves Theorem 5.1.

5.4.1 The $\textsc{Lcp}^k$ *case*

In this section we generalize the result on LCP to the case where Bob holds multiple strings. Here, Alice knows a string A and Bob knows strings $B_1, \ldots, B_k$, where all strings are drawn from an alphabet Σ known to the parties.

The main idea is to substitute the equality-tests by membership queries. We first generalize Lemma 5.4 to the $\textsc{Lcp}^k$-case.

Lemma 5.7 *The* $\textsc{Lcp}^k$*-problem has a randomized public-coin* $O(\lg \ell)$*-round communication protocol with* $O(\lg \ell \lg k)$ *communication complexity, where* ℓ *is the length of the maximal longest common prefix between* A *and any* B_i*.*

Proof Along the same lines as the proof of Lemma 5.4, Alice and Bob perform membership-queries on exponentially increasing prefixes, and then, perform membership-queries to guide a binary search. They use Lemma 5.1 with $m = 2\lg k$, and exploit shared randomness as in the previous case. Again, the probability of a false positive is $\leq 1/4$, and the faults are independent. Thus Lemma 5.3 and Lemma 5.2 gives us an $O(\lg \ell/Q)$ round communication protocol with total error probability $1 - Q$ for any constant choice of $Q < 1/2$.

Since there are $O(\lg \ell)$ rounds in which we spend $O(\lg k)$ communication, the total communication becomes $O(\lg \ell \lg k)$.

We go on to show that the maximal $\textsc{Lcp}(A, B_i)$ can be determined from solving $\textsc{Lcp}^k$ on Z_A and $\{Z_{B_1}, \ldots, Z_{B_k}\}$ with only one additional round and $O(\lg n)$ communication.

Lemma 5.8 *Let* $Z_A, Z_{B_1}, \ldots, Z_{B_k}$ *be the LZ77 parses of the strings* $A, B_1, \ldots, B_k$*. If Alice knows* A*, and Bob knows* $B_1, \ldots, B_k$ *and the length of the maximal longest common prefix between* Z_A *and any* Z_{B_i}*, they can find* $\max_i \textsc{Lcp}(A, B_i)$ *in* $O(1)$ *rounds and* $O(\lg n)$ *communication.*

Proof In this case, Bob holds a set, $\mathcal{B}'$, of at least one string that matches Alice's first z phrases, and no strings that match Alice's first $z + 1$ phrases. Thus, if Alice sends the offset and length of her next phrase, he may determine $\textsc{Lcp}(A, B_i)$ for all strings $B_i \in \mathcal{B}'$. Since the maximal $\textsc{Lcp}$ among $B_i \in \mathcal{B}'$ is indeed the maximal over all $B_i \in \mathcal{B}$, we are done.

Combining Lemma 5.7 and Lemma 5.8 we get Theorem 5.2.

5.5 Self-referencing LZ77

We now consider how to handle LZ77 parses with self-references. The main hurdle is that Lemma 5.5 does not apply in this case as there is no bound on the phrase length except the length of the string. This becomes a problem when the parties need to agree on the number of bits to use per symbol when computing LCP of Z_A and Z_B, but also when Alice needs to send Bob the source and length of a phrase in order for him to decide $\textsc{Lcp}(A, B)$.

First we show how Alice and Bob can find a bound on the number of random bits to use per symbol when computing $\textsc{Lcp}(Z_A, Z_B)$.

Lemma 5.9 *Bob and Alice can find an upper bound ℓ' on the length ℓ of the longest common prefix between A and B where*

1. *$\ell' \leq \ell^2$ using $O(\lg\lg\ell)$ rounds and $O(\lg\lg\ell)$ total communication,*
2. *$\ell' \leq |A|^2$ using $O(1)$ rounds and $O(\lg\lg\lg|A|)$ total communication.*

Proof Part (1): Alice and Bob do a double exponential search for ℓ and find a number $\ell \leq \ell' \leq \ell^2$ using equality checks on prefixes of their uncompressed strings in $O(\lg\lg\ell)$ rounds. Again, at the cost of only a constant factor, we apply Lemma 5.3 to deal with the probability of false positives.

Part (2): Alice sends the minimal i such that $|A| \leq 2^{2^i}$ thus $i = \lceil\lg\lg|A|\rceil$ can be written in $O(\lg\lg\lg|A|)$ bits. Alice and Bob can now use $n = 2^{2^i}$ as an upper bound for ℓ, since $\ell \leq |A| \leq 2^{2^i} < |A|^2$.

Assume that Alice and Bob find a bound ℓ' using one of those techniques, then they can safely truncate their strings to length ℓ'. Now they know that every symbol in Z_A and Z_B can be written with $O(\lg\ell' + \lg|\Sigma|)$ bits, and thus, they agree on the number of random bits to use per symbol when doing equality (membership) tests. Using Lemma 5.4 they can now find the length of the longest common prefix between Z_A and Z_B in $O(\lg\ell)$ rounds with $O(\lg\ell)$ communication.

We now show how to generalize Lemma 5.6 to the case of self-referential parses.

Lemma 5.10 *Let A and B be strings and let Z_A and Z_B be their respective* self-referential *LZ77 parses. If Alice knows A and Bob knows B and the length of the longest common prefix between Z_A and Z_B, then they can determine the length ℓ of the longest common prefix of A and B in*

1. *$O(1 + \lg\lg\ell)$ rounds and $O(\lg\ell)$ communication,*
2. *$O(1)$ rounds and $O(\lg\ell + \lg\lg\lg|A|)$ communication.*

Proof Let s_i, e_i and l_i be the respective source, border and length of the i^{th} phrase in Z_A. The proof is the same as in Lemma 5.6 except that the length l_{z+1} of the $(z+1)^{th}$ phrase in Z_A that Alice sends to Bob is no longer bounded by ℓ.

There are two cases. If $l_{z+1} \leq 2e_z$, then $l_{z+1} \leq 2\ell$, and Alice can send l_{z+1} to Bob in one round and $O(\lg\ell)$ bits and we are done.

If $l_{z+1} > 2e_z$ then the source of the $(z+1)^{th}$ phrase must overlap with the phrase itself and thus the phrase is periodic with period length at most e_z and has at least 2 full repetitions of its period. Alice sends the starting position of the source of the phrase s_{i+1} along with a message indicating that we are in this case to Bob in $O(\lg\ell)$ bits. Now Bob can check if they agree on next $2e_z$ symbols. If this is not the case, he has also determined $\textsc{Lcp}(A, B)$ and we are done. Otherwise, they agree on the next $2e_z$ symbols and therefore the $(z+1)^{th}$ phrase in the parse of both A and B is periodic and they have the same period. What remains is to determine which phrase that is shorter. Let l_a and l_b denote the lengths of respectively Alice's and Bob's next phrase. Then (1) follows from Alice and Bob first computing a number $\ell' \leq \ell^2$ using a double exponential search and equality checks in $O(\lg\lg\ell)$ rounds and total communication. Clearly either l_a or l_b must be shorter than ℓ' and the party with the shortest phrase sends its length to the other party in $O(\lg\ell)$ bits and both can then determine $\textsc{Lcp}(A, B)$. To get the result in (2) Alice sends the smallest integer i such that $l_a \leq 2^{2^i}$ in a single round and $O(\lg\lg\lg|A|)$ bits of communication. Bob then observes that if $l_b \leq 2^{2^{i-1}}$, then $l_b = \ell$ and he sends ℓ to Alice using $O(\lg\ell)$ bits. If $l_b > 2^{2^i}$ then $l_a = \ell$ and he informs Alice to send him l_a in $O(\lg\ell)$ bits. Finally, if $2^{2^i-1} < l_b$ and $l_a \leq 2^{2^i} \leq \ell^2$ he sends l_b to Alice using $O(\lg\ell)$ bits.

Theorem 5.3 now follows from Lemmas 5.4, 5.9, and 5.10.

5.5.1 $\textsc{Lcp}^k$ *in the self-referential case.*

Finally, we may generalize Theorem 5.2 to the self-referential case. Substituting equality with membership, we may directly translate Lemma 5.9:

Lemma 5.11 *Bob and Alice can find an upper bound on the length ℓ' of the maximal longest common prefix between A and $B_1, \ldots, B_k$ where*

1. *$\ell' \leq \ell^2$ using $O(\lg\lg\ell)$ rounds and $O(\lg\lg\ell\lg k)$ total communication,*
2. *$\ell' \leq |A|^2$ using $O(1)$ round and $O(\lg\lg\lg|A|)$ total communication.*

Using the lemma above, we can generalize Corollary 5.10 to the $\textsc{Lcp}^k$-case.

Lemma 5.12 *Let A and $B_1, \ldots, B_k$ be strings, and let Z_A and Z_{B_i} be their respective* self-referential *LZ77 parses. If Alice knows A and Bob knows $B_1, \ldots, B_k$ and Bob knows the length of the maximal longest common prefix between Z_A and any Z_{B_i}, then they can determine ℓ in*

1. *$O(1 + \lg\lg\ell)$ rounds and $O(\lg\ell\lg k)$ communication,*
2. *$O(1)$ rounds and $O(\lg\ell\lg k + \lg\lg\lg|A|)$ communication.*

Proof tweak. Alice and Bob have already found a common prefix of size e_z – question is whether a longer common prefix exists. As before, if Alice's next phrase is shorter than $2e_z$, she may send it. Otherwise, she sends the offset, and indicates we are in this case. Now, Bob can check if *any* of his strings agree with Alice's on the next $2e_z$ symbols. If none do, we are done. If several do, he forgets all but the one with the longest $(z+1)^{st}$ phrase, and continue as in the proof of Corollary 5.10.

Theorem 5.4 now follows from the combination of Lemmas 5.11 and 5.12.

5.6 Obtaining a Trade-Off via D-ary Search.

We show that the technique of Buhrman et al. [16], to compute $\textsc{Lcp}$ of two strings of length n in $O(1)$ rounds and $O(n^{\varepsilon})$ communication, can be used to obtain a compressed communication complexity. Note that we again consider LZ77 compression without self-references. We first show the following generalization of Lemma 5.4.

Lemma 5.13 *Let A and B be strings over an alphabet Σ known to the parties. The* $\textsc{Lcp}$ *problem has a public-coin randomized communication protocol with*

1. *$O(1)$ rounds and $O(|A|^{\varepsilon})$ communication*

2. $O(\lg\lg\ell)$ rounds and $O(\ell^\varepsilon)$ communication

where ℓ is the length of the longest common prefix between A and B, and $\varepsilon > 0$ is any arbitrarily small constant.

Proof Assume the parties agree on some parameter C and have previous knowledge of some constant ε' with $0 < \varepsilon' < \varepsilon$ (i.e. ε' and ε are plugged into their protocol). They perform a D-ary search in the interval $[-1, C]$ with $D = C^{\varepsilon'}$. In each round, they split the feasible interval into D chunks, and perform equality tests from Corollary 5.1 with $m = 2\lg(D/\varepsilon')$ on the corresponding prefixes. The feasible interval is updated to be the leftmost chunk where the test fails. There are $\lg_D C = 1/\varepsilon' = O(1)$ rounds. The communication per round is $2D\lg(D/\varepsilon')$ and the total communication is $1/\varepsilon' \cdot 2D\lg(D/\varepsilon') = O(C^{\varepsilon'}\lg C)$. The probability of a false positive for the equality test is 2^{-m}, and thus, by a union bound over D comparisons in each round and $1/\varepsilon'$ rounds, the combined probability of failure becomes $1/4$.

1. Alice sends $|A|$ to Bob in $\lg|A| = O(|A|^\varepsilon)$ bits and they use $C = |A|$. The total communication is then $O(C^{\varepsilon'}\lg C) = O(|A|^\varepsilon)$ with $O(1)$ rounds.

2. Alice and Bob use Lemma 5.9 to find an ℓ' such that $\ell \leq \ell' \leq \ell^2$ in $O(\lg\lg\ell)$ rounds and communication. They run the D-ary search protocol where $\varepsilon' < \varepsilon/4$, setting $C = \ell'$. The extra communication is $O(C^{\varepsilon'}\lg C) = O(\ell^\varepsilon)$. ■

We can now combine Lemmas 5.13, 5.5, and 5.6 to prove Theorem 5.5. Alice and Bob construct the LZ77 parse of their respective strings and interpret the parses as strings, denoted by Z_A and Z_B. They first use Lemma 5.13 to determine $\textsc{Lcp}(Z_A, Z_B)$, and then Lemma 5.6 to determine $\textsc{Lcp}(A, B)$. The parties use $2i + \lg|\Sigma|$ random bits for the i^{th} symbol, which suffices by Lemma 5.5. This enables them to apply Lemma 5.13 to Z_A and Z_B. In conclusion this proves Theorem 5.5.

We note without proof that this trade-off also generalizes to self-referential parses by paying an additive extra $O(\lg\lg\lg|A|)$ in communication for Theorem 5.5 (1) and an additive $O(\lg\ell)$ communication cost for Theorem 5.5 (2). The same goes for $\textsc{Lcp}^k$ where the communication increases by a factor $O(\lg k)$ simply by increasing m by a factor $\lg k$ and using the techniques already described.

CHAPTER 6

FAST DYNAMIC ARRAYS

Philip Bille Anders Roy Christiansen Mikko Berggreen Ettienne Inge Li Gørtz

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 $O(n^{\varepsilon})$ for $\varepsilon > 0$ 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 [15, 25, 35, 36, 46, 57, 58, 82] 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 [25] gives a data structure that matches the lower bound of $\Omega(\lg n/\lg\lg n)$ showed by Fredman and Saks [36]. 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* [46] 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 [35]. 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 ≥ 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 $a \mod b = a - qb$ where q is the largest integer such that $q \cdot b \leq a$. Define $A[i,j] \mod w$ for some integer w to be the elements $A[i \mod w], A[(i+1) \mod w], \ldots, A[j \mod w]$, i.e. $A[4,7] \mod 5 = A[4], A[0], A[1], A[2]$. 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 [46] 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. The capacity $\mathrm{cap}(v)$ of a node v is $w^{\mathrm{height}(v)+1}$ and the *size* $size(v) = |A(v)| \leq \mathrm{cap}(v)$ is the number of elements in the sequence $A(v)$. For a node v with children $c_1, c_2, \ldots, c_w$, $A(v)$ is a circular shift of the

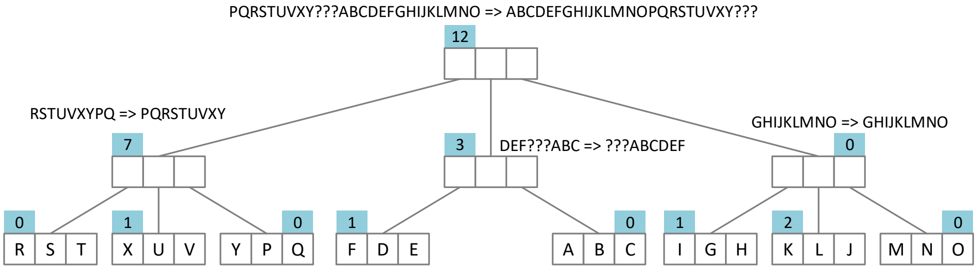


Figure 6.1: An illustration of a tiered vector with $l = w = 3$. The elements are letters, and the tiered vector represents the sequence ABCDEFGHIJKLMNOPQRSTUVX. The elements in the leaves are the elements that are actually stored. The number above each node is its offset. The strings above an internal 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.

concatenation of the elements represented by its children, $A(c_1) \cdot A(c_2) \cdot \ldots \cdot A(c_w)$. 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 access(v, i) returns the element $A(v)[i]$ and is recursively computed as follows:

v is internal: Compute $i' = (i + \text{off}(v)) \mod \text{cap}(v)$, let v' be the $\lfloor i' \cdot w/\text{cap}(v) \rfloor^{th}$ child of v and return the element access$(v', i' \mod \text{cap}(v'))$.

v is leaf: Compute $i' = (i + \text{off}(v)) \mod w$ 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 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 access(v, i, m) report the elements $A(v)[i \ldots i{+}m{-}1]$ in order. The operation can recursively be defined as:

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

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

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 $c_l, c_{l+1}, \ldots, c_r$ that covers the range of elements $A(v)[i, i + m - 1]$ to be shifted. All elements in $c_{l+1}, \ldots, c_{r-1}$ 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 $\mathsf{shift}(v, e, i, m)$ 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 $i_l = (i+\text{off}(v)) \mod \text{cap}(v)$, and let $i_r = (i_l+m) \mod \text{cap}(v)$. The children of v that must be updated are in the range $[\lfloor i_l \cdot w/\text{cap}(v)\rfloor, \lfloor i_r \cdot w/\text{cap}(v)\rfloor] \mod w$ call these $c_l, c_{l+1}, \ldots, c_r$. Let $e_l = \mathsf{shift}(c_l, e, i_l, \min(m, \text{cap}(c_l) - i_l))$. Let $e_i = \mathsf{update}(c_i, size(c) - 1, e_{i-1})$ and set $\text{off}(c_i) = (\text{off}(c_i) - 1) \mod \text{cap}(c)$ for $c_i = c_{l+1}, \ldots, c_{r-1}$. Finally call $\mathsf{shift}(c_r, e_{r-1}, 0, i_r \mod \text{cap}(c_r))$.

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

An insertion $\mathsf{insert}(i, e)$ can then be performed as $\mathsf{shift}(root, e, i, size(root) - i - 1)$. 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 $\mathsf{delete}(i)$ can be performed as $\mathsf{shift}(r, \perp, 0, i)$ followed by an update of the root's offset to $(\text{off}(r) + 1) \mod \text{cap}(r)$.

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 $w = \Theta(n^\varepsilon)$ where $\varepsilon = 1/l$ 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 ([46]) *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 $\Theta(n^{1-\varepsilon})$ 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 $O(n^{1-\varepsilon})$. At most two leaves are only partially full, therefore the total space is now again reduced to $O(n^{1-\varepsilon})$. 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 $n^{\varepsilon(\text{height}(v)+1)}$ and the number of bits needed to store all the offsets is therefore $\sum_{i=0}^{l} n^{1-i\varepsilon} \lg(n^{i\varepsilon}) = \lg(n) \sum_{i=0}^{l} i\varepsilon n^{1-i\varepsilon} \approx \varepsilon n^{1-\varepsilon} \lg(n)$ (for sufficiently large n). Thus the $n^{1-\varepsilon}$ offsets can be stored in approximately $\varepsilon n^{1-\varepsilon}$ words giving a space reduction of a constant factor ε. 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 the 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[1]. 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[2] added/deleted, though in the case of "vector" only 10.000 elements were inserted/deleted to make the experiments terminate in reasonable time.

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 $O(\lg n)$. On the other hand, we would expect insertion/deletion to be significantly slower since it runs in $O(n^{1/l})$ time compared to $O(\lg n)$ (where $l = 4$ in these tests). We see our 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 $O(\lg n)$ time for

[1] 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

[2] In order to not impact timing, a simple access pattern has been used instead of a normal pseudo-random generator.

	tiered vector	*multiset*	*s / t*	*vector*	*v / t*
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 μs	1.65 μs	0.92	21675.49 μs	12082.33
insert in end	7.28 ns	242.90 ns	33.38	2.93 ns	0.40
successor	0.55 μs	1.53 μs	2.75	0.36 μs	0.65
delete	1.92 μs	1.78 μs	0.93	21295.25 μs	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.

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.

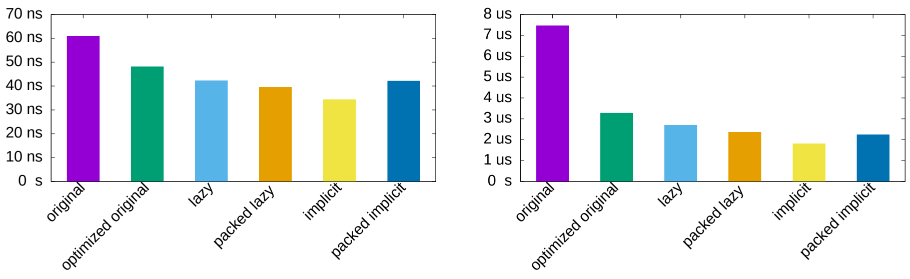
To sum up, the tiered vectors performs better than multiset on all tests but insertion, where it performs only slightly worse.

6.6.2 *Tiered Vector Variants*

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

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



(a) *access*

(b) *insert*

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

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.

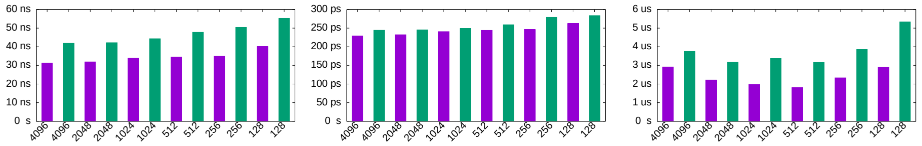
Packed Implicit Packing the offsets array could lead to better cache performance due to the smaller memory footprint and therefore yield better overall performance. As can be seen on Figure 6.2, the smaller memory footprint did not improve the performance in practice. The simple reason for this is that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

In 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.



(a) *access*

(b) *range access*

(c) *insert*

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

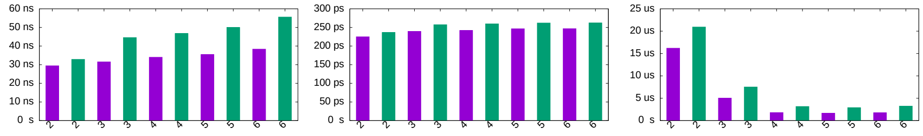
6.6.4 Height Experiments

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.



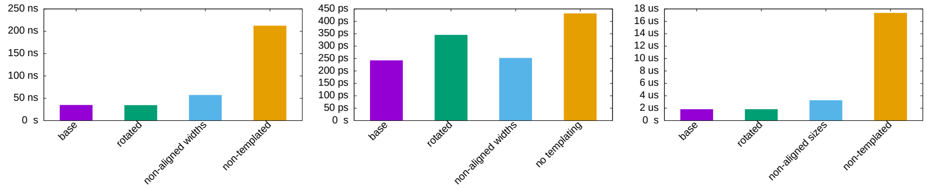
(a) *access(i)* (b) *access(i,m)* (c) *insert*

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

6.6.5 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 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 the ensure that there are no significant



(a) *access* (b) *range access* (c) *insert(i,x)*

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

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 the 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 involve big instances of the dynamic array problem.

CHAPTER A

APPENDIX

A.1 Mergable Dictionaries

In this section we show how to extend the mergeable dictionary to support shifts. We first give a simple reduction showing that the requirement that the sets are disjoint can be lifted without affecting the asymptotic complexity as long as $n \leq \mathcal{U}$. We then move on to consider the internals of the data structure and the changes needed to support shifts.

Disjointness

We use the following simple reduction to avoid breaking the disjointness when shifting the sets. Let $y_1 < \ldots < y_n$ be the values stored in the singleton sets. Consider instead a problem from the universe $\{1, 2, \ldots, \mathcal{U}^2\}$ and and let $f : \mathcal{U} \to \mathcal{U}^2$ be the function $f(x) = (x-1)(\mathcal{U}+1)$ and let $g : \mathcal{U}^2 \to \mathcal{U}$ be the function $g(x) = \lceil x/(\mathcal{U}+1) \rceil + 1$. We now let the values of the singleton sets be $z_i = f(y_i)$ for $i = 1, \ldots, n$. Replace any $\texttt{shift}(G, x)$ by $\texttt{shift}(G, x \cdot \mathcal{U})$. Because $z_i \not\equiv z_j \pmod{\mathcal{U}}$ the sets will always remain disjoint. Replace any $(A, B) \leftarrow \texttt{split}(G, x)$ by $(A, B) \leftarrow \texttt{split}(G, f(x))$ and when reporting all values $s_1, s_2, \ldots$ of a set S report instead $g(s_1), g(s_2), \ldots$.

Biased Skip Lists

The mergeable dictionary given by Iacono and Özkan [51] uses biased skip lists to maintain every set $G \in \mathcal{G}$. The skip list was first introduced by Pugh [81] as a probabilistic alternative to dynamic binary search trees supporting the same set of operations in $O(\lg n)$ expected time. Munro et al. [71] show how to achieve worst-case bounds, Bagchi et al. [4] generalises to biased skip lists with finger searches and Iacono and Özkan show how to support finger split, finger reweight and finger join (we give the details of these operations later).

A biased skip list (BSL) S stores an ordered set X where each element $x \in X$ corresponds to a node $x \in S$. Node x has *weight* $w_x \geq 1$ and some integral *height*, $h_x \geq 1$, and the *rank* of x is defined as $r_x = \lfloor \lg w_x \rfloor$. The *height* of S denoted $H(S)$ is the maximal height amongst the nodes in S. Each node x is represented by a linked list of length $h_x + 1$ called the *tower* of that node. The *level-j predecessor* of a node x is the maximal node $y < x$ with height $h_y \geq j$ and the *level-j successor* is defined symmetrically. Let y be the level-k predecessor (successor) of x then the k^{th} element in the tower of x contains pointers to the k^{th} element in the tower of y with one exception: If y is the predecessor (successor) of x in the set X then the pointer at level $\min\{h_x, h_y\} - 1$ is nil and the pointers below this level are undefined. Assume that there are sentinel nodes of height $H(S)$ at the beginning (with key $-\infty$) and the end (with key ∞) of S. The sentinels are not actually needed but eases the explanation. Two distinct elements x and y are called *consecutive* if and only if they are linked together in S. We orient the pointers so that the skip list stores items in left-to-right order and the node levels progress from bottom to top. There are three invariants for biased skip lists:

(I1) Each item x has height $h_x \geq r_x$.

(I2) There are never more than 6 consecutive items of any height.

(I3) For each node x and for all $r_x < i \leq h_x$ there are at least 2 nodes of height $i - 1$ between x and any consecutive node of height at least i.

To support the shift operation, we extend the biased skip list by storing an integral offset with each of the pointers between the towers of the nodes in S initially set to 0. Consider any sequence of shift, join and merge operations starting from n singleton biased skip lists and track one element x through this sequence. If x has been affected by l shift operations, then the value of x should be $o_x + k_1 + k_2 + \cdots + k_l$ where k_i the integer value shifted by in the i^{th} shift and o_x is the original value of x. Denote by x_s the *shift* of x (at some point in time). The *accumulated offset* of a node x is the sum of the offsets of the pointers on any path from a sentinel to x. We introduce the following invariant to support lazy shifts of biased skip lists:

(I4) The accumulated offset of any path from x to y is $y_s - x_s$.

It follows immediately that we can search for an element using standard skip list search [81] and report its value correctly by keeping track of the accumulated offset while navigating the pointers of a biased skip lists. Similarly, we report the correct value of all elements by keeping track of accumulated offset while traversing a biased skip list.

Trivially, invariant **(I4)** is true when creating a singleton BSL with the offset of all pointers set to 0. We now show how to shift a biased skip list:

Shift(S, k) To shift S by k we increment the offset on the right-pointers of the left sentinel and the left pointers on the right sentinel by k and decrement the pointers with opposite direction by k. Let s_x be the shift of some element $x \in S$ before this shift then the shift of x after this operation is $s_x + k$. Any path starting from a sentinel has an increased accumulated offset of k and it follows that invariant **(I4)** is still true after the shift. The shift operation does not change the structure of S and therefore it does not invalidate the invariants **(I1 - I3)**. The operation takes linear time in the height of the sentinels which is $O(\lg n)$ [71].

The operations reweight, join and split involves splicing and splitting pointers and changing the height of nodes which again involves splicing and splitting pointers. We refer the reader to [51] for the details on the operations including complexities and proofs of correctness. We now give the details on how to maintain the offsets of pointers during splicing and splitting which is the only change required for the operations.

Delete Pointer Deleting a pointer between any two towers does not invalidate **(I4)** and thus no offsets need to change.

New Pointer Whenever a new pointer $x \to y$ between some level in the towers of x and y is introduced set the offset of that pointer to $y_s - x_s$ where y_s and x_s is the shift of y and x respectively.

The new pointer introduces a path between x and y which has accumulated offset $y_s - x_s$. Thus if **(I4)** was true before, it remains true.

Merge

Two biased skip lists A and B are merged using a folklore version of merging called *segment merging*. Assume w.l.o.g. that $\min(A) < \min(B)$. If we consider the elements of A and B in increasing order, the observation is that we can split A and B into ordered segments $A_1, A_2, \dots, A_i$ and $B_1, B_2, \dots, B_j$ where $j \in [i-1; i]$, and $\max(A_i) < \min(A_j)$ and $\max(B_i) < \min(B_j)$ for $i < j$ and $\max(A_i) < \min(B_i) < \max(B_i) < \min(A_{i+1})$ for all i. The merge strategy is then to find the min and max element of each segment, extract the segments using split operations and merge all the segments using join operations.

Iacono & Özkan [51] find the node that separates the segments A_i and A_{i+1} by searching for the successor of $\max(B_i)$ in A starting from the node $f = \min(A_i)$ and similarly for B. Thus they perform a finger search starting from f (initially the finger is the left sentinel). By keeping pointers to all these fingers, the splits and joins are also done as finger operations. This is the key to obtaining the desired complexity bound. Since the merge starts from a left sentinel it is easy to keep track of the accumulated offset while navigating the skip lists.

Finally, the amortized complexities of the search, join, split and reweight are not affected by shifts as they solely depend on the internal structure of the biased skip lists and not the values stored in the nodes [4, 51]. The amortized complexity of the merge operation is proven using the following potential function:

Let D_i be the data structure containing the dynamic collection of disjoint sets $\mathcal{G}^{(i)} = \{G_1^{(i)}, G_2^{(i)}, \dots\}$ after the i^{th} operation. Let $G = \{x_1, x_2, \dots\}$ be the values of the elements of G in increasing order then

$$\phi(G) = \sum_{i=1}^{|G|} (\lg(x_i - x_{i-1}) + \lg(x_{i+1} - x_i))$$

where we define $x_1 - x_0 = 1$ and $x_{|G|+1} - x_{|G|} = 1$. Then the potential after the i^{th} operation is

$$\Phi(D_i) = c_d \cdot \sum_j \phi(S_j^{(i)})$$

where c_d is a constant [51]. I.e. the potential is the sum of the logarithm to the gaps between the elements in every set. This measure depends on the relative difference between the values of the elements inside every set and not the absolute value, thus the potential is not affected by a shift. Therefore the amortized cost of the shift operation is $O(\lg \mathcal{U})$ and the complexities of the other BSL operations are unchanged which concludes the proof of Theorem 4.5.

BIBLIOGRAPHY

[1] `www.gzip.org`.

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *Proceedings of the 11th Annual Symposium on Discrete Algorithms*. Citeseer, 2000.

[3] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. 35th FOCS*, pages 714–721. IEEE Computer Society, 1994.

[4] A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, May 2005.

[5] Z. Bar-Yossef, T. S. Jayram, R. Krauthgamer, and R. Kumar. The sketching complexity of pattern matching. In *Proc. 8th RANDOM*, pages 261–272, 2004.

[6] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *Proc. 18th ESA*, LNCS, vol 6346, pages 427–438. Springer Berlin Heidelberg, 2010, (appendix H.3 can be found at http://www.itu.dk/people/pagh/papers/prefix.pdf).

[7] D. Belazzougui, F. Cunial, T. Gagie, N. Prezza, and M. Raffinot. Composite repetition-aware data structures. In *Proc. 26st CPM*, LNCS, vol 9133, pages 26–39. Springer Cham, 2015.

[8] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. O. Pereira, S. J. Puglisi, and Y. Tabei. Queries on lz-bounded encodings. In *Proc. DCC 2015*, pages 83–92. IEEE Computer Society, 2015.

[9] D. Belazzougui, J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lempel-Ziv Decoding in External Memory. In *Proceedings of 15th SEA*, pages 63–74, 2016.

[10] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. 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.

[11] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. Time-space trade-offs for lempel-ziv compressed indexing. *Theoretical Computer Science*, 713:66 – 77, 2018.

[12] P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space trade-offs for longest common extensions. In *Proc. 23rd CPM*, LNCS, vol 7354. Springer Berlin Heidelberg, 2012.

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

[14] D. Breslauer and Z. Galil. Real-time streaming string-matching. *ACM Trans. Algorithms*, 10(4):22:1–22:12, 2014.

[15] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro, and R. 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.

[16] H. Buhrman, M. Koucký, and N. Vereshchagin. Randomised individual communication complexity. In *Proc. 23rd CCC*, pages 321–331, 2008.

[17] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proc. 27th SoCG*, pages 1–10. ACM, 2011.

[18] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Information Theory*, 51(7):2554–2576, 2005.

[19] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Ann. Math. Stat.*, 23:493–507, 1952.

[20] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Inf. Syst.*, 61:1–23, 2016.

[21] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. 19th SPIRE*, LNCS, vol 7608, pages 180–192. Springer Berlin Heidelberg, 2012.

[22] R. Cole and R. Hariharan. Approximate string matching: A simpler faster algorithm. *SIAM Journal on Computing*, 31(6):1761–1782, 2002.

[23] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32 – 53, 1986.

[24] M. Crochemore, L. Ilie, and W. F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *Proceedings of 2008 DCC*, pages 482–488, 2008.

[25] P. 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.

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

[27] M. Farach and M. Thorup. String Matching in Lempel—Ziv Compressed Strings. *Algorithmica*, 20(4):388–404, 1998.

[28] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, Oct. 1994.

[29] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st FOCS*, pages 390–398. IEEE Computer Society, 2000.

[30] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th SODA*, pages 269–278. Society for Industrial and Applied Mathematics, 2001.

[31] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[32] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007.

[33] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proceedings of 23rd ESA*, pages 533–544. 2015.

[34] J. Fischer, T. I, and D. Köppl. Lempel Ziv Computation In Small Space (LZ-CISS). In *Proceedings of 26th CPM*, pages 172–184, 2015.

[35] G. N. Frederickson. Implicit data structures for the dictionary problem. *J. ACM*, 30(1):80–94, Jan. 1983.

[36] 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.

[37] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[38] 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.

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

[40] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. 11th LATIN*, LNCS, vol 8392, pages 731–742. Springer Berlin Heidelberg, 2014.

[41] T. Gagie, P. Gawrychowski, and S. J. Puglisi. Approximate pattern matching in LZ77-compressed texts. *Journal of Discrete Algorithms*, 32:64–68, 2015.

[42] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in bwt-runs bounded space. *arXiv preprint arXiv:1705.10382*, 2017.

[43] T. Gagie and S. J. Puglisi. Searching and indexing genomic databases via kernelization. *Frontiers in Bioengineering and Biotechnology*, 3:12, 2015.

[44] P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: Fast, Simple, and deterministic. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6942:421–432, 2011.

[45] P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Łącki, and P. Sankowski. Optimal dynamic strings. *arXiv preprint arXiv:1511.02612*, 2015.

[46] M. T. Goodrich and J. G. Kloss. *Tiered Vectors: Efficient Dynamic Arrays for Rank-Based Sequences*, pages 205–216. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[47] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th SODA*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

[48] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. In *Proc. 15th SODA*, pages 636–645. Society for Industrial and Applied Mathematics, 2004.

[49] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd STOC*, pages 397–406. ACM, 2000.

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

[51] J. Iacono and Ö. Özkan. Mergeable dictionaries. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6198(PART 1):164–175, 2010.

[52] A. Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms (TALG)*, 11(3):20, 2015.

[53] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proceedings of 24th CPM*, pages 189–200, 2013.

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

[55] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proceedings of 3rd WSP*, pages 141–155, 1996.

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

[57] J. Katajainen. *Worst-Case-Efficient Dynamic Arrays in Practice*, pages 167–183. Springer International Publishing, Cham, 2016.

[58] J. Katajainen and B. B. Mortensen. *Experiences with the Design and Implementation of Space-Efficient Deques*, pages 39–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[59] S. Kreft and G. Navarro. Lz77-like compression with fast random access. In *Proceedings of the 2010 Data Compression Conference*, DCC '10, pages 239–248, Washington, DC, USA, 2010. IEEE Computer Society.

[60] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoret. Comp. Sci.*, 483:115 – 133, 2013.

[61] E. Kushilevitz and N. Nisan. *Communication complexity*. Cambridge University Press, 1997.

[62] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[63] E. Lehman and A. Shelat. Approximation algorithms for grammar-based compression. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 205–212, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[64] A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[65] M. Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, LNCS, vol 8066, pages 267–302. Springer Berlin Heidelberg, 2013.

[66] V. Mäkinen. Compact suffix array. In *Proc. 11th CPM*, LNCS, vol 3109, pages 305–319. Springer Berlin Heidelberg, 2000.

[67] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Bio.*, 17(3):281–308, 2010.

[68] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[69] P. B. Miltersen, N. Nisan, S. Safra, and A. Wigderson. On data structures and asymmetric communication complexity. *J. Comp. Syst. Sci.*, 57(1):37 – 49, 1998.

[70] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

[71] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[72] G. Navarro. Indexing highly repetitive collections. In *Proc. 23rd IWOCA*, LNCS, vol 7643, pages 274–279. Springer Berlin Heidelberg, 2012.

[73] G. Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[74] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[75] I. Newman. Private vs. common random bits in communication complexity. *Inf. Proc. Lett.*, 39(2):67 – 71, 1991.

[76] N. Nisan. The communication complexity of threshold gates. *Combinatorics, Paul Erdos is Eighty*, 1:301–315, 1993.

[77] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and lz factorization in compressed space. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2016*, pages 158–170, Czech Technical University in Prague, Czech Republic, 2016.

[78] A. Policriti and N. Prezza. Fast online Lempel-Ziv factorization in compressed space. In *Proceedings of 22rd SPIRE*, pages 13–20, 2015.

[79] A. Policriti and N. Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica, Special Issue on Compact Data Structures*, 79:1–26, 2017.

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

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

[82] R. Raman, V. Raman, and S. S. Rao. *Succinct Dynamic Data Structures*, pages 426–437. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[83] A. Rao and A. Yehudayoff. *Communication Complexity (Early Draft)*. `https://homes.cs.washington.edu/~anuprao/pubs/book.pdf`, 2018.

[84] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[85] 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.

[86] P. Sena and S. Venkateshb. Lower bounds for predecessor searching in the cell probe model. *J. Comp. Syst. Sci.*, 74:364–385, 2008.

[87] T. A. Starikovskaya. Communication and streaming complexity of approximate pattern matching. In *Proc. 28th CPM*, pages 13:1–13:11, 2017.

[88] T. A. Starikovskaya. Streaming and property testing algorithms for string processing. 26th London Stringology Days, 2018.

[89] I. Tomohiro. Longest common extension with recompression. 2017.

[90] A. C.-C. Yao. Some complexity questions related to distributive computing(preliminary report). In *Proc. 11th STOC*, pages 209–213, 1979.

[91] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, (3), 1977.

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