Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

A Practical Grammar-Based Index for Finding Approximate Longest Common Substrings

Bachelor's Thesis

2024
Zuzana Skubeňová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

A Practical Grammar-Based Index for Finding Approximate Longest Common Substrings

Bachelor's Thesis

Study program:	Informatics
Field of study:	Informatics
Training workplace:	Department of Computer Science
Supervisor:	Mgr. Adrián Goga
Consultant:	Dr. Travis Gagie

Bratislava, 2024
Zuzana Skubeňová

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Zuzana Skubeňová
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: A Practical Grammar-Based Index for Finding Approximate Longest Common Substrings

Annotation: The text index by Gagie et al. (2023) can efficiently find a fraction of the longest common substring of any pattern P with respect to a text T, using only space proportional to the compressed size of T. This is especially useful for real-world datasets, many of which are highly repetitive and therefore also compressible. Since the aforementioned work only describes the method in theory, the goal of this thesis is to explore its practicality and implement a preliminary version of it.

Aim: The goal of this thesis is to implement and evaluate a preliminary version of the ALCS index data structure by Gagie et al. (2023).

Literature: Gagie, T., Kashgouli, S., Navarro, G. (2023). A Simple Grammar-Based Index for Finding Approximately Longest Common Substrings. In: Nardini, F.M., Pisanti, N., Venturini, R. (eds) String Processing and Information Retrieval. SPIRE 2023. Lecture Notes in Computer Science, vol 14240. Springer, Cham.

Navarro, G. (2016). Compact data structures: A practical approach. Cambridge University Press.

Keywords: longest common substring, compressed text index, repetitive data

Supervisor:	Mgr. Adrián Goga
Consultant:	Dr. Travis Gagie
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	31.10.2023
Approved:	31.10.2023

doc. RNDr. Dana Pardubská, CSc.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Zuzana Skubeňová
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: A Practical Grammar-Based Index for Finding Approximate Longest Common Substrings
Praktický index pre hľadanie takmer najdlhších spoločných podreťazcov založený na bezkontextových gramatikách

Anotácia: Textový index od Gagieho a spol. (2023) je dátová štruktúra umožňujúca efektívne vyhľadávanie zlomku najdlhšieho spoločného podreťazca vzoru P vzhľadom na indexovaný text T, pričom používa iba priestor úmerný veľkosti komprimovaného T. Toto je dôležité pre objemné dáta z reálneho sveta, ktoré sú častokrát vysoko repetitívne a teda aj komprimovateľné. Keďže uvedené výsledky sú zatiaľ popísané iba v teoretickej rovine, cieľom tejto práce je vyhodnotiť praktickosť tejto metódy a implementovať jej zjednodušenú verziu.

Cieľ: Cieľom tejto práce je implementovať a vyhodnotiť zjednodušenú verziu indexovej dátovej štruktúry od Gagieho a spol. (2023).

Literatúra:

Kľúčové slová: najdlhší spoločný podreťazec, komprimovaný textový index, repetitívne dáta

Vedúci: Mgr. Adrián Goga
Konzultant: Dr. Travis Gagie
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 31.10.2023

Dátum schválenia: 31.10.2023

doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my thesis supervisor, Mgr. Adrián Goga, for his guidance, support, ideas, and feedback throughout the entire process of this research. I would also like to thank my thesis consultant, Dr. Travis Gagie, whose expert advice and suggestions were extremely helpful.

Special thanks go to my friends, who were always there to discuss ideas and provide moral support.

Last but not least, I want to thank my mum and my siblings for their support and encouragement.

Abstrakt

V tejto práci implementujeme zjednodušenú verziu indexovej dátovej štruktúry ALCS. Táto dátová štruktúra nájde približne najdlhší spoločný podreťazec ľubovoľného vzoru P a textu T. Následne túto dátovú štruktúru vyhodnotíme na dátach sekvencií genómov vírusu covid.

Kľúčové slová: najdlhší spoločný podreťazec, komprimovaný textový index, repetitívne dáta

Abstract

In this thesis, we implement a preliminary version of the ALCS index data structure. This data structure finds an approximately longest common substring of any pattern P and text T. Then, we evaluate this data structure on data of covid genomes sequences.

Keywords: longest common substring, compressed text index, repetitive data

Contents

Introduction

In the real world, we encounter huge amounts of data, yet much of this data contains low actual information content. Data across various fields, such as bioinformatics (e.g., DNA sequences) and astronomy (e.g., telescope networks), tend to be highly repetitive [17]. It is imperative to separate the concept of data size from its actual information content. Consequently, new data representations are required to ensure that the space of data is proportional to its information content. While some compressed data structures exist to reduce the size of large texts, there is also a need for the ability to access data directly and perform sophisticated queries without decompression. Compressed text indices offer a promising solution. The indices represent a collection of strings in a compressed format and enabling fast pattern matching without the need for decompression.

The primary aim of this thesis is to implement and evaluate a preliminary version of the ALCS index data structure proposed by Gagie et al. [9]. The ALCS index data structure efficiently identifies a fraction of the longest common substring of any given pattern P and text T, while using only space proportional to the compressed size of T.

The first chapter introduces fundamental terminology and definitions that will be used throughout this thesis. Additionally, we describe related work and provide an overview of the ALCS data structure, which serves as the theoretical basis for our implementation.

In the second chapter, we provide a detailed description of our implementation of data structure along with its algebraic background. We also discuss the issues we encountered during the implementation process.

In the third chapter, we evaluate the data structure through two experiments on real-world data, presenting and analyzing the obtained results.

Chapter 1

Compressed text indices

Recent years in string collection indexing have marked a significant milestone, allowing their representation within compressed space. This means that we can efficiently store large amounts of string data while saving memory. In computational tasks involving strings, we typically preprocess the text, allowing us to later perform operations more efficiently. Text preprocessing involves preparing and transforming raw text into a format that is more suitable for indexing. Text indexing is the process of creating a structured representation of the text data to enable fast and efficient retrieval. Additionally, this indexing technique provides indexed search functionalities, enabling rapid retrieval of specific substrings or patterns within the compressed data. This approach enables us to handle large volumes of string data with ease and speed.

Many of these rapidly expanding string collections exhibit high repetitiveness, which means the amount of information in those data is relatively low. However, the statistical compression methods used for classical collections fail to recognize this repetitiveness [17]. Hence, the new generation of data structures, which is able to handle the huge repetitive string collections, has been initiated.

Compressed text indices are purpose-built structures designed to enhance search effectiveness, enabling expedited search capabilities and rapid access to patterns or substrings within the data. Commonly utilized methods include compressed versions of suffix trees [22] and arrays [15], and the FM-Index (with the Burrows-Wheeler Transform as its building block) by Ferragina and Manzini [7].

1.1 Notion of entropy in high repetitiveness scenario

Statistical entropy [Shannon,[20]] provides an optimal and attainable measure of compressibility in statistical compression, where the goal is to exploit frequency skew. Although statistical entropy is originally defined for infinite sources, it can be adapted to individual strings. The measure derived for individual strings, known as empiri-

cal entropy [Cover and Thomas, [5]], serves as a feasible lower bound on the storage space required for a semistatic statistical compressor to effectively compress that string. However, statistical entropy fails to adequately capture other sources of compressibility, particularly repetitiveness.

In 1948, Shannon [20] proposed a measure of compressibility that exploits the varying probabilities of symbols emitted by a source. The source is "memoryless" in most basic form of statistical entropy and emits each symbol $a \in \Sigma$ with a fixed probability p_a. This entropy is then defined as:

$$\mathcal{H}(\{p_a\}) = \sum_{a \in \Sigma} p_a \log \frac{1}{p_a}$$

Entropy serves as a tool for quantifying information. The amount of information is most commonly expressed in bits or units derived from them [6]. However, empirical entropy is blind to long repetitions in text in the sense that $\mathcal{H}(S) \leq \mathcal{H}(SS)$ [14]. Consequently, any compressor achieving the empirical entropy will compress $S \cdot S$ to approximately twice the space it uses to compress S. Considering the repetitiveness, it would be more beneficial to compress S in any form and then somehow indicate that the second copy of S follows. Efficient compression and searching through vast repetitive data are crucial. Hence, there is a pressing need for more efficient data structures.

1.2 Preliminaries

In this section, we introduce the basic definitions and concepts that will be used throughout this thesis. We define fundamental terms such as strings, indexed pattern matching, grammar compression, straight-line programs, and more. These definitions will provide the necessary mathematical background for our implementation.

1.2.1 Strings

A string [17], denoted as $S = S[1 \dots n]$, is a sequence of symbols from a finite set of characters Σ, referred to as the alphabet. Here, we assume $\Sigma = \{1, 2, \dots \sigma\}$. We denote the length of a string $S[1 \dots n]$ as $|S| = n$. A symbol of S at the i-th position is denoted as $S[i]$. The substring of S from position i to position j $(i < j)$ is denoted as $S[i \dots j] = S[i] \dots S[j]$. In cases where $i > j$, then $S[i \dots j] = \varepsilon$, the empty string. The string $S[1 \dots n]$ read in reverse is denoted as $S^{rev} = S[n] \dots S[1]$. A prefix of S of length j is a substring denoted by $S[1 \dots j] = S[1] \dots S[j]$, while a suffix is represented by $S[i \dots n]$, or equivalently $S[i \dots]$. Next, we denote the concatenation of strings S_1 and S_2 as $S_1 S_2 = S_1[1] \dots S_1[|S_1|]S_2[1] \dots S_2[|S_2|]$, i.e., the symbols of S_2 are appended after the symbols of S_1. We sometimes denote concatenation as aS or Sa, where $a \in \Sigma$

represents a single symbol of length 1. The lexicographic order for strings is defined same as in a dictionary. Let S_1, S_2 be the strings. If $S_1 = \varepsilon$, then $S_1 \leq S_2$. Otherwise, let $S_1 = aX$, $S_2 = bY$, where X and Y are the strings and $a, b \in \Sigma$ be the symbols. Then $S_1 \leq S_2$ if :

- $a < b$
- $a = b$ and $X \leq Y$

1.2.2 Indexed Pattern Matching

The problem of indexed pattern matching [18] involves building a data structure, denoted as an index, to increase efficiency in searching for patterns in the string $S[1 \dots n]$. Let $P[1 \dots m]$ be a string pattern, whose occurrences in S are to be found. The output is the set of all occurrences Occ, such that $Occ = \{i, S[i \dots i+m-1] = P\}$. We aim for the time complexity to be independent of n entirely and to depend only polynomially on the length of P. When considering input reading and output writing, the optimal search time is $O(m+Occ)$. Pattern matching can be performed on a collection of strings $S_1, \dots, S_d$ (\$-terminated), by concatenating the strings into a single one $S = S_1 \dots S_d$ and then applying pattern matching on S.

1.2.3 Suffix Trees

The suffix tree [17] is a digital tree that contains all suffixes of string S. In the suffix tree, every suffix of S labels a single root-to-leaf path. Accordingly, no node has two distinct children labeled by the same symbol. Moreover, every path of nodes with a single child is compressed into single edge labeled by the concatenation of labels from all the edges of the path. Every leaf in the suffix tree corresponds to a suffix, while each internal node corresponds to a substring of S that appears more than once. The suffix tree can be represented within $O(n)$ space.

The suffix array [18] of string $S[1 \dots n]$ is the array $A[1 \dots n]$, where the positions of the suffixes of S are stored in lexicographic order.

The suffix array of the string $S = alabaralalabarda\$$:

Suffix array

17	16	3	11	1	9	7	5	13	4	12	15	2	10	8	6	14
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

String

a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

1.2.4 Grammar Compression

One of the frequently used compression techniques is based on context-free grammars introduced by Kieffer and Yang [13]. For a given string $S = [1 \dots n]$, we construct a context-free grammar that generates only the string S, which we then use as a compressed representation. One of the many reasons why grammar compression is widely used is that grammars allow for direct access to the compressed string in $O(\log n)$ time [17]. We denote the size of a grammar as g, defined as the sum of the lengths of the right-hand sides of the rules. Furthermore, we denote $exp(X)$ to be the expansion of the nonterminal X (i.e., the string). It is then obvious that $|X| = |exp(X)|$.

Straight-line programs (SLPs) [10] are a specific type of context-free grammar that generate only a single finite string S. When represented in a Chomsky's normal form, it consists of rules formatted as $X \to Y\,Z$, where X is nonterminal, Y can represent either terminal or nonterminal, and Z can also represent either a terminal or a nonterminal. When given a repetitive string S of length n, it can be represented with an SLP consisting of g rules, where $g \ll n$. An example of an SLP with non-binary rules for the string $T = alabaralalabarda\$$ is also illustrated in Fig. 1.1.

It is possible for it to be more compact than alternative methods, such as the run-length encoded Burrows-Wheeler Transform [17]. However, determining the smallest context-free grammars for specific texts is an NP-complete problem [21, 2]. Despite this difficulty, there exists some heuristics that offer assistance, such as BigRePair[1] [8], that produces very small SLPs in real scenarios.

In many cases, such as random access, it is desirable for the grammar to be balanced, ensuring that the paths from the root to any leaf in the parse tree are not excessively long. This property can be defined simply as the depth of the parse tree will be $O(\log n)$. However, this does not say anything about the complexity of the subtrees, which can still be quite unbalanced. Therefore, Charikar et al. [2] strengthened this condition:

Definition 1.2.1. [2] For a constant $0 < \alpha \leq 1/2$, an SLP is said to be α-*balanced* if, for every rule $X \to Y\,Z$, it holds that

$$\frac{\alpha}{1-\alpha} \leq \frac{|Y|}{|Z|} \leq \frac{1-\alpha}{\alpha}\,.$$

The definiton 1.2.1 means, that $exp(Y)$ and $exp(Z)$ have approximately similar lengths.

The parse tree [17] of the grammar is a tree whose vertices are labeled by nonterminal symbols and terminal symbols of the grammar. Its root bears the label of the initial symbol. Each internal node is labeled with a nonterminal X: if $X \to Y_1, \dots, Y_k$,

[1] Available online at `https://gitlab.com/manzai/bigrepair`.

then node X has k children labeled from left to right as $Y_1, \ldots, Y_k$. Thus, the leaves are labeled corresponding to the terminals that form S. The size of the parse tree is $O(n)$ according to its definition.

The grammar tree [17] is obtained by pruning the parse tree, which retains only one internal node labeled X for each nonterminal (in our case, we consider retaining the leftmost occurrence). All other occurrences of X are converted to leaves by pruning their subtrees. In a grammar of size g, as the grammar tree has k children for each unique nonterminal $X \to Y_1, \ldots, Y_k$, in addition to the root, the number of nodes is $g+1$ [17]. A grammar tree imposes a partition of the string S into at most g substrings, each covered by a leaf of the grammar tree. Each leaf represents either a terminal or a pruned nonterminal. Consequently, we can define a left-to-right parse with at most g phrases. The size of the grammar tree is $O(g)$ according to its definition.

Fig. 1.1 shows an example of a context-free grammar with its parse tree and a grammar tree.

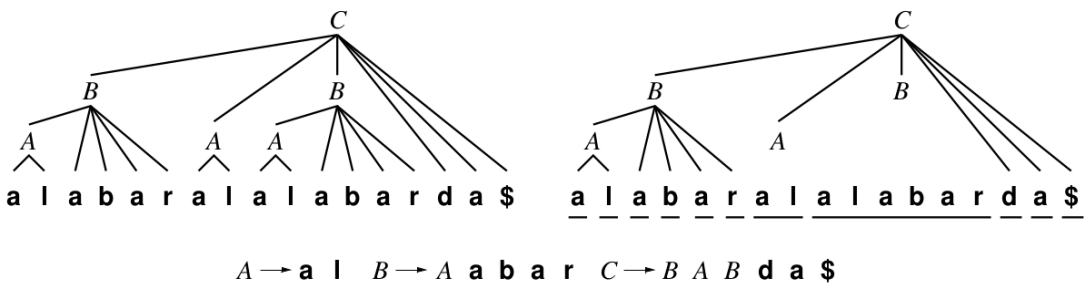


Figure 1.1: A context-free grammar that generates the string $S = alabaralalabarda\$$ of size $g = 13$ is illustrated with its parse tree (left) and a grammar tree (right), highlighting the parsing of string S with underlined phrases. The grammar rules are presented at the bottom. Image sourced from Navarro's article [18].

1.2.5 String attractors

The string attractor [17] of string S is a set $\Gamma \subseteq \{1, 2, \ldots, |S|\}$, of positions in S such that any substring $S[i..j]$ must have an occurrence $S[i'..j']$ that contains an element of Γ.

Definition 1.2.2. [12] A *string attractor* of a string T is a set of γ positions $\Gamma = j_1, \ldots, j_\gamma$ such that every substring $T[i \ldots j]$ has an occurrence $T[i' \ldots j'] = T[i \ldots j]$ with $j_k \in [i', j']$, for some $j_k \in \Gamma$.

The interval $[i, j]$ is an interval of positive integers. The size of an attractor Γ of string S is denoted as $\gamma(S)$. It is also invariant to string reversals: $\gamma(S) = \gamma(S^{rev})$.

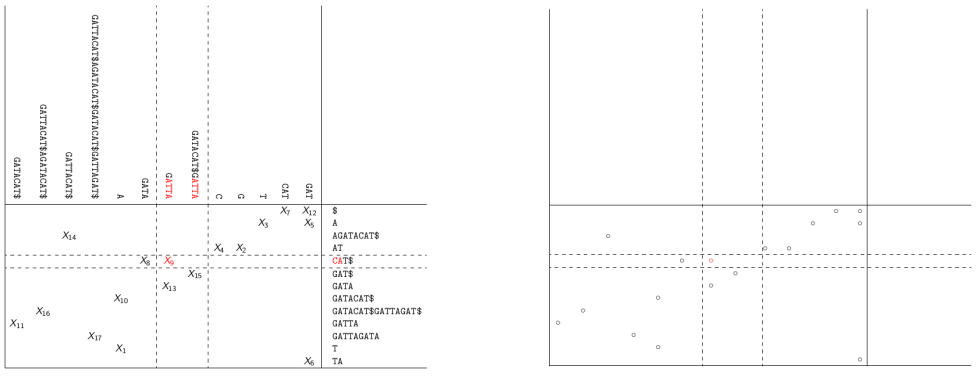
Repetitive strings have small attractors, and it holds that $\gamma \leq g^*$, where g^* is the size of the smallest SLP [12].

Example. An attractor of string $S = alabaralalabarda\$$ is $\Gamma = \{4, 6, 7, 8, 15, 17\}$. Γ is also the smallest possible attractor, with a size equal to the size σ of the alphabet Σ, and it is evident that $\gamma \geq \sigma$. For instance, the substring $S[3..9] = bara$ includes positions 6 and 7 from Γ. Similarly, $S[5..12] = aralalab$ contains positions 6, 7, 8, 15, and 17 from Γ. Additionally, $S[1..5] = alaba$ includes position 4 from Γ, and $S[10..17] = labard$ contains positions 8, 15, and 17 from Γ.

Note that it is NP-complete to find the smallest attractor size for string S [12]. However, utilizing a parse of the grammar tree enables us to obtain a sufficiently good string attractor. We can obtain such a string attractor of size $O(g)$ by considering the boundaries of parse phrases defined by the grammar tree. Further, when given string S and its string attractor Γ of size γ for S, we can construct an SLP for S with a size of $O(\gamma \log(n/\gamma))$ [4].

1.2.6 Orthogonal range queries

Orthogonal range queries are used for various purposes, primarily in computational geometry, and are performed in multi-dimensional data structures. These queries involve processing a set of objects to determine which objects intersect with a query object defined by a range. The data structure, performing orthogonal range queries, represents



(a) (b)

Figure 1.2: A grammar parsing of the text $T = GATTACAT\$AGATACAT\$GATACAT\$GATTAGAT\$GATTAGATA\$$ into 13 phrases. Let $X \to YZ$ be a rule in a grammar G. The discrete two-dimensional grid has one row and one column per element of T, $exp(Y)$ have co-lexicographic position i in T and $exp(Z)$ have lexicographic position j in Y (figure (a)). Then, we set a point at position (i, j) in the grid (figure (b)). Image sourced from Travis's slides based on [9].

a set of points and a two-dimensional grid, describing a successive reshuffling process where the points are initially sorted by one coordinate and end up sorted by another.

An example of a discrete grid is shown in Figure 1.2.

The wavelet tree [16] is a space-efficient data structure used for orthogonal range queries. Its space adapts to different entropy measures of the encoded data, allowing for compressed representation. Wavelet trees can represent a grid of points by storing the coordinates and reordering the points based on the coordinates. They are a slightly less general data structure of Chazelle [3]. Wavelet trees have been extensively used in compressed text indexing data structures to represent grids and enable efficient range counting and reporting of the points within a rectangle. Sophisticated wavelet tree variants can achieve optimal time complexity for grid range counting ($O(\log n \,/\log\log n)$) and reporting ($O((1+occ)\log^{\varepsilon} n)$) within $O((1\,/\varepsilon)n\log n)$ bits of space [16].

1.2.7 Karp–Rabin fingerprints

Assume the alphabet $\Sigma = \{1, ..., \sigma\}$. Let $c > \sigma$ be randomly chosen positive integer, S is a string with its maximum length N and $2N^{c+4} \leq p \leq 4N^{c+4}$ is a prime number. The Karp-Rabin fingerprint [1] of string S is then defined as

$$\phi(S) = \left(\sum_{i=0}^{|S|-1} S[i] \cdot c^i \right) \bmod p$$

Karp-Rabin fingerprints ensure that if two strings S_1, S_2 are equal, then their fingerprints $\phi(S_1)$ and $\phi(S_2)$ will also be equal. Moreover, if $S_1 \neq S_2$, then also $\phi(S_1) \neq \phi(S_2)$ with probability at least $1 - n^{-c}$.

When given string $S = S_1 S_2$, decomposable into a prefix S_1 and suffix S_2, then fingerprints can be composed as

$$\phi(S) = \phi(S_1) \oplus \phi(S_2) = \phi(S_1) + c^{|S_1|} \cdot \phi(S_2) \bmod p,$$

where $\oplus$ denotes addition modulo p.

Karp and Rabin [11] proposed fingerprinting to solve the classic pattern matching problem. To solve the problem the algorithm computes the fingerprint of P (string pattern) and compares it to the fingerprint of all substrings of length $|P|$ in S (string). To efficiently compute all fingerprints of length $|P|$ in S, the key observation is that the fingerprint for $S[i, i+|P|]$ can be computed from the fingerprint of $S[i-1, i-1+|P|]$ in constant time using simple properties of composition of fingerprints, as we mentioned above. To find an occurrence of P in S, one simply compares the fingerprint of P with the fingerprints of length $|P|$ in S. When the fingerprints are equal, there is a high probability that it is an occurrence of P.

With the Karp-Rabin fingerprinting method, if a given string S of length n is compressed into an SLP G of size g, it is possible to construct $O(g)$ space data structures that compute $\phi(S[i..j])$ in time $O(\log n)$ [1].

1.2.8 Maximal Exact Matches (MEM)

Exact pattern matching is often used for indexing highly repetitive data, but it may not fully reflect the needs of the real world due to various reasons, such as errors, noise, or complex patterns. Therefore, inexact pattern matching methods, such as Maximal Exact Matches (MEMs), are used to handle more complex and variable patterns. A maximal exact match is a maximal substring of the pattern that appears in the indexed text, formally defined as follows:

Definition 1.2.3. [19] A Maximal Exact Match (MEM) of a pattern $P[1..m]$ in a string T is a substring $P[i \dots j]$ that occurs in T, but in addition

- $i = 1$ or $P[i-1 \dots j]$ does not occur in T,
- $j = m$ or $P[i \dots j+1]$ does not occur in T.

This concept has various use cases, such as identifying similarities between two genomes or finding long unchanged segments within a gene. It can be used for read alignment on a reference genome, where the segment's (substring) length typically ranges in the hundreds or thousands, while the length of the genome (text) can extend into the billions.

We can find MEMs in an indexed text T in $O(m)$ time using a suffix tree [19], by creating a suffix tree concatenated string T and P, and then traversing the tree to identify the MEMs.

1.3 Related work

In this section, we provide a fundamental overview of key methodologies for managing compressed string collections.

1.3.1 Grammar based indices

As mentioned in section 1.2.4, one of the useful compression techniques involves constructing a context-free grammar. Then we can directly access the data without the requirement for decompression. Additionally, sophisticated queries can be performed on the compressed data. This is called Compressed text indexing [17] or parsing-based indexing. The key idea is that we obtain parsing by building the grammar tree, which

divides $S[1..n]$ into p phrases: $S = S_1..S_p$. Based on the parsing, we can classify the occurrences of any pattern $P[1..m]$ into two types [18]:

- the primary occurrences
- the secondary occurrences

The primary occurrences cross phrase boundaries, while the secondary occurrences are contained within a single phrase. Every substring S has an occurrence, that is primary. Consider the grammar tree: every secondary occurrence must have an occurrence somewhere to the left. The nonterminal, whose expansion contains this occurrence was pruned, as it has an earlier occurrence. Therefore, when you start from the secondary occurrence and track to the left to find the original occurrence, you locate the primary occurrence.

Parsing-based indexing, with using SLPs, is achieved by detecting primary occurrences using an $O(g)$ space data structure and obtaining secondary occurrences from the primary ones, also using $O(g)$ space.

The main idea behind how primary occurrences can be tracked is as follows: every primary occurrence of pattern P in S can be uniquely described by an interval $\langle i, j \rangle$, indicating the leftmost phrase S_i it intersects and the position j of P that aligns at the end of that phrase. We have two sets X (the reversed phrase contents) and Y (the suffixes) that are lexicographically sorted and used in bidimensional grid of size $p \times p$. The grid has exactly p points, one per row and per column. If the xth element of X in lexicographic order is X_i and the yth element of Y in lexicographic order is Y_i for some i, then there is a point labeled i at (x, y) in the grid. To obtain primary occurrences, we first find the intervals of P in X and Y, and then we retrieve the points in the two-dimensional range. For each retrieved point (x, y), labeled as i, we report the primary occurrence $\langle i, j \rangle$.

Finally, from the primary occurrences, we find the secondary occurrences, which are all occurrences of P in S.

1.3.2 ALCS

In this subsection, we will elucidate some basic information about the data structure presented in the article by Gagie, Kashgouli and Navarro [9], upon which our data structure is based.

The article proposes a method for finding approximately longest common substring of a pattern P) in a string (text T) using a simple grammar-based index. The longest common substrings are essentially the longest MEMs. They demonstrate how, given positive constants ϵ and δ, and an α-balanced straight-line program with g rules for a text $T[1..n]$, it is possible to build an index of $O(g)$ space complexity. This index,

when given a pattern $P[1..m]$, can find, with high probability, a substring of length $\ell = (1/(1-\epsilon))^k$ of P that occurs in T in $O(m \log^{\delta} g)$ time. The discovered substring's length ℓ is guaranteed to be at least a $(1-\epsilon)$ fraction of the longest common substring of P and T.

Data structure

The data structure, constructed from an α-balanced SLP G, stores sets of prefixes and suffixes of $exp(X)$ of each nonterminal $X \in G$ with exponentially increasing lengths. These sets are referred to as prefix and suffix blocks, respectively.

Definition 1.3.1. [9] Let X be a symbol in G, and let $0 < \epsilon < 1$ be a fixed constant. For each $0 \leq k \leq \log_{1/(1-\epsilon)} |X|$, we define $exp(X)[1..\lceil 1/(1-\epsilon)^k \rceil]$ as a *prefix block*, and $exp(X)[[X] - \lceil 1/(1-\epsilon)^k \rceil + 1..|X|]$ as a *suffix block*.

When given the value ϵ, we denote sets $\mathcal{X}$, $B_{\text{pref}}, B_{\text{suff}}$ as:

$$\mathcal{X} = \{exp(X), X \text{ is a symbol in } G\},$$
$$B_{\text{pref}} = \{B, B \text{ is a prefix block of a symbol } X \text{ in } G\},$$
$$B_{\text{suff}} = \{B, B \text{ is a suffix block of a symbol } X \text{ in } G\}.$$

Then, we compute the Karp-Rabin fingerprint $\phi(B)$ and the lexicographic range $[s_B, e_B]$ of the strings in $\mathcal{X}$ that are prefixed by B, for each prefix block B in the set B_{pref}. Then, we store each pair $(\phi(B), [s_B, e_B])$ in a hash table H_{pref}, using $\phi(B)$ as the key and $[s_B, e_B]$ as the associated value.

Similarly, we compute the fingerprint $\phi(B)$ and the co-lexicographic range $[s_B, e_B]$ of the strings in $\mathcal{X}$ that are suffixed by B, for each suffix block B in the set B_{suff}. Afterwards, we also store each pair $(\phi(B), [s_B, e_B])$ in a hash table H_{suff}, with $\phi(B)$ as the key and $[s_B, e_B]$ as the value.

The sizes of both prefix blocks $|B_{\text{pref}}|$ and suffix blocks $|B_{\text{suff}}|$ are $O(g)$, implying that the sizes of the hash tables are also $O(g)$ [9]. Furthermore, their combined size, $|B_{\text{pref}}| + |B_{\text{suff}}|$, is also $O(g)$ [9].

The last component of the data structure consists of a discrete two-dimensional grid of g points. Let $X \to YZ$ be a rule in G. Each point is positioned at (i, j) in the grid, where i represents the co-lexicographic position of $exp(Y)$ in T, and j represents the lexicographic position of $exp(Z)$ in T. The grid is represented in $O(g)$ space.

Finally, the entire data structure consists of H_{pref}, H_{suff}, and the two-dimensional grid, totaling $O(g)$ space.

Queries

As a result of mentioned data structure, when given pattern P of length $|P| > 1$ in T, then exists:

- an index p, such as $1 \leq p < |P|$
- a point (i, j) in two-dimensional grid, where
  - i is the co-lexicographic range of $exp(Y) \in \mathcal{X}$, suffixed by $P[1..p]$
  - j is the lexicographic range of $exp(Z) \in \mathcal{X}$, prefixed by $P[p+1..|P|]$

We denote the longest common substring of P and T as L. According to the properties of the data structure, authors proved that it suffices to find a substring of length $\ell = (1/(1-\epsilon))^k$ of L. There is such index p that divides L into L_Y and L_Z. Then, let $X \to YZ$ be a rule in G. Furthermore, we denote exponents k_Y and k_Z, which are upper boundaries of lengths L_Y and L_Z. To sum up, authors have shown, that it suffices to find suffix of length $\ell_Y = \lceil (1/(1-\epsilon))^{k_Y} \rceil$ of $exp(Y)$ and a prefix of length $\ell_Z = \lceil (1/(1-\epsilon))^{k_Z} \rceil$ of $\exp(Z)$.

To find an approximate longest common substring between T and $P[1..m]$, the authors presented the following algorithm:

Algorithm 1 The algorithm returning an approximation to the length of the longest common substring between $T[1 \ldots n]$ and $P[1 \ldots m]$. This algorithm is from ALCS article [9].

```
1: ℓ ← 0
2: for p ← 1 to m do
3:     for k_Y ← 0 to ⌊log_{1/(1−ε)} p⌋ do
4:         ℓ_Y ← ⌈(1/(1 − ε))^{k_Y}⌉
5:         [s_Y, e_Y] ← search H_suff for P[p−ℓ_Y+1..p]
6:         if [s_Y, e_Y] was found then
7:             for k_Z ← 0 to ⌊log_{1/(1−ε)}(m − p)⌋ do
8:                 ℓ_Z ← ⌈(1/(1 − ε))^{k_Z}⌉
9:                 [s_Z, e_Z] ← search H_pref for P[p + 1..p + ℓ_Z]
10:                if [s_Z, e_Z] was found then
11:                    if G has a point in [s_Y, e_Y] × [s_Z, e_Z] then
12:                        ℓ ← max(ℓ, ℓ_Y + ℓ_Z)
13: return ℓ
```

Chapter 2

Implementation

In this chapter, we describe the implementation of a preliminary version of the ALCS index data structure by Gagie et al. [2013, [9]]. This data structure is designed to find all matches of approximately half the desired length that can be extended to matches of approximately the desired length, although it may also identify some matches of about half the desired length that cannot be so extended.

As the input of our data structure, we consider an SLP program G constructed by the BigRePair software[1]. BigRePair takes a text $T = [1 \dots n]$ as input and outputs a constructed SLP G. BigRePair software is a version of the popular RePair heuristic (Larsson and Moffat),which produces small $SLPs$ and works better than methods that guarantee some approximation ratio. However, the original RePair does not scale to large data, as it requires linear memory.

The rules of G consist of terminals and nonterminals in Chomsky normal form. The root's nonterminal, placed at the end of the file, represents the entire text T.

2.1 Preliminary version of ALCS

In this section, we elucidate the mathematical background for our work. We consider two major parts: First, we build an index, and then we perform the querying.

2.1.1 Building the Index

In this part, we construct an index of hashes for all extensions of nonterminals. The index is represented as a hashtable consisting of prefix and suffix blocks. We use Karp-Rabin fingerprinting to hash terminals, nonterminals, and the blocks.

First, we compute the lengths of the prefix and suffix blocks. To achieve this, we use the following algorithm: given a positive integer n (the size of text T) and a positive

[1] Available online at `https://gitlab.com/manzai/bigrepair`.

real $\varepsilon < 1$, we greedily compute a set $\mathcal{L}$ of $O(\log_{1/(1-\epsilon)} n)$ positive integers. This set ensures that for any positive integer $h \leq n$, there exists an element $\ell \in \mathcal{L}$ such that $(1-\varepsilon)h < \ell \leq h$. To initialize $\mathcal{L}$, we start with $\{1\}$ and iterate from 2 to n. If i is at least $1/(1-\varepsilon)$ times larger than the largest element currently in $\mathcal{L}$, we insert i into $\mathcal{L}$. For example, if $n = 20$ and $\varepsilon = 1/4$, then the set $\mathcal{L}$ representing the sizes of the blocks is $\mathcal{L} = \{1, 2, 3, 4, 6, 8, 11, 15, 20\}$.

Now that we have computed the lengths of the blocks, we proceed to hash them. Given $T[1 \dots n]$ and a string attractor A for T, we build a hash table of all the distinct blocks in T, with their Karp-Rabin fingerprints as keys and their statuses as left-blocks (prefix blocks) or right-blocks (suffix blocks) or both as satellite data. A *left-block* is a substring of T whose length is in $\mathcal{L}$ and whose first character's position is in A, while a *right-block* is a substring of T whose length is in $\mathcal{L}$ and whose first character's position is immediately after an element of A.

2.1.2 Querying

In this second part, we describe the algorithm for finding all matches of approximately half the desired length of text T and pattern P. We utilize the built index from the previous part as input.

Given a pattern $P[1 \dots m]$ and a positive length $L \leq n$, we set ℓ to be $\lceil L/2 \rceil$. Then, we set k to the predecessor in $\mathcal{L}$. For example, if $\mathcal{L} = \{1, 2, 3, 4, 6, 8, 11, 15, 20\}$ and when given $L = 20$, then $\ell = 10$ and $k = 11$.

Subsequently, we scan P with a sliding window of length k and find all substrings of P of length k whose fingerprints match the fingerprints of blocks. If the fingerprint of the substring in the window matches a left-block's fingerprint, we output "left" and the position of its last character. If it matches a right-block's fingerprint, we output "right" and the position of its first character.

Finally, we consider a substring $P[i \dots i+L-1]$ of P with length L that occurs in T. By the definition of a string attractor, for some occurrence of $P[i \dots i+L-1]$ in T and some j with $i \leq j \leq i+L-1$, $P[j]$ corresponds to an element of A. Consider the (non-empty) prefix $P[i \dots j]$ and the (possibly empty) suffix $P[j+1 \dots i+L-1]$, one of which must have a length of at least $\lceil L/2 \rceil$. If $|P[i \dots j]| \geq \lceil L/2 \rceil$, then $P[j-\ell+1 \dots j]$ will match a left-block, and we output "left" and j. If $|P[j+1 \dots i+L]| \geq \lceil L/2 \rceil$, then $P[j+1 \dots j+\ell]$ will match a right-block, and we output "right" and $j+1$. On the other hand, if $P[j-\ell+1 \dots j]$ is not equal to a left-block, then with high probability, its fingerprint will not match any of theirs, and we will not output "left" and j. Similarly, if $P[j+1 \dots j+\ell]$ is not equal to a right-block, then with high probability, its fingerprint will not match any of theirs, and we will not output "right" and $j+1$.

2.2 Programming Language

For implementing our data structure, we chose `C` due to its speed and efficiency. We primarily used the data types `unsigned int`, `uint64_t`, and `unsigned __int128` to efficiently handle large numbers and ensure precise arithmetic operations.

2.3 Code

In this subsection, we describe the data structure we implemented. Our implementation consists of three parts, which are implemented in three `C` files: `grammar.c`, `index.c` and `queries.c`. In file `grammar.c`, we implemented fundamental functionalities for the grammar, such as parsing the BigRePair output, hash methods, and computing the length of nonterminals. The second file, `index.c`, provides functionality for constructing the grammar index and saving the built index to disk as a binary file. The last file `queries.c`, contains functionalities based on index pattern matching. It includes reading the constructed index from disk, reading the pattern, and executing the query.

2.3.1 `grammar.c`

In the file `grammar.c`, we included methods and data structures related to grammar processing. We computed the sizes of nonterminals using the Depth-First Search algorithm (tree version). For performing Karp-Rabin fingerprints, we chose a Mersenne prime number $p = 2^{61} - 1$. Mersenne prime numbers have efficient binary representations, which makes them advantageous for certain computational tasks. We also employed methods for exponentiation and multiplication with the modulo of a Mersenne prime number[2]. Additionally, we obtained the random constant $c = 28222$ from the website random.org, which generates true random numbers. This randomness comes from atmospheric noise[3].

We implemented the Karp-Rabin firgerprinting method to hash all nonterminals. To compute the fingerprint of a substring $T[1 \dots i]$, we start from the root nonterminal. Considering the size of the left child (left side of the rule), there are three possibilities. First, if the size of the left child is exactly i, then $\phi(T[1 \dots i])$ is the fingerprint of the left child (terminal or nonterminal). Second, if i is smaller than the size of the left child, we recursively call this method for the left child. Last, if i is larger then the size of the left child (it extends into the right child), then according to the Karp-Rabin fingerprints (1.2.7), we compose the fingerprint of the whole left child and the result of

[2]Available online at `https://github.com/dominikkempa/lz77-to-slp/blob/main/src/karp_rabin_hashing.cpp#L110`.

[3]`https://www.random.org/`

a recursive call on the right child multiplied by the constant c raised to the power of the size of the left child:

$$\phi(\text{left}) \oplus \text{c}^{|\text{left}|} \cdot \phi(\text{T}[|\text{left}| \ldots \text{i}]) \mod \text{p}.$$

We get a fingerprint of nonterminal X by computing the fingerprint of the substring $T[i \ldots j]$, where $exp(X) = T[i \ldots j]$ is the first occurrence of nonterminal X. According to the properties of Karp-Rabin fingerprinting, we can obtain the fingerprint of a substring $T[i \ldots j]$ by computing the fingerprints of substrings $T[1 \ldots i]$ and $T[1 \ldots j]$. Then, we get the fingerprint of the substring $T[i \ldots j]$ as:

$$\frac{\phi(T[1 \ldots j]) - \phi(T[1 \ldots i])}{c^i} \mod p.$$

In modular arithmetic, it is more efficient to divide by multiplying by the inverse number of c modulo p. Therefore, we obtain the fingerprint of $T[i \ldots j]$ as follows:

$$(\phi(T[1 \ldots j]) - \phi(T[1 \ldots i]) \cdot c_{inv}^i \mod p,$$

where c_{inv} is the inverse of c modulo p. The inverse number c_{inv} only needed to be computed once, at the start of the program, and used as a constant throughout. The value of the inverse is $c_{inv} = 1738410520411018574$ and was computed in the file `inverseNum.c`.

Another computation required was the start position of the first occurrence of each terminal and nonterminal expansion $exp(X)$. To achieve this, we used recursion, starting at the root nonterminal and recursively calling the function for the left and right children (left and right sides of the rule). This approach helps us find the leftmost occurrence of each nonterminal.

2.3.2 `index.c`

The file `index.c` builds an index. Initially, it processes command-line arguments to read a `.plainspl` format file and a constant *varepsilon* such that $0 < \varepsilon < 1$. The `.plainspl` file consists of grammar rules, which are represented as two integers in line separated by a space. Numbers smaller than 256 represent terminals, and numbers 256 and larger represent nonterminals. The first line of the file represents the first nonterminal, i.e. 256. The last line of the file represents the root. The constant ε is used to compute the lengths of blocks. Fig. 2.1 shows an example of `.plainslp` input rules file (left top corner) and its tree representation.

Next, we compute the lengths of the prefix and suffix blocks in a for-loop using the algorithm mentioned in 2.1.1.

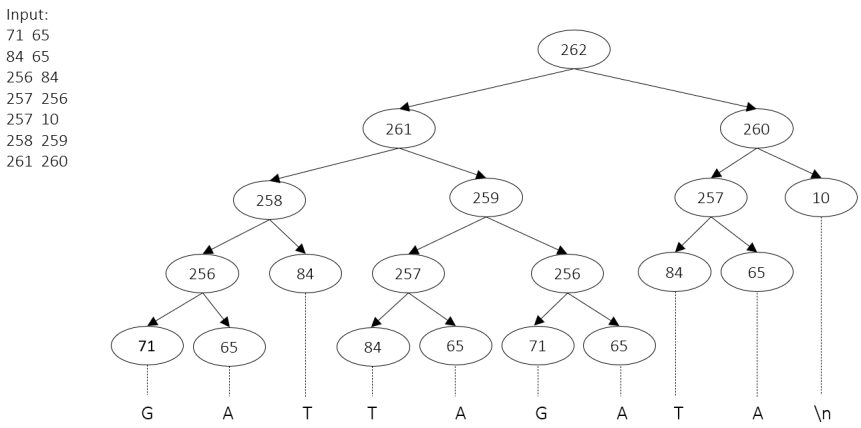


Figure 2.1: An example of input file `.plainslp` and its parse tree representation of $T = GATTAGATA\backslash n$

Further, we create prefix and suffix hash tables as described in 2.1.1. For each nonterminal X and for every block length $l \in \mathcal{L}$, if $l < |X|$, we add to the prefix hash table the fingerprint of the prefix of X of length l as the key, and the first position of $exp(X)$ in T as the value. We perform similar steps for the suffix hash table.

Finally, we save the set $\mathcal{L}$ and both hash tables to a binary file and write it to disk.

For example, for input rules from Fig. 2.1 and for $\varepsilon = 0.5$, built index have 23 blocks and occupies 304 B of disk space.

2.3.3 `queries.c`

In the file `queries.c`, we perform queries. Initially, we process command-line arguments to read the built index (hash tables), the pattern from a text file, and the constant L, used in computing the length of the sliding window.

Next, we load the block lengths and the hash tables from the binary file. Then, we use binary search to find the length k such that k is the predecessor in $\mathcal{L}$ of length $\lceil L/2 \rceil$.

We then scan the pattern P with a sliding window of length k. We hash the the sliding window and compare the computed fingerprint with the hashes of blocks in the prefix and suffix hash tables. Scanning and computing the fingerprint of a sliding window of length k can be achieved in linear time using the properties of Karp-Rabin hash with the following algorithm:

First, we compute the fingerprint $\phi(P[1 \dots k])$ and find matches with fingerprints in the hash tables. To compute the fingerprint $\phi(P[2 \dots k+1])$, we subtract $\phi(P[1 \dots k]) - \phi(P[1])$. Then, we multiply the result by the constant c_{inv} modulo p and add $\phi(P[k+1]) \cdot c^k \mod p$.

Finally, we save to the result file every match of fingerprints that we find. Specifi-

cally, we print 'l' and the value from the prefix hash table (position) if we find a match with a prefix block, and 'r' and the value from the suffix hash table if the match is with a suffix block. Additionally, we save the result file to disk.

2.4 Issues

In this subsection, we discuss some of the issues we needed to solve during the process.

2.4.1 Data structure and Performing Queries

During the implementation, we had some uncertainties about storing and transferring hash tables. Another major problem was understanding the query algorithm, which involves non-trivial algebraic background.

2.4.2 Finding the Inverse Number c_{inv}

One surprising problem (in the sense that we did not expect it to be an issue) was computing the inverse number of a constant c modulo p. In modular arithmetic, several algorithms exist to find the inverse number. However, most of them were unusable as we compute with a large prime number, $p = 2^{61} - 1$. The first problem was that some algorithms have high time complexity. The next problem we encountered while finding the inverse was data overflow. We solved the problem of finding the inverse number in the file `inverseNum.c` by using the GMP library[4].

2.4.3 Integer Overflow

Since all hashes of nonterminals are large numbers and we needed to perform multiplication and exponentiation on them, this caused integer overflow. Despite using 64-bit integers, some operations (mainly power) caused overflow issues. To solve this, we needed special algorithms optimized for this kind of computation. As a result, we used some methods implemented by Dominik Kempa[5].

[4] `https://gmplib.org/`

[5] Available online at: `https://github.com/dominikkempa/lz77-to-slp/blob/main/src/karp_rabin_hashing.cpp#L55`.

Chapter 3

Experiments

In this chapter, we present the results of the experimental evaluation of our implementation. The experiments are provided on real-world data (sequences of covid genomes). The aim of these experiments is to verify the functionality and effectiveness of our implementation of data structure. We will test building of the index and the execution of queries on large patterns (genomes).

3.1 Experiment 1

In the first experiment, we observed the impact of the parameter $0 < \varepsilon < 1$ on the number of prefix and suffix blocks.

3.1.1 Description

For building the index, we chose different number of sequences $\{1, 50, 100, 200, 500, 1000, 2000, 5000, 10000\}$ from the covid genome dataset[1], selecting only the characters A, C, G, T. The largest sequence (10k) contains 293,791,351 characters. For our experiment, we chose $\varepsilon \in \{0.1, 0.25, 0.5, 0.75, 0.9\}$.

We built the index individually for each sequence and parameter ε. Different ε values result in different lengths of prefix and suffix blocks for each nonterminal. Therefore, we recorded the number of all blocks (prefix and suffix combined) for each index built on all data sequences with different ε values.

3.1.2 Results

In our experiment, we observed that the parameter ε has a significant impact on the number of hashed blocks. For example, for the largest covid genome sequence of length

[1] Available online at `https://github.com/fmaguire/recomb_mem_test_data/tree/master/fasta`

$10k$, the index built with $\varepsilon = 0.1$ had 5.65×10^6 blocks, while the index built with $\varepsilon = 0.9$ had only 0.29×10^6 blocks.

Fig. 3.1 shows the relationship between the parameter ε, the length of covid genome sequences, and the number of blocks.

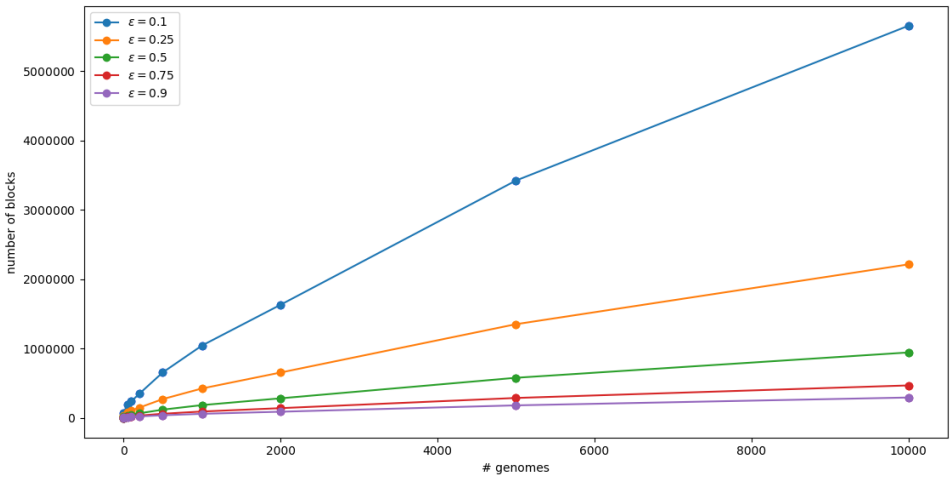


Figure 3.1: Number of blocks for different sequences of the covid genome.

3.2 Experiment 2

In our second experiment, we observed the impact of the parameter L while performing queries.

3.2.1 Description

We built an index on 10,000 sequences of the covid genome with the parameter $\varepsilon = 0.25$. The built index occupies 26.6 MB of disk space and contains a total of 2,213,078 blocks. For our pattern, we used a single covid genome extracted from a different set of 90 covid genomes[2]. We performed queries with different parameters $L \in \{2, 5, 10, 25, 50, 100, 200\}$. Our goal was to observe how many matches of the hash of the sliding window we found in the hash blocks of the built index.

3.2.2 Results

During the second experiment, we observed that only short sequences are shared between two genomes. The exact numbers of found matches are shown in Table 3.1.

[2]Available online at `https://github.com/fmaguire/recomb_mem_test_data/blob/master/query_fasta/recombinant_query_seqs.fasta.xz`

L	genom1	genom2	genom3	genom4	genom5	genom6	genom7	genom8	genom9	genom10
2	5,227,008	7,136,410	6,346,491	7,095,061	7,134,029	7,081,064	7,084,795	7,133,494	7,130,078	7,010,883
5	55	68	294	135	68	88	294	68	68	294
10	12	16	114	183	16	19	114	16	16	114
25	0	0	2	1	0	2	2	0	0	2
50	0	0	4	1	0	3	4	0	0	4
100	0	0	2	0	0	0	2	0	0	2
200	0	0	2	0	0	0	2	0	0	2

Table 3.1: Number of matches found for genomes genom1 - genom10 for different parameters L.

Conclusion

In this thesis, we first briefly introduced the topic of compressed text indices and established the necessary terminology. Furthermore, we elucidated the ALCS data structure by Gagie et al. [9] and then described the algebraic background of our data structure. Next, we implemented a preliminary version of the ALCS index data structure[3]. The implementation consisted of building an index and performing queries.

To build an index, two files were utilized: `grammar.c` and `index.c`. In `grammar.c`, methods related to grammar such as computing sizes of nonterminals and fingerprinting are implemented, while `index.c` serves for two major functionalities: reading input and creating hash tables. Building an index is fast even for large data (approximately 30s for 10k covid genomes), which also includes saving the built index to disk.

For performing queries, we implemented the file `queries.c`. File `queries.c` handles reading the built index from disk and then hashing the sliding window of pattern P and compares the hash values with hash tables. Finally, founded matches are reported to the result file.

We then proceeded to evaluate the implementation. Two experiments on real-world data (sequences of covid genomes) were provided. These experiments aim to prove that the data structure is suitable for such data. In the first experiment, when building the index, we observed the significant impact of input parameter ε on the number of prefix and suffix blocks with different lengths of genome sequences. In the second experiment, when performing queries, we searched for matches between the hash of the sliding window of the pattern and the hash blocks of the built index with different input parameters L.

The results of the experiments proved that our implementation works effectively; however, there are still possibilities for optimization and improvement. One major optimization is to check whether matches to prefix and suffix blocks with length $\ell > (1-\epsilon)\lceil L/2 \rceil$ can be extended to matches of length almost L. Another minor optimization is, for example, in computing a set $\mathcal{L}$ of lengths of blocks. This can potentially be done in $O(logn)$ (currently it is computed in $O(n)$), where n is the size of the text T.

[3]Implementation available at `https://github.com/zuzanaSKB/ALCS`.

Bibliography

[1] Philip Bille, Patrick Hagge Cording, Inge Li Gørtz, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *CoRR*, abs/1305.2777, 2013.

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

[3] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17:427–462, 1988.

[4] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17(1), dec 2021.

[5] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley, 2006.

[6] Martin Stanek Daniel Olejár. Úvod do teórie kódovania. 5 2011.

[7] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*, pages 390–398. IEEE, 2000.

[8] Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: Rescaling repair with rsync. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, pages 35–44, Cham, 2019. Springer International Publishing.

[9] Travis Gagie, Sana Kashgouli, and Gonzalo Navarro. A simple grammar-based index for finding approximately longest common substrings. 4(3):1–7, 2013.

[10] Moses Ganardi. Compression by contracting straight-line programs, 2021.

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

[12] Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC '18. ACM, June 2018.

[13] John C Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

[14] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

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

[16] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[17] Gonzalo Navarro. Indexing highly repetitive string collections, part i: repetitiveness measures. *ACM Computing Surveys (CSUR)*, 54(2):1–28, 2021.

[18] Gonzalo Navarro. Indexing highly repetitive string collections, part ii: Compressed indexes. *ACM Computing Surveys (CSUR)*, 54(2):1–31, 2021.

[19] Gonzalo Navarro. Computing mems on repetitive text collections. 10 2022.

[20] Shannon. A mathematical theory of communication. *Bell Syst. Techn. J.*, 27:398–403, 1948.

[21] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[22] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

Attachment A: Contents of the electronic attachment

The source code of the program is located in the electronic attachment attached to the paper. The source code is also published on the website: `https://github.com/zuzanaSKB/ALCS`.