Compact Data Structures for Faster String Processing

鶴田, 和弥

出版情報：九州大学, 2020, 博士（情報科学）, 課程博士
バージョン：
権利関係：

Compact Data Structures for Faster String Processing

Kazuya Tsuruta

January, 2021

Abstract

The amount of machine-readable data is increasing at an enormous and unprecedented rate. For many institutions and companies, the costs required to store and manage all of this increasing data make it virtually impossible to do so. Therefore, new foundational technologies for information systems that allow us to store, manage, search, and analyze data at smaller expenses, are strongly desired.

In this thesis we address (1) the substring search problem, (2) the prefix search problem and (3) the shortest unique substring (SUS) problem, and develop space-efficient data structures that enable us to quickly respond to queries.

The *substring search problem* is, given two strings T and P, called the *text* and *pattern*, to find all occurrences of P within T. In the case of static text, it is possible to build from T a data structure, called an *index structure*, for fast query processing. The most well-known examples of such index structures would be the suffix trees (Weiner 1973), the directed acyclic word graphs (Blumer et al. 1985; Crochemore 1986), the suffix arrays (Manber and Myers 1993). However, such an index structure consumes about 5 through 20 times the memory size of T. For this reason, several studies have been undertaken on *compressing* index structures. There are two lines. One is to compress indices from an information-theoretic viewpoint, and the other is to augment compressed texts with additional data structures. In this thesis, we focus on the latter, especially on index structures based on the *grammar-based compression*. A context-free grammar is said to *admissible* if it generates a single string. The grammar-based compression is a compression scheme which builds from T an admissible grammar G generating T. This compression scheme is most suited for *highly-repetitive* texts.

Claude and Navarro (2012) proposed the first grammar-based index structure of size $O(g)$ which can be built from an admissible grammar G generating T in $O(n + g \lg g)$ time, where n is the length of T and g is the size of G. It answers a substring query in $O(m^2 \lg \log_g n + (m + occ) \lg g)$, where m is the length of P and occ is the number of occurrences of P within T.

We note that this method is applicable to an *arbitrary* admissible grammar. However, it is impractical for a large value of m. Inspired by the Lyndon trees (Barcelo 1990), we introduce the *Lyndon SLPs*, a subclass of admissible grammars. We then propose the *Lyndon SLP based index*, a new index structure of $O(g_L)$ words of space which can be built in $O(n \lg n)$ expected time using $O(n)$ space, where g_L is the size of the Lyndon SLP. The proposed index is capable of finding all occurrences of P within T in $O(m + \lg m \lg n + occ \lg g_L)$ time. That is, we successfully remove the m^2 factor. Moreover, we show that the compression ratio of this method is competitive to some of the existing grammar-based indices by computational experiments.

The *prefix search problem* is, given a finite set K of k strings and a pattern P, to determine the subset of K consisting of texts that begin with P. An example of index structures for this problem is the Patricia tree (Morrison 1968). In this thesis, we focus on the dynamic case where K dynamically changes. Assuming that the symbols of strings of K are drawn from an alphabet of size $\sigma \leq 2^w$, we propose *dynamic* index structures of $N \lg \sigma + \Theta(k \lg N)$ bits of space or of $t \lg \sigma + \Theta(kw)$ bits of space, where N is the total length of strings of K, t is the number of nodes of a trie representing K and w is the computer word length, under the assumption of the w-bit word RAM model.. It responds to a prefix query P of length m in $O(m/\alpha + \lg \alpha + ans)$ time expected time, and performs an insertion/deletion of a string of length m in $O(m/\alpha + \lg \alpha)$ time, where ans is the number of answer strings in K, $\alpha = w/\lg \sigma$. The proposed indices are faster than existing dynamic indices when $m \geq \alpha$.

The *shortest unique substring* (SUS) problem is, given a text T of length n and a positive integer p with $1 \leq p \leq n$, to compute the set $SUS(p)$ of shortest unique substrings $T[i..j]$ with $i \leq p \leq j$, where a substring of T is said to be *unique* if it occurs just once in T. Pei et al. (2013) addressed the problem of finding one element of $SUS(p)$ and presented an index structure of $O(n)$ words of space that enables constant-time response. However, construction of this index structure consumes $O(n^2)$ time. In this thesis, we tackle the problem of finding *all* SUSs and present a new index structure that enables us to enumerate the elements of $SUS(p)$ in linear time proportional to $|SUS(p)|$. We also show an algorithm for constructing the index structure in $O(n)$ time and space.

Acknowledgments

I really appreciate all the support I received from everyone. First of all, I am deeply grateful to Professor Masayuki Takeda who is my supervisor and the committee chair of my thesis. He encouraged me to go to the PhD course, and taught many things to me, how to research, how to think, and so on.

I would also like to express my appreciation to Professor Kenji Ono and Associate Professor Daisuke Ikeda, who are the members of the committee of my thesis. I also thank all of those in Department of Informatics, Kyushu University, for their generous support.

I would also express my appreciation to Professor Hideo Bannai, Associate Professor Shunsuke Inenaga, Assistant Professor Yuto Nakashima and Assistant Professor Dominik Köppl. They helped with my research, and made my life in laboratory fruitful. I am deeply grateful to Professor Eiji Takimoto and Associate Professor Kohei Hatano. They taught me many Machine Learning techniques. The results in the thesis were partially published in the Proceedings of 40th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'14), the Proceedings of the Data Compression Conference 2020 (DCC 2020) and IPSJ Transactions on Mathematical Modeling and Its Applications Vol. 13, No. 2. I am thankful for all editors, committees, anonymous referees, and publishers.

I am also grateful to the technical staffs of our laboratory. They always supported and took care of me.

I express my gratitude to Dr. Sunsuke Kanda and his colleagues. I enjoyed discussions with them.

I would like to thank all of those in JVIS Corporation where I am employed. They support me, not limited to my job.

Last, but not least, I really thank my family for their support.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Problems and Contributions 1

1.2 Organization of This Thesis 5

2 Preliminaries **6**

2.1 Notation 6

2.2 Computation Model 7

3 Grammar-compressed Self-index with Lyndon Words **8**

3.1 Related Work 9

3.2 Preliminaries 10

3.3 Lyndon SLP 14

3.4 Lyndon SLP Based Self-Index 17

3.5 Conclusion 23

4 A Dynamic Trie Tailored for Fast Prefix Searches **25**

4.1 Related Work 27

4.2 Keyword Dictionary c-trie++ 29

4.3 Experiments 37

4.4 Detailed Space Analysis 43

4.5 Additional Experiments 43

4.6 Conclusion 57

5 Shortest Unique Substrings Queries in Optimal Time **58**

5.1 Preliminaries 59

5.2 Algorithm 61

5.3 Computational Experiments 67

6 Conclusion **71**

Chapter 1

Introduction

1.1 Problems and Contributions

In this thesis we address the following three problems:

Substring search problem. Given two strings T and P called *text* and *pattern* respectively, to find all occurrences of P within T.

Prefix search problem. Given a finite set D of strings, called *dictionary*, and another string P called *pattern*, to determine the strings in D that begin with P.

Shortest unique substring (SUS) problem. Given a string T called *text* and a positive integer p with $1 \leq p \leq |T|$, to find all shortest substring $T[i..j]$ with $i \leq p \leq j$ occurring just once in T.

For each problem, we develop a compact data structure for time efficient query processing.

1.1.1 Self-index for substring search

A context-free grammar is said to *represent* a string T if it generates the language consisting only of T. Grammar-based compression [61] is, given a string T, to find a small size description of T based on a context-free grammar that represents T. By eliminating repetitions, grammar-based compression is especially useful for *highly-repetitive strings*. Due to this merit, it has attracted research interests not only in improving the compression ratio but also in applications of this compressed representation. Notable applications are substring extraction or compressed string processing, such as the q-gram frequency calculation [38]. Compressed string processing is the approach to process a string given by its compressed representation, i.e., without explicitly

decompressing it. The goal of all these applications is to propose approaches that are faster than the naive way of explicitly decompressing and processing compressed strings, all while working in compressed space.

A *self-index* is a data structure that is a full-text index, i.e., supports various pattern matching queries on the text, and also provides random access to the text, usually without explicitly holding the text itself. Examples are the compressed suffix array [40, 43, 64], the compressed compact suffix array [70], and the FM index [32]. These self-indexes are, however, unable to fully exploit the redundancy of highly repetitive strings. To exploit such repetitiveness, Claude and Navarro [22] proposed the first self-index based on grammar-based compression. The method is based on a *straight-line program (SLP)*, a context-free grammar representing a single string in the Chomsky normal form. Plenty of grammar-based self-indexes have already been proposed (e.g., [23, 24, 84, 97, 98]).

In this thesis, we first introduce a new class of SLPs, named the *Lyndon SLP*, inspired by the Lyndon tree [8]. We then propose a self-index structure of $O(g)$ words of space that can be built from a string T in $O(n \lg n)$ expected time. The proposed self-index can find the starting positions of all occurrences of a pattern P of length m in $O(m + \lg m \lg n + occ \lg g)$ time, where n is the length of T, g is the size of the Lyndon SLP for T, and occ is the number of occurrences of P in T.

1.1.2 Compact dynamic index for prefix search

A *keyword* K is a string that is uniquely associated with an integer called the *identifier* of K (see Example 1). A *keyword dictionary* is a data structure that maintains a dynamic set $\mathcal{K}$ of keywords, and provides the following operations for a string S on it:

- insert(S) inserts S into $\mathcal{K}$, and returns its identifier. The keyword dictionary must guarantee that the identifiers of all stored keywords are unique and that each identifier is constant until its respective keyword is deleted.
- lookup(S) returns the identifier of S if $S \in \mathcal{K}$, or returns the invalid identifier $\perp$ otherwise.
- delete(K) removes the keyword K from $\mathcal{K}$.
- locatePrefix(S) returns an iterator on the set of identifiers of all keyword in $\mathcal{K}$ having S

as a prefix. The iterator can report the next occurrence in constant time[1].

Example 1. For the keywords $K_1 = \texttt{brausende}$, $K_2 = \texttt{brauereibräute}$, $K_3 = \texttt{brauen}$, $K_4 = \texttt{brauchbares}$, $K_5 = \texttt{brausendes}$ and $K_6 = \texttt{brauereibier}$, each subscript i is the identifier of keyword K_i.

We neglect the string dictionary operation access(i) returning the keyword of an identifier i, as this function can be realized by a separate data structure (in case of a trie, e.g., an array of pointers in which the i-th entry points to the node of the trie representing the keyword access(i)). For the performance of practical keyword dictionaries like RDF stores (e.g., [74]), insertions, lookups, and prefix queries are the most crucial operations, on which we want to focus in this thesis.

Keyword dictionaries are an integral data structure with a plethora of applications, e.g., n-gram language models [87], compression [33], input method editors [63], query auto-completion [45], or range query filtering [109].

In this thesis, we present a new keyword dictionary based on the compact trie: *Given a dynamic set $\mathcal{K}$ of k keywords whose characters are drawn from an integer alphabet of size $\sigma \leq 2^w$, there is a keyword dictionary representing $\mathcal{K}$ in either $n \lg \sigma + \Theta(k \lg n)$ or $|T| \lg \sigma + \Theta(kw)$ bits of space, where $n = \sum_{K \in \mathcal{K}} |K|$ is the total length of all keywords of $\mathcal{K}$, $|T|$ is the number of nodes of a trie representing $\mathcal{K}$ and w is the computer word length. It supports all keyword dictionary operations in $O(m/\alpha + \lg \alpha)$ expected time with $\alpha = w/\lg \sigma$ on an input string of length m.*

The above time and space bounds are an improvement to all previously known compact trie representations such as [12, 99]. One of the most important applications to compact tries is the *suffix tree* [104], which is a compact trie of all suffixes of the input string. Prefix searches arise in various uses of suffix trees, e.g., computing matching statistics [41], online suffix tree construction [103], online Lempel-Ziv 77 factorization [110], just to name a few. Hence, the time bound for prefix search is of significant theoretical interest, and our compact trie moves the best known upper bound closer to the trivial lower bound $\Omega(m/\alpha)$ for reading a pattern of length m word-packed. Also, with delete and insert operations, one can efficiently maintain the *sparse suffix tree* [55] for a dynamic set of suffixes to index.

Our experiments reveal that the above improvements are also practically significant. We note that other previous trie data structures mentioned earlier have the following drawbacks: (1)

[1] We return an iterator instead of this set, since most of the later explained data structures support all operations in the same time $O(t)$ for some t, while this operation would take $O(t + s)$ time, if the returned set has size s.

For the HAT-trie or the double array, there are no known non-trivial space and construction time bounds as their constructions are based on heuristics. In practice, they are also not favorable for prefix queries. (2) Trie data structures based on the Bonsai trie have the major drawback that enumerating children is done by querying for each possible edge label in a brute force manner. So they are no-good candidates for prefix search queries, and are therefore omitted in our practical evaluation. (3) The trie data structure of Jansson et al. [50] looks theoretically appealing, but uses theoretically sophisticated data structures for which an efficient implementation looks cumbersome.

1.1.3 Optimal solution to SUS problem

The shortest unique substring problem was proposed by Pei et al. [86]. Given a string S and position p, the problem is to find a *shortest unique substring* (SUS) of S that contains position p, that is, a substring that only occurs once in S, and whose occurrence contains position p. They also consider a version of the problem where S may be preprocessed, and SUS queries for arbitrary positions may be answered efficiently.

For the first version of the problem, Pei et al. [86] presented an algorithm that computes the SUS for any given position p in $O(n)$ time and space, where n is the length of string S. For the second version, they present an $O(hn)$ time and $O(n)$ space preprocessing algorithm which allows queries to be answered in constant time, where h is a value depending on S. However, h is only bounded by $O(n)$, and in the worst case, this results in $O(n^2)$ time preprocessing.

First, we give optimal time solutions for both problems and show that S can be preprocessed in $O(n)$ time so that a SUS for any query position can be answered in $O(1)$ time. This considerably improves the theoretical worst case running time compared to Pei et al. [86], allowing us to output a SUS for *all* positions in the string in $O(n)$ total time. Second, we consider the general problem of computing all SUSs that contain a given position. Although there can be multiple shortest substrings that contain a given query position, Pei et al. [86] only considered the problem of answering a single SUS that contains a position. We show that the same linear time preprocessing above also allows us to return *all* SUSs that contain a given query position in $O(k)$ time, where k is the size of output. Finally, we implement our algorithm and show through computational experiments that our algorithm is much more practical and scalable compared to an the algorithm by Pei et al.

After this work, many studies have been performed on the SUS problem and its variants.

We mention some of such studies. Ileri et al. [48] presented another optimal solution to the same problem, which consumes less memory than ours from a practical viewpoint. Hon et al. [44] presented an *in-place* algorithm for solving the single-SUS problem. Ganguly et al. [37] discussed time-space trade-offs on the single-SUS problem. They showed that given a query position p, a SUS for p can be computed (1) in $O(n\tau^2 \log \frac{n}{\tau})$ time using S and an additional $O(n/\tau)$ words of working space; and (2) in $O(n\tau^2 \log n)$ time using S and an additional $O(n/\tau)$ words and $4n + o(n)$ bits of working space, where τ is a given parameter. Hu et al. [46] considered a more general problem: *Given a query interval* $[i..j]$ *with* $[i..j] \subseteq [1..n]$*, output all SUSs that contain* $[i..j]$. They proposed a data structure that returns all SUSs for the query interval in optimal time. Mieno et al. [79] proposed a data structure of $\lceil(\log_2 3 + 1)n\rceil + o(n)$ bits of space that returns all SUSs for a given query interval in optimal time. Mieno et al. [78] considered a *compressed version* of this problem: *Given a run-length encoded string* S*, construct a data structure that returns all SUSs for a query interval*. They presented a data structure of $O(r)$ space, which can be constructed in $O(r \log r)$ time and answers queries in $O(\sqrt{\log m/\log\log m} + k)$ time, where r is the compressed size of S.

1.2 Organization of This Thesis

The rest of this thesis is organized as follows. In Chapter 2 we give some notations. In Chapter 3, we introduce a new grammar-based compression scheme named *Lyndon grammar compression*, and compare its compression performance with several other grammar compression methods by computational experiments. Then we develop a new self-index structure based on it. In Chapter 4, we present a compact dynamic index structure for prefix search, and compare its performance with existing dynamic index structures. In Chapter 5, we address the SUS problem and show an optimal algorithm.

Chapter 2

Preliminaries

2.1 Notation

Let Σ be an alphabet. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε is the string of length 0. For a string $T = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of T, respectively. A prefix (resp. suffix) x of T is called a *proper prefix* (resp. suffix) of T if $x \neq T$. T^ℓ denotes the ℓ times concatenation of the string T. The i-th character of a string T is denoted by $T[i]$, where $i \in [1..|T|]$. For a string T and two integers i and j with $1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j. For convenience, let $T[i..j] = \varepsilon$ when $i > j$. For any string P and T, we call a position i an *occurrence* of P in T, if $T[i..i + |P| - 1] = P$.

For any integers $i \leq j$, let $[i..j]$ denote an interval, i.e., the set of integers $\{i, \ldots, j\}$, and let $|[i..j]| = j - i + 1$ denote the length of the interval. For convenience, let $[i..j]$ denote the empty set when $i > j$.

Given two distinct positions i, j $(i < j)$, we say that i is to the *left* and j the *right*. Two distinct intervals are *nested*, if one is a subset of the other. For two non-nested intervals $[i..j]$ and $[i'..j']$, we say that $[i..j]$ is to the left and $[i'..j']$ is to the right, if $i < j$. Since, for any interval $[i..j]$ $(1 \leq i \leq j \leq |T|)$ there is a corresponding substring $T[i..j]$ of T, we abuse the language and will many times call an interval a substring.

We denote by $\log_b x$ the logarithm of x to the base b. We often use $\lg x$ as an abbreviation of $\log_2 x$.

2.2 Computation Model

Our model of computation is the standard word RAM model of word size w. We can read and process $O(w)$ bits in constant time. Let n be a natural number with $n = O(2^w)$. Storing an integer of the domain $[1..n]$ costs $\lg n$ bits such that pointers for the problem size n can be represented in $\lg n$ bits (like in the transdichotomous model). The choice of this model (severing the connection between word size and the logarithm of the problem size) is justified by the fact that the register sizes of SIMD instruction sets is increasing since the recent years significantly (e.g., AVX512 with 512-bit registers).

Chapter 3

Grammar-compressed Self-index with Lyndon Words

A context-free grammar is said to *represent* a string T if it generates the language consisting of T and only T. Grammar-based compression [61] is, given a string T, to find a small size description of T based on a context-free grammar that represents T. By eliminating repetitions, grammar-based compression is especially useful for *highly-repetitive strings*. Due to this merit, it has attracted research interests not only in improving the compression ratio but also in applications of this compressed representation. Notable applications are substring extraction or compressed string processing, such as the q-gram frequency calculation [38]. Compressed string processing is the approach to process a string given by its compressed representation, i.e., without explicitly decompressing it. The goal of all these applications is to propose approaches that are faster than the naive way of explicitly decompressing and processing compressed strings, all while working in compressed space.

A *self-index* is a data structure that is a full-text index, i.e., supports various pattern matching queries on the text, and also provides random access to the text, usually without explicitly holding the text itself. Examples are the compressed suffix array [40, 43, 64], the compressed compact suffix array [70], and the FM index [32][2]. These self-indexes are, however, unable to fully exploit the redundancy of highly repetitive strings. To exploit such repetitiveness, Claude and Navarro [22] proposed the first self-index based on grammar-based compression. The method is based on a *straight-line program (SLP)*, a context-free grammar representing a single string in the Chomsky normal form. Plenty of grammar-based self-indexes have already been proposed (e.g., [23, 98, 97, 84]).

[2]Navarro and Mäkinen [81] published an excellent survey on this topic.

In this chapter, we first introduce a new class of SLPs, named the *Lyndon SLP*, inspired by the Lyndon tree [8]. We then propose a self-index structure of $O(g)$ words of space that can be built from a string T in $O(n \lg n)$ expected time. The proposed self-index can find the starting positions of all occurrences of a pattern P of length m in $O(m + \lg m \lg n + occ \lg g)$ time, where n is the length of T, g is the size of the Lyndon SLP for T and occ is the number of occurrences of P in T.

This result was originally published in [102].

3.1 Related Work

The *smallest grammar problem* is, given a string T, to find the context-free grammar G representing T with the smallest possible size, where the *size* of G is the total length of the right-hand sides of the production rules in G. Since the smallest grammar problem is NP-hard [96], many attempts have been made to develop small-sized context-free grammars representing a given string T. LZ78 [111], LZW [105], Sequitur [83], Sequential [61], LongestMatch [61], Re-Pair [65], and Bisection [60] are grammars based on simple greedy heuristics. Among them Re-Pair is known for achieving high compression ratios in practice.

Approximations for the smallest grammar have also been proposed. The AVL grammars [89] and the α-balanced grammars [19] can be computed in linear time and achieve the currently best approximation ratio of $O(\lg(|T|/g_T^*))$ by using the LZ77 factorization and the balanced binary grammars, where g_T^* denotes the smallest grammar size for T. Other grammars with linear-time algorithms achieving the approximation $O(\lg(|T|/g_T^*))$ are LevelwiseRePair [91] and Recompression [51]. They basically replace di-grams with a new variable in a bottom-up manner similar to Re-Pair, but use different mechanisms to select the di-grams. On the other hand, LCA [92] and its variants [93, 72, 73] are known as scalable practical approximation algorithms. The core idea of LCA is the *edit-sensitive parsing (ESP)* [26], a parsing algorithm developed for approximately computing the edit distance with moves. The *locally-consistent-parsing (LCP)* [90] is a generalization of ESP. The *signature encoding (SE)* [75], developed for equality testing on a dynamic set of strings, is based on LCP and can be used as a grammar-transform method. The ESP index [98, 97] and the SE index [84] are grammar-based self-indexes based on ESP and SE, respectively.

While our experimental section (Section 3.3.3) serves as a practical comparison between the sizes of some of the above mentioned grammars, Table 3.1 gives a comparison with some

theoretically appealing index data structures based on grammar compression. There, we chose the indexes of Claude and Navarro [23], Gagie et al. [34], and Christiansen et al. [21] because these indexes have non-trivial time complexities for answering queries. We observe that our proposed index has the fastest construction among the chosen grammar indexes, and is competitively small if $g = o(\gamma \lg(n/\gamma))$ while being clearly faster than the first two approaches for long patterns. It is worth pointing out that the grammar index of Christiansen et al. [21] achieves a grammar size whose upper bound $O(\gamma \lg(n/\gamma))$ matches the upper bound of the size g_T^* of the smallest possible grammar. Unfortunately, we do not know how to compare our result within these terms in general.

Table 3.1: Complexity bounds of self-indexes based on grammar compression.

n is the length of T, z is the number of LZ77 [110] phrases of T, γ is the size of the smallest string attractor [59] of T, g is the size of the Lyndon SLP of T, $\hat{g}$ is the size of a given admissible grammar, $\epsilon > 0$ is a constant, m is the length of a pattern P, and occ is the number of occurrences of P in T.

Construction space (in words) and time

Index	Space	Time
[23]	$O(n)$	$O(n + \hat{g} \lg \hat{g})$
[21]	$O(n)$	$O(n \lg n)$ expected
[21]	$O(n)$	$O(n \lg n)$ expected
ours	$O(n)$	$O(n \lg n)$ expected

Needed space (in words) and query time for a pattern of length m

Index	Space	Locate Time
[23]	$O(\hat{g})$	$O(m^2 \lg \log_{\hat{g}} n + (m + occ) \lg \hat{g})$
[34]	$O(\hat{g} + z \lg \lg z)$	$O(m^2 + (m + occ) \lg \lg n)$
[21]	$O(\gamma \lg(n/\gamma))$	$O(m + \lg^\epsilon \gamma + occ \lg^\epsilon(\gamma \lg(n/\gamma)))$
[21]	$O(\gamma \lg(n/\gamma) \lg^\epsilon(\gamma \lg(n/\gamma)))$	$O(m + occ)$
Theorem 2	$O(g)$	$O(m + \lg m \lg n + occ \lg g)$

3.2 Preliminaries

3.2.1 Lyndon words and Lyndon trees

Let $\preceq$ denote some total order on Σ that induces the lexicographic order $\preceq$ on Σ^*. We write $u \prec v$ to imply $u \preceq v$ and $u \neq v$ for any $u, v \in \Sigma^*$.

Definition 1 (Lyndon Word [69]). *A non-empty string $w \in \Sigma^+$ is said to be a* Lyndon word *with respect to $\prec$ if $w \prec u$ for every non-empty proper suffix u of w.*

By this definition, all characters ($\in \Sigma^1$) are Lyndon words.

Definition 2 (Standard Factorization [20, 67]). *The* standard factorization *of a Lyndon word w with $|w| \geq 2$ is an ordered pair (u, v) of strings u, v such that $w = uv$ and v is the longest proper suffix of w that is also a Lyndon word.*

Lemma 1 ([9, 67]). *For a Lyndon word w with $|w| > 1$, the standard factorization (u, v) of w always exists, and the strings u and v are Lyndon words.*

The Lyndon tree of a Lyndon word w, defined below, is the full binary tree induced by recursively applying the standard factorization on w.

Definition 3 (Lyndon Tree [8]). *The* Lyndon tree *of a Lyndon word w, denoted by $LTree(w)$, is an ordered full binary tree defined recursively as follows:*

- *if $|w| = 1$, then $LTree(w)$ consists of a single node labeled by w;*
- *if $|w| \geq 2$, then the root of $LTree(w)$, labeled by w, has the left child $LTree(u)$ and the right child $LTree(v)$, where (u, v) is the standard factorization of w.*

3.1 shows an example of a Lyndon tree for the Lyndon word `aababaababb`.

3.2.2 Admissible grammars and straight-line programs (SLPs)

An admissible grammar [61] is a context-free grammar that generates a language consisting only of a single string. Formally, an *admissible grammar (AG)* is a set of production rules $\mathcal{G}_{\mathrm{AG}} = \{X_i \rightarrow \mathrm{expr}_i\}_{i=1}^{g}$, where X_i is a *variable* and expr_i is a non-empty string over $\Sigma \cup \{X_1, \ldots, X_{i-1}\}$, called an *expression*. The variable X_g is called the *start symbol*. We denote by $val(X_i)$ the string derived by X_i. We say that an admissible grammar $\mathcal{G}_{\mathrm{AG}}$ *represents* a string T if $T = val(X_g)$. To ease notation, we sometimes associate $val(X_i)$ with X_i. The *size* of $\mathcal{G}_{\mathrm{AG}}$ is the total length of all expressions expr_i. We assume that any admissible grammar has no useless symbols.

It should be stated that the above definition of admissible grammar is different with but equivalent to the original definition in [61], which defines an admissible grammar to be a

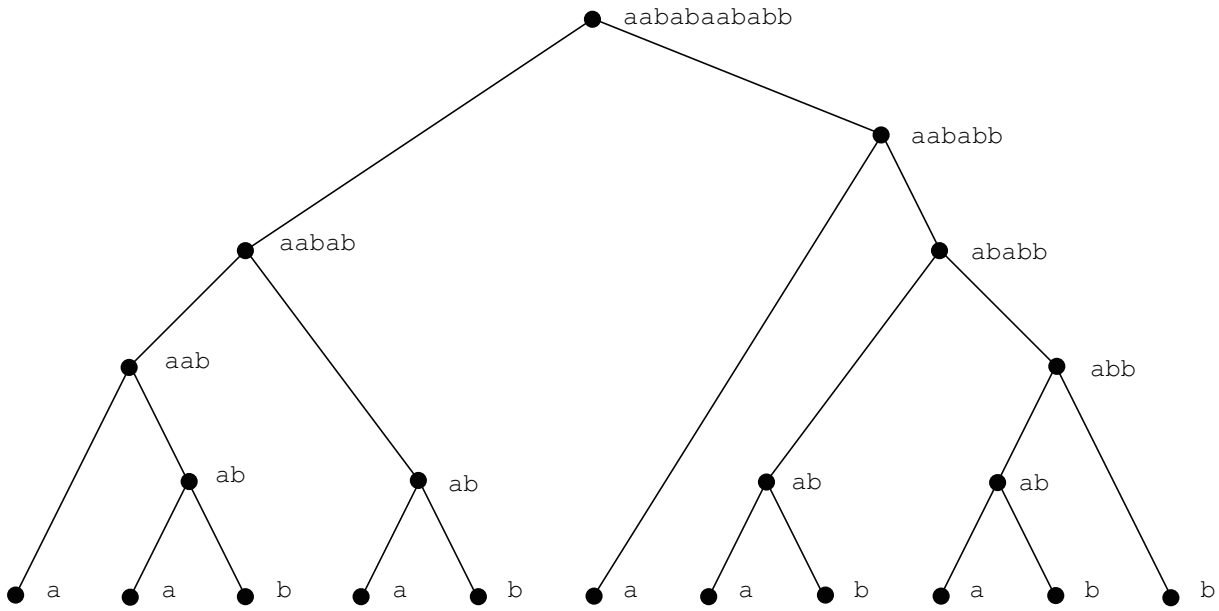


Figure 3.1: The Lyndon tree for the Lyndon word aababaababb with respect to the order a $\prec$ b, where each node is accompanied by its derived string to its right.

context-free grammar G satisfying the conditions: (1) G is deterministic, i.e., for every variable A there is exactly one production rule of the form $A \to \gamma$, where γ is a non-empty string consisting of variables and characters; (2) G has no production rule of the form $A \to \varepsilon$; (3) The language $L(G)$ of G is not empty; and (4) G has no useless symbols, i.e., every symbol appears in some derivation that begins with the start symbol and ends with a string consisting only of characters.

A *straight-line program* (*SLP*) is an admissible grammar in the Chomsky normal form, namely, each production rule is either of the form $X_i \to a$ for some $a \in \Sigma$ or $X_i \to X_{i_\mathrm{L}} X_{i_\mathrm{R}}$ with $i > i_\mathrm{L}, i_\mathrm{R}$. Note that $\mathcal{G}_\mathrm{SLP}$ can derive a string up to length $O(2^g)$. This can be seen by the example string $T = \mathtt{a} \cdots \mathtt{a}$ consisting of $n = 2^\ell$ a's, where the smallest SLP $\{X_1 \to \mathtt{a}\} \cup \bigcup_{j=2}^{\ell+1} \{X_j \to X_{j-1} X_{j-1}\}$ has size $2\ell + 1$.

The derivation tree $\mathcal{T}_{\mathcal{G}_\mathrm{SLP}}$ of $\mathcal{G}_\mathrm{SLP}$ is a labeled ordered binary tree, where each internal node is labeled with a variable in $\{X_1, \ldots, X_g\}$, and each leaf is labeled with a character in Σ. The root node has the start symbol X_g as label. An example of the derivation tree of an SLP is shown in Figure 3.2.

3.2.3 Grammar irreducibility

An admissible grammar is said to be *irreducible* if it satisfies the following conditions:

C-1. Every variable other than the start symbol is used more than once (**rule utility**);

C-2. All pairs of symbols have at most one non-overlapping occurrence in the right-hand sides of the production rules (**di-gram uniqueness**); and

C-3. Distinct variables derive different strings.

Grammar-based compression is a combination of

1. the *grammar transform*, i.e., the computation of an admissible grammar G representing the input string T, and

2. the *grammar encoding*, i.e., an encoding for G.

Kieffer and Yang [61] showed that a combination of an irreducible grammar transform and a zero order arithmetic code is universal, where a grammar transform is said to be *irreducible* if the resulting grammars are irreducible.

If an admissible grammar G is not irreducible, we can apply at least one of the following reduction rules [61] to make G irreducible:

R-1. Replace each variable X_i occurring only once in the right-hand sides of the production rules with expr_i and remove the rule $X_i \to \mathrm{expr}_i$. We also remove all production rules with useless symbols.

R-2. Given there are at least two non-overlapping occurrences of a string γ of symbols with $|\gamma| \geq 2$ in the right-hand sides of the production rules, replace each of the occurrences of γ with the variable X_i, where $X_i \to \gamma$ is an existing or newly created production rule. Recurse until no such γ longer exists.

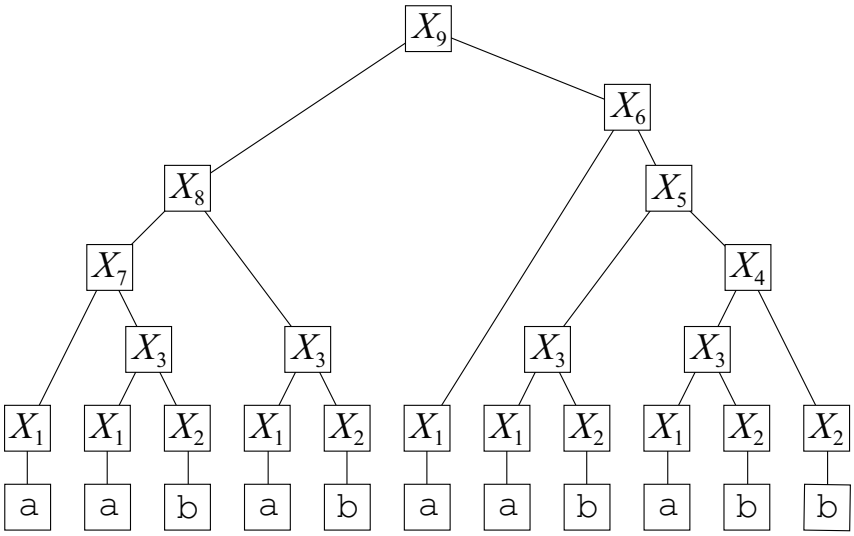
R-3. For each two distinct variables X_i and X_j deriving an identical string, (a) replace all occurrences of X_j with X_i in the right-hand sides of the production rules, and (b) remove the production rule $X_j \to \mathrm{expr}_j$ and discard the variable X_j. Consequently, there are no two distinct variables X_i and X_j with $val(X_i) = val(X_j)$. This operation possibly makes some variables useless; the production rules with such variables will be removed by R-1.

3.3 Lyndon SLP

In what follows, we propose a new SLP, called Lyndon SLP. A *Lyndon SLP* is an SLP $\mathcal{G}_{\text{LYN}} = \{X_i \to \text{expr}_i\}_{i=1}^{g}$ representing a Lyndon word, and satisfies the following properties:

- The strings $val(X_i)$ are Lyndon words for all variables X_i.
- The standard factorization of the string $val(X_i)$ is $(val(X_{i_\text{L}}), val(X_{i_\text{R}}))$ for every rule $X_i \to X_{i_\text{L}} X_{i_\text{R}}$.
- No pair of distinct variables X_i and X_j satisfies $val(X_i) = val(X_j)$.

The derivation tree (when excluding its leaves) of $\mathcal{T}_{\mathcal{G}_{\text{LYN}}}$ is isomorphic to the Lyndon tree of T (cf. Figure 3.2).



$$
\begin{aligned}
X_1 &\to \texttt{a} \\
X_2 &\to \texttt{b} \\
X_3 &\to X_1 X_2 \\
X_4 &\to X_3 X_2 \\
X_5 &\to X_3 X_4 \\
X_6 &\to X_1 X_5 \\
X_7 &\to X_1 X_3 \\
X_8 &\to X_7 X_3 \\
X_9 &\to X_8 X_6
\end{aligned}
$$

Figure 3.2: *Left*: The derivation tree of the Lyndon SLP $\mathcal{G}_{\text{LYN}}$ with $g = 9$ representing the Lyndon word $T = \texttt{aababaababb}$. *Right*: The production rules of $\mathcal{G}_{\text{LYN}}$.

The rest of this chapter is devoted to algorithmic aspects regarding the Lyndon SLP. We study its construction (Section 3.3.1), practically evaluate its size (Section 3.3.3), and propose an index data structure on it (Section 3.4). For that, we work in the word RAM model supporting packing characters of sufficiently small bit widths into a single machine word. Let w denote the machine word size in bits.

We fix a text $T[1..n]$ over an integer alphabet Σ with size $\sigma = n^{O(1)}$. If T is not a Lyndon word, we prepend T with a character smaller than all other characters appearing in T. Let g denote the size $|\mathcal{G}_{\text{LYN}}|$ of the Lyndon SLP $\mathcal{G}_{\text{LYN}}$ of T.

Lemma 2 (Algorithm 1 of [7]). *We can construct the Lyndon tree of T in $O(n)$ time.*

3.3.1 Constructing Lyndon SLPs

The algorithm of Bannai et al. [7, Algorithm 1] builds the Lyndon tree *online* from right to left. We can modify this algorithm to create the Lyndon SLP of T by storing a dictionary for the rules and a reverse dictionary for looking up rules: Whenever the algorithm creates a new node u, we query the reverse dictionary with u's two children v and w for an existing rule $X \rightarrow X_v X_w$, where X_v and X_w are the variables representing v and w. If such a rule exists, we assign u the variable X, otherwise we create a new rule $X_u \rightarrow X_v X_w$ and put this new rule into both dictionaries. The dictionaries can be implemented as balanced search trees or hash tables, featuring $O(n \lg g)$ deterministic construction time or $O(n)$ expected construction time, respectively.

In the static setting (i.e., we do not work online), deterministic $O(n)$ time can be achieved with the enhanced suffix array [71, 1] supporting constant time longest common extension queries. We associate each node v of the Lyndon tree with the pair $(|T[i..j]|, rank(i))$, where $T[i..j]$ is the substring derived from the non-terminal representing v, and $rank(i)$ is the lexicographic rank of the suffix starting at position i. Then, sort all nodes according to their associated pairs with a linear-time integer sorting algorithm. By using longest common extension queries between adjacent nodes of equal length in the sorted order, we can determine in $O(1)$ time per node whether they represent the same string, and if so, assign the same variable (otherwise assign a new variable).

3.3.2 Lyndon array simulation

As a by-product, we can equip the Lyndon SLP of T with the indexing data structure of Bille et al. [15] to support character extraction and navigation in $O(\lg n)$ time. This allows us to compute the i-th entry of the Lyndon array [7] in $O(\lg n)$ time, where the i-th entry of the Lyndon array of T stores the length of the longest Lyndon word starting at $T[i]$. For that, given a text position i, we search for the highest Lyndon tree node having $T[i]$ as its leftmost leaf. Given the rightmost leaf of this node represents $T[j]$, the longest Lyndon word starting at $T[i]$ has the length $j-i+1$. (Otherwise, there would be a higher node in the Lyndon tree representing a longer Lyndon word starting at $T[i]$.)

Lemma 3. *There is a data structure of size $O(g)$ that can retrieve the longest Lyndon word starting at $T[i]$ in $O(\lg n)$ time.*

3.3.3 Computational experiments

We empirically benchmark the grammar sizes obtained by the Lyndon SLP to highlight its potential as a grammar compressor. As benchmark datasets we used four highly repetitive texts consisting of the files `cere`, `einstein.de.txt`, `kernel`, and `world_leaders` from the Pizza & Chili corpus (`http://pizzachili.dcc.uchile.cl`). We used the natural order implied by the ASCII code for building the Lyndon SLPs. We compared the size of the resulting Lyndon grammars with the resulting grammars of Re-Pair, LCA, Recompression. We used existing implementations of Re-Pair (`https://users.dcc.uchile.cl/~gnavarro/software/`) and of LCA (`http://code.google.com/p/lcacomp/`). The outputs of LCA, Recompression and our method are SLPs, while those of Re-Pair are AGs (and not necessarily SLPs). For a fair comparison, we compared the resulting grammar sizes either in an SLP representation, or in a common AG representation.

SLP We keep the resulting grammar of the Lyndon SLP, LCA, and Recompression as it is, but transform the output of Re-Pair to an SLP. To this end, we observe that Re-Pair consists of (a) a list of non-terminals whose right hand sides are already of length two, and (b) a start symbol whose right hand side is a string of symbols of arbitrary size. Consequently, to transform this grammar to an SLP, it is left to focus on the start symbol: We replace greedily di-grams in the right hand side of the start symbol until it consists only of two symbols.

AG We process each grammar in the following way: First, we remove the production rules of the form $X_i \to a \in \Sigma$ by replacing all occurrences of X_i with a. Subsequently, we apply the reduction rule R-1 of Section 3.2.3.

We collected the obtained grammar sizes in Table 3.2. There, we observe that the Lyndon SLP is no match for Re-Pair, but competitive with LCA and Recompression. Although this evaluation puts Re-Pair in a good light, it seems hard to build an index data structure on this grammar that can be as efficient as the self-index data structure based on the Lyndon SLP, which we present in the next section.

Table 3.2: Sizes of the resulting grammars benchmarked in Section 3.3.3.

collection		Re-Pair	LCA	Recompression	Lyndon SLP
`cere`	SLP	6,433,183	9,931,777	8,537,747	13,026,562
	AG	4,057,693	6,513,345	5,309,789	7,469,979
`einstein.de.txt`	SLP	125,343	251,411	202,749	205,348
	AG	84,493	168,193	127,790	123,963
`kernel`	SLP	2,254,840	4,065,522	3,587,382	4,201,895
	AG	1,373,244	2,507,291	2,135,779	2,400,211
`world_leaders`	SLP	601,757	1,243,757	1,023,739	911,222
	AG	398,234	809,163	636,700	552,497

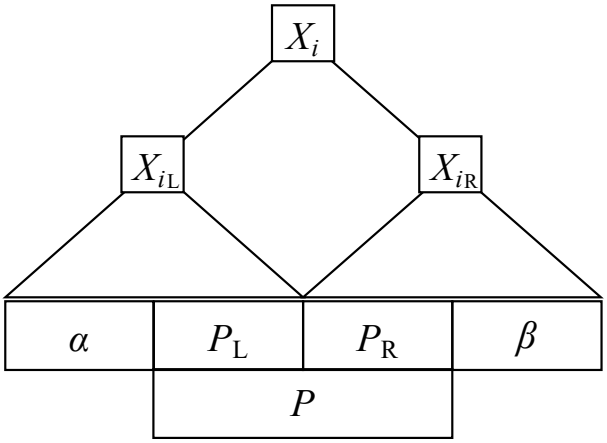


Figure 3.3: A partition pair $(P_{\mathrm{L}}, P_{\mathrm{R}})$ of a pattern P with one of its associated tuples (X_i, α, β).

3.4 Lyndon SLP Based Self-Index

Given a Lyndon SLP of size g, we can build an indexing data structure on it to query all occurrences of a pattern P of length $m \in [1..n]$ in T. We call this query $locate(P)$. Our data structure is based on the approach of [22]. This approach separates the occurrences of a pattern into so-called primary occurrences and secondary occurrences. It first locates the primary occurrences and, with the help of these, it subsequently locates the secondary occurrences. To this end, it locates primary occurrences with a labeled binary relation data structure, and subsequently locates the secondary occurrences with the grammar tree. In our case, we find the primary occurrences with so-called partition pairs.

A *partition pair* (at position i) of a pattern $P[1..m]$ is a pair $(P[1..i], P[i+1..m])$ with $i \in [1..m]$ such that there exists a rule $X_i \rightarrow X_{i_{\mathrm{L}}} X_{i_{\mathrm{R}}}$ with $val(X_{i_{\mathrm{L}}})$ and $val(X_{i_{\mathrm{R}}})$ having $P[1..i]$ and $P[i+1..m]$ as a (not necessarily proper) suffix and as a prefix, respectively. Similar to the grammar proposed in Section 6.1 of [21], we can bound the number of partition pairs by

$O(\lg m)$ by carefully selecting all possible partition pairs:

Given a partition pair $(P_{\mathrm{L}}, P_{\mathrm{R}})$ of P, let $X_i \to X_{i_{\mathrm{L}}} X_{i_{\mathrm{R}}}$ be a rule such that $val(X_{i_{\mathrm{L}}})$ and $val(X_{i_{\mathrm{R}}})$ have P_{L} and P_{R} as a suffix and as a prefix, respectively. Consequently, there exist two strings α and β such that $val(X_{i_{\mathrm{L}}}) = \alpha P_{\mathrm{L}}$ and $val(X_{i_{\mathrm{R}}}) = P_{\mathrm{R}}\beta$ (cf. Figure 3.3). By the definition of the Lyndon tree of the text T, $(val(X_{i_{\mathrm{L}}}), val(X_{i_{\mathrm{R}}})) = (\alpha P_{\mathrm{L}}, P_{\mathrm{R}}\beta)$ is the standard factorization of $val(X_i) = \alpha P_{\mathrm{L}} P_{\mathrm{R}} \beta$. According to the standard factorization, $P_{\mathrm{R}}\beta$ is the longest suffix of $val(X_i)$ that is a Lyndon word. For the proofs of Lemmas 10 and 11, we use this notation and call the tuple (X_i, α, β) a *tuple associated with* $(P_{\mathrm{L}}, P_{\mathrm{R}})$.

Let us take $P := \mathtt{bab}$ as an example. The only partition pair is $(\mathtt{b}, \mathtt{ab})$. Considering the Lyndon grammar of our example text given in Figure 3.2, the tuples associated with $(\mathtt{b}, \mathtt{ab})$ are $(X_8, \mathtt{aa}, \varepsilon)$ and $(X_5, \mathtt{a}, \mathtt{b})$.

Note that $|\alpha| = 0$ if P is a Lyndon word. If P is a proper prefix of a Lyndon word[3], then α may be empty. If P is a not a (not necessarily proper) prefix of a Lyndon word, then $|\alpha| > 0$ (since $\alpha P_{\mathrm{L}} P_{\mathrm{R}} \beta$ is a Lyndon word).

3.4.1 Associated tuples with non-empty α

We want to reduce the number of possible partition pairs from m to $O(\lg m)$. A first idea is that only the beginning positions of the Lyndon factors of P contribute to potentially partition pairs. We prove this in Lemma 8, after defining the Lyndon factors:

The (composed) *Lyndon factorization* [20] of a string $P \in \Sigma^+$ is the factorization of P into a sequence $P_1^{\tau_1} \cdots P_p^{\tau_p}$ of lexicographically decreasing Lyndon words $P_1, \ldots, P_p$, where (a) each $P_x \in \Sigma^+$ is a Lyndon word, and (b) $P_x \succ P_{x+1}$ for each $x \in [1..p)$. P_x and $P_x^{\tau_x}$ are called *Lyndon factor* and *composed Lyndon factor*, respectively.

Lemma 4 (Algorithm 2.1 of [29])**.** *The Lyndon-factorization of a string can be computed in linear time.*

We borrow from Section 2.2 of [47] the notation $\mathit{lfs}_P(x) := P_x^{\tau_x} \cdots P_p^{\tau_p}$ for the suffix of P starting with the x-th Lyndon factor. Given $\lambda_P \in [1..p]$ is the smallest integer such that $\mathit{lfs}_P(x+1)$ is a prefix of P_x for every $x \in [\lambda_P..p-1]$, $\mathit{lfs}_P(x)$ is called a *significant suffix* of P for every $x \in [\lambda_P..p]$. Consequently, $\mathit{lfs}_P(p) = P_p^{\tau_p}$ is a significant suffix.

In what follows, we show that P_{R} of a partition pair $(P_{\mathrm{L}}, P_{\mathrm{R}})$ has to start with a Lyndon factor (Lemma 8), and further has to start with a composed Lyndon factor (Lemma 10). Finally,

[3] I.e., there is a string $S \in \Sigma^+$ such that PS is a Lyndon word.

we refine this result by restricting P_{R} to begin with a significant suffix (Lemma 11) whose number is bounded by the following lemma:

Lemma 5 (Lemma 12 of [47])**.** *The number of significant suffixes of P is $O(\lg m)$.*

In what follows, we study the occurrences of P in T under the circumstances that T is represented by its Lyndon tree induced by the standard factorization, while P is represented by its Lyndon factors.

Lemma 6 (Proposition 1.10 of [29])**.** *The longest prefix of P that is a Lyndon word is the first Lyndon factor P_1 of P.*

Lemma 7 (Lemma 5.4 of [7])**.** *Given a production $X_j \to X_{j_{\mathrm{L}}} X_{j_{\mathrm{R}}} \in \mathcal{G}_{\mathrm{LYN}}$, there is no Lyndon word that is a substring of $val(X_j) = val(X_{j_{\mathrm{L}}}) val(X_{j_{\mathrm{R}}})$ beginning in $val(X_{j_{\mathrm{L}}})$ and ending in $val(X_{j_{\mathrm{R}}})$, except $val(X_j)$.*

Lemma 8. *Given $(P_{\mathrm{L}}, P_{\mathrm{R}})$ is a partition pair of a pattern P, P_{R} starts with a Lyndon factor of P if there is an associated tuple (X_i, α, β) with $|\alpha| > 0$.*

Proof. Since $|\alpha| > 0$ holds, P is a proper substring of $val(X_i)$. Then P_{R} must start with a Lyndon factor of P according to Lemma 7. □

Lemma 9 (Proposition 1.3 of [29])**.** *Given two Lyndon words α, β with $\alpha \prec \beta$, the concatenation $\alpha\beta$ is also a Lyndon word.*

Lemma 10. *Given $(P_{\mathrm{L}}, P_{\mathrm{R}})$ is a partition pair of a pattern P, P_{R} starts with a composed Lyndon factor of P if there is an associated tuple (X_i, α, β) with $|\alpha| > 0$.*

Proof. Let (X_i, α, β) be a tuple associated with $(P_{\mathrm{L}}, P_{\mathrm{R}})$. Assume for the contrary that P_{R} does not start with any composed Lyndon factors of P, namely, there exists $x \in [1..p]$ and $k \in [1..\tau_x - 1]$ such that P_{L} and P_{R} have $P_x^{\tau_x - k}$ and P_x^k as a suffix and prefix, respectively (cf. Figure 3.4). By the assumption, $val(X_{i_{\mathrm{R}}}) = P_x^k lfs_P(x+1)\beta$ is the longest Lyndon word that is a suffix of $val(X_i)$. Since $P_x \prec val(X_{i_{\mathrm{R}}})$ and P_x is a Lyndon word, $P_x val(X_{i_{\mathrm{R}}})$ is also a Lyndon word by Lemma 9. This contradicts that $val(X_{i_{\mathrm{R}}})$ is the longest Lyndon word that is a suffix of $val(X_i)$. □

Lemma 10 helps us to concentrate on the *composed* Lyndon factors. Next, we show that only those composed Lyndon factors are interesting that start with a significant suffix:

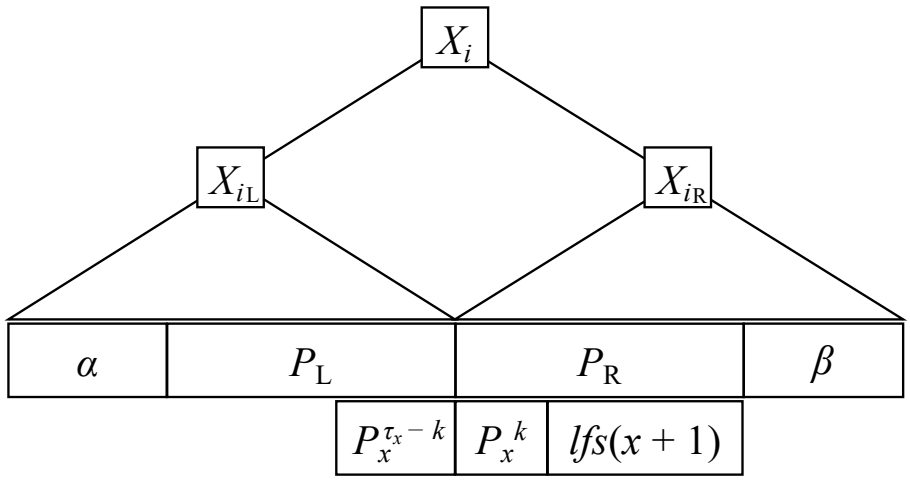


Figure 3.4: Setting of the proof of Lemma 10.

Lemma 11. *Given $(P_{\mathrm{L}}, P_{\mathrm{R}})$ is a partition pair of a pattern P, then P_{R} is a significant suffix of P if there is an associated tuple (X_i, α, β) with $|\alpha| > 0$.*

Proof. Let (X_i, α, β) be a tuple associated with $(P_{\mathrm{L}}, P_{\mathrm{R}})$ and $|\alpha| > 0$. By Lemma 10, there exists $x \in [1..p]$ such that $P_{\mathrm{R}} = P_x^{\tau_x} \cdots P_p^{\tau_p}$. Assume for the contrary that $x < \lambda_P$, i.e., P_{R} is not a significant suffix of P. By definition, $lfs_P(x) \succ lfs_P(x+1)$ holds. Since $lfs_P(x+1)$ is not a prefix of $lfs_P(x)$, $lfs_P(x)\beta \succ lfs_P(x+1)\beta$ also holds. This implies that $P_{\mathrm{R}} = lfs_P(x)\beta$ is not a Lyndon word, a contradiction. □

This, together with Lemma 5, yields the following corollary.

Corollary 1. *There are $O(\lg m)$ partition pairs of P associated with a tuple (X_i, α, β) with $|\alpha| > 1$.*

Let us take $P :=$ `abacabadabacababa` as an elaborated example. Its composed Lyndon factorization is $P = P_1 P_2 P_3^2 P_4$, where its Lyndon factors are $P_1 =$ `abacabad`, $P_2 =$ `abac`, $P_3 =$ `ab`, and $P_4 =$ `a` with $\lambda_P = 3$. Hence, $lfs_P(3)$ and $lfs_P(4)$ are significant suffixes. Its potential partition pairs are $(P_1P_2, P_3^2P_4)$, $(P_1P_2P_3^2, P_4)$. There is no Lyndon SLP such that another partitioning like $(P_1, P_2P_3^2P_4)$ or $(P_1P_2P_3, P_3P_4)$ would have an associated tuple according to Lemmas 11 and 10, respectively.

3.4.2 Associated tuples with empty α

Given a partition pair $(P_{\mathrm{L}}, P_{\mathrm{R}})$ associated with a tuple $(X_i, \varepsilon, \beta)$, we consider two cases depending on $|P_{\mathrm{L}}|$: In the case of $|P_{\mathrm{L}}| = 1$, $(P[1], P[2..m])$ may be a partition pair of P. In the

case of $|P_{\mathrm{L}}| \geq 2$, suppose that $P' = P[2..m]$, $\alpha' = P[1]$ and $(P'_{\mathrm{L}}, P'_{\mathrm{R}})$ is a partition pair of P' with associated tuple (X_i, α', β). Then, $(P[1]P'_{\mathrm{L}}, P'_{\mathrm{R}})$ is a partition pair of P with associated tuple $(X_i, \varepsilon, \beta)$. We can use Lemmas 8, 10 and 11 to restrict P'_{R} starting with a significant suffix of $P[2..m]$ (cf. Figure 1).

Corollary 2. *There are $O(\lg m)$ partition pairs of P associated with a tuple $(X_i, \varepsilon, \beta)$.*

Combining Corollary 1 with Corollary 2 yields the following theorem and the main result of this subsection:

Theorem 1. *There are $O(\lg m)$ partition pairs of a pattern of length m.*

3.4.3 Locating a pattern

In the following, we use the partition pairs to find all primary occurrences. We do this analogously as for the Γ-tree (Section 3.1 of [82]) or for special grammars (Section 6.1 of [21]).

Lemma 12 (Lemma 5.2 of [36])**.** *Let $\mathcal{S}$ be a set of strings and assume that we can (a) extract a substring of length ℓ of a string in $\mathcal{S}$ in time $f_e(\ell)$ and (b) compute the Karp-Rabin fingerprint [57] of a substring of a string in $\mathcal{S}$ in time f_h. Then we can build a data structure of $O(|\mathcal{S}|)$ words solving the following problem in $O(m \lg \sigma/w + t(f_h + \lg m) + f_e(m))$ time: given a pattern $P[1..m]$ and $t > 0$ suffixes $Q_1, \ldots, Q_t$ of P, discover the ranges of strings in (the lexicographically-sorted) $\mathcal{S}$ prefixed by $Q_1, \ldots, Q_t$.*

Lemma 13 (Theorem 1.1 of [15])**.** *For an AG of size g representing a string of length n we can extract a substring of length ℓ in time $O(\ell + \lg n)$ after $O(g)$ preprocessing time and space.*

Lemma 14 (Theorem 1 of [13])**.** *Given a string of length n represented by an SLP of size g, we can construct a data structure supporting fingerprint queries in $O(g)$ space and $O(\lg n)$ deterministic query time. This data structure can be constructed in $O(n \lg n)$ randomized time (cf. Section 2.4 of [35]) by using Karp, Miller and Rosenberg's [56] renaming algorithm to make all fingerprints unique.*

With Lemmas 13 and 14 we have $f_e(\ell) = O(\ell + \lg n)$ and $f_h = O(\lg n)$ in Lemma 12, respectively, leading to:

Corollary 3. *There is a data structure using $O(g)$ space such that, given a pattern $P[1..m]$ with $m \leq n$, it can find all variables whose derived strings have one of t selected suffix of P as a prefix in $O(m \lg \sigma/w + t(\lg n + \lg m) + \ell + \lg n)$ time.*

Corollary 3 yields $O(m \lg n)$ time for $t = m$, i.e., when we need to split the pattern at each position. It yields $O(m \lg \sigma/w + \lg m \lg n)$ time for $t = \lg m$, i.e., the case for Section 6.1 of [21] and for the Lyndon SLP thanks to Lemma 11 (we assume that the pattern is not longer than the text).

We can retrieve the associated tuples of all primary occurrences by plugging the variables retrieved in Corollary 3 into a data structure for labeled binary relations [22].

For that, we generate two list $\mathcal{L}$ and $\mathcal{L}^{\mathrm{REV}}$ of all variables $X_1, \cdots, X_g$ of the grammar sorted lexicographically by their derived strings and the reverses of their derived strings, respectively. Both lists allow us to answer a prefix (resp. suffix) query by returning a range of variables having the prefix (resp. suffix) in question. The query is performed by the data structure described in Lemma 12 (with $\mathcal{S}$ being either $\mathcal{L}$ or $\mathcal{L}^{\mathrm{REV}}$). Finally, we can plug the obtained ranges into the labeled binary relation data structure of Claude and Navarro [22]:

Lemma 15 (Theorem 3.1 of [22])**.** *Given two list $\mathcal{L}$ and $\mathcal{L}^{\mathrm{REV}}$ of variables sorted lexicographically by their expressions and its reversed strings, we can built a data structure of $O(g)$ words of space in $O(g \lg g)$ time for supporting the following query: Given a partition pair $(P_{\mathrm{L}}, P_{\mathrm{R}})$ and ranges in $\mathcal{L}$ and $\mathcal{L}^{\mathrm{REV}}$ of those variables whose derived strings have $val(P_{\mathrm{R}})$ as a prefix and $val(P_{\mathrm{L}})$ as a suffix, this data structure can retrieve all associated tuples of $(P_{\mathrm{L}}, P_{\mathrm{R}})$ in $O((1 + occ') \lg g)$ time, where occ' denotes their number.*

The time complexity of Corollary 3 and Lemma 12 is based on the assumption that we have (static) z-fast tries [11] built on the lists $\mathcal{L}$ and $\mathcal{L}^{\mathrm{REV}}$[4], which we can build in $O(g)$ expected time and space (Section 6.6 (3) of [21]).

Since there are $O(\lg m)$ partition pairs according to Corollary 1, applying Lemma 15 over all $O(\lg m)$ partition pairs yields $O(\lg m \lg g + occ \lg g)$ time, where occ denotes the number of all primary occurrences.

Corollary 4. *We can find the primary occurrences of a pattern P in*

$$\underbrace{O(m)}_{Lemma\ 4} + \underbrace{O(m \lg \sigma/w + \lg m \lg n)}_{Lemma\ 3} + \underbrace{O(\lg m \lg g + occ \lg g)}_{Lemma\ 15}$$

$= O(m + \lg m \lg n + occ \lg g)$ *time.*

[4]We use again the derived string or, respectively, the reverse of the derived string of each non-terminal in one of the lists as its respective keyword to insert into the trie.

Finally, we use the derivation tree to find the remaining (secondary) occurrences of the pattern:

3.4.4 Search for secondary occurrences

We follow Claude and Navarro [23] improving the search of the secondary occurrences in [22] by applying reduction rule R-1 to enforce C-1 (see Section 3.2.3). The resulting admissible grammar $\mathcal{G}_{\mathrm{AG}}$ is no longer an SLP in general. Since we only remove variables with a single occurrence, the size of $\mathcal{G}_{\mathrm{AG}}$ is $O(g)$. Consequently, we can store both $\mathcal{G}_{\mathrm{AG}}$ *and* $\mathcal{G}_{\mathrm{SLP}}$ in $O(g)$ space.

Lemma 16 (Section 5.2 of [23])**.** *Given the associated tuples of all partition pairs, we can find all occ occurrences of P in T with $\mathcal{G}_{\mathrm{AG}}$ in $O(occ \lg g)$ time.*

Remembering that we split the analysis of an associated tuple in the cases $|\alpha| = 0$ (Section 3.4.2) and $|\alpha| > 0$ (Corollary 4), we observe that the time complexity of the latter case dominates. Combining this time with Lemma 16 yields the time complexity for answering $locate(P)$ with the Lyndon SLP:

Theorem 2. *Given the Lyndon SLP of T, there is a data structure using $O(g)$ words that can be constructed in $O(n \lg n)$ expected time, supporting $locate(P)$ in $O(m + \lg m \lg n + occ \lg g)$ time for a pattern P of length m.*

Note that the $O(n \lg n)$ expected construction time is due to the data structure described in Lemma 14.

3.5 Conclusion

We introduced a new class of SLPs, named the *Lyndon SLP*, and proposed a self-index structure of $O(g)$ words of space, which can be built from an input string T in $O(n \lg n)$ expected time, where n is the length of T and g is the size of the Lyndon SLP for T. By exploiting combinatorial properties on Lyndon SLPs, we showed that $locate(P)$ can be computed in $O(m + \lg m \lg n + occ \lg g)$ time for a pattern P of length m, where occ is the number of occurrences of P. This is better than the $O(m^2 \lg \log_{\hat{g}} n + (m + occ) \lg \hat{g})$ query time of the SLP-index by Claude and Navarro [23] (cf. Table 3.1), which works for a *general* admissible grammar of size $\hat{g}$.

We have not implemented the proposed self-index structure, and comparing it with other self-index implementations such as the FM index [30], the LZ index [3], the ESP index [98], or the LZ-end index [62] will be a future work. Also, we want to speed up the query time to $O(m \lg \sigma / w + \lg m \lg n + occ \lg g)$ by applying broadword techniques for determining the Lyndon factors of the pattern P (cf. Corollary 4), where σ is the alphabet size and w is the computer word length.

Chapter 4

A Dynamic Trie Tailored for Fast Prefix Searches

A *keyword* K is a string that is uniquely associated with an integer called the *identifier* of K. A *keyword dictionary* is a data structure that maintains a dynamic set of keywords $\mathcal{K}$, and provides the following operations for a string S on it:

- insert(S) makes S a keyword, inserts S into $\mathcal{K}$, and returns its identifier. The keyword dictionary must guarantee that the identifiers of all stored keywords are unique and that each identifier is constant until its respective keyword is deleted.
- lookup(S) returns the identifier of S if $S \in \mathcal{K}$, or returns the invalid identifier $\perp$ otherwise.
- delete(K) removes the keyword K from $\mathcal{K}$.
- locatePrefix(S) returns an iterator on the set of identifiers of all keyword in $\mathcal{K}$ having S as a prefix. The iterator can report the next occurrence in constant time[5].

We neglect the string dictionary operation access(i) returning the keyword of an identifier i, as this function can be realized by a separate data structure (in case of a trie, e.g., an array of pointers in which the i-th entry points to the node of the trie representing the keyword access(i)). For the performance of practical keyword dictionaries like RDF stores (e.g., [74]), insertions, lookups, and prefix queries are the most crucial operations, on which we want to focus in this chapter.

[5]We return an iterator instead of this set, since most of the later explained data structures support all operations in the same time $O(t)$ for some t, while this operation would take $O(t + s)$ time, if the returned set has size s.

In this chapter, we present a new keyword dictionary based on the compact trie:

Theorem 3. *Given a dynamic set $\mathcal{K}$ of k keywords whose characters are drawn from an integer alphabet of size $\sigma \leq 2^w$, there is a keyword dictionary representing $\mathcal{K}$ in either $n \lg \sigma + \Theta(k \lg n)$ or $|T| \lg \sigma + \Theta(kw)$ bits of space, where $n = \sum_{K \in \mathcal{K}} |K|$ is the total length of all keywords of $\mathcal{K}$ and $|T|$ is the number of nodes of a trie representing $\mathcal{K}$. It supports all keyword dictionary operations in $O(m/\alpha + \lg \alpha)$ expected time with $\alpha = w/\lg \sigma$ on an input string of length m.*

The time and space bounds of Theorem 3 are an improvement to all previously known compact trie representations such as [12, 99]. One of the most important applications to compact tries is the *suffix tree* [104], which is a compact trie of all suffixes of the input string. Prefix searches arise in various uses of suffix trees, e.g., computing matching statistics [41], online suffix tree construction [103], online Lempel-Ziv 77 factorization [110], just to name a few. Hence, the time bound for prefix search is of significant theoretical interest, and our compact trie moves the best known upper bound closer to the trivial lower bound $\Omega(m/\alpha)$ for reading a pattern of length m word-packed. Also, with delete and insert operations, one can efficiently maintain the *sparse suffix tree* [55] for a dynamic set of suffixes to index.

Our experiments reveal that the above improvements are also practically significant. We note that other previous trie data structures mentioned earlier have the following drawbacks: (1) For the HAT-trie or the double array, there are no known non-trivial space and construction time bounds as their constructions are based on heuristics. In practice, they are also not favorable for prefix queries. (2) Trie data structures based on the Bonsai trie have the major drawback that enumerating children is done by querying for each possible edge label in a brute force manner. So they are no-good candidates for prefix search queries, and are therefore omitted in our practical evaluation. (3) The trie data structure of Jansson et at. [50] looks theoretically appealing, but uses theoretically sophisticated data structures for which an efficient implementation looks cumbersome.

For the rest of this chapter, we fix a dynamic set $\mathcal{K}$ consisting of k keywords with a total length of $n = \sum_{K \in \mathcal{K}} |K|$. The keywords of $\mathcal{K}$ do not have to be prefix-free.

This result primarily appeared in [101].

4.1 Related Work

Keyword dictionaries are an integral data structure with a plethora of applications (e.g., n-gram language models [87], compression [33], input method editors [63], query auto-completion [45], or range query filtering [109]). As a well-studied abstract data type they also have many representations. We refer to standard literature like [94, Chapter 5.2], [76, Chapter 28], or [80, Chapter 8.5.3] for an introduction to common representations like tries. Here, we highlight some of the most recent representations. For the analysis, let $|T| \leq n$ denote the number of nodes of a trie T storing $\mathcal{K}$, and let m be the length of an input string for one of the keyword dictionary operations.

- The HAT-trie [5] is a practically optimized version of the burst trie [42]. It suppresses the number of trie nodes by selectively collapsing subtries into cache-conscious hash tables of strings [6]. Although there is no discussion of prefix searches in [5], the implementation of Tessil[6] supports locatePrefix. We are unaware of any theoretical results regarding space or time.

- The Bonsai trie [27] is a trie whose nodes are maintained in a compact hash table [25]. Modern variants [88] use $O(n \lg \sigma)$ bits of space in expectancy, and perform insert and lookup in $O(m)$ expected time. However, it is not clear how to perform locatePrefix efficiently.

- Kanda et al. [53] proposed a dynamic variant of the path decomposed trie of Grossi and Ottaviano [39] by means of *incremental* path decomposition. This dynamic trie supports insert and lookup in $O(m)$ expected time. However, there is no discussion about prefix searches. Actually, as Kanda's trie is based on the Bonsai trie, it faces the same problem for locatePrefix.

- The double array [2] simulates a trie by using two integer arrays to find a child in constant time, and thus can perform lookup in $O(m)$ time. Although the double array includes some vacant slots and consumes $\Omega(n \lg n)$ bits, those vacant slots have a negligible memory effect in practical implementations such as the Cedar trie [108]. In the static setting, Kanda et al. [52] proposed a practically compressed data structure for the two arrays. However, for any of these data structures, it is not clear to us what time is needed for answering locatePrefix.

[6] `https://github.com/Tessil/hat-trie`

- Jansson et at. [50] presented a dynamic trie using $O(|T| \lg \sigma)$ bits, in which a leaf can be inserted or deleted in $O((\lg \lg |T|)^2 / \lg \lg \lg |T|)$ time. This trie can compute a prefix search in $O((m/ \lg_\sigma |T|)(\lg \lg |T|)^2 / \lg \lg \lg |T|)$ time [50, Theorem 1]. In an alternative representation, this trie supports insertions and deletions of leaves in $O(\lg \lg |T|)$ expected amortized time while supporting a prefix search in $O(m/ \lg_\sigma |T| + \lg \lg |T|)$ worst-case time [50, Theorem 2].

- The *(dynamic) z-fast trie* is a keyword dictionary of Belazzougui et al. [12], which uses $|T| \lg \sigma + \Theta(kw)$ bits of space, and supports all operations in $O(m/\alpha + \lg m + \lg \lg \sigma)$ expected time[7].

- Takagi et al. [99] proposed the *dynamic packed compact trie*, whose name we abbreviate to *packed c-trie*. The packed c-trie uses $|T| \lg \sigma + \Theta(kw)$ bits of space, and supports all operations in $O(m/\alpha + \lg w)$ expected time.

- HOT [17] is an algorithmically engineered trie that applied different strategies depending on the distribution of the common prefix lengths of the keywords to obtain high fan-outs and minimize the depth of the trie. It also applies AVX2 instructions for lookup queries.

The following keyword dictionaries are static, but share common traits with our proposed data structure:

- Grossi and Ottaviano [39] proposed a cache-friendly trie dictionary through path decomposition [31]. An operation can be carried out in $O(m + h \lg \sigma)$ time, where h is the height of the path-decomposed trie. The data structure is stored in compressed space by exploiting text compression techniques and succinct data structures.

- Marisa trie, developed by Yata [107], is a static trie that consists of recursively compressed Patricia tries stored in the LOUDS representation [49]. It recursively encodes edge labels in a Patricia trie using another Patricia trie. Yata's implementation[8] supports prefix searches.

- Arz and Fischer [4] proposed a static compressed trie by adapting the LZ78 parsing to basic dictionary operations such as lookup. It represents $\mathcal{K}$ in $O(k \lg n + n \lg \sigma)$ bits

[7] This time bound can be achieved by omitting the jump pointers in [12, Section 3.4] since their maintenance needs additional time. The jump pointers are used to enable additional operations on the trie such as predecessor queries, on which we do not put a focus in this chapter.

[8] `https://github.com/s-yata/marisa-trie`

of space by leveraging the LZ78 compression. It can answer lookup in $O(m)$ expected time. However, we are not aware of whether this data structure supports efficient prefix searches.

- Bille et al. [14] presented a static keyword dictionary using $O(n \lg n)$ bits of space and $O(n)$ time to represent $\mathcal{K}$. It supports queries in $O(m/\alpha + \lg m + \lg \lg \sigma)$ time.
- A recent approach is due to Bille et al. [16], who proposed a static keyword dictionary with $O(n \lg \sigma)$ bits of space using $O(\min(m \lg \sigma, m + \lg n))$ time for an operation in the pointer machine model.
- The fast succinct trie (FST) is a trie data structure used in the succinct range filter [109]. An FST is divided into two layers at a specific height. The top layer is represented by a *speed*-optimized trie while the bottom layer is represented by a *space*-optimized trie. Both tries are represented as level-order unary degree sequences [49].

4.2 Keyword Dictionary c-trie++

Focusing on fast prefix searches, our idea is to devise a new keyword dictionary based on the compact trie data structures, as they are practically faster than approaches based on the double array when the prefixes in question are relatively short to the stored keywords. Our approach, called c-trie++ for *improved compact trie*, is a hybrid of the z-fast trie and the packed c-trie. Like these two trie representations, the compact trie is decomposed in a macro trie storing micro tries.

For a formal explanation of this decomposition, let the *string depth* of a node u denote the length of the concatenation of all labels on the path from the macro trie root to u. Further, we assume that the keyword set $\mathcal{K}$ is prefix-free such that each leaf corresponds to one keyword. In the general case, we would not only consider leaves but also internal nodes corresponding to a keyword. Our starting point is a compact trie. If there is an edge leading to an internal node, we split up this edge by creating additional nodes on this edge whose string depths are a multiple of α. Subsequently, we put all nodes whose string depths are a multiple of α into the macro trie. Let u be one of these nodes, and let $d\alpha$ be its string depth. Then u becomes the root of a micro trie if it has more than one descendant in the compact trie whose string depth is at most $(d + 1)\alpha$. Suppose that u is the root of a micro trie, then this micro trie stores all of u's

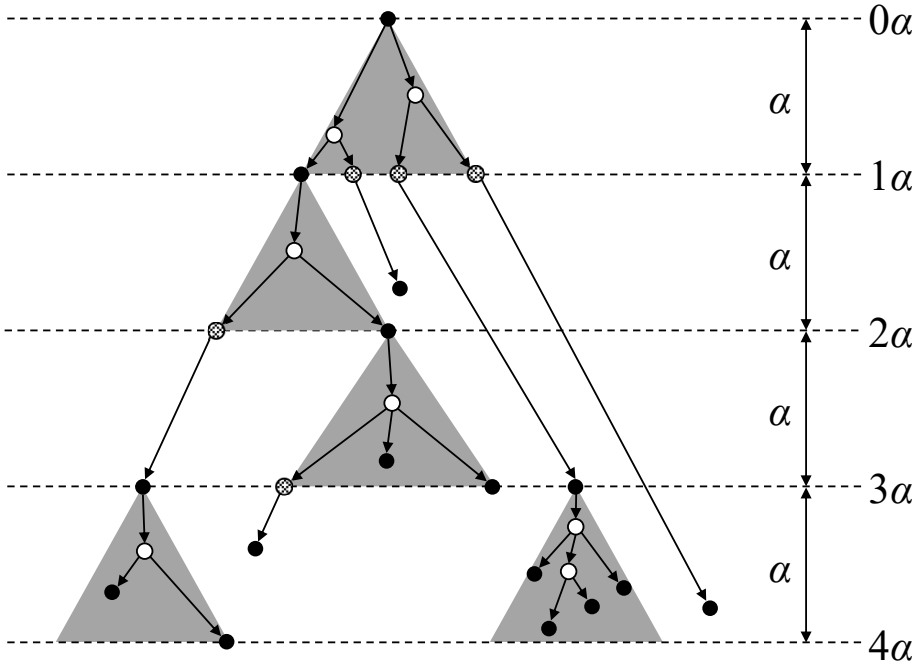


Figure 4.1: The macro trie of a c-trie++ instance. Micro tries are represented by shaded triangles (cf. [99, Figure 2]). Circles filled with black color are macro trie nodes. Hollow circles are nodes stored exclusively in a micro trie. Cross-hatched circles are nodes of a micro trie that are not present in the compact trie (as they have only one child). These nodes are leaves of a micro trie, and are needed for navigating between the micro trie and the macro trie nodes below of it.

descendants (of the compact trie) whose string depths are at most $(d+1)\alpha$. Every edge (w, v) from a node w of u's micro trie leading to a descendant v of u with a string depth larger than $(d+1)\alpha$ is split to (w, x) and (x, v) for an artificial node x with string depth $(d+1)\alpha$ (cf. the cross-hatched circles in Figure 4.1). Finally, leaves of the compact trie are macro trie nodes. As previously explained, there can additionally be micro trie nodes if (a) their string depth is between $d\alpha$ and $(d+1)\alpha$ and (b) they have an ancestor with string depth $d\alpha$ that is the root of the respective micro trie. Consequently, the total number of micro and macro trie nodes in bounded by $O(k)$, where k is the number of nodes in the compact trie. Figure 4.1 captures this schematically.

For c-trie++, we use the trie decomposition of the packed c-trie for the macro trie. Our micro tries are *alphabet-aware z-fast tries*. We maintain all keywords in an array of pointers to keywords of total size $k \lg n + n \lg \sigma$ bits. We represent a substring of a keyword with a starting position and a length, which can be stored in $2 \lg n$ bits.

The z-fast trie proposed by Belazzougui et al. [12] works on binary strings. Their results on micro trees work for binary strings up to length $O(w)$. However, it is easy to modify these micro

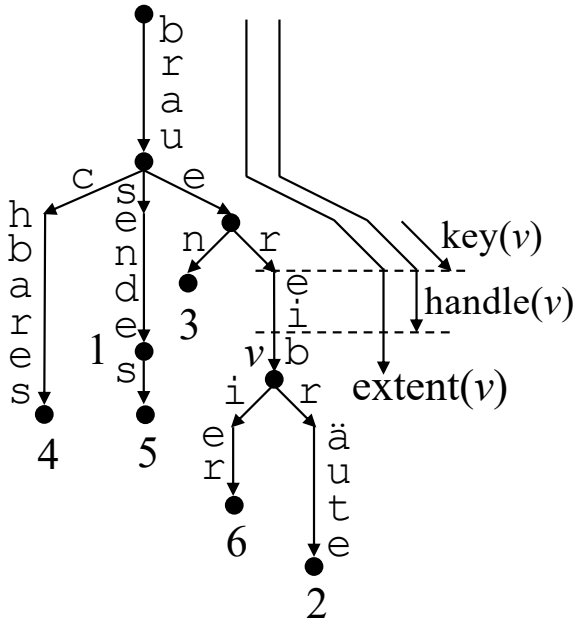


Figure 4.2: The micro trie built on our running example $\mathcal{K} = \{K_1 = \texttt{brausende}, K_2 = \texttt{brauereibräute}, K_3 = \texttt{brauen}, K_4 = \texttt{brauchbares}, K_5 = \texttt{brausendes}, K_6 = \texttt{brauereibier}\}$, which is not prefix-free. A leaf u storing number i is associated with the identifier i, i.e., extent$(u) = K_i$. In this example, the node v storing the extent `brauereib` has two children w_1 and w_2, which are determined by their keys key$(w_1) = \texttt{i}$ and key$(w_2) = \texttt{r}$, respectively. If we assume that eight characters fit into a computer word, then the extent of v is outside of the micro trie containing the root node. This fact is symbolized by the dashed line separating the eighth and the ninth character of extent(v).

trees [12, Theorem 1] to work with strings on the alphabet Σ up to length $O(w/\lg\sigma) = O(\alpha)$ by packing $O(\alpha)$ characters in a constant number of machine words:

Lemma 17. *Let $\mathcal{K}$ be a dynamic set of k keywords whose characters are drawn from an alphabet of size $\sigma \leq 2^w$. Given that each keyword of $\mathcal{K}$ has a length of $O(\alpha)$, there is a keyword dictionary representing $\mathcal{K}$ in either $n\lg\sigma + \Theta(k\lg n)$ or $|T|\lg\sigma + \Theta(kw)$ bits of space, where $\alpha = w/\lg\sigma$, $n = \sum_{K\in\mathcal{K}}|K| \leq \alpha|\mathcal{K}|$ is the total length of all keywords of $\mathcal{K}$, and $|T|$ is the number of nodes of a trie representing $\mathcal{K}$. It supports all keyword dictionary operations in either $O(\lg\alpha)$ expected time or $O(\lg\alpha\lg^2\lg\sigma/\lg\lg\lg\sigma)$ deterministic time.*

Proof. The main difference is that the original micro trie is a binary tree as its edge labels are drawn from a binary alphabet. Since the edge labels in our alphabet-aware variant are characters drawn from the integer alphabet Σ, traversing from a node to a specific child now costs $O(\sigma)$ time. We improve this time by augmenting each node with a data structure maintaining its children such that, given a node v and a character c, we can navigate from v to its child

connected with the edge starting with c by querying this data structure having stored c and v as key and value, respectively. This data structure can be realized with a hash table with constant expected time, or with a predecessor data structure like [10] taking $O(m \lg n)$ bits and supporting all operations in $O(\lg \lg \sigma \lg \lg m / \lg \lg \lg \sigma) = O(\lg^2 \lg \sigma / \lg \lg \lg \sigma)$ deterministic time when storing $m \leq \sigma$ elements (the space bounds are due to the fact that we store pointers to the specific children as satellite data). This sums up to $O(k \lg n)$ bits as we have $O(k)$ trie nodes. □

An operation with a string of length m with $m = \Omega(\alpha)$ (but with $m = O(2^w)$) involves the traversal of the macro tree, which is done in $O(m/\alpha)$ expected time[9] for all keyword dictionary operations [99]. Combining the operations in the macro trie and in micro tries gives $O(m/\alpha + \lg \alpha)$ total time, and concludes Theorem 3.

4.2.1 Micro Tries

For explaining c-trie++ in detail, we start with a review of the z-fast trie under the light of our alphabet-aware variant. We say that a node v is *associated with the identifier* of a keyword K if we can read K by following the path from the root to v. The alphabet-aware z-fast trie is a compact trie in which each leaf v is associated with the identifier of a keyword. An internal node has at least two children unless it is also associated with the identifier of a keyword. If the set of keywords $\mathcal{K}$ is prefix-free, then there are no nodes with a single child.

Figure 4.2 shows an instance of such a trie. The figure also depicts the following definitions that are substrings or nodes associated to each node of an alphabet-aware z-fast trie.

- key(v) is the first character in label of the edge connecting v with its parent. It is undefined if v is the root.
- extent(v) is the string obtained by concatenating the edge labels of the path from the root node to v.
- exit(S) is the highest node v for which, among all other nodes, the longest common prefix between S and extent(v) is the longest.
- parex(S) is the parent node of exit(S), or a special symbol $\perp$ with extent$(\perp) = 1$ if exit(S) is the root node.

[9] See Section 4.2.2 for a detailed description of the macro trie.

It is left to explain for what handle(v) stands in the figure. For that we need the notion of 2-fattest numbers [12, Definition 1]. The *2-fattest number* of an interval $[\ell..r]$ of positive integers $0 < \ell < r$ is the integer in $[\ell..r]$ with the most trailing zeros in its binary representation. Given a node v with its parent u, we can compute the 2-fattest number f of $[|\text{extent}(u)| + 1..|\text{extent}(v)|]$ to determine the handle of v, which is handle(v) := extent$(v)[1..f]$. In case that v is the root, we set handle(v) to the empty string.

For supporting the keyword dictionary operations, we need operations to descend in a micro tree. For that, as already described in the proof of Lemma 17, each internal node u stores a dictionary $\mathsf{DicChild}_u$ to access one of its child nodes v by the character key(v). Additionally, the trie maintains a dictionary DicHandle that can address each internal node u by its handle handle(u).

For the algorithmic part, we follow Algorithm 1 and Section 3.3 of [12]. Given a pattern P of length $O(\alpha)$, this algorithm locates exit(P) and parex(P). Having exit(P) and parex(P), we can perform all keyword dictionary operations as in the z-fast trie. The idea of the algorithm is to perform a search on the interval $[\ell..r]$, which is set to $[1..|P|]$ at the beginning to try to find the lowest node whose handle is a prefix of P. The search handles this interval similarly to a binary search. For explanation, the algorithm is divided into rounds. In each round, it (a) either enlarges ℓ or shrinks r, (b) computes the 2-fattest number f of $[\ell..r]$, and (c) queries DicHandle with the handle $P[1..f]$. If there is a node v with handle$(v) = P[1..f]$, the algorithm has matched $P[1..f]$ with this node and simulates the descending to this trie node by setting $\ell \leftarrow |\text{extent}(v)|$. Otherwise (there is no such node v), the algorithm sets $r \leftarrow f-1$ to aim for jumping to a node whose extent is less than f. The algorithm stops when it finds either exit(P) and parex(P) [12, Theorem 3], which is after $O(\lg |P|)$ rounds. If exit(P) is found, it has previously already computed parex(P). Otherwise, it takes that child of parex(P) whose edge connected to parex(P) leads us to exit(P). For finding this child, the algorithm uses $\mathsf{DicChild}_{\text{parex}(P)}$. Finally, for updates we follow the same steps as described in [12, Section 5].

In the context of the example of Figure 4.2, this algorithm applied to $P =$ `brauereibock` gives us the node exit(P), which is the node v visualized in Figure 4.2. From there, we can query $\mathsf{DicChild}_{\text{exit}(P)}$ for the predecessor (resp. successor) with the character $\circ$ to find the predecessor (resp. successor) of P, which is K_6 (resp. K_2).

4.2.2 Macro Trie

It is left to describe the macro trie borrowed from the packed c-trie, and to analyze the space and time complexity of c-trie++. The macro trie is needed to cope with keywords longer than α characters, or w bits. The rough idea is to partition a long keyword into chunks of w bits, and maintain the chunks in a dictionary DicChunk similar to DicHandle, mapping w-bit chunks to macro trie nodes. Given that the root is at height 0, a node on a height h of the macro trie is endowed with

- a micro trie representing its descendants whose extents are at most $(h+1)w$ bits long, and with
- a DicChunk representing its children whose extents are longer than $(h+1)w$ bits.

Its DicChunk stores the w-bit substring starting at the $(hw+1)$-th bit of the extents of its respective children. An update of the trie involves a lookup of the insertion or deletion position, and a modification of DicChunk or a micro trie.

Space Complexity Our keyword dictionary c-trie++ maintains $O(k)$ macro and $O(k)$ micro nodes. Each node stores a pointer to a substring of a keyword. The keywords are stored either in a concatenated string of length $n \lg \sigma$, or are compressed via front coding [106, Section 4.1] taking $|T| \lg \sigma$ bits in total. We store extent(v) of a node v either as two n-bit pointers to the concatenated string (former case) or verbatim in w-bits (latter case). Since the number of total nodes stored in the DicChilds, the DicHandles and the DicChunks is $O(k)$, the data structure needs in total either $n \lg \sigma + O(k \lg n)$ or $|T| \lg \sigma + \Theta(kw)$ bits.

Time Complexity Given a pattern P of length m, we can traverse the macro trie by visiting at most m/α macro trie nodes to find the micro trie τ storing the node whose extent has the longest common prefix with P. After reaching τ, we can compute the handle of a node from its extent in constant time, since the 2-fattest number in $[\ell..r]$ is the integer $(-1 << \texttt{msb}((\ell - 1) \oplus r)) \,\&\, r$, where <<, $\texttt{msb}$, $\oplus$ and $\&$ denote the bitwise left shift, the function retrieving the most significant bit, the bitwise exclusive-`OR` and the bitwise `AND` operators, respectively. In total, we query $O(m/\alpha)$ DicChunks, τ's DicHandle $O(\lg \alpha)$ times, and DicChild$_{\text{parex}(P)}$ at most one time, yielding $O(m/\alpha + \lg \alpha)$ expected time as claimed in Theorem 3 if all dictionaries can lookup an entry in constant expected time. Choosing a suitable representation for DicHandle,

DicChild, and DicChunk is the major task of the next subsection dealing with practical aspects of c-trie++.

4.2.3 Implementation Techniques

On the practical side, our major improvements are based on the following ideas:

- Representing each node by an identifier (ID) to store IDs instead of node pointers.
- Storing a global mapping from extent(v) to node IDs.
- Represent the dictionaries with different data structures with focus on either speed or memory efficiency.

Micro Tries Each node v stores its extent extent(v), which can be represented in a constant number of computer words. From extent(v) we can deduce handle(v) and key(v) in constant time. Therefore, the dictionaries DicChild and DicHandle have no need to store the keys of their entries, as they both only have to maintain the nodes with which a dictionary can restore the respective keys on demand. That said, a lookup of a node v with a key handle(v) (resp. key(v)) needs to compute handle(w) (resp. key(w)) of each node w in question for comparison. By conducting this check, the benefits of current processors featuring large cache lines become negligible in this context. Here, we embrace the cuckoo hashing [85] technique, which has strong theoretical results in the pointer machine model.

Cuckoo Hashing Our cuckoo hash table H uses three hash functions. For faster hashing, we restrict the hash table size $|H|$ to be a power of two. This allows us to map a hash value to $[1..|H|]$ more quickly by using bit shifts instead of a modulo operation (cf. the discussion in [95, Section 1]; however, new techniques [66] can speed this up). An insertion collision occurs if each of the entries located by the hash functions is already occupied. Given such a collision on inserting an element e, we start a random walk by selecting the i-th hash functions h_i for a random i, swapping $H[h_i]$ with e and recurse. If this walk is unsuccessful after a certain number of steps, the hash table doubles its size. To keep the memory requirement at minimum, the chosen hash functions are determined at startup and are the same across all cuckoo hash tables. The hash functions are based on three xorshift operations borrowed from MurmurHash[10] and

[10] `https://github.com/aappleby/smhasher/wiki/MurmurHash3`

two multiplications with different 64-bit integer seeds. Unwisely chosen seeds can result in a failure of the data structure, as the hash functions are immutable (changing would cause to rehash *all* cuckoo hash table instances). However, this was not a problem in our experiments. While insertions take $O(1)$ expected time for a sufficiently small *load factor*, i.e., the maximum ratio between the number of stored elements and $|H|$ before doubling the size of H, a lookup takes $O(1)$ worst case time. The load factor does not have much influence on the final size, since a higher load factor makes it more probable that an insertion collision exceeds the threshold of maximal iterations. Setting this threshold to a smaller value boosts the insertion speed at the expense of a higher risk of creating an unnecessarily large table. However, preliminary experiments were in favor for a small threshold around 100 iterations. For the experiments in the following section, we fixed it to 100, and set the load factor to 0.9.

Node Factory In our setting, we assume that k is much small than n. Otherwise, c-trie++ becomes unfavorable with respect to other trie data structures like the Bonsai trie. That is because our trie data structure contains $\Theta(k)$ nodes in total. However, using w bits for a pointer to a node is wasteful. Instead, we want to store node pointers in $\Theta(\lg k)$ bits as hinted in the description of our computational model in Chapter 2. For that, we store each node in a global two-dimensional array that assigns each node an integer represented in $\Theta(\lg k)$ bits, which we set to 32 bits for the experiments. By storing 32-bit integers instead of pointers on commodity computers with a word size of $w = 64$ bits, we can roughly halve the memory requirement for maintaining DicChild and DicHandle.

Macro Trie Like for DicHandle, we use a cuckoo hash table for representing the DicChunks. We again just store the nodes in the cuckoo hash table, since we can restore their keys by extracting the respective w bit substring in constant time. We also maintain a separate node factory storing the macro trie nodes.

Practical Considerations In practice, the Cuckoo hash tables used for representing the dictionaries DicChild waste non-negligible space as (a) each micro trie node stores such a hash table, and (b) the hash tables may not always become full. For space efficiency, we did not follow this approach, but instead represent all DicChild dictionaries of a micro trie with a single trie data structure in first-child next-sibling representation (see [68] for a definition). In this representation, we maintain two arrays for (a) the first children and (b) the next siblings, where (a)

Table 4.1: Characteristics of our keyword sets. The total length of all keywords is n. The number of keywords is k. The average and maximum length of a keyword is written in the columns *∅len* and *max-len*, respectively. The columns *∅LCP* and *max-LCP* show, respectively, the average length and the maximal length of the longest common prefixes of all keywords. The number of nodes a compact trie C stores is given by $|C|$. The packed c-trie, the z-fast trie, and c-trie++ have the same number of nodes.

$\mathcal{K}$	$\frac{n}{10^6}$	σ	$\frac{k}{10^3}$	∅len	max-len	∅LCP	max-LCP	$\frac{\lvert T\rvert}{10^6}$	$\frac{\lvert C\rvert}{10^3}$
`proteins`	903	26	2,982	302.8	36,805	38.8	16,190	787	5,778
`urls`	1,413	98	18,564	76.1	2,048	60.9	2,006	282	35,343
`dblp.xml`	169	96	2,950	57.6	685	34.4	104	68	5,900
`geographic`	107	134	7,308	14.6	151	8.5	247	45	12,802
`commoncrawl`	121	113	1,995	61.0	1,194,988	12.9	119,276	96	3,740
`vital`	243	203	494	493.3	9,794	12.7	1,806	238	986

and (b) are pointers gained from the node factory. For navigation in the first-child next-sibling representation it is necessary to know the character of the in-going edge of each node v, but this information is already given by querying key(v).

4.3 Experiments

Finally, we analyze the empirical performance of c-trie++ with respect to time and memory consumption. In particular, we are interested in the running time of insert, lookup, and locatePrefix. For that, we implemented c-trie++ in C++. Our implementation is available at `https://gitlab.com/habatakitai/ctriepp`. For the experiments, we set up a machine equipped with CentOS 6.10, with an Intel Xeon X5560 processor running at 2.80 GHz, and with 198GB of main memory.

4.3.1 Datasets

For an objective evaluation, we took a variety of data sets having different characteristics (cf. Table 4.1):

- `proteins` contains different sequences of amino acids.
- `dblp.xml` is part of the XML dump of the `dblp.org` website.
- `urls` is a crawl of webpages of the `.uk` domain from the WebGraph framework[11].

[11] `http://law.di.unimi.it/webdata/uk-2002`

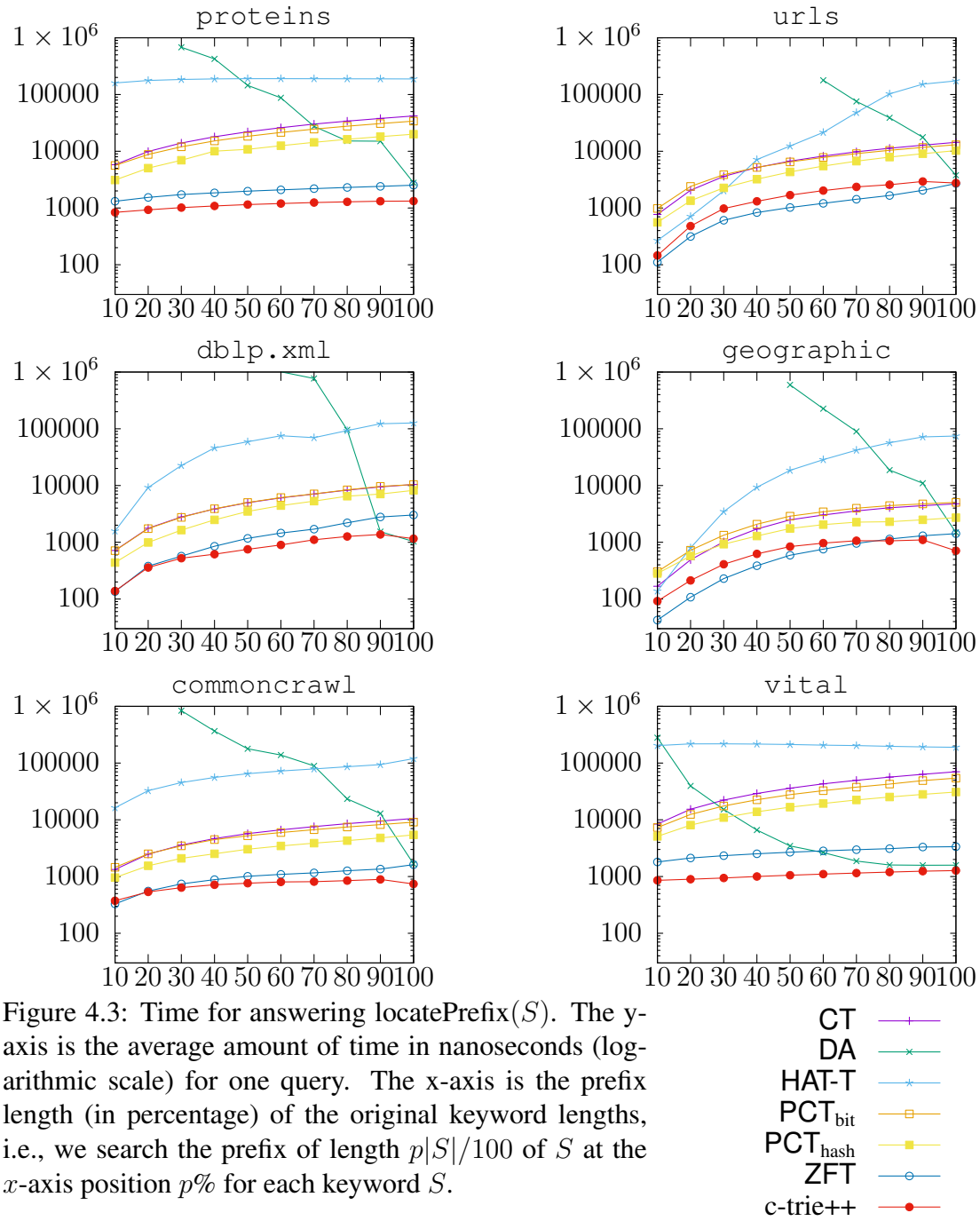


Figure 4.3: Time for answering locatePrefix(S). The y-axis is the average amount of time in nanoseconds (logarithmic scale) for one query. The x-axis is the prefix length (in percentage) of the original keyword lengths, i.e., we search the prefix of length $p|S|/100$ of S at the x-axis position $p\%$ for each keyword S.

Table 4.2: Insertion of all keywords in *random* order. We measured (a) the average time per keyword and (b) the memory needed for inserting all keywords of the respective data set. The (a) fastest time and the (b) lowest memory footprint for each keyword set and for each group of contestants (compact tries or double array tries) is highlighted in bold font. For each instance, we measured the maximal virtual memory resident set size (VmRSS), which is the second integer in the file `/proc/self/statm`.

(a) Time in Nanoseconds

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
`proteins`	45,508.6	45,041.0	51,994.3	3,683.2	**2,349.2**	2,088.1	**1,805.5**
`urls`	13,459.0	10,580.3	8,659.0	**4,216.7**	4,646.1	2,702.8	**1,228.9**
`dblp.xml`	10,066.5	8,595.6	8,413.1	3,309.3	**3,035.1**	**1,202.8**	1,371.4
`geographic`	4,711.8	4,791.4	4,548.5	**2,223.4**	2,427.5	961.6	**595.6**
`commoncrawl`	11,077.5	11,029.6	12,269.6	2,368.5	**2,260.2**	904.9	**824.3**
`vital`	71,666.6	75,433.8	96,319.3	3,515.9	**2,002.2**	**1,869.5**	2,151.1

(b) Memory in Megabytes

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
`proteins`	3,053.12	3,053.12	4,424.64	549.88	**418.10**	2,142.68	**892.66**
`urls`	8,551.47	8,551.48	9,465.49	3,731.14	**2,046.61**	**932.01**	1,317.75
`dblp.xml`	1,450.76	1,450.77	1,871.57	552.14	**305.74**	187.41	**144.77**
`geographic`	3,029.85	3,029.86	5,252.34	1,204.07	**719.36**	234.96	**164.29**
`commoncrawl`	1,040.79	1,040.77	1,685.80	330.03	**214.35**	269.03	**140.81**
`vital`	743.69	743.70	1,130.68	84.29	**58.12**	322.22	**239.12**

Table 4.3: Average time for lookup(K) in nanoseconds. We created a list L storing all keywords $K \in \mathcal{K}$, and shuffled it. We measured the time of a linear scan over L during which we locate each visited keyword in the respective trie created in Table 4.2, and divided this time by $|\mathcal{K}|$, which yields the average times shown in this table.

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
`proteins`	42,199.7	33,678.0	20,011.6	2,530.4	**1,332.2**	1,413.3	**609.0**
`urls`	14,411.8	13,087.0	10,279.9	3,067.1	**2,801.6**	2,624.1	**559.1**
`dblp.xml`	10,454.9	8,990.6	6,869.7	2,205.5	**1,161.4**	989.6	**439.6**
`geographic`	4,764.3	5,016.1	2,726.1	1,449.5	**711.1**	423.0	**243.6**
`commoncrawl`	10,667.9	9,071.7	5,423.6	1,646.8	**742.4**	636.8	**299.6**
`vital`	71,552.6	52,391.1	29,774.7	2,806.9	**1,204.6**	1,138.9	**682.6**

- `geographic` contains names of different geographic locations collected by the GeoNames database[12]. Our keywords are extracted from the `ascii name` column.
- `commoncrawl` is a web crawl containing the ASCII-encoded content (without HTML tags) of random web pages extracted from Common Crawl[13].
- `vital` is the main text extracted from the most vital Wikipedia articles.

The data sets `proteins` and `dblp.xml` are from the Pizza&Chili Corpus[14]. The data sets `commoncrawl` and `vital` are provided by the tudocomp framework [28].

We interpreted each data set as a single string on the byte alphabet. We partitioned this string into keywords by splitting it either at newline characters or at full stops, and removed all duplicates afterwards. The resulting keyword sets are the input of our experiments.

4.3.2 Contestants

We compared c-trie++ with keyword dictionary representations featuring also a low memory footprint. We present two groups of contestants. The first group consists of trie data structures based on the double array:

- DA: the double array [2] implementation of the Cedar library[15].
- HAT-T: the HAT-trie [5] implementation of Tessil[16]. This implementation exploits that keywords have a small length in practice. The default implementation assumes that all these lengths can be stored in 16 bits, which is not true for the data set `commoncrawl`. We therefore evaluated the HAT-trie with 16 and 32 bits for the lengths, and took the minimum time and minimum space of both variants for the evaluation.

As we will see in the following, the keyword dictionaries of the first group are lightweight and overall efficient but perform prefix searches poorly. The second group consists of other compact trie data structures:

- CT: a compact trie without word packing.

[12] `http://download.geonames.org/export/dump/allCountries.zip`
[13] https://commoncrawl.org/
[14] `http://pizzachili.dcc.uchile.cl`
[15] `http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/`
[16] `https://github.com/Tessil/hat-trie`

- $\mathsf{PCT}_{\text{bit}}$: a packed c-trie using bit parallelism to compare compact words.
- $\mathsf{PCT}_{\text{hash}}$: a packed c-trie using additionally the hash table implementation `unordered_map` of the C++ standard library as a dictionary in each micro trie for retrieving a node by its extent (it is similar to our DicHandle, but uses the extents instead of the handles as keys).
- ZFT: our z-fast trie transportation from an implementation in Java[17] to C++.

The implementations of the compact trie and the packed c-tries are due to Takagi et al. [99]. The implementations $\mathsf{PCT}_{\text{bit}}$ and $\mathsf{PCT}_{\text{hash}}$ pack characters in 32-bit integers, whereas all other implementations use 64-bit integers, which reflect the machine word size of commodity computers nowadays. All implementations (of both groups) are written in C++, and compiled with `gcc-8.2.0` in the highest optimization mode `-O3`.

In what follows, we evaluate c-trie++ with our contestants on the aforementioned data sets. Our focus is set on prefix queries, as this operation is one of the main purposes for using compact tries.

4.3.3 Evaluation of the Construction

In the first experiment, we measured the time it takes to insert all keywords of a data set into a keyword dictionary in random order. We give the results in Table 4.2. This table reveals that the construction of c-trie++ is faster than the construction of every packed trie (i.e., CT, $\mathsf{PCT}_{\text{bit}}$, $\mathsf{PCT}_{\text{hash}}$, and ZFT). Except for ZFT, its final size is also an improvement to the sizes of those data structures. If the average keyword length is sufficiently large, c-trie++ is also superior to DA and HAT-T in both time and space while being inferior when maintaining mostly short keywords.

4.3.4 Evaluation of the Queries

Our next and final experiments measure the performance of lookup and locatePrefix queries.

Locate Prefix Queries A major highlight is the time needed for locatePrefix(S) queries shown in Figure 4.3. Instead of returning an iterator to a set as requested at the beginning of this chapter, we require each keyword dictionary to return the complete set of all keywords

[17] This implementation is part of Vigna's Sux4J library, located at `https://github.com/vigna/Sux4J`.

having S as a prefix. In this setting, c-trie++ dominates most of the time. Interestingly, DA becomes faster for longer prefixes. This effect can be explained as follows: First recognize by Table 4.3 that DA has competitive lookup times, allowing the trie to match a pattern at high speed. The matching locates the lowest node v whose extent is a prefix of S. After locating v, it resorts to exploring the entire subtree of v. If v is a deep node, chances are that its subtree size is rather small, enabling DA to process v's subtree quickly.

Lookup Queries The results for lookup are collected in Table 4.3. In all instances, c-trie++ answered lookup queries faster than all packed tries. However, HAT-T, followed by DA, provide the fastest solutions for answering lookup.

4.4 Detailed Space Analysis

At the beginning of this chapter, we gave the space bounds of the packed trie data structures in terms of the total length of all keywords n and in the size of the non-compact trie T. To ease the understanding, we present a tabular representation.

Trie	Space in Bits	Setting
c-trie++	$n \lg \sigma + O(k \lg n)$	1
c-trie++	$\|T\| \lg \sigma + O(kw)$	2
compact trie	$\|T\| \lg \sigma + O(k \lg \|T\|)$	2
z-fast trie [12]	$\|T\| \lg \sigma + O(kw)$	2
c-packed trie [99]	$\|T\| \lg \sigma + O(kw)$	2

In Setting 1, we concatenate all keywords to a large string of length n. In this large string, we can address every substring with two pointers of $\lg n$ bits. We omit Setting 1 for the other compact trie data structures as these (expect the plain compact trie) use auxiliary data structures taking $O(kw)$ bits. In Setting 2, we represent each keyword K with front coding [106, Section 4.1], i.e., we represent K by $K[\ell + 1..|K|]$ if the longest common prefix of K with its lexicographically preceding keyword in $\mathcal{K}$ is ℓ. Hence, we store the suffix $K[\ell + 1..|K|]$ in a string. By doing so for each keyword, we store k strings with a total length of $|T|$. To access packed characters like handle(v) in constant time, we store handle(v) (using w bits) in the node v, causing $O(kw)$ bits of additional space. In the plain compact trie, we do not augment the nodes with packed characters.

4.5 Additional Experiments

In the following, we present some additional statistics and evaluations.

Statistics The statistics in the previous section only sketch the characteristics of the used keyword sets. Here, we like to present a more profound analysis by showing different distributions in Tables 4.4,4.5a and 4.6. We see that the lengths have a distribution that is more Gaussian, and by no means uniform. The lengths have also an impact on the sizes and shapes of the dictionaries, as can be seen in Table 4.6.

Dictionary Representations of c-trie++ The distributions in Tables 4.6a and 4.6b justify our selection of a lightweight data structure with worse asymptotic behavior (sorted lists) for DicChild, and the use of the more heavyweight cuckoo hash table for DicHandle. We also did experiments with unsorted lists storing newly inserted elements at their end. These experiments showed that unsorted lists feature a small speed-up for tiny instances while becoming early slow after a number of insertions.

Sorted Insertion In the previous section, we covered the case of creating a trie on keywords shuffled in a random order R, and subsequently queried the trie with the keywords in another random order R'. However, one might question whether other possibilities like building a keyword dictionary with lexicographically sorted keywords, or querying it with keywords arranged in the same order as in the construction is advantageous. For that, we revisit the construction in Table 4.7, filling a keyword dictionary now with keywords in lexicographically sorted order. Comparing to Table 4.2, the space requirement in both scenarios is nearly the same for each keyword dictionary. However, a lexicographically sorted insertion speeds up the construction of all of instances.

More Queries Having two scenarios for trie construction, we can also think about different orders of how to query the data structures. Here, we present a Cartesian product of these orders, shown in Table 4.8 for lookup, and in Figures 4.4, 4.5, 4.6, and 4.7 for locatePrefix. We see a remarkable speedup of the query operations of all keyword dictionary implementations when they are fed with keywords in lexicographically order. The best bets can be placed on the setting of Table 4.8a and Figure 4.4. A slightly slower variant is to query in random order (Table 4.8b and Figure 4.5). The execution times of the keyword dictionaries fed in random order follow with a large gap. Here, the order in which the queries are executed has again only a slight impact on the execution times. We obtain the fastest execution times when querying the keywords in the same order as we built a keyword dictionary (Table 4.8d and Figure 4.5). ZFT and c-trie++ can take advantage of the case when the queries are in lexicographic order (Table 4.8c and Figure 4.6), while the other implementations are slightly faster in the random case (Table 4.3 and Figure 4.3).

Deletions We also ran experiments for the delete operation, which we conducted in the same fashion as the experiments for lookup. We put the results in Table 4.9.

Table 4.4: Histogram ofs keyword lengths of the longest common prefixes (LCPs) of the keywords. While Table 4.1 captures the average and maximal lengths of the keywords and their LCPs, these tables give an insight in the distributions of the lengths and the LCPs. A length is counted in the i-th row if is i for $i = 1$ and $i = 2$, or belongs in $[2^{i-2} + 1..2^{i-1}]$ for $i \geq 3$.

(a) $\#len \leftrightarrow |len|$ Histogram

i	proteins	urls	dblp.xml	geographic	commoncrawl	vital
1	19	85	2	11	97	39
2	132	851	1	262	1,546	26
4	5,485	7,888	0	31,036	31,931	131
8	36,973	25,921	5	1,270,765	137,074	726
16	75,796	24,188	25	3,899,303	636,922	2,298
32	66,530	197,634	395,244	1,838,186	445,153	4,932
64	130,527	8,620,706	1,801,952	263,086	369,674	12,007
128	481,117	8,463,502	723,011	5,398	255,830	32,038
256	818,538	1,100,909	29,782	7	61,018	75,871
512	955,403	100,867	213	0	36,936	166,775
1,024	343,983	19,207	2	0	11,627	165,169
2,048	57,653	2,946	0	0	4,464	33,599
4,096	8,691	0	0	0	1,878	857
8,192	1,145	0	0	0	795	14
16,384	83	0	0	0	256	1
32,768	15	0	0	0	99	0
65,536	2	0	0	0	52	0
131,072	0	0	0	0	39	0
262,144	0	0	0	0	5	0
524,288	0	0	0	0	3	0
1,048,576	0	0	0	0	2	0
2,097,152	0	0	0	0	1	0

Original z-fast trie The original implementation of the z-fast trie of Vigna is written in Java as part of his Sux4J library. As a supplement, we conducted our experiments of this implementation on the same machine. However, we could not build this trie for the keyword set `vital`. The time and space needed for the trie construction are given in Table 4.10. Its time for lookup and locatePrefix are shown in Table 4.11 and Figure 4.8, respectively. Its time for delete is given in Table 4.12. Unfortunately, we received runtime failures on several instances, which we marked with *N/A* (for not available) in the experiments.

Table 4.5: Histogram of the lengths of the longest common prefixes (LCPs) of the keywords. While Table 4.1 captures the average and maximal lengths of the keywords and their LCPs, these tables give an insight in the distributions of the lengths and the LCPs. A length is counted in the i-th row if is i for $i = 1$ and $i = 2$, or belongs in $[2^{i-2} + 1..2^{i-1}]$ for $i \geq 3$.

(a) $\#LCP \leftrightarrow |LCP|$ Histogram

i	proteins	urls	dblp.xml	geographic	commoncrawl	vital
0	22	91	2	84	101	111
1	490	2,633	19	2,225	6,012	1,850
2	9,014	11,115	20	19,636	50,615	6,079
4	470,608	29,492	5	635,924	306,121	28,013
8	1,432,010	26,723	2,663	3,838,361	574,787	118,627
16	203,019	76,180	556,906	2,457,041	780,370	240,884
32	207,474	1,459,143	862,179	319,203	173,276	92,179
64	205,067	10,668,966	1,398,593	34,830	77,137	4,730
128	204,307	5,814,357	129,715	749	21,026	1,043
256	155,849	429,835	134	0	3,870	559
512	73,927	37,058	0	0	1,247	309
1,024	17,440	8,263	0	0	507	93
2,048	2,468	847	0	0	193	5
4,096	335	0	0	0	48	0
8,192	60	0	0	0	18	0
16,384	1	0	0	0	70	0
32,768	0	0	0	0	0	0
65,536	0	0	0	0	0	0
131,072	0	0	0	0	3	0

Table 4.6: Histogram of (a) micro tries or (b) internal micro trie nodes storing a specific number of (a) child nodes or (b) internal nodes representing the sizes of (a) all DicHandle instances or (b) all DicChild instances. A (a) micro trie or (b) internal node is counted in the i-th row if the number of its stored nodes is i for $i = 1$ and $i = 2$, or in $[2^{i-2} + 1..2^{i-1}]$ for $i \geq 3$. None of the keyword sets is prefix-free, as can be seen by the fact that there are nodes with only a single child.

(a) #DicHandle ↔ |DicHandle| Histogram

i	proteins	urls	dblp.xml	geographic	commoncrawl	vital
1	692,786	1,996,651	233,983	474,823	180107	57649
2	72,926	419,911	46,975	126,163	36273	13791
4	26,863	278,813	27,291	70,145	19255	8641
8	7,696	143,852	16,392	30,265	8500	4097
16	1,705	66,594	1,1386	13,357	3449	1651
32	420	27,161	6,411	6,424	1195	618
64	89	11,108	3,214	2,952	488	254
128	24	4,574	1,152	1,241	194	105
256	5	1,633	302	472	100	25
512	1	580	110	191	38	13
1024	1	75	37	68	37	3
2048	0	0	18	21	1	0
4096	0	0	8	9	0	0
8192	0	0	4	0	0	0
16384	0	1	0	1	0	0
32768	0	1	0	0	0	0
65536	0	0	0	0	0	1
131072	0	0	1	0	0	0
262144	0	0	0	0	0	0
524288	0	0	0	0	1	0
1048576	1	0	0	0	0	0

(b) #DicChild ↔ |DicChild| Histogram

i	proteins	urls	dblp.xml	geographic	commoncrawl	vital
1	27,933	189,554	106	204,565	30,276	279
2	1,220,896	3,939,539	808,559	1,644,531	468,330	164,154
4	231,439	1,594,225	313,011	716,500	175,809	54,646
8	86,483	886,825	116,020	288,994	69,654	19,579
16	42,571	507,437	53,258	104,526	47,619	6,272
32	13,894	34,609	14,298	28,445	6365	1,466
64	0	1,221	656	301	1,201	283
128	0	5	7	8	124	7

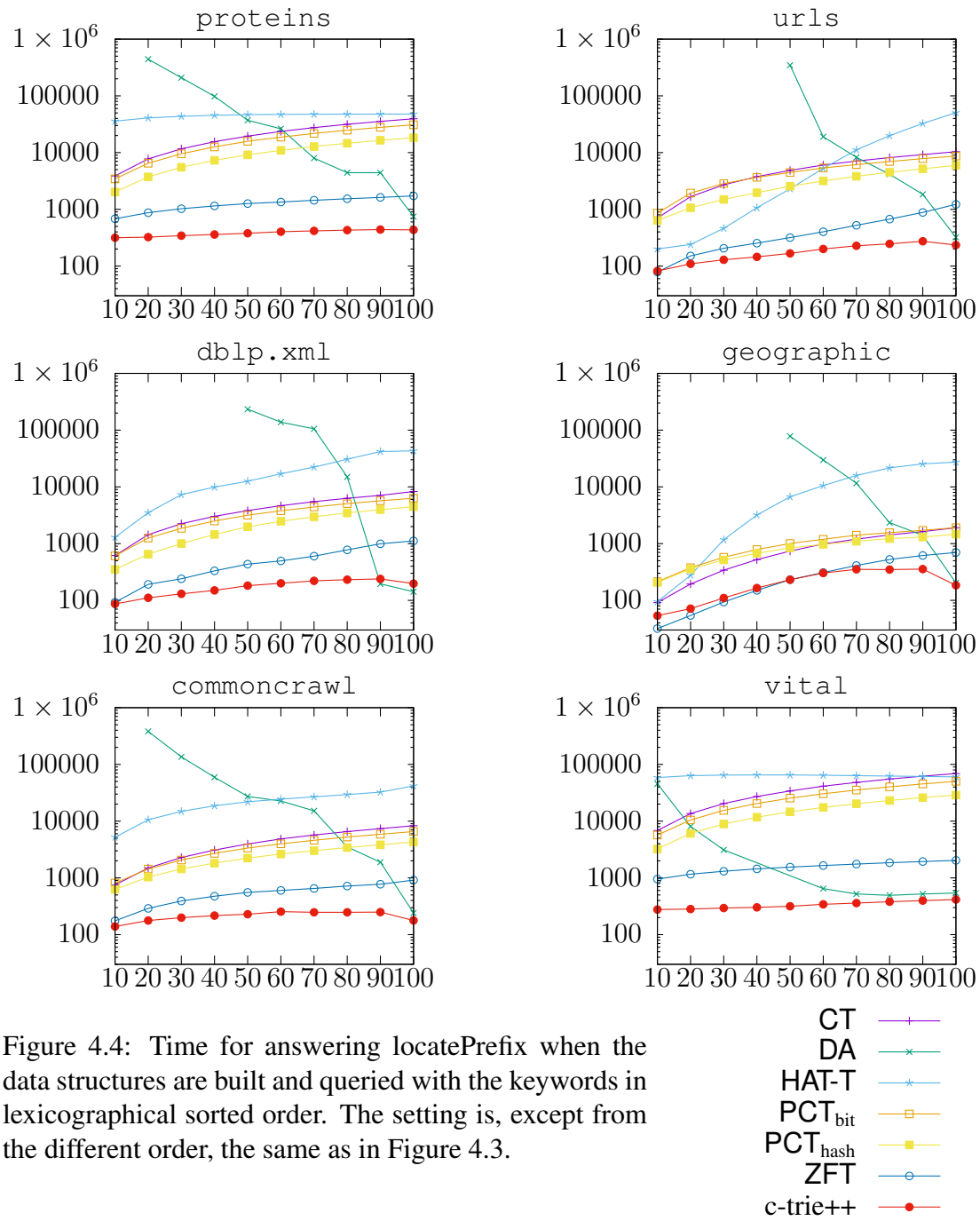


Figure 4.4: Time for answering locatePrefix when the data structures are built and queried with the keywords in lexicographical sorted order. The setting is, except from the different order, the same as in Figure 4.3.

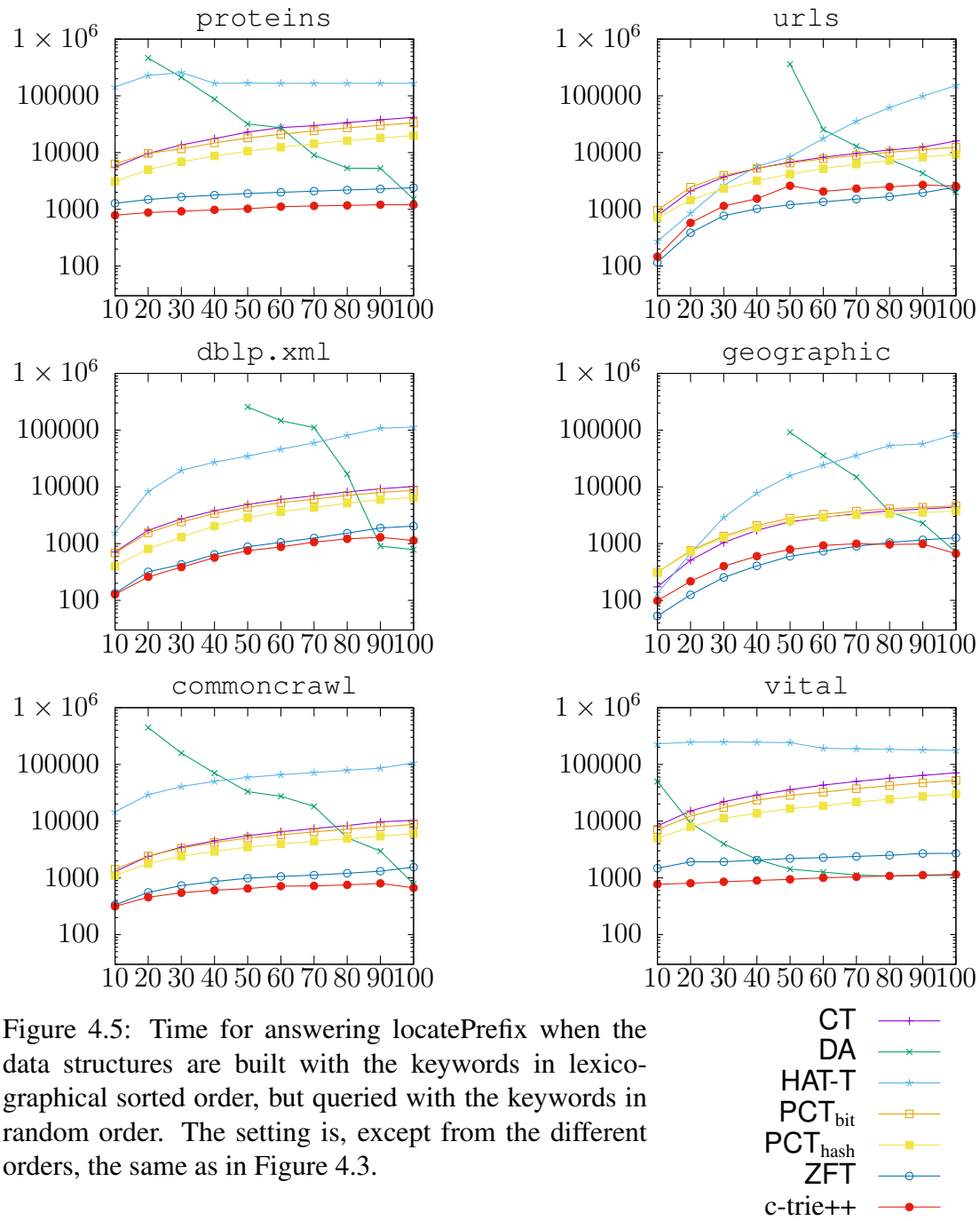


Figure 4.5: Time for answering locatePrefix when the data structures are built with the keywords in lexicographical sorted order, but queried with the keywords in random order. The setting is, except from the different orders, the same as in Figure 4.3.

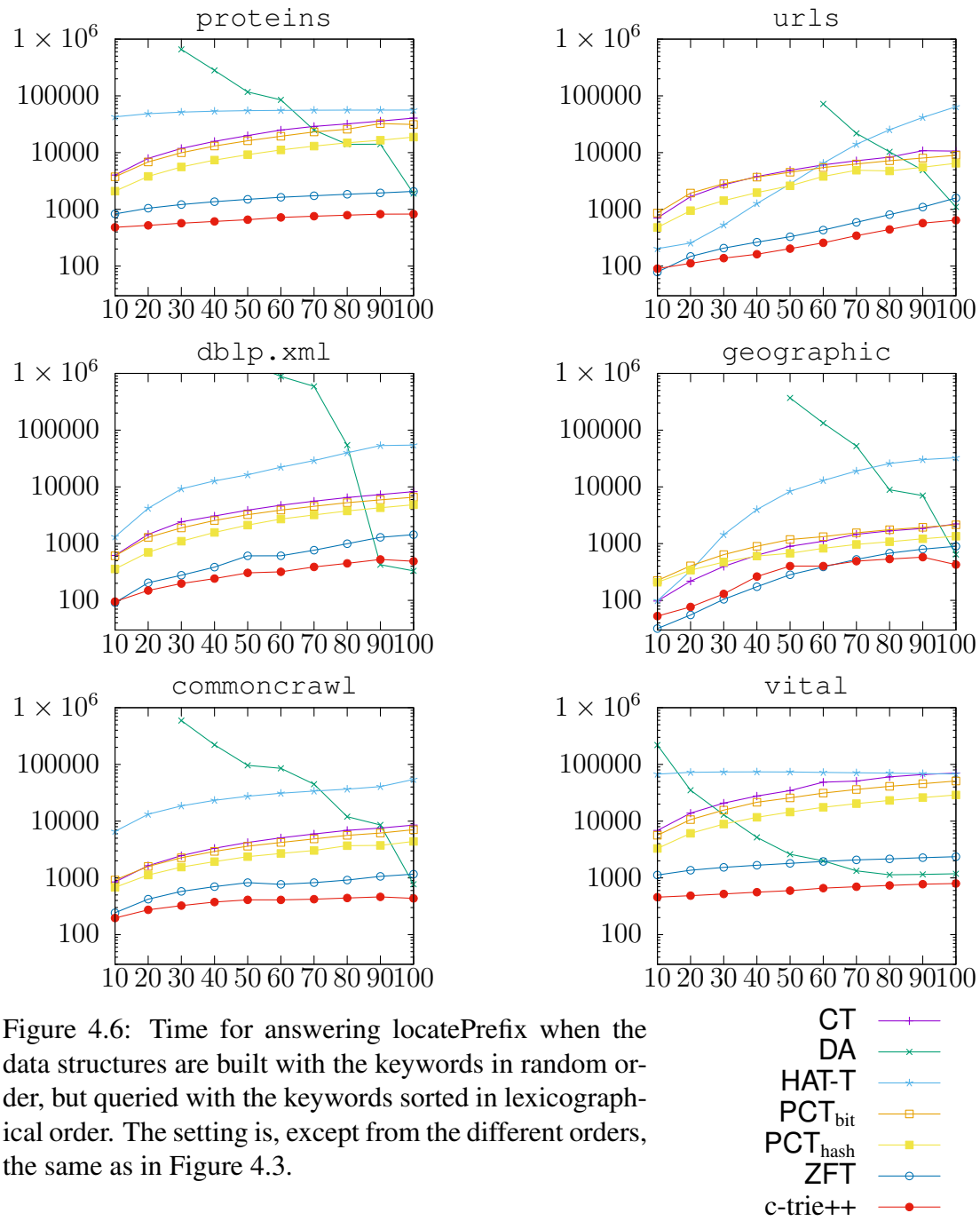


Figure 4.6: Time for answering locatePrefix when the data structures are built with the keywords in random order, but queried with the keywords sorted in lexicographical order. The setting is, except from the different orders, the same as in Figure 4.3.

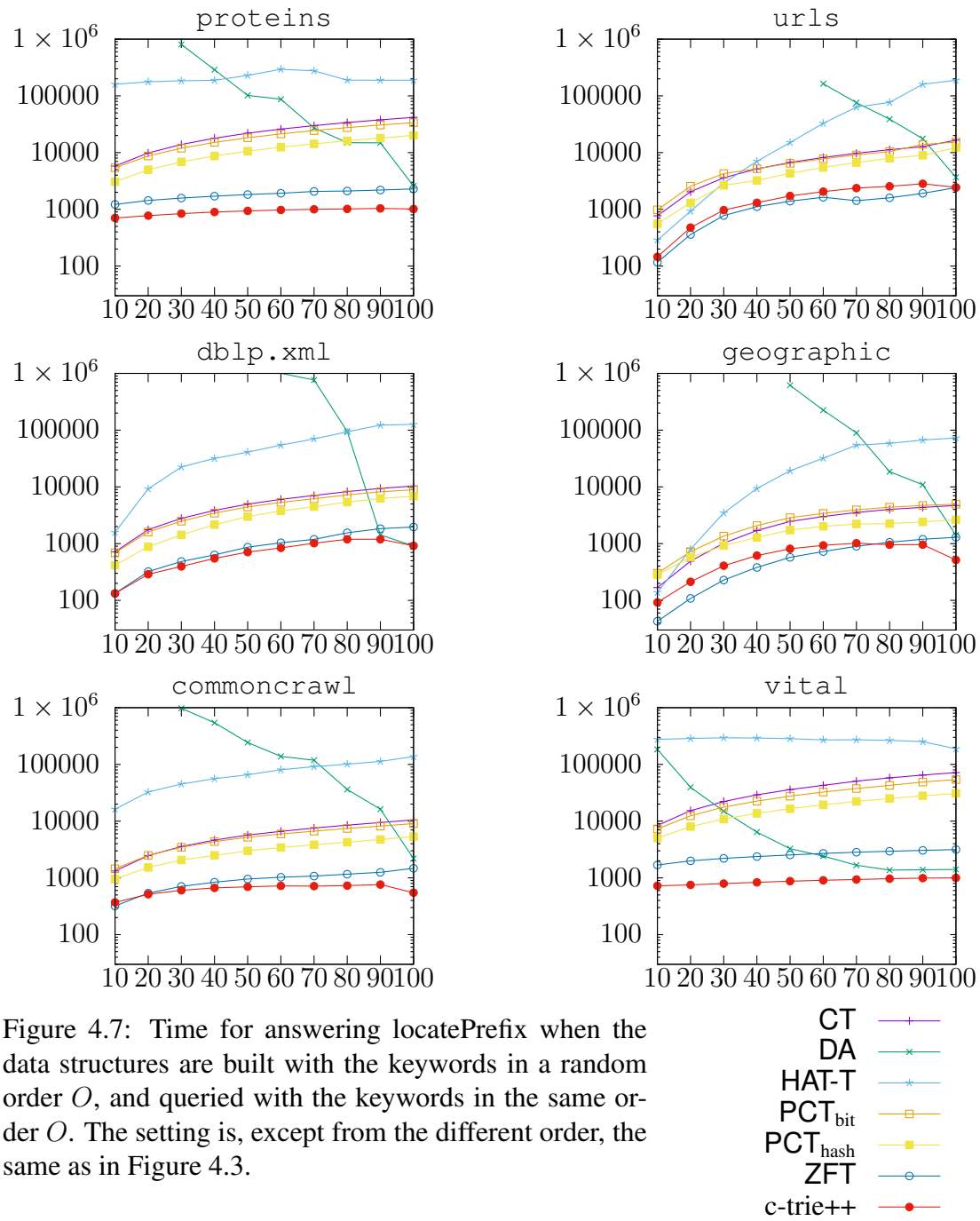


Figure 4.7: Time for answering locatePrefix when the data structures are built with the keywords in a random order O, and queried with the keywords in the same order O. The setting is, except from the different order, the same as in Figure 4.3.

Table 4.7: Insertion of all keywords in *lexicographical* order. Except to the ordering of the keywords, the setting is the same as in Table 4.2.

(a) Time in Nanoseconds

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	39,716.6	38,547.4	48,384.0	2,623.4	**1,369.1**	1,225.3	**853.3**
urls	9,849.2	6,398.6	4,786.9	2,604.1	**709.5**	**610.9**	480.7
dblp.xml	7,736.4	5,713.0	5,645.8	2,051.3	**736.6**	**451.9**	810.4
geographic	2,342.1	2,089.6	2,605.7	1,305.1	**1,035.8**	**237.0**	258.3
commoncrawl	8,419.2	8,012.2	9,930.3	1,485.4	**1,072.3**	**370.3**	385.4
vital	63,719.1	65,684.8	90,066.2	3,187.2	**865.9**	1,313.2	**1,266.6**

(b) Memory in Megabytes

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	2,889.31	2,889.32	4,376.2	549.87	**422.68**	1,779.47	**890.14**
urls	8,533.40	8,533.41	10,027.4	3,731.14	**2,046.45**	**1,017.18**	1,302.21
dblp.xml	1,445.39	1,445.40	1,850.2	552.14	**305.70**	173.62	**141.59**
geographic	3,029.50	3,029.51	4,952.8	1,204.07	**719.35**	251.86	**159.23**
commoncrawl	1,023.88	1,023.87	1,598.8	330.03	**220.18**	174.45	**139.61**
vital	695.96	695.97	1,098.4	84.29	**58.12**	261.09	**238.09**

Table 4.8: Average time for lookup(K) in nanoseconds. We create a trie by inserting keywords contained a list L whose elements are (a-b) lexicographically sorted or (c-d) in a random order R. We stick to the setting of Table 4.8, where we used L for the queries. However, before the querying, we (b) shuffled L, (a,c) sorted the elements in L lexicographically, or (d) kept L as it is.

(a) Sorted - Sorted

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	39,357.5	30,758.8	18,392.0	1,748.6	**421.1**	391.6	**256.0**
URLs	10,296.2	8,781.5	5,945.6	1,223.0	**240.6**	155.2	**138.7**
dblp.xml	7,957.8	6,372.7	4,481.0	1,121.3	**190.4**	**111.9**	136.0
geographic	1,839.2	1,925.0	1,436.4	717.3	**179.6**	**44.8**	65.4
commoncrawl	8,273.8	6,555.0	4,294.6	930.4	**169.7**	**95.9**	99.6
vital	69,059.5	49,871.5	28,850.4	2,046.7	**404.6**	526.3	**346.1**

(b) Sorted - Order R

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	40,154.2	31,487.0	18,817.7	2,440.1	**1,155.6**	1,084.5	**627.3**
urls	10,725.0	9,287.6	6,376.8	**2,476.8**	2,470.9	1,739.7	**575.4**
dblp.xml	8,194.8	6,609.8	4,781.5	2,054.1	**1,084.9**	746.0	**459.7**
geographic	2,054.8	2,130.7	1,376.0	1,288.4	**636.4**	353.5	**246.9**
commoncrawl	8,697.6	6,892.4	4,220.5	1,575.9	**627.5**	470.1	**305.6**
vital	71,081.0	53,701.4	29,366.9	2,726.1	**1,111.8**	1,020.7	**681.9**

(c) Order R - Sorted

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	42,934.5	33,231.1	19,988.6	2,050.3	**805.6**	691.6	**309.3**
urls	14,563.1	12,500.3	9,321.5	1,598.1	**635.6**	361.4	**177.5**
dblp.xml	10,180.9	8,702.2	6,496.8	1,451.4	**473.0**	297.9	**161.5**
geographic	4,408.0	4,665.0	3,746.0	959.1	**407.6**	162.3	**84.9**
commoncrawl	10,370.6	8,761.5	6,016.1	1,175.2	**423.4**	267.0	**123.3**
vital	71,992.7	53,526.8	30,583.7	2,341.1	**778.6**	788.0	**411.4**

(d) Order R - Order R

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	42,134.8	33,626.8	19,904.1	2,299.5	**1,016.8**	1,211.9	**605.4**
urls	14,329.6	13,008.3	10,187.0	2,462.7	**2,410.2**	2,491.6	**556.6**
dblp.xml	10,398.2	8,938.6	6,801.3	1,979.3	**920.32**	863.5	**436.3**
geographic	4,703.1	4,966.8	2,644.0	1,296.5	**501.7**	387.6	**240.4**
commoncrawl	10,624.8	9,040.9	5,353.6	1,496.8	**550.5**	553.6	**295.7**
vital	71,523.2	52,342.0	29,680.5	2,579.2	**943.7**	952.8	**665.1**

Table 4.9: Average time for delete(K) in nanoseconds. For Sub-Table (Order R - Order R'), the setting with two different random orders R and R' is the same as in Table 4.3. For the other sub-tables, the setting is giving in Table 4.8.

(a) Order R - Order R'

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	-	-	-	3,676.4	**2,012.0**	1,606.1	**1,187.7**
urls	-	-	-	5,677.7	**4,045.5**	3,060.8	**886.5**
dblp.xml	-	-	-	3,501.5	**2,219.4**	1,211.7	**667.4**
geographic	-	-	-	2,254.6	**1,761.8**	787.8	**494.3**
commoncrawl	-	-	-	2,526.7	**1,645.4**	868.5	**573.1**
vital	-	-	-	3,727.4	**1,780.8**	**1,042.0**	1,302.7

(b) Sorted - Sorted

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	-	-	-	2,457.1	**875.4**	**476.1**	568.3
urls	-	-	-	2,471.2	**526.4**	**211.6**	293.1
dblp.xml	-	-	-	2,059.1	**520.6**	**153.3**	236.3
geographic	-	-	-	1,161.2	**608.1**	**77.6**	143.4
commoncrawl	-	-	-	1,465.1	**575.0**	**129.5**	207.6
vital	-	-	-	2,704.5	**813.3**	**437.0**	733.8

(c) Sorted - Order R

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	-	-	-	2,455.5	**874.1**	**476.0**	567.4
urls	-	-	-	2,476.7	**525.6**	**211.5**	292.6
dblp.xml	-	-	-	2,065.0	**522.4**	**153.4**	237.1
geographic	-	-	-	1,158.1	**608.3**	**77.6**	144.1
commoncrawl	-	-	-	1,561.8	**574.5**	**129.5**	207.5
vital	-	-	-	2,692.4	**814.1**	**435.7**	764.0

(d) Order R - Sorted

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	-	-	-	3,651.9	**2,012.4**	1,603.5	**1,187.7**
urls	-	-	-	4,257.0	**3,976.2**	3,051.1	**883.9**
dblp.xml	-	-	-	3,493.1	**2,141.1**	1,221.0	**671.1**
geographic	-	-	-	2,296.8	**1,750.3**	782.9	**494.6**
commoncrawl	-	-	-	2,513.5	**1,632.9**	875.8	**568.9**
vital	-	-	-	3,701.9	**1,781.7**	**1,026.0**	1,305.5

(e) Order R - Order R

$\mathcal{K}$	CT	PCT_{bit}	PCT_{hash}	ZFT	c-trie++	DA	HAT-T
proteins	-	-	-	3,959.9	**2,006.1**	1,607.1	**1,194.2**
urls	-	-	-	4,340.2	**3,986.6**	3,061.0	**885.7**
dblp.xml	-	-	-	3,438.7	**2,147.2**	1,227.4	**668.4**
geographic	-	-	-	2,236.0	**1,765.64**	783.1	**493.7**
commoncrawl	-	-	-	2,516.1	**1,629.7**	878.6	**570.9**
vital	-	-	-	3,696.0	**1,779.0**	**1,041.6**	1,297.9

Table 4.10: Inserting of all keywords in the z-fast trie Java-implementation.

(a) Time in Nanoseconds

$\mathcal{K}$	Random	Sorted
proteins	3,896.6	2,764.8
urls	3,056.7	2,038.7
dblp.xml	2,727.1	1,693.6
geographic	2,802.9	1,831.0
commoncrawl	2,883.3	1,714.9
vital	N/A	N/A

(b) Memory in Megabytes

$\mathcal{K}$	Random	Sorted
proteins	1,629.60	1,630.81
urls	2,764.73	2,341.69
dblp.xml	989.38	1,026.62
geographic	1,043.65	1,075.91
commoncrawl	244.94	245.73
vital	N/A	N/A

Table 4.11: Average time for answering lookup(K) with the z-fast trie Java-implementation. Times are in nanoseconds. The table covers the settings of Tables 4.3 (Order R - Order R'), 4.8a (Sorted - Sorted), 4.8c (Order R - Sorted), 4.8b (Sorted - Order R), and 4.8d (Order R - Order R), where R and R' are two different random orderings.

$\mathcal{K}$	R-R'	S-S	S-R	R-S	R-R
proteins	5,093.5	4,798.0	5,403.4	5,136.4	5,052.2
urls	2,384.2	1,655.2	2,615.3	1,730.9	2,438.1
dblp.xml	1,778.6	1,322.2	1,848.7	1,265.9	2,165.1
geographic	1,254.7	749.2	1,416.0	1,154.6	1,233.8
commoncrawl	2,032.3	1,351.9	1,870.8	1,404.8	1,648.0
vital	N/A	N/A	N/A	N/A	N/A

Table 4.12: Average time for answering delete(K) with the z-fast trie Java-implementation. The meaning of the column captions is the same as in Table 4.11.

$\mathcal{K}$	R-R'	S-S	S-R	R-S	R-R
proteins	5358.4	4173.3	4899.6	4205.4	4421.9
urls	4412.0	2904.0	3997.9	2848.4	4253.2
dblp.xml	N/A	N/A	N/A	N/A	N/A
geographic	2476.4	1223.1	1525.9	2690.6	2344.4
commoncrawl	N/A	N/A	N/A	N/A	N/A
vital	N/A	N/A	N/A	N/A	N/A

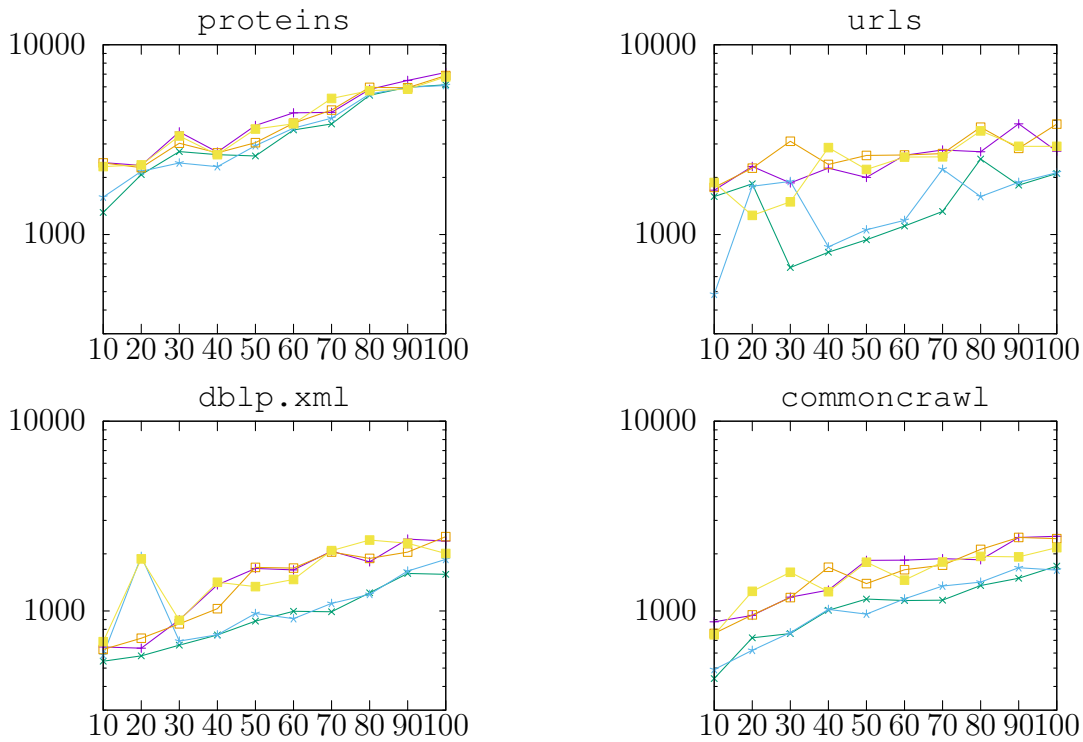


Figure 4.8: Time for answering locatePrefix with the z-fast trie Java-implementation. The plots cover the settings of Figures. 4.3 (Order R - Order R'), 4.4 (Sorted - Sorted), 4.5 (Order R - Sorted), 4.6 (Sorted - Order R), and 4.7 (Order R - Order R), where R and R' are two different random orderings.

4.6 Conclusion

We have presented the trie data structure c-trie++ to cope with the demands for fast prefix searches like auto-completion [18]. In case that prefix queries dominate dynamic operations like insertions with respect to their quantity, the keyword dictionary c-trie++ offers one of the best trade-offs among all tested candidates.

4.6.1 Future Work

We can speed up the insertions of keywords that share long prefixes with other keywords by vectorization. That is because the word packing approach for comparing two strings interpreted as two packed strings can be vectorized. Recent instruction sets like AVX feature instructions for this task.

Table 4.6a reveals that some instances of DicHandle grow extremely large while most of the other instances maintain only few entries. For the large ones, we can use a compact hash tablethat stores quotients instead of the values, where a quotient has bit length $v - \lg |H|$ if the values can be represented in v bits (we set v to 32 bits in Section 4.2.3).

Considering different hash table layouts, we conducted an experiment with the linear probing hash table of Rigtorp[18] storing nodes along with the (redundant) keys. While using much more space, this hash table performed only slightly better than the cuckoo hash table, even with a load factor of 0.5. Dropping the keys as we did in Section 4.2.3, a hash table with linear probing will likely be outperformed by our cuckoo hash table as cache effects become negligible.

Table 4.6 reveals that none of our data sets is prefix-free. In a more enhanced evaluation, we would like to conduct our experiments after a preprocessing step in which we discard every keyword that is a prefix of another keyword.

[18] `https://github.com/rigtorp/HashMap`

Chapter 5

Shortest Unique Substrings Queries in Optimal Time

The shortest unique substring problem was proposed by Pei et al. [86]. Given a string S and position p, the problem is to find a *shortest unique substring* (SUS) of S that contains position p, that is, a substring that only occurs once in S, and whose occurrence contains position p. They also consider a version of the problem where S may be preprocessed, and SUS queries for arbitrary positions may be answered efficiently.

For the first version of the problem, Pei et al. [86] presented an algorithm that computes the SUS for any given position p in $O(n)$ time and space, where n is the length of string S. For the second version, they present an $O(hn)$ time and $O(n)$ space preprocessing algorithm which allows queries to be answered in constant time, where h is a value depending on S. However, h is only bounded by $O(n)$, and in the worst case, this results in $O(n^2)$ time pre-processing.

The contributions of this chapter is as follows: First, we give optimal time solutions for both problems and show that S can be preprocessed in $O(n)$ time so that a SUS for any query position can be answered in $O(1)$ time. This considerably improves the theoretical worst case running time compared to Pei et al. [86], allowing us to output a SUS for *all* positions in the string in $O(n)$ total time. Second, we consider the general problem of computing all SUSs that contain a given position. Although there can be multiple shortest substrings that contain a given query position, Pei et al. [86] only considered the problem of answering a single SUS that contains a position. We show that the same linear time pre-processing above also allows us to return *all* SUSs that contain a given query position in $O(k)$ time, where k is the size of the output. Finally, we implement our algorithm and show through computational experiments that our algorithm is much more practical and scalable compared to an the algorithm by Pei et al.

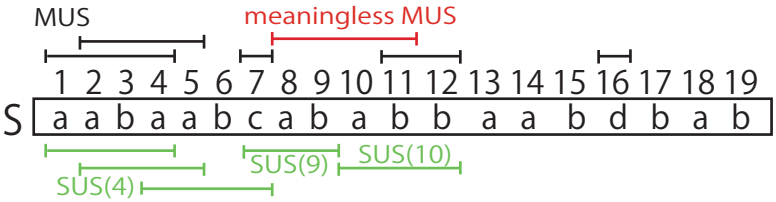


Figure 5.1: Example of a string and its SUSs (see Definition 4) and MUSs (see Definition 5). Although all 6 MUSs are depicted, $SUS(p)$ is depicted only for positions $4, 9$ and 10. $MUS_S = \{[1..4], [2..5], [7..7], [8..11], [11..12], [16..16]\}$. $SUS_S(4) = \{[1..4], [2..5], [4..7]\}$, $SUS_S(9) = \{[7..9]\}$, $SUS_S(10) = \{[10..12]\}$. The MUS $[8..11]$ is meaningless since no SUS contains it, while the others are meaningful (see Definition 10).

This result primarily appeared in [100].

5.1 Preliminaries

5.1.1 Unique Substrings

We say that a substring x of S is *unique*, if there is exactly one occurence of x in S. When x is unique, the interval $[i..i + |x| - 1]$ such that $S[i..i + |x| - 1] = x$ is called a unique interval of S. We say that a unique substring x of S contains position p, if $x = S[i..i + |x| - 1]$ and $p \in [i..i + |x| - 1]$. It is easy to see that any string that contains a substring that is unique, is also unique, and any interval that contains a sub-interval that is unique, is also unique.

Definition 4 (Shortest Unique Substring)**.** *A substring x is a shortest unique substring (SUS) of S that contains position p, if $x = S[i..j]$ is unique in S, $i \leq p \leq j$, and no other substring $x' = S[i'..j']$ such that $i' \leq p \leq j'$ and $j' - i' < j - i$ is unique in S.*

Note that there can be more than one SUS that contains position p as shown in the following example. Let $SUS_S(p)$ denote the set of intervals corresponding to SUSs of S that contains position p. Note that $SUS_S(p) \neq \emptyset$ for any position $1 \leq p \leq |S|$.

Example 2 (SUS)**.** Let S = `aabaabcababbaabdbab`. Then, $SUS_S(2) = \{[1..4], [2..5]\}$, $SUS_S(4) = \{[1..4], [2..5], [4..7]\}$, $SUS_S(9) = \{[7..9]\}$, $SUS_S(10) = \{[10..12]\}$. (See Figure 5.1)

In this chapter, we focus on the following problems.

Problem 1 (SUS query)**.** *Given string S of length n, compute for all positions p $(1 \leq p \leq n)$, a shortest unique substring that contains position p.*

Problem 2 (All SUS query)**.** *Given string S of length n, compute for all positions p ($1 \leq p \leq n$), all shortest unique substrings that contain position p.*

Problem 1 was first considered by Pei et al. [86]. They first gave a simple $O(n)$ time algorithm for computing an SUS for a single p. However, this would result in $O(n^2)$ time for computing a SUS for each p. Thus, they further showed an improved algorithm which pre-process S in $O(hn)$ time, and allows queries for any p in $O(1)$ time, where h is a parameter that depends on S. This results in an $O(hn)$ time solution for computing the SUS for all positions $1 \leq p \leq n$. Although Pei et al. [86] gave empirical evidence that h is not very large in practice, they were not able to give a good theoretical bound on h, mentioning that h can be as large as $O(n)$, resulting in $O(n^2)$ time worst case pre-processing time.

In this chapter, we give optimal time solutions for both problems, and show that S can be preprocessed in $O(n)$ time so that the queries can be answered in $O(k)$ time, for any query position p, where k is the size of the output. Noting that k is $O(1)$ for Problem 1, this results in an $O(n)$, i.e. a truly linear time solution for computing the SUS for all positions $1 \leq p \leq n$.

Like the algorithm by Pei et al. [86], our algorithm finds SUSs, based on the concept of Minimal Unique Substrings defined below.

Definition 5 (Minimal Unique Substring)**.** *A substring x of S is a minimal unique substring (MUS) if x is unique in S, and no proper substring of x is unique in S.*

Let MUS_S denote the set of intervals corresponding to MUSs of string S. Notice that by definition, MUSs of S can overlap with each other, but cannot be nested. This implies that there can exist at most one MUS starting at a given position in S. Also, since there must exist at least one MUS, we have $0 < |MUS_S| \leq |S|$.

Example 3 (MUS)**.** Let $S =$ `aabaabcababbaabdbab`, the same string as in Example 2. $MUS_S = \{[1..4], [2..5], [7..7], [8..11], [11..12], [16..16]\}$. (See Figure 5.1)

5.1.2 Data Structures

We utilize the following data structures and algorithms. While the main data structure used by Pei et al. [86] was the suffix tree [104], we use the suffix array [71], which is theoretically almost equivalent to the suffix tree, but more time and space efficient in practice.

Definition 6 (Suffix Array [71])**.** *The suffix array SA of a string S of length n is a permutation of integers $\{1, \ldots, n\}$, such that $SA[i] = j$ represents the ith lexicographically smallest suffix $S[j..n]$ of S.*

Theorem 4 ([54])**.** *Assuming an integer alphabet, the suffix array of a string S of length n can be constructed in $O(n)$ time.*

Definition 7 (Rank array)**.** *The rank array SA^{-1} of a string S of length n, is a permutation of integers $\{1, \ldots, n\}$, such that $SA^{-1}[SA[i]] = i$.*

Given SA, SA^{-1} can be computed in $O(n)$ time by a simple loop over SA.

Definition 8 (LCP array)**.** *The longest common prefix (lcp) array LCP of a string S of length n, is an array of integers where $LCP[1] = 0$ and $LCP[i]$ for $1 < i \leq n$ holds the length of the longest common prefix between suffix $S[SA[i-1]..n]$ and $S[SA[i]..n]$, where SA is the suffix array of S.*

Theorem 5 ([58])**.** *Given string S of length n and its suffix array SA, the lcp array LCP of S can be computed in $O(n)$ time.*

5.2 Algorithm

5.2.1 Finding all MUSs

Here, we describe how to find all MUSs of a string S in linear time, using the suffix and lcp arrays of S.

Lemma 18. *All MUSs of a string S of length n can be found in $O(n)$ time and space.*

Proof. Let SA and LCP respectively be the suffix array and lcp array of S. For any suffix $S[j..n]$ where $SA[i] = j$ (or $SA^{-1}[j] = i$), the shortest prefix of $S[j..n]$ that is unique is given by $S[j..j + \ell_j]$ where

$$\ell_j = \begin{cases} \max\{LCP[i], LCP[i+1]\} & 1 \leq i < n \\ LCP[i] & i = n. \end{cases}$$

The definition of ℓ_j implies that $S[j..j + \ell_j - 1]$ is not unique. Thus, $S[j..j + \ell_j]$ is the only candidate for a MUS starting at position j, and is a MUS if and only if $S[j + 1..j + \ell_j]$ is not

unique. Since the definition of ℓ_{j+1} implies that $S[j+1..j+\ell_{j+1}]$ is not unique, $S[j..j+\ell_j]$ is a MUS if and only if $j+\ell_j \leq j+\ell_{j+1}$. Once SA, SA^{-1}, and LCP are computed in $O(n)$ time, this can be checked in $O(1)$ time for each j. Therefore, the lemma follows since ℓ_j for all $1 \leq j \leq n$ can be computed in a total of $O(n)$ time. □

5.2.2 SUSs from MUSs

Next, we consider the relation between MUSs and SUSs.

Definition 9. *For an interval $[i..j]$ and position p, let $cover([i..j], p)$ denote the smallest interval that contains $[i..j]$ and p, i.e. $cover([i..j], p) = [\min(i, p).. \max(j, p)]$. We say that $cover([i..j], p)$ is derived from interval $[i..j]$ and position p.*

We first show that any $SUS_S(p)$ is derived from an element in MUS_S. The following Lemma is essentially the same as Theorem 2 in [86], but the statement has been reworded for our purposes.

Lemma 19. *For any position p and interval $[i..j] \in SUS_S(p)$, there exists exactly one sub-interval $[i'..j'] \in MUS_S$ of $[i..j]$ such that $[i..j] = cover([i'..j'], p)$.*

Proof. Since $[i..j]$ is unique, it must contain at least one minimal unique sub-interval. Let $[i'..j']$ be any MUS contained in the SUS $[i..j]$. Since $i \leq p \leq j$, $cover([i'..j'], p)$ is unique, contains position p, and is a sub-interval of $[i..j]$. Thus, $[i..j] = cover([i'..j'], p)$ must hold, since otherwise, $cover([i'..j'], p)$ would be an interval shorter than $[i..j]$ containing position p, contradicting the assumption that $[i..j]$ is an SUS.

Next we show that there is exactly one MUS contained in a SUS. Suppose there are two distinct minimal unique sub-intervals $[i_1..j_1]$ and $[i_2..j_2]$ of $[i..j]$. From the above arguments, $[i..j] = cover([i_1..j_1], p) = cover([i_2..j_2], p)$ must hold. Since MUSs cannot be nested, both must be proper sub-intervals of $[i..j]$, and we assume without loss of generality that $i \leq i_1 < i_2$ and $j_1 < j_2 \leq j$. However, if $i \leq p < j$, then $cover([i_1..j_1], p) \neq [i..j]$ since $\max\{p, j_1\} < j$, and if $i < p \leq j$, then $cover([i_2..j_2], p) \neq [i..j]$ since $\min\{p, i_2\} > i$. Thus, there can only be one MUS that is contained in a given SUS. □

For the purpose of describing our algorithm, we define a generalization of SUSs with respect to a subset of MUSs, namely, MUSs that begin at or before a certain position. Let $MUS_S^k = \{[i..j] \in MUS_S \mid i \leq k\}$. We define $SUS_S^k(p)$ to be the subset of intervals which are shortest, of

the intervals that can be derived from intervals in MUS_S^k and position p, i.e., $[i..j] \in SUS_S^k(p)$ if $[i..j] = cover([i'..j'], p)$ for some $[i'..j'] \in MUS_S^k$, and $|[i..j]| \leq |cover([i''..j''], p)|$ for any $[i''..j''] \in MUS_k$. Let $lmSUS_S^k(p)$ denote the leftmost interval of $SUS_S^k(p)$, and $lmMUS_S^k(p)$ the interval in MUS_k that derives $lmSUS_S^k(p)$.

Note that $MUS_S = MUS_S^n$, and $SUS_S(p) = SUS_S^n(p)$. Also note that although for any $k < k'$, $MUS_S^k \subseteq MUS_S^{k'}$, it is not necessarily the case that $SUS_S^k(p) \subseteq SUS_S^{k'}(p)$.

Next, we define the concept of *meaningful* and *meaningless* MUSs, which is the main difference of our algorithm with [86].

Definition 10 (Meaningful Minimal Unique Substring)**.** *We say that an interval $[i..j] \in MUS_S^k$ is* meaningful *with respect to MUS_S^k, if, for some position p, $cover([i..j], p) \in SUS_S^k(p)$. Otherwise, we say that a minimal unique substring is meaningless with respect to MUS_S^k.*

Example 4 (Meaningful MUS)**.** Let $S =$ `aabaabcababbaabdbab`, the same string as in Example 2. Then, the set of MUSs $\{[1..4], [2..5], [7..7], [11..12], [16..16]\}$ are meaningful, since they respectively derive SUSs corresponding to positions 4, 9 and 10. However, MUS $[8..11]$ is meaningless, it does not derive any SUS. (See Figure 5.1)

Observation 1. *For any $k < k'$, if an interval $[i..j] \in MUS_S^k$ is meaningless with respect to MUS_S^k, then it is meaningless with respect to $MUS_S^{k'}$.*

Let $MMUS_S^k$ denote the set of all meaningful MUSs with respect to MUS_S^k. We first show that if we have an array $MMUS_S = MMUS_S^n$ of meaningful MUSs with respect to MUS_S, in order of their occurrence, and for each position p we hold an index $L[p]$ such that $MMUS_S[L[p]] = lmMUS_S^n(p)$, we can answer $SUS_S(p)$ in $O(|SUS_S(p)|)$ time.

To prove this, we give a more specific characterization of which MUSs can derive elements of $SUS_S(p)$. Let $MUS_S(p)$ denote the set of MUSs that contain position p, i.e.,

$$MUS_S(p) = \{S[i..j] \in MUS_S \mid i \leq p \leq j\}.$$

$MUS_S(p)$ can be empty. For any position p, let $pred_S(p) = [i..j]$ represent the rightmost MUS that occurs before position p if one exists, that is, $[i..j] \in MUS_S$, $j < p$, and there exists no $[i'..j'] \in MUS_S$ such that $j < j' < p$. Similarly, let $succ_S(p) = [i..j]$ represent the leftmost MUS that occurs after position p if one exists, that is, $[i..j] \in MUS_S$, $i > p$, and there exists no $[i'..j'] \in MUS_S$ such that $p < i' < i$. We say that the set $\{pred_S(p), succ_S(p)\} \cup MUS_S(p)$ is the MUSs in the *neighborhood* of position p.

The following lemma shows that $|cover([i..j], p)|$ for MUSs $[i..j]$ in the neighborhood of position p which are meaningful with respect to MUS_S^k and are to the right of $lmMUS_S^k(p)$ (including $lmMUS_S^k(p)$), form a monotonically increasing sequence.

Lemma 20. *Consider any position p and integer k, and let $[i..j] = lmMUS_S^k(p)$. Any two distinct intervals $[i_1..j_1], [i_2..j_2] \in \{\{pred_S(p), succ_S(p)\} \cup MUS_S(p)\} \cap MMUS_S^k$ such that $i \leq i_1 < i_2$, satisfy $|cover([i_1..j_1], p)| \leq |cover([i_2..j_2], p)|$.*

Proof. Suppose to the contrary that $|cover([i_1..j_1], p)| > |cover([i_2..j_2], p)|$. Since $cover([i..j], p) \in SUS_S^k(p)$, it holds that $|cover([i..j], p)| \leq |cover([i_2..j_2], p)| < |cover([i_1..j_1], p)|$. For all positions $i \leq p' < p$, it holds that $|cover([i..j], p')| \leq |cover([i..j], p)| < |cover([i_1..j_1], p)|$. Since $[i..j] = lmMUS_s^k(p)$ and $i < i_1$, it holds that $[i_1..j_1] \neq pred_s(p)$ and $p' < p \leq j_1$. Since $p' < p \leq j_1$, it holds that $|cover([i_1..j_1], p)| \leq |cover([i_1..j_1], p')|$. Similarly, for all positions $p < p' < j_2$, it holds that $|cover([i_2..j_2], p')| = |cover([i_2..j_2], p)| < |cover([i_1..j_1], p)|$. Since $|cover([i_1..j_1], p)| > |cover([i_2..j_2], p)|$, it holds that $[i_1..j_1] \neq succ_s(p)$ and $i_1 \leq p < p'$. It holds that $|cover([i_1..j_1], p)| \leq |cover([i_1..j_1], p')|$.

Also, for any position $p' < i$, $|cover([i..j], p)| < |cover([i_1..j_1], p)|$, and for any position $p' > j_2$, $|cover([i_2..j_2], p)| < |cover([i_1..j_1], p)|$. This implies that $[i_1..j_1]$ cannot be meaningful for all positions $1 \leq p' \leq n$, and must be meaningless with respect to MUS_S^k, contradicting the assumption that $[i_1..j_1] \in MMUS_S^k$. Thus, it must be that $|cover([i_1..j_1], p)| \leq |cover([i_2..j_2], p)|$. □

The next lemma shows that intervals in $SUS_S^k(p)$ are the shortest ones derived from MUSs in the neighborhood of position p which are meaningful with respect to MUS_S^k.

Lemma 21. *Consider position p, integer k, interval $[i..j] \in MUS_S^k$, and let $Y = \{\{pred_S(p), succ_S(p)\} \cup MUS_S(p)\} \cap MMUS_S^k$. If $cover([i..j], p) \in SUS_S^k(p)$, then $[i..j] \in Y$ and $|cover([i..j], p)| \leq |cover([i'..j'], p)|$ for all intervals $[i'..j'] \in Y$.*

Proof. Assume $cover([i..j], p) \in SUS_S^k(p)$ holds. Since $Y \subseteq MUS_S^k$ and by the defintion of $SUS_S^k(p)$, $|cover([i..j], p)| \leq |cover([i'..j'], p)|$ holds for all $[i'..j'] \in Y$.

It is easy to see that $[i..j]$ cannot be to the left of $pred_S(p)$, since then, $|cover([i..j], p)| > |cover(pred_S(p), p)|$ and $[i..j]$ could not be in $SUS_S^k(p)$. Similarly, $[i..j]$ cannot be to the right of $succ_S(p)$, since then, $|cover([i..j], p)| > |cover(succ_S(p), p)|$ and again, $[i..j]$ could not be in $SUS_S^k(p)$.

Finally, by the definition of meaningful, $[i..j] \in MMUS_S^k$. □

Algorithm 1: $SUS_S(p)$ from L and $MMUS_S$.

Input: position p, $MMUS_S$, L
Output: $SUS_S(p)$

```
t ← L[p];
l ← |cover(MMUS_S[t], p)| ;                    // length of SUS
while |cover(MMUS_S[t], p)| = l do
    output cover(MMUS_S[t], p);
    t ← t + 1;
end
```

Theorem 6. *Given an array $MMUS_S$ of all meaningful MUSs with respect to MUS_S in order of occurrence, and an array L of size n, where, for each position $1 \leq p \leq n$, $MMUS_S[L[p]] = lmMUS_S^n(p)$, we can compute $SUS_S(p)$ in $O(|SUS_S(p)|)$ time.*

Proof. The pseudo code of the algorithm is shown in Algorithm 1. By definition of $MMUS_S$ and L, it is clear that the first output is $lmSUS_S^n(p)$, i.e., the leftmost SUS that contains position p. From Lemma 19 and by the definition of a meaningful interval, it is easy to see that all MUSs that derive elements in $SUS_S(p)$ must be in $MMUS_S$.

It remains to prove that each element in $SUS_S(p)$ is derived from MUSs in a contiguous range in $MMUS_S$. This can be seen from Lemmas 20 and 21, which claim that all MUSs in $SUS_S(p)$ are in the neighborhood of position p that are meaningful with respect to MUS_S, and for all such meaningful MUSs $[i..j]$ to the right of $lmMUS_S^n(p)$ (including $lmMUS_S^n(p)$), $cover([i..j], p)$ forms a monotonically increasing sequence. □

Next we show that $MMUS_S$ and L can be constructed in linear time, by incrementally updating $MMUS_S^k$ and L. Let L^k denote an array of indices where $MMUS_S^k[L^k[p]] = lmMUS_S^k(p)$.

Lemma 22. *$L^{p-1}[p]$ is either the MUS $[i..j]$ pointed to by $L^{p-1}[p-1]$, or the next MUS $[i'..j']$ in $MMUS_S^{p-1}$, i.e., the one pointed to by $L^{p-1}[p-1]+1$.*

Proof. By definition, $[i..j] = lmMUS_S^{p-1}(p-1)$. Let $[i''..j'']$ be an arbitary interval in $MMUS_S^{p-1}$ to the right of $[i..j]$. Then, $[i''..j''] \in MUS_S^{p-1}(p-1) \cap MUS_S^{p-1}(p)$, since we have that $i < i'' \leq p-1$, and if $j < j'' < p$, then $|cover([i..j], p-1)| = |[i..p-1]| > |[i''..p-1]| = |cover([i''..j''], p-1)|$ contradicting the definition of $[i..j]$. Thus, we have that $|cover([i''..j''], p-1)| = |cover([i''..j''], p)|$, and from Lemma 20, these values are monotonically increasing.

Therefore, the first one, which is $[i'..j'] = MMUS_S^{p-1}[L^{p-1}[p-1]+1]$, gives the smallest value. Note that $[i_p..j_p] = lmMUS_S^{p-1}(p)$ cannot be to the left of $[i..j]$; If $p \leq j_p$, then since $i_p < i < p \leq j_p < j$ and from the definition of $[i_p..j_p]$, we have $|cover([i_p..j_p], p-1)| = |cover([i_p..j_p], p)| \leq |cover([i..j], p)| = |cover([i..j], p-1)|$ which contradicts the definition of $[i..j]$ If $j_p \leq p-1$, then $cover([i_p..j_p], p-1)+1 = cover([i_p..j_p], p) \leq cover([i..j], p) \leq cover([i..j], p-1)+1$, again contradicting the definition of $[i..j]$. Thus, $lmMUS_S^{p-1}(p)$ must be either $[i..j]$ or $[i'..j']$. □

Theorem 7. *$MMUS_S$ and L can be constructed in linear time.*

Proof. The pseudo code of the algorithm is shown in Algorithm 2. The algorithm computes $MMUS_S$ and L for increasing positions. For each value of p, we assume that $MMUS_S^{p-1}$ and $L^{p-1}[1..p-1]$ are correctly computed, and we update them to correct values of $MMUS_S^p$ and $L^p[1..p]$.

Lines 3-8 in Algorithm 2 compute $L^{p-1}[p]$ from $L^{p-1}[p-1]$, and $MMUS_S^{p-1}$. The correctness can be seen from Lemma 22. The calculation for updating L can be done in constant time for each position.

Next, we show how to compute $MMUS_S^p$ and $L^p[1..p]$ given $MMUS_S^{p-1}$ and $L^{p-1}[1..p]$. The existence of an MUS starting at position p can be checked in constant time with Lemma 18. If there exists no such MUS, then, since $MUS_S^{p-1} = MUS_S^p$, $MMUS_S^p = MMUS_S^{p-1}$ and $L^p[p] = L^{p-1}[p]$, and no update is required. If there does exist $[p..e] \in MUS_S$ for some $e \geq p$, we check previous positions $i \leq p$ to see if $L^{p-1}[i]$ needs to be updated to $L^p[i]$. Such positions i satisfy $|cover(MMUS_S^{p-1}[L^{p-1}[i]], i)| > |cover([p..e], i)|$, and if for some position j this does not hold, then it is easy to see that it does not hold for all $j' \leq j$. Let j be the rightmost position such that the condition does not hold, i.e., $L^{p-1}[1..j]$ does not need to be updated.

If $j = p$, this means that no values in $L^{p-1}[1..p]$ need to be updated, and $L^p[p] = L^{p-1}[p]$. Concerning updating $MMUS_S^{p-1}$, we can easily see that $cover([p..e], n) \in SUS_S^p(n)$, and thus $[p..e]$ will be the last element in $MMUS_S^p$. However, MUSs in $MMUS_S^{p-1}$ may become meaningless with respect to MUS_S^p, because of the addition of $[p..e]$. These are the ones to the right of $[i'..j'] = MMUS_S^{p-1}[L^p[p]]$. They can be found and removed in line 13, whose correctness can be seen from Lemma 20.

If $j < p$, MUSs in $MMUS_S^{p-1}[L[j]+1..\ell]$ such that
$|cover(MMUS_S[k'], j)| > |cover(MMUS_S[j], j)|$, i.e., those that do not derive an interval in

$SUS_S^p(j)$ become meaningless with respect to MUS_S^p, so are removed in line 15. The correctness can also be seen from Lemma 20.

Although there may be more than a constant number of positions and MUSs that need to be updated with the addition of $[p..e]$, the cost can be amortized. Such operations correspond to lines 11, 13, 15, and 16 of Algorithm 2.

The time required for lines 11 and 16 is linear in the number of updates required for L. We show that $L[p]$ for each p is updated only a constant number of times. $L^{p-1}[p]$ is first determined at lines 3-8, with respect to MUS_S^{p-1}, pointing to $pred_S(p)$ or the leftmost shortest element in $MUS_S(p) \cap MUS_S^{p-1}$. It can be seen from Lemma 21 that for all $p' \geq p$, $L^{p'}[p]$ can only point to $pred_S(p)$, the leftmost shortest element in $MUS_S(p)$ $(= MUS_S(p) \cap MUS_S^{p'})$, or $succ_S(p)$. There are only two possibly remaining MUSs that will be added to $MUS_S(p) \cap MUS_S^{p-1}$ and update $L[p]$; an MUS in $MUS_S(p)$ beginning at position p, or $succ_S(p)$. Thus, the total time for this is linear in the number of positions.

The time required for lines 13 and 15, is linear in the number of intervals added or deleted from $MMUS$. Since each interval in $MMUS$ is added or removed at most once, the total time for this update is linear in the total number of MUSs in S, which is $O(n)$. Thus, the total time of the algorithm is $O(n)$. □

From Theorems 6 and 7, we obtain the following main theorem.

Theorem 8. *A string S of length n can be preprocessed in $O(n)$ time and space so that shortest unique substring queries can be answered in $O(k)$ time, where k is the number of shortest substrings returned. Notably, outputting a single SUS can be done in $O(1)$ time.*

5.3 Computational Experiments

We implemented our algorithm using the C++ language. All computational experiments were conducted on a MacPro (Early 2008) with two 3.2GHz Quad Core Xeon processors and 18GB Memory (DDR2 FB-DIMM 800MHz). We use libdivsufsort [19] for construction of the suffix array.

We used data taken from the Pizza & Chile corpus [20], namely, english texts, DNA sequences, XML, and protein sequences. We compared our algorithm with the implementation RSUS

[19] `http://code.google.com/p/libdivsufsort/`
[20] `http://pizzachili.dcc.uchile.cl/texts.html`

of [86] available at [21]. RSUS is actually a combination of an interface for the R language [22] and core routines written in C++. For comparison in our experiments, we modified the RSUS C++ routines to be called from a C++ program so that all programs utilize only the C++ language.

The results of experiments for the 4 data are shown in Table 5.1. We take a prefix of length n for each data, and measure the running times of RSUS [86], and TSUS (the implementation of the algorithm in this thesis). The entries marked N/A for RSUS was when the time exceeded 1 hour, at which time the execution of the program was stopped. The cause for the sudden increase in running times for RSUS was due to the fact that RSUS consumed all of the available physical memory. The results show that our algorithm is much faster (as fast as 20 times) in preprocessing time compared to RSUS.

[21] `https://bitbucket.org/wush_iis/rsus`
[22] `http://www.r-project.org`

Algorithm 2: Create array $MMUS_S$ of meaningful MUSs and an array of pointers L to $lmMUS$ for each position of string S

Input: LCP and RANK array for string S.
Output: $MMUS[1..|MMUS.size()|]$: array of meaningful MUSs; $L[1..n]$: index in $MMUS$ of leftmost SUS for each position.

```
1  for p ← 1 to n do
2      ℓ ← MMUS.size();
       // lmMUS for position p wrt MUS_S^{p-1} is either the same
          as p − 1, or the next one.
3      if p = 1 then
4          L[1] ← 1;      // Core MUS of position 1 is leftmost MUS.
5      else if L[p − 1] < ℓ and
        |cover(MMUS[L[p − 1] + 1], p)| < |cover(MMUS[L[p − 1]], p)| then
6          L[p] ← L[p − 1] + 1;
7      else
8          L[p] ← L[p − 1];
       // update MMUS and L to values wrt MUS_S^p
9      if exists MUS: newMUS = [p, e] for some e ≥ p. then   // O(1) time using
        LCP and RANK array
10         if ℓ > 0 then
               // j:  rightmost position that doesn't need update
11             j ← max{i ≤ p | |cover(MMUS[L[i]], i)| ≤ |cover(newMUS, i)|};
12             if j = p then   // No updates for L.  Remove meaningless
                MUSs from MMUS
13                 MMUS ← MMUS[1..k] where
                    k = max{k′ ≤ ℓ | |cover(MMUS[k′], p)| ≤ |cover(newMUS, p)|};
14             else          // remove meaningless MUSs after the one
                pointed by j and newMUS
15                 MMUS ← MMUS[1..k] where k = max{k′ ≤ ℓ |
                    |cover(MMUS[k′], j)| ≤ |cover(MMUS[L[j]], j)|};
16                 for j + 1 ≤ i ≤ p do  L[i] ← k + 1;   // update L to new MUS
17                 ;
18         MMUS.push_back(newMUS);
```

Table 5.1: Comparison of Computation Time in seconds.

	english ($\|\Sigma\|$ =239)		dna ($\|\Sigma\|$ =16)		dblp.xml ($\|\Sigma\|$ =97)		protein ($\|\Sigma\|$ =27)	
n (MB)	TSUS	RSUS	TSUS	RSUS	TSUS	RSUS	TSUS	RSUS
10	4.21	122.31	4.79	18.63	3.42	14.34	4.01	28.28
20	9.16	324.58	10.54	40.46	7.44	29.98	9.04	66.74
30	14.13	445.84	16.45	61.80	11.43	46.51	14.57	108.00
40	20.14	500.19	23.06	84.75	16.17	62.76	21.68	151.85
50	25.62	580.00	29.31	107.34	20.35	78.73	28.90	197.99
60	31.20	667.16	36.08	131.38	24.62	95.55	35.61	242.55
70	38.26	N/A	43.90	N/A	30.14	728.71	43.96	N/A
80	44.00	N/A	50.83	N/A	34.67	N/A	51.01	N/A
90	50.37	N/A	57.88	N/A	39.03	N/A	58.13	N/A
100	56.71	N/A	65.17	N/A	43.30	N/A	64.22	N/A

Chapter 6

Conclusion

In this thesis, we addressed (1) the substring search problem; (2) the prefix search problem; and (3) the shortest unique substring (SUS) problem and tried to develop compact data structures for them.

In Chapter 3, we introduced a new grammar-based compression scheme named *Lyndon grammar compression*, and compared its compression performance with several other grammar compression methods by computational experiments. Based on this compression scheme, we developed a new self-index structure of $O(g)$ words of space, which can be built from a string T in $O(n \lg n)$ expected time, where n is the length of T and g is the size of the Lyndon SLP for T. It takes only $O(m + \lg m \lg n + occ \lg g)$ time to find all occurrences of pattern P of length m, where occ is the number of occurrences of P in T.

In Chapter 4, we developed a compact dynamic index structure of $O(n)$ words of space for the prefix search problem, which can be built from multiple keywords with total length n. It supports all keyword dictionary operations (prefix-search, insertion and deletion) in $O(m/\alpha + \lg \alpha)$ expected time with $\alpha = w/\lg \sigma$ on input string of length m.

In Chapter 5, we showed an optimal solution to the SUS problems. We showed that S can be preprocessed in $O(n)$ time so that a SUS for any query position can be answered in $O(ans)$ time, where ans is the number of outputs.

Bibliography

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

[2] J. Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Trans. Software Eng.*, 15(9):1066–1077, 1989.

[3] D. Arroyuelo and G. Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. *Inf. Comput.*, 209(7):1070–1102, 2011.

[4] J. Arz and J. Fischer. Lempel-Ziv-78 compressed string dictionaries. *Algorithmica*, 80(7):2012–2047, 2018.

[5] N. Askitis and R. Sinha. Engineering scalable, cache and space efficient tries for strings. *VLDB J.*, 19(5):633–660, 2010.

[6] N. Askitis and J. Zobel. Cache-conscious collision resolution in string hash tables. In *Proc. SPIRE*, volume 3772 of *LNCS*, pages 91–102, 2005.

[7] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.

[8] H. Barcelo. On the action of the symmetric group on the free Lie algebra and the partition lattice. *J. Comb. Theory, Ser. A*, 55(1):93–129, 1990.

[9] F. Bassino, J. Clément, and C. Nicaud. The standard factorization of Lyndon words: an average point of view. *Discret. Math.*, 290(1):1–25, 2005.

[10] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

[11] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *Proc. ESA*, volume 6346 of *LNCS*, pages 427–438, 2010.

[12] D. Belazzougui, P. Boldi, and S. Vigna. Dynamic z-fast tries. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 159–172, 2010.

[13] P. Bille, I. L. Gørtz, P. H. Cording, B. Sach, H. W. Vildhøj, and S. Vind. Fingerprints in compressed strings. *J. Comput. Syst. Sci.*, 86:171–180, 2017.

[14] P. Bille, I. L. Gørtz, and F. R. Skjoldjensen. Deterministic indexing for packed strings. In *Proc. CPM*, volume 78 of *LIPIcs*, pages 6:1–6:11, 2017.

[15] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.

[16] P. Bille, I. Li Gørtz, P. Gawrychowski, G. M. Landau, and O. Weimann. Top Tree Compression of Tries. *arXiv 1902.02187*, 2019.

[17] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In *Proc. SIGMOD*, pages 521–534, 2018.

[18] F. Cai and M. de Rijke. A survey of query auto completion in information retrieval. *Foundations and Trends in Information Retrieval*, 10(4):273–363, 2016.

[19] 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.

[20] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, IV. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

[21] A. R. Christiansen, M. B. Ettienne, T. Kociumaka, G. Navarro, and N. Prezza. Optimal-time dictionary-compressed indexes. *arxiv:1811.12779*, 2018.

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

[23] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. SPIRE*, volume 7608 of *LNCS*, pages 180–192, 2012.

[24] F. Claude, G. Navarro, and A. Pacheco. Grammar-compressed indexes with logarithmic search time. *CoRR*, abs/2004.01032, 2020.

[25] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9):828–834, 1984.

[26] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007.

[27] J. J. Darragh, J. G. Cleary, and I. H. Witten. Bonsai: a compact representation of trees. *Softw., Pract. Exper.*, 23(3):277–291, 1993.

[28] P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane. Compression with the tudocomp framework. In *Proc. SEA*, volume 75 of *LIPIcs*, pages 13:1–13:22, 2017.

[29] J. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[30] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:1.12:1 – 1.12:31, 2008.

[31] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *Proc. PODS*, pages 181–190, 2008.

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

[33] J. Fischer and D. Köppl. Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 191–207, 2017.

[34] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. LATA*, volume 7183 of *LNCS*, pages 240–251, 2012.

[35] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. LATIN*, volume 8392 of *LNCS*, pages 731–742, 2014.

[36] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.

[37] A. Ganguly, W. Hon, R. Shah, and S. V. Thankachan. Space-time trade-offs for finding shortest unique substrings and maximal unique matches. *Theor. Comput. Sci.*, 700:75–88, 2017.

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

[39] R. Grossi and G. Ottaviano. Fast compressed tries through path decompositions. *ACM Journal of Experimental Algorithmics*, 19(1), 2014.

[40] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proc. STOC*, pages 397–406, 2000.

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

[42] S. Heinz, J. Zobel, and H. E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002.

[43] W. Hon, T. W. Lam, K. Sadakane, and W. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. ISAAC*, volume 2906 of *LNCS*, pages 240–249, 2003.

[44] W. Hon, S. V. Thankachan, and B. Xu. In-place algorithms for exact and approximate shortest unique substring problems. *Theor. Comput. Sci.*, 690:12–25, 2017.

[45] B. P. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *Proc. WWW*, pages 583–594, 2013.

[46] X. Hu, J. Pei, and Y. Tao. Shortest unique queries on strings. In *SPIRE*, volume 8799 of *Lecture Notes in Computer Science*, pages 161–172. Springer, 2014.

[47] T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theor. Comput. Sci.*, 656:215–224, 2016.

[48] A. M. Ileri, M. O. Külekci, and B. Xu. A simple yet time-optimal and linear-space algorithm for shortest unique substring queries. *Theor. Comput. Sci.*, 562:621–633, 2015.

[49] G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.

[50] J. Jansson, K. Sadakane, and W. Sung. Linked dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica*, 71(4):969–988, 2015.

[51] A. Jez. Approximation of grammar-based compression via recompression. *Theor. Comput. Sci.*, 592:115–134, 2015.

[52] S. Kanda, K. Morita, and M. Fuketa. Compressed double-array tries for string dictionaries supporting fast lookup. *Knowl. Inf. Syst.*, 51(3):1023–1042, 2017.

[53] S. Kanda, K. Morita, and M. Fuketa. Practical implementation of space-efficient dynamic keyword dictionaries. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 221–233, 2017.

[54] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. ICALP*, volume 2719 of *LNCS*, pages 943–955. Springer, 2003.

[55] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON*, volume 1090 of *LNCS*, pages 219–230, 1996.

[56] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. STOC*, pages 125–136, 1972.

[57] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[58] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. CPM*, volume 2089 of *LNCS*, pages 181–192, 2001.

[59] D. Kempa and N. Prezza. At the roots of dictionary compression: string attractors. In *Proc. STOC*, pages 827–840, 2018.

[60] J. Kieffer, E. Yang, G. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Trans. Information Theory*, 46(4):1227–1245, 2000.

[61] J. C. Kieffer and E. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Information Theory*, 46(3):737–754, 2000.

[62] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013.

[63] T. Kudo, T. Hanaoka, J. Mukai, Y. Tabata, and H. Komatsu. Efficient dictionary and language model compression for input method editors. In *Proc. of the Workshop on Advances in Text Input Methods*, pages 19–25, 2011.

[64] T. W. Lam, K. Sadakane, W. Sung, and S. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. COCOON*, volume 2387 of *LNCS*, pages 401–410. Springer, 2002.

[65] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. DCC*, pages 296–305, 1999.

[66] D. Lemire, O. Kaser, and N. Kurz. Faster remainder by direct computation: Applications to compilers and software libraries. *arXiv 1902.01961*, 2019.

[67] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983.

[68] A. Lovrencic and P. E. Black. *Binary tree representation of trees*. U.S. National Institute of Standards and Technology, 2008.

[69] R. C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77(2):202–215, 1954.

[70] V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In *Proc. CPM*, volume 3109 of *LNCS*, pages 420–433, 2004.

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

[72] S. Maruyama, H. Sakamoto, and M. Takeda. An online algorithm for lightweight grammar-based compression. *Algorithms*, 5(2):2014–235, 2012.

[73] S. Maruyama, Y. Tabei, H. Sakamoto, and K. Sadakane. Fully-online grammar compression. In *Proc. SPIRE*, volume 8214 of *LNCS*, pages 218–229, 2013.

[74] R. Mavlyutov, M. Wylot, and P. Cudré-Mauroux. A comparison of data structures to manage uris on the web of data. In *Proc. ESWC*, volume 9088 of *LNCS*, pages 137–151, 2015.

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

[76] D. P. Mehta and S. Sahni. *Handbook of Data Structures and Applications*. Chapman & Hall/CRC Computer and Information Science Series. CRC Press, 2004.

[77] L. Mercier and P. Chassaing. The height of the Lyndon tree. In *Proc. FPSAC*, pages 957–968, 2013.

[78] T. Mieno, S. Inenaga, H. Bannai, and M. Takeda. Shortest unique substring queries on run-length encoded strings. In *MFCS*, volume 58 of *LIPIcs*, pages 69:1–69:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[79] T. Mieno, D. Köppl, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Space-efficient algorithms for computing minimal/shortest unique substrings. *Theor. Comput. Sci.*, 845:230–242, 2020.

[80] G. Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

[81] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2:1–2:61, 2007.

[82] G. Navarro and N. Prezza. Universal compressed text indexing. *Theor. Comput. Sci.*, 762:41–50, 2019.

[83] C. G. Nevill-Manning, I. H. Witten, and D. L. Maulsby. Compression by induction of hierarchical grammars. In *Proc. DCC*, pages 244–253, 1994.

[84] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and lz factorization in compressed space. *Discret. Appl. Math.*, 274, 2019.

[85] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[86] J. Pei, W. C.-H. Wu, and M.-Y. Yeh. On shortest unique substring queries. In *Proc. ICDE*, pages 937–948, 2013.

[87] G. E. Pibiri and R. Venturini. Efficient data structures for massive n-gram datasets. In *Proc. SIGIR*, pages 615–624, 2017.

[88] A. Poyias and R. Raman. Improved practical compact dynamic tries. In *Proc. SPIRE*, volume 9309 of *LNCS*, pages 324–336, 2015.

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

[90] S. C. Sahinalp and U. Vishkin. Data compression using locally consistent parsing. Technical report, University of Maryland Department of Computer Science, 1995.

[91] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2–4):416–430, 2005.

[92] H. Sakamoto, T. Kida, and S. Shimozono. A space-saving linear-time algorithm for grammar-based compression. In *Proc. SPIRE*, volume 3246 of *LNCS*, pages 218–229, 2004.

[93] H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozono. A space-saving approximation algorithm for arammar-based compression. *IEICE Transactions*, 92-D(2):158–165, 2009.

[94] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 2014.

[95] J. Sheldon, W. Lee, B. Greenwald, and S. P. Amarasinghe. Strength reduction of integer division and modulo operations. In *Proc. LCPC*, volume 2624 of *LNCS*, pages 254–273, 2001.

[96] J. A. Storer. NP-completeness results concerning data compression. Technical Report 234, Dept. of Electrical Engineering and Computer Science, Princeton University, 1977.

[97] Y. Takabatake, K. Nakashima, T. Kuboyama, Y. Tabei, and H. Sakamoto. siEDM: an efficient string index and search algorithm for edit distance with moves. *Algorithms*, 9(2):26:1–26:18, 2016.

[98] Y. Takabatake, Y. Tabei, and H. Sakamoto. Improved ESP-index: A practical self-index for highly repetitive texts. In *Proc. SEA*, volume 8504 of *LNCS*, pages 338–350, 2014.

[99] T. Takagi, S. Inenaga, K. Sadakane, and H. Arimura. Packed compact tries: A fast and efficient data structure for online string processing. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100-A(9):1785–1793, 2017.

[100] K. Tsuruta, S. Inenaga, H. Bannai, and M. Takeda. Shortest unique substrings queries in optimal time. In *Proc. SOFSEM*, volume 8327 of *Lecture Notes in Computer Science*, pages 503–513, 2014.

[101] K. Tsuruta, D. Köppl, S. Kanda, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. c-trie++: A dynamic trie tailored for fast prefix searches. In *Proc. DCC*, pages 243–252, 2020.

[102] K. Tsuruta, D. Köppl, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Grammar-compressed self-index with lyndon words. *IPSJ Transactions on Mathematical Modeling and Its Applications*, 13(2):84–92, 2020.

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

[104] P. Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[105] T. A. Welch. A technique for high performance data compression. *IEEE Computer*, 17:8–19, 1984.

[106] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[107] S. Yata. Dictionary compression by nesting Prefix/Patricia tries. In *Proc. of the 17th Annual Meeting of the Association for Natural Language*, 2011.

[108] N. Yoshinaga and M. Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In *Proc. COLING*, pages 1091–1102, 2014.

[109] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. SuRF: Practical range query filtering with fast succinct tries. In *Proc. SIGMOD*, pages 323–336, 2018.

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

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