2021

Efficient Data Structures for Text Processing Applications

Paniz Abedin
University of Central Florida

EFFICIENT DATA STRUCTURES FOR TEXT PROCESSING APPLICATIONS

by

PANIZ ABEDIN
M.Sc University of Central Florida, 2020
B.Sc Sharif University of Technology, 2015

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2021

Major Professor: Sharma Thankachan

ABSTRACT

This thesis is devoted to designing and analyzing efficient text indexing data structures and associated algorithms for processing text data. The general problem is to preprocess a given text or a collection of texts into a space-efficient index to quickly answer various queries on this data. Basic queries such as counting/reporting a given pattern's occurrences as substrings of the original text are useful in modeling critical bioinformatics applications. This line of research has witnessed many breakthroughs, such as the suffix trees, suffix arrays, FM-index, etc. In this work, we revisit the following problems:

1. The Heaviest Induced Ancestors problem

2. Range Longest Common Prefix problem

3. Range Shortest Unique Substrings problem

4. Non-Overlapping Indexing problem

For the first problem, we present two new space-time trade-offs that improve the space, query time, or both of the existing solutions by roughly a logarithmic factor. For the second problem, our solution takes linear space, which improves the previous result by a logarithmic factor. The techniques developed are then extended to obtain an efficient solution for our third problem, which is newly formulated. Finally, we present a new framework that yields efficient solutions for the last problem in both cache-aware and cache-oblivious models.

ACKNOWLEDGMENTS

I would like to gratefully thank my advisor, Dr. Sharma Thankachan, for all his help and support throughout this journey. I would have hardly imagined writing this dissertation without his encouragement, invaluable guidance, and kindness. He, without a doubt, made my stay at UCF a memorable experience. I also wish to thank the esteemed members of my dissertation committee Dr. Shibu Yooseph, Dr. Wei Zhang, Dr. Varadraj Gurupur, and Dr. Samiul Hasan for their insightful comments and help.

I would also like to thank my family Ramak, Hamid, and Sina for their unrelenting support and encouragement. This would not have been possible without your emotional support. Thanks for sticking with me all the way to the end.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Texts are fundamental data types in Bioinformatics, Data compression and Search engines. An abundance of challenges while using this type of data motivates theoretical and experimental research on algorithms and data structures for text processing. Pattern matching is the main problem in this field of research in which we are given a text and the task is to maintain a data structure, so that, whenever a pattern comes as a query, all of its occurrences can be returned or identified efficiently [98]. A full-text index is a data structure storing a text (a string or a set of strings) and supporting string matching queries. Suffix tree is a classical full-text index which was introduced by Weiner [120] for solving pattern matching queries in a time proportional to the length of the query pattern. Let $\mathsf{T}[1,n]$ be a text of length n and $\mathsf{T}[i,n]$ be the suffix starting at position i. The suffix tree of a text $\mathsf{T}[1,n]$ is a compact trie storing all of its suffixes ($\mathsf{T}[i,n], \forall i \in \{1 \cdots n\}$) and it can be constructed in $O(n)$ time. Given a text $\mathsf{T}[1,n]$, we can preprocess T using its suffix tree in $O(n)$ time so that later we can find out whether an unknown pattern P of length m has an occurrence in T or not in $O(m)$ time. The linear running time for the pattern matching problem is exceptionally important due to the huge size of input texts in modern applications [42]. Although a suffix tree takes linear space for its construction, the overhead is relatively large. Thus, more space-efficient indexes such as suffix arrays [91] and other compact data structures have been developed [99]. In this thesis, we study various string matching queries for which we provide efficient data structures and algorithms. All results throughout Chapters 3, 4 and 5 assume the standard unit-cost word RAM model, in which any standard arithmetic or boolean bitwise operation on word-sized operands takes constant time. the space is measured in words of $\log n$ bits unless specified otherwise.

Map

In Sections 1.1, 1.2, 1.3 and 1.4 we briefly explain the main queries that are focused in this thesis. In Section 1.5, the problems for which we provide improved data structures, as well as our results for each of them, have been described. In Chapter 2, we bring the preliminaries including the definitions and data structures that are used in our approaches. Chapters 3, 4, 5 and 6 are our framework for the heaviest induced ancestors, Range Longest Common prefix, Range Shortest Unique Substrings and Non-Overlapping Indexing problems respectively.

1.1 Longest Common Substring Queries

Given two strings T_1 and T_2 , each of length at most n, the longest common substring(LCS) problem is finding a longest substring which occurs in both T_1 and T_2. Using suffix tree data structure explained in Section 1, the LCS problem can be solved in $O(n)$ time, which is optimal [120]. The LCS problem is a classic problem in String algorithms. Different types of LCS problem have been studied [25, 85, 94]. In problems that the input data is large, the suffix tree solution for the LCS problem would be impractical. Thus, compressed and succinct data structures for solving these types of queries have been developed [100]. In 2013, Gagie et al. introduced the *Heaviest induced Ancestors problem (HIA)* which can be applied to build an LZ-compressed index [48, 84] to answer LCS problem [57]. The Heaviest Induced Ancestors problem is a useful primitive in several generalizations of LCS queries such as Dynamic Longest Common Substring, Longest Common Substring of LZ77 Compressed Strings, and All-Pairs Longest Common Substring Problems. In Chapter 3 we present our framework for solving the HIA problem.

1.2 Longest Common Prefix Queries

Let $\mathsf{T}[1,n]$ be a text of length n and $\mathsf{T}[i,n]$ be the suffix starting at position i. The Longest Common Extension problem takes a string T as an input and for each pair (i,j) computes the longest substring of T which starts at both positions i and j. We can generalize this definition for any two input strings. For any two strings X and Y, let $\mathsf{LCP}(X,Y)$ denote their longest common prefix which is an important primitive employed in various string matching algorithms. By preprocessing a text $\mathsf{T}[1,n]$ (over an alphabet set Σ) into a suffix tree data structure, we can compute the longest common prefix of any two suffixes of T, say $\mathsf{T}[i,n]$ and $\mathsf{T}[j,n]$, denoted by $\mathsf{LCP}(\mathsf{T}[i,n],\mathsf{T}[j,n])$, in constant time. By the LCP notation, we can formulate the LCE queries as follows:

Definition 1 (LCE queries)**.** *The* LCE *of a text* $\mathsf{T}[1,n]$ *w.r.t a query* (i,j)*, where* $i,j \in [1,n]$ *is*

$$\mathsf{LCE}(i,j) = \mathsf{LCP}(\mathsf{T}[i,n],\mathsf{T}[j,n])$$

From now onward, we use the shorthand notation $\mathsf{lcp}(i,j)$ for the length of $\mathsf{LCP}(\mathsf{T}[i,n],\mathsf{T}[j,n])$. Given its wide range of applicability, various generalizations of LCP has also been studied [20, 21, 16, 103, 58]. The notation and concept of LCP have been applied in fundamental string matching problems. A suffix tree data structure along with the *Lowest Common Ancestor* queries in trees can be used to solve LCP queries in constant time [68, 67, 30, 28]. In most cases, LCP information are computed along with the construction of suffix arrays. Augmenting the suffix arrays with the LCP information allows to simulate the bottom-up traversal of suffix trees without explicitly storing the suffix tree data structures [80]. LCP queries are essential tools for computing maximal repeats, palindromes, unique substrings, matching with wild cards, dictionary matching, document retrieval, approximate pattern matching and other fundamental string problems [67]. In 1988, Landau and Vishkin introduced a 2-D array called MAX-LENGTH for a text which stores the

LCE values for each (i, j) pairs. Then, they present an algorithm for finding all occurrences of a query pattern of length m in an input text of length n allowing $k >= 0$ mismatches, where k is an input integer [87]. In 1993, Udi Manber and Gene Myers introduced the LCP array as a data structure to improve the running time of their string search algorithm. Their algorithm takes $O(n \log n)$ time to compute the LCP array along with the suffix array construction [92]. Later on, several linear time algorithms have been published for the LCP array construction [80, 51, 93]. The computation of LCP is important in compression as well. For instance, in order to compute the Ziv-Lempel compression the LCP computation is required [124]. In this thesis, we focus on the "range" versions of this problem. This is a natural generalization of the LCP problem. The line of research was initiated by Cormode and Muthukrishnan. They studied the *Interval Longest Common Prefix* (Interval-LCP) problem in the context of data compression [41, 82, 88].

Definition 2 (Interval-LCP)**.** *The Interval-*LCP *of a text* $\mathsf{T}[1,n]$ *w.r.t a query* (p, α, β)*, where* $p, \alpha, \beta \in [1,n]$ *and* $\alpha < \beta$ *is*

$$\mathsf{ilcp}(p, \alpha, \beta) = \max\{\mathsf{lcp}(p, i) \mid i \in [\alpha, \beta]\}$$

As observed by Keller et al. [82], any Interval-LCP query on T can be reduced to two orthogonal range successor/predecessor queries over n points in two dimensions (2D). Therefore, using the best known data structures for orthogonal range successor/predecessor queries [101], we can answer any Interval-LCP query on T in $O(\log^{\varepsilon} n)$ time using an $O(n)$ space data structure, where $\varepsilon > 0$ is an arbitrarily small positive constant. Moreover, queries with $p \in [\alpha, \beta]$ can be answered in faster $O(\log^{\varepsilon} \delta)$ time, where $\delta = \beta - \alpha + 1$ is the length of the input range [103].

We study on another variation of LCP, studied by Amir et al. [20, 21], is the following.

Definition 3 (Range-LCP)**.** *The Range-LCP of a text* $\mathsf{T}[1,n]$ *w.r.t a range* $[\alpha, \beta]$*, where* $1 \leq \alpha <$

$\beta \leq n$ *is*

$$\mathsf{rlcp}(\alpha,\beta) = \max\{\mathsf{lcp}(i,j) \mid i \neq j \textit{ and } i,j \in [\alpha,\beta]\}$$

Chapter 4 presents our techniques for solving $\mathsf{rlcp}(.,.)$ queries.

1.3 Shortest Unique Substrings Queries

Finding regularities in strings is one of the main topics of combinatorial pattern matching and its applications [89]. Among the most well-studied types of string regularities is the notion of repeat. Let $\mathsf{T}[1,n]$ be a string of length n. A substring $\mathsf{T}[i,j]$ of T is called a repeat if it occurs more than once in T. The notion of unique substring is dual: it is a substring $\mathsf{T}[i,j]$ of T that does not occur more than once in T. Computing repeats and unique substrings has applications in computational biology [110, 70] and in information retrieval [105, 83].

In this thesis, we are interested in the notion of shortest unique substring. All shortest unique substrings of string T can be computed in $O(n)$ time using the suffix tree data structure [67, 121]. Many different problems based on this notion have already been studied. Pei et al. [105] considered the following problem on the so-called position (or point) queries. Given a position i of T, return a shortest unique substring of T covering i. The authors gave an $O(n^2)$-time and $O(n)$-space algorithm, which finds the shortest unique substring covering every position of T. Since then, the problem has been revisited and optimal $O(n)$-time algorithms have been presented by Ileri et al. [75] and by Tsuruta et al. [116]. Several other variants of this problem have been investigated [1, 9, 63, 62, 77, 71, 97, 111, 96, 119].

We introduce a natural generalization of the shortest unique substring problem. Specifically, our focus is on the range version of the problem, which we call the *Range Shortest Unique Substring*

(rSUS) problem. The task is to construct a data structure over T to be able to answer the following type of online queries efficiently. Given a range $[\alpha,\beta]$, return a shortest substring $\mathsf{T}[k,k+h-1]$ of T with exactly one occurrence (starting position) in $[\alpha,\beta]$; i.e., $k \in [\alpha,\beta]$, there is no $k' \in [\alpha,\beta]$ $(k' \neq k)$ such that $\mathsf{T}[k,k+h-1] = \mathsf{T}[k',k'+h-1]$, and h is minimal. Note that this substring, $\mathsf{T}[k,k+h-1]$, may end at a position $k+h-1 > \beta$. Further note that there may be multiple shortest unique substrings.

Range queries are a classic data structure topic [125, 31, 29]. A range query $q = f(A,i,j)$ on an array of n elements over some set S, denoted by $A[1,n]$, takes two indices $1 \leq i \leq j \leq n$, a function f defined over arrays of elements of S, and outputs $f(A[i,j]) = f(A[i],\ldots,A[j])$. Range query data structures have also been considered specifically for strings [22, 17, 2, 61]. For instance, in bioinformatics applications we are often interested in finding regularities in certain regions of a DNA sequence [106, 10, 26, 76, 11]. In the *Range-LCP* problem, defined by Amir et al. [22], the task is to construct a data structure over T to be able to answer the following type of online queries efficiently. Given a range $[\alpha,\beta]$, return $i,j \in [\alpha,\beta]$ such that the length of the longest common prefix of $\mathsf{T}[i,n]$ and $\mathsf{T}[j,n]$ is maximal among all pairs of suffixes within this range. The state of the art is an $O(n)$-word data structure supporting $O(\log^{O(1)} n)$-time (polylogarithmic-time) queries [2] (see also [95, 61]).

1.4 Non-overlapping Indexing

Text indexing is fundamental to many areas in Computer Science such as Information Retrieval, Bioinformatics, etc. The primary goal here is to pre-process a long text $\mathsf{T}[1,n]$ (given in advance), such that whenever a shorter pattern $P[1,m]$ comes as query, all occ occurrences (or simply, starting positions) of P in T can be reported efficiently. Such queries can be answered in optimal $O(m+\mathsf{occ})$ time using the classic *Suffix tree* data structure [117, 122]. It takes $O(n)$ words of space. In

this thesis, we focus on a variation of the text indexing problem, known as the *non-overlapping indexing*, which is central to data compression [24, 40].

The non-overlapping indexing problem is defined as follows: pre-process a given text $\mathsf{T}[1,n]$ of length n into a data structure such that whenever a pattern $P[1,m]$ comes as an input, we can efficiently report the largest set of non-overlapping occurrences of P in T. The best-known solution is by Cohen and Porat [ISAAC 2009]. The size of their structure is $O(n)$ words and the query time is optimal $O(m+\mathsf{nocc})$, where nocc is the output size. Later, Ganguly *et al.* [CPM 2015 and Algorithmica 2020] proposed a compressed space solution. We study this problem in the cache-oblivious model and present a new data structure of size $O(n\log n)$ words. It can answer queries in optimal $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}}{B})$ I/O operations, where B is the block size. The space can be improved to $O(n\log_{M/B} n)$ in the cache-aware model, where M is the size of main memory. Additionally, we study a generalization of this problem with an additional range $[s,e]$ constraint. Here the task is to report the largest set of non-overlapping occurrences of P in T, that are within the range $[s,e]$. We present an $O(n\log^2 n)$ space data structure in the cache-aware model that can answer queries in optimal $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}_{[s,e]}}{B})$ I/O operations, where $\mathsf{nocc}_{[s,e]}$ is the output size.

1.5 Our Contributions

1.5.1 The Heaviest Induced Ancestors Problem

The *heaviest induced ancestors* (HIA) problem is defined as follows: Let T_1 and T_2 be two weighted trees, having n_1 and n_2 nodes respectively, and let $n = n_1 + n_2$. The weight of a node u in either of the trees is given by $\mathsf{W}(u)$ and $\mathsf{W}(u) > \mathsf{W}(\mathsf{parent}(u))$, where $\mathsf{parent}(u)$ is the parent node of u. For convenience, the pre-order rank of a node u is also denoted by u. Each tree has exactly $m \leq \min\{n_1,n_2\}$ leaves. Leaves in both trees are labeled and the labeling of the leaves in T_2 is

a permutation of the labeling of the leaves in T_1. A pair of nodes, one each from T_1 and T_2, are *induced* if the leaves in the respective subtrees have at least one common label. For any two nodes u and v in a tree, the node v is an ancestor of u iff v is on the path from u to the root of the tree. Moreover, v is a proper ancestor u only if $u \neq v$. We revisit the problem, introduced by Gagie *et al.* [57]. In this section, we present an overview to our framework and in Chapter 3, we provide the details of our results. This work previously appeared as Abedin, Paniz, et al. "The heaviest induced ancestors problem revisited." Annual Symposium on Combinatorial Pattern Matching (CPM 2018) [6]. An extension of this work is currently under the review in Algorithmica journal.

Problem 1 (Heaviest Induced Ancestors (HIA) Problem [57])**.** *Given a node $u_1 \in \mathsf{T}_1$ and a node $u_2 \in \mathsf{T}_2$, find $\mathsf{HIA}(u_1,u_2)$, which is defined as the pair of* induced *nodes (u_1^*,u_2^*) with the highest combined weight $\mathsf{W}(u_1^*)+\mathsf{W}(u_2^*)$, such that u_1^* (resp., u_2^*) is an ancestor of u_1 (resp., u_2).*

Gagie *et al.* [57] achieved the following space-time trade-offs in the standard word RAM model of computation with word size $\Omega(\log n)$ bits. Here and henceforth, ε is an arbitrarily small positive constant.

- an $O(n\log^2 n)$-word and $O(\log n \log\log n)$ query time
- an $O(n\log n)$-word and $O(\log^2 n)$ query time
- an $O(n)$-word data and $O(\log^{3+\varepsilon} n)$ query time.

We present two new data structures, with improved bounds to answer the heaviest induced ancestors query over two trees of n nodes in total as follows:

Theorem 1. *A heaviest induced ancestors query over two trees of n nodes in total can be answered*

- *in $O(\log n \log\log n)$ time using an $O(n\log n)$-word data structure, or*

- *in* $O\left(\frac{\log^2 n}{\log\log n}\right)$ *time using an* $O(n)$*-word data structure.*

Applications to String Matching

Let $\mathsf{LCS}(X,Y)$ denote the longest common substring (LCS) of two strings X and Y.

1.5.1.1 Longest Common Substring of LZ77 Compressed Strings

Problem 2. *Build a data structure for a string S of length N, whose LZ77 parsing contains n phrases, that supports the following query: given a pattern P, report* $\mathsf{LCS}(S,P)$.

If one were to forego the compression requirement, the problem can be easily solved by maintaining a suffix tree [120] of S in $O(N)$ words yielding $O(|P|)$ query time. On the other hand, we can also answer $\mathsf{LCS}(S,P)$ queries using compressed/succinct data structures, such as the FM Index or Compressed Suffix Array [50, 66, 108], with a slight slow down in the query time. However, for strings having a repetitive structure, LZ77-based compression techniques [127] offer better space-efficiency than that obtained using FM-Index or Compressed Suffix Array.

Gagie *et al.* [57] showed that Problem 2 can be solved using an $O(n\log N + n\log^2 n)$-word index with a very high probability in $O(|P|\log n\log\log n)$ query time. Alternatively, they also presented an $O(n\log N)$-word index with query time $O(|P|\log^2 n)$. Using Theorem 1 and the techniques in [57], we present an improved result for Problem 2 (see Theorem 2). We omit the details as they are immediate from the discussions in [57].

Theorem 2. *Given a string S of length N, we can build an $O(n\log N)$-word structure that reports* $\mathsf{LCS}(S,P)$ *in $O(|P|\log n\log\log n)$ time with a very high probability, where n is the number of phrases in an LZ77 parsing of S.*

1.5.1.2 All-Pairs Longest Common Substring Problem

Here we are given a collection $T_1, T_2, \ldots, T_d$ of d strings, each of length roughly n, and the task is to compute $\mathsf{LCS}(T_i, T_j)$ for all (i, j) pairs. This is a useful primitive in several bioinformatics applications [12, 39]. However, a conditional lower bound based on the boolean matrix multiplication suggests that significant improvements over the naive $O(d^2 n)$ time algorithm is unlikely [114] in the general case. However, we present an improved solution for the cases where many strings are highly similar (a.k.a highly repetitive). Specifically, we present a data structure of space and preprocessing time $\tilde{O}(nd)$, that computes $\mathsf{LCS}(T_i, T_j)$ for any i, j in time $\tilde{O}\left(\frac{\min\{|T_i|,|T_j|\}}{|\mathsf{LCS}(T_i,T_j)|}\right)$. We defer details to Section 3.3.1.

1.5.1.3 Dynamic Longest Common Substring Problem

In [18], Amir *et al.* introduced the following problem: build a data structure over two strings T_1 and T_2 of total n characters over an alphabet set Σ, such that given a query (p, α), where $p \in [p, |T_1|]$ and $\alpha \in \Sigma$, report $\mathsf{LCS}(T_1^*, T_2)$, where T_1^* is a new string obtained by changing the pth character of T_1 by α. They presented an $O(n \log^3 n)$ space data structure with $O(\log^3 n)$ query time. We not only improve, but also propose a solution to solve a more general case, where the query consists of a set S of $(position, character)$ pairs, and the task is to compute $\mathsf{LCS}(T_1^*, T_2)$, where T_1^* is obtained from T_1 by making the changes (substitutions) as specified by S. We achieve the same space-time trade-offs as that of Theorem 1, where time is the per substitution. Details are deferred to Section 3.3.2. See [14, 15, 23, 56, 55, 118] for other related work.

The problem is even more complicated when the changes allowed in both strings. Amir *et al.* [19] proposed an $\tilde{O}(n)$ space solution with query time $\tilde{O}(n^{2/3})$. A new result improving this query time to $\tilde{O}(1)$ has been announced recently [102].

1.5.2 The Range-LCP Problem

We revisit the range variation of the LCP queries, studied by Amir et al. [20, 21] which is defined as 3. In Chapter 4, we provide the details of our results. This work previously appeared as Abedin, Paniz, et al. "A linear-space data structure for range-LCP queries in poly-logarithmic time." Theoretical Computer Science 822 (2020), and Computing and Combinatorics (COCOON 2018) [3, 2].

Problem 3. *The Range-LCP of a text* $\mathsf{T}[1,n]$ *w.r.t a range* $[\alpha,\beta]$*, where* $1 \leq \alpha < \beta \leq n$ *is*

$$\mathsf{rlcp}(\alpha,\beta) = \max\{\mathsf{lcp}(i,j) \mid i \neq j \text{ and } i,j \in [\alpha,\beta]\}$$

In order to efficiently solve the data structure version of this problem, Amir et al. [20, 21] introduced the concept of "**bridges**" and "**optimal bridges**" and showed that any Range-LCP query on $\mathsf{T}[1,n]$ can be reduced to an equivalent 2D range maximum query over a set of $O(n\log n)$ weighted points in 2D. Therefore, an $O(n\log^{1+\varepsilon} n)$ space data structure with $O(\log\log n)$ query time is immediate from the best known result for 2D range maximum problem [36]. The construction time is $O(n\log^2 n)$. By choosing an alternative structure for 2D (2-sided) range maximum query [1], the space can be improved to $O(n\log n)$ with a slowdown in query time to $O(\log n)$. This sets an interesting question, whether it is possible to reduce the space further without sacrificing the poly-logarithmic query time. Unfortunately, the query times of the existing linear space solutions (listed below) are dependent on the parameter δ, which is $\Theta(n)$ in the worst case.

- $O(n)$ space and $O(\delta\log\log n)$ query time [20, 21].
- $O(n)$ space and $O(\sqrt{\delta}\log^{\varepsilon}\delta)$ query time [103] (also see [61])

[1] See Theorem 9 in [104] on sorted dominance reporting in 3D.

To this end, we present our main contribution below. Our model of computation is the word RAM with word size $\Theta(\log n)$.

Theorem 3. *A text $\mathsf{T}[1,n]$ can be preprocessed into an $O(n)$ space data structure in $O(n\log n)$ time, such that any Range-LCP query on T can be answered in $O(\log^{1+\varepsilon} n)$ time.*

1.5.3 The Range-SUS Problem

An *alphabet* Σ is a finite nonempty set of elements called *letters*. We fix a *string* $\mathsf{T}[1,n] = \mathsf{T}[1]\cdots\mathsf{T}[n]$ over Σ. The *length* of T is denoted by $|\mathsf{T}| = n$. By $\mathsf{T}[i,j] = \mathsf{T}[i]\cdots\mathsf{T}[j]$, we denote the *substring* of T starting at position i and ending at position j of T. We say that another string P has an *occurrence* in T or, more simply, that P *occurs* in T if $\mathsf{P} = \mathsf{T}[i, i+|\mathsf{P}|-1]$, for some i. Thus, we characterize an occurrence of P by its *starting position* i in T. A *prefix* of T is a substring of T of the form $\mathsf{T}[1,i]$ and a *suffix* of T is a substring of T of the form $\mathsf{T}[i,n]$.

We next formally define the next main problem considered in this thesis. In Chapter 5, we provide the details of our results. This work previously appeared as Abedin, Paniz, et al. "Range shortest unique substring queries." International Symposium on String Processing and Information Retrieval. Springer, Cham, 2019 and "Efficient data structures for range shortest unique substring queries." Algorithms 13.11 (2020) [5, 4]

Problem 4. rSUS

Preprocess: String $\mathsf{T}[1,n]$.

Query: Range $[\alpha,\beta]$, where $1 \le \alpha \le \beta \le n$.

Output: (p,ℓ) such that $\mathsf{T}[p, p+\ell-1]$ is a shortest string with exactly one occurrence in $[\alpha,\beta]$.

If $\alpha = \beta$ the answer $(\alpha, 1)$ is trivial. So, in the rest we assume that $\alpha < \beta$.

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Example 1. Given $\mathsf{T} = \mathbf{caabcaddaacaddaaaabac}$ and a query $[\alpha, \beta] = [5, 16]$, we need to find a shortest substring of T with exactly one occurrence in $[5, 16]$. The output here is $(p, \ell) = (10, 2)$, because $\mathsf{T}[10, 11] = \mathbf{ac}$ is the shortest substring of T with exactly one occurrence in $[5, 16]$.

Our main results are summarized below. We consider the standard word-RAM model of computations with w-bit machine words, where $w = \Omega(\log n)$, for stating our results.

Theorem 4. *We can construct an $O(n \log n)$-word data structure which can answer any* rSUS *query on $T[1, n]$ in $\mathcal{O}(\log_w n)$ time.*

Theorem 5. *We can construct an $\mathcal{O}(n)$-word data structure which can answer any* rSUS *query on $T[1, n]$ in $\mathcal{O}(\sqrt{n} \log^{\varepsilon} n)$ time, where $\varepsilon > 0$ is an arbitrarily small constant.*

1.5.4 I/O-Efficient Data Structures for Non-Overlapping Indexing

In Chapter 6, we focus on a variation of the text indexing problem, known as the *non-overlapping indexing*, which is central to data compression [24, 40]. This work previously appeared as Hooshmand, Sahar, et al. "Non-Overlapping Indexing-Cache Obliviously." Annual Symposium on Combinatorial Pattern Matching (CPM 2018) and "I/O-efficient data structures for non-overlapping indexing." Theoretical Computer Science 857 (2021) [73, 74].

Problem 5 (Non-overlapping Indexing)**.** *Preprocess a text $\mathsf{T}[1, n]$ into a data structure that supports the following query: given a pattern $P[1, m]$, report the largest set of occurrences of P in T (denote its size by* nocc*), such that any two (distinct) text positions in the output are separated by at least m characters.*

The *range non-overlapping indexing* is a generalization of the above problem.

Problem 6 (Range Non-overlapping Indexing)**.** *Preprocess a text* $\mathsf{T}[1,n]$ *into a data structure that supports the following query: given a pattern* $P[1,m]$ *and a range* $[s,e]$*, report the largest set of occurrences of P in* T*, that are within the range* $[s,e]$ *(denote its size by* $\mathsf{nocc}_{[s,e]}$*), such that any two (distinct) text positions in the output are separated by at least m characters.*

Both these problems are well studied in the internal memory (RAM) model of computation. The initial solutions were obtained via a reduction to the *orthogonal range next value* problem. Although efficient, they were not optimal in terms of query time [44, 81, 101]. The first non-trivial solutions were by Cohen and Porat [40]: they presented an $O(n)$ space and optimal $O(m+\mathsf{nocc})$ query time solution for Problem 5 and an $O(n\log^{\varepsilon} n)$ space and near-optimal $O(m+\log\log n+\mathsf{nocc}_{[s,e]})$ query time solution for Problem 6, where $\varepsilon>0$ is an arbitrarily small constant. To achieve these results, they exploited the periodicity of both pattern and (substrings of) text. Subsequently, Ganguly *et al.* [59, 60] presented the first succinct space solution for Problem 5 and an optimal query time solution for Problem 6. We revisit Problems 5 and 6 in the secondary memory model and present the following results.

Theorem 6. *There exists a data structure that is initialized with a text* $\mathsf{T}[1,n]$*, takes* $O(n\log n)$ *words of space and supports the following query in the cache-oblivious model. Given a string* $P[1,m]$*, the data structure reports the largest set of non-overlapping occurrences of* $P[1,m]$ *in* T *in their sorted order in optimal* $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}}{B})$ *I/O operations, where* nocc *is the output size.*

The space complexity can be improved in the cache-aware model.

Theorem 7. *There exists a data structure that is initialized with a text* $\mathsf{T}[1,n]$*, takes* $O(n\log_{M/B} n)$ *words of space and supports the following query in the cache-aware model. Given a string* $P[1,m]$*, the data structure reports the largest set of non-overlapping occurrences of* $P[1,m]$ *in* T *in their sorted order in optimal* $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}}{B})$ *I/O operations, where* nocc *is the output size.*

Theorem 8. *There exists an $O(n\log^2 n)$ space data structure for the range non-overlapping indexing problem in the cache-aware model, where n is the length of the input text* T. *The data structure supports reporting the largest set of non-overlapping occurrences of an input pattern $P[1,m]$ within a given range $[s,e]$ in their sorted order in optimal $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}_{[s,e]}}{B})$ I/O operations, where $\mathsf{nocc}_{[s,e]}$ is the output size.*

We now present a brief description of our models of computations.

1.5.4.1 The Models of Computations

The data structures presented in Chapters 3, 4 and 5 are in the Word-RAM model. In the Word-RAM model, we assume that the memory is partitioned into continuous blocks (or words). At any point of time, we can load data into a block or access data within a block in constant time. We always assume the size of a block is $\Theta(\log n)$ bits, where n denotes the size of the problem in hand. For Non-overlapping indexing problem, we present our data structures in the secondary memory model in the context of very large input data. Here we assume that the data (and the data structure) is too big to fit within the main memory, therefore deployed in a (much larger, but slower) secondary memory. Popular models of computation are (i) the cache-aware model and (ii) the cache-oblivious model.. In the *cache-aware model* (a.k.a. external memory model, I/O model, and disk access model), introduced by Aggarwal and Vitter [8] the CPU is connected directly to an internal memory (of size M words), which is then connected to a very large external memory (disk). The disk is partitioned into blocks/pages and the size of each block is B words. The CPU can only work on data inside the internal memory. Therefore, to work on some data in the external memory, the corresponding blocks have to be transferred to internal memory. The transfer of a block from external memory to internal memory (or vice versa) is referred to as an I/O operation. The operations inside the internal memory are orders of magnitude faster than the time for an I/O

operation. Therefore, they are considered free, and the efficiency of an algorithm is measured in terms of the number of I/O operations. The *cache-oblivious model* is essentially the same as above, except the following key twist: M and B are unknown at the time of the design of algorithms and data structures [54, 53]. This means, if a cache-oblivious algorithm performs optimally between two levels of the memory hierarchy, then it is optimal at any level of the memory hierarchy. Lastly, cache-oblivious algorithms are usually more intricate than cache-aware algorithms. Generally, we assume $M > B^{2+\Theta(1)}$, known as the tall cache assumption.

CHAPTER 2: PRELIMINARIES AND TERMINOLOGIES

In this section, we bring various definitions, notations and known data structures that are utilized in our newly introduced data structures and algorithms.

2.1 Suffix Trees and Suffix Arrays

Let $T[1,n]$ be a text over an alphabet set Σ and let $\$ \notin \Sigma$ be a special symbol. The suffix tree ST_T (resp., prefix tree PT_T) of T is a compact trie over all strings in the set $S = \{T[i,n] \circ \$ \mid i \in [1,n]\}$ (resp., $S^R = \{\overleftarrow{T[1,i]} \circ \$ \mid i \in [1,n]\}$). Here $\circ$ denotes concatenation and $\overleftarrow{T[1,i]}$ denotes the reverse of $T[1,i]$. The suffix array SA_T and the inverse suffix array ISA_T (resp., the prefix array PA_T and the inverse prefix array IPA_T) of T are arrays of length n, such that $\mathsf{SA}_T[i] = j$ and $\mathsf{ISA}_T[j] = i$ (resp., $\mathsf{PA}_T[i] = j$ and $\mathsf{IPA}_T[j] = i$) iff $T[j,n] \circ \$$ (resp., $\overleftarrow{T[1,j]} \circ \$$) is the ith smallest string in S (resp., S^R) in the lexicographic order.

Both ST_T and PT_T has exactly n leaves, and the edges are labeled. For any node u in either of the tree, we use $\mathsf{path}(u)$ to denote the concatenation of edge labels on the path from the root to u. The $\mathsf{path}(\cdot)$ of ith leftmost leaf in ST_T (resp., PT_T) is the same as the ith smallest string in S (resp., S^R) in the lexicographic order. Therefore, for any two nodes u and v within the same tree with w being their lowest common ancestor (LCA), the longest common prefix (LCP) of $\mathsf{path}(u)$ and $\mathsf{path}(v)$ is the same as $\mathsf{path}(\mathsf{LCA}(u,v))$. Note that the prefix tree (resp., prefix array) of a text is equivalent to the suffix tree (resp., suffix array) of the reverse of the text.

In addition, $\mathsf{size}(u)$ denote the number of leaves under u. The locus of a string S in ST, denoted by $\mathsf{locus}(S)$ is the node closest to root, such that S is a prefix of the node's path. The array SA is called the *suffix array* of T. The suffix range of a string S, denoted by $[\mathsf{sp}(S), \mathsf{ep}(S)]$ is the

range of (contiguous) leaves in the subtree of $\mathsf{locus}(S)$. Therefore, the set of occurrences of S is $\{\mathsf{SA}[i] \mid \mathsf{sp}(S) \leq i \leq \mathsf{ep}(S)\}$. The suffix range of S can be computed in $O(|S|)$ time. Space is $O(n)$ words for both suffix array and suffix tree.

The suffix tree over a collection of strings $T_1, T_2, \ldots, T_d$ over an alphabet set Σ is called a generalized suffix tree (GST), which is a compact trie over all strings in $\cup_{j=1}^{d} \{T_j[i, |T_j|] \circ \$_j \mid i \in [1, |T_j|]\}$, where $\$_1, \$_2, \ldots, \$_d$ are distinct symbols that do not appear in Σ. The corresponding (generalized) suffix array, prefix tree, and prefix array can be defined similarly.

All the above data structures can be constructed in linear space and time (assuming integer alphabet) [120, 46]. After that, we can find $\mathsf{LCA}(\cdot,\cdot)$ in $O(1)$ time [29]. Therefore, the length of the longest common prefix (or suffix) of any two strings (or their substrings) in the collection can be computed in constant time. A weighted level ancestor query (u,l) asks to report the highest ancestor w of u, such that $|\mathsf{path}(w)| \geq l$. By preprocessing the suffix tree into a linear space data structure, the level answer queries on suffix trees can be answered in constant time [65, 47]. We refer to [67] for further reading.

There also exist linear-space suffix tree representations in both secondary memory models. The solution in the cache-aware model is by Ferragina [49] and is called the String-B tree. The cache-oblivious solution is by Brodal and Fagerberg [34], which is based on an intricate concept of decomposing the tree into what they call Giraffe trees. In both cases, the locus, as well as the suffix range of any string S can be computed in optimal $O(|S|/B + \log_B n)$ I/O operations.

2.2 Predecessor/Successor Queries

Let $\mathcal{S}$ be a subset of $\mathcal{U} = \{0, 1, 2, 3, \ldots, U-1\}$ of size n. A predecessor search query p on $\mathcal{S}$ asks to return p if $p \in \mathcal{S}$, else return $\max\{q < p \mid q \in \mathcal{S}\}$. Similarly, a successor query p on $\mathcal{S}$ asks to

return p if $p \in \mathcal{S}$, else return $\min\{q > p \mid q \in \mathcal{S}\}$. By preprocessing S into a y-fast trie of size $O(n)$ words, we can answer such queries in $O(\log\log U)$ time [123].

2.3 Fully-Functional Succinct Tree

Let T be a tree having n nodes, such that nodes are numbered from 1 to n in the ascending order of their pre-order rank. Also, let ℓ_i denote the ith leftmost leaf. Then by maintaining an index of size $2n + o(n)$ bits, we can answer the following queries on T in constant time [109]:

- $\mathsf{parent}_{\mathsf{T}}(u)$ = parent of node u.
- $\mathsf{size}_{\mathsf{T}}(u)$ = number of leaves in the subtree of u.
- $\mathsf{nodeDepth}_{\mathsf{T}}(u)$ = number of nodes on the path from u to the root of T.
- $\mathsf{levelAncestor}_{\mathsf{T}}(u, D)$ = ancestor w of u such that $\mathsf{nodeDepth}(w) = D$.
- $\mathsf{lMost}_{\mathsf{T}}(u) = i$, where ℓ_i is the leftmost leaf in the subtree of u.
- $\mathsf{rMost}_{\mathsf{T}}(u) = j$, where ℓ_j is the rightmost leaf in the subtree of u.
- $\mathsf{lca}_{\mathsf{T}}(u, v)$ = lowest common ancestor (LCA) of two nodes u and v.

We omit the subscript "T" if the context is clear.

2.4 Range Maximum Query (RMQ) and Path Maximum Query (PMQ)

Let $A[1, n]$ be an array of n elements. A range maximum query $\mathsf{RMQ}_A(a, b)$ asks to return $k \in [a, b]$, such that $A[k] = \max\{A[i] \mid i \in [a, b]\}$. Path maximum query (PMQ) (or bottleneck edge query [45])

is a generalization of RMQ from arrays to trees. Let T be a tree having n nodes, such that each node u is associated with a score. A path maximum query $\mathsf{PMQ}_{\mathsf{T}}(a,b)$ returns the node k in T, where k is a node with the highest score among all nodes on the path from node a to node b. Cartesian tree based solutions exist for both problems. The space and query time are $2n+o(n)$ bits and $O(1)$, respectively [45, 52].

2.5 2D-Range Maximum Query (2D-RMQ)

Let $\mathcal{S}$ be a set of m weighted points in a $[1,n]\times[1,n]$ grid. A 2D-RMQ with input (a,b,a',b') asks to return the highest weighted point in $\mathcal{S}$ within the orthogonal region corresponding to $[a,b]\times[a',b']$. Data structures with the following space-time trade-offs are known for this problem.

- $O(m)$ space, $O(m\log m)$ preprocessing time and $O(\log^{1+\varepsilon} m)$ query time [38].
- $O(m\log^{\varepsilon} n)$ space, $O(m\log m)$ preprocessing time and $O(\log\log n)$ query time [36].

2.6 Orthogonal Range Queries in 2-Dimension

Let $\mathcal{P}$ be a set of n points in an $[1,n]\times[1,n]$ grid. Then,

- An orthogonal range counting query (a,b,c,d) on $\mathcal{P}$ returns the cardinality of $\{(x,y)\in\mathcal{P} \mid x\in[a,b], y\in[c,d]\}$
- An orthogonal range emptiness query (a,b,c,d) on $\mathcal{P}$ returns "EMPTY" if the cardinality of the set $\{(x,y)\in\mathcal{P} \mid x\in[a,b], y\in[c,d]\}$ is zero. Otherwise, it returns "NOT-EMPTY".
- An orthogonal range predecessor query (a,b,c) on $\mathcal{P}$ returns the point in $\{(x,y)\in\mathcal{P} \mid x\in[a,b], y\leq c\}$ with the highest y-coordinate value, if one exists.

- An orthogonal range successor query (a,b,c) on $\mathcal{P}$ returns the point in $\{(x,y) \in \mathcal{P} \mid x \in [a,b], y \geq c\}$ with the lowest y-coordinate value, if one exists.
- An orthogonal range selection query (a,b,k) on $\mathcal{P}$ returns the point in $\{(x,y) \in \mathcal{P} \mid x \in [a,b]\}$ with the kth lowest y-coordinate value.

By maintaining an $O(n)$ word structure, we can answer orthogonal range counting queries in $O(\log / \log\log n)$ time [78], orthogonal range emptiness queries in $O(\log^{\varepsilon} n)$ time [36], orthogonal range predecessor/successor queries in $O(\log^{\varepsilon} n)$ time [101] and orthogonal range selection queries in $O(\log n / \log\log n)$ time [35]. Alternatively, by maintaining an $O(n \log\log n)$ space structure, we can answer orthogonal range emptiness and orthogonal range predecessor/successor queries in $O(\log\log n)$ time [36, 126].

2.7 Heavy Path and Heavy Path Decomposition

We now define the heavy path decomposition [69, 112] of a tree T having n nodes. First, the nodes in T are categorized into light and heavy. The root node is *light* and exactly one child of every internal node is heavy. Specifically, the child having the largest number of nodes in its subtree (ties are broken arbitrarily). The first heavy path of T is the path starting at T's root, and traversing through every heavy node to a leaf. Each off-path subtree of the first heavy path is further decomposed recursively. Clearly, a tree with m leaves has m heavy paths. Let u be a node on a heavy path H, then $\mathsf{hp\_root}(u)$ is the highest node on H and $\mathsf{hp\_leaf}(u)$ is the lowest node on H. Note that $\mathsf{hp\_root}(\cdot)$ is always light.

Lemma 1. For a tree having n nodes, the path from the root to any leaf traverses at most $\lceil \log n \rceil$ light nodes. Consequently, the sum of the subtree sizes of all light nodes (i.e., the starting node of a heavy path) put together is at most $n\lceil \log n \rceil$ [112].

CHAPTER 3: THE HEAVIEST INDUCED ANCESTORS PROBLEM

In this chapter, we describe our framework for solving the *heaviest induced ancestors* problem which is defined as follows: let T_1 and T_2 be two weighted trees, where the weight $\mathsf{W}(u)$ of a node u in either of the two trees is more than the weight of u's parent. Additionally, the leaves in both trees are labeled and the labeling of the leaves in T_2 is a permutation of those in T_1. A node $x \in \mathsf{T}_1$ and a node $y \in \mathsf{T}_2$ are induced, iff their subtrees have at least one common leaf label. A heaviest induced ancestor query $\mathsf{HIA}(u_1,u_2)$ is: given a node $u_1 \in \mathsf{T}_1$ and a node $u_2 \in \mathsf{T}_2$, output the pair (u_1^*,u_2^*) of induced nodes with the highest combined weight $\mathsf{W}(u_1^*)+\mathsf{W}(u_2^*)$, such that u_1^* is an ancestor of u_1 and u_2^* is an ancestor of u_2.

We present two data structures for solving problem 1 with improved space-time trade-offs as follows:

- in $O(\log n \log\log n)$ time using an $O(n \log n)$-word data structure, or
- in $O\left(\dfrac{\log^2 n}{\log\log n}\right)$ time using an $O(n)$-word data structure.

3.1 Our Framework

We assume that both trees T_1 and T_2 are compacted, i.e., any internal node has at least two children. This ensures that the number of internal nodes is strictly less than the number of leaves (m). Thus, $n \leq 4m-2$. We remark that this assumption can be easily removed without affecting the query time. We maintain the tree topology of T_1 and T_2 succinctly in $O(n)$ bits with constant time navigational support (refer to Section 2.3). Define two arrays, $\mathsf{Label}_k[1,m]$ for $k = 1$ and 2, such that $\mathsf{Label}_k[j]$ is the label associated with the jth leaf node in T_k. The following is a set of m

two-dimensional points based on tree labels.

$$\mathcal{P} = \{(i,j) \mid i,j \in [1,m] \text{ and } \mathsf{Label}_1[i] = \mathsf{Label}_2[j]\}$$

We pre-process $\mathcal{P}$ into a data structure, to support various range queries described in Section 2.6. For range counting and selection, we maintain data structures with $O(n)$ space and $O(\log n/\log\log n)$ time. For range successor/predecessor and emptiness queries, we have two options: and $O(n\log\log n)$ space structure with $O(\log\log n)$ time, and an $O(n)$ space structure with $O(\log^{\varepsilon} n)$ time. We employ the first result in our $O(n\log n)$ space solution and the second result in our $O(n)$ space solution.

3.1.1 Basic Queries

Lemma 2 (Induced-Check)**.** *Given two nodes x,y, where $x \in \mathsf{T}_1$ and $y \in \mathsf{T}_2$, we can check if they are induced or not*

- *in $O(\log\log n)$ time using an $O(n\log\log n)$ space structure, or*
- *in $O(\log^{\varepsilon} n)$ time using an $O(n)$ space structure.*

Proof. The task can be reduced to a range emptiness query, because x and y are induced iff the set $\{(i,j) \in \mathcal{P} \mid (i,j) \in [\mathsf{lMost}(x), \mathsf{rMost}(x)] \times [\mathsf{lMost}(y), \mathsf{rMost}(y)]\}$ is not empty. □

Definition 4 (Partner)**.** The partner of a node $x \in \mathsf{T}_1$ w.r.t a node $y \in \mathsf{T}_2$, denoted by $\mathsf{partner}(x/y)$ is the lowest ancestor y' of y, such that x and y' are induced. Likewise, $\mathsf{partner}(y/x)$ is the lowest ancestor x' of x, such that x' and y are induced.

Lemma 3 (Find Partner)**.** *Given two nodes x,y, where $x \in \mathsf{T}_1$ and $y \in \mathsf{T}_2$, we can find $\mathsf{partner}(x/y)$ as well as $\mathsf{partner}(y/x)$*

- *in $O(\log\log n)$ time using an $O(n\log\log n)$ space structure, or*
- *in $O(\log^{\varepsilon} n)$ time using an $O(n)$ space structure.*

Proof. To find $\mathsf{partner}(x/y)$, first check if x and y are induced. If yes, then $\mathsf{partner}(x/y) = y$. Otherwise, find the last leaf node $\ell_a \in \mathsf{T}_2$ before y in pre-order, such that x and ℓ_a are induced (ℓ_a denotes a-th leftmost leaf). Also, find the first leaf node $\ell_b \in \mathsf{T}_2$ after y in pre-order, such that x and ℓ_b are induced. Both tasks can be reduced to orthogonal range predecessor/successor queries.

$$(\cdot, a) = \arg\max_j \{(i,j) \in \mathcal{P} \mid (i,j) \in [\mathsf{lMost}(x), \mathsf{rMost}(x)] \times [1, \mathsf{lMost}(y)]\}$$

$$(\cdot, b) = \arg\min_j \{(i,j) \in \mathcal{P} \mid (i,j) \in [\mathsf{lMost}(x), \mathsf{rMost}(x)] \times [\mathsf{rMost}(y), m]\}$$

Clearly, an ancestor of y and x are induced iff either ℓ_a or ℓ_b is in its subtree. Therefore, we report the lowest node among $u_a = \mathsf{lca}(\ell_a, y)$ and $u_b = \mathsf{lca}(\ell_a, y)$ as $\mathsf{partner}(x/y)$. The computation of $\mathsf{partner}(y/x)$ is analogous.

□

3.1.2 Overview

For any two nodes u and v in the same tree T, define $\mathsf{Path}(u, v, \mathsf{T})$ as the set of nodes on the path from u to v. Let root_1 be the root of T_1 and root_2 be the root of T_2. Throughout this chapter, (u_1, u_2) denotes the input and $\mathsf{HIA}(u_1, u_2) = (u_1^*, u_2^*)$ denotes the output. Clearly, $u_2^* = \mathsf{partner}(u_1^*/u_2)$ and $u_1^* = \mathsf{partner}(u_2^*/u_1)$. Therefore,

$$(u_1^*, u_2^*) = \arg\max_{(x,y)} \{W(x) + W(y) \mid y \in \mathsf{Path}(\mathsf{root}_2, u_2, \mathsf{T}_2) \textbf{ and } x = \mathsf{partner}(y/u_1)\}$$

To evaluate the above equation efficiently, we explore the heavy path decomposition of T_2.

Definition 5 (Special Nodes)**.** For each light node $w \in \mathsf{T}_2$, we identify a set $\mathsf{Special}(w)$ of nodes in T_1 (which we call special nodes) as follows: a leaf node $\ell_i \in \mathsf{T}_1$ is *special* iff ℓ_i and w are induced. An internal node in T_1 is special iff it is the lowest common ancestor of two special leaves. Additionally, for each node $x \in \mathsf{Special}(w)$, define its score w.r.t. w as the sum of weights of x and the node $\mathsf{partner}(x/\mathsf{hp\_leaf}(w)) \in \mathsf{T}_2$. Formally,

$$\mathsf{score}_w(x) = W(x) + W(\mathsf{partner}(x/\mathsf{hp\_leaf}(w))$$

Moreover, $|\mathsf{Special}(w)| \leq 2\mathsf{size}(w) - 1$ and $\sum_{w \text{ is a light node}} |\mathsf{Special}(w)| = O(n \log n)$.

To answer an HIA query (u_1, u_2), we first identify some nodes in T_1 and T_2 as follows. Nodes $w_1 = \mathsf{root}_2, w_2, \ldots, w_k$ are the *light* nodes in $\mathsf{Path}(\mathsf{root}_2, u_2, \mathsf{T}_2)$ (in the ascending order of their pre-order ranks). Nodes $t_1, t_2, \ldots, t_k$ are also in $\mathsf{Path}(\mathsf{root}_2, u_2, \mathsf{T}_2)$, such that $t_k = u_2$ and $t_h = \mathsf{parent}(w_{h+1})$ for $h < k$. Therefore, $\mathsf{Path}(\mathsf{root}_2, u_2, \mathsf{T}_2) = \cup_{h=1}^{k} \mathsf{Path}(w_h, t_h, \mathsf{T}_2)$. Next, $\alpha_1, \alpha_2, \ldots, \alpha_k$ and $\beta_1, \beta_2, \ldots, \beta_k$ are nodes in $\mathsf{Path}(\mathsf{root}_1, u_1, \mathsf{T}_1)$, such that for $h = 1, 2, \ldots, k$, $\alpha_h = \mathsf{partner}(t_h/u_1)$ and $\beta_h = \mathsf{partner}(w_h/u_1)$. Clearly, there exists an $f \in [1, k]$ such that $u_2^* \in \mathsf{Path}(w_f, t_f, \mathsf{T}_2)$. See Figure 1 for an illustration. We now present several lemmas, which form the basis of our solution.

Lemma 4. *The node* $u_1^* \in \mathsf{Path}(\alpha_f, \beta_f, \mathsf{T}_1)$.

Proof. We prove this via proof by contradiction arguments.

- Suppose u_1^* is a proper ancestor of α_f. Then, α_f are t_f induced and $W(\alpha_f) + W(t_f) > W(u_1^*) + W(u_2^*)$, a contradiction. Therefore, u_1^* is in the subtree of α_f.
- Suppose u_1^* is in the proper subtree of β_f. Then, u_1^* and w_f are also induced. Therefore,

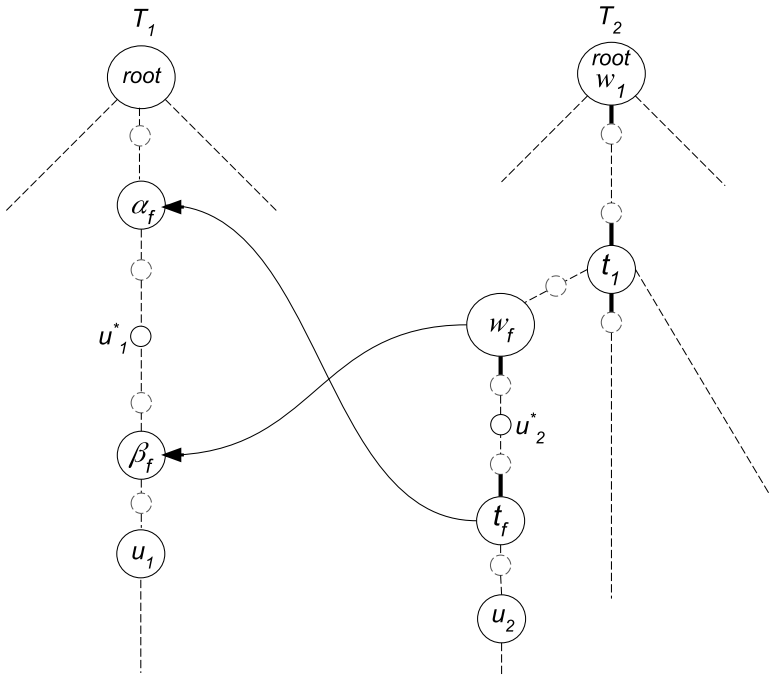

Figure 3.1: We refer to Section 3.1.2 for the description of this figure.

$\mathsf{partner}(w_f/u_1)$ is u_1^* or a node in the subtree of u_1^*. This implies, $\beta_f = \mathsf{partner}(w_f/u_1)$ is in the proper subtree of β_f, a contradiction. Therefore, u_1^* is an ancestor of β_f.

This completes the proof.

□

Lemma 5. *The node* $u_1^* \in \mathsf{Special}(w_f) \cup \{\beta_f\}$.

Proof. Let z (if exists) be the first node in $\mathsf{Special}(w_f)$ on the path from u_1^* to β_f. Then,

- if z exists, then $u_1^* \notin \mathsf{Special}(w_f)$ gives a contradiction as follows. The intersection of the

following two sets is empty: (i) set of labels of the leaves in the subtree of u_1^*, but not in the subtree of z and (ii) set of labels associated with the leaves in the subtree of w_f. This implies, z and u_2^* are induced (because u_1^* and u_2^* are induced) and $W(z)+W(u_2^*) > W(u_1^*)+W(u_2^*)$, a contradiction.

- otherwise, if z does not exist, then it is possible that $u_1^* \notin \mathsf{Special}(w)$. However, in this case, $u_1^* = \beta_f$ (proof follows from similar arguments as above).

In summary, $u_1^* \in \mathsf{Special}(w_f) \cup \{\beta_f\}$.

□

Lemma 6. *For any* $x \in \mathsf{Path}(\alpha_f, \beta_f, \mathsf{T}_1) \backslash \{\alpha_f\}$,
$\mathsf{partner}(x/u_2) = \mathsf{partner}(x/\mathsf{hp\_leaf}(w_f))$.

Proof. We claim that for any $x \in \mathsf{Path}(\alpha_f, \beta_f, \mathsf{T}_1) \backslash \{\alpha_f\}$, $\mathsf{partner}(x/u_2)$ is a proper ancestor of t_f. The proof follows from contradiction as follows. Suppose, there exists an $x \in \mathsf{Path}(\alpha_f, \beta_f, \mathsf{T}_1) \backslash \{\alpha_f\}$, such that $\mathsf{partner}(x/u_2)$ is in the subtree of t_f. Then, x and t_f are induced. This means, $\alpha_f = \mathsf{partner}(t_f/u_1)$ is a node in the subtree of x, a contradiction.

Since, $\mathsf{partner}(x/u_2)$ is a proper ancestor of t_f, $\mathsf{partner}(x/u_2) = \mathsf{partner}(x/r)$ for any node r in the subtree of t_f. Therefore, by choosing $r = \mathsf{hp\_leaf}(w_f)$, we obtain Lemma 6.

□

Corollary 1. *For any* $x \in \Big(\mathsf{Path}(\alpha_f, \beta_f, \mathsf{T}_1) \backslash \{\alpha_f\}\Big)$,

$$W(x) + W(\mathsf{partner}(x/u_2)) = W(x) + W(\mathsf{partner}(x/\mathsf{hp\_leaf}(w_f))) = \mathsf{score}_{w_f}(x)$$

Lemma 7. *The node* $u_1^* \in \{\alpha_f, \beta_f, \gamma_f\}$*, where*

$$\gamma_f = \arg\max_x \{\mathsf{score}_{w_f}(x) \mid x \in \mathsf{Special}(w_f) \cap \Big(\mathsf{Path}(\alpha_f, \beta_f, \mathsf{T}_1) \backslash \{\alpha_f, \beta_f\}\Big)$$

Proof. Follows from Lemma 4, Lemma 5, Lemma 6 and Corollary 1. □

Lemma 8. *Let* $\mathcal{C} = \cup_{h=1}^{k} \{\alpha_h, \beta_h, \gamma_h\}$*, where*

$$\gamma_h = \arg\max_x \{\mathsf{score}_{w_h}(x) \mid x \in \mathsf{Special}(w_h) \cap \Big(\mathsf{Path}(\alpha_h, \beta_h, \mathsf{T}_1) \backslash \{\alpha_h, \beta_h\}\Big)$$

Then,

$$(u_1^*, u_2^*) = \arg\max_{(x,y)} \{W(x) + W(y) \mid x \in \mathcal{C} \textbf{ \textit{and}} \ y = \mathsf{partner}(x/u_2)\}$$

Proof. Since f is unknown, we invoke Lemma 6 for $f = 1, 2, 3, \ldots, k \leq \log n$. □

Next, we show how to transform the result in Lemma 8 into an efficient data structure.

3.2 Our Data Structures

We start by defining a crucial component of our solution.

Definition 6 (Induced Subtree)**.** The induced subtree $\mathsf{T}_1(w)$ of T_1 w.r.t. a light node $w \in \mathsf{T}_2$ is a tree having exactly $|\mathsf{Special}(w)|$ number of nodes, such that

- for each node $x \in \mathsf{T}_1(w)$, there exists a node $\mathsf{Map}_w(x) \in \mathsf{Special}(w)$ and
- for each $x' \in \mathsf{Special}(w)$, there exists a node $\mathsf{invMap}_w(x') \in \mathsf{T}_1(w)$, such that

$$\mathsf{lca}_{\mathsf{T}_1}(\mathsf{Map}_w(x), \mathsf{Map}_w(y)) = \mathsf{Map}_w(\mathsf{lca}_{\mathsf{T}_1(w)}(x, y))$$

Note that a node x is a leaf in $\mathsf{T}_1(w)$ iff $\mathsf{Map}_w(x)$ is a leaf in $\mathsf{T}_1(w)$. In the following lemmas, we present two space-time trade-offs on induced subtrees.

Lemma 9. *By maintaining an $O(n \log n)$ space structure, we can compute $\mathsf{Map}_w(\cdot)$ and $\mathsf{invMap}_w(\cdot)$ for any light node $w \in \mathsf{T}_2$ in time $O(1)$ and $O(\log\log n)$, respectively.*

Proof. Let $L_w[1, |\mathsf{Special}(w)|]$ be an array, such that $L_w[x] = \mathsf{Map}_w(x)$. For each w, maintain L_w and a y-fast trie [123] over it. The total space is $O(n \log n)$. Now, any $\mathsf{Map}_w(\cdot)$ query can be answered in constant time. Also, for any $x' \in \mathsf{Special}(w)$, $\mathsf{invMap}_w(x')$ is the number of elements in L_w that are $\leq x'$. Therefore, an $\mathsf{invMap}_w(\cdot)$ can be reduced to a predecessor search and answered in $O(\log\log n)$ time.

□

Lemma 10. *By maintaining an $O(n)$ space structure, we can compute $\mathsf{Map}_w(\cdot)$ and $\mathsf{invMap}_w(\cdot)$ for any light node $w \in \mathsf{T}_2$ in time $O(\log n / \log\log n)$.*

Proof. Let node p be the rth leaf in $\mathsf{T}_1(w)$ and $q = \mathsf{Map}_w(p)$ be the sth leaf in T_1. Then, s is the x-coordinate of the rth point in $\{(i, j) \in \mathcal{P} \mid (i, j) \in [1, m] \times [\mathsf{lMost}(w), \mathsf{rMost}(w)]\}$ in the ascending order of x-coordinates. Also, r is the number of points in $\{(i, j) \in \mathcal{P} \mid (i, j) \in [1, s] \times [\mathsf{lMost}(w), \mathsf{rMost}(w)]\}$. Therefore, given p, we can compute r, then s and q in $O(\log n / \log\log n)$ time via a range selection query on $\mathcal{P}$. Similarly, given q, we can compute s and then r and p in $O(\log n / \log\log n)$ time via a range counting query on $\mathcal{P}$.

Now, if p is an internal node in $\mathsf{T}_1(w)$, then $\mathsf{Map}_w(p)$ is the same as $\mathsf{lca}_{\mathsf{T}_1}(\mathsf{Map}_w(\ell_L), \mathsf{Map}_w(\ell_R))$, where ℓ_L and ℓ_R are the first and last leaves in the subtree of p. Similarly, if q is an internal node

in T_1, then $\mathsf{invMap}_w(q) = \mathsf{lca}_{\mathsf{T}_1(w)}(\mathsf{invMap}_w(\ell_A), \mathsf{invMap}_w(\ell_B))$ as follows:

$$(A, \cdot) = \arg\min_i \{(i,j) \in \mathcal{P} \mid (i,j) \in [\mathsf{lMost}(q), \mathsf{rMost}(q)] \times [\mathsf{lMost}(w), \mathsf{rMost}(w)]\}$$

$$(B, \cdot) = \arg\max_i \{(i,j) \in \mathcal{P} \mid (i,j) \in [\mathsf{lMost}(q), \mathsf{rMost}(q)] \times [\mathsf{lMost}(w), \mathsf{rMost}(w)]\}$$

Here, A and B can be computed via range successor/predecessor queries in $O(\log^{\varepsilon} n)$ time. Therefore, the total time is $\log^{\varepsilon} n + \log n / \log\log n = O(\log n / \log\log n)$ time.

□

Lemma 11. *Given an input (a,b,w), where w is a light node in T_2 and, a and b are nodes in $\mathsf{T}_1(w)$, we can report the node with the highest $\mathsf{score}_w(\mathsf{Map}_w(\cdot))$ over all nodes on the path from a to b in $\mathsf{T}_1(w)$ in $O(1)$ time using an $O(n)$ space structure.*

Proof. For each $\mathsf{T}_1(w)$, maintain the Cartesian tree for path maximum query (refer to Section 2.4). Space for a particular w is $|\mathsf{Special}(w)|(2+o(1))$ bits and space over all light nodes w in T_2 is $O(n \log n)$ bits (from Lemma 1), equivalently $O(n)$ words. For an input (a,b,w), the answer is $\mathsf{PMQ}_{\mathsf{T}_1(w)}(a,b)$. □

3.2.1 Our $O(n \log n)$ Space Data Structure

We maintain T_1 and T_2 explicitly, so that the weight of any node in either of the trees can be accessed in constant time. Moreover, we maintain fully-functional succinct representation of their topologies (refer to Section 2.3) for supporting various operations in $O(1)$ time. Additionally, we maintain the structures for answering Induced-Check and Find-Partner queries in $O(\log\log n)$ time, data structures for range predecessor/successor queries on $\mathcal{P}$ in $O(\log\log n)$ time (refer to Section 2.6) and the structures described in Lemma 9 and Lemma 11. Thus, the total space is

$O(n \log n)$ words.

We now present the algorithm for computing the output (u_1^*, u_2^*) for a given input (u_1, u_2). The following are the key steps.

1. Find w_h and t_h for $h = 1, 2, \ldots, k \leq \log n$.
2. Find α_h and β_h for $h = 1, 2, \ldots, k \leq \log n$.
3. Let α'_h be the first and β'_h be the last special node (w.r.t. w_h) on the path from α_h (excluding α_h) to β_h (excluding β_h). Also, let

$$\gamma_h = \mathsf{Map}_{w_h}\Big(\mathsf{PMQ}_{\mathsf{T}_1(w_h)}\Big(\mathsf{invMap}_{w_h}(\alpha'_h), \mathsf{invMap}_{w_h}(\beta'_h)\Big)\Big)$$

   Compute γ_h for $h = 1, 2, \ldots, k \leq \log n$.
4. Obtain $\mathcal{C} = \cup_{h=1}^{k} \{\alpha_h, \beta_h, \gamma_h\}$ and report

$$(u_1^*, u_2^*) = \arg\max_{(x,y)} \{W(x) + W(y) \mid x \in \mathcal{C} \textbf{ and } y = \mathsf{partner}(x/u_2)\}$$

The correctness follows immediately from Lemma 8. We now bound the time complexity. Step 1 takes $O(k)$ time and step 2 takes $O(k)$ number of Find-Partner queries with $O(\log \log n)$ time per query. The procedure for computing α'_h and β'_h is the following.

Find the child α''_h of α_h on the path from α_h to β_h. Then $\alpha'_h = \mathsf{lca}_{\mathsf{T}_1}(\ell_{a_h}, \ell_{b_h})$, where ℓ_{a_h} (resp. ℓ_{b_h}) is the first (resp. last) special leaf in the subtree of α''_h (w.r.t w_h). To compute a_h and b_h, we rely on range predecessor/successor queries on $\mathcal{P}$:

$$(a_h, \cdot) = \arg\min_i \{(i, j) \in \mathcal{P} \mid (i, j) \in [\mathsf{lMost}(\alpha''_h), \mathsf{rMost}(\alpha''_h)] \times [\mathsf{lMost}(w_h), \mathsf{rMost}(w_h)]\}$$

$$(b_h, \cdot) = \arg\max_i \{(i,j) \in \mathcal{P} \mid (i,j) \in [\mathsf{lMost}(\alpha''_h), \mathsf{rMost}(\alpha''_h)] \times [\mathsf{lMost}(w_h), \mathsf{rMost}(w_h)]\}$$

Find the rightmost special (w.r.t. w_h) leaf ℓ_{d_h} before β_h and the leftmost special (w.r.t. w_h) leaf ℓ_{g_h} after the last leaf in the subtree of β_h. For this, we rely on range predecessor/successor queries on $\mathcal{P}$:

$$(d_h, \cdot) = \arg\max_i \{(i,j) \in \mathcal{P} \mid (i,j) \in [1, \mathsf{lMost}(\alpha''_h) - 1] \times [\mathsf{lMost}(w_h), \mathsf{rMost}(w_h)]\}$$

$$(g_h, \cdot) = \arg\min_i \{(i,j) \in \mathcal{P} \mid (i,j) \in [\mathsf{rMost}(\alpha''_h) + 1, m] \times [\mathsf{lMost}(w_h), \mathsf{rMost}(w_h)]\}$$

Then, $\beta'_h = \mathsf{lca}_{\mathsf{T}_1}(\ell_{d_h}, \ell_{g_h})$ if β_h and w_h are not induced (i.e., there does not exist a special node (w.r.t. w_h) under β_h). Otherwise, β'_h is the lowest node among $\mathsf{lca}_{\mathsf{T}_1}(\ell_{d_h}, \beta_h)$ and $\mathsf{lca}_{\mathsf{T}_1}(\beta_h, \ell_{g_h})$.

The time for a range predecessor/successor query on $\mathcal{P}$ is $O(\log\log n)$. Therefore, computation of α'_h and β'_h takes $O(\log\log n)$ time, and an additional $O(\log\log n)$ for evaluating γ_h. Therefore, the total time for step 3 is $O(k\log\log n)$. Finally, step 4 also takes $O(k\log\log n)$ time. By putting everything together, the total time complexity is $k\log\log n = O(\log n \log\log n)$.

3.2.2 Our Linear Space Data Structure

We obtain our linear space data structure by replacing all super-linear space components in the previous solution by their space efficient counterparts. Specifically, we use linear space structures for Induced-Check, Find-Partner, and range predecessor/successor with query time $O(\log^{\varepsilon} n)$. Also, we use the structure in Lemma 10 instead of the structure in Lemma 9. Thus, the total space is $O(n)$ words.

The query algorithm remains the same. The time complexity is: $O(k)$ for step 1, $O(k\log^{\varepsilon} n)$ for step 2, $O(k\log n/\log\log n)$ for step 3 and $O(k\log^{\varepsilon} n)$ for step 4. Thus, total time is $k\log n/\log\log n = O(\log^2 n/\log\log n)$.

A Note on Construction

We remark that our data structures can be constructed in $O(n \log^{O(1)} n)$ time.

3.3 Applications to String Processing

3.3.1 All-Pairs Longest Common Substring Problem

The formal problem definition along with our result are as below:

Definition 7. *Given a collection $T_1, T_2, \ldots, T_d$ of d strings, compute $\mathsf{LCS}(T_i, T_j)$ for all (i, j) pairs.*

Theorem 9. *A collection $T_1, T_2, \ldots, T_d$ of d strings of total length N can be preprocessed into a data structure in $\tilde{O}(N)$ space and time, that supports an $\mathsf{LCS}(T_i, T_j)$ query for any i, j pair in $\tilde{O}\left(\frac{\min\{|T_i|,|T_j|\}}{|\mathsf{LCS}(T_i,T_j)|}\right)$ time.*

3.3.1.1 Data Structure

We maintain a generalized suffix tree GST and a generalized prefix tree GPT over the collection. Additionally, for each $i \in [1, d]$, we maintain our HIA data structure over the pair of trees ST_{T_i} and PT_{T_i}, such that the weight of nodes being their $\mathsf{stringDepth}$. We say that a node u_1 in ST_{T_i} and a node u_2 in PT_{T_i} are induced iff there exists a k, such that the leaf (say ℓ_a) corresponding to the suffix $T_i[k, |T_i|]$ is under u_1 and the leaf (say ℓ_b) corresponding to the (reverse of the) prefix $T_i[1, k-1]$ is under u_2. To align this with the original definition of HIA problem, we assign label k to both ℓ_a and ℓ_b. The total space, as well as the preprocessing time, is bounded by $\tilde{O}(N)$.

3.3.1.2 Query Algorithm

Without loss of generality, we assume $|T_i| \leq |T_j|$. First, we present an efficient procedure for accomplishing the following task: compute $\lambda(T_i, k, T_j)$, the length of the longest substring of T_i, which covers a position in $\{k-1, k\}$ and also has an occurrence in T_j. We say that a substring $T_j[x, y]$ covers k iff $k \in [x, y]$.

- Find the largest f and the largest r, such that $T_i[k, k+f-1]$ and $T_i[k-r, k-1]$ occurs in T_j. Then, find the lowest node u_1 in ST_{T_j} and the lowest node u_2 in PT_{T_j}, such that $T_i[k, k+f-1]$ is a prefix of $\mathsf{path}(u_1)$ and $\overleftarrow{T_i[k-r, k-1]}$ is a prefix of $\mathsf{path}(u_2)$. This step can be implemented in $O(\log n)$ time using standard techniques.
- If $W(u_1) = f$ and $W(u_2) = r$, simply find $(u_1^*, u_2^*) = \mathsf{HIA}(u_1, u_2)$ and report $W(u_1^*) + W(u_2^*)$. Otherwise, find $(u_1', u_2') = \mathsf{HIA}(\mathsf{parent}(u_1), \mathsf{parent}(u_2))$, the lowest ancestor u_2'' of u_2 (resp., u_1'' of u_1), such that u_1 and u_2'' (resp., u_1'' and u_2) are induced. Then, report $\max\{f + W(u_2''), W(u_1'') + r, W(u_1') + W(u_2')\}$.

The time complexity is $\tilde{O}(1)$ and the correctness can be easily verified. We can use the above procedure to check whether $|\mathsf{LCS}(T_i, T_j)| \geq \tau$ for any given τ as follows: repeat the above procedure for all values of $k \in \{\tau, 2\tau, 3\tau, \dots\}$. Then report YES if at least one among the answers is greater than or equal to τ, and NO otherwise. Therefore, via a simple binary search on τ, we can compute $|\mathsf{LCS}(T_i, T_j)|$. Note that the values of τ chosen are always greater than $|\mathsf{LCS}(T_i, T_j)|/2$. This yields the desired query time.

3.3.2 Dynamic Longest Common Substring Problem

Problem 7 (LCS after k changes)**.** *Given two strings T_1 and T_2 of total length n over an alphabet set Σ, build a data structure that supports the following query: given a set S of k $(position, character)$ pairs, where*

$$S = \{(p_i, \alpha_i) \mid p_i \in [1, |T_1|], i \in [1,k] \textbf{ and } \alpha_i \in \Sigma\}$$

report (the length of) $\mathsf{LCS}(T_1^*, T_2)$, where T_1^* is the string obtained from T_1 by replacing its p_i-th character by α_i for each $i \in [1,k]$.

We achieve the following result.

Theorem 10. *There exist data structures with the following space-time trade-offs for the* LCS after k changes *problem on two strings of total length n:*

1. *$O(n \log n)$ word and $O(k \cdot \log n \log\log n)$ query time, and*
2. *$O(n)$ word and $O\left(k \cdot \dfrac{\log^2 n}{\log\log n}\right)$ query time.*

Let $p_1 < p_2 < \cdots < p_k$ and $\phi = \cup_{i=1}^{k} \{p_i - 1, p_i\}$. Our approach is to find the longest common substring of T_1^* and T_2 (i) that does not cover any position in ϕ and (ii) covers a position in ϕ, and then report the longest among them as $\mathsf{LCS}(T_1^*, T_2)$. We handle these cases separately, and the following convention will be used: any subarray or substring over a range $[a,b]$ is empty if $a > b$.

Handling Case 1

Let $\psi = \{[1, p_1 - 2], [p_1 + 1, p_2 - 2], [p_2 + 1, p_3 - 2], \ldots, [p_k + 1, n]\}$. Clearly, the answer we are looking for is $\max\{|\mathsf{LCS}(T_1^*[i,j], T_2)| \mid [i,j] \in \psi\}$. Since $\mathsf{LCS}(T_1^*[i,j], T_2) = \mathsf{LCS}(T_1[i,j], T_2)$ for all

$[i,j] \in \psi$, we can solve this case in $O(k \log n)$ time using the structure below.

Lemma 12. *The strings T_1 and T_2 can be preprocessed into an $O(n)$ structure, where $|T_1| + |T_2| = n$, that supports the following query in $O(\log n)$ time: given a range $[x,y]$, report (the length of)* $\mathsf{LCS}(T_1[x,y], T_2)$.

Proof. Maintain an array L, where $L[i] = |\mathsf{LCP}(T_1[i,|T_1|], T_2)|$ and range maximum query (RMQ) data structure over it. To answer a query $[x,y]$, first find the rightmost position z, such that $z + L[z] - 1 < y$. This is possible in $O(\log n)$ time via a binary search since $i + L[i] - 1$ is monotonic. Then, compute $|\mathsf{LCS}(T_1[x,z]), T_2)| = \mathsf{RMQ}_L(x,z)$ and $|\mathsf{LCS}(T_1[z+1,y]), T_2)| = y - (z+1) + 1$, and report the largest among them.

□

Handling Case 2

We maintain a generalized suffix tree and a generalized prefix tree of T_1 and T_2. Also, maintain our HIA data structure over the pair of trees ST_{T_2} and PT_{T_2} as in Section 3.3.1.1. Moreover, we can compute (u_1^i, u_2^i, f_i, r_i) for $i = 1,2,\ldots,k$ by processing the input set (of k changes) in $O(k \log n)$ time using standard techniques, where

1. f_i is the length of the longest substring of T_1^* that starts at position p_i
2. r_i is the length of the longest substring of T_1^* that ends at $p_i - 1$.
3. u_1^i is the lowest node in ST_{T_2}, s.t. $T_1[p_i, p_i + f_i - 1]$ is a prefix of $\mathsf{path}(u_1^i)$.
4. u_2^i is the lowest node in PT_{T_2}, s.t. $\overleftarrow{T_1[p_i - r_i, p_i - 1]}$ is a prefix of $\mathsf{path}(u_2^i)$.

Finally, for each p_i, we compute the (length of the) longest common substring of T_1^* and T_2 covering a position in $\{p_i - 1, p_i\}$ using an HIA query as in Section 3.3.1.2, and report the largest among.

In summary, the total space is dominated by the space of our HIA structure and the complexity of query time per change is $O(\log n)$ plus the time for an HIA query. This completes the proof of Theorem 10.

3.4 Conclusion

In this chapter, we revisited the heaviest induced ancestors problem which is initially introduced by Gagie et al. [57]. We presented two new space-time trade-offs that improve the space, query time, or both of the existing solutions by roughly a logarithmic factor. We also showed that our solution can be deployed further to solve multiple string matching problems such as All-Pairs Longest Common Substring and Dynamic Longest Common Substring Problems. In addition, our data structure for the heaviest induced ancestors problem can be used to compute the Longest Common Substring of LZ77 Compressed Strings and matching statistics on repetitive texts [64].

CHAPTER 4: THE RANGE LONGEST COMMON PREFIX PROBLEM

We briefly sketch the framework by Amir et al. [20] in Section 4.1. Section 4.2 and Section 4.3 are dedicated for the details of our solution.

4.1 Amir et al.'s Framework

We start with some definitions.

Definition 8 (Bridges)**.** *Let i and j be two distinct positions in the text T with $i < j$, and let $h = \mathsf{lcp}(i,j)$ and $h > 0$. Then, we call the tuple (i,j,h) a bridge. Moreover, we call h its height, i its left leg and j its right leg, and $\mathsf{LCP}(\mathsf{T}[i,n],\mathsf{T}[j,n])$ its label.*

Let $\mathcal{B}_{all}$ be the set of all such bridges. Then clearly,

$$\mathsf{rlcp}(\alpha,\beta) = \max\left(\{h \mid (i,j,h) \in \mathcal{B}_{all} \textbf{ and } i,j \in [\alpha,\beta]\} \cup \{0\}\right)$$

Therefore, by mapping each bridge $(i,j,h) \in \mathcal{B}_{all}$ to a 2D point (i,j) with weight h, the problem can be reduced to a 2D-RMQ problem (refer to Section 2.5). This yields an $O(|\mathcal{B}_{all}| \log^{\varepsilon} n)$ space data structure with query time $O(\log\log n)$. Unfortunately, this is not a space efficient approach as the size of $\mathcal{B}_{all}$ is $\Theta(n^2)$ in the worst case. To circumvent this, Amir et al. [20] introduced the concept of optimal bridges.

Definition 9 (Optimal Bridges)**.** *A bridge $(i,j,h) \in \mathcal{B}_{all}$ is optimal if there exists no other bridge (i',j',h'), such that $i',j' \in [i,j]$ and $h' \geq h$.*

Let $\mathcal{B}_{opt}$ be the set of all optimal bridges. Then, it is easy to observe that

$$\mathsf{rlcp}(\alpha,\beta) = \max\{h \mid (i,j,h) \in \mathcal{B}_{opt} \textbf{ and } i,j \in [\alpha,\beta]\}.$$

Thus, to answer an rlcp query, it is sufficient to examine the bridges in $\mathcal{B}_{opt}$, instead of all the bridges in $\mathcal{B}_{all}$. The crux of Amir et al.'s [20] data structure is the following lemma.

Lemma 13 ([20])**.** *The size of $\mathcal{B}_{opt}$ is $O(n\log n)$.*

Therefore, by applying the above reduction (from Range-LCP to 2D-RMQ) on the bridges in $\mathcal{B}_{opt}$, they got an $O(|\mathcal{B}_{opt}|\log^{\varepsilon} n) = O(n\log^{1+\varepsilon} n)$ space data structure with query time $O(\log\log n)$. Additionally, they showed that there exist cases where the the size of $\mathcal{B}_{opt}$ is $\Omega(n\log n)$. For example, when T is a Fibonacci word (see Section 4 in [20] for its definition). This means that the bound on the number of optimal bridges is tight.

4.2 Our Framework

We first construct a data structure for solving the special case of queries: $\mathsf{rlcp}(\alpha,\beta) = 0$. This means all symbols in $\mathsf{T}[\alpha,\beta]$ are distinct. To solve queries for such ranges, we construct an integer array $P[1,n]$ so that $P[i] = \max(\{j \mid j < i \textbf{ and } T[j] = T[i]\} \cup \{-\infty\})$. That is, $P[i]$ is the position of the predecessor of $T[i]$. Then it holds

$$\mathsf{rlcp}(\alpha,\beta) = 0 \iff \max\{P[i] | i \in [\alpha,\beta]\} < \alpha.$$

Therefore we can solve the query in constant time using a linear space data structure for range maximum query on P. Below we focus on solving queries for the case $\mathsf{rlcp}(\alpha,\beta) > 0$.

We present a replacement for optimal bridges, called *special bridges*. Special bridges are defined

from the standpoint of Heavy Path Decomposition.

Definition 10 (Special Bridges)**.** *Let u_i, u_j in ST be the children of $\mathsf{lca}(\ell_{\mathsf{ISA}[i]}, \ell_{\mathsf{ISA}[j]})$ on the path to $\ell_{\mathsf{ISA}[i]}, \ell_{\mathsf{ISA}[j]}$, respectively. A bridge $(i,j,h) \in \mathcal{B}_{all}$ is special if it satisfies at least one of two following conditions.*

1. *u_i is a light node **and** $j = \min\{x \mid (i,x,h) \in \mathcal{B}_{all}\}$*
2. *u_j is a light node **and** $i = \max\{x \mid (x,j,h) \in \mathcal{B}_{all}\}$*

Let $\mathcal{B}_{spe}$ be the set of all special bridges. If a bridge (i,j,h) is optimal, $i = \max\{x \mid (x,j,h) \in \mathcal{B}_{all}\}$ and $j = \min\{x \mid (i,x,h) \in \mathcal{B}_{all}\}$. Moreover, $u_i \neq u_j$ because $h = |\mathsf{LCP}(\mathsf{T}[i,n], \mathsf{T}[j,n])| = |\mathsf{path}(\mathsf{lca}(\ell_{\mathsf{ISA}[i]}, \ell_{\mathsf{ISA}[j]}))|$. Because at least one of u_i, u_j is a light node, (i,j,h) is a special bridge. Thus, $\mathcal{B}_{opt} \subseteq \mathcal{B}_{spe}$, therefore it holds

$$\mathsf{rlcp}(\alpha,\beta) = \max\{h \mid (i,j,h) \in \mathcal{B}_{spe} \textbf{ and } i,j \in [\alpha,\beta]\}.$$

From Lemma 14, $|\mathcal{B}_{spe}| = \Theta(|\mathcal{B}_{opt}|)$, the same space-time trade-off as in Amir et al. [20] can be obtained by employing special bridges instead of optimal bridges. However, the main advantage over optimal bridges is that special bridges can be encoded efficiently, in $O(1)$-bits per bridge.

Lemma 14. *The size of $\mathcal{B}_{spe}$ is $O(n \log n)$.*

Proof. From Lemma 1, the number of light ancestors for any leaf is at most $\log_2 n$. Moreover, for each pair consisting of a leaf and its light ancestor, there exist at most two special bridges. Therefore, the number of special bridges is bounded by $O(n \log n)$.

□

We now present an overview of our solution.

4.2.1 An Overview of Our Data Structure

We start by defining two queries, which are weaker than Range-LCP.

Definition 11. *For a parameter $\Delta = \Theta(\log n)$, a query $\mathcal{E}_\Delta(\alpha,\beta)$ asks to return an estimate τ of $\mathsf{rlcp}(\alpha,\beta) > 0$, such that*

$$\tau \leq \mathsf{rlcp}(\alpha,\beta) < \tau + \Delta$$

Definition 12. *A query $\mathcal{Q}(\alpha,\beta,h)$ asks to return* **YES** *if there exists special bridge (i,j,h), such that $i,j \in [\alpha,\beta]$ and $h > 0$. Otherwise, $\mathcal{Q}(\alpha,\beta,h)$ returns* **NO**.

Definition 13. *A query $\mathcal{Q}(\alpha,\beta,h)$ asks to return* **YES** *if there exists special bridge (i,j,h), such that $i,j \in [\alpha,\beta]$ and $h > 0$. Otherwise, $\mathcal{Q}(\alpha,\beta,h)$ returns* **NO**.

The following two are the main components of our data structure.

1. A linear space structure for $\mathcal{E}_\Delta(\cdot,\cdot)$ queries in $O(\log^{1+\varepsilon} n)$ time.
2. A linear space structure for $Q(\cdot,\cdot,\cdot)$ queries in $O(\log^{\varepsilon} n)$ time.

Our algorithm for computing $\mathsf{rlcp}(\alpha,\beta)$ is straightforward. First we check if $\mathsf{rlcp}(\alpha,\beta) = 0$ by using the range maximum data structure for P. If it is, we are done. Otherwise we obtain $\tau = \mathcal{E}_\Delta(\alpha,\beta)$. Then, for $h = \tau, \tau+1, \tau+2, ..., \tau+\Delta-1$, compute $\mathcal{Q}(\alpha,\beta,h)$. Then report

$$\mathsf{rlcp}(\alpha,\beta) = \max\{h \mid h \in [\tau, \tau+\Delta-1] \textbf{ and } \mathcal{Q}(\alpha,\beta,h) = \textbf{ YES }\}$$

The time complexity is $O(\log^{1+\varepsilon} n + \Delta \cdot \log^{\varepsilon} n) = O(\log^{1+\varepsilon} n)$ and the space complexity is $O(n)$, as claimed. In what follows, we present the details of these two components of our data structure.

4.3 Details of the Components

We maintain the suffix tree ST of T and the linear space data structure for various 2D range successor/predecessor queries (in $O(\log^{\varepsilon} n)$ time [101]) over the following set of n points.

$$\mathcal{P} = \{(i, \mathsf{SA}[i]) \mid i \in [1, n]\}$$

We rely on this structure for computing interval-LCP and left-leg/right-leg queries (to be defined next).

Lemma 15. *We can answer an interval-LCP query* $\mathsf{ilcp}(p, \alpha, \beta)$ *in time* $O(\log^{\varepsilon} n)$.

Proof. Find the leaf $\ell_{\mathsf{ISA}[p]}$ first. Then find the rightmost leaf ℓ_x before $\ell_{\mathsf{ISA}[p]}$ and the leftmost leaf ℓ_y after $\ell_{\mathsf{ISA}[p]}$, such that $\mathsf{SA}[x], \mathsf{SA}[y] \in [\alpha, \beta]$. We can rely on the following queries for this:

$$x = \mathsf{ORQ}([-\infty, p-1], [\alpha, \beta]) \textbf{ and } y = \mathsf{ORQ}([p+1, +\infty], [\alpha, \beta])$$

Clearly, $\mathsf{ilcp}(p, \alpha, \beta)$ is given by $\max\{\mathsf{lcp}(p, x), \mathsf{lcp}(p, y)\}$. This completes the proof.

□

Definition 14. *For* $i, j \in \mathbb{Z}, i, j \geq 0$ *and* $h \in \mathbb{Z}, h > 0$*, if there exists a special bridge* $(i, \cdot, h) \in \mathcal{B}_{spe}$*, we define*

$$\mathsf{rightLeg}(i, h) = \min\{x \mid (i, x, h) \in \mathcal{B}_{spe}\},$$

otherwise

$$\mathsf{rightLeg}(i, h) = +\infty.$$

Similarly, if there exists a special bridge $(\cdot, j, h) \in \mathcal{B}_{spe}$,

$$\mathsf{leftLeg}(j,h) = \max\{x \mid (x,j,h) \in \mathcal{B}_{spe}\},$$

otherwise

$$\mathsf{leftLeg}(j,h) = -\infty.$$

Note that special bridges $(i,\cdot,h) \in \mathcal{B}_{spe}$ and $(\cdot,j,h) \in \mathcal{B}_{spe}$ may not be unique and leftLeg, rightLeg represent the closest left leg and the closest right leg, respectively.

Lemma 16. *By maintaining a linear space data structure, we can answer* $\mathsf{rightLeg}(k,h)$ *and* $\mathsf{leftLeg}(k,h)$ *queries in* $O(\log^{\varepsilon} n)$ *time.*

Proof. Find the ancestor u (if it exists) of $\ell_{\mathsf{ISA}[k]}$, such that $|\mathsf{path}(u)| = h$ via a weighted level ancestor query (see Section 2.1). If u does not exist, then $\mathsf{rightLeg}(k,h) = +\infty$ and $\mathsf{leftLeg}(k,h) = -\infty$. Otherwise, let u' be the child of u, such that $\ell_{\mathsf{ISA}[k]}$ is under u'. Also, let $[x,y]$ and $[x',y']$ be the range of leaves under u and u', respectively. Then compute

$$p = \min(\mathsf{ORQ}([x,y]\backslash[x',y'],[k+1,+\infty]),+\infty),$$

$$q = \max(\mathsf{ORQ}([x,y]\backslash[x',y'],[-\infty,k-1]),-\infty).$$

If $p = +\infty$, $\mathsf{rightLeg}(k,h) = +\infty$. Otherwise, let $[x_p,y_p]$ be the range of leaves under the child of u that is over $\ell_{\mathsf{ISA}[p]}$. If u' is a heavy node and $\mathsf{ORQ}([x,y]\backslash[x_p,y_p],[-\infty,p-1]) \neq k$, $\mathsf{rightLeg}(k,h) = +\infty$. Otherwise, $\mathsf{rightLeg}(k,h) = p$. Similarly, If $q = -\infty$, $\mathsf{leftLeg}(k,h) = -\infty$. Otherwise, let $[x_q,y_q]$ be the range of leaves under the child of u that is over $\ell_{\mathsf{ISA}[q]}$. If u' is a heavy node and $\mathsf{ORQ}([x,y]\backslash[x_q,y_q],[q+1,+\infty]) \neq k$, $\mathsf{leftLeg}(k,h) = -\infty$. Otherwise, $\mathsf{leftLeg}(k,h) = q$. This completes the proof.

□

The structures described in Lemma 15 and Lemma 16 are the building blocks of our main components, to be described next. The following observation is exploited in both.

Lemma 17. *Suppose that* $(i,j,h) \in \mathcal{B}_{spe}$*. Then,* $\forall k \in [1, h-1]$*, there exists at least one of* $(i+k, \cdot, h-k) \in \mathcal{B}_{spe}$ *such that* $\mathsf{rightLeg}(i+k, h-k) \in (i+k, j+k]$ *and* $(\cdot, j+k, h-k) \in \mathcal{B}_{spe}$ *such that* $\mathsf{leftLeg}(j+k, h-k) \in [i+k, j+k)$.

Proof. Given $\mathsf{lcp}(i,j) = h$, we have $\mathsf{lcp}(i+k, j+k) = (h-k)$. Let u'_{i+k}, u'_{j+k} be the child of $u = \mathsf{lca}(\ell_{\mathsf{ISA}[i+k]}, \ell_{\mathsf{ISA}[j+k]})$, such that $\ell_{\mathsf{ISA}[i+k]}, \ell_{\mathsf{ISA}[i+k]}$ is under u'_{i+k}, u'_{j+k}, respectively. Then, at least one of u'_{i+k} and u'_{j+k} is a light node. From the definition of a special bridge, there exists at least one of $(i+k, \cdot, h-k) \in \mathcal{B}_{spe}$ and $(\cdot, j+k, h-k) \in \mathcal{B}_{spe}$. From $\mathsf{lcp}(i+k, j+k) = (h-k)$, if there exists $(i+k, \cdot, h-k) \in \mathcal{B}_{spe}$, $\mathsf{rightLeg}(i+k, h-k) \leq j+k$. Similarly, if there exists $(\cdot, j+k, h-k) \in \mathcal{B}_{spe}$, $\mathsf{leftLeg}(j+k, h-k) \geq i+k$. This completes the proof.

□

4.3.1 The Structure for Estimating Range-LCP

Let $\mathcal{B}_t$ denotes the set of all special bridges with height t. Also, for $f = 0, 1, 2, \ldots, (\Delta - 1)$, where $\Delta = \Theta(\log n)$, define $\mathcal{C}_f$ to be the set of all special bridges with its height divided by Δ leaving remainder f. Specifically,

$$\mathcal{C}_f = \bigcup_{k=0}^{\left\lfloor \frac{n-f}{\Delta} \right\rfloor} \mathcal{B}_{(f+k\Delta)}$$

Let $\mathcal{C}_\pi : \pi \in [0, \Delta - 1]$ be the smallest set among all $\mathcal{C}_f$'s. Its size can be bounded by $O((n \log n)/\Delta)$ (by pigeonhole principle), which is $O(n)$. Also $|\mathcal{B}_1|$ can be bounded by $O(n)$. We map each special

bridge $(i,j,h) \in \mathcal{C}_\pi \cup \mathcal{B}_1$ into a 2D point (i,j) with weight h and maintain the linear-space data structure over them for answering 2D-RMQ. We use the linear-space structure by Chazelle [38]. The space is $|\mathcal{C}_\pi \cup \mathcal{B}_1| = O(n)$ words and the query time is $O(\log^{1+\varepsilon} n)$.

Our Algorithm

Let (α^*, β^*, h^*) be the tallest special bridge, such that both $\alpha^*, \beta^* \in [\alpha, \beta]$.

Two possible scenarios are

1. $\beta^* \in (\alpha, \beta - \Delta]$: We query on the 2D-RMQ structure over $\mathcal{C}_\pi \cup \mathcal{B}_1$ and find the tallest bridge $(i', j', h') \in \mathcal{C}_\pi \cup \mathcal{B}_1$, such that $i', j' \in [\alpha, \beta]$ and $h' > 0$. We claim that $h^* \in [h', h' + \Delta)$. Proof follows from Lemma 17.

2. $\beta^* \in (\beta - \Delta, \beta]$: We can rely on Interval-LCP queries. Specifically, $h^* = \max\{\mathsf{ilcp}(p, \alpha, \beta) \mid p \in (\beta - \Delta, \beta]\}$.

Because we do not know which case occurs, we try both. Then we have

$$\mathcal{E}_\Delta(\alpha, \beta) = \max\Big(\{\mathsf{ilcp}(p, \alpha, \beta) \mid p \in (\beta - \Delta, \beta]\} \cup \{h'\}\Big).$$

Note that if $h^* > 0$, $\mathcal{E}_\Delta(\alpha, \beta)$ always returns a positive value.

The time complexity is proportional to that of one 2D-RMQ and at most Δ number of Interval-LCP queries. That is, $O(\log^{1+\varepsilon} n + \Delta \cdot \log^{\varepsilon} n) = O(\log^{1+\varepsilon} n)$ time.

4.3.2 The Structure for Handling $\mathcal{Q}(\alpha,\beta,h)$ Queries

Recall that $\mathcal{B}_t$ is the set of all special bridges with height t. Let L_t^1 represent the sorted list of left-legs of all bridges in $\mathcal{B}_t$ in the form of a y-fast trie for fast predecessor/successor search. Also, let R_t^1 be another array, such that $R_t^1[k] = \mathsf{rightLeg}(L_t^1[k],t)$. Let R_t^2 represent the sorted list of right-legs of all bridges in $\mathcal{B}_t$ in the form of a y-fast trie for fast predecessor search. Also, let L_t^2 be another array, such that $L_t^2[k] = \mathsf{leftLeg}(R_t^2[k],t)$.

Also, let

$$\mathcal{S}_\pi = \{1, \pi, \pi+\Delta, \pi+2\Delta, \pi+3\Delta, ..., (\pi + \lfloor (n-\pi)/\Delta \rfloor \Delta)\}.$$

For each $t \in [1,n]$, we maintain a separate structure that can answer queries of the type $\mathcal{Q}(\cdot,\cdot,t)$. Based on whether h in the query $\mathcal{Q}(\alpha,\beta,h)$ is in $\mathcal{S}_\pi$ or not, we have two cases.

Case 1: $h \in \mathcal{S}_\pi$

To handle this case, we maintain L_t^1 and the succinct data structure for range minimum query (RMQ) on R_t^1 for all $t \in \mathcal{S}_\pi$. The total space is $|\mathcal{C}_\pi \cup \mathcal{B}_1| = O(n)$ words. Therefore, any query $\mathcal{Q}(\alpha,\beta,h)$ with $h \in \mathcal{S}_\pi$ can be answered using the following steps.

1. Find the smallest k, such that $L_h^1[k] \geq \alpha$ via a successor query.
2. Then, find the index k' corresponding to the smallest element in $R_h^1[k, |R_h^1|]$ using a range minimum query. Note that R_h^1 is not stored.
3. Then, report "YES" if $\mathsf{rightLeg}(L_h^1[k'],h) \leq \beta$, and report "NO" otherwise.

The time complexity is $(\log\log n + \log^\varepsilon n) = O(\log^\varepsilon n)$. The correctness can be easily verified.

Case 2: $h \notin \mathcal{S}_\pi$

We first show how to design a structure for a predefined h. Let $q = \mathsf{pred}(h, S_\pi) = \pi + \Delta \cdot \lfloor (h-\pi)/\Delta \rfloor$ if $h \geq \pi$, otherwise $q = 1$. Also, let $z = (h-q)$. Note that for each special bridge (i, j, h), there exists at least one of special bridges $(i+z, \cdot, h-z) = (i+z, \cdot, q)$ and $(\cdot, j+z, h-z) = (\cdot, j+z, q)$.

Now, define arrays $R_h^{1'}$ and $L_h^{2'}$ of length $|\mathcal{B}_q|$, such that for any $k \in [1, |\mathcal{B}_q|]$, $R_h^{1'}[k] = \mathsf{rightLeg}((L_q^1[k] - z), h)$ and $L_h^{2'}[k] = \mathsf{leftLeg}((R_q^2[k] - z), h)$. Our data structures are a succinct range minimum query (RMQ) structure over $R_h^{1'}$ and a succinct range maximum query structure over $L_h^{2'}$. We now show how to answer a $Q(\alpha, \beta, h)$ query using $R_h^{1'}, L_h^{2'}, L_q^1$ and R_q^2. The steps are as follows.

1. Find the smallest k, such that $(L_q^1[k] - z) \geq \alpha$. We perform a successor query on L_q^1 for this.
2. Then, find the index k' corresponding to the smallest element in $R_h^{1'}[k, |R_q^1|]$ using a range minimum query.
3. Then, report "YES" if $\mathsf{rightLeg}(L_q^1[k'] - z, h) \leq \beta$, otherwise continue to the next step.
4. Then, find the largest l, such that $(R_q^2[l] - z) \leq \beta$. We perform a predecessor query on R_q^2 for this.
5. Then, find the index l' corresponding to the largest element in $L_h^{2'}[1, l]$ using a range maximum query.
6. Then, report "YES" if $\mathsf{leftLeg}(R_q^2[l'] - z, h) \geq \alpha$, and report "NO" otherwise.

The time complexity is $(\log\log n + \log^\varepsilon n) = O(\log^\varepsilon n)$. The space complexity for a fixed h is $|\mathcal{B}_q|(4 + o(1))$ bits. Therefore, by maintaining the above structure for all values of h, we can

answer $\mathcal{Q}(\alpha,\beta,h)$ for any α,β and h in $O(\log^{\varepsilon} n)$ time. Total space (in bits) is:

$$(4+o(1))\sum_{h=1}^{n}|\mathcal{B}_{\pi+\Delta\cdot\lfloor(h-\pi)/\Delta\rfloor}| = (4+o(1))\Delta\sum_{q\in\mathcal{S}_\pi}|\mathcal{B}_q| = O(n\log n).$$

4.4 Conclusion

In this chapter, we revisited the Range Longest Common Prefix Problem which is initially introduced by Amir et al. [20]. They proposed an $O(n\log^{1+\varepsilon} n)$ space structure with query time $O(\log\log n)$, and a linear space (i.e., $O(n)$ words) structure with query time $O(\delta\log\log n)$, where $\delta=\beta-\alpha+1$ is the length of the input range and $\varepsilon>0$ is an arbitrarily small constant. Our data structure can answer the Range LCP queries in poly-logarithmic time using a linear space data. We showed, any Range-LCP query on the text $\mathsf{T}[1,n]$ can be answered in $O(\log^{1+\varepsilon} n)$ time using a linear space data structure. We remark that our data structure can be constructed in $O(n\log n)$ time.

CHAPTER 5: THE RANGE SHORTEST UNIQUE SUBSTRING PROBLEM

Given string T as input, the *Shortest Unique Substring* problem is to find a shortest substring of T that does not occur elsewhere in T. In this chapter, we introduce the range variant of this problem, which we call the *Range Shortest Unique Substring* problem. The task is to construct a data structure over T answering the following type of online queries efficiently. Given a range $[\alpha,\beta]$, return a shortest substring $\mathsf{T}[i,j]$ of T with exactly one occurrence in $[\alpha,\beta]$. In this chapter, we present our framework for solving the *Range Shortest Unique Substring* Problem. In Section 5.1, we propose our $\mathcal{O}(n\log n)$-word data structure with $\mathcal{O}(\log_w n)$ query time, where $w=\Omega(\log n)$ is the word size. Additionally, in Section 5.2, we present an $\mathcal{O}(n)$-word data structure with $\mathcal{O}(\sqrt{n}\log^{\varepsilon} n)$ query time, where $\varepsilon>0$ is an arbitrarily small constant.

5.1 An $\mathcal{O}(n\log n)$-Word Data Structure

Our construction is based on ingredients such as the suffix tree [121], heavy-light decomposition [112], and a geometric data structure for rectangle stabbing [37]. Let us start with some definitions.

Definition 15. *For a position $k\in[1,n]$ and $h\geq 1$, we define* $\mathsf{Prev}(k,h)$ *and* $\mathsf{Next}(k,h)$ *as follows:*

$$\mathsf{Prev}(k,h)=\max_j\{\{j<k \mid \mathsf{T}[k,k+h-1]=\mathsf{T}[j,j+h-1]\}\cup\{-\infty\}\}$$

$$\mathsf{Next}(k,h)=\min_j\{\{j>k \mid \mathsf{T}[k,k+h-1]=\mathsf{T}[j,j+h-1]\}\cup\{+\infty\}\}.$$

Intuitively, let x and y be the occurrences of $\mathsf{T}[k,k+h-1]$ right before and right after the position k,

respectively. Then, $\mathsf{Prev}(k,h) = x$ and $\mathsf{Next}(k,h) = y$. If x (resp., y) does not exist, then $\mathsf{Prev}(k,h) = -\infty$ (resp., $\mathsf{Next}(k,h) = +\infty$).

Definition 16. *Let $k \in [a,b]$. We define $\lambda(a,b,k)$ as follows:*

$$\lambda(a,b,k) = \min\{h \mid \mathsf{Prev}(k,h) < a \ \textbf{\textit{and}} \ \mathsf{Next}(k,h) > b\}.$$

Intuitively, $\lambda(a,b,k)$ denotes the length of the shortest substring that starts at position k with exactly one occurrence in $[a,b]$.

Definition 17. *For a position $k \in [1,n]$, we define C_k as follows:*

$$C_k = \{h > 1 \mid (\mathsf{Next}(k,h), \mathsf{Prev}(k,h)) \neq (\mathsf{Next}(k,h-1), \mathsf{Prev}(k,h-1))\} \cup \{1\}$$

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Example 2. (Running Example for Definition 17) Let $\mathsf{T} = \mathbf{caabcaddaacaddaaaabac}$ and $k = 10$. We have that $(\mathsf{Next}(10,1), \mathsf{Prev}(10,1)) = (12,9)$, $(\mathsf{Next}(10,2), \mathsf{Prev}(10,2)) = (20,-\infty)$, and $(\mathsf{Next}(10,3), \mathsf{Prev}(10,3)) = (+\infty,-\infty)$. Thus, $C_{10} = \{2,3\} \cup \{1\} = \{1,2,3\}$.

Intuitively, C_k stores the set of candidate lengths for shortest unique substrings starting at position k. We make the following observation.

Observation 1. *$\lambda(a,b,k) \in C_k$, for any $1 \leq a \leq b \leq n$.*

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

Example 3. (Running Example for Observation 1) Let $\mathsf{T} = \mathbf{caabcaddaacaddaaaabac}$ and $k = 10$. We have that $C_{10} = \{1,2,3\}$. For $a = 5$ and $b = 16$, $\lambda(5,16,10) = 2$, denoting substring **ac**. For $a = 5$ and $b = 20$, $\lambda(5,20,10) = 3$, denoting substring **aca**.

The following combinatorial lemma is crucial for efficiency.

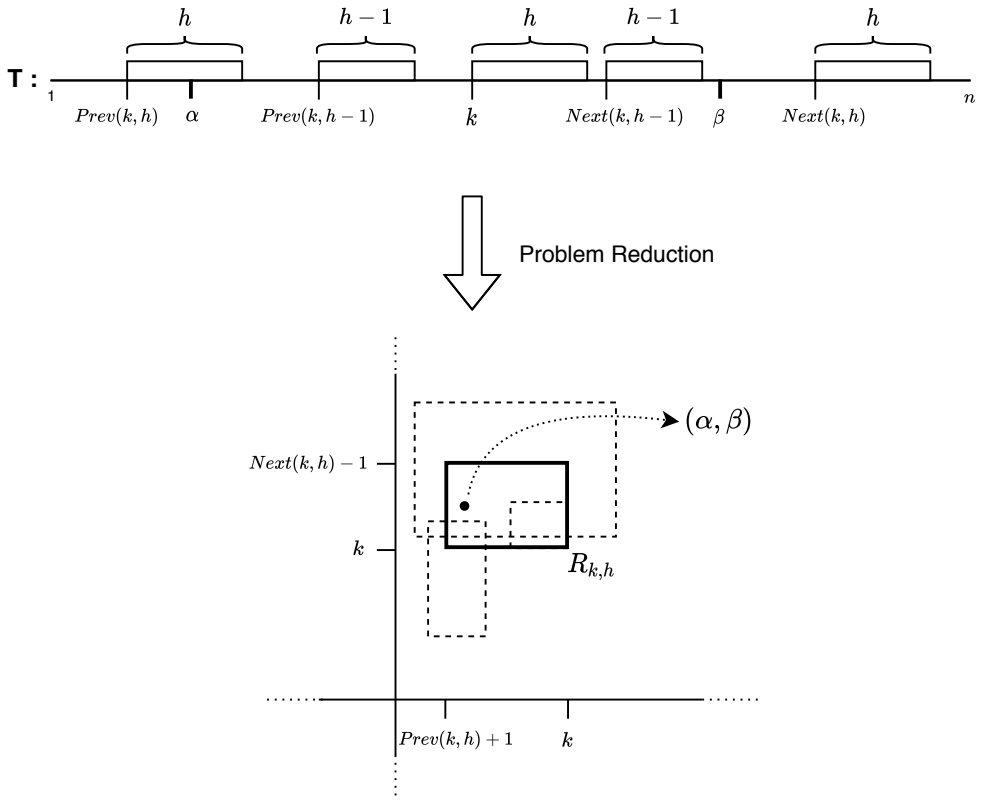

Figure 5.1: Illustration of the problem reduction: (k,h) is the output of the rSUS problem with query range $[\alpha,\beta]$, where $h = \lambda(\alpha,\beta,k) \in C_k$. $R_{k,h}$ is the lowest weighted rectangle in $\mathcal{R}$ containing the point (α,β).

Lemma 18. $\sum_k |C_k| = \mathcal{O}(n \log n)$.

Proof. The proof of Lemma 18 is deferred to Section 5.1. □

We are now ready to present our construction. By Observation 1, for a given query range $[\alpha,\beta]$, the answer (p,ℓ) we are looking for is the pair (k,h) with the minimum h under the following conditions: $k \in [\alpha,\beta]$, $h \in C_k$, $\mathsf{Prev}(k,h) < \alpha$ and $\mathsf{Next}(k,h) > \beta$. Equivalently, (p,ℓ) is the pair (k,h) with the minimum h, such that $h \in C_k$, $\alpha \in (\mathsf{Prev}(k,h),k]$, and $\beta \in [k,\mathsf{Next}(k,h))$. We map

each $h \in C_k$ into a weighted rectangle $R_{k,h}$ with weight h which is defined as follows:

$$R_{k,h} = [\mathsf{Prev}(k,h)+1, k] \times [k, \mathsf{Next}(k,h)-1].$$

Let $\mathcal{R}$ be the set of all such rectangles, then the lowest weighted rectangle in $\mathcal{R}$ stabbed by the point (α,β) is $R_{p,\ell}$. In short, an rSUS query on $\mathsf{T}[1,n]$ with an input range $[\alpha,\beta]$ can be reduced to an equivalent top-1 rectangle stabbing query on a set $\mathcal{R}$ of rectangles with input point (α,β). In the *2-d Top-1 Rectangle Stabbing* problem, we preprocess a set of weighted rectangles in 2-d so that given a query point q the task is to report the largest (or lowest) weighted rectangles containing q [37]. Similarly, here, the task is to report the lowest weighted rectangle in $\mathcal{R}$ containing the point (α,β) (see Figure 5.1 for an illustration). By Lemma 18, we have that $|\mathcal{R}| = \mathcal{O}(n \log n)$. Therefore, by employing the optimal data structure for top-1 rectangle stabbing presented by Chan et al. [37], which takes $\mathcal{O}(|\mathcal{R}|)$-word space supporting $\mathcal{O}(\log_w |\mathcal{R}|)$-time queries, we arrive at the space-time trade-off of Theorem 4. This completes our construction.

Proof of Lemma 18

Let $\mathsf{lcp}(i,j)$ denote the length of the longest common prefix of the suffixes of T starting at positions i and j in T. Also, let S denote the set of all (x,y) pairs, such that $1 \le x < y \le n$ and $\mathsf{lcp}(x,y) > \mathsf{lcp}(x,z)$, for all $z \in [x+1, y-1]$. The proof can be broken down into Lemma 19 and Lemma 20.

Lemma 19. $\sum_k |C_k| = \mathcal{O}(|S|)$.

Proof. Let us fix a position k. Let

$$C'_k = \{h > 1 \mid \mathsf{Prev}(k,h) \neq \mathsf{Prev}(k,h-1)\}$$

$$C''_k = \{h > 1 \mid \mathsf{Next}(k,h) \neq \mathsf{Next}(k,h-1)\}.$$

Clearly we have that $C_k = C'_k \cup C''_k \cup \{1\}$.

The following statements can be deduced by a simple contradiction argument:

1. Let $i = \mathsf{Prev}(k,h) \neq -\infty$, where $h \in C'_k$, then $i = \mathsf{Prev}(k, \mathsf{lcp}(i,k))$
2. Let $j = \mathsf{Next}(k,h) \neq \infty$, where $h \in C''_k$, then $j = \mathsf{Next}(k, \mathsf{lcp}(k,j))$.

Figure 5.2 illustrates the proof for the first statement. The second one can be proved in a similar fashion.

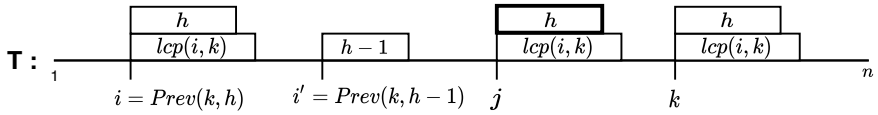

Figure 5.2: Let $h \in C'_k$ and $i = \mathsf{Prev}(k,h)$. By contradiction, assume that there exists $j \in (i,k)$ such that $j = \mathsf{Prev}(k, \mathsf{lcp}(i,k))$. Since $h \leq \mathsf{lcp}(i,k)$, $\mathsf{T}[j, j+h-1] = \mathsf{T}[k, k+h-1]$. This is a contradiction with $i = \mathsf{Prev}(k,h)$. Thus, $i = \mathsf{Prev}(k, \mathsf{lcp}(i,k))$.

Clearly, $|C'_k|$ is proportional to the number of (i,k) pairs such that $\mathsf{lcp}(i,k) \neq 0$ and $i = \mathsf{Prev}(k, \mathsf{lcp}(i,k))$. Similarly, $|C''_k|$ is proportional to the number of (k,j) pairs such that $\mathsf{lcp}(k,j) \neq 0$ and $j = \mathsf{Next}(k, \mathsf{lcp}(k,j))$. Therefore, $\sum_k |C_k|$ is proportional to the number of (x,y) pairs, such that $\mathsf{lcp}(x,y) \neq 0$ and $\mathsf{lcp}(x,y) > \mathsf{lcp}(x,z)$, for all $z \in [x+1, y-1]$. This completes the proof of Lemma 19. □

Lemma 20. $|S| = \mathcal{O}(n \log n)$.

Proof. Consider the *suffix tree* data structure of string $\mathsf{T}[1,n]$, which is a compact trie of the n suffixes of T appended with a letter $\$ \notin \Sigma$ [121]. This suffix tree consists of $n+1$ leaves (one for each suffix of T) and at most n internal nodes. The edges are labeled with substrings of T. Let u be the lowest common ancestor of the leaves corresponding to the strings $\mathsf{T}[x,n]\$$ and $\mathsf{T}[y,n]\$$. Then,

the concatenation of the edge labels on the path from the root to u is exactly the longest common prefix of $\mathsf{T}[x,n]\$$ and $\mathsf{T}[y,n]\$$. For any node u, we denote by $\mathsf{size}(u)$ the total number of leaf nodes of the subtree rooted at u.

We decompose the nodes in the suffix tree into *light* and *heavy* nodes. The root node is light and for any internal node, exactly one child is heavy. Specifically, the heavy child is the one having the largest number of leaves in its subtree (ties are broken arbitrarily). All other children are light. This tree decomposition is known as *heavy-light decomposition*. We have the following critical observation. Any path from the root to a leaf node contains many nodes, however, the number of light nodes is at most $\log n$ [112, 69]. Also, corresponding to the $n+1$ leaves of the suffix tree, there are $n+1$ paths from the root to the leaves. Therefore, the sum of subtree sizes over all light nodes is $\mathcal{O}(n \log n)$.

We are now ready to complete the proof. Let $S_u \subseteq S$ denote the set of pairs (x,y), such that the lowest common ancestor of the leaves corresponding to suffixes $\mathsf{T}[x,n]\$$ and $\mathsf{T}[y,n]\$$ is u. Clearly, the paths from the root to the leaves corresponding to suffixes $\mathsf{T}[x,n]\$$ and $\mathsf{T}[y,n]\$$ pass from two distinct children of node u and then at least one of the two must be a light node. There are two cases. In the first case, both leaves are under the light children. In the second case, one leaf is under a light child and the other is under the heavy child. In both cases, we have at least one leaf under a light node. If we fix the leaf which is under the light node, we can enumerate the total number of pairs based on the subtree size of the light nodes. Therefore, $|S_u|$ is at most twice the sum of $\mathsf{size}(\cdot)$ over all light children of u. Since $|S| = \sum_u |S_u|$, we can bound $|S|$ by the sum of $\mathsf{size}(\cdot)$ over all light nodes in the suffix tree, which is $\mathcal{O}(n \log n)$. This completes the proof of Lemma 20. $\square$

5.2 An $\mathcal{O}(n)$-Word Data Structure

This section is dedicated to proving Theorem 5. For simplicity, we focus only on the computation of the length ℓ of the output (p,ℓ).

Let SA be the *suffix array* of string T of length n, which is a permutation of $\{1,\cdots,n\}$, such that $\mathsf{SA}[i]=j$ if $\mathsf{T}[j,n]$ is the ith lexicographically smallest suffix of T [92]. Further let SA^{-1} be the *inverse suffix array* of string T of length n, which is a permutation of $\{1,\cdots,n\}$, such that $\mathsf{SA}^{-1}[\mathsf{SA}[i]]=i$. Moreover, SA of T can be constructed in linear time and space [46, 79].

We observe that an $\mathcal{O}(\beta-\alpha+1)$-time solution is straightforward with the aid of the suffix tree of T as follows. First, identify those leaves corresponding to the suffixes starting within $[\alpha,\beta]$ using the inverse suffix array of T and mark them. Then, for each marked leaf, identify its lowest ancestor node (and double mark it), such that a marked neighbor is also under it. This can be done via at most two $\mathcal{O}(1)$-time Lowest Common Ancestor (LCA) queries over the suffix tree of T using $\mathcal{O}(n)$ additional space [29]. Then, find the minimum over the string-depth of all double-marked nodes, add 1 to it, and report it as the length ℓ. The correctness is readily verified.

We employ the above procedure when $\beta-\alpha+1<3\Delta$, where Δ is a parameter to be set later. We now consider the case when $\beta-\alpha+1\geq 3\Delta$. Note that ℓ is the smallest element in $S^*=\{\lambda(\alpha,\beta,k)\mid k\in[\alpha,\beta]\}$. Let α' be the smallest number after α and β' be the largest number before β such that α' and β' are multiples of Δ. Then, S^* can be written as the union of $S'=\{\lambda(\alpha,\beta,k)\mid k\in[\alpha,\alpha'-1]\cup[\beta'+1,\beta]\}$ and $S''=\{\lambda(\alpha,\beta,k)\mid k\in[\alpha',\beta']\}$. Furthermore, S'' can be written as $S''_1\cup S''_2\cup S''_3$, where

- $S''_1=\{h=\lambda(\alpha,\beta,k)\mid k\in[\alpha',\beta'],\mathsf{Prev}(k,h)\in[\alpha'-\Delta,\alpha-1]\}$
- $S''_2=\{h=\lambda(\alpha,\beta,k)\mid k\in[\alpha',\beta'],\mathsf{Next}(k,h)\in[\beta+1,\beta'+\Delta]\}$

- $S_3'' = \{h = \lambda(\alpha,\beta,k) \mid k \in [\alpha',\beta'], \mathsf{Prev}(k,h) \in (-\infty, \alpha' - \Delta - 1], \mathsf{Next}(k,h) \in [\beta' + \Delta + 1, \infty)\}$.

Our algorithm is based on a solution to the *Orthogonal Range Predecessor/Successor in 2-d* problem. A set $\mathcal{P}$ of n points in an $[1,n] \times [1,n]$ grid can be preprocessed into a linear-space data structure, such that the following queries can be answered in $\mathcal{O}(\log^{\varepsilon} n)$ time per query [101]:

- $\mathsf{ORQ}([x',x''],[-\infty,y'']) = \arg\max_j\{(i,j) \in \mathcal{P} \cap [x',x''] \times [-\infty,y'']\}$
- $\mathsf{ORQ}([-\infty,x''],[y',y'']) = \arg\max_i\{(i,j) \in \mathcal{P} \cap [-\infty,x''] \times [y',y'']\}$
- $\mathsf{ORQ}([x',x''],[y',+\infty]) = \arg\min_j\{(i,j) \in \mathcal{P} \cap [x',x''] \times [y',+\infty]\}$
- $\mathsf{ORQ}([x',+\infty],[y',y'']) = \arg\min_i\{(i,j) \in \mathcal{P} \cap [x',+\infty] \times [y',y'']\}$.

We next show how to maintain additional structures, so that the smallest element in each of the above sets can be efficiently computed and thus the smallest among them can be reported as ℓ.

- **Computing the Smallest Element in S':** For each $k \in [\alpha, \alpha' - 1] \cup [\beta' + 1, \beta]$, we compute $\lambda(\alpha,\beta,k)$ and report the smallest among them. We handle each $\lambda(\alpha,\beta,k)$ query in time $\mathcal{O}(\log^{\varepsilon} n)$ as follows: first find the leaf corresponding to the string position k in the suffix tree of T, then the last (resp., first) leaf on its left (resp., right) side, such that the string position x (resp., y) corresponding to it is in $[\alpha,\beta]$, and report $1 + \max\{\mathsf{lcp}(k,x), \mathsf{lcp}(k,y)\}$. To enable the computation of x (resp., y) efficiently, we preprocess the suffix array into an $\mathcal{O}(n)$-word data structure that can answer orthogonal range predecessor (resp., successor) queries in $\mathcal{O}(\log^{\varepsilon} n)$ time [101].
- **Computing the Smallest Element in S_1'':** For each $r \in [\alpha' - \Delta, \alpha - 1]$, we compute the smallest element in $\{h = \lambda(\alpha,\beta,k) \mid k \in [\alpha',\beta'], \mathsf{Prev}(k,h) = r\}$ and report the smallest

among them. The procedure is the following: find the leaf corresponding to the string position r in the suffix tree of T and the last (resp., first) leaf on its left (resp., right) side, such that its corresponding string position x (resp., y) is in $[\alpha', \beta']$ (via orthogonal range successor/predecessor queries as earlier). Then, $t = \max\{\mathsf{lcp}(r,x), \mathsf{lcp}(r,y)\}$ is the length of the longest prefix of $\mathsf{T}[r,n]$ with an occurrence d in $[\alpha', \beta']$. However, we need to verify if occurrence d is unique and its $\mathsf{Prev}(d,t) = r$. For this, find the two leftmost occurrences of $\mathsf{T}[r, r+t-1]$ after r, denoted by x' and y' ($x' < y'$), via two orthogonal range successor queries. If y' does not exist, set $y' = +\infty$. Then report $\lambda(\alpha, \beta, d)$ if $\alpha' \leq x' \leq \beta' < y'$. Otherwise, report $+\infty$.

- **Computing the Smallest Element in S''_2:** For each $r \in [\beta+1, \beta'+\Delta]$, we compute the smallest element in $\{h = \lambda(\alpha, \beta, k) \mid k \in [\alpha', \beta'], \mathsf{Next}(k,h) = r\}$ and report the smallest among them. The procedure is analogous to that of S''_1; i.e., find the length t of the longest prefix of $\mathsf{T}[r,n]$ with an occurrence d in $[\alpha', \beta']$. Then, find the two rightmost occurrences of $\mathsf{T}[r, r+t-1]$ before r, denoted by x' and y' ($x' < y'$), via two orthogonal range successor queries. If x' does not exist, set $x' = +\infty$. Then report $\lambda(\alpha, \beta, d)$ if $x' < \alpha' \leq y' \leq \beta'$. Otherwise, report $+\infty$.

- **Computing the Smallest Element in S''_3:** The set S''_3 can be written as $\{\lambda(\alpha'-\Delta, \beta'+\Delta, k) \mid k \in [\alpha', \beta']\}$, which is now dependent only on α', β' and Δ. Therefore, our idea is to precompute and explicitly store the minimum element in $\{\lambda(a-\Delta, b+\Delta, k) \mid k \in [a,b]\}$ for all (a,b) pairs, where both a and b are multiples of Δ, and for that the desired answer can be retrieved in constant time. The additional space needed is $\mathcal{O}((n/\Delta)^2)$.

We set $\Delta = \lfloor \sqrt{n} \rfloor$. The total space is then $\mathcal{O}(n)$ and the total time is $\mathcal{O}(\Delta \log^{\varepsilon} n) = \mathcal{O}(\sqrt{n} \log^{\varepsilon} n)$. We therefore arrive at Theorem 5.

5.3 Final Remarks

We introduced the Range Shortest Unique Substring (rSUS) problem, the range variant of the Shortest Unique Substring problem. We presented an $\mathcal{O}(n \log n)$-word data structure with $\mathcal{O}(\log_w n)$ query time, where $w = \Omega(\log n)$ is the word size, for this problem. We also presented an $\mathcal{O}(n)$-word data structure with $\mathcal{O}(\sqrt{n} \log^{\varepsilon} n)$ query time, where $\varepsilon > 0$ is an arbitrarily small constant.

We leave the following related questions unanswered:

1. Can we design an $\mathcal{O}(n)$-word data structure for the rSUS problem with polylogarithmic query time?
2. Can we design an efficient solution for the k mismatches/edits variation of the rSUS problem, perhaps using the framework of [115, 113, 13]?
3. Can our reduction from Section 5.1 be extended to other types of string regularities, such as shortest absent words [27]?

CHAPTER 6: INDEXING CACHE-OBLIVIOUSLY

In this chapter, we focus on a variation of the text indexing problem, known as the *non-overlapping indexing*, which is central to data compression [24, 40]. Specifically, we study two problems 5 and 6 which are defined in Section 1.4.

6.1 Sorted Range Reporting on Arrays

Definition 18. *Let $A[1,n]$ be an array of n integers. A sorted range reporting query $[s,e]$ on A asks to report the elements in the subarray $A[s,e]$ in their ascending order.*

We now present two useful results.

Lemma 21. *There exists an $O(n\log n)$ space cache-oblivious data structure that can answer sorted range reporting queries on arrays in optimal $O((e-s+1)/B)$ I/O operations.*

Proof. Let $L(t,l)$ be the list of all $(i,A[i])$ pairs with $i \in [t,t+l-1]$ in the ascending order of $A[i]$. Maintain $L(1,l), L(1+l,2l), L(1+2l,3l), \ldots, L(1+l\lfloor n/l \rfloor, n)$ for $l = 2,4,8,\cdots,2^{\lfloor \log n \rfloor}$. The total space is $O(n\log n)$.

To answer a query $[s,e]$, compute $l = 2^{\lfloor \log(e-s+1) \rfloor}$ and $k = l\lfloor e/l \rfloor$. Then, simply merge the two sorted lists $L(k-l+1,k)$ and $L(k+1,k+l)$ and discard those elements that are not in $A[s,e]$. The correctness follows from the fact that $k-l+1 \leq s \leq e \leq k+l$. The number of I/O operations required is $2l/B = O((e-s+1)/B)$.

□

Lemma 22. *There exists an $O(n\log_{M/B}n)$ space cache-aware data structure that can answer sorted range reporting queries on arrays in optimal $O((e-s+1)/B)$ I/O operations.*

Proof. Since M and B are known in advance, maintain $L(1,l),L(1+l,2l),L(1+2l,3l),\ldots,L(1+l\lfloor n/l\rfloor,n)$ with $l=(M/B)^{j/2}$ for $j=0,1,2,\ldots,\lfloor\log_{M/B}n\rfloor$. The total space is $O(n\log_{M/B}n)$.

Given a query $[s,e]$, first compute l and j, such that $t=(M/B)^{j/2}\leq e-s+1\leq(M/B)^{(j+1)/2}$. If t is small enough, say below $(M/B)^{1/2}$, then the entire subarray $A[s,e]$ can be taken to the internal memory and sorted. Otherwise, the query can be answered by merging $O((M/B)^{1/2}\log_{M/B}n)$ sorted lists of total length $\Theta(e-s+1)$. This can be implemented in optimal $O((e-s+1)/B)$ I/O operations.

□

6.2 Non-Overlapping Indexing - Cache Obliviously

Our data structure consists of a (cache-oblivious) suffix tree ST, and a sorted range reporting structure as in Lemma 21 over the suffix array SA. Additional structures will be introduced along the way. We start with a definition.

Definition 19 (Shortest Period)**.** *Let Q be the shortest prefix of P such that P can be written as the concatenation of $\alpha\geq 1$ copies of Q and a (possibly empty) prefix R of Q. i.e., $P=Q^{\alpha}R$. Then, we denote the length of Q by* $\mathsf{period}(P)$.

We say that an input pattern P is periodic if $\mathsf{period}(P)\leq|P|/2$ (equivalently $\alpha\geq 2$), else (i.e., $\alpha=1$), we say P is *aperiodic*. The first step of our query algorithm is to check if P is periodic or not, and we rely on the result in Lemma 23.

Lemma 23. *Given a pattern $P[1,m]$ which appears at least once in T, there exists an algorithm that finds if P is periodic or not in $O(m/B+\log_B n)$ I/O operations using an $O(n\log n)$ space structure. Also, the algorithm returns $\mathsf{period}(P)$ if P is periodic.*

Proof. We start with some terminologies. A substring $\mathsf{T}[i,i+l-1]$ is *right-maximally-periodic* iff $\mathsf{T}[i,i+l-1]$ is periodic and $\mathsf{T}[i,i+l]$ is aperiodic. Let $l_1,l_2,...,l_k$ be the lengths of all substrings starting at i in their ascending order that are *right-maximally-periodic* and let $q_1,q_2,...,q_k$ be their respective $\mathsf{period}(\cdot)$'s. From the definition of periodic and aperiodic, $l_{j-1} \leq q_j$ and $2\cdot q_j \leq l_j$, therefore $2\cdot l_{j-1} \leq l_j$ for all $j \in [2,k]$. Also, $2^{k-1}l_1 \leq l_k \leq n-i+1$ and the number k of *right-maximally-periodic* prefixes of $\mathsf{T}[i,n]$ can be bounded by $1+\log(l_k/l_1) = O(\log n)$. Moreover, any substring $\mathsf{T}[i,i+m-1]$ of length m is periodic (with period q_j) iff $2\cdot q_j \leq m \leq l_j$ for some j.

The proof of Lemma 23 is straightforward from the discussion above. For each $\mathsf{T}[i,n]$, we maintain the lengths and periods of all its right-maximally-periodic prefixes. The space required is $O(n\log n)$ words. When a pattern $P[1,m]$ comes, the algorithm first finds an occurrence i of P in T in $O(m/B+\log_B n)$ I/O operations. Then from the lengths of all right-maximally-periodic prefixes (and their periods) of $\mathsf{T}[i,n]$, it decides if P is periodic or not in $(\log n)/B = O(\log_B n)$ I/O operations. The algorithm also retrieves $\mathsf{period}(P)$, if P is periodic.

□

6.2.1 Handling aperiodic case

When P is aperiodic, $\mathsf{occ} = \Theta(\mathsf{nocc})$. Therefore, obtain all occurrences of P in their ascending order and do the following: report the first occurrence and report any other occurrence iff it is not overlapping with the last reported occurrence. This step can be implemented in $\mathsf{occ}/B = \Theta(\mathsf{nocc}/B)$ I/O operations. Thus, the total number of I/O operations in this case is $O(m/B+$

$\log_B n + \mathsf{nocc}/B$).

6.2.2 Handling periodic case

We start with the following simple observation by Ganguly *et al.* [60].

Observation 2. *If we list all the occurrences of $P = Q^{\alpha}R$ in T in the ascending order, we can see clusters of occurrences holding the following property: two consecutive occurrences (i) within a cluster, are exactly* $\mathsf{period}(P)$ *distance apart and (ii) not within a cluster cannot have an overlap of length* $\mathsf{period}(P)$ *or more.*

Lemma 24. *Let π be the number of clusters. Then $\pi = O(\mathsf{nocc})$.*

Proof. Two occurrences i, j not within the same cluster overlap only if i is the last occurrence in a cluster and j is the first occurrence within the next cluster (follows from Observation 2(ii)). However, only one of them can be a part of the final output. Therefore, $\mathsf{nocc} \geq \pi/2$.

□

Definition 20. *An occurrence is a cluster-head (resp., cluster-tail) iff it is the first (resp., last) occurrence within a cluster. Also, let L' (resp., L'') be the list of all cluster heads (resp., tails) in their* ascending order.

Observe that the distance between two consecutive non-overlapping occurrences within the same cluster, denoted by λ is $\mathsf{period}(P) \cdot \lceil m/\mathsf{period}(P) \rceil$. See Fig 6.1.

Let C_i be the ith leftmost cluster and S_i (resp., S_i^*) be the largest set of non-overlapping occurrences in C_i including (resp., excluding) the first occurrence $L'[i]$ in C_i. Specifically,

$$S_i = \{L'[i] + k\lambda \mid \text{ for } k = 0, 1, 2, 3, .. \textit{ as long as } L'[i] + k\lambda \leq L''[i]\}$$

$$S_i^* = \{\mathsf{period}(P) + L'[i] + k\lambda \mid \text{ for } k = 0,1,2,3,.. \textit{ as long as } \mathsf{period}(P) + L'[i] + k\lambda \leq L''[i]\}$$

Therefore, the final output can be generated by just examining L' and L'' using the procedure in Algorithm 1. This step takes only $O(\mathsf{nocc}/B)$ I/O operations. The correctness follows from Observation 2.

What remains to show is, how to compute L' and L'' efficiently.

Algorithm 1 Reports the largest set of non-overlapping occurrences of P in T.

```
1: report S_1
2: for (i = 2 to π) do
3:     if (the last reported occurrence and L'[i] are non-overlapping) then report S_i
4:     else report S_i^*
5: end for
```

Computing L'': The List of Cluster Tails

We use the following observation by Ganguly *et al.* [60]: a text position y is the rightmost occurrence of P within a cluster (i.e., cluster-tail) iff $\mathsf{T}[y,n]$ is prefixed by $P = Q^{\alpha}R$, but not by $QP = Q^{1+\alpha}R$. This means, L'' is the sorted list of all elements in the set $\{\mathsf{SA}[i] \mid i \in [\mathsf{sp}(P), \mathsf{ep}(P)] \wedge i \notin [\mathsf{sp}(QP), \mathsf{ep}(QP)]\}$ of size π (see Fig 6.2).

The first step is to compute $\mathsf{locus}(P)$ and $\mathsf{locus}(QP)$, which takes $|P|/B + |QP|/B + \log_B n = O(m/B +$

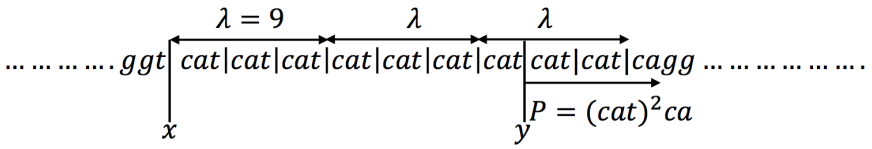

Figure 6.1: Here $P = catcatca$, x is the cluster-head and $y = x + 21$ is the cluster-tail. Then, the largest set of non-overlapping occurrences with the first occurrence included, and the first occurrence excluded are $\{x, x+9, x+18\}$ and $\{x+3, x+12, x+21\}$, respectively.

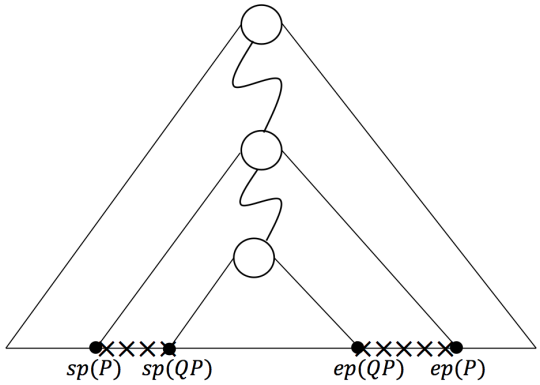

Figure 6.2: Highlighted are the regions corresponding to cluster tails.

$\log_B n$) I/O operations. Then L'' can be obtained via two sorted range reporting queries on SA, in $O(\pi/B)$ I/O operations.

Computing L': The List of Cluster Heads

Clearly, for each cluster of P in T, there is a corresponding cluster of $\overleftarrow{P}$ in $\overleftarrow{\mathsf{T}}$. Here $\overleftarrow{\mathsf{T}}$ (resp., $\overleftarrow{P}$) is the reverse of T (resp., P). Then, we have the following observation.

Observation 3. *Let z be the last occurrence of $\overleftarrow{P}$ within a cluster of $\overleftarrow{P}$ in $\overleftarrow{T}$, then $(n+2-m-z)$ is the first occurrence of P within the corresponding cluster of P in T.*

Therefore, we simply construct and maintain our previous data structure for computing L'', but on $\overleftarrow{T}$. When P comes as input to the original problem, we find L'' corresponding to $\overleftarrow{P}$ in $\overleftarrow{T}$. Then, simply report $(n+2-m-L''[i])$'s in the descending order of i. Note that we need to maintain the suffix tree (and its cache-oblivious version) of $\overleftarrow{T}$ as well. Additional space required is $O(n\log n)$.

In summary, both L' and L'' can be computed in $O(m/B+\log_B n+\pi/B)$ I/O operations using an $O(n\log n)$ space structure. The final output can be generated in additional $O(\mathsf{nocc}/B)$ I/O operations. Also, $\pi = O(\mathsf{nocc})$ from Lemma 24. Therefore, by combining everything, we have the following result.

Theorem 11. *There exists a data structure that is initialized with a text* $\mathsf{T}[1,n]$*, takes* $O(n\log n)$ *words of space and supports the following query in the cache-oblivious model. Given a string* $P[1,m]$*, the data structure reports the largest set of non-overlapping occurrences of* $P[1,m]$ *in* T *in their sorted order in optimal* $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}}{B})$ *I/O operations, where* nocc *is the output size.*

The space complexity can be improved in the cache-aware model.

Theorem 12. *There exists a data structure that is initialized with a text* $\mathsf{T}[1,n]$*, takes* $O(n\log_{M/B} n)$ *words of space and supports the following query in the cache-aware model. Given a string* $P[1,m]$*, the data structure reports the largest set of non-overlapping occurrences of* $P[1,m]$ *in* T *in their sorted order in optimal* $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}}{B})$ *I/O operations, where* nocc *is the output size.*

Proof. We modify our data structure for Theorem 11 as follows. Replace the sorted range reporting structure of Lemma 21 by Lemma 22. Also, discard the structure of Lemma 23, because in the cache-aware model, $\mathsf{period}(P)$ can be computed in $O(m/B)$ I/O operations [107].

□

6.3 Range Non-Overlapping Indexing in Cache-Aware Model

The range non-overlapping problem is to preprocess a text $\mathsf{T}[1,n]$ into a data structure that supports the following query: given a pattern $P[1,m]$ and a range $[s,e]$, report the largest set of occurrences of P in T, that are within the range $[s,e]$ (denote its size by $\mathsf{nocc}_{[s,e]}$), such that any two (distinct) text positions in the output are separated by at least m characters. The problem is a combination of the non-overlapping indexing problem and the position restricted pattern matching problem [32, 33, 43, 72, 86, 90].

When $s=1$ or $e=n$, we call this *the one-sided range non-overlapping indexing* problem. We now proceed to present our solutions in the cache-aware model.

6.3.1 The Data Structure

In [7], Afshani *et al.* showed that an array $A[1,n]$ can be preprocessed into an $O(n\log n)$ space structure, such that given a query $([i,j],k)$, we can report the top-k elements in $A[i,j]$ in their sorted order in optimal $O(\log_B n + k/B)$ I/O operations. Using standard techniques, this result can be extended as follows: an array $A[1,n]$ can be preprocessed into an $O(n\log^2 n)$ space structure, such that given a query $([i,j],[s,e],k)$, we can report the top-k elements in $\{A[t] \mid t \in [i,j], A[t] \in [s,e]\}$ in sorted order in optimal number of I/O operations. The main component of data structure for range non-overlapping indexing is essentially this structure with A being the suffix array of T. As before, we handle the aperiodic and periodic cases separately.

6.3.2 Handling aperiodic case

Find the suffix range of P and then report all those occurrences of P that are within $[s,e]$ in their sorted order. This step can be performed in optimal I/O operations using the structure described above. Then, scan them in the ascending order and do the following: report the first occurrence and report any other occurrence iff it is not overlapping with the last reported occurrence. Total number of I/O operations required is $O(m/B + \log_B n + \mathsf{nocc}_{[s,e]}/B)$.

6.3.3 Handling periodic case

First, we find all cluster-heads that are within the range $[s,e]$ in their sorted order. Additionally, we find the first occurrence of P in $\mathsf{T}[s,e]$ and add to the beginning of this list and call it L'. Similarly, find all cluster-tails that are within the range $[s,e]$ in their sorted order. Additionally, we find the last occurrence of P in $\mathsf{T}[s,e]$ and add it to the end of this list and call it L''. In order to perform these steps in optimal I/O operations, we rely on our $O(n\log^2 n)$ space structure. To obtain our final answer, we simply run Algorithm 1 described in Section 6.2.2. The correctness and the I/O complexity can be easily verified.

Theorem 13. *There exists an $O(n\log^2 n)$ space data structure for the range non-overlapping indexing prob-*

lem in the cache-aware model, where n is the length of the input text T. *The data structure supports reporting the largest set of non-overlapping occurrences of an input pattern* $P[1,m]$ *within a given range* $[s,e]$ *in their sorted order in optimal* $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}_{[s,e]}}{B})$ *I/O operations, where* $\mathsf{nocc}_{[s,e]}$ *is the output size.*

Remark

In the case of one-sided range non-overlapping indexing, the space complexity of the result in Theorem 13 can be improved to $O(n\log n)$.

6.4 Conclusion

Reporting or counting the occurrences of a query pattern p in an input text T is a well-studied problem which can be solved efficiently using the traditional suffix tree data structure in $\mathcal{O}(|p|+\mathsf{occ})$ [67]. However, reporting the non-overlapping occurrences of p in T in optimal time is more complicated. In this chapter, we studied the non-overlapping indexing problem in the cache-oblivious model. We presented a new data structure of size $O(n\log n)$ words that can answer queries in optimal $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}}{B})$ I/O operations, where B is the block size. The space can be improved to $O(n\log_{M/B} n)$ in the cache-aware model, where M is the size of main memory. Additionally, we studied a generalization of this problem with an additional range $[s,e]$ constraint. Here the task is to report the largest set of non-overlapping occurrences of P in T, that are within the range $[s,e]$. We presented an $O(n\log^2 n)$ space data structure in the cache-aware model that can answer queries in optimal $O(\frac{m}{B}+\log_B n+\frac{\mathsf{nocc}_{[s,e]}}{B})$ I/O operations, where $\mathsf{nocc}_{[s,e]}$ is the output size.

LIST OF REFERENCES

[1] Paniz Abedin, M Oğuzhan Külekci, and Sharma V Thankachan. "A Survey on Shortest Unique Substring Queries". In: *Algorithms* 13.9 (2020), p. 224.

[2] Paniz Abedin et al. "A Linear-Space Data Structure for Range-LCP Queries in Poly-Logarithmic Time". In: *Computing and Combinatorics - 24th International Conference, COCOON 2018, Qing Dao, China, July 2-4, 2018, Proceedings*. 2018, pp. 615–625. DOI: 10.1007/978-3-319-94776-1_51. URL: https://doi.org/10.1007/978-3-319-94776-1%5C_51.

[3] Paniz Abedin et al. "A linear-space data structure for range-LCP queries in poly-logarithmic time". In: *Theoretical Computer Science* 822 (2020), pp. 15–22.

[4] Paniz Abedin et al. "Efficient Data Structures for Range Shortest Unique Substring Queries". In: *Algorithms* 13.11 (2020), p. 276. DOI: 10.3390/a13110276. URL: https://doi.org/10.3390/a13110276.

[5] Paniz Abedin et al. "Range Shortest Unique Substring Queries". In: *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*. Ed. by Nieves R. Brisaboa and Simon J. Puglisi. Vol. 11811. Lecture Notes in Computer Science. Springer, 2019, pp. 258–266. DOI: 10.1007/978-3-030-32686-9_18. URL: https://doi.org/10.1007/978-3-030-32686-9%5C_18.

[6] Paniz Abedin et al. "The heaviest induced ancestors problem revisited". In: *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[7] Peyman Afshani, Gerth Stølting Brodal, and Norbert Zeh. "Ordered and Unordered Top-K Range Reporting in Large Data Sets". In: *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA,*

January 23-25, 2011. 2011, pp. 390–400. DOI: 10.1137/1.9781611973082.31. URL: https://doi.org/10.1137/1.9781611973082.31.

[8] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems". In: *Commun. ACM* 31.9 (1988), pp. 1116–1127. DOI: 10.1145/48529.48535. URL: http://doi.acm.org/10.1145/48529.48535.

[9] Daniel R. Allen, Sharma V. Thankachan, and Bojian Xu. "A Practical and Efficient Algorithm for the k-mismatch Shortest Unique Substring Finding Problem". In: *Proceedings of the 2018 ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB 2018, Washington, DC, USA, August 29 - September 01, 2018*. 2018, pp. 428–437. DOI: 10.1145/3233547.3233564. URL: https://doi.org/10.1145/3233547.3233564.

[10] Yannis Almirantis et al. "On avoided words, absent words, and their application to biological sequence analysis". In: *Algorithms Mol. Biol.* 12.1 (2017), 5:1–5:12. DOI: 10.1186/s13015-017-0094-z. URL: https://doi.org/10.1186/s13015-017-0094-z.

[11] Yannis Almirantis et al. "On overabundant words and their application to biological sequence analysis". In: *Theor. Comput. Sci.* 792 (2019), pp. 85–95. DOI: 10.1016/j.tcs.2018.09.011. URL: https://doi.org/10.1016/j.tcs.2018.09.011.

[12] Srinivas Aluru. *Handbook of computational molecular biology*. CRC Press, 2005.

[13] Srinivas Aluru, Alberto Apostolico, and Sharma V. Thankachan. "Efficient Alignment Free Sequence Comparison with Bounded Mismatches". In: *Research in Computational Molecular Biology - 19th Annual International Conference, RECOMB 2015, Warsaw, Poland, April 12-15, 2015, Proceedings*. Ed. by Teresa M. Przytycka. Vol. 9029. Lecture Notes in Computer Science. Springer, 2015, pp. 1–12. DOI: 10.1007/978-3-319-16706-0_1. URL: https://doi.org/10.1007/978-3-319-16706-0%5C_1.

[14] Amihood Amir and Itai Boneh. “Locally Maximal Common Factors as a Tool for Efficient Dynamic String Algorithms”. In: *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*. Ed. by Gonzalo Navarro, David Sankoff, and Binhai Zhu. Vol. 105. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 11:1–11:13. DOI: 10.4230/LIPIcs.CPM.2018.11. URL: https://doi.org/10.4230/LIPIcs.CPM.2018.11.

[15] Amihood Amir and Eitan Kondratovsky. “Searching for a Modified Pattern in a Changing Text”. In: *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*. Ed. by Travis Gagie et al. Vol. 11147. Lecture Notes in Computer Science. Springer, 2018, pp. 241–253. DOI: 10.1007/978-3-030-00479-8_20. URL: https://doi.org/10.1007/978-3-030-00479-8%5C_20.

[16] Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. “Range LCP Queries Revisited”. In: *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*. 2015, pp. 350–361. DOI: 10.1007/978-3-319-23826-5_33. URL: https://doi.org/10.1007/978-3-319-23826-5%5C_33.

[17] Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. “Range LCP Queries Revisited”. In: *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*. 2015, pp. 350–361. DOI: 10.1007/978-3-319-23826-5_33. URL: https://doi.org/10.1007/978-3-319-23826-5%5C_33.

[18] Amihood Amir et al. “Longest Common Factor After One Edit Operation”. In: *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*. Ed. by Gabriele Fici, Marinella Sciortino,

and Rossano Venturini. Vol. 10508. Lecture Notes in Computer Science. Springer, 2017, pp. 14–26. DOI: 10.1007/978-3-319-67428-5_2. URL: https://doi.org/10.1007/978-3-319-67428-5%5C_2.

[19] Amihood Amir et al. "Longest Common Substring Made Fully Dynamic". In: *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*. Ed. by Michael A. Bender, Ola Svensson, and Grzegorz Herman. Vol. 144. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 6:1–6:17. DOI: 10.4230/LIPIcs.ESA.2019.6. URL: https://doi.org/10.4230/LIPIcs.ESA.2019.6.

[20] Amihood Amir et al. "Range LCP". In: *ISAAC*. 2011, pp. 683–692.

[21] Amihood Amir et al. "Range LCP". In: *J. Comput. Syst. Sci.* 80.7 (2014), pp. 1245–1253. DOI: 10.1016/j.jcss.2014.02.010. URL: https://doi.org/10.1016/j.jcss.2014.02.010.

[22] Amihood Amir et al. "Range LCP". In: *J. Comput. Syst. Sci.* 80.7 (2014), pp. 1245–1253. DOI: 10.1016/j.jcss.2014.02.010. URL: https://doi.org/10.1016/j.jcss.2014.02.010.

[23] Amihood Amir et al. "Repetition Detection in a Dynamic String". In: *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*. Ed. by Michael A. Bender, Ola Svensson, and Grzegorz Herman. Vol. 144. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 5:1–5:18. DOI: 10.4230/LIPIcs.ESA.2019.5. URL: https://doi.org/10.4230/LIPIcs.ESA.2019.5.

[24] Alberto Apostolico and Franco P Preparata. "Data structures and algorithms for the string statistics problem". In: *Algorithmica* 15.5 (1996), pp. 481–494.

[25] Michael Arnold and Enno Ohlebusch. "Linear time algorithms for generalizations of the longest common substring problem". In: *Algorithmica* 60.4 (2011), pp. 806–818.

[26] Lorraine A. K. Ayad, Solon P. Pissis, and Dimitris Polychronopoulos. “CNEFinder: finding conserved non-coding elements in genomes”. In: *Bioinformatics* 34.17 (2018), pp. i743–i747. DOI: 10.1093/bioinformatics/bty601. URL: https://doi.org/10.1093/bioinformatics/bty601.

[27] Carl Barton et al. “Linear-time computation of minimal absent words using suffix array”. In: *BMC Bioinform.* 15 (2014), p. 388. DOI: 10.1186/s12859-014-0388-9. URL: https://doi.org/10.1186/s12859-014-0388-9.

[28] Michael A Bender and Martin Farach-Colton. “The LCA problem revisited”. In: *Latin American Symposium on Theoretical Informatics*. Springer. 2000, pp. 88–94.

[29] Michael A. Bender and Martin Farach-Colton. “The LCA Problem Revisited”. In: *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*. Ed. by Gaston H. Gonnet, Daniel Panario, and Alfredo Viola. Vol. 1776. Lecture Notes in Computer Science. Springer, 2000, pp. 88–94. DOI: 10.1007/10719839_9. URL: https://doi.org/10.1007/10719839%5C_9.

[30] Michael A. Bender et al. “Lowest common ancestors in trees and directed acyclic graphs”. In: *J. Algorithms* 57.2 (2005), pp. 75–94. DOI: 10.1016/j.jalgor.2005.08.001. URL: https://doi.org/10.1016/j.jalgor.2005.08.001.

[31] Omer Berkman and Uzi Vishkin. “Recursive Star-Tree Parallel Data Structure”. In: *SIAM J. Comput.* 22.2 (1993), pp. 221–242. DOI: 10.1137/0222017. URL: https://doi.org/10.1137/0222017.

[32] Philip Bille and Inge Li Gørtz. “Substring Range Reporting”. In: *Algorithmica* 69.2 (2014), pp. 384–396. DOI: 10.1007/s00453-012-9733-4. URL: https://doi.org/10.1007/s00453-012-9733-4.

[33] Sudip Biswas et al. “Position-restricted substring searching over small alphabets”. In: *J. Discrete Algorithms* 46-47 (2017), pp. 36–39. DOI: 10.1016/j.jda.2017.10.001. URL: https://doi.org/10.1016/j.jda.2017.10.001.

[34] Gerth Stølting Brodal and Rolf Fagerberg. “Cache-oblivious string dictionaries”. In: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*. 2006, pp. 581–590. URL: http://dl.acm.org/citation.cfm?id=1109557.1109621.

[35] Gerth Stølting Brodal and Allan Grønlund Jørgensen. “Data Structures for Range Median Queries”. In: *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*. Ed. by Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra. Vol. 5878. Lecture Notes in Computer Science. Springer, 2009, pp. 822–831. DOI: 10.1007/978-3-642-10631-6_83. URL: https://doi.org/10.1007/978-3-642-10631-6%5C_83.

[36] Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. “Orthogonal range searching on the RAM, revisited”. In: *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*. 2011, pp. 1–10. DOI: 10.1145/1998196.1998198. URL: http://doi.acm.org/10.1145/1998196.1998198.

[37] Timothy M. Chan et al. “Orthogonal Point Location and Rectangle Stabbing Queries in 3-d”. In: *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*. 2018, 31:1–31:14. DOI: 10.4230/LIPIcs.ICALP.2018.31. URL: https://doi.org/10.4230/LIPIcs.ICALP.2018.31.

[38] Bernard Chazelle. “A Functional Approach to Data Structures and Its Use in Multidimensional Searching”. In: *SIAM J. Comput.* 17.3 (1988), pp. 427–462. DOI: 10.1137/0217026. URL: https://doi.org/10.1137/0217026.

[39] Sriram P. Chockalingam, Sharma V. Thankachan, and Srinivas Aluru. “A parallel algorithm for finding all pairs *k*-mismatch maximal common substrings”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*. Ed. by John West and Cherri M. Pancake. IEEE Computer Society, 2016, pp. 784–794. DOI: 10.1109/SC.2016.66. URL: https://doi.org/10.1109/SC.2016.66.

[40] Hagai Cohen and Ely Porat. “Range Non-overlapping Indexing”. In: *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*. 2009, pp. 1044–1053. DOI: 10.1007/978-3-642-10631-6_105. URL: http://dx.doi.org/10.1007/978-3-642-10631-6_105.

[41] Graham Cormode and S. Muthukrishnan. “Substring compression problems”. In: *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*. 2005, pp. 321–330. URL: http://dl.acm.org/citation.cfm?id=1070432.1070478.

[42] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

[43] Maxime Crochemore et al. “Improved Algorithms for the Range Next Value Problem and Applications”. In: *STACS 2008, 25th Annual Symposium on Theoretical Aspects of Computer Science, Bordeaux, France, February 21-23, 2008, Proceedings*. 2008, pp. 205–216. DOI: 10.4230/LIPIcs.STACS.2008.1359. URL: http://dx.doi.org/10.4230/LIPIcs.STACS.2008.1359.

[44] Maxime Crochemore et al. “Improved algorithms for the range next value problem and applications”. In: *Theor. Comput. Sci.* 434 (2012), pp. 23–34.

[45] Erik D. Demaine, Gad M. Landau, and Oren Weimann. “On Cartesian Trees and Range Minimum Queries”. In: *Algorithmica* 68.3 (2014), pp. 610–625. DOI: 10.1007/s00453-012-9683-x. URL: https://doi.org/10.1007/s00453-012-9683-x.

[46] Martin Farach. “Optimal Suffix Tree Construction with Large Alphabets”. In: *38th Annual Symposium on Foundations of Computer Science, FOCS ’97, Miami Beach, Florida, USA, October 19-22, 1997*. IEEE Computer Society, 1997, pp. 137–143. DOI: 10.1109/SFCS.1997.646102. URL: https://doi.org/10.1109/SFCS.1997.646102.

[47] Martin Farach and S. Muthukrishnan. “Perfect Hashing for Strings: Formalization and Algorithms”. In: *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings*. 1996, pp. 130–140. DOI: 10.1007/3-540-61258-0_11. URL: https://doi.org/10.1007/3-540-61258-0%5C_11.

[48] Martin Farach and Mikkel Thorup. “String matching in lempel—ziv compressed strings”. In: *Algorithmica* 20.4 (1998), pp. 388–404.

[49] Paolo Ferragina and Roberto Grossi. “The String B-tree: A New Data Structure for String Search in External Memory and Its Applications”. In: *J. ACM* 46.2 (1999), pp. 236–280. DOI: 10.1145/301970.301973. URL: https://doi.org/10.1145/301970.301973.

[50] Paolo Ferragina and Giovanni Manzini. “Indexing compressed text”. In: *J. ACM* 52.4 (2005), pp. 552–581. DOI: 10.1145/1082036.1082039. URL: http://doi.acm.org/10.1145/1082036.1082039.

[51] Johannes Fischer. “Inducing the LCP-array”. In: *Workshop on Algorithms and Data Structures*. Springer. 2011, pp. 374–385.

[52] Johannes Fischer and Volker Heun. “Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays”. In: *SIAM J. Comput.* 40.2 (2011), pp. 465–492. DOI: 10.1137/090779759. URL: https://doi.org/10.1137/090779759.

[53] Matteo Frigo et al. “Cache-Oblivious Algorithms”. In: *40th Annual Symposium on Foundations of Computer Science, FOCS ’99, 17-18 October, 1999, New York, NY, USA*. 1999, pp. 285–298. DOI: 10.1109/SFFCS.1999.814600. URL: https://doi.org/10.1109/SFFCS.1999.814600.

[54] Matteo Frigo et al. “Cache-Oblivious Algorithms”. In: *ACM Trans. Algorithms* 8.1 (2012), 4:1–4:22. DOI: 10.1145/2071379.2071383. URL: http://doi.acm.org/10.1145/2071379.2071383.

[55] Mitsuru Funakoshi et al. “Faster Queries for Longest Substring Palindrome After Block Edit”. In: *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*. Ed. by Nadia Pisanti and Solon P. Pissis. Vol. 128. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 27:1–27:13. DOI: 10.4230/LIPIcs.CPM.2019.27. URL: https://doi.org/10.4230/LIPIcs.CPM.2019.27.

[56] Mitsuru Funakoshi et al. “Longest substring palindrome after edit”. In: *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*. Ed. by Gonzalo Navarro, David Sankoff, and Binhai Zhu. Vol. 105. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 12:1–12:14. DOI: 10.4230/LIPIcs.CPM.2018.12. URL: https://doi.org/10.4230/LIPIcs.CPM.2018.12.

[57] Travis Gagie, Pawel Gawrychowski, and Yakov Nekrich. “Heaviest Induced Ancestors and Longest Common Substrings”. In: *Proceedings of the 25th Canadian Conference on Computational Geometry, CCCG 2013, Waterloo, Ontario, Canada, August 8-10, 2013*. Carleton University, Ottawa, Canada, 2013. URL: http://cccg.ca/proceedings/2013/papers/paper%5C_29.pdf.

[58] Travis Gagie et al. “Document Listing on Repetitive Collections”. In: *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*. 2013, pp. 107–119. DOI: 10.1007/978-3-642-38905-4_12. URL: https://doi.org/10.1007/978-3-642-38905-4%5C_12.

[59] Arnab Ganguly, Rahul Shah, and Sharma V Thankachan. “Succinct non-overlapping indexing”. In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2015, pp. 185–195.

[60] Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. “Succinct Non-overlapping Indexing”. In: *Algorithmica* 82.1 (2020), pp. 107–117. DOI: 10.1007/s00453-019-00605-5. URL: https://doi.org/10.1007/s00453-019-00605-5.

[61] Arnab Ganguly et al. “A Linear Space Data Structure for Range LCP Queries”. In: *Fundam. Inform.* 163.3 (2018), pp. 245–251. DOI: 10.3233/FI-2018-1741. URL: https://doi.org/10.3233/FI-2018-1741.

[62] Arnab Ganguly et al. “Space-time trade-offs for finding shortest unique substrings and maximal unique matches”. In: *Theor. Comput. Sci.* 700 (2017), pp. 75–88. DOI: 10.1016/j.tcs.2017.08.002. URL: https://doi.org/10.1016/j.tcs.2017.08.002.

[63] Arnab Ganguly et al. “Space-Time Trade-Offs for the Shortest Unique Substring Problem”. In: *27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia*. 2016, 34:1–34:13. DOI: 10.4230/LIPIcs.ISAAC.2016.34. URL: https://doi.org/10.4230/LIPIcs.ISAAC.2016.34.

[64] Younan Gao. “Computing Matching Statistics on Repetitive Texts”. In: *arXiv preprint arXiv:2111.00376* (2021).

[65] Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. “Weighted Ancestors in Suffix Trees”. In: *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw,*

Poland, September 8-10, 2014. Proceedings. 2014, pp. 455–466. DOI: 10.1007/978-3-662-44777-2_38. URL: https://doi.org/10.1007/978-3-662-44777-2%5C_38.

[66] Roberto Grossi and Jeffrey Scott Vitter. "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching". In: *SIAM J. Comput.* 35.2 (2005), pp. 378–407. DOI: 10.1137/S0097539702402354. URL: https://doi.org/10.1137/S0097539702402354.

[67] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN: 0-521-58519-8. DOI: 10.1017/cbo9780511574931. URL: https://doi.org/10.1017/cbo9780511574931.

[68] Dov Harel and Robert Endre Tarjan. "Fast Algorithms for Finding Nearest Common Ancestors". In: *SIAM J. Comput.* 13.2 (1984), pp. 338–355. DOI: 10.1137/0213024. URL: https://doi.org/10.1137/0213024.

[69] Dov Harel and Robert Endre Tarjan. "Fast Algorithms for Finding Nearest Common Ancestors". In: *SIAM J. Comput.* 13.2 (1984), pp. 338–355. DOI: 10.1137/0213024. URL: https://doi.org/10.1137/0213024.

[70] Bernhard Haubold et al. "Genome comparison without alignment using shortest unique substrings". In: *BMC Bioinformatics* 6 (2005), p. 123. DOI: 10.1186/1471-2105-6-123. URL: https://doi.org/10.1186/1471-2105-6-123.

[71] Wing-Kai Hon, Sharma V. Thankachan, and Bojian Xu. "In-place algorithms for exact and approximate shortest unique substring problems". In: *Theor. Comput. Sci.* 690 (2017), pp. 12–25. DOI: 10.1016/j.tcs.2017.05.032. URL: https://doi.org/10.1016/j.tcs.2017.05.032.

[72] Wing-Kai Hon et al. “On position restricted substring searching in succinct space”. In: *J. Discrete Algorithms* 17 (2012), pp. 109–114. DOI: 10.1016/j.jda.2012.09.002. URL: http://dx.doi.org/10.1016/j.jda.2012.09.002.

[73] Sahar Hooshmand et al. “I/O-efficient data structures for non-overlapping indexing”. In: *Theoretical Computer Science* 857 (2021), pp. 1–7.

[74] Sahar Hooshmand et al. “Non-Overlapping Indexing-Cache Obliviously”. In: *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[75] Atalay Mert Ileri, M. Oguzhan Külekci, and Bojian Xu. “Shortest Unique Substring Query Revisited”. In: *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*. 2014, pp. 172–181. DOI: 10.1007/978-3-319-07566-2_18. URL: https://doi.org/10.1007/978-3-319-07566-2%5C_18.

[76] Costas S. Iliopoulos et al. “Maximal Motif Discovery in a Sliding Window”. In: *String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*. 2018, pp. 191–205. DOI: 10.1007/978-3-030-00479-8_16. URL: https://doi.org/10.1007/978-3-030-00479-8%5C_16.

[77] Hiroe Inoue et al. “Algorithms and combinatorial properties on shortest unique palindromic substrings”. In: *J. Discrete Algorithms* 52 (2018), pp. 122–132. DOI: 10.1016/j.jda.2018.11.009. URL: https://doi.org/10.1016/j.jda.2018.11.009.

[78] Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. “Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Counting”. In: *Algorithms and Computation, 15th International Symposium, ISAAC 2004, Hong Kong, China, December 20-22, 2004, Proceedings*. Ed. by Rudolf Fleischer and Gerhard Trippen. Vol. 3341. Lec-

ture Notes in Computer Science. Springer, 2004, pp. 558–568. DOI: 10.1007/978-3-540-30551-4_49. URL: https://doi.org/10.1007/978-3-540-30551-4%5C_49.

[79] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. "Linear work suffix array construction". In: *J. ACM* 53.6 (2006), pp. 918–936. DOI: 10.1145/1217856.1217858. URL: https://doi.org/10.1145/1217856.1217858.

[80] Toru Kasai et al. "Linear-time longest-common-prefix computation in suffix arrays and its applications". In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 2001, pp. 181–192.

[81] Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein. "Range Non-overlapping Indexing and Successive List Indexing". In: *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings*. 2007, pp. 625–636. DOI: 10.1007/978-3-540-73951-7_54. URL: http://dx.doi.org/10.1007/978-3-540-73951-7_54.

[82] Orgad Keller et al. "Generalized substring compression". In: *Theor. Comput. Sci.* 525 (2014), pp. 42–54. DOI: 10.1016/j.tcs.2013.10.010. URL: https://doi.org/10.1016/j.tcs.2013.10.010.

[83] Dmitry V. Khmelev and William J. Teahan. "A Repetition Based Measure for Verification of Text Collections and for Text Categorization". In: *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*. SIGIR '03. Toronto, Canada: ACM, 2003, pp. 104–110. ISBN: 1-58113-646-3. DOI: 10.1145/860435.860456. URL: http://doi.acm.org/10.1145/860435.860456.

[84] Tomasz Kociumaka. "Efficient data structures for internal queries in texts". In: (2019).

[85] Tomasz Kociumaka, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. “Sublinear space algorithms for the longest common substring problem”. In: *European Symposium on Algorithms*. Springer. 2014, pp. 605–617.

[86] Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. “Persistency in Suffix Trees with Applications to String Interval Problems”. In: *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings*. Ed. by Roberto Grossi, Fabrizio Sebastiani, and Fabrizio Silvestri. Vol. 7024. Lecture Notes in Computer Science. Springer, 2011, pp. 67–80. DOI: 10.1007/978-3-642-24583-1_8. URL: https://doi.org/10.1007/978-3-642-24583-1%5C_8.

[87] Gad M Landau and Uzi Vishkin. “Fast string matching with k differences”. In: *Journal of Computer and System Sciences* 37.1 (1988), pp. 63–78.

[88] Moshe Lewenstein. “Orthogonal Range Searching for Text Indexing”. In: *Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*. 2013, pp. 267–302. DOI: 10.1007/978-3-642-40273-9_18. URL: https://doi.org/10.1007/978-3-642-40273-9%5C_18.

[89] M. Lothaire. *Applied Combinatorics on Words*. Cambridge University Press, 2005. ISBN: 978-0-521-84802-2. URL: http://www.cambridge.org/gb/knowledge/isbn/item1172552/?site%5C_locale=en%5C_GB.

[90] Veli Mäkinen and Gonzalo Navarro. “Position-Restricted Substring Searching”. In: *LATIN 2006: Theoretical Informatics, 7th Latin American Symposium, Valdivia, Chile, March 20-24, 2006, Proceedings*. Ed. by José R. Correa, Alejandro Hevia, and Marcos A. Kiwi. Vol. 3887. Lecture Notes in Computer Science. Springer, 2006, pp. 703–714. DOI: 10.1007/11682462_64. URL: https://doi.org/10.1007/11682462%5C_64.

[91] U Manber and G Myers. “Suffix arrays: a new method for on-line search”. In: *SIAM Journal on Comput ing* (2003), p. 22.

[92] Udi Manber and Gene Myers. “Suffix arrays: a new method for on-line string searches”. In: *siam Journal on Computing* 22.5 (1993), pp. 935–948.

[93] Giovanni Manzini. “Two space saving tricks for linear time LCP array computation”. In: *Scandinavian Workshop on Algorithm Theory*. Springer. 2004, pp. 372–383.

[94] Wataru Matsubara et al. “Efficient algorithms to compute compressed longest common substrings and compressed palindromes”. In: *Theoretical Computer Science* 410.8-10 (2009), pp. 900–913.

[95] Kotaro Matsuda et al. “Compressed Orthogonal Search on Suffix Arrays with Applications to Range LCP”. In: *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*. 2020, 23:1–23:13. DOI: 10.4230/LIPIcs.CPM.2020.23. URL: https://doi.org/10.4230/LIPIcs.CPM.2020.23.

[96] Takuya Mieno et al. “Compact Data Structures for Shortest Unique Substring Queries”. In: *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*. Vol. 11811. Lecture Notes in Computer Science. Springer, 2019, pp. 107–123. DOI: 10.1007/978-3-030-32686-9_8. URL: https://doi.org/10.1007/978-3-030-32686-9%5C_8.

[97] Takuya Mieno et al. “Shortest Unique Substring Queries on Run-Length Encoded Strings”. In: *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*. 2016, 69:1–69:11. DOI: 10.4230/LIPIcs.MFCS.2016.69. URL: https://doi.org/10.4230/LIPIcs.MFCS.2016.69.

[98] James Morris Jr and Vaughan Pratt. *A linear pattern-matching algorithm*. 1970.

[99] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

[100] Gonzalo Navarro and Veli Mäkinen. “Compressed full-text indexes”. In: *ACM Computing Surveys (CSUR)* 39.1 (2007), 2–es.

[101] Yakov Nekrich and Gonzalo Navarro. “Sorted Range Reporting”. In: *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4-6, 2012. Proceedings*. Ed. by Fedor V. Fomin and Petteri Kaski. Vol. 7357. Lecture Notes in Computer Science. Springer, 2012, pp. 271–282. DOI: 10.1007/978-3-642-31155-0_24. URL: https://doi.org/10.1007/978-3-642-31155-0%5C_24.

[102] Paweł Gawrychowski Panagiotis Charalampopoulos and Karol Pokorski. “Dynamic Longest Common Substring in Polylogarithmic Time”. In: *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, 2020*.

[103] Manish Patil, Rahul Shah, and Sharma V Thankachan. “Faster range LCP queries”. In: *International Symposium on String Processing and Information Retrieval*. Springer. 2013, pp. 263–270.

[104] Manish Patil et al. “Categorical range maxima queries”. In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS’14, Snowbird, UT, USA, June 22-27, 2014*. 2014, pp. 266–277. DOI: 10.1145/2594538.2594557. URL: https://doi.org/10.1145/2594538.2594557.

[105] Jian Pei, Wush Chi-Hsuan Wu, and Mi-Yen Yeh. “On shortest unique substring queries”. In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 2013, pp. 937–948. DOI: 10.1109/ICDE.2013.6544887. URL: https://doi.org/10.1109/ICDE.2013.6544887.

[106] Solon P. Pissis. “MoTeX-II: structured MoTif eXtraction from large-scale datasets”. In: *BMC Bioinform.* 15 (2014), p. 235. DOI: 10.1186/1471-2105-15-235. URL: https://doi.org/10.1186/1471-2105-15-235.

[107] Kangho Roh et al. “External Memory Algorithms for String Problems”. In: *Fundam. Inform.* 84.1 (2008), pp. 17–32. URL: http://content.iospress.com/articles/fundamenta-informaticae/fi84-1-03.

[108] Kunihiko Sadakane. “Succinct representations of lcp information and improvements in the compressed suffix arrays”. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*. 2002, pp. 225–232. URL: http://dl.acm.org/citation.cfm?id=545381.545410.

[109] Kunihiko Sadakane and Gonzalo Navarro. “Fully-Functional Succinct Trees”. In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*. 2010, pp. 134–149. DOI: 10.1137/1.9781611973075.13. URL: https://doi.org/10.1137/1.9781611973075.13.

[110] Chris Schleiermacher et al. “REPuter: the manifold applications of repeat analysis on a genomic scale”. In: *Nucleic Acids Research* 29.22 (Nov. 2001), pp. 4633–4642. ISSN: 0305-1048. DOI: 10.1093/nar/29.22.4633.

[111] Daniel W. Schultz and Bojian Xu. “On k-Mismatch Shortest Unique Substring Queries Using GPU”. In: *Bioinformatics Research and Applications - 14th International Symposium, ISBRA 2018, Beijing, China, June 8-11, 2018, Proceedings*. 2018, pp. 193–204. DOI: 10.1007/978-3-319-94968-0_18. URL: https://doi.org/10.1007/978-3-319-94968-0%5C_18.

[112] Daniel Dominic Sleator and Robert Endre Tarjan. “A Data Structure for Dynamic Trees”. In: *Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA*. 1981, pp. 114–122. DOI: 10.1145/800076.802464. URL: http://doi.acm.org/10.1145/800076.802464.

[113] Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. “A Provably Efficient Algorithm for the *k*-Mismatch Average Common Substring Problem”. In: *J. Comput. Biol.*

23.6 (2016), pp. 472–482. DOI: 10.1089/cmb.2015.0235. URL: https://doi.org/10.1089/cmb.2015.0235.

[114] Sharma V. Thankachan, Sriram P. Chockalingam, and Srinivas Aluru. "An Efficient Algorithm for Finding All Pairs k-Mismatch Maximal Common Substrings". In: *Bioinformatics Research and Applications - 12th International Symposium, ISBRA 2016, Minsk, Belarus, June 5-8, 2016, Proceedings*. Ed. by Anu G. Bourgeois et al. Vol. 9683. Lecture Notes in Computer Science. Springer, 2016, pp. 3–14. DOI: 10.1007/978-3-319-38782-6_1. URL: https://doi.org/10.1007/978-3-319-38782-6%5C_1.

[115] Sharma V. Thankachan et al. "Algorithmic Framework for Approximate Matching Under Bounded Edits with Applications to Sequence Analysis". In: *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*. 2018, pp. 211–224. DOI: 10.1007/978-3-319-89929-9_14. URL: https://doi.org/10.1007/978-3-319-89929-9%5C_14.

[116] Kazuya Tsuruta et al. "Shortest Unique Substrings Queries in Optimal Time". In: *SOFSEM 2014: Theory and Practice of Computer Science - 40th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 26-29, 2014, Proceedings*. 2014, pp. 503–513. DOI: 10.1007/978-3-319-04298-5_44. URL: https://doi.org/10.1007/978-3-319-04298-5%5C_44.

[117] Esko Ukkonen. "On-Line Construction of Suffix Trees". In: *Algorithmica* 14.3 (1995), pp. 249–260. DOI: 10.1007/BF01206331. URL: http://dx.doi.org/10.1007/BF01206331.

[118] Yuki Urabe et al. "Longest Lyndon Substring After Edit". In: *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*. Ed. by Gonzalo Navarro, David Sankoff, and Binhai Zhu. Vol. 105. LIPIcs. Schloss Dagstuhl - Leibniz-

Zentrum für Informatik, 2018, 19:1–19:10. DOI: 10.4230/LIPIcs.CPM.2018.19. URL: https://doi.org/10.4230/LIPIcs.CPM.2018.19.

[119] Kiichi Watanabe et al. "Shortest Unique Palindromic Substring Queries on Run-Length Encoded Strings". In: *Combinatorial Algorithms - 30th International Workshop, IWOCA 2019, Pisa, Italy, July 23-25, 2019, Proceedings*. 2019, pp. 430–441. DOI: 10.1007/978-3-030-25005-8_35. URL: https://doi.org/10.1007/978-3-030-25005-8%5C_35.

[120] Peter Weiner. "Linear Pattern Matching Algorithms". In: *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*. 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13. URL: https://doi.org/10.1109/SWAT.1973.13.

[121] Peter Weiner. "Linear Pattern Matching Algorithms". In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*. SWAT '73. Washington, DC, USA: IEEE Computer Society, 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13. URL: https://doi.org/10.1109/SWAT.1973.13.

[122] Peter Weiner. "Linear Pattern Matching Algorithms". In: *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*. 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13. URL: http://dx.doi.org/10.1109/SWAT.1973.13.

[123] Dan E. Willard. "Log-Logarithmic Worst-Case Range Queries are Possible in Space Theta(N)". In: *Inf. Process. Lett.* 17.2 (1983), pp. 81–84. DOI: 10.1016/0020-0190(83)90075-3. URL: https://doi.org/10.1016/0020-0190(83)90075-3.

[124] Ross N Williams. "An extremely fast Ziv-Lempel data compression algorithm". In: *1991 Data Compression Conference*. IEEE Computer Society. 1991, pp. 362–363.

[125] Andrew C. Yao. "Space-time Tradeoff for Answering Range Queries (Extended Abstract)". In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*.

STOC '82. San Francisco, California, USA: ACM, 1982, pp. 128–136. ISBN: 0-89791-070-2. DOI: 10.1145/800070.802185. URL: http://doi.acm.org/10.1145/800070.802185.

[126] Gelin Zhou. "Two-dimensional range successor in optimal time and almost linear space". In: *Inf. Process. Lett.* 116.2 (2016), pp. 171–174. DOI: 10.1016/j.ipl.2015.09.002. URL: https://doi.org/10.1016/j.ipl.2015.09.002.

[127] Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression". In: *IEEE Trans. Information Theory* 23.3 (1977), pp. 337–343. DOI: 10.1109/TIT.1977.1055714. URL: https://doi.org/10.1109/TIT.1977.1055714.