7-26-2022

2-Dimensional String Problems: Data Structures and Quantum Algorithms

Dhrumilkumar Patel
Louisiana State University and Agricultural and Mechanical College

2-DIMENSIONAL STRING PROBLEMS: DATA STRUCTURES AND QUANTUM ALGORITHMS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Masters in Computer Science

in

The Department of Computer Science

by
Dhrumilkumar Patel
B.Tech., International Institute of Information Technology, Hyderabad, India, 2019
August 2022

Dedicated to my Mummy, Papa, and Poodi

Acknowledgements

My Master's journey was indeed a difficult landscape with many surprising uphill and downhill battles. Reaching my goal would not be possible without the guidance of some people.

I first thank my advisor Rahul Shah for taking me as his student and imparting his wisdom along the way. I still remember when I first met him in his office and discussed a personal problem that I had. He was really understanding and helped me overcome that issue. The most important thing that I learned from his unending criticism is how to approach a problem and come up with some novel ideas. Whenever I had a doubt about something, I would not hesitate calling him even at midnight. I suppose the kind of support and environment that he provided made me stronger. I only wish if I am able to implement and maintain his teachings in the future.

I am also in debt to my co-advisor Mark Wilde for providing a significant boost to my initial interest in the world of Quantum Computing. I cannot think of a better guide than Mark for navigating this amazing world and meeting new people along the way. It is quite motivating to observe him work alongside you and tackling stunning problems. I still remember the day I was introduced to the field of quantum information theory in 2016 by his book. From initially reading his books and papers to working with him directly was a amazing step forward in my quantum journey, which I did not anticipate, but I am really grateful for.

I next thank Arnab Ganguly and Sharma Thankachan for their collaborations. I would like to thank my peers Vishal Singh, Soorya Rethinasamy, Aby Philip, Margarite LaBorde, and Arshag Danageozia for endless meetings and presentations that help me understand topics in quantum information theory different from mine. I specially would like to thank Aliza Siddiqui for all the telephonic, as well as one-on-one discussions about the future of our careers. It was good to see that I share my dual interests, i.e, in computer science and quantum information theory, with someone else.

I finally thank my friends and family for their undying support and understanding. I am grateful for all the friendships that I had along the way. Each was different from the other and gave different perspectives in life. I specially would like to thank Simran Dhingra, Shalin Sheth, and Nidhi Khulkarni for all the late night philosophical talks, amazing Indian food and watching random comedy scenes of movies. Lastly, I would like to thank my parents and my little brother Poojan without whom this journey is meaningless. I really appreciate them sending me to the USA for advancing my career. It is not an easy task emotionally and financially. Finally, I would like to say that my father is my superman.

Table of Contents

List of Figures

Abstract

The field of stringology studies algorithms and data structures used for processing strings efficiently. The goal of this thesis is to investigate 2-dimensional (2D) variants of some fundamental string problems, including *Exact Pattern Matching* and *Longest Common Substring*.

In the 2D pattern matching problem, we are given a matrix $\mathsf{M}[1\mathinner{.\,.}n, 1\mathinner{.\,.}n]$ that consists of $N = n \times n$ symbols drawn from an alphabet Σ of size σ. The query consists of a $m \times m$ square matrix $\mathsf{P}[1\mathinner{.\,.}m, 1\mathinner{.\,.}m]$ drawn from the same alphabet, and the task is to find all the locations of P in M. For such square patterns, data structures such as suffix trees and suffix arrays exist for the task of efficient pattern matching. However, a suffix tree occupies $O(N \log N)$ bits, which is significantly more than that of the original text's size of $N \log \sigma$ bits. Therefore, the design of compressed data structures, that supports pattern matching queries efficiently and occupies space close to the original text's size, is imperative. In this thesis, we show an interesting result by designing a compact text index of size $O(N \log \log N + N \log \sigma)$ bits that at least supports efficient inverse suffix array queries. Although, the question of designing a compressed text index that would lead to efficient pattern matching is still evasive, this index gives a hope on the existence of a full 2D compressed text index with all functionalities similar to that of 1D case.

On the other hand, the Longest Common 2D substring problem consists of two 2D strings (matrices), and the task is to report the size of the longest common 2D substring (submatrix) of these 2D strings. It is interesting to know if there exists a sublinear-time algorithm for solving this task. We answer this question positively by presenting a sublinear-time *quantum* algorithm. In addition to this, we prove that any quantum algorithm requires at least $\tilde{\Omega}(N^{2/3})$ time to solve this problem.

Chapter 1.

Introduction

In the field of theoretical computer science, the study of fundamental string problems, also known as *stringology*, (the name was coined by computer scientist Zvi Galil), is an important area of research that has a rich literature. Some of these fundamental string problems include *Exact Pattern Matching*, *Longest Common Substring*, *Longest Palindromic Substring*, and *Lexicographical Minimal Substring.* Out of which, we focus on the 2-dimensional variants of the Exact Pattern Matching and Longest Common Substring problems for the purpose of this thesis. The Exact Pattern Matching problem has a wide variety of applications in the field of bioinformatics, genemoics, databases, data-mining, network systems, etc. On the other hand, the Longest Common Substring problem has played a vital role in plagiarism detection and data deduplication. Stringology has been a classic field of research since the last 50 years, and most of these problems have linear-time algorithms. The key idea of string algorithms is to manipulate strings of characters in such a way that it becomes easier to do certain operations, including comparing them, counting characters, or inspecting its properties, in an efficient manner. For doing this, these string algorithms predominately exploit some hidden structure associated with strings.

In the Exact Pattern Matching problem, we are given a 1-dimensional (1D) string or *text* T of length N. The characters of this text are drawn from an alphabet Σ of size σ. Along with this text, we are also given another 1D string P of length M, which is often called a *pattern*, as an input. The task is to *locate* all the occurrences of P in T. Sometimes, the task is to just *count* the number of times P occurs in T. There exist linear-time algorithms that locate all the occurrences of P in T in $O(N + M)$ time [KMP77, KR87]. Usually, the text T is a fixed string and is much longer than the highly variable pattern P. Thus, the most natural thing to do is to preprocess the text and construct a data structure, that can efficiently answer pattern matching queries. Such data structures are known as *text indexes.* Most popular text indexes include a *suffix* *tree* [McC76a, Ukk95, FFM00, Wei73] and *suffix array* [MM93] that are constructed employing the suffixes of a given text. However, a suffix tree and suffix array both occupy $O(N \log N)$ bits of space, which is in contrast to $N\lceil \log \sigma \rceil$ bits of space occupied by the original text T. Just to give an intuition of how much space these data structures take in comparison to the original text, let us take an example of a human DNA. There are at least 3 billion base pairs in a human genome, which takes approximately 0.8 GB of memory. However, the suffix tree of this genome occupies around 35-45 GB of memory. This memory consumption is even more during the construction of this tree. This example motivates the intuition that the factor of $\log N$ in memory of the suffix tree quickly becomes an overhead as compared to the factor of $\log \sigma$, which stays the same for a given problem. Such a drawback of the memory usage of a suffix tree and suffix array led to the idea of *compressed text indexes*, which asks the following question:

Is it possible to design a text index that occupies space similar to the original text such that the pattern-matching query time stays within poly-logarithmic factor of the suffix tree or suffix array?

Grossi and Vitter [GV05], Ferragina and Manzini [FM05] positively answered the above question by presenting compressed text indexes known as *Compressed Suffix Array* and *FM-index*, respectively. On the other hand, the introduction of the *Compressed Suffix Tree* ensured full-functionalities of suffix trees simulated in compressed space of $O(N \log \sigma)$ bits [Sad07]. At the heart of FM-index lies a key property known as *Last-to-Front mapping* (LF-mapping), which can be efficiently computed using the *Burrows-Wheeler transform* of the original text. As LF-mapping is cricitcal for the purpose of this thesis, we discuss more about it in the ensuing chapter. Furthermore, these initial breakthroughs led to the field of compressed text indexing, which has seen a myriad number of results in the last two decades with many positive developments [Nav16].

There are other variants of text indexing problems where suffix trees and suffix arrays exist, but their compressed counterparts have yet to be found. One of these problems, which has proven to be hard in this context, is the problem of 2-dimensional (2D) pattern matching. In this problem, a text is arranged as a matrix $\mathsf{M}[1\,..\,n, 1\,..\,n]$ and consists of $N = n \times n$ symbols, which are drawn from an alphabet Σ of size σ. The query consists of a $m \times m$ square matrix $\mathsf{P}[1\,..\,m, 1\,..\,m]$, which is also drawn from the same alphabet. Now, the task is to find all the locations of P in M. The patterns can be of any size, but as long as they are square in shape, data structures such as suffix tree and suffix array exist [Gia95, KKP98] for the task of efficient 2D pattern matching. These data structures are essentially 2D counterparts of the data structures previously mentioned for the 1D pattern matching problem. A natural question which arises here is what do we mean by a *suffix* of a matrix. In the following chapter, we go over the definition of a 2D suffix of a matrix and also briefly discuss on how such suffixes can be indexed using a suffix tree akin to the 1D case. The problem of designing a text index for the 2D pattern matching problem in compressed space of $O(N \log \sigma)$ bits, based on the idea of Burrows-Wheeler transform or otherwise, has been long open. There were some attempts and partial results [AG04, MN08], but these mainly focused on entropy compression, without first addressing the more fundamental problem of achieving the optimal space complexity. This gives rise to the following fundamental question regarding compressed text indexing in the 2D pattern matching problem:

Analogous to the 1D pattern matching problem, is it possible to design a succinct or compressed text index for the 2D case?

However, achieving a complexity breakthrough similar to the 1D case has yet to be found, in Chapter 3, we present a text index that at least can answer inverse suffix array queries in near compact space in $O(\mathsf{polylog}(n))$ time. Note that similar to a suffix array, an inverse suffix array is also important in the context of pattern matching. We show this by introducing a novel technique named LFISA-mapping that is an analogue of the LF-mapping operation. Our 2D succinct text index design is based on two 1D compressed suffix trees, and it occupies $O(N \log\log N + N \log \sigma)$-bits of space as compared to previous non-compact space of $O(N \log N)$-bits.

Another fundamental string problem, that we study in this thesis, is the 2-dimensional variant of the Longest Common Substring problem. But first, let us look at the classic version of this problem. In the Longest Common Substring problem, we are given two strings S and T of length N as an input. The task is simple, and it is to report the maximum possible length of a substring that exists in both S and T. To solve this task, there exists a linear-time algorithm, which is based on the suffix tree of the concatenated string $\mathsf{S}\$\mathsf{T}\#$ [Wei73, Far97]. Here, \$ and # are used as delimiters. Recently, due to a rapid growth of interest in the field of quantum computing, some of the stringology research community is asking the following question:

Is it possible to design a quantum algorithm that can offer some speedup compared to classical algorithms for these fundamental string problems?

Le Gall and Seddighin answered the above question positively by proposing sublinear-time quantum algorithms for some of these fundamental string problems [LGS22]. Out of which, they gave a $\tilde{O}(N^{5/6})$-time quantum algorithm for the Longest Common Substring problem. In addition to this, they proved that any quantum algorithm for this problem requires at least $\tilde{\Omega}(N^{2/3})$ time. This gap in time complexity was successfully filled by the authors of Ref. [AJ21], where they proposed a $\tilde{O}(N^{2/3})$-time quantum algorithm for the Longest Commong Substring problem.

In this thesis, we investigate the 2D version of the Longest Common Substring problem. As an input to this modified problem, we are given two 2D strings (matrices), and the task is to report the size of the longest common 2D substring (submatrix) of these 2D strings. Here, the input 2D strings are square in shape. Note that we refer to this problem as the longest common 2D substring instead of the largest common submatrix problem as it makes the naming convention easily extendable to higher dimensional strings.

Similar to the original Longest Common Substring problem, there exists a linear-time algorithm, which is also based on the suffix tree constructed by the 2D suffixes of the input 2D strings. But for the 2D case, the question of designing a sublinear-time quantum algorithm is still open. In Chapter 4, we positively answer this question based on a conjecture that we mention later on in this thesis. In addition to presenting a quantum algorithm for the Longest Common 2D Substring problem, we prove that any quantum algorithm for this problem requires at least $\tilde{\Omega}(N^{2/3})$ time.

Checkpoints: Here, we present the layout of this thesis, mentioning briefly about the chapters and their contents. To begin with, Chapter 2 sets up the ground by shortly discussing some relevant concepts needed to understand the algorithms presented in this thesis. Chapter 3 provides the details related to our compact text index that can answer inverse suffix array queries efficiently. In Chapter 4, we present our quantum algorithm for the Longest Common 2D Substring problem. Finally, we conclude our thesis with some open problems and future directions.

Chapter 2.

Preliminaries

To begin with, we need to understand some basic concepts associated with the traditional or 1D pattern matching problem that includes suffix tree and suffix array. This development will help in understanding similar concepts that are generalized to the 2D case. We also briefly give an overview of two basic quantum algorithms, including Grover's search algorithm and quantum walk algorithm, as they prove to be useful subroutines for our quantum algorithm.

In the traditional or 1D pattern matching problem, we are given a 1D text T of length N over an alphabet Σ, where Σ is a finite totally ordered set of size σ, and a 1D pattern P of size M, which is also drawn from the same alphabet. The task is to obtain the locations of all the occurrences of P in T. We use the notation occ to denote the number of such occurrences for a particular problem instance. As mentioned before in Chapter 1, in most cases, P is a much smaller string than T, which is a long fixed string. Thus, pre-processing T and creating a data structure or an *index* beforehand, in order to make it query efficient, makes more sense than reading the entire text repeatedly for answering every new query. This led to the development of *text indexes*, including suffix trees and suffix arrays.

Answering a pattern matching query efficiently means that we can locate all the occurrences of P in $O((M + \mathsf{occ})\mathsf{polylog}(N))$ time. The text indexes such as suffix trees and suffix arrays occupy $\Theta(N \log N)$ or $\tilde{\Theta}(N)$ bits of space, which is quite large compared to the size of the text T, i.e., $N\lceil \log \sigma \rceil$. A text index is called a *compact* text index if it occupies $\Theta(N \log \sigma)$ bits of space, which is quite comparable to the actual text size.

In the next section, we recall some important definitions and data structures that are relevant for both the problems considered in this thesis.

2.1. Suffix Tree, Suffix Array, and Inverse Suffix Array

Let $\mathsf{S}_{\mathsf{suff}} = \{\mathsf{T}[i \mathinner{..} n] | 1 \leq i \leq n\}$ be the set of all the suffixes of T. The *suffix tree* (denoted by ST) of T is an edge-labeled compact trie constructed from all the suffixes in $\mathsf{S}_{\mathsf{suff}}$ [McC76b, Ukk95, FFM00, Wei73]. In ST, concatenating all the edge labels on a particular root-to-leaf path, we get one of the suffixes in $\mathsf{S}_{\mathsf{suff}}$. In other words, each leaf of ST corresponds to a suffix of T. Additionally, as each suffix $\mathsf{T}[i \mathinner{..} n]$ in $\mathsf{S}_{\mathsf{suff}}$ is uniquely identified with its starting position i in T, we can map text positions to the leaves of ST.

Now, we introduce some additional notations associated with a suffix tree that we use in this thesis. We refer to a character on an edge of ST as a *point*. Given a point c on ST, let $\mathsf{string}(c)$ denote the concatenation of all the characters from the root to c (including c) along the root-to-c path of ST. The depth of this point c on the root-to-c path of ST is given by the length of $\mathsf{string}(c)$, i.e., $\mathsf{depth}(c) = \mathsf{length}(\mathsf{string}(c))$. A node (or a vertex) of ST is a also point because it is represented by the character (point) just above it. The locus u of c on ST is the highest node of ST such that $\mathsf{string}(c)$ is the prefix of $\mathsf{string}(u)$. We denote it as $u = \mathsf{locus}(c)$. We say that a point c on ST is marked iff $\mathsf{marked}(c) = 1$, and its 0 otherwise. We denote the leftmost and rightmost leaves of the subtree of a particular point c as $\mathsf{lleaf}(c)$ and $\mathsf{rleaf}(c)$, respectively. Specifically, the notations $\mathsf{lleaf}(c)$ and $\mathsf{rleaf}(c)$ represent the leaf numbers of the respective leaves, where the numbering of the leaves start from the leftmost leaf of ST. Additionally, we denote r^{th} leftmost leaf of ST as ℓ_r. The difference between the above two notations of a leaf is that the latter notation is point-independent. Let $\mathsf{lca}(c_1, c_2)$ denote the *lowest common ancestor* of points c_1 and c_2. The lowest common ancestor as the name suggests is the node which is a common ancestor of two points with the maximum depth.

Upon traversal of the leaves of a suffix tree from left-to-right, we get suffixes sorted lexicographically, and then storing the corresponding text positions in an array gives an indexing data structure called *suffix array* (SA) [MM93]. Here by $i = \mathsf{SA}[r]$, we mean that the r^{th} leftmost leaf (ℓ_r) in ST corresponds to the suffix $\mathsf{T}[i \mathinner{..} n]$. In other words, r is the lexicographical order or *rank* of the suffix $\mathsf{T}[i \mathinner{..} n]$. Similarly, the *inverse suffix* *array* (ISA) is defined as $\mathsf{ISA}[i] = \mathsf{SA}^{-1}[i] = r$. In other words, the inverse suffix array maps each text position i to the leaf position r in ST.

2.2. LF-mapping

The Last-to-Front mapping or in short LF-mapping is a mapping between a leaf ℓ_r of ST associated with the suffix $T[i\,..\,n]$ to a leaf $\ell_{r'}$ associated with the suffix $T[i-1\,..\,n]$. In other words, given the lexicographical rank of $T[i\,..\,n]$, it outputs the lexicographical rank of $T[i-1\,..\,n]$. Formally, LF-mapping is defined in terms of the suffix array as $\mathsf{LF}(r) = \mathsf{SA}^{-1}[\mathsf{SA}[r]-1]$. This way of computing LF-mapping requires us to store the entire suffix array. However, in case of compressed text indexes, including FM-index, we only store a *sampled suffix array.* Hence, such indexes efficiently computes the LF-mapping using the *Burrows Wheeler Transform (BWT)* [BW94] of the original text along with some auxiliary counting data structures. We do not dwell into the details of BWT in this thesis. The LF computation via BWT lies at the heart of BWT-based text indexes, which enables them to answer pattern matching queries without actually storing the costly suffix array and instead replacing it with a sampled suffix array.

2.3. Succinct Trees with Full Functionality

A fully functional compact/compressed suffix tree is realized using three components: 1) its *compressed tree topology* that supports navigational functionalities (see Fact 2.1 below), 2) a *compressed suffix array* (see Fact 2.2 below), and 3) some auxiliary data structures that support *longest common prefix* (LCP) information.

Fact 2.1 (Fully-Functional Succinct Suffix Tree [Sad07])**.** *The topology of a suffix tree can be encoded in $4n + o(n)$ bits to support the following operations in $O(1)$ time:*

- $\mathsf{pre\text{-}order}(u)/\mathsf{post\text{-}order}(u)$*: pre-order/post-order rank of a node u,*

- $\mathsf{parent}(u)$*: parent of a node u,*

- $\mathsf{nodeDepth}(u)$*: number of edges on the root-to-u path of a node u,*

- $\mathsf{child}(u, q)$*: qth leftmost child of a node u,*

- $\mathsf{sibRank}(u)$*: number of children of* $\mathsf{parent}(u)$ *to the left of a node u,*

- $\mathsf{lca}(u, v)$*: lowest common ancestor (LCA) of two nodes u and v,*

- $\mathsf{lleaf}(u)/\mathsf{rleaf}(u)$*: the leftmost/rightmost leaf in the subtree of a node u,*

- $\mathsf{levelAncestor}(u, d)$*: ancestor of a node u such that* $\mathsf{nodeDepth}(u) = d$

Fact 2.2 (Compressed Suffix Array [Sad07]). *The compressed suffix array part of the above compressed suffix tree can be encoded in $O(n \log \sigma)$ bits to support the following operations:*

- $\mathsf{lookup}(r)$*: returns* $\mathsf{SA}[r]$ *in time* $O(\log^{\epsilon} N)$

- $\mathsf{inverse}(i)$*: returns* $r = \mathsf{SA}^{-1}[i]$ *in time* $O(\log^{\epsilon} N)$*.*

2.4. 2-Dimensional Suffix Tree, Suffix Array, and Inverse Suffix Array

In this section, we generalize the aforementioned notions of a 1D suffix tree, 1D suffix array, and 1D inverse suffix array to the 2D case. We utilize these concepts for building a near compact size data structure that can efficiently answer inverse suffix array queries for 2D pattern matching problem. To begin with, we recall the problem of 2D pattern matching.

In the 2D pattern matching problem, a text is arranged as a matrix $\mathsf{M}[1\,..\,n, 1\,..\,n]$ and consists of $N = n \times n$ symbols, which are drawn from an alphabet Σ of size σ. The query consists of a $m \times m$ square matrix $\mathsf{P}[1\,..\,m, 1\,..\,m]$, which is also drawn from the same alphabet. Now, the task is to find all the locations in M where P appears as a contiguous submatrix. The patterns can be of any size, but as long as they are square in shape, data structures like suffix tree and suffix array exist [Gia95, KKP98] for the task of efficient pattern matching. These data structures are essentially 2D counterparts of the data structures previously discussed for traditional 1D pattern matching problem. These data structures work on the basis of a linearization of 2D suffixes of M, which would preserve the prefix-match property, i.e., every pattern match is a prefix of some suffix. In order to understand the concepts of a suffix tree and suffix array associated with the 2D pattern matching problem, we need to first define what we mean by a 2D suffix of M.

For a 1D text T, an i^{th} suffix is the largest substring of T starting from the i^{th} position, i.e., $\mathsf{T}[i\mathinner{.\,.}n]$. Similarly, this way of defining a suffix can be extended to 2D suffixes of a matrix as well. We define a 2D suffix in the following way:

Definition 2.3. A 2D suffix $S^{\mathsf{2D}}_{i,j}$ defined for a position (i,j) is the largest square submatrix of a matrix M starting at (i,j) position, i.e., $\mathsf{M}[i\mathinner{.\,.}i+l, j\mathinner{.\,.}j+l]$, where $l = n - \max\{i,j\}$.

Giancarlo [GG97] proposed a method for linearizing a 2D suffix such that the final form follows the constraints of *completeness* and *common prefix property* similar to 1D suffixes. The *completeness* constraint is that every square submatrix of M in a linear form must correspond to some *prefix* (whatever the definition of *prefix* is) of some suffix of M, each represented linearly. The *common prefix constraint* is that a square submatrix of M should be a *prefix* of some suffixes of M after linearizing them. Giancarlo first defined L*suffix*, which is a linear representation of a 2D suffix. Here, L stands for *linear*. An Lsuffix $S^{\mathsf{L}}_{i,j}$ of a 2D suffix $S^{\mathsf{2D}}_{i,j}$ is the concatenation of strings $a_0, a_1, a_2, \ldots, a_l$, where $a_0 = \mathsf{M}[i,j]$ and $a_k = \mathsf{M}[i+k, j..j+k-1] \cdot \mathsf{M}[i..i+k, j+k]$. Here, $l = n - \max\{i,j\}$ for all $k \neq 0$ (see Figure 3.1 for example). Here, $\alpha \cdot \beta$ refers to the concatenation of strings α and β.

Let S^{L} be the set of all Lsuffixes of M. Here, $|\mathsf{S}^{\mathsf{L}}| = N$ because there are N 2D suffixes of M. Let ST^{L} be the suffix tree constructed from Lsuffixes in S^{L} (also known as Lsuffix tree). The uncompressed version of ST^{L} [Gia95] takes $\Theta(N \log N)$ bits of space, which is very large compared to the optimal space required to store the original matrix M, i.e., $N\lceil \log \sigma \rceil$. Similarly, the uncompressed version of the suffix array (SA^{L}) [KKP98] for such suffixes also requires $\Theta(N \log N)$ bits of space. The suffix array and inverse suffix array are defined in a similar fashion as defined in the 1D case. For the suffix array, given a rank r, it outputs the position in the matrix of the corresponding Lsuffix $S^{\mathsf{L}}_{i,j}$ i.e. $\mathsf{SA}^{\mathsf{L}}[r] = (i, j)$. Furthermore, the inverse suffix array for Lsuffixes is defined as $\mathsf{ISA}^{\mathsf{L}}[i, j] = r$. We continue this development further in Chapter 3, where we propose a compact text index that supports efficient inverse suffix array queries.

2.5. Quantum Primitives

In this section, we briefly go over some of the fundamental quantum algorithms, including Grover's search and quantum walks. It is worth understanding them first because these algorithms prove to be useful subroutines for our quantum algorithm for the 2D Longest Common Substring problem.

2.5.1. Grover's Search

Lov Grover first proposed a quantum algorithm for searching an item in an unstructured database [Gro96]. This algorithm is well known as Grover's search algorithm. More formally, we can state this searching task in the following way:

Search: Given a classical function $f : \{0, 1\}^n \to \{0, 1\}$, where n is the size of input bit strings, find an input bit string x such that $f(x) = 1$ or report that no such input bit string exists.

Using a classical computer, we need to perform $O(N)$ queries to solve the above task. Here, $N = 2^n$. However, Grover's search algorithm solves this task quadratically faster than a classical computer with a high probability. By quadratically faster we mean that it requires $\tilde{O}(\sqrt{N})$ queries instead of $O(N)$ queries. Furthermore, if the number of input bit strings, that outputs 1, is M, then Grover's search algorithm finds one such input bit string using $\tilde{O}(\sqrt{N/M})$ queries to the input oracle.

2.5.2. Quantum Walks

A quantum walk is a quantum version of a classical random walk. Here, we specifically discuss about the quantum walk algorithm proposed by Magniez, Nayak, Roland, and Santha [MNRS11]. We apply this walk on Johnson graphs.

A Johnson graph $J(m, r)$ has vertices corresponding to the subsets of $\{1, \dots, m\}$, each with r elements. There is an edge between two vertices if their respective subsets differ by one element. In other words, suppose R_1 and R_2 are r-subsets of the set $\{1, \dots, m\}$ such that $|R_1 \cap R_2| = r - 1$, then the vertices corresponding to these two r-subsets are neighbours.

Suppose we have a Johnson graph with some marked vertices and the task is to find one of these marked vertices. More formally, we can state the task in the following way:

Find Marked Vertices: Given a Johnson graph $J(m, r)$ such that ϵ fraction of vertices are marked. The task is to find one of these marked vertices.

In the quantum walk algorithm, each vertex $u \in \binom{m}{r}$ is represented as a unique quantum state. In addition to this, we augment each vertex with some data $\mathsf{data}(u)$, which helps in efficiently finding a marked vertex. Let S denote the cost associated with *setting up* or initializing the data. Let U denote the cost of *updating* the data from $\mathsf{data}(u)$ to $\mathsf{data}(v)$, where u and v are neighbouring vertices. Additionally, let C denote the cost for *checking* if the current vertex is a marked vertex or not. Here, by cost we mean time or query complexity.

A classical random walk algorithm finds a marked vertex with a cost of the order $\mathsf{S} + \left(\frac{m}{r}\right)^2 (r\mathsf{U} + \mathsf{C})$. However, a quantum walk algorithm provides a quadratic speedup over its classical counterpart. In other words, a quantum walk algorithm can search for a marked vertex with a cost of the order $\mathsf{S} + \frac{m}{r} (\sqrt{r}\mathsf{U} + \mathsf{C})$.

Chapter 3.

Inverse Suffix Array Queries for 2-Dimensional Pattern Matching in Near-Compact Space

As discussed in Chapter 1, although, a compact text index that can efficiently answer 2D pattern matching queries has yet to be found, in this chapter, we present a compact text index that can answer inverse suffix array queries in $O(\mathsf{polylog}(N))$ time. We show this by introducing a novel technique named LFISA-mapping that is an analogue of LF-mapping operation typically associated with Burrows–Wheeler Transform. This technique works with the *linearization* scheme for 2D suffixes introduced in Ref. [Gia95] (see Section 2.4. of Chapter 2). Our 2D succinct text index design is based on two 1D compressed suffix trees, and ,overall, it occupies $O(N \log\log N + N \log\sigma)$ bits of space as compared to previous uncompressed indexes that took $O(N \log N)$ bits of space. Here, N is the size of the input matrix and σ is the size of the alphabet.

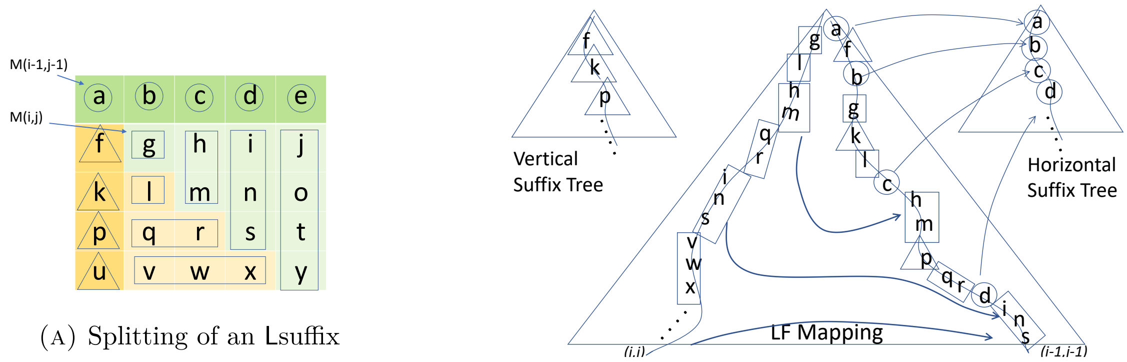
The following theorem states the objective of this chapter more formally:

Theorem 3.1. *The text index for a matrix* M *of size* $N = n \times n$ *can be encoded in* $O(N \log\sigma + N \log\log N)$*-bit space, and an entry in the inverse suffix array* $\mathsf{ISA}^{\mathsf{L}}$ *can be decoded in time* $O(\log N \cdot t_{\mathsf{LFISA}})$*, where* $t_{\mathsf{LFISA}} = O((\log N/\log\log N)^3)$.

Proof. Refer to Section 3.4. for the proof. □

3.1. Definitions

Recall that we introduced notions of 2D suffixes, 2D suffix tree, 2D suffix array, and 2D inverse suffix array in Chapter 2. We also presented a method for linearizing 2D suffixes of a matrix in that chapter. In this section, we advance that discussion by introducing new concepts related to the 2D pattern matching problem, including LFISA-mapping.



(A) Splitting of an Lsuffix

(B) LF-mapping

FIGURE 3.1: Lsuffixes and LF-mapping. a) *Splitting of an* L*suffix*: The characters inside circles are a part of the horizontal suffix $S^{\mathsf{H}}_{i-1,j-1} = abcde\ldots$, and this suffix resides on the horizontal suffix tree. Similarly, the characters inside triangles are a part of the vertical suffix $S^{\mathsf{V}}_{i,j-1} = fkpu...$ and this suffix resides on the vertical suffix tree. Additionally, the linear form of the 2D suffix starting from the position (i, j) is formed by the characters inside rectangles i.e. $S^{\mathsf{L}}_{i,j} = g \cdot l \cdot hm \cdot qr \cdot ins \cdot vwx \cdot joty$ and this linear suffix resides on the Lsuffix tree (the biggest tree on the right). Here $\alpha \cdot \beta$ denotes concatenation of strings α and β. Now the Lsuffix starting at the position $(i-1, j-1)$ is formed by characters of these three sequences, i.e., $S^{\mathsf{L}}_{i-1,j-2} = a \cdot f \cdot bg \cdot kl \cdot chm \cdot pqr \cdot dins \cdot uvwx \cdot ejoty$ b) *LF-mapping*: LF-mapping takes from the leaf corresponding to the Lsuffix starting at a position (i, j) to that of the Lsuffix starting at a position $(i-1, j-1)$. A lot of new characters are introduced in doing so. Therefore, the LF-mapping operation in case of 2D pattern matching problem is not trivial to evaluate.

First, we introduce the LF-mapping operation associated with 2D suffixes (denoted as LF^{L}-mapping) in the following way:

$$\mathsf{LF}^{\mathsf{L}}(r) = \mathsf{ISA}^{\mathsf{L}}[i-1, j-1], \tag{3.1}$$

where $\mathsf{SA}^{\mathsf{L}}[r] = (i, j)$. We assume that $\mathsf{ISA}^{\mathsf{L}}[0, j'] = \mathsf{ISA}^{\mathsf{L}}[i', 0] = \emptyset$. In other words, LF^{L}-mapping operation outputs the rank of an Lsuffix $S^{\mathsf{L}}_{i-1,j-1}$ given the rank of the diagonally-below Lsuffix, i.e., $S^{\mathsf{L}}_{i,j}$.

Figure 3.1 shows an example of a particular LF-mapping operation and how new characters get introduced when going from the Lsuffix $S^{\mathsf{L}}_{i,j}$ to $S^{\mathsf{L}}_{i-1,j-1}$ in contrast to the addition of only one character (in front) in the case of 1D suffixes, i.e., going from $\mathsf{T}[i\,..\,n]$ to $\mathsf{T}[i-1..n]$. This is the reason why it is not trivial to evaluate LF-mapping for the 2D case.

As LF^L-mapping is related to the suffix array SA^L, we introduce a similar mapping for the inverse suffix array ISA^L, which we call *LF-mapping for ISA* ($\mathsf{LFISA}^\mathsf{L}$). We define this mapping in the following manner:

$$\mathsf{LFISA}^\mathsf{L}(i,j) = \mathsf{ISA}^\mathsf{L}[i-1, j-1]. \tag{3.2}$$

Specifically, given the position and rank of an Lsuffix $S^\mathsf{L}_{i,j}$, $\mathsf{LFISA}^\mathsf{L}$-mapping outputs the rank of the diagonally-above Lsuffix, i.e., $S^\mathsf{L}_{i-1,j-1}$. Here, for computational purposes, we provide $\mathsf{ISA}^\mathsf{L}[i,j]$ as an additional parameter. The psuedocode for computing $\mathsf{LFISA}^\mathsf{L}(i,j,\mathsf{ISA}^\mathsf{L}[i,j])$ is given in Section 3.3.2.3..

Now, in order to compute the value of an ISA^L entry, as storing the entire ISA^L takes much space, we sample it and store only those $\mathsf{ISA}^\mathsf{L}[i,j]$ values such that $i = 1 + (k-1)\Delta$, where $k = \{1, 2, ..., \lceil \frac{\sqrt{N}}{\Delta} \rceil\}$. This reduces the problem of computing an ISA^L value to computing at most Δ $\mathsf{LFISA}^\mathsf{L}$-mapping operations. Now, in the latter sections, we show how to compute $\mathsf{LFISA}^\mathsf{L}$-mapping in $t_\mathsf{LFISA} = O((\log N/\log\log N)^3)$ time using our $O(N\log\sigma + N\log\log N)$-bit index. Therefore, ISA^L value for any position in the matrix can be calculated in $t_\mathsf{ISA} = \Delta \cdot t_\mathsf{LFISA} = O(\log N \cdot t_\mathsf{LFISA})$ time, as we take $\Delta = O(\log N)$ for our case.

Firstly, given a matrix M, we linearize it horizontally by concatenating all the rows of M one after another to get a single 1D text T^H of length N. The set of all the suffixes of T^H is defined as $\mathsf{S}^\mathsf{H} = \{\mathsf{T}^\mathsf{H}[i..N] | 1 \leq i \leq N\}$. We denote such suffixes as horizontal or Hsuffixes. Let ST^H be the compressed suffix tree obtained from all the Hsuffixes of T^H. Secondly, by concatenating all the columns into a single text T^V, we linearize M vertically. The set of all the suffixes of T^V is defined as $\mathsf{S}^\mathsf{V} = \{\mathsf{T}^\mathsf{V}[i..N] | 1 \leq i \leq N\}$. Such suffixes are denoted as vertical or Vsuffixes. Here, let ST^V be the compressed suffix tree constructed from all the Vsuffixes of T^V. From the context of M, Hsuffix and Vsuffix starting from the position (i,j) can be written in the following way:

$$S^\mathsf{H}_{i,j} = \mathsf{M}[i, j..n] \cdot \mathsf{M}[i+1, 1..n] \cdot \mathsf{M}[i+2, 1..n] \cdot ... \cdot \mathsf{M}[n, 1..n]$$

$$S^{\mathsf{V}}_{i,j} = \mathsf{M}[i..n, j] \cdot \mathsf{M}[1..n, j+1] \cdot \mathsf{M}[1..n, j+2] \cdot ... \cdot \mathsf{M}[1..n, n].$$

Finally, as ST^{H} and ST^{V} are the compact versions of the original suffix trees, they only occupy $O(N \log \sigma)$ bits of space which is very close to the space required by the original matrix [Sad07]. Fact 2.1 from Chapter 2 mentions their full functionalities. Next, we relate all these defined suffixes.

3.2. Splitting of an Lsuffix

In this section, we show how to split an Lsuffix into three different subsequences. Given an Lsuffix $S^{\mathsf{L}}_{i,j}$ in the 2D form, we can split it into three subsequences: 1) The horizontal subsequence, i.e., the first row $\mathsf{M}[i, j\,..\,n]$, 2) the vertical subsequence, i.e., the first column $\mathsf{M}[i+1\,..\,n, j]$, and 3) the subsequence (linear form) of the remaining square submatrix, i.e., $S^{\mathsf{L}}_{i+1,j+1}$. An example of such a splitting is provided in Figure 3.1. Here, $\mathsf{M}[i, j\,..\,n]$ and $\mathsf{M}[i+1\,..\,n, j]$ subsequences come from the Hsuffix $S^{\mathsf{H}}_{i,j}$ and Vsuffix $S^{\mathsf{V}}_{i+1,j}$, respectively.

Let h_k and v_k be $(k+1)^{th}$ characters of $\mathsf{M}[i, j\,..\,n]$ and $\mathsf{M}[i+1\,..\,n, j]$, respectively. Now as mentioned before, an Lsuffix $S^{\mathsf{L}}_{i,j}$ is the concatenation of strings $a_0, a_1, a_2, \ldots, a_l$, where $a_0 = \mathsf{M}[i, j]$ and $a_k = \mathsf{M}[i+k, j\,..\,j+k-1] \cdot \mathsf{M}[i\,..\,i+k, j+k]$, which is of length $2k+1$ and $l = n - \max(i, j)$ for $k \neq 0$. Similarly, let $S^{\mathsf{L}}_{i+1,j+1}$ be the concatenation of strings $b_0, b_1, b_2, \ldots, b_{l-1}$, where $b_0 = \mathsf{M}[i+1, j+1]$ and $b_k = \mathsf{M}[(i+1)+k, (j+1)\,..\,(j+1)+k-1] \cdot \mathsf{M}[(i+1)\,..\,(i+1)+k, (j+1)+k]$ for all $k \neq 0$. For simplicity, we break each a_k and b_k into two parts in the following way:

$$\begin{aligned}
a^{'}_k &= \mathsf{M}[i+k, j\,..\,j+k-1] \\
a^{''}_k &= \mathsf{M}[i\,..\,i+k, j+k] \\
b^{'}_k &= \mathsf{M}[(i+1)+k, (j+1)\,..\,(j+1)+k-1] \\
b^{''}_k &= \mathsf{M}[(i+1)\,..\,(i+1)+k, (j+1)+k].
\end{aligned}$$

We can rewrite a_k in the following way:

$$a_k^{'} = \mathsf{M}[i+k, j..j+k-1] = v_{k-1}b_{k-1}^{'}$$
$$a_k^{''} = \mathsf{M}[i..i+k, j+k] = h_k b_{k-1}^{''}$$

Therefore, we can say that a_k is the concatenation of strings $v_{k-1}, b_{k-1}^{'}, h_k$, and $b_{k-1}^{''}$. We have $b_0^{'} = \emptyset\ b_0^{''} = b_0$, and $a_0 = h_0$ because $h_0 = \mathsf{M}[i, j]$. We want to redirect the reader's attention to Figure 3.1 where we showcase an example that helps in better understanding of the above concept.

Now, given a_k we can get $v_{k-1}, b_{k-1}^{'}, h_k$, and $b_{k-1}^{''}$ because v_{k-1} and h_k are characters and $b_{k-1}^{'}$ and $b_{k-1}^{''}$ are strings of length $k-2$ and $k-1$, respectively. Here, v_{k-1} and h_k can be thought of as delimiters of the string a_k, and these two uniquely break down a_k into its constituents. Since we know that given a_k we can get $v_{k-1}, b_{k-1}^{'}, h_k, b_{k-1}^{''}$ and vice versa, we denote the *horizontal component* of the Lsuffix $S_{i,j}^{\mathsf{L}}$ by $\mathsf{hc}(S_{i,j}^{\mathsf{L}}) = h_0h_1h_2...h_l$. Similarly, we denote the *vertical component* of $S_{i,j}^{\mathsf{L}}$ by $\mathsf{vc}(S_{i,j}^{\mathsf{L}}) = v_0v_1v_2...v_{l-1}$ and the *square component* of $S_{i,j}^{\mathsf{L}}$ by $\mathsf{sc}(S_{i,j}^{\mathsf{L}}) = b_0b_1b_2...b_{l-1}$. Likewise, we can define these three components for any prefix pf of the Lsuffix $S_{i,j}^{\mathsf{L}}$. We can state the following fact about the relation between the length of the three components of the prefix pf of $S_{i,j}^{\mathsf{L}}$ and its length. Here, by length, we mean the length of the string.

Fact 3.2. $\mathsf{length}(pf) = \mathsf{length}(\mathsf{hc}(pf)) + \mathsf{length}(\mathsf{vc}(pf)) + \mathsf{length}(\mathsf{sc}(pf))$

Now, intuitively, we use such a splitting to evaluate a single LFISA-mapping operation.

3.3. Computing LFISA-mapping

Just to recall, we have three suffix trees based on three different types of suffix definitions as shown before, i.e. ST^{L}, ST^{H} and ST^{V}. Here, we store ST^{H} and ST^{V} as compressed suffix trees (CST) [Sad07] with full functionalities (see Fact 2.1 and 2.2 in Chapter 2 for more details), and together they occupy only $O(N \log \sigma) + O(N \log \sigma) = O(N \log \sigma)$ bits of space. On the contrary, we won't be storing the entire ST^{L} but only the compressed topology of the tree that has navigational functionalities each supported in constant time and occupies $4N + o(N)$ bits of space (see Fact 2.1 in Chapter 2). In the ensuing subsection, first, we show a scheme of marking some relevant points on these trees and then explain how these marked points will help in computing LFISA-mapping. We call this step the *construction step* of our algorithm.

3.3.1. Marking Scheme and Mapping

Firstly, we mark some nodes on Lsuffix tree ST^{L}. We mark a node v_i^{L} of ST^{L} such that $v_i^{\mathsf{L}} = \mathsf{lca}(\ell_{(i-1)g+1}, \ell_{ig})$, where $i = \{1, 2, ..., \lceil \frac{N}{g} \rceil\}$ and g is a *grouping factor*. Furthermore, we define $G_i = [(i-1)g + 1, ig]$ as a *grouping interval.* For our case, we shall use $g = \lceil \log^3 N \rceil$. Hence, the total number of marked nodes on ST^{L} is $O(\frac{N}{\log^3 N})$. Now, we define *marked ancestor*, *lowest marked ancestor*, *cover* of a leaf, and $\mathsf{coveredby}(v^{\mathsf{L}})$ set of a marked node in the following way:

Definition 3.3 (Marked Ancestor)**.** A marked node v^{L} is a marked ancestor of a leaf ℓ iff v^{L} lies on the root-to-leaf path of ℓ in the suffix tree.

Definition 3.4 (Lowest Marked Ancestor)**.** A node v^{L} is the lowest marked ancestor of a leaf ℓ iff it is the lowest (one with the maximum string depth) among all the marked ancestors of ℓ.

Definition 3.5 (Cover)**.** A node v^{L} is the cover of the leaf ℓ iff it is the lowest marked ancestor of ℓ.

Definition 3.6 ($\mathsf{coveredby}(v^{\mathsf{L}})$ set)**.** A $\mathsf{coveredby}(v^{\mathsf{L}})$ is the set of the leaves for which v^{L} is the cover.

As mentioned before in Section 3.2. showcasing the splitting of an Lsuffix, given a marked node v^{L}, we split its associated string, i.e., $\mathsf{string}(v^{\mathsf{L}})$ into its horizontal, vertical, and square components, i.e., $\mathsf{hc}(\mathsf{string}(v^{\mathsf{L}}))$, $\mathsf{vc}(\mathsf{string}(v^{\mathsf{L}}))$ and $\mathsf{sc}(\mathsf{string}(v^{\mathsf{L}}))$, respectively.

For a marked node v^L in ST^L, we mark a point p^H in ST^H corresponding to its horizontal component such that $\mathsf{string}(p^\mathsf{H}) = \mathsf{hc}(\mathsf{string}(v^\mathsf{L}))$. Similarly, we mark points p^V and p^L corresponding to its vertical and square components in ST^V and ST^L, respectively. We refer these points as *shadow* points. Just to recall, a point is any character on the edge of the suffix tree. Note that v^L is not the same marked node as p^L even though they are marked on the same tree (see Figure 3.2). We repeat the above process for every marked node on v^L in ST^L.

At the end of the marking process, let MP^H, MP^V, and MP^L be the sets of all the shadow points on ST^H, ST^V, and ST^L, respectively. Hence, a marked node v^L in ST^L can be viewed as a unique triplet of shadow points in ST^H, ST^V, and ST^L, i.e., $v^\mathsf{L} = (p^\mathsf{H}, p^\mathsf{V}, p^\mathsf{L})$. Therefore, the total number of shadow points in each tree is bounded because the number of marked nodes are bounded. In other words, $|\mathsf{MP}^\mathsf{H}| = |\mathsf{MP}^\mathsf{V}| = |\mathsf{MP}^\mathsf{L}| = O(\frac{N}{\log^3 N})$, where $|X|$ is the cardinality of a set X. Due to this one-to-one correspondence between a marked node and triplet of shadow points, we define a set $U \subseteq \mathsf{MP}^\mathsf{H} \times \mathsf{MP}^\mathsf{V} \times \mathsf{MP}^\mathsf{L}$ which consists of only those triplets of shadow points that represent valid marked nodes.

Now, we state our central task as follows:

Given $\mathsf{ISA}^\mathsf{L}[i,j]$, compute $\mathsf{ISA}^\mathsf{L}[i-1,j-1]$.

We shall preprocess a given 2D text of size N as an input and then contruct data structures that occupies near-compact space and achieve the aforementioned task in $O(\mathsf{polylog} N)$ time. The next step in this is to use the above marked and shadow points for computing $\mathsf{LFISA}^\mathsf{L}(i, j, \mathsf{ISA}^\mathsf{L}[i,j])$. In the following subsection, we show the details on how to achieve this task, and thereafter we outline its pseudocode for understanding the overall picture.

3.3.2. Computing $\mathsf{LFISA}^\mathsf{L}(\cdot)$

In this section, we show the details for evaluating $\mathsf{LFISA}^\mathsf{L}(i, j, \mathsf{ISA}^\mathsf{L}[i,j])$ given a matrix position (i,j) and $\mathsf{ISA}^\mathsf{L}[i,j]$ as inputs. First, using the $\mathsf{inverse}(\cdot)$ function of ST^H and

$\mathsf{ST^V}$ (see Fact 2.1 in Chapter 2), we evaluate inverse suffix array values $\mathsf{ISA^H}[i-1, j-1]$ and $\mathsf{ISA^V}[i, j-1]$, respectively. For simplicity, let $\mathsf{ISA^L}[i, j] = r, \mathsf{ISA^H}[i-1, j-1] = h, \mathsf{ISA^V}[i, j-1] = v$, and $\mathsf{ISA^L}[i-1, j-1] = \mathsf{LFISA^L}(i, j, \mathsf{ISA^L}[i, j]) = s$.

As the inverse suffix array values are related to the leaves of a suffix tree, let ℓ_h, ℓ_v, and ℓ_r be h^{th}, v^{th}, and r^{th} leftmost leaves in their respective suffix trees. The aim here is to find the leaf ℓ_s in $\mathsf{ST^L}$ using the information provided by the shadow points of our index along the root-to-leaf paths of ℓ_h, ℓ_v and ℓ_r in t_{LFISA} time. We shall use some auxiliary data structures that we introduce latter.

Now, we define a set as $A = \{(p^{\mathsf{H}}, p^{\mathsf{V}}, p^{\mathsf{L}}) \in U |\ \ell_h, \ell_v$, and ℓ_r lie in the respective subtrees of $p^{\mathsf{H}}, p^{\mathsf{V}}$ and $p^{\mathsf{L}}\}$. To put it another way, A is a set of valid triplets of shadow points that lie on the root-to-leaf paths of ℓ_h, ℓ_v and ℓ_r in their respective trees. Out of all the valid triplets that are in A, let a specific triplet or its corresponding marked node v^{L}_{max} be defined as follows,

$$v^{\mathsf{L}}_{max} = \underset{v^{\mathsf{L}}=(p^{\mathsf{H}}, p^{\mathsf{V}}, p^{\mathsf{L}}) \in A}{\text{argmax}} (\mathsf{depth}(\mathsf{string}(v^{\mathsf{L}}))). \tag{3.1}$$

Recall that there is a one-to-one correspondence between the marked nodes in $\mathsf{ST^L}$ and triplets in U.

Lemma 3.7 proves that the marked node v^{L}_{max} is the *lowest marked ancestor (or cover)* of the leaf ℓ_s. Therefore, the marked node v^{L}_{max} along with some augmenting information shown in later subsection, will lead us to the leaf ℓ_s, which is what we are interested in. But first, we need to obtain v^{L}_{max}. For that, let us refer the query given by (3.3.2.) as *lowest marked ancestor query.*

Lemma 3.7. *The marked node v^{L}_{max} in $\mathsf{ST^L}$ is the lowest marked ancestor (or cover) of the leaf ℓ_s.*

Proof. Firstly, we prove that a valid triplet $v^{\mathsf{L}} = (p^{\mathsf{H}}, p^{\mathsf{V}}, p^{\mathsf{L}}) \in A$ is a marked ancestor of the leaf ℓ_s. As p^{H} is a shadow point on the root-to-leaf path of ℓ_h, $\mathsf{string}(p^{\mathsf{H}})$ is a prefix of the horizontal suffix $S^{\mathsf{H}}_{i-1,j-1}$ because $\mathsf{ISA}^{\mathsf{H}}[i-1, j-1] = h$. Similarly, $\mathsf{string}(p^{\mathsf{V}})$ and $\mathsf{string}(p^{\mathsf{L}})$ are prefixes of the vertical suffix $S^{\mathsf{V}}_{i,j-1}$ and Lsuffix $S^{\mathsf{L}}_{i,j}$, respectively.

Furthermore, as $\mathsf{string}(p^{\mathsf{H}})$, $\mathsf{string}(p^{\mathsf{V}})$, and $\mathsf{string}(p^{\mathsf{L}})$ are the horizontal, vertical, and square components of the $\mathsf{string}(v^{\mathsf{L}})$, respectively (according to the marking scheme), one of the occurrences of $\mathsf{string}(v^{\mathsf{L}})$ in its 2D form is at matrix position $(i-1, j-1)$. Therefore, it is a prefix of the suffix starting at the position $(i-1, j-1)$, which in its linear form is represented as $S^{\mathsf{L}}_{i-1,j-1}$. Therefore, v^{L} lies on the root-to-leaf path of the leaf representing the Lsuffix $S^{\mathsf{L}}_{i-1,j-1}$, and that leaf is ℓ_s. Hence, v^{L} is a marked ancestor of ℓ_s.

Moreover, as $v^{\mathsf{L}}_{max} \in A$ and is the output of the lowest marked ancestor query that maximizes over string depth over all triplets $v^{\mathsf{L}} \in A$, v^{L}_{max} is the *lowest marked ancestor or cover* of ℓ_s.

□

For obtaining v^{L}_{max}, we reduce the above lowest marked ancestor query to a *stabbing-max* query. This reduction is interesting and useful in our context due to the result mentioned in Theorem 3.8. The details concerning this reduction is discussed in the next subsection. Furthermore, after finding v^{L}_{max}, in order to uniquely go to the correct leaf ℓ_s, we store additional augmenting information that is discussed in a latter subsection. This shows the computation of an $\mathsf{LFISA}^{\mathsf{L}}$ operation. The time complexity of such an operation and the space complexity of our index is discussed in Section **??**.

3.3.2.1. Reduction to 3-dimensional (3D) Stabbing-Max Query

In this section, we show how to reduce that the aforementioned lowest marked ancestor query to a *3-dimensional (3D) stabbing-max* query. In [Nek11], the authors proved the following theorem:

Theorem 3.8. *[Nek11] Given a set I of n 3D rectangles in $\mathbb{R}^3$, where each rectangle rec has a weight $w(rec)$ associated with it, the task of finding a rectangle with maximum*

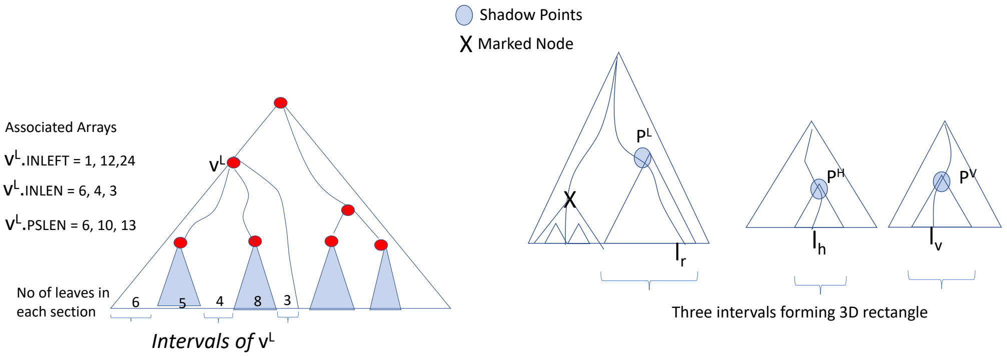


FIGURE 3.2: For a particular marked node v^{L} in ST^{L} (shown in red color), the array INLEFT corresponding to v^{L} stores the start of its associated intervals. Likewise, the array INLEN stores the length of such intervals and the array PSLEN is the prefix-sum array of INLEN. The points $(p^{\mathsf{L}}, p^{\mathsf{H}}, p^{\mathsf{V}})$ are the shadow points of v^{L} (shown as X in the figure shown on the right side).

weight containing (or stabbed by) a 3D query point q can be done in $O((\frac{\log n}{\log\log n})^3)$ time using a data structure occupying $O(n(\frac{\log n}{\log\log n})^2)$ bits of space.

We define the sides of a 3D rectangle rec for each marked node $v^{\mathsf{L}} = (p^{\mathsf{H}}, p^{\mathsf{V}}, p^{\mathsf{L}})$ in U as follows:

$$\begin{aligned}
(x_{left}, x_{right}) &= \Big(\mathsf{lleaf}(p^{\mathsf{H}}), \mathsf{rleaf}(p^{\mathsf{H}})\Big),\\
(y_{up}, y_{down}) &= \Big(\mathsf{lleaf}(p^{\mathsf{V}}), \mathsf{rleaf}(p^{\mathsf{V}})\Big),\\
(z_{front}, z_{back}) &= \Big(\mathsf{lleaf}(p^{\mathsf{L}}), \mathsf{rleaf}(p^{\mathsf{L}})\Big),\\
w(rec) &= \mathsf{depth}(\mathsf{string}(v^{\mathsf{L}})).
\end{aligned}$$

This shows that each triplet in U or its corresponding marked node v^{L} in ST^{L} is uniquely represented by a weighted rectangle.

Next, we define a 3D query point as $q = (h, v, r)$. Therefore, the output of this 3D stabbing-max query is the rectangle with the maximum weight. This rectangle corresponds to the cover of the leaf ℓ_s, i.e, v^{L}_{max}. Furthermore, after obtaining the cover of the leaf, we now provide details on what augmenting information to store in order to get the desired leaf uniquely, i.e., ℓ_s, in the next subsection.

3.3.2.2. Augmenting Information for getting ℓ_s from its Cover v^{L}_{max}

In this section, we explain the procedure of obtaining the correct leaf ℓ_s from its cover v^{L}_{max} by storing this leaf's rank q (say). Now, we define the task for this section: given q and v^{L}_{max}, find the q^{th} leftmost leaf in $\mathsf{coveredby}(v^{\mathsf{L}}_{max})$ (see Definition 3.6).

The challenge here lies due to the fact that v^{L}_{max} may have multiple marked nodes in its subtree, and because of this there may be leaves in its subtree whose lowest marked ancestor or cover is not v^{L}_{max}. Therefore, the set of leaves for which v^{L} is the cover, i.e., $\mathsf{coveredby}(v^{\mathsf{L}})$, can be represented as a set of contiguous intervals. Let us denote this set as $CI = \{I_1, I_2, ..., I_k\}$. Here, $I_i = [a_i, b_i]$, where $i \in \{1, 2, ..., k\}$, and all the leaves between ℓ_{a_i} and ℓ_{b_i} belongs to $\mathsf{coveredby}(v^{\mathsf{L}})$. Here CI is an abbreviation for covered intervals.

Lemma 3.9 proves that the total number of intervals in CI is $O(\sigma)$, i.e., $k = O(\sigma)$. Additionally, this lemma establishes that the total number of leaves, for which v^{L} is the cover, is $\sum_{a=1}^{k} |I_a| = O(k \log^3 N) = O(\sigma \log^3 N)$, where $|I|$ is the length of an interval I. Furthermore, since there is a one-to-one correspondence between a marked node and weighted rectangle, as shown before, we store augmenting information for each rectangle rather than storing it explicitly for the respective marked node. Let the rectangle associated with v^{L}_{max} be denoted as *rec*. Therefore, we let $CI_{rec} = CI$ for simplicity.

Lemma 3.9. *The total number of intervals in CI_{rec} is $O(\sigma)$, and the total number of leaves, for which a marked node (here v^{L}_{max}) is a cover, is $O(\sigma \log^3 N)$.*

Proof. Let c^{L} be one of the child nodes of v^{L}_{max}. Assume that $\mathsf{lleaf}(c^{\mathsf{L}})$ and $\mathsf{rleaf}(c^{\mathsf{L}})$ lie inside the intervals I_i and I_j respectively. First, we prove that I_i and I_j are consecutive intervals.

Suppose there is an interval I_k between I_i and I_j. This means that I_k is entirely contained inside the subtree of c^{L}. In other words, there is an interval of leaves I_k completely inside the subtree of c^{L} for which v^{L}_{max} is the cover. This implies that there is at least one grouping interval of leaves completely contained inside the subtree of c^{L} for which v^{L}_{max} is the lowest common ancestor (lca) of its leftmost and rightmost leaves (see marking scheme for more details). But this is not possible as for v^{L}_{max} to be the lca, the leftmost and rightmost leaves of that grouping interval should exist on two separate downward branches of v^{L}_{max}. This is the contradiction. Therefore, this means that there is no grouping interval completely contained inside the subtree of the child node c^{L}. Hence, there is no I_k that is entirely contained inside the subtree of c^{L}.

The above argument implies that I_i and I_j are consecutive intervals. Therefore, the subtree of a child node of v^{L}_{max} overlaps with at most 2 consecutive intervals in CI_{rec}. Furthermore, there are at most σ child nodes of v^{L}_{max}. Hence, the total number of intervals in CI_{rec} is $O(\sigma)$.

Secondly, there are at most $O(\sigma)$ grouping intervals under the subtree of v^{L}_{max} for which v^{L}_{max} is the lca of its leftmost and rightmost leaves, as each grouping interval need to span over two separate downward branches of v^{L}_{max} for v^{L}_{max} to be that lca. Additionally, the total number of leaves in all such grouping intervals combined is bounded by $O(\sigma \cdot g) = O(\sigma \cdot \log^3 N)$ where g is the grouping factor. This implies that the total number of leaves for which v^{L}_{max} is the lowest marked ancestor or the cover is $(\sigma \log^3 N)$. □

The above lemma proves that the set of leaves, for which v^{L} is the cover, is divided into contiguous intervals of leaves. Now in order to go from v^{L} to ℓ_s, first we store some information to retrieve which interval this leaf belongs to, and then where exactly that leaf is inside that interval.

For each marked node (here v^{L}_{max} or its associated rec), we begin by storing the start of each interval in an array $\mathsf{INLEFT}_{\mathsf{rec}}[\cdot]$. Additionally, we store the size of such intervals in another array $\mathsf{INLEN}_{\mathsf{rec}}[\cdot]$. Moreover, we store the prefix-sum array of $\mathsf{INLEN}_{\mathsf{rec}}[\cdot]$ in an array $\mathsf{PSLEN}_{\mathsf{rec}}[\cdot]$ (see Figure 3.2 for example). Since we are not storing the entire $\mathsf{ISA}^{\mathsf{L}}[\cdot,\cdot]$ because it requires $O(\log N)$ bits for each leaf, instead we store what we call a $\mathsf{miniISA}^{\mathsf{L}}[\cdot,\cdot]$. This is a 2D array that stores a $O(\log\sigma + \log\log^3 N)$-bit number for each matrix position (i, j). This is because each entry in the $\mathsf{miniISA}^{\mathsf{L}}[i, j]$ is the lexicographical rank of the leaf associated with $\mathsf{ISA}^{\mathsf{L}}[i, j]$ under its lowest marked ancestor and the total number of leaves for which a marked node is the lowest marked ancestor is $O(\sigma \log^3 N)$ (Lemma 3.9). Now let $\mathsf{miniISA}^{\mathsf{L}}[i, j] = q$. First we do binary search of q in $\mathsf{PSLEN}_{\mathsf{rec}}[\cdot]$ and get the index e such that the value of $\mathsf{PSLEN}_{\mathsf{rec}}[e]$ is the largest number smaller than q. Now return the final output $s = \mathsf{INLEFT}_{rec}[e] + (q - \mathsf{PSLEN}_{rec}[e])$.

3.3.2.3. Pseudocode of $\mathsf{LFISA}^{\mathsf{L}}$-mapping Operation

Below, we outline the pseudocode for $\mathsf{LFISA}^{\mathsf{L}}$-mapping operation:

Algorithm 1: $\mathsf{LFISA}^{\mathsf{L}}(i, j, \mathsf{ISA}^{\mathsf{L}}(i, j))$

```
$h = \mathsf{ST}^{\mathsf{H}}.inverse(i, j)$
$v = \mathsf{ST}^{\mathsf{V}}.inverse(i, j)$
$s = \mathsf{ISA}^{\mathsf{L}}[i, j]$
$rec = \mathsf{3d\_stabbing\_max}(h, v, s)$
$q = \mathsf{miniISA}^{\mathsf{L}}[i, j]$
$e = binary\_search(\mathsf{PSLEN}_{rec}, q)$
$s = \mathsf{INLEFT}_{rec}[e] + (q - \mathsf{PSLEN}_{rec}[e])$
return $s$
```

3.4. Space and Time Complexity

3.4.1. Space Complexity

At the end of the construction step, we have three suffix trees in our index based on three different types of suffix definitions, along with some auxiliary structures that we store. The horizontal and vertical suffix trees, i.e., ST^{H} and ST^{V}, are stored as compressed suffix trees (see Fact 2.1 and 2.2 in Chapter 2), which together occupy $O(N \log \sigma) + O(N \log \sigma) = O(N \log \sigma)$ bits of space. On the contrary, we only store the compressed topology for the

Lsuffix tree ST^{L} rather than storing the entire suffix tree. This compressed topology provides navigational functionalities, and overall it occupies $4N + o(N)$ bits of space (see Fact 2.1 in Chapter 2).

As previously mentioned in the marking scheme section, the number of marked nodes on ST^{L} is $O(N/\log^3 N)$. Thus, the number of their corresponding shadow points on ST^{L}, ST^{H}, and ST^{V} is also $O(N/\log^3 N)$. Additionally, due to one-to-one correspondence between marked nodes and 3D rectangles, the number of such rectangles is also bounded from above by the same factor.

Each 3D rectangle has a set of arrays associated with it. The length of each of these arrays ($\mathsf{INLEFT}_{\mathsf{rec}}[\cdot]$, $\mathsf{INLEN}_{\mathsf{rec}}[\cdot]$,$\mathsf{PSLEN}_{\mathsf{rec}}[\cdot]$) is the number of intervals under the marked node of that rectangle. As per the marking scheme, the number of grouping intervals is bounded by $O(N/\log^3 N)$. Therefore, the total number of intervals across all the rectangles is also bounded by $O(N/\log^3 N)$ [implication from Lemma 3.9]. Each number in these auxiliary data structures take $O(\log N)$ bits to store. Identifiers for each marked node or shadow points also take at most $O(\log N)$ bits. Thus, the storage space required for all the auxiliary structures is $O(N/\log^2 N) = o(N)$ bits.

If there are t rectangles, the data structure for stabbing-max query takes $O(t(\log t/\log\log t)^2)$ bits of space [Nek11], which is $O(t\log^2 t)$ bits of space. By taking $t = O(N/\log^3 N)$ for our case, the stabbing-max data structure takes $O(N/\log N)$ words of space which is equivalent to $O(N)$ bits of space. This is because each word occupies $O(\log N)$ bits of space.

Finally, for our $\mathsf{miniISA}^{\mathsf{L}}$ structure, we simply store a matrix of dimensions $n \times n$, with each entry $\mathsf{miniISA}^{\mathsf{L}}[i, j]$ taking $O(\log\sigma + \log\log N)$ bits. This is because an entry in $\mathsf{miniISA}^{\mathsf{L}}$ stores a position of the desired leaf among at most $\sigma\log^3 N$ leaves, which have the same lowest marked node. Hence, in total, we get $O(N\log\sigma + N\log\log N)$ bits of space for this part. Additionally, we store the sampled inverse suffix array which has $O(N/\log N)$ elements, where each element takes $O(\log N)$ bits. Therefore, overall, it takes $O(N)$ bits of space.

After summing up all the five parts that are considered in our index, we get $O(N \log \sigma) + o(N) + O(N) + O(N \log \sigma + N \log \log N) + O(N)$ bits of space. This simplifies to $O(N \log \sigma + N \log \log N)$ bits as claimed in Theorem 3.1.

3.4.2. Time Complexity

For evaluating the time complexity of the query evaluation, as a key component, we first focus on computing the $\mathsf{LFISA}^{\mathsf{L}}$-mapping operation. We follow the pseudocode (see Algorithm 3.3.2.3.) step by step for this. The first two steps take $t_{inverse}$ as given by CST, which is $O(\log^{\epsilon} n)$ (see Fact 2.2 in Chapter 2). The third step takes $O(1)$ time since the value is provided as a part of the function. The most time consuming part is the stabbing-max data structure, which takes $O((\log N / \log \log N)^3)$ time. Finding corresponding marked node can be done in $O(1)$ time using succinct tree data structure and searching for prefix sum in the array associated with the rectangle can be done via binary search in $O(\log N)$ time. Thus, overall, the $\mathsf{LFISA}^{\mathsf{L}}$-mapping operation takes $O((\log N / \log \log N)^3)$ time. Finally, considering that our query algorithm for $\mathsf{ISA}^{\mathsf{L}}$ can have at most $\log N$ applications of the $\mathsf{LFISA}^{\mathsf{L}}$-mapping operations, we get our total time complexity as $O(\log^4 N/(\log \log N)^3)$ (as claimed in Theorem 3.1).

3.5. Conclusion

To conclude, we provide an $O(N \log \sigma + N \log \log N)$-bit index that supports inverse suffix array queries in $O(\log^4 N/(\log \log N)^3)$ time. Even though the main goal of developing 2D text index that can allow pattern matching, i.e., to compute suffix array (SA) value or LF values efficiently is not achieved, we think this is a significant step forward in understanding the structure of the problem. Exploring the inter-relations here may lead us to better tools to compute LF operation efficiently in compact space.

Chapter 4.

Sublinear-Time Quantum Algorithm for 2-Dimensional Longest Common Substring

We begin by formally defining the 2-dimensional Longest Common Substring problem (2D-LCS) in the following manner:

Definition 4.1 (2-dimensional Longest Common Substring)**.** Let S and T be input 2D strings (matrices), each of size $N = n \times n$ and are drawn from an alphabet Σ of size σ. Then the task is to find the maximum length ℓ such that there occurs a 2D substring (submatrix) of size $L = \ell \times \ell$ in both S and T.

We already know that there exists an $\tilde{O}(N)$-time classical algorithm, which is based on first constructing a 2D suffix tree and then answering a 2D-LCS query. However, we are interested in understanding if it is possible to design a sublinear-time quantum algorithm under the assumption that quantum query model is our input model (see Section 4.1.). In this chapter, we focus on solving the *decision version* of the 2D-LCS problem: given 2D input strings S and T of size $N = n \times n$, decide if they have a common 2D substring of size at least $D = d \times d$. We perform binary search over the threshold length d in order to output the size of the longest common 2D substring.

Our quantum algorithm for solving the above decision problem is based on an *anchoring* technique, which was previously used for classical and quantum 1D-LCS algorithms [AJ21]. For this anchoring technique, we first need to construct an *anchor* set. We discuss more about anchor sets latter in this chapter. For the 1D-LCS problem, there exists a sublinear-time algorithm for constructing an anchor set of small size. However, the question of designing an algorithm for constructing anchor sets for 2D-LCS is still open. For the purpose of this thesis, we assume that there exists a sublinear-time algorithm for anchor set construction. We formalize this notion in Conjecture 4.6. We now summarize our contribution in the following theorem and prove this theorem in Section 4.2.:

Theorem 4.2. *If Conjecture 4.6 is true, then the decision version of the 2-dimensional Longest Common Substring problem with a threshold length d can be solved using a quantum algorithm with* $\tilde{O}\left(N^{2/3}d^{1/3}\right)$ *query and time complexity.*

In addition to proving the aforementioned theorem, we provide a lower bound of quantum query complexity for the 2D-LCS problem. This bound does not match with the query complexity of our quantum algorithm, implying that either there exists a better lower bound and our algorithm is optimal, or there exists a more efficient algorithm that matches this lower bound. We prove this bound by a reduction from the Element Distinctioness Problem [Amb07]. Finally, we formally state this result in the following theorem and prove this theorem in Section 4.3.:

Theorem 4.3 (Lower Bound)**.** *The 2-Dimensional Longest Common Substring problem requires an algorithm with* $\tilde{\Omega}(N^{2/3})$ *quantum query complexity.*

4.1. Useful Subroutines

We start by defining our computational model. Our computational model is based on the quantum query model, where we assume that the input 2D strings are accessed using *quantum oracles.* This type of assumption is quite standard in the field of quantum algorithms. Let S be an input 2D string of size $N = n \times n$, whose characters are drawn from an alphabet Σ. Assume that we have an access to an oracle O_{S} such that the following unitary mapping holds:

$$O_{\mathsf{S}}|i, j, c\rangle \to |i, j, c \oplus \mathsf{S}[i, j]\rangle, \tag{4.0}$$

where given indices $i, j \in [n]$ and a fixed character $c \in \Sigma$, O_s provides access to the character at the (i, j) position of S. The symbol $\oplus$ denote the XOR operation between the binary encodings of characters. The quantum oracles can be queried in superposition, where each query incurs a unit cost.

The *query complexity* of an algorithm is the number of queries made to the input oracle for solving a given problem. On the other hand, the *time complexity* of an algorithm also counts the number of elementary gates associated with each unitary operations besides the input oracle. One of the useful subroutines for designing a time-efficient version of a query-efficient quantum algorithm is *quantum random access memory* (QRAM). In QRAM, we have a working memory for storing all the items already queried till now, and we have access to its elements at unit cost. The assumption of QRAM is also quite standard in the design of time-efficient implementations of query-efficient quantum algorithms.

We have already briefly introduced quantum primitives such as Grover's search algorithm and quantum walk algorithm in Chapter 2. Using Grover's search algorithm as a subroutine, one can compare any given two strings, as well as compute the length of the longest common prefix (LCP). By comparing two strings we mean that one can determine if they are the same or not. We formalize this notion in the following lemma:

Lemma 4.4 (Lexicographical Comparison of Strings and Computing thier LCP)**.** *Given two 1D strings* S *and* T *of length* N*, one can determine their lexicographical order, i.e.,* $\mathsf{S} \prec \mathsf{T}$*,* $\mathsf{S} \succ \mathsf{T}$*, or* $\mathsf{S} = \mathsf{T}$*, in* $\tilde{O}(\sqrt{N})$ *time using Grover's algorithm. Additionally, we can determine their LCP within the time complexity.*

Proof. The authors of Ref [OR07] show how to compare two strings using Grover's search algorithm in $\tilde{O}(\sqrt{N})$ time. This determines the lexicographical ordering between the two given strings. Furthermore, for computing LCP, one can perform binary search over the length of the prefixes and at each iteration compare the prefixes using the above string-comparison quantum algorithm. □

4.2. A Quantum Algorithm for 2D-LSC

In this section, we present our sublinear-time quantum algorithm based on the anchoring technique, which was previously used for classical and quantum 1D-LCS algorithms [AJ21]. We employ the quantum walk framework discussed in Chapter 2 to apply this technique.

Our quantum algorithm consists of two steps: 1) construction of small anchor sets and 2) anchoring via the quantum walk framework. First, in Section 4.2.1., we conjecture that there exists an efficient sublinear-time construction algorithm that outputs small anchor sets. Then, in Section 4.2.2., we show that our quantum walk algorithm, based on these anchor sets, have sublinear quantum query complexity. Additionally, we design data structures for our quantum walk algorithm and show that our algorithm runs in sublinear time as well. In other words, the time complexity of our algorithm is up to $n^{o(1)}$ factors of the query complexity.

4.2.1. Constructing Small Anchor Sets

We follow Ref. [AJ21] for some standard definitions and terminologies. First, we extend the definition of *good anchor sets* to the 2D case and introduce *good 2D anchor sets* in the following manner:

Definition 4.5 (Good 2D Anchor Sets)**.** Let S and T be input 2D strings. We say that $A_{\mathsf{S}}, A_{\mathsf{T}} \subseteq \{((1,1),(1,1)),\ldots,((n,n),(n,n))\}$ are good anchor sets if the longest common 2D substring of S and T has size at least $D = d \times d$, then there exists positions (i, i') and (j, j') and shifts $h', h'' \in [0, d]$, such that $\mathsf{S}[i \mathinner{..} i+d, i' \mathinner{..} i'+d] = \mathsf{T}[j \mathinner{..} j+d, j' \mathinner{..} j'+d]$, $((i{+}h', i'{+}h'), (i{+}d{+}h'', i'{+}d{-}h'')) \in A_{\mathsf{S}}$, and $((j{+}h', j'{+}h'), (j{+}d{+}h'', j'{+}d{-}h'')) \in A_{\mathsf{T}}$.

Here, each anchor is a tuple of positions on a 2D string. Suppose $A = A_{\mathsf{S}} \cup A_{\mathsf{T}} = \{A_1, \ldots, A_M\}$, and there exists a quantum algorithm, such that given an index $1 \leq m \leq$

M, it computes an element A_m in $T(N, d)$ time. If this computation is possible, then we say that A is $T(N, d)$-quantum-time constructible.

We prefer an anchor set with a size much smaller than the size of the input 2D strings, i.e, $M \ll N$, as well as a sublinear-time algorithm to construct this set. For 1D strings, we can use methods such as *difference covers* [BK03] and *partitioning sets* [BGP] to construct anchor sets. There exists a trade-off between the size and construction time of anchor sets using these methods. Using difference covers, there exists an algorithm that constructs an anchor set of size $O(N/\sqrt{d})$. This way of constructing anchor sets does no depend on the input strings. However, another method, which is based on partitioning sets, is dependent on the input strings and constructs an anchor set of size $O(N/d)$. Although, the size of the anchor set is smaller using the latter method, its construction time is not sublinear. Therefore, this method is not useful for our case. Then, there is a hybrid method that is based on the notion of *approximate difference covers* combined with the idea of the *string synchronizing sets*, introduced in Ref. [AJ21]. This hybrid approach constructs anchor sets in $\tilde{O}\left(N/d^{3/4}\right)$ time, such that m^{th} anchor can be computed in $T = \tilde{O}(\sqrt{d})$ time.

However, it is not trivial to extend any of the above methods from the 1D case to construct good 2D anchor sets that has a favourable trade-off between the construction time and size of the anchor sets. Therefore, we assume that if the following conjecture holds, then our quantum algorithm runs in sublinear quantum query and time complexity:

Conjecture 4.6. *There exists a sublinear time construction algorithm that generates an $\tilde{O}(d)$-quantum-time constructible anchor set of size $M = O(N/d)$.*

Before moving forward, recall that we can linearize a 2D string using the linearization scheme of Ref. [Gia95]. We mentioned this scheme before when we defined 2D suffixes in Chapter 2. We denote such a linear form of a 2D string S by $\mathsf{L}(\mathsf{S})$. Now, if we split a 2D subtring relative to a position (i, i'), there there will be four regions or quadrants. We denote the $d \times d$ 2D string representing the upper-left quadrant by $\mathsf{UL}^d(i, i')$. Similarly, we denote upper-right, lower-left, and lower-right quadrants by $\mathsf{UR}^d(i,i')$, $\mathsf{LL}^d(i,i')$ and $\mathsf{LR}^d(i,i')$, respectively. For linearizing these 2D strings, we begin linearizing from (i,i') and then continue linearizing until we hit the diagonally-opposite end of the square.

For every anchor $A(k) = ((a_k, a'_k), (b_k, b'_k))$, we associate it with the following $d \times d$ size 2D strings:

$$
\begin{aligned}
&\mathsf{SQ}_1(k) = \mathsf{UL}^d(a_k, a'_k),\\
&\mathsf{SQ}_2(k) = \mathsf{LR}^d(a_k, a'_k),\\
&\mathsf{SQ}_3(k) = \mathsf{UL}^d(a_k + d/2, a'_k + d/2),\\
&\mathsf{SQ}_4(k) = \mathsf{LR}^d(a_k - d/2, a'_k - d/2),\\
&\mathsf{SQ}_5(k) = \mathsf{UR}^d(b_k, b'_k),\\
&\mathsf{SQ}_6(k) = \mathsf{LL}^d(b_k, b'_k),\\
&\mathsf{SQ}_7(k) = \mathsf{UR}^d(b_k + d/2, b'_k - d/2),\\
&\mathsf{SQ}_8(k) = \mathsf{LL}^d(b_k - d/2, b'_k + d/2).
\end{aligned}
$$

We also keep track of which set the anchor $A(k)$ belongs to, i.e, $A(k) \in A_{\mathsf{S}}$ or $A(k) \in A_{\mathsf{T}}$. For simplicity of the presentation, we say that $A(k)$ is *blue* if $A(k) \in A_{\mathsf{S}}$, and $A(k)$ is *green* if $A(k) \in A_{\mathsf{T}}$. Similarly, we color the above mentioned 2D strings depending on which colored anchor they associate with.

From the above development, we make the following observation:

Proposition 4.7 (Witness Pair)**.** *The longest common 2D substring of* S *and* T *has size at least* $D = d \times d$*, if and only if, there exists a blue anchor* $A(x) = ((a_x, a'_x), (b_x, b'_x))$ *and green anchor* $A(y) = ((a_y, a'_y), (b_y, b'_y))$*, for some* $x, y \in \{1, \ldots, M\}$*, such that exactly one of the following cases holds. We call such pairs as witness pairs. Here, the LCP of two 2D strings is evaluated by first linearizing them.*

- *Case 1:*

$$\begin{aligned}
\mathsf{lcp}\left(\mathsf{UL}^d(a_x, a'_x), \mathsf{UL}^d(a_y, a'_y)\right) &\leq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x, b'_x), \mathsf{UR}^d(b_y, b'_y)\right) &\leq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UL}^d(a_x, a'_x), \mathsf{UL}^d(a_y, a'_y)\right) + \mathsf{lcp}\left(\mathsf{LR}^d(a_x, a'_x), \mathsf{LR}^d(a_y, a'_y)\right) &\geq d \times d,\\
\mathsf{lcp}\left(\mathsf{UL}^d(a_x + d/2, a'_x + d/2), \mathsf{UL}^d(a_y + d/2, a'_y + d/2)\right) &\geq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x, b'_x), \mathsf{UR}^d(b_y, b'_y)\right) + \mathsf{lcp}\left(\mathsf{LL}^d(b_x, b'_x), \mathsf{LL}^d(b_y, b'_y)\right) &\geq d \times d,\\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x + d/2, b'_x - d/2), \mathsf{UR}^d(b_y + d/2, b'_y - d/2)\right) &\geq d/2 \times d/2.
\end{aligned}$$

- *Case 2:*

$$\begin{aligned}
\mathsf{lcp}\left(\mathsf{UL}^d(a_x, a'_x), \mathsf{UL}^d(a_y, a'_y)\right) &\leq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x, b'_x), \mathsf{UR}^d(b_y, b'_y)\right) &\geq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UL}^d(a_x, a'_x), \mathsf{UL}^d(a_y, a'_y)\right) + \mathsf{lcp}\left(\mathsf{LR}^d(a_x, a'_x), \mathsf{LR}^d(a_y, a'_y)\right) &\geq d \times d,\\
\mathsf{lcp}\left(\mathsf{UL}^d(a_x + d/2, a'_x + d/2), \mathsf{UL}^d(a_y + d/2, a'_y + d/2)\right) &\geq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x, b'_x), \mathsf{UR}^d(b_y, b'_y)\right) + \mathsf{lcp}\left(\mathsf{LL}^d(b_x, b'_x), \mathsf{LL}^d(b_y, b'_y)\right) &\geq d \times d,\\
\mathsf{lcp}\left(\mathsf{LL}^d(b_x - d/2, b'_x + d/2), \mathsf{LL}^d(b_y - d/2, b'_y + d/2)\right) &\geq d/2 \times d/2.
\end{aligned}$$

- *Case 3:*

$$\begin{aligned}
\mathsf{lcp}\left(\mathsf{UL}^d(a_x, a'_x), \mathsf{UL}^d(a_y, a'_y)\right) &\geq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x, b'_x), \mathsf{UR}^d(b_y, b'_y)\right) &\leq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UL}^d(a_x, a'_x), \mathsf{UL}^d(a_y, a'_y)\right) + \mathsf{lcp}\left(\mathsf{LR}^d(a_x, a'_x), \mathsf{LR}^d(a_y, a'_y)\right) &\geq d \times d,\\
\mathsf{lcp}\left(\mathsf{LR}^d(a_x - d/2, a'_x - d/2), \mathsf{LR}^d(a_y - d/2, a'_y - d/2)\right) &\geq d/2 \times d/2,\\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x, b'_x), \mathsf{UR}^d(b_y, b'_y)\right) + \mathsf{lcp}\left(\mathsf{LL}^d(b_x, b'_x), \mathsf{LL}^d(b_y, b'_y)\right) &\geq d \times d,\\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x + d/2, b'_x - d/2), \mathsf{UR}^d(b_y + d/2, b'_y - d/2)\right) &\geq d/2 \times d/2.
\end{aligned}$$

- *Case 4:*

$$\begin{aligned}
\mathsf{lcp}\left(\mathsf{UL}^d(a_x, a'_x), \mathsf{UL}^d(a_y, a'_y)\right) &\geq d/2 \times d/2, \\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x, b'_x), \mathsf{UR}^d(b_y, b'_y)\right) &\geq d/2 \times d/2, \\
\mathsf{lcp}\left(\mathsf{UL}^d(a_x, a'_x), \mathsf{UL}^d(a_y, a'_y)\right) + \mathsf{lcp}\left(\mathsf{LR}^d(a_x, a'_x), \mathsf{LR}^d(a_y, a'_y)\right) &\geq d \times d, \\
\mathsf{lcp}\left(\mathsf{LR}^d(a_x - d/2, a'_x - d/2), \mathsf{LR}^d(a_y - d/2, a'_y - d/2)\right) &\geq d/2 \times d/2, \\
\mathsf{lcp}\left(\mathsf{UR}^d(b_x, b'_x), \mathsf{UR}^d(b_y, b'_y)\right) + \mathsf{lcp}\left(\mathsf{LL}^d(b_x, b'_x), \mathsf{LL}^d(b_y, b'_y)\right) &\geq d \times d, \\
\mathsf{lcp}\left(\mathsf{LL}^d(b_x - d/2, b'_x + d/2), \mathsf{LL}^d(b_y - d/2, b'_y + d/2)\right) &\geq d/2 \times d/2.
\end{aligned}$$

Intuitively, the above cases tell us the positions of the anchor $A(x)$ with respect to a longest common 2D substring it is anchoring. Similarly, it tells the position of the anchor $A(y)$.

4.2.2. Anchoring via the Quantum Walk Framework

Now, we present our quantum walk algorithm that finds such a witness pair. From the above proposition, we know that if such a witness pair exists, then there exists a common 2D substring of size at least $D = d \times d$.

We begin by defining a Johnson graph $J(M, r)$ on which we will do our walk. Here, M is the size of our anchor set $A = A_{\mathsf{S}} \cup A_{\mathsf{T}} = \{A(1), \dots, A_M\}$. Please refer Chapter 2 for a brief overview of the quantum walk framework. Each vertex in this graph corresponds to an r-subset of $\{1, \dots, M\}$. Therefore, the graph consists of $\binom{M}{r}$ number of vertices. Furthermore, a vertex $R = \{k_1, k_2, \dots, k_r\}$ is called a *marked* vertex, if and only if, R contains a witness pair. Overall, if S and T have a common 2D substring of size $D = d \times d$, then at least $\binom{M-2}{r-2}/\binom{M}{r}$ fraction of vertices of the graph are marked vertices. On the contrary, if S and T do not have a common 2D substring of size $D = d \times d$, then there are no marked vertices. We also augment a vertex $R = \{k_1, k_2, \dots, k_r\}$ with the following data:

- The set $R = \{k_1, k_2, \dots, k_r\}$ itself.

- The anchors $\{A(k_1), \ldots, A(k_r)\}$.

- The array $\{k_1^{\mathsf{SQ}_1}, \ldots, k_r^{\mathsf{SQ}_1}\}$ ordered in a such way that $\mathsf{L}(\mathsf{SQ}_1(k_i^{\mathsf{SQ}_1})) \preceq \mathsf{L}(\mathsf{SQ}_1(k_{i+1}^{\mathsf{SQ}_1}))$ for all $1 \leq i \leq r$. Similarly, we store such ordered arrays for $\{\mathsf{SQ}_2(k_i)\}_i, \ldots, \{\mathsf{SQ}_8(k_i)\}_i$.

- The LCP array $h_1^{\mathsf{SQ}_1}, \ldots, h_{r-1}^{\mathsf{SQ}_1}$, where $h_i^{\mathsf{SQ}_1} = \mathsf{lcp}\left(\mathsf{SQ}_1(k_i^{\mathsf{SQ}_1}), (\mathsf{SQ}_1(k_{i+1}^{\mathsf{SQ}_1})\right)$. Similarly, we store LCP arrays for $\{\mathsf{SQ}_2(k_i)\}_i, \ldots, \{\mathsf{SQ}_8(k_i)\}_i$.

Note that we do not need to store entire 2D strings $\{\mathsf{SQ}_1(k_i)\}_i, \ldots, \{\mathsf{SQ}_8(k_i)\}_i$ in order to solve our central problem of searching for a witness pair. We can efficiently evaluate LCP between any pair of 2D strings, say $\mathsf{SQ}_1(k_i^{\mathsf{SQ}_1})$ and $\mathsf{SQ}_1(k_j^{\mathsf{SQ}_1})$, using the above two arrays associated with 2D strings $\{\mathsf{SQ}_1(k_i)\}_i$.

We are now in a position to define our key task that we are trying to solve:

> **Check for a Witness Pair:** Given the above mentioned augmented data with respect to a vertex R, find if R consists of a pair of anchors such that one of the cases of Proposition 4.7 hold.

We show how to implement the above task in sublinear time in Section 4.2.3. using our quantum walk algorithm. For now, we evaluate the query complexity of our quantum walk algorithm in order to solve the above task. As mentioned before in Chapter 2, a quantum walk algorithm can search for a marked vertex with a cost of the following order:

$$S + \frac{M}{r}\left(\sqrt{r}U + C\right), \tag{4.0}$$

where S, U, and C are the costs associated with the stepup, update, and checking steps of our algorithm. For evaluating the query complexity of our algorithm, we need to determine these costs in terms of the query complexity.

Proposition 4.8 (Checking Cost in the Query Complexity)**.** *We can check whether a vertex $R = \{k_1, \ldots, k_r\}$ is marked or not using just the data associated associated with*

R. Therefore, the checking cost is zero because we do not need to make any additional queries to the input oracle.

Proof. Let P be a $d \times d$ size 2D substring of S or T, and let A be its anchor. From Proposition 4.7, we say that the 2D strings defined for A completely covers P. Additionally, the data associated with the anchor A exactly determines the LCP value of every pair. Therefore, we do not need to make additionaly queries to the input oracle for checking if a vertex consists of a witness pair or not. □

Next, we evaluate the cost of an update step of our walk, i.e, going from a vertex to one of its neighbours. Recall that any two vertices of a Johnson graph are neighbours, if and only if, their respective sets differ in exactly two elements. In other words, if we delete an element from the set and insert a new one, then this updated set corresponds to one of the neighbours of the vertex.

Algorithm 2: Inserting a new anchor with an index k

```
1 Compute the anchor $A(k) = ((a_k, a'_k), (b_k, b'_k))$
2 Compute the lexicographical rank $i$ of $\mathsf{SQ}_1(k)$ among $\left\{\mathsf{SQ}_1(k_1^{\mathsf{SQ}_1}), \ldots, \mathsf{SQ}_1(k_r^{\mathsf{SQ}_1})\right\}$
3 Compute $h_{\text{pred}} = \mathsf{lcp}(\mathsf{SQ}_1(k), \mathsf{SQ}_1(k_{i-1}^{\mathsf{SQ}_1}))$
4 Compute $h_{\text{succ}} = \mathsf{lcp}(\mathsf{SQ}_1(k), \mathsf{SQ}_1(k_i^{\mathsf{SQ}_1}))$
5 Compute $h = \mathsf{lcp}(\mathsf{SQ}_1(k_{i-1}^{\mathsf{SQ}_1})), \mathsf{SQ}_1(k_i^{\mathsf{SQ}_1}))$
6 Update the indices as $\left(k_1^{\mathsf{SQ}_1}, \ldots, k_{i-1}^{\mathsf{SQ}_1}, k, k_i^{\mathsf{SQ}_1}, \ldots, k_r^{\mathsf{SQ}_1}\right)$
7 Using $h$ update the LCP array as $\left(h_1^{\mathsf{SQ}_1}, \ldots, h_{i-2}^{\mathsf{SQ}_1}, h_{\text{pred}}, h_{\text{succ}}, h_i^{\mathsf{SQ}_1}, \ldots, h_r^{\mathsf{SQ}_1}\right)$
8 Repeat Steps 3-8 for $\mathsf{SQ}_2(k), \ldots, \mathsf{SQ}_8(k)$
```

Proposition 4.9 (Update Cost in the Query Complexity)**.** *The update cost of each step of our quantum walk algorithm is $U = \tilde{O}(d)$.*

Proof. In an update step, we insert a new anchor k, as well as delete an existing anchor from a given set $\{k_1, \ldots, k_r\}$. The pseudocode for inserting a new anchor point and updating its corresponding data is given in Algorithm 2. On the other hand, deleting an anchor along with its associated data is precisely the reverse of the insertion process.

Therefore, the query complexity of the deletion process stays the same as that of the insertion process.

We follow the pseudocode given by Algorithm 2 for computing the query complexity of the above insertion process. Step 1 computes the anchor $A(k)$ in $\tilde{O}(d)$ (see Conjecture 4.6). In Step 2, for computing the lexicographical rank of $\mathsf{SQ}_1(k)$, we perform binary search over the set $\left\{\mathsf{SQ}_1(k_1^{\mathsf{SQ}_1}), \ldots, \mathsf{SQ}_1(k_r^{\mathsf{SQ}_1})\right\}$. During each step of this binary search, we need to compare the given two 2D strings of size $D = d \times d$, and this comparison can be done in $\tilde{O}(\sqrt{D}) = \tilde{O}(d)$ time, according to Lemma 4.4. Steps 3, 4, and 5 involve computing LCP values between two 2D strings of size D, which can be done in $\tilde{O}(d)$ (see Lemma 4.4). Steps 6 and 7 are $\tilde{O}(1)$-time operations, because here we are just updating the arrays with newly computing LCP values. Finally, we are repeating the above steps a constant number of times. Therefore, the total query complexity of inserting a new anchor is $\tilde{O}(d)$. □

The setup step of our quantum walk algorithm actually involves r insertions. Therefore, we can make the following statement about the query complexity of the setup step:

Proposition 4.10 (Setup Cost in the Query Complexity)**.** *The setup cost of our quantum walk algorithm is $S = \tilde{O}(rd)$.*

Bringing all the aforementioned costs together, we obtain the following query complexity of our quantum walk algorithm:

$$S + \frac{M}{r}\left(\sqrt{r}U + C\right) = rd + \frac{Md}{\sqrt{r}}. \tag{4.0}$$

Now, substituting $r = M^{2/3}$ and $M = \tilde{O}(N/d)$ (see Conjecture 4.6), we get the overall query complexity is $\tilde{O}(N^{2/3}d^{1/3})$. This is the query complexity claimed in Theorem 4.2, and this completes the proof of one part of this theorem. For the other part of this theorem, we need to prove that the time complexity of our algorithm matches our query complexity up to poly-logarithmic factors. Therefore, in Section 4.2.3., we begin by first presenting data structures for storing and retrieving the data associated with the anchors in a time efficient manner. Then, we show how to use these data structures in order to efficiently implement the setup, update, and checking steps.

4.2.3. Time-efficient Implementation

Our quantum walk algorithm consists of the following two steps that are time consuming:

- Insert and delete operations in each update step.
- Checking if a vertex is marked or not during the checking step.

In this section, we present data structures that implement the above two steps in a time-efficient manner. Then, we show that using these data structures, the setup, update, and checking step cost in terms of time complexity are $S = \tilde{O}(rd),\ U = \tilde{O}(d),$ and $C = \tilde{O}(\sqrt{rd})$. This implies that the overall time complexity of our quantum walk algorithm is $\tilde{O}(N^{2/3}d^{1/3})$.

Data structures for insert and delete operations: We prefer data structures that can perform insertions and deletions in $\tilde{O}(1)$ time so as to maintain time complexities of update and setup steps within the same order as their query complexity, i.e., $S = \tilde{O}(rd)$ and $U = \tilde{O}(d)$. However, there are additional constraints that these data structures should follow in order to be useful for our quantum walk algorithm. These constraints are as follows:

- The data structure needs to be *history-independent*, i.e., the data structure should solely depend on the data being stored and not on the series of operations that resulted in this data.
- The data structure should have a *worst-case* time complexity for all the required operations and not expected or amortized time complexity.

The above constraints rule out most of the data structures that support these update operations in $\tilde{O}(1)$ time. Ambainis [Amb07] introduced a data structure based on *hash tables* and *skip lists* that satisfies both the constraints and supports inserting, deleting, and searching in constant time. Furthermore, the authors of Ref. [BLPS21] gave a data structure that also supports indexing, i.e., given an index k, retrieve the k^{th} element from the set.

As our quantum walk algorithm is over the Johnson graph $J(M, r)$, for consistency, we mention the size of the following data structures in terms of r. First, we use hash tables, introduced in Ref. [Amb07], for efficient lookup operations. For more details, please refer to this reference.

Fact 4.11 (Hash Tables)**.** *A hash table supports the following operations in $\tilde{O}(1)$ time by maintaining a set of at most r key-value pairs $\{(key_1, value_1), \ldots, (key_r, value_r)\}$, where $key_i \in [M]$, within $\tilde{O}(r)$ space. This table satisfies the history-independence property.*

- ***Lookup:*** *Given a key $\in [M]$, return the value associated with it.*
- ***Insertion:*** *Given a $(key, value)$ pair, insert it into the existing table.*
- ***Deletion:*** *Given a $(key, value)$ pair, delete it from the existing table.*

We also need a data structure that supports indexing, insertion, deletion, and other operations that support range-minimum queries. For this, we use a skip list, which was first introduced in Ref. [Amb07]. We actually employ the modified version of skip lists, that was presented in Ref. [AJ21] to support indexing and other operations.

Fact 4.12. *A skip list is a history-independent probabilistic data structure that supports the following operations in $\tilde{O}(1)$ time and with high success probability by maintaining an array of items $(key_1, value_1), \ldots, (key_r, value_r)$, where $key_i \in [M]$, within $\tilde{O}(r)$ space:*

- ***Insertion:*** *Given an index $i \in [r]$ and a $(key, value)$ pair, insert this pair into the list at the i^{th} index.*

- ***Deletion:*** *Given an index $i \in [r]$, delete the i^{th} key-value pair from the list.*

- ***Location:*** *Given a key, find its index or location in the list.*

- ***Indexing:*** *Given an index $i \in [r]$, return the i^{th} key-value pair.*

- ***Range-minimum query****: Given an index range $[a, b]$, return $\min_{i\in[a,b]} value_i$.*

Now, we use hash tables (see Fact 4.11) for storing the anchor $A(k)$ corresponding to each index $k \in R$. We employ skip list data structure, from Fact 4.12, for maintaining lexicographical orderings and LCP arrays. For $k \in R$ and $i \in [8]$, we use the notation $\mathsf{lexir}^{\mathsf{SQ}_i}(k)$ to denote the lexicographical rank of $\mathsf{SQ}_i(k)$ among the 2D strings $\{\mathsf{SQ}_i(k)\}_{k\in R}$. Using the aforementioned data structures, given $k_1, k_2 \in R$, we can also compute the following useful information in constant time:

- We can compute the color of k and sizes of 2D strings $\{\mathsf{SQ}_i(k)\}_{i\in[8]}$ by quickly looking up for $A(k)$ from the hash table.

- We can compute $\mathsf{lexir}^{\mathsf{SQ}_i}(k)$ for $i \in [8]$ by using the location operation of the respective skip list (see Fact 4.12).

- We can compute $\mathsf{lcp}\left(\mathsf{SQ}_i(k_1), \mathsf{SQ}_i(k_2)\right)$ for $i \in [8]$ by first computing $j = \mathsf{lexir}^{\mathsf{SQ}_i}(k_1)$ and $j' = \mathsf{lexir}^{\mathsf{SQ}_i}(k_2)$. Then assuming $j' > j$, we can compute $\mathsf{lcp}\left(\mathsf{SQ}_i(k_1), \mathsf{SQ}_i(k_2)\right) = \mathsf{lcp}\left(\mathsf{SQ}_i\left(k_j^{\mathsf{SQ}_1}\right), \mathsf{SQ}_i\left(k_{j'}^{\mathsf{SQ}_1}\right)\right) = \min\left\{h_j^{\mathsf{SQ}_1}, \ldots, h_{j'}^{\mathsf{SQ}_1}\right\}$ using the range-minimum query operation of the corresponding skip list, which is a constant time operation. Similar argument holds when $j' < j$.

Data structures for the checking step: In addition to the above data structures, we also need a 6D range-sum data structure because our task for our checking step (see Task 4.2.2.) reduces to a 6D orthogonal range query. But first, we outline our quantum algorithm that solves Task 4.2.2. during the checking step. This will help in formalizing the intuition of the above reduction to a 6D orthogonal range query.

Algorithm 3: Find a Witness Pair - Checking Step Algorithm

```
Grover's Iterations over blue anchor indices $k_{\text{blue}} \in R$
    Obtain the anchor $A(k) = ((a_k, a'_k), (b_k, b'_k))$ using the hash table
    /* Without the loss of generality, we assume $0 \leq b_k - a_k \leq d/2$.
       Similar steps are executed for other cases with a small
       difference.                                                        */
    Find a range $[\text{start}^{\mathsf{SQ}_1}, \text{end}^{\mathsf{SQ}_1}]$, such that
      $\mathsf{lcpSQ}_1(k_i^{\mathsf{SQ}_1}, \mathsf{SQ}_1(k_{\text{blue}}^{\mathsf{SQ}_1}) \geq \frac{d+a_x+a'_x-b_x-b'_x}{2}$
    Find a range $[\text{start}^{\mathsf{SQ}_2}, \text{end}^{\mathsf{SQ}_2}]$, such that
      $\mathsf{lcpSQ}_2(k_i^{\mathsf{SQ}_2}, \mathsf{SQ}_2(k_{\text{blue}}^{\mathsf{SQ}_2}) \geq d - \frac{d+a_x+a'_x-b_x-b'_x}{2}$
    Find a range $[\text{start}^{\mathsf{SQ}_3}, \text{end}^{\mathsf{SQ}_3}]$, such that
      $\mathsf{lcpSQ}_3(k_i^{\mathsf{SQ}_3}, \mathsf{SQ}_3(k_{\text{blue}}^{\mathsf{SQ}_3}) \geq d/2 + \frac{d+a_x+a'_x-b_x-b'_x}{2}$
    Find a range $[\text{start}^{\mathsf{SQ}_5}, \text{end}^{\mathsf{SQ}_5}]$, such that
      $\mathsf{lcpSQ}_5(k_i^{\mathsf{SQ}_5}, \mathsf{SQ}_5(k_{\text{blue}}^{\mathsf{SQ}_5}) \geq \frac{d-a_x+a'_x+b_x-b'_x}{2}$
    Find a range $[\text{start}^{\mathsf{SQ}_6}, \text{end}^{\mathsf{SQ}_6}]$, such that
      $\mathsf{lcpSQ}_6(k_i^{\mathsf{SQ}_6}, \mathsf{SQ}_6(k_{\text{blue}}^{\mathsf{SQ}_6}) \geq d - \frac{d-a_x+a'_x+b_x-b'_x}{2}$
    Find a range $[\text{start}^{\mathsf{SQ}_7}, \text{end}^{\mathsf{SQ}_7}]$, such that
      $\mathsf{lcpSQ}_7(k_i^{\mathsf{SQ}_7}, \mathsf{SQ}_7(k_{\text{blue}}^{\mathsf{SQ}_7}) \geq d/2 + \frac{d-a_x+a'_x+b_x-b'_x}{2}$
    if there exists a green index $k_{\text{green}} \in R$ such that the following holds: the point
      $(\mathsf{lexir}_{\mathsf{SQ}_1}(k_{\text{green}}), \ldots, \mathsf{lexir}_{\mathsf{SQ}_6}(k_{\text{green}}))$ lies inside the 6D rectangle created by the
      above mentioned ranges then
        return True
return False
```

Lemma 4.13 (6D Range-sum). *There exists a history-independent data structure that maintains a set of 6D points, where each coordinate is an integer, in $\tilde{O}(r)$ space, and this data structure supports the following operations in $O(1)$ time and high success probability.*

- *Insertion: Given a new 6D point, insert it in the current set.*
- *Deletion: Given a 6D point, delete it from the current set if it exists.*
- *Range-sum query: Given a 6D rectangle, find the number of points in the current set lying inside this rectangle.*

Proof. We use a cascaded segment tree for our data structure. Let $\mathcal{N} = [1, \ldots, N]$ be a set from which each coordinate is drawn, where N is a power of two. We first show how to make a 1D segment tree. Then we can extend this construction to the 6D case. For this, we break up the range $\mathcal{N}$ into smaller sub-segments in the following manner:

$$\begin{aligned}
\mathcal{N}_1 &= \{[1, \ldots, N]\}, \\
\mathcal{N}_2 &= \{[1, \ldots, N/2], [N/2+1, \ldots, N]\}, \\
\mathcal{N}_3 &= \{[1, \ldots, N/4], [N/4+1, \ldots, N/2], [N/2+1, \ldots, 3N/4], [3N/4+1, \ldots, N]\}, \\
&\vdots \\
\mathcal{N}_{\log N} &= \{[1, \ldots, 1], [2, \ldots, 2], [3, \ldots, 3], \ldots, [N, \ldots, N]\}.
\end{aligned}$$

Now, it is clear that a range $[x, y]$ can be represented as a disjoint union of $2 \log N$ sub-segments of $\mathcal{N}$. We can now extend this to the 6D case by incorporating a segment tree of segment trees, i.e., a cascaded segment tree with 6 levels of cascading, where a 6D rectangle can be represented as a disjoint union of $O(\log^6 N)$ 6D sub-rectangles of $\mathcal{N}^6$. For every 6D sub-rectangles with non-zero sum, we store this sum in a hash table. Using this data structure, given a query 6D rectangle, we can answer range-sum queries by adding the individual sums of its $O(\log^6 N)$ 6D sub-rectangles in $\tilde{O}(1)$ time. Finally, when inserting a new element, we just need to update the range-sum of $O(\log^6 N)$ many rectangles, which can be done in $\tilde{O}(1)$ time. □

Currently, we are at a stage, where we can analyse the time complexity of Algorithm 3 by following its pseudocode step-by-step. In this algorithm, we perform Grover's iterations over the blue indices, and as there can be at most r blue indices associated with a vertex, total number of such iterations is $\tilde{O}(\sqrt{r})$. The checking step will be time-efficient if each iteration can be performed in $O(\mathsf{polylog}(N))$ time. In Step 3, we compute the range $[\text{start}^{\mathsf{SQ}_1}, \text{end}^{\mathsf{SQ}_1}]$ following the condition mentioned in the pseudocode. This range can be computed in $\tilde{O}(1)$ time using the LCP array associated with 2D strings $\{\mathsf{SQ}_1(k)\}_{k \in R}$.

Similarly, in Step 4-8, we obtain such ranges corresponding to other 2D strings based on some conditions mentioned before. Finally, Step 9 is reduced to a 6D orthogonal range query, which can be again computed in $\tilde{O}(1)$ time using the technique presented in Ref. [AJ21]. Indeed, the authors used their technique for 2D orthogonal range queries, but it can be easily extendable to 6D orthogonal range queries. Thus, the overall time complexity of Algorithm 3 is $C = \tilde{O}(\sqrt{r})$.

We know that the cost associated with the quantum walk algorithm to find a marked vertex of a Johnson's graph is $S + M/r(\sqrt{r}U + C)$. Substituting $S = \tilde{O}(rd)$, $U = \tilde{O}(d)$, $C = \tilde{O}(\sqrt{r})$, $r = M^{2/3}$, and $M = \tilde{O}(N/d)$ in this expression, we obtain the total time complexity of our quantum walk algorithm as $\tilde{O}(N^{2/3}d^{1/3})$. This completes the proof of the second part of Theorem 4.2.

4.3. Lower Bound

In this section, we provide a lower bound to the 2D Longest Common Substring problem by reducing the Element Distinctness problem to it, consequently proving Theorem 4.3.

Let us take the following version of the Element Distinctness problem: Let A be a list of ℓ elements drawn from an alphabet of size ℓ. The list A is prepared in such a way that either all its elements are distinct or there exists one element that repeats twice. The task is to decide which is the case. Ref [Amb07] proves that the above problem has a $\Omega(\ell^{2/3})$-query lower bound. Now, we take an alphabet of size ℓ for our 2D-LCS problem. We draw $\ell/2$ characters uniformly at random from the alphabet and prepare a 2D string S of size $\sqrt{\ell/2} \times \sqrt{\ell/2}$. With the rest of the characters, we prepare a 2D string T of the same size. Now, if all the elements of A are distinct , then S and T does not have a common substring. On the contrary, if there exists an element that repeats in A, then with probability at least 1/2, S and T have a common 2D substring of size 1. This shows a randomized reduction from the Element Distinctness problem to the 2D-LCS problem.

Therefore, by taking $N = \ell/2$, we proved that the 2D-LCS problem requires a quantum algorithm with $\Omega(N^{2/3})$ quantum query complexity.

4.4. Conclusion

To conclude, we design a sublinear-time quantum algorithm for the 2-dimensional Longest Common Substring problem. Specifically, the query, as well as time complexity of our proposed quantum walk algorithm is $\tilde{O}(N^{2/3}d^{1/3})$. Also, we gave a lower bound for this problem by reducing the Element Distinctness problem to it, consequently proving that any quantum algorithm solving the 2-dimensional Longest Common Substring problem requires at least $\tilde{\Omega}(N^{2/3})$ time.

Some of the open questions and future directions are as follows:

- Closing the gap in query and time complexity either by presenting a better quantum algorithm or by proving a tighter lower bound that matches our complexity.
- Our sublinear-time quantum algorithm relies on the conjecture that there exists a sublinear-time quantum algorithm for constructing anchor sets. Therefore, another step is to prove or disprove this conjecture.

Bibliography

[AG04] Jeffrey Scott Vitter Ankur Gupta, Roberto Grossi. Entropy-compressed indexes for multidimensional pattern matching. In *DIMACS working group on Burrows-Wheeler Transform*, 2004.

[AJ21] Shyan Akmal and Ce Jin. Near-optimal quantum algorithms for string problems, 2021.

[Amb07] Andris Ambainis. Quantum walk algorithm for element distinctness. *SIAM Journal on Computing*, 37(1):210–239, 2007.

[BGP] Or Birenzwige, Shay Golan, and Ely Porat. *Locally Consistent Parsing for Text Indexing in Small Space*, pages 607–626.

[BK03] Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In *Annual Symposium on Combinatorial Pattern Matching*, pages 55–69. Springer, 2003.

[BLPS21] Harry Buhrman, Bruno Loff, Subhasree Patro, and Florian Speelman. Limits of quantum speed-ups for computational geometry and other problems: Fine-grained complexity via quantum walks, 2021.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[FFM00] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005. *An extended abstract appeared in FOCS 2000 under the title "Opportunistic Data Structures with Applications".*

[GG97] Raffaele Giancarlo and Roberto Grossi. Suffix tree data structures for matrices. In Alberto Apostolico and Zvi Galil, editors, *Pattern Matching Algorithms*, pages 293–340. Oxford University Press, 1997.

[Gia95] Raffaele Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM J. Comput.*, 24(3):520–562, 1995.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[GV05] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005. *An extended abstract appeared in STOC 2000.*

[KKP98] Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park. Constructing suffix arrays for multi-dimensional matrices. In Martin Farach-Colton, editor, *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20-22, 1998, Proceedings*, volume 1448 of *Lecture Notes in Computer Science*, pages 126–139. Springer, 1998.

[KMP77] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[LGS22] François Le Gall and Saeed Seddighin. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[McC76a] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, apr 1976.

[McC76b] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[MM93] Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[MN08] Veli Mäkinen and Gonzalo Navarro. On self-indexing images - image compression with added value. In *2008 Data Compression Conference (DCC 2008), 25-27 March 2008, Snowbird, UT, USA*, pages 422–431. IEEE Computer Society, 2008.

[MNRS11] Frédéric Magniez, Ashwin Nayak, Jérémie Roland, and Miklos Santha. Search via quantum walk. *SIAM Journal on Computing*, 40(1):142–164, 2011.

[Nav16] Gonzalo Navarro. *Compact Data Structures - A Practical Approach.* Cambridge University Press, 2016.

[Nek11] Yakov Nekrich. A dynamic stabbing-max data structure with sub-logarithmic query time. In Takao Asano, Shin-Ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, volume 7074 of *Lecture Notes in Computer Science*, pages 170–179. Springer, 2011.

[OR07] David Oliveira and Rubens Ramos. Quantum bit string comparator: Circuits and applications. *Quantum Computers and Computing*, 7, 01 2007.

[Sad07] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973.

Vita

Dhrumil Patel was born in Ahmedabad, India in 1995. He received his bachelor's degree from International Institute of Information Technology, Hyderabad, India. Dhrumil published 7 papers during his master's journey at Louisiana State University (LSU). He anticipates his graduation from LSU with a master's degree on August 2022. His interests lie in the field of algorithms, especially quantum algorithms, data structures, and computational complexity.