A KRAKEN-LIKE TOOL WITH k GIVEN AT QUERY TIME AND AN INDEX FOR FINDING APPROXIMATELY LONGEST COMMON SUBSTRINGS

by

Sana Kashgouli

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
August 2023

© Copyright by Sana Kashgouli, 2023

To the Almighty God,

this thesis is dedicated to your divine guidance and unwavering presence in my life.

To my parents,

this thesis is dedicated to your love and support throughout my pursuit for education.

Table of Contents

List of Figures . . . . **v**

Abstract . . . . **vi**

Acknowledgements . . . . **vii**

Chapter 1 Introduction **1**

Chapter 2 Preliminaries **4**

2.1 Kraken 4

2.2 Bidirectional FM-index 5

2.3 Position-only RMQ 6

2.4 LCA 6

2.5 LZ77 6

2.6 α-balanced grammars 7

Chapter 3 KATKA: a Kraken-like tool with k given at query time **8**

3.1 Introduction 8

3.2 Design 9

3.3 Queries 11

Chapter 4 An index for finding approximately longest common substrings **15**

4.1 Introduction 15

4.2 Data structure 16

4.3 Queries 17

4.4 Faster queries 20

Chapter 5 Progress towards adapting KATKA for MEMs **23**

Chapter 6 Discussion **26**

Bibliography **28**

List of Figures

3.1 A small phylogenetic tree. 9
3.2 The grid we build for the concatenation in our example. 11
3.3 The compact tries for the concatenation in our example. . . . 12

Abstract

For highly repetitive texts such as pangenomic databases, indexes based on grammars (or Lempel-Ziv parses, string attractors, etc.) are usually significantly smaller than indexes based on the Burrows-Wheeler Transform (BWT). Nevertheless, they are not widely used in practice, probably because research on grammar-based indexes has focused almost exclusively on exact pattern matching.

In the first main part of this thesis, we describe a new tool, KATKA, that stores a phylogenetic tree such that later, given a sequence pattern $P[1..m]$ and an integer k, it can quickly return the root of the smallest subtree of the tree containing all of the leaves corresponding to the genomes in which the k-mer $P[i..i+k-1]$ occurs, for $1 \leq i \leq m-k+1$. The phylogenetic tree is any tree representing the relationships between the genomes/sequence sets. Note that this tree is not necessarily built from the whole genomes/sequence sets represented in it. The approach taken in KATKA is similar to the functionality of the popular metagenomic-classification tool Kraken but with k given at query time instead of at database construction time.

In the second main part, we show how, given positive constants ϵ and δ, and an α-balanced straight-line program with g rules for a text $T[1..n]$, we can build an $O(g)$-space index that, given a pattern $P[1..m]$, in $O(m \log^{\delta} g)$ time finds with high probability a substring of P that occurs in T and whose length is at least a $(1-\epsilon)$ fraction of the longest common substring of P and T. The correctness can be ensured within the same query time in expectation.

KATKA is currently based on an LZ77-index, but we discuss how we may be able to implement it on top of a grammar-based index, taking advantage of our results in the second part.

We close with a brief discussion of future work.

Acknowledgements

First and foremost, I express my deepest gratitude to Allah the most Almighty and most Merciful, whose grace and blessings have illuminated my path and strengthened me throughout this academic journey. This thesis reached its current form due to the invaluable assistance and guidance provided by numerous individuals. I am thankful to my supervisor Travis Gagie, for his remarkable support that has given me the opportunity to expand my knowledge and gain valuable experiences. I appreciate that he involved me in various ideas, taught me new ways to approach problems, and helped me broaden my network and improve my professional skills by introducing me to new places, people and events. Thank you to my readers, Dr. Finlay Maguire and Dr. Nicola Prezza. I am thankful for working with Ben Langmead from Johns Hopkins University and Christina Boucher from the University of Florida and I appreciate their guidance and advise on our joint projects. My co-authors, Ben Langmead and Gonzalo Navarro in papers submitted to 29th and 30th SPIRE. Thank you to Meng He and Robert Beiko, for helping me learn more on data structures and bioinformatics. Professors and TAs at the faculty of Computer Science. My colleagues Yansong Li, Nate Brown, Nicola Cotumaccio, Younan Gao. Dominik Köppl and Jarno Alanko for sharing their coding skills. A sincere thank you to Ryan MacArthur for pushing me forward and believing in me. I wholeheartedly thank my parents Parvaneh Ravan and Ali Kashgouli for their unconditional love and generous support. Finally, thanks to my dear friends and family members for their heartwarming support and endearing words of encouragement. This thesis is funded by NSERC grant RGPIN-07185-2020 and subawards from NIH grant R01HG011392 and NSF IIBR grant 2029552.

Chapter 1

Introduction

The seeds of this thesis were planted during conversations between my supervisor, Travis Gagie, and Ben Langmead at Johns Hopkins University, Gonzalo Navarro at the University of Chile, and Finlay Maguire at Dalhousie University.

Ben explained to Travis how Kraken [53] performs metagenomic classification — by storing, for each k-mer in a phylogenetic tree, a pointer to the lowest common ancestor (LCA) of all the genomes containing that k-mer — and told him that Nasko et al. [39] found that as datasets grow, the LCAs tend to get higher in the tree and less informative, unless we increase k. Increasing k requires re-indexing all the genomes in the phylogenetic tree and, if the coverage varies, no single value of k may work well for all parts of the tree anyway. We can index the genomes for several values of k simultaneously, but that increases both the space and query time by a factor roughly equal to the number of values of k that we use.

Ben had his PhD student, Omar Ahmed, supervise a summer intern, Marie Cheng, while she investigated whether maximal exact matches (MEMs) give better results than k-mers [14]. Her results indicate they do, but that leaves the problem of finding the LCAs for MEMs. Ben suggested extending the r-index [24] or MONI [50] with range-minimum and range-maximum data structures over the document array [38], to find the leftmost and rightmost occurrences of MEMs. Travis remembered that indexes based on Lempel-Ziv (LZ) compression find the leftmost occurrence of patterns as part of their normal operation, so he suggested using an index based on two LZ77 [35] parses, one for the concatenation of the genomes and the other for the reverse of that concatenation.

Travis gave the problem to me and together with Ben we wrote a paper [22] — the basis for chapter 3 in this thesis — for the 29th International Symposium on String Processing and Information Retrieval (SPIRE '22) in Concepción, Chile, which I presented; the video is available at [1]. Although we were not able to find

MEMs and their LCAs quickly, we described how an alternative to Kraken that we call KATKA (Finnish for "shrimp", to contrast with Kraken) can take k at query time, rather than at construction time. We have not implemented KATKA yet, as we are still trying to have it find MEMs' LCAs efficiently.

As a joint work with Paweł Gawrychowski and Yakov Nekrich, Travis wrote the first paper about finding maximum (rather than maximal) exact matches — that is, longest common substrings (LCSs) — with LZ77-based indexes, for the 25th Canadian Conference on Computational Geometry (CCCG '13). There have been several follow-up papers [2, 11], but none of the data structures seem easy to implement. A PhD student in our faculty, Younan Gao, investigated finding MEMs with LZ77-based indexes for a project in Travis' course and published his results [25] at the IEEE Data Compression Conference (DCC '22) last year, and Gonzalo [44] very recently gave stronger results at the 34th Symposium on Combinatorial Pattern Matching (CPM '23), but those solutions do not seem easy to implement either. As a result, we started looking at practical ways to compute quickly approximate LCSs (ALCSs), meaning exact matches guaranteed to be within a small factor of maximum.

At SPIRE, Gonzalo commented to Travis that it would be more efficient to use a single grammar-based index rather than one based on two LZ77 parses. Travis realized that, if we used an α-balanced grammar, then our index for ALCSs could use space proportional to the number of rules in the grammar. This led to the paper [23] — the basis for chapter 4 — that we have submitted to SPIRE '23, to be held in Pisa. Slowing the queries by a constant factor, we can find all approximately maximal exact matches (AMEMs) that are almost maximum, but finding their left and right occurrences is problematic. Also, Travis has a summer student, Christian Simoneau, investigating whether the requirement that grammars be α-banaced is really necessary. His preliminary results suggest there are not, and we can use grammars produced by RePair [34] and BigRePair [21].

Back in 2021, before the first COVID recombinations were detected, Fin asked Travis about detecting such recombinations automatically. Travis suggested using Kraken, hoping that the LCAs from the part of a recombined genome from one strain would lie along one path descending from the root, and the LCAs from the other strain would lie along another path descending from the root. Since we were already

working on KATKA, we decided to test this idea with MEMs instead of k-mers. Fin gave us a dataset of $10,000$ COVID genomes together with some manually recombined COVID genomes.

With KATKA still in progress, Travis asked his post-doctoral researcher, Jarno Alanko, to find the MEMs and their LCAs using Ben's idea (but with a bi-directional FM-index [33] instead of an r-index [24] or MONI [50]). Unfortunately, the results were disappointing: there were no clearly visible paths. Rather than giving up, however, we decided to try to find recombination sites directly, without using LCAs. Our idea is to look for positions in an allegedly recombined genomes that do not have any AMEMs extending far to both the left and the right, since this could indicate an unfamiliar variation or a recombination site.

The rest of this thesis is organized as follows: we present some necessary technical background in Chapter 2, present KATKA in Chapter 3, present our index for finding AMEMs in Chapter 4, present our current work on using AMEMs in KATKA in Chapter 5, and discuss in Chapter 6 the idea above (of finding areas not covered by AMEMs extending far to either side) and another idea, as possible directions for future work.

In addition to the work in this thesis, I co-authored "MONI can find k-MEMs" [51] during my master's and presented it at the 34th Symposium on Combinatorial Pattern Matching (CPM 2023) in Paris in July 2023 [30].

Chapter 2

Preliminaries

2.1 Kraken

When it comes to finding species of a genomic sample, it is common to come across similarities between the sample and the previously known species, and with the help of alignment methods, we can find out the origin of that sample. Kraken 1 is a useful bioinformatics tool that helps with classifying and labeling these samples which are made of taxonomic short sequence reads. Prior to Kraken 1, the fastest and most accurate program was BLAST [4]. Some machine learning techniques were implemented to boost BLAST's accuracy by working on larger data sets, and even though they brought more precision in the classification of the short reads, they failed to perform as quickly. Abundance estimation programs were also another attempt to optimize BLAST in terms of speed by minimizing the data sets and storing only a few sample genes of each genome, which did speed up the time but performed poorly in terms of accuracy since it could classify less number of reads from each metagenome compared to machine learning classifiers. However, Kraken 1 has been able to deliver the most accurate results in the shortest time using an algorithm that is comparable to machine learning methods in terms of accuracy. Kraken 1 uses a database containing a great number of genomes increasing continuously, which is the reason behind its high accuracy. It also uses the Lowest Common Ancestor (LCA) data structure on the taxonomic tree, which finds the LCA among all genomes that include the sequence's k-mer. Based on the Taxa and its ancestors, a new smaller tree is born from the taxonomic tree called the "Classification Tree". Each node of this tree has a weight indicating the number of k-mers in the family that the sequence belongs to. Each path in this tree has a "Root to Leaf" (RTL) score which equals sum of the weights of all nodes in that path. The path with the highest RTL contains a leaf that is used for the classification of the sequence, and the path itself is called the "Classification Path". [53] With the help of minimizers, Kraken

1 indexes a sorted list of LCA and k-mer couples that will then help with mapping k-mers to LCAs. [52] Minimizers were introduced by Roberts et al. [49] in 2004, with the purpose of enhancing the binning algorithm applied in k-mers; After setting l as a fixed size for the minimizers, in which $l \leq k$, we sort the forward strand and the reversed strand l-mers of a read, and the first l-mer appearing in the lexicographic sorted list of XORed l-mers will represent the minimizer of that read. An updated version of Kraken 1 however, takes another approach to searching and matching. The new version, Kraken 2, uses a compressed hash table that takes 2/3 less memory than the standard hash table. Instead of storing k-mers and making the comparison between them and reads from the reference, which is the method used in Kraken 1, Kraken 2 stores minimizers and compares them with the reads. All the mentioned improvements lead Kraken 2 to take 85% less memory in total compared to Kraken 1.

Variable length methods have the potential to be as effective as k-mers. A recent study at Johns Hopkins University [14], shows that MEMs have the potential to outperform k-mers and can be a strong competitor to k-mers. MEMs are parts between the read and the genome that are exactly the same and cannot be extended from left or right. When dealing with genetically diverse groups, MEMs may be the better choice over k-mers due to their superior performance.

2.2 Bidirectional FM-index

FM index, first proposed by Ferragina and Manzini in 2000 [18], is data structure based on the first and the last columns of the Burrows-Wheeler Matrix (BWM) of the text [9] that allows for sequence alignment with quick retrieval taking time linear in the pattern length, as well as compressed memory. Lam et al. in 2009 suggested bidirectional FM-index [33] which performs alignment from left to right in addition to alignment from right to left, respectively known as forward searching and backward searching. This ability permits to shift from one search to the other during the pattern alignment by storing a new version of Burrows-Wheeler Transform (BWT) called 2BWT. BWT is the rightmost column of the BWM and this updated version of BWT enables insertion and deletion with the help of Hamming distance [27] and edit distance [36].

2.3 Position-only RMQ

Range Minimum Query (RMQ) is one of the data structures used to find the value of the minimal value p in sub-array A$[l..r]$ of array A$[1..n]$ where $1 \leq l \leq p \leq r \leq n$. Position-only RMQ gives the position of p in the query array. [6] For example, in array $A[5,4,3,2,4,8,1,4]$, the position-only RMQ of the sub-array $A[3..6]$ will return 4 since $A[4] = 2$ is the smallest value in that sub-array. The same approach can be taken to find the position of the maximal value in the query. The most time and space efficient approach [19] takes constant time and linear space in bits by applying a LCA query on the Cartesian tree [16] of the array.

2.4 LCA

One of the helpful data structures used in pattern matching is the Lowest Common Ancestor (LCA). It allows finding the lowest common ancestor of two nodes n_1 and n_2 in a tree. In this thesis we also consider its application to find the lowest node in a phylogenetic tree whose subtree contains all the genomes in which a pattern occurs, by finding the LCA of the leftmost and rightmost genomes in which the pattern occurs. In section 3.1 of this thesis we see an example of how LCAs work. The time complexity of finding LCAs is O(1) with linear space in bits.

2.5 LZ77

LZ77 was first mentioned by Lempel and Ziv in 1976 as a lossless compression and indexing algorithm, which can be found as the basis of many popular compression formats such as *PNG* and *ZIP*. Given text T as the concatenation of several genomes separated by a special character *\$*, LZ77 index parses T into phrases that are either a character we have never seen before or the longest prefix of what is left that occurs in what we have seen before and is followed by another character. We will go through an example in section 3.2 of this thesis.

2.6 α-balanced grammars

In chapter 4, our index uses grammar-based compression, which compresses a text $T[1..n]$ by building and storing a context-free grammar that generates only T [31]. We focus in particular on *straight-line programs (SLPs)*, where each rule is of the form $X \rightarrow Y\,Z$, where Y and Z are terminals or nonterminals (called symbols). If T is repetitive, then it can be represented with an SLP of g rules, with $g \ll n$. Grammar-based indices [15] aim to use space linear in the grammar size while offering indexed searches for patterns $P[1..m]$, that is, enumerating all the positions in T where P occurs. Following Charikar et al. [12], we write $\langle X \rangle$ and $[X]$ to denote the string symbol X expands to and the length of that expansion, respectively. Our work builds on α-balanced SLPs, defined next. There exist practical constructions of small α-balanced grammars from repetitive texts [48].

Definition 2.6.1 ([12]). For a constant $0 < \alpha \leq 1/2$, an SLP is said to be *α-balanced* if, for every rule $X \rightarrow Y\,Z$, it holds that

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

Chapter 3

KATKA: a Kraken-like tool with k given at query time

3.1 Introduction

Kraken [53] is a popular tool that addresses the basic problem of determining where a fragment of DNA occurs in the Tree of Life, which arises for every sequencing read in a metagenomic dataset. Kraken takes a phylogenetic tree T and an integer k and stores T such that later, given a pattern $P[1..m]$, it can quickly return the root of the smallest subtree of T containing all the leaves representing genomes in which the k-mer $P[i..i+k-1]$ occurs, for $1 \leq i \leq m-k+1$. For example, if T is the small phylogenetic tree shown in Figure 3.1, $k = 3$, and $P =$ `TAGACA`, then Kraken returns

- 8 for `TAG` (which occurs in `GATTAGAT` and `GATTAGATA`),
- 6 for `AGA` (which occurs in `AGATACAT`, `GATTAGAT` and `GATTAGATA`),
- NULL for `GAC` (which does not occur in T),
- 2 for `ACA` (which occurs in `GATTACAT`, `AGATACAT` and `GATACAT`).

Notice that not all the genomes in the subtree returned for $P[i..i+k]$ need to contain it: `AGA` does not occur in `GATTACAT` or `GATACAT`.

Kraken is widely used in metagenomic analyses, especially taxonomic classification, but there are some applications for which we would rather give k at query time instead of at construction time. For example, Nasko et al. [39] showed that "the [reference] database composition strongly influence[s] the performance", with larger k values generally working better as the database grows. When the representation of strains or species in the database is skewed, therefore, it may be hard to choose a single k that works well for all of them (although, unique k-mer counting modifications of Kraken 1 and Kraken 2 may help; however it is out of the scope of this thesis, more details can be found here: . [13].). In this chapter we describe a new tool,

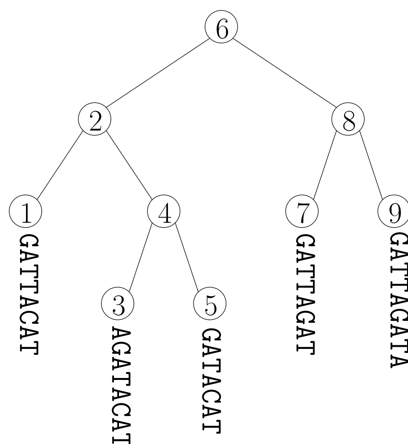

Figure 3.1: A small phylogenetic tree.

KATKA, that allows k to be chosen at query time. We are still optimizing, testing and extending KATKA and the experimental results will be reported in the future.

3.2 Design

To simplify our presentation, in this chapter we assume T is binary (although our approach generalizes to higher-degree trees). KATKA consists of three main components:

- a modified LZ77-index for the concatenation of the genomes in T, in the order they appear from left to right in T and separated by copies of a special character $;
- a modified LZ77-index for the reverse of that concatenation;
- a lowest common ancestor (LCA) data structure for T.

Given $P[1..m]$ and k, we use the first and second indexes to find the leftmost and rightmost genomes in T, respectively, that contain the k-mer $P[i..i+k-1]$, for $1 \leq i \leq m-k+1$; we then use the LCA data structure to find the lowest common ancestor of those two genomes. Since the two indexes are symmetric and the LCA data structure takes only about 2 bits per vertex in T and has constant query time, we describe only the first index.

To build the index for the concatenation, we compute its LZ77 parse and consider the phrases and co-lexicographically sort the set of their maximal non-empty suffixes not containing `$`, and consider the suffixes of the concatenation starting at phrase boundaries, and lexicographically sort the set of their maximal prefixes not containing `$` (including the empty string ε after the last phrase boundary). We discard any of those maximal prefixes that do not occur starting at a phrase boundary immediately preceded by one of those maximal suffixes.

For our example, if the concatenation is

`GATTACAT$AGATACAT$GATACAT$GATTAGAT$GATTAGATA` ,

then its LZ77 parse is

`G A T TA C AT$ AG ATA CAT$G ATACAT$GATT AGAT$ GATTAGATA` ,

the co-lexicographically sorted set of maximal suffixes is

`A`, `TA`, `ATA`, `GATTAGATA`, `C`, `G`, `AG`, `T`, `GATT` ,

and the lexicographically sorted set of maximal prefixes is

ε, `AGAT`, `AGATACAT`, `AT`, `ATACAT`, `ATTACAT`, `CAT`,
`GATTACAT`, `GATTAGATA`, `TACAT`, `TTACAT`,

but we discard `GATTACAT`, `AGATACAT` and `GATTAGATA` because they do not occur starting at a phrase boundary immediately preceded by one of the maximal suffixes.

We build a grid with the number ℓ at position (x, y) if the genome at the ℓth vertex from the left in T is the first one in which there is a phrase boundary immediately preceded by the co-lexicographically xth of the maximal suffixes and immediately followed by the lexicographically yth of the maximal prefixes. Notice this grid will be of size at most $z \times z$ with at most z numbers on it, where z is the number of phrases in the LZ77 parse of the concatenation. Figure 3.2 shows the grid for our example.

We store data structures such that given strings α and β, we can quickly find the minimum number in the box $[x_1, x_2] \times [y_1, y_2]$ on the grid, where $[x_1, x_2]$ is the co-lexicographic range of the maximal suffixes ending with α and $[y_1, y_2]$ is the lexicographic range of the maximal prefixes starting with β. (For the index for the reversed

A	TA	ATA	GATTAGATA	C	G	AG	T	GATT	
			9						ε
								7	AGAT
				1					AT
					5	3			ATACAT
					1				ATTACAT
	1	3							CAT
							1		TACAT
1									TTACAT

Figure 3.2: The grid we build for the concatenation in our example.

concatenation, we find the maximum in the query box.) In our example, if $\alpha = \texttt{G}$ and $\beta = \texttt{AT}$, then we should find 1.

For example, we can store Patricia trees for the compact tries for the reversed maximal suffixes and the maximal prefixes, together with a data structure supporting fast sequential access to the concatenation starting at any phrase boundary. In the literature (see [40] and references therein), the latter is usually an augmented straight-line program (SLP) for the concatenation; if the genomes in T are similar enough, however, then in practice it could probably be simply a VCF file. (We note that we can reuse the access data structure for the index for the reversed concatenation, augmented to support fast sequential access also at phrase boundaries in the reverse of the concatenation.) Figure 3.3 shows the compact tries for our example, with each black leaf indicating that one of the strings in the set ended at the parent of that leaf.

Nekrich [47] recently showed how to store the grid in $O(z)$ space and support 2-dimensional range-minimum queries in $O(\log^\epsilon z)$ time, for any constant $\epsilon > 0$. For simplicity, we consider his data structure in our analysis even though we are not aware of any implementation yet.

3.3 Queries

Given a pattern $P[1..m]$ and an integer k, for every substring $P[i..j]$ of P with length at most k, we find and verify the locus for the reverse of $P[i..j]$ in the compact trie

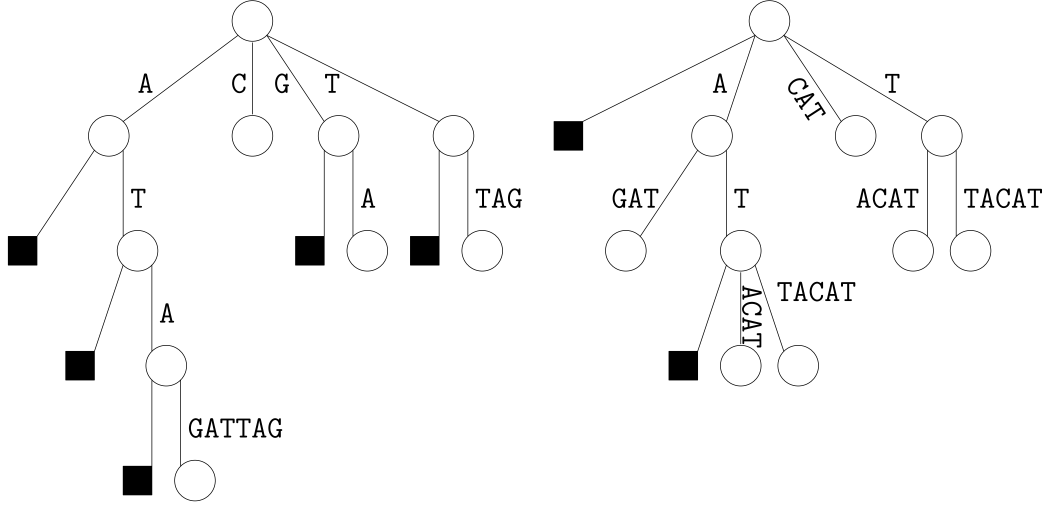

Figure 3.3: The compact tries for the concatenation in our example.

for the reversed maximal suffixes, and the locus for $P[i..j]$ in the compact trie for the maximal prefixes. (Patricia trees can return false positives when the sought pattern does not occur, so we must verify the loci by, for example, extracting their path labels from the SLP.)

By combining the searches for $P[i], P[i..i+1], \ldots, P[i..i+k-1]$, we make a total of $O(m)$ descents in the Patricia trees, each to a string-depth of at most k; we extract $O(m)$ substrings from the concatenation, each of length at most k and starting at a phrase boundary, to verify the loci. With care, this takes a total of $O(km)$ time in the worst case. When searching standard LZ77-indexes in practice, however, "queries often die in the Patricia trees" [43] — because of mismatches between characters in the pattern and the first characters in edge labels — which speeds up queries.

For each k-mer $P[i..i+k-1]$ in P and each way to split $P[i..i+k-1]$ into a non-empty prefix $P[i..j]$ and a suffix $P[j+1..i+k-1]$, we use a 2-dimensional range-minimum query to find the minimum number in the box for $\alpha = P[i..j]$ and $\beta = P[j+1..i+k-1]$ in $O(\log^{\epsilon} z)$ time.

By the definition of the LZ77 parse, the first occurrence of $P[i..i+k-1]$ in the concatenation crosses or ends at a phrase boundary. It follows that, by taking the minimum of the minima, in $O(k \log^{\epsilon} z)$ time we find the leftmost genome in T that contains $P[i..i+k-1]$. Repeating this for every value of i takes $O(km \log^{\epsilon} z)$ time.

By storing symmetric data structures for the reverse of the concatenation and

querying them, we can find the rightmost genome in T that contains $P[i..i+k-1]$, for $1 \leq i \leq m-k+1$. With the LCA data structure for T, we can find the lowest common ancestor of the two genomes, which is the root of the smallest subtree of T containing all the genomes in which the k-mer $P[i..i+k-1]$ occurs.

For our example, if $P =$ `TAGACA` and $k = 3$, then we find and verify the loci for

`T, A, AT, G, GA, GAT, A, AG, AGA, C, CA, CAG, A, AC, ACA`

in the compact trie for the reversed maximal suffixes, and the loci for

`A, AG, AGA, G, GA, GAC, A, AC, ACA, C, CA, A`

in the compact trie for the maximal prefixes.

For $P[1..3] =$ `TAG`, we look up the minimum number 7 in the box for $\alpha =$ `T` and the locus $\beta =$ `AGAT` for `AG`; since `G` has no locus in the compact trie for the maximal prefixes and `GAT` has no locus in the compact trie for the maximal reversed suffixes, we correctly conclude that the leftmost genome in T containing `TAG` is at vertex 7. A symmetric process with the index for the reversed concatenation tells us the rightmost genome in T containing `TAG` is at vertex 9, and then an LCA query tells us that vertex 8 is the root of the smallest subtree containing all the genomes in which `TAG` occurs.

Theorem 3.3.1. *Given a phylogenetic tree T whose g genomes have total length n, we can store T in $O(z \log n + g/\log n)$ space, where z is the number of phrases in the LZ77 parse of the concatenation of the genomes in T (separated by copies of a special character), such that when given a pattern $P[1..m]$ and an integer k, for $1 \leq i \leq m-k+1$ we can find the root of the smallest subtree of T containing all genomes in which the k-mer $P[i..i+k-1]$ of P occurs, in $O(km \log^{\epsilon} z)$ total time.*

Proof. The LCA data structure takes $2g + o(g)$ bits, which is $O(g/\log n)$ words (assuming $\Omega(\log n)$-bit words). An SLP for the concatenation with bookmarks permitting sequential access with constant overhead from the phrase boundaries in the parses of the concatenation and its reverse, takes $O(z \log n)$ space. For the concatenation, the Patricia trees and the instance of Nekrich's 2-dimensional range-minimum data structure take $O(z)$ space; for the reverse of the concatenation, they take space proportional to the number of phrases in its LZ77 parse, which is $O(z \log n)$. In total, we

use $O(z \log n + g/ \log n)$ space. As we have described, we make $O(m)$ descents in the Patricia trees, each to string-depth at most k, and extract only $O(m)$ substrings, each of length at most k, from the concatenation and its reverse. The time is dominated by the $O(km)$ range-minimum queries, which take $O(\log^{\epsilon} z)$ time each. □

Chapter 4

An index for finding approximately longest common substrings

4.1 Introduction

Recent years have witnessed a sustained effort for indexing highly repetitive text collections within compressed space and supporting exact pattern matching [41, 42]. Exact pattern matching is however insufficient in some applications. In Bioinformatics, for example when storing repetitive collections formed by genomes of the same species, matching strings is rarely useful. Instead, one may be interested in finding long substrings of a string that appear in the sequence collections, to find for example conserved regions of a genome in a population.

The research on matching the longest possible substrings using these indices is scarce, however. A recent result [44] finds all the maximal exact matches (MEMs) of a pattern $P[1..m]$ in a text $T[1..n]$ that is indexed with a grammar. By building on an arbitrary (run-length) context-free grammar of size g, the index is of size $O(g)$ and finds all the MEMs in time $O(m^2 \log^\delta g)$, for any constant $\delta > 0$ (see also [25]). If the grammar is of a kind called locally consistent, the time improves to $O(m \log m(\log m + \log^\delta n))$. Other results (see [44]) require larger indices.

In this chapter we consider the simpler problem of finding one longest common substring between P and T (i.e., a longest MEM). Further, we are satisfied with a common substring whose length is at least $1 - \epsilon$ times the longest one, for some fixed $0 < \epsilon < 1$. We show that, on α-balanced grammars [12, 48], this can be solved with high probability in time $O(m \log^\delta g)$ for any fixed constant $\delta > 0$. The correctness of the answer can be ensured if the time holds in expectation.

4.2 Data structure

Our data structure is built from an α-balanced SLP G. For each nonterminal X in this SLP, the structure stores a set of prefixes and suffixes of $\langle X\rangle$, of exponentially increasing lengths. Those are called prefix and suffix blocks, respectively.

Definition 4.2.1. Let X be a symbol in G and fix a constant $0 < \epsilon < 1$. Then, for each $0 \leq k \leq \log_{1/(1-\epsilon)}[X]$, we call $\langle X\rangle[1..\lceil 1/(1-\epsilon)^k\rceil]$ a *prefix block* and $\langle X\rangle[[X]-\lceil 1/(1-\epsilon)^k\rceil+1..[X]]$ a *suffix block*.

Precisely, given ϵ, consider the following sets:

$$\begin{aligned}\mathcal{X} &= \{\langle X\rangle,\, X \text{ is a symbol in } G\},\\ \mathcal{B}_{pref} &= \{B,\, B \text{ is a prefix block of a symbol } X \text{ in } G\},\\ \mathcal{B}_{suff} &= \{B,\, B \text{ is a suffix block of a symbol } X \text{ in } G\}.\end{aligned}$$

For every prefix block $B \in \mathcal{B}_{pref}$, we compute B's Karp-Rabin [29] hash $h(B)$ and the lexicographic range $[s_B, e_B]$ of the strings in $\mathcal{X}$ that are prefixed by B. We store each pair $(h(B), [s_B, e_B])$ in a perfect hash table H_{pref}, with $h(B)$ as the key and $[s_B, e_B]$ as the value. Symmetrically, for each suffix block $B \in \mathcal{B}_{suff}$, we compute B's Karp-Rabin hash $h(B)$ and the co-lexicographic range $[s_B, e_B]$ of the strings in $\mathcal{X}$ that are suffixed by B, storing each pair $(h(B), [s_B, e_B])$ in a perfect hash table H_{suff} with $h(B)$ as the key and $[s_B, e_B]$ as the value. The Karp-Rabin hash function $h(B)$ is designed to have no collision between substrings of T, which can be built in $O(n \log n)$ expected time [8]. With low probability, however, there may be collisions between substrings of a pattern P and blocks of T.

We now show that $|\mathcal{B}_{pref}|$ and $|\mathcal{B}_{suff}|$ are $O(g)$, and therefore our hash tables are of size $O(g)$ as well.

Lemma 4.2.1. *If $X \to Y\,Z$ is a rule in G, then only $O(1)$ prefix blocks $B \in \mathcal{B}_{pref}$ are prefixes of $\langle X\rangle$ but not of $\langle Y\rangle$, and only $O(1)$ suffix blocks $B \in \mathcal{B}_{suff}$ are suffixes of $\langle X\rangle$ but not of $\langle Z\rangle$.*

Proof. By Def. 2.6.1, we have

$$[X] \;=\; [Y] + [Z] \;\leq\; \left(1 + \frac{1-\alpha}{\alpha}\right) \cdot [Y] \;=\; \frac{[Y]}{\alpha}\,,$$

so the number of prefix blocks that are prefixes of $\langle X \rangle$ but not $\langle Y \rangle$ is, by Def. 4.2.1,

$$\log_{\frac{1}{1-\epsilon}} [X] - \log_{\frac{1}{1-\epsilon}} [Y] + O(1) \;=\; \log_{\frac{1}{1-\epsilon}} \frac{[X]}{[Y]} + O(1) \;\leq\; \log_{\frac{1}{1-\epsilon}} \frac{1}{\alpha} + O(1) \;=\; O(1)\,.$$

Symmetrically, because $[X] \leq [Z]/\alpha$, the number of suffix blocks that are suffixes of $\langle X \rangle$ but not of $\langle Z \rangle$ is $O(1)$. □

Corollary 4.2.1.1. *The number of prefix and suffix blocks is* $|\mathcal{B}_{pref}| + |\mathcal{B}_{suf}| = O(g)$.

Proof. By Lemma 4.2.1, each symbol X of G, of which there are g, contributes $O(1)$ prefix blocks to $\mathcal{B}_{pref}$ and $O(1)$ suffix blocks to $\mathcal{B}_{suf}$. □

The final component of our data structure is a discrete two-dimensional grid $\mathcal{G}$, with one row and one column per element of $\mathcal{X}$. Let

- $X \rightarrow Y\,Z$ be a rule in G,
- $\langle Y \rangle$ have co-lexicographic position i in $\mathcal{X}$, and
- $\langle Z \rangle$ have lexicographic position j in $\mathcal{X}$,

then we set a point at position (i, j) in the grid. We label this point with the position where $\langle Y \rangle$ ends inside an occurrence of $\langle X \rangle$ in T (i.e., if we choose the occurrence $T[a..b] = \langle X \rangle$, then the label of the point is $a + [Y] - 1$). The grid has g points, thus it can be represented in $O(g)$ space and answer range emptiness queries in $O(\log^{\delta} g)$ time, for any constant $\delta > 0$ [10].

Our whole data structure then comprises H_{pref}, H_{suff}, and $\mathcal{G}$, which add up to $O(g)$ space. We note that the values $[s_B, e_B]$ stored in H_{pref} are the lexicographic ranges of grid columns corresponding to strings in $\mathcal{X}$ prefixed with B, and those stored in H_{suff} are the co-lexicographic ranges of grid rows corresponding to strings in $\mathcal{X}$ suffixed with B.

4.3 Queries

Our searches build on a key result used in all grammar-based indices [15].

Lemma 4.3.1. *Let string S, of length $|S| > 1$, appear in T. Then, there is an index $1 \leq p < |S|$ and a point (i, j) in $\mathcal{G}$ such that*

- *i is the co-lexicographic range of a string $\langle Y \rangle \in \mathcal{X}$ suffixed by $S[1..p]$ and*

- *j is the lexicographic range of a string $\langle Z \rangle \in \mathcal{X}$ prefixed by $S[p+1..|S|]$.*

Proof. Note that S appears as a substring of the expansion of the initial symbol and, possibly, of others. If we order the rules $X \to Y\,Z$ so that Y and Z are listed before X, then the first time S appears as a substring of $\langle X \rangle$, it must appear as the concatenation of a nonempty suffix of $\langle Y \rangle$ and a nonempty prefix of $\langle Z \rangle$. The lemma then follows from the definition of $\mathcal{G}$. □

Now let L be the longest common substring of P and T and assume $|L| > 1$. Per Def. 4.2.1, let $k = \lfloor \log_{1/(1-\epsilon)} |L| \rfloor$. We note that

$$\left(\frac{1}{1-\epsilon}\right)^{k} > \left(\frac{1}{1-\epsilon}\right)^{\left(\log_{\frac{1}{1-\epsilon}} |L|\right)-1} = (1-\epsilon) \cdot |L|.$$

Thus, for our purposes, it suffices to find a substring of length $\ell = (1/(1-\epsilon))^k$ of L. By Lemma 4.3.1, there exists an index $1 \leq p < |L|$ such that $L_Y = L[1..p]$ suffixes some $\langle Y \rangle \in \mathcal{X}$, $L_Z = L[p+1..|L|]$ prefixes some $\langle Z \rangle \in \mathcal{X}$, and there is a rule $X \to Y\,Z$ in G. Further, let $k_Y = \lfloor \log_{1/(1-\epsilon)} |L_Y| \rfloor$ and $k_Z = \lfloor \log_{1/(1-\epsilon)} |L_Z| \rfloor$. By the same argument above, it follows that

$$\left(\frac{1}{1-\epsilon}\right)^{k_Y} > (1-\epsilon) \cdot |L_Y| \text{ and } \left(\frac{1}{1-\epsilon}\right)^{k_Z} > (1-\epsilon) \cdot |L_Z|.$$

Therefore, it suffices to find a suffix of length $\ell_Y = \lceil (1/(1-\epsilon))^{k_Y} \rceil$ of $\langle Y \rangle$ and a prefix of length $\ell_Z = \lceil (1/(1-\epsilon))^{k_Z} \rceil$ of $\langle Z \rangle$ to form a substring of L of length $\ell_Y + \ell_Z > (1-\epsilon) \cdot (|L_Y| + |L_Z|) = (1-\epsilon) \cdot |L|$, because $L = L_Y \cdot L_Z$.

Per Def. 4.2.1, those suffixes $L'_Y = L_Y[|L_Y| - \ell_Y + 1..\ell_Y]$ are suffix blocks, and those prefixes $L'_Z = L_Z[1..\ell_Z]$ are prefix blocks, and therefore they are stored in our hash tables. Thus, if we search H_{suff} for L'_Y and retrieve the associated range $[s_Y, e_Y]$, and search H_{pref} for L'_Z and retrieve the associated range $[s_Z, e_Z]$, we will find a point in the (row,column) range $[s_Y, e_Y] \times [s_Z, e_Y]$ of $\mathcal{G}$.

The correctness of Algorithm 1 stems from this discussion. A position of the common substring found is obtained by noticing that, when we assign ℓ in line 12, the string occurs at $P[p - \ell_Y + 1..p + \ell_Z]$ and $T[t - \ell_Y + 1..t + \ell_Z]$, where t is the label of any point in the grid range.

Algorithm 1 The simple algorithm returning an approximation to the length of the longest common substring between T and $P[1..m]$.

```
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:                     end if
14:                 end if
15:             end for
16:         end if
17:     end for
18: end for
19: return ℓ
```

Since we do not know $|L|$ beforehand, the algorithm tries all the possible values for k_Y and k_Z, which yields a time complexity dominated by $O(m \log^2 m)$ range emptiness queries, that is, $O(m \log^2 m \log^\delta n)$ [10]. We note that, since the hashes are of Karp-Rabin type, we can precompute in $O(m)$ time the hash of every prefix, $h(P[1..p])$, and then we can compute in constant time the hash of every substring of P by operating with the modular inverses of the hashes [45]. If there is a collision we may find a false positive.

Note that Algorithm 1 will find only the empty string if $|L| = 1$, as we assumed $|L| > 1$. In case the algorithm returns zero, we must determine if $|L| = 1$ by checking if some symbol of P appears as a terminal in G; this is easily done with additional $O(m)$ time and $O(g)$ space.

4.4 Faster queries

We can reduce the time complexity of Algorithm 1 by decreasing the number of combinations (k_Y, k_Z) we explore. The algorithm may try out $\Theta(\log^2 m)$ combinations per value of p, but several of those are redundant. For example, if the range $[s_Y, e_Y] \times [s_Z, e_Z]$ corresponding to the pair (k_Y, k_Z) is empty, then so is the range $[s'_Y, e'_Y] \times [s_Z, e_Z]$ corresponding to $(k_Y + 1, k_Z)$, as well as the range $[s_Y, e_Y] \times [s'_Z, e'_Z]$ corresponding to $(k_Y, k_Z + 1)$. It then suffices to explore *maximal* combinations (k_Y, k_Z). Further redundant work is done among values of p: we may be working on maximal combinations (k_Y, k_Z) that nevertheless yield shorter strings than one we had already obtained with a previous value of p.

To avoid redundant work, we will visit only the combinations (k_Y, k_Z) for which $\ell_Y + \ell_Z > \ell$; recall that ℓ is the maximum length $\ell_Y + \ell_Z$ obtained so far. Therefore, every time we find a nonempty range in $\mathcal{G}$, the value of ℓ increases. We say those combinations are *useful*. The other combinations, where either the searches in H_{pref} or in H_{suff} fail, or they succeed but the resulting range in $\mathcal{G}$ is empty, are *useless*. We will count useful and useless combinations separately.

Since there are only $O(\log^2 m)$ combinations (k_Y, k_Z), there exist $O(\log^2 m)$ different values $\ell_Y + \ell_Z$. Since the value of ℓ never decreases along the process, there are only $O(\log^2 m)$ situations in which a new value of $\ell_Y + \ell_Z$ can increase ℓ. This implies that the total number of useful combinations we visit is $O(\log^2 m)$.

To keep the number of useless combinations low, we will visit the space (k_Y, k_Z) in some suitable order. We first consider all the combinations where $k_Y \geq k_Z$, and then where $k_Z > k_Y$. We analyze the former case; the other is symmetric. We visit the values of k_Y in increasing order, and the values of k_Z in increasing order for each value of k_Y. Each new visited value k_Y is first combined with the smallest k_Z for which $\ell_Y + \ell_Z > \ell$. If this leads to a nonempty range in $\mathcal{G}$, then this is a useful combination, for which we have already accounted. The successive values of k_Z we try out from there are all useful, until we finally fail to find a nonempty range—and this then a useless combination— or until $k_Z > k_Y$. We do not consider further values $k_Z \leq k_Y$ in the first case because they will also fail to produce a nonempty range in $\mathcal{G}$.

Thus, each value of k_Y we visit leads to zero or more useful combinations possibly followed by a single useless one. We say that k_Y *succeeds* if it produces at least one

useful combination; otherwise it *fails*. If k_Y succeeds, then the cost of its last useless combination, if any, can be charged to the useful ones it produced. Therefore we only need to count the number of values of k_Y that fail. We will now show that a sequence of consecutive values of k_Y that fail has $O(1)$ combinations (all of them useless), and therefore their cost can also be charged to the preceding or following value of k_Y that succeeds. Only a sequence of all-failing values of k_Y cannot be accounted for in that way, but this can only be one sequence per value of p, adding up to $O(m)$ cost for the useless combinations.

The value of ℓ does not change across a sequence of failing values of k_Y. We never visit values $\ell_Y \leq \ell/2$: since $\ell_Z \leq \ell_Y$, they could not increase ℓ. A failing sequence of visited values k_Y then starts with some $\ell_Y > \ell/2$ and increments k_Y successively, combining it with nonincreasing values of k_Z. In this sequence, the first combination (k_Y, k_Z) we try for each k_Y, with the smallest k_Z that yields $\ell_Y + \ell_Z > \ell$, is useless, so we visit only that smallest value of k_Z per value of k_Y. We proceed increasing k_Y, always failing, until ℓ_Y exceeds ℓ, at which point the smallest value of k_Z that makes $\ell_Y + \ell_Z > \ell$ is 0. If such combination also fails, there is no point in continuing with larger values of ℓ_Y, because even combined with $k_Z = 0$ will not yield a useful combination. Since ℓ_Y is exponential in k_Y, there are only $O(1)$ values of k_Y that yield values $\ell/2 < \ell_Y \leq \ell$. Only $O(1)$ combinations are then tried along a sequence of failing values of k_Y.

Overall, we have $O(\log^2 m)$ steps charged to useful combinations and $O(m)$ to useless ones. Multiplied by the range emptiness time complexity, this yields $O(m \log^\delta g)$ total time. Note that we obtain a correct result only with high probability because we check only that $h(L_Y)$ and $h(L_Z)$ match the hash values of the corresponding block prefixes and suffixes. To ensure correctness, we can store the nonterminal $X \rightarrow Y\,Z$ associated with the point connecting $\langle Y \rangle$ and $\langle Z \rangle$ in $\mathcal{G}$, so as to verify the correctness our answer in $O(m)$ time by extracting a suffix of $\langle Y \rangle$ and a prefix of $\langle Z \rangle$ in optimal time [26]. If our answer turns out to be incorrect (which happens with low probability) we can re-run the algorithm, this time verifying every potentially useful combination, in total time $O(m^2)$. We can thus ensure correct results by making our time $O(m \log^\delta n + m + n^{-c} m^2) = O(m \log^\delta n)$ in expectation (for any constant $c > 2$).

The construction time of our structure is dominated by the construction of the

Karp-Rabin hash function with no collisions between blocks of T [45, Sec. 4].

Theorem 4.4.1. *Given positive constants ϵ and δ, and an α-balanced straight-line program with g rules for a text $T[1..n]$, we can build in $O(n \log n)$ expected time an $O(g)$-space index with which, given a pattern $P[1..m]$, in $O(m \log^{\delta} g)$ time we can find with high probability a substring of P that occurs in T and whose length is at least a $(1 - \epsilon)$ fraction of the longest common substring of P and T. The correctness can be guaranteed with time still $O(m \log^{\delta} g)$, yet in expectation.*

Chapter 5

Progress towards adapting KATKA for MEMs

In addition to optimizing and testing KATKA, we are also investigating adapting it to work with maximal exact matches (MEMs) instead of k-mers. For example, if we store $O(z)$-space z-fast tries [5] for the Patricia trees then, for each way to split P into a non-empty prefix $P[1..i]$ and a suffix $P[i+1..m]$, we can find the loci of $P[1..i]$ reversed and $P[i+1..m]$ in $O(\log m)$ time. We can verify those loci in $O(\log n)$ time by augmenting the SLP to return fingerprints, without changing its $O(z \log n)$ space bound [7].

With an $O(z)$-space data structure supporting heaviest induced ancestor queries in $O\left(\frac{\log^2 z}{\log \log z}\right)$ time [3, 20, 25], in that time we can find the longest substring $P[h..j]$ with $h \leq i \leq j$ that occurs in T with $P[h..i]$ immediately to the left of a phrase boundary and $P[h+1..j]$ immediately to its right. Note $P[h..j]$ must be a MEM. With 2-dimensional range-minimum and range-maximum queries, we can find the indexes of the leftmost and rightmost genomes in which $P[h..i]$ occurs immediately to the left of a phrase boundary and $P[h+1..j]$ immediately to its right. We still use a total of $O(z \log n + g/\log n)$ space and now we use a total of $O\left(m\left(\frac{\log^2 z}{\log \log n} + \log n\right)\right)$ time.

Unfortunately, we may not find every MEM this way: it may be that, for some MEM $P[h..j]$ and every i between h and j, either $P[h..j]$ is not split into $P[h..i]$ and $P[i+1..j]$ by any phrase boundary or some longer MEM is split into $P[h'..i]$ and $P[i+1..j']$ by a phrase boundary. For any MEM we do not find, however, we do find another MEM at least as long that overlaps it. A more serious drawback to this scheme is that it is probably quite impractical (for example, we are not aware of any implementation of a data structure supporting fast heaviest induced ancestor queries, either).

We have been trying to take advantage of our results from Chapter 4 to speed up KATKA working with something like MEMs. We say a list L of triples is an *occurrence*

list for P with respect to T if every triple (s, ℓ, q) in L encodes an occurrence of a substring of P in T,

$$P[s..s + \ell - 1] = T[q..q + \ell - 1] \, .$$

We say L is *left-bounding* for a MEM $P[i..j]$ if L contains a triple (s, ℓ, q) such that

- $P[s..s + \ell - 1]$ is a substring of $P[i..j]$,

$$i \leq s \leq s + \ell - 1 \leq j \, ;$$

- $P[s..s + \ell - 1]$ covers all but an ϵ-fraction of $P[i..j]$,

$$\ell \geq (1 - \epsilon)(j - i + 1) \, ;$$

- q is at most the starting position of the leftmost occurrence of $P[i..j]$ in T.

We define a *right-bounding* occurrence list for $P[i..j]$ symmetrically — with q now at least the starting position of the rightmost occurrence of $P[i..j]$ in T — and we just say L is *bounding* for $P[i..j]$ if it is both left- and right-bounding.

Theorem 5.0.1. *We can add $O(g)$ words to the index in Theorem 4.4.1 such that, given $\beta \geq 2$ together with P, in $O(m \log^{\delta}(g) \log \beta)$ time we can build an occurrence list of length $O(m \log \beta)$ that is bounding simultaneously for all MEMs whose lengths are within a β factor of maximum.*

Proof. We show only how we build such an occurrence list that is left-bounding for all such MEMs simultaneously, since building one that is right bounding is symmetric and their concatenation is bounding.

Recall that each point (x, y) on the grid indicates that the co-lexicographically xth prefix ending at a boundary between symbols' expansions, is immediately followed by the lexicographically yth suffix starting at such a boundary. We assign each point (x, y) weight equal to the position in T of the corresponding boundary, and replace the data structure supporting range-emptiness queries on the grid with one supporting range-minimum queries.

We first use the index in Theorem 4.4.1 to compute a $(1 - \epsilon)$-approximation ℓ^* of the length of the longest common substring of P and T, in $O(m \log^{\delta} g)$ time. We then make a second pass over P and check every way to split P into a prefix and a suffix,

and every combination of suffix- and prefix-block sizes that could result in matches of length between $(1-\epsilon)\ell/\beta$ and ℓ^*, which takes $O(m \log^\delta(g) \log \beta)$ total time.

For each pair of suffix- and prefix-blocks we check, our query on the grid now returns the position of the leftmost boundary between a symbol's expansion ending with that suffix-block and a symbol's expansion starting with that prefix-block, if there is one. For every query on the grid that returns a value, we store in the occurrence list a triple consisting of the starting position s in P of the suffix-block, the combined length ℓ of the two blocks, and the starting position q in T of the suffix-block (which is the position of the leftmost boundary minus the length of the suffix-block).

Consider the leftmost occurrence $T[p..p+j-i+1]$ in T of a MEM $P[i..j]$ whose length is within a β factor of maximum. Consider the lowest node v in the parse tree whose subtree covers $T[p..p+j-i+1]$, and the boundary between v's left and right subtrees. If we take the largest suffix-block B_{suf} to the left of that boundary that is contained in $T[p..p+j-i+1]$ and the largest prefix-block B_{pref} to the right of that boundary that is contained in $T[p..p+j-i+1]$ then their concatenation has length at least $(1-\epsilon)\ell/\beta$ — so we consider that combination of blocks — and covers all but an ϵ-fraction of $T[p..p+j-i+1] = P[i..j]$.

When we query the grid for the combination of B_{suf} and B_{pref}, it returns the position of the leftmost boundary between occurrences of B_{suf} and B_{pref} — which is at most the position of the boundary between v's left and right subtrees. It follows that we store a triple that makes the occurrence list left-bounding for $P[i..j]$. $\square$

We note that if we set β to a constant then our queries take $O(m \log^\delta g)$ time, as in Theorem 4.4.1, and if we set it to m then they take $O(m \log^\delta(g) \log m)$ time but we build an occurrence list that is bounding for all MEMs simultaneously.

Chapter 6

Discussion

As we noted in Chapter 2, we are now considering looking for recombination sites directly, without using LCA, by looking for positions that do not have any AMEMs extending far in both directions. This way, we do not need to worry about finding the leftmost and rightmost occurrences of the AMEMs. If we find a section of a genome that is not contained in any AMEM then, by extending the length by a factor of $1/(1-\epsilon)$ in either direction, we obtain a substring that is guaranteed not to occur in the genomes in the phylogenetic tree. We leave testing this idea as future work.

Very recently, Ben Langmead mentioned another tool being developed at Johns Hopkins, called Panagram [28], by Michael Schatz's lab. Basically, Panagram stores the k-mers in a collection of genomes such that, given a section of one of those genomes, it can display in how many genomes of the collection each k-mer in that section occurs. Ben and PhD student Stephen Hwang at Johns Hopkins have been looking at making Panagram use MEMs instead of k-mers, but at the moment the construction is not always practical.

Travis has suggested using an RLZ-index [17, 46] for document listing, as follows:

- we build and store an artificial reference for the collection of genomes [37] and build an RLZ-index for the collection, computing the parse [32] relative to that artificial reference;

- in the grid structure for locating primary occurrences, we colour the point for each phrase boundary to indicate in which genome the boundary is located;

- in the grid structure for locating secondary occurrences (which are copied directly from occurrences in the artificial reference), we colour the point for each phrase's source to indicate in which genome the phrase is located (disallowing phrases spanning more than one genome);

- given a pattern, we used coloured range-reporting queries on the grids to list the distinct genomes containing the pattern, in time bounded in terms of the number of genomes we find (since the grids return each distinct genome at most once for a primary occurrence and once for a secondary occurrence).

Assuming there are not too many genomes in the collection, document listing is a reasonable substitute for document counting. Combining this idea with the ideas in Chapters 4 and 5, we should be able to find the AMEMs in the given query section and estimate in how many distinct genomes they each appear, in nearly linear time.

Bibliography

[1] Spire 2022 - jornada inicial — youtube.com. `https://www.youtube.com/watch?v=gymo8L8TqNM&t=7297s`, 2022. [Accessed 12-Jul-2023].

[2] Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem: Better data structures and applications. *Algorithmica*, 84(7):2088–2105, 2022.

[3] Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem: Better data structures and applications. *Algorithmica*, 84(7):2088–2105, jul 2022.

[4] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[5] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In Mark de Berg and Ulrich Meyer, editors, *Algorithms – ESA 2010*, pages 427–438, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

[7] Philip Bille, Inge Li Gørtz, Patrick Hagge Cording, Benjamin Sach, Hjalte Wedel Vildhøj, and Søren Vind. Fingerprints in compressed strings. *Journal of Computer and System Sciences*, 86:171–180, June 2017.

[8] Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time–space trade-offs for longest common extensions. *Journal of Discrete Algorithms*, 25:42–50, 2014. 23rd Annual Symposium on Combinatorial Pattern Matching.

[9] Michael. Burrows and David Wheeler. A block-sorting lossless data compression algorithm. *SRS Research Report*, 124, 1994.

[10] Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range searching on the ram, revisited, 2011.

[11] Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In *47th International Colloquium on Automata, Languages, and Programming (ICALP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[12] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, July 2005.

[13] Shifu Chen, Changshou He, Yingqiang Li, Zhicheng Li, and Charles E Melançon III. A computational toolset for rapid identification of sars-cov-2, other viruses, and microorganisms from sequencing data. *bioRxiv*, 2020.

[14] Marie Cheng, Omar Ahmed, Anna Liebhoff, and Ben Langmead. *Factors Affecting kmer Specificity & Alternative Approaches for Metagenomic Classification.* Unpublished internal meeting document., 2022.

[15] Francisco Claude Faust, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 06 2021.

[16] Erik D Demaine, Gad M Landau, and Oren Weimann. On cartesian trees and range minimum queries. In *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36*, pages 341–353. Springer, 2009.

[17] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative lempel–ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014.

[18] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[19] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[20] Travis Gagie, Pawel Gawrychowski, and Yakov Nekrich. Heaviest induced ancestors and longest common substrings. In *Proceedings of CCCG '13*, 2013. Canadian Conference on Computational Geometry CCCG 2013 ; Conference date: 08-08-2013 Through 10-08-2013.

[21] Travis Gagie, Tomohiro I, Giovanni Manzini, Gonzalo Navarro, Hiroshi Sakamoto, and Yoshimasa Takabatake. Rpair: rescaling repair with rsync. In *International Symposium on String Processing and Information Retrieval*, pages 35–44. Springer, 2019.

[22] Travis Gagie, Sana Kashgouli, and Ben Langmead. Katka: A kraken-like tool with k given at query time. In Diego Arroyuelo and Barbara Poblete, editors, *String Processing and Information Retrieval*, pages 191–197, Cham, 2022. Springer International Publishing.

[23] Travis Gagie, Sana Kashgouli, and Gonzalo Navarro. A simple grammar-based index for finding approximately longest common substrings. Submitted, 2023.

[24] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. SIAM, 2018.

[25] Younan Gao. Computing matching statistics on repetitive texts. In Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, editors, *Data Compression Conference, DCC 2022, Snowbird, UT, USA, March 22-25, 2022*, pages 73–82. IEEE, 2022.

[26] Leszek Gasieniec, Roman M Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In *DCC*, page 458. Citeseer, 2005.

[27] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[28] Katharine Jenike, Sam Kovaka, Matthias Benoit, Srividya Ramakrishnan, Shujun Ou, James Saterlee, Stephan Hwang, Iacopo Gentile, Anat Hendelman, Michael Passalacqua, Xingang Wang, Michael Alonge, Hamsini Suresh, Ryan Santos, Blaine Fitzgerald, Gina Robitaille, Edeline Gagnon, Melissa Kramer, Sara Goodwin, W. Richard McCombie, Jaime Prohens, Tiina E. Särkinen, Amy Frary, Jesse Gillis, Joyce Van Eck, Ben Langmead, Zachary B. Lippman, and Michael C. Schatz. Panagram: Alignment-free and interactive pan-genome visualization, 2023. Poster at RECOMB-Seq; `https://github.com/kjenike/panagram`.

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

[30] Sana Kashgouli. MONI can find k-MEMs (CPM 2023). `https://youtu.be/P312oAzZpms`, 2023. [Accessed 18-Jul-2023].

[31] J.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.

[32] Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *International Symposium on String Processing and Information Retrieval*, pages 201–206. Springer, 2010.

[33] T. W. Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and S. M. Yiu. High throughput short read alignment via bi-directional bwt. In *2009 IEEE International Conference on Bioinformatics and Biomedicine*, pages 31–36, 2009.

[34] N Jesper Larsson and Alistair Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

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

[36] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[37] Kewen Liao, Matthias Petri, Alistair Moffat, and Anthony Wirth. Effective construction of relative lempel-ziv dictionaries. In *Proceedings of the 25th International Conference on World Wide Web*, pages 807–816, 2016.

[38] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, page 657–666, USA, 2002. Society for Industrial and Applied Mathematics.

[39] Daniel J. Nasko, Sergey Koren, Adam M. Phillippy, and Todd J. Treangen. Refseq database growth influences the accuracy of k-mer-based lowest common ancestor species identification. *Genome Biology*, 19(1):165, Oct 2018.

[40] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[41] Gonzalo Navarro. Indexing highly repetitive string collections, part i: Repetitiveness measures. *ACM Comput. Surv.*, 54(2), mar 2021.

[42] Gonzalo Navarro. Indexing highly repetitive string collections, part ii: Compressed indexes. *ACM Comput. Surv.*, 54(2), feb 2021.

[43] Gonzalo Navarro. Personal communications, 2022.

[44] Gonzalo Navarro. Computing mems on repetitive text collections. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[45] Gonzalo Navarro and Nicola Prezza. Universal compressed text indexing. *Theoretical Computer Science*, 762:41–50, 2019.

[46] Gonzalo Navarro and Víctor Sepúlveda. Practical indexing of repetitive collections using relative lempel-ziv. In *2019 Data Compression Conference (DCC)*, pages 201–210. IEEE, 2019.

[47] Yakov Nekrich. New data structures for orthogonal range reporting and range minima queries. In *Proceedings of the Thirty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '21, page 1191–1205, USA, 2021. Society for Industrial and Applied Mathematics.

[48] Tatsuya Ohno, Keisuke Goto, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. Lz-abt: A practical algorithm for α-balanced grammar compression. In Costas Iliopoulos, Hon Wai Leong, and Wing-Kin Sung, editors, *Combinatorial Algorithms*, pages 323–335, Cham, 2018. Springer International Publishing.

[49] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

[50] Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. Moni: A pangenomics index for finding mems. *bioRxiv*, 2021.

[51] Igor Tatarnikov, Ardavan Shahrabi Farahani, Sana Kashgouli, and Travis Gagie. Moni can find k-mems. In *34th Annual Symposium on Combinatorial Pattern Matching*, 2023.

[52] Derrick E. Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with kraken 2. *Genome Biology*, 20(1):257, Nov 2019.

[53] Derrick E. Wood and Steven L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3):R46, Mar 2014.