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

Dhrumil Patel ✉
Division of Computer Science, Louisiana State University, Baton Rouge, LA, USA

Rahul Shah ✉
Division of Computer Science, Louisiana State University, Baton Rouge, LA, USA

Abstract

In a 2-dimensional (2D) pattern matching problem, the text is arranged as a matrix $\mathsf{M}[1..n, 1..n]$ and consists of $N = n \times n$ symbols drawn from alphabet set Σ of size σ. The query consists of a $m \times m$ square matrix $\mathsf{P}[1..m, 1..m]$ drawn from the same alphabet set Σ and 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 trees and suffix array exist [5, 8] for the task of efficient pattern matching. These are essentially 2D counterparts of classic suffix trees and arrays known for traditional 1-dimensional (1D) pattern matching. They work based on linearization of 2D suffixes which would preserve the prefix match property (i.e., every pattern match is a prefix of some suffix).

The main limitation of the suffix trees and the suffix arrays (in 1D) was their space utilization of $O(N \log N)$ bits, where N is the size of the text. This was suboptimal compared to $N \log \sigma$ bits of space, which is information theoretic optimal for the text. With the advent of the field of succinct/compressed data structures, it was possible to develop compressed variants of suffix trees and array based on Burrows-Wheeler Tansform and LF-mapping (or Φ function) [7, 4, 15]. These data structures indeed achieve $O(N \log \sigma)$ bits of space or better. This gives rise to the question: analogous to 1D case, can we design a succinct or compressed index for 2D pattern matching? Can there be a 2D compressed suffix tree? Are there analogues of Burrows–Wheeler Transform or LF-mapping? The problem has been acknowledged for over a decade now and there have been a few attempts at applying Φ function [1] and achieving entropy based compression [10]. However, achieving the complexity breakthrough akin to 1D case has yet to be found.

In this paper, we still do not know how to answer suffix array queries in $O(N \log \sigma)$ bits of space - which would have led to efficient pattern matching. However, for the first time, we show an interesting result that it is indeed possible to compute inverse suffix array (ISA) queries in near compact space in $O(\mathsf{polylog} n)$ time. Our 2D succinct text index design is based on two 1D compressed suffix trees and it takes $O(N \log \log N + N \log \sigma)$ bits of space which is much smaller than its naive design that takes $O(N \log N)$ bits.

Although the main problem is still evasive, this index gives a hope on the existence of a full 2D succinct index with all functionalities similar to that of 1D case.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern Matching, Succinct Data Structures

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2021.60

1 Introduction

In the classical pattern matching problem we are given a text $\mathsf{T}[1..n]$ over an alphabet Σ, which is a finite totally ordered set of size σ and a pattern $\mathsf{P}[1..m]$ drawn from the same alphabet set. The task is to find locations of all occurrences of pattern P in T. This has been a classic field of research since last 50 years and many algorithms were developed to achieve this task in optimal time complexity of $O(n + m)$ [9]. In data structural sense, the problem becomes to index the text so that patterns can be taken as queries. Data structures like suffix trees were proposed for this task which took optimal $O(n)$ (words of) space and optimal

© Dhrumil Patel and Rahul Shah;
licensed under Creative Commons License CC-BY 4.0

32nd International Symposium on Algorithms and Computation (ISAAC 2021).
Editors: Hee-Kap Ahn and Kunihiko Sadakane; Article No. 60; pp. 60:1–60:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$O(m)$ time for the query. It was seen that even though taking theoretically optimal space, the space utilization of suffix trees often times could be 50 times the size of original text data. Space saving structures like suffix arrays were introduced which showed substantial space savings at the cost of slightly worse query times. However, when measured in bits, these still take $O(n \log n)$ bits as against the information theoretic optimal of $O(n \log \sigma)$ bits. FM-Index [4] and compressed suffix array (CSA) [7] were the first to achieve this goal. Introduction of the compressed suffix tree (CST) ensured full-functionalities of suffix trees simulated in compressed space of $O(n \log \sigma)$ bits (or even lower in entropy compressed sense) [15]. This led to the field of compressed text indexing which has seen a myriad of results in last two decades with many positive developments [13].

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 the problems which has proven to be hard in this context is the problem of 2D pattern matching.

In the 2D pattern matching problem, the text is arranged as a matrix $\mathsf{M}[1..n, 1..n]$ and consists of $N = n \times n$ symbols drawn from the alphabet Σ of size σ. The query consists of $m \times m$ square matrix $\mathsf{P}[1..m, 1..m]$ drawn from the same alphabet set Σ and 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 the data structures like suffix trees and suffix array exist [5, 8]. The suffix starting from any location $\mathsf{M}[i, j]$ is the largest square matrix which fits within M and whose top-left corner is $\mathsf{M}[i, j]$. The suffixes can be linearized [5] and indexed using a trie akin to the 1D suffix tree. The problem of designing an index for 2D pattern matching in compact $O(N \log \sigma)$ space (based on suffix trees/arrays BWT or otherwise) has been long open. There were some attempts and partial results [1, 10] but they mainly focused on entropy compression, without first addressing the more fundamental problem of achieving the optimal space complexity (compact space). This gives rise to some fundamental questions: analogous to 1D case, can we design a succinct or compressed index for 2D pattern matching? Can there be a compressed suffix tree?

However, achieving the complexity breakthrough similar to 1D case has yet to be found, in this paper, we present a text index that can answer inverse suffix array (ISA) queries in near compact space 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 *linearization* scheme of Reference [5]. Our 2D succinct text index design is based on two 1D compressed suffix trees, and it takes $O(N \log \log N + N \log \sigma)$ bits of space as compared to previous non-compact space of $O(N \log N)$ bits.

2 Preliminaries

First, we show an overview of the classical pattern matching problem and its associated terminology. Next, we extend the same for the 2D pattern matching problem, where we provide additional definitions associated with the problem.

2.1 Classical Pattern Matching Problem

Let $\mathsf{S} = \{\mathsf{T}[i..n] | 1 \leq i \leq n\}$ 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 S [12, 16, 3, 17]. In the suffix tree, concatenating all the edge labels on a particular root-to-leaf path, we get one of the suffixes in S. In other words, each leaf of ST corresponds to a suffix of T. Additionally, as each suffix $\mathsf{T}[i..n]$ in S is uniquely identified with its starting position i in T, we can map

text positions to leaves of ST. Upon traversal of the leaves from left-to-right, we get suffixes sorted lexicographically, and storing the corresponding text positions in an array gives an indexing data structure called *suffix array* (SA) [11]. Here by $i = \mathsf{SA}[r]$, we mean that the leaf with its corresponding text position i is the rth leftmost leaf (ℓ_r) in ST. In other words, r is the lexicographical order or *rank* of the suffix $\mathsf{T}[i..n]$. 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.

The LF-mapping is the relation between the leaves ℓ_r and $\ell_{r'}$ ($\mathsf{LF}(r) = r'$) such that their corresponding text positions are i and $i-1$ respectively. Formally, LF-mapping is defined in terms of the suffix array as $\mathsf{LF}(r) = \mathsf{SA}^{-1}[\mathsf{SA}[r]-1]$. But the index such as the FM-index efficiently computes the LF-mapping using the *Burrows Wheeler Transform (BWT)* [2] of the original text along with some auxiliary counting data structures. This computation lies at the heart of BWT based text indexes that enables them to answer pattern matching queries without actually storing the costly suffix array and instead replacing it with a *sampled suffix array.*

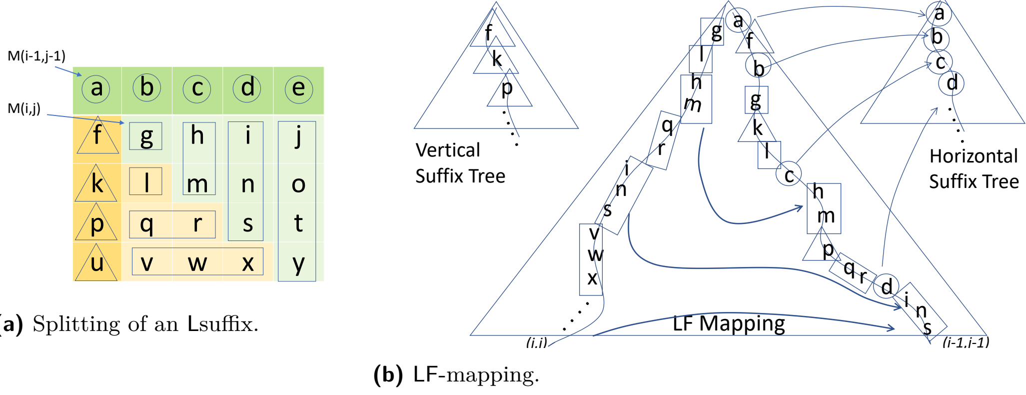
On the contrary, just storing a CSA in itself does not provide all the required functionalities that a full CST provide. Therefore, a CST with full functionalities is needed and is realised using three components: 1) its underlying CSA 2) the *compressed tree topology* that provides navigational operations where each operation takes $O(1)$ time and 3) some auxiliary data structures providing the *longest common prefix* (LCP) information. Moreover, the full list of operations supported by CST is given in the Appendix. Out of which one of the most important operations is to answer inverse suffix array queries i.e. given CSA, an ISA entry can be decoded in $O(\log^{\epsilon} N)$ time for some constant $\epsilon > 0$. Now in the ensuing subsection, we formally go over the 2D pattern matching problem and associated terminology.

2.2 2D Pattern Matching Problem

Let M be a square matrix of dimension $N = n \times n$ where every element $\mathsf{M}[i,j]$ is taken from an alphabet Σ which is a finite totally ordered set of size σ. The query consists of a square pattern $\mathsf{P}[1..m, 1..m]$ also drawn from Σ and the goal is to find all the occurrences of P in M.

In a 1D text, an ith suffix is the largest substring of the text starting from the ith position i.e. $\mathsf{T}[i..n]$. Similarly, this way of defining a *suffix* can be extended to 2D suffixes of a matrix. A 2D suffix $S^{\mathsf{2D}}_{i,j}$ defined for a position (i,j) is the largest square submatrix of M starting at (i,j) position i.e. $\mathsf{M}[i..i+l, j..j+l]$, where $l = n - \max(i,j)$. Giancarlo [6] proposed a way of *linearization* of 2D suffixes such that they follow the constraints of *completeness* and *common prefix property* similar to 1D suffixes. The *completeness* constraint is that every square submatrix of M in the 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 proposed 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 $a_0, a_1, a_2, ..., a_l$ 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 $k \neq 0$ (see Figure 1 for example). Here $\alpha \cdot \beta$ refers to the concatenation of the strings α and β.

Let $\mathsf{S^L}$ be the set of all such Lsuffixes of M. Here $|\mathsf{S^L}| = N$ as there are total N suffixes. Let $\mathsf{ST^L}$ be the compact trie (suffix tree) constructed from Lsuffixes in $\mathsf{S^L}$ (also known as Lsuffix tree). The uncompressed version of $\mathsf{ST^L}$ [5] 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 suffix array ($\mathsf{SA^L}$) [8] for such suffixes also



(a) Splitting of an Lsuffix.

(b) LF-mapping.

Figure 1 Lsuffixes and LF-mapping. a) *Splitting of an Lsuffix*: The characters inside the circle are a part of the horizontal suffix $S^{\mathsf{H}}_{i-1,j-1} = abcde...$ and it resides on the horizontal suffix tree. Similarly, The characters inside the triangle 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 the rectangle i.e. $S^{\mathsf{L}}_{i,j} = g \cdot l \cdot hm \cdot qr \cdot ins \cdot vwx \cdot joty$ and it 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 position (i, j) to that of Lsuffix starting at 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.

requires $\Theta(N \log N)$ bits of space. The suffix array and inverse suffix array is defined in a similar fashion as defined in the linear case. For suffix array, given the 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, inverse suffix array is defined as $\mathsf{ISA}^{\mathsf{L}}[i, j] = r$. Additionally, we introduce the LF-mapping with respect to the 2D case (LF^{L}-mapping) and we define it as follows,

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

Here, $\mathsf{ISA}^{\mathsf{L}}[0, j'] = \mathsf{ISA}^{\mathsf{L}}[i', 0] = \emptyset$. In other words, LF^{L}-mapping operation outputs the rank of the Lsuffix $S^{\mathsf{L}}_{i-1,j-1}$ given the rank of Lsuffix $S^{\mathsf{L}}_{i,j}$ (i.e. it goes diagonally above). Figure 1 shows an example of a particular LF-mapping operation and how new characters get introduced when going from 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 the LF^{L}-mapping is related to the SA^{L}, we introduce a similar mapping for ISA which we call LF-*mapping for ISA* ($\mathsf{LFISA}^{\mathsf{L}}$). We define it as,

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

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 7. In other words, given the position and the rank of the Lsuffix $S^{\mathsf{L}}_{i,j}$, $\mathsf{LFISA}^{\mathsf{L}}$-mapping outputs the rank of the Lsuffix $S^{\mathsf{L}}_{i-1,j-1}$ (diagonally above). Now, in order to compute the value of any $\mathsf{ISA}^{\mathsf{L}}$ entry, as storing the entire $\mathsf{ISA}^{\mathsf{L}}$ 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 $\mathsf{ISA}^{\mathsf{L}}$ 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, $\mathsf{ISA}^{\mathsf{L}}$ 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.

Till now, succinct versions of 2D text indexes have been evasive because the intuition behind the pattern matching in 1D case does not extend directly to the 2D case due to the non trivial nature of 2D LF-mapping. However, this does not restrict us from asking the fundamental question as to whether it is possible to design a compact text index for matrices. In this paper, we propose the design of a text index that can atleast answer inverse suffix array queries in near compact space using $\mathsf{LFISA}^{\mathsf{L}}$-mapping. Although, a solution to the main question evades us, this is a ray of hope. Now, the following theorem states the objective of the paper more formally as,

▶ **Theorem 1.** *The text index for matrix* M *of size* $N = n \times n$ *can be encoded in* $O(N\log\sigma + N\log\log N)$*-bit space and any 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. See Sections 8.1 and 8.2 for the proof. ◀

Our approach. The intuition here is that we split the Lsuffix for which we need the ISA value into three subsequences, and thereby solve the problem for each subsequence to eventually solve for the main Lsuffix. We discuss this splitting in detail in the later section. But in order to understand this dividing strategy, first we define what we call *horizontal* and *vertical suffixes* (or in short Hsuffix and Vsuffix respectively) and also how they relate to Lsuffixes.

3 Horizontal and Vertical Suffixes

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 text 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 such Vsuffixes of T^{V}. From the context of M, Hsuffix and Vsuffix starting from $\mathsf{M}[i,j]$ are written as

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

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

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 [15]. Their full functionalities are provided in the Appendix. Next, we relate all these defined suffixes.

4 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 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 us denote h_k and v_k as the $(k+1)^{th}$ 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 $a_0, a_1, a_2, ..., a_l$ 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 $b_0, b_1, b_2, ..., b_{l-1}$ where $b_0 = \mathsf{M}[i+1, j+1]$ and $b_k = \mathsf{M}[(i+1)+k, (j+1)..(j+1)+k-1]\mathsf{M}[(i+1)..(i+1)+k, (j+1)+k]$ when $k \neq 0$. For simplicity we break each a_k and b_k into two parts as follows,

$$a_k^{'} = \mathsf{M}[i+k, j..j+k-1]$$
$$a_k^{''} = \mathsf{M}[i..i+k, j+k]$$
$$b_k^{'} = \mathsf{M}[(i+1)+k, (j+1)..(j+1)+k-1]$$
$$b_k^{''} = \mathsf{M}[(i+1)..(i+1)+k, (j+1)+k]$$

We can write a_k in terms of h_k and v_k b_k as follows,

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

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

Now, given a_k we can get $v_{k-1}, b_{k-1}^{'}, h_k, b_{k-1}^{''}$ as v_{k-1} and h_k are characters and $b_{k-1}^{'}$ and $b_{k-1}^{''}$ are the 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 breaks down a_k into its constituents. Now 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 entire Lsuffix $S^{\mathsf{L}}_{i,j}$ by $\mathsf{hc}(S^{\mathsf{L}}_{i,j}) = h_0h_1h_2...h_l$. Similarly, we denote the *vertical component* by $\mathsf{vc}(S^{\mathsf{L}}_{i,j}) = v_0v_1v_2...v_{l-1}$ and the *square component* by $\mathsf{sc}(S^{\mathsf{L}}_{i,j}) = b_0b_1b_2...b_{l-1}$. Likewise, we can define the same for any prefix pf of the Lsuffix $S^{\mathsf{L}}_{i,j}$. We can state the following fact about the relation between the length of the three components of the prefix pf of $S^{\mathsf{L}}_{i,j}$ and its length. Here, by length, we mean the length of the string.

▶ **Fact 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. Next, we go over some of the basic terminologies of a suffix tree that will be needed in understanding the construction stage of our text index.

5 Terminology of a Suffix Tree (ST)

A suffix tree is an edge-labelled compact trie. We call any character on the edge of the suffix tree be represented as a *point*. Given any point c on the ST, $\mathsf{string}(c)$ represents the concatenation of all the characters from root to that point (including c) along the root to c path of ST. The string depth of a point c on a path of ST is given by the length of $\mathsf{string}(c)$ i.e. $\mathsf{depth}(c) = \mathsf{length}(\mathsf{string}(c))$. A node of the ST is also a point as that node is represented by the character just above it. The locus u of a point c is the highest node of ST such that $\mathsf{string}(c)$ is the prefix of $\mathsf{string}(u)$ (lets denote it as $u = \mathsf{locus}(c)$). Now, we define whether

a point c is marked or not as $\mathsf{marked}(c) = 1$ or 0 respectively. The leftmost and rightmost leaves in the subtree of a particular point c are given as $\mathsf{lleaf}(c)$ and $\mathsf{rleaf}(c)$ respectively. Here $\mathsf{lleaf}(c)$ and $\mathsf{rleaf}(c)$ give the leaf rank from left in ST. In general, we denote r^{th} leftmost leaf of ST as ℓ_r. Let $\mathsf{lca}(c_1, c_2)$ be the *lowest common ancestor* of points c_1 and c_2. The lowest common ancestor as the name suggests is the common ancestor node of two points and is the farthest from the root.

6 Computing LFISA-mapping in time $O((\log N/\log\log N)^3)$ using Compact Space

Just to recall, we have three suffix trees based on three different types of suffix definitions as shown before, i.e. $\mathsf{ST^L}$, $\mathsf{ST^H}$ and $\mathsf{ST^V}$. Here, we store $\mathsf{ST^H}$ and $\mathsf{ST^V}$ as compressed suffix trees (CST) [15] with full functionalities (See Theorem 10 and 11 in Appendix) and they together 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 $\mathsf{ST^L}$ 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 Theorem 10 in Appendix). In the ensuing subsection, firstly we show a scheme of marking some relevant points on these trees (construction stage) and then explain how this will help in computing LFISA-mapping.

6.1 Marking Scheme and Mapping

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

▶ **Definition 3** (Marked Ancestor). *A marked node v^{L} is the marked ancestor of a leaf ℓ if v^{L} lies on the path from root to leaf ℓ in the suffix tree.*

▶ **Definition 4** (Lowest Marked Ancestor). *A node v^{L} is the lowest marked ancestor of the leaf ℓ if it is the lowest (one with the maximum string depth) among all the marked ancestors of ℓ.*

▶ **Definition 5** (Cover). *A node v^{L} is the cover of the leaf ℓ if it is the lowest marked ancestor of ℓ.*

▶ **Definition 6** ($\mathsf{coveredby}(v^{\mathsf{L}})$ set). *A $\mathsf{coveredby}(v^{\mathsf{L}})$ set is the set of the leaves for which v^{L} is the cover.*

As mentioned before in Section 4 showcasing the splitting of an Lsuffix, given a marked node v^{L}, its associated string i.e. $\mathsf{string}(v^{\mathsf{L}})$ can be split 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 $\mathsf{ST^L}$, we mark a point p^{H} in $\mathsf{ST^H}$ 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 $\mathsf{ST^V}$ and $\mathsf{ST^L}$ respectively. We call them the *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 2). We repeat the above process for every marked node on v^{L} in $\mathsf{ST^L}$.

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. Formally, we have $|\mathsf{MP}^{\mathsf{H}}|$, $|\mathsf{MP}^{\mathsf{V}}|$ and $|\mathsf{MP}^{\mathsf{L}}|$ as bounded by $O(\frac{N}{\log^3 N})$, where $|X|$ is the cardinality of the set X. Due to this one-to-one correspondence between a marked node and the 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 which come from the 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 the text and contruct data structures that will take near compact space and achieve this task in $O(\mathsf{polylog} N)$ time. The main step in this is 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, and thereafter we outline the pseudocode for the same as $\texttt{LFISA}^{\mathsf{L}}(\texttt{i}, \texttt{j}, \texttt{ISA}^{\mathsf{L}}(\texttt{i}, \texttt{j}))$ in the subsection 6.2.3.

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

In the section, we show the details for the evaluation of $\mathsf{LFISA}^{\mathsf{L}}(i, j, \mathsf{ISA}^{\mathsf{L}}[i, j])$ given the matrix position (i, j) and $\mathsf{ISA}^{\mathsf{L}}[i, j]$. Firstly, using the $\mathsf{inverse}(\cdot)$ function of ST^{H} and ST^{V} (See Theorem 10 in Appendix), we evaluate the inverse suffix array values $\mathsf{ISA}^{\mathsf{H}}[i-1, j-1]$ and $\mathsf{ISA}^{\mathsf{V}}[i, j-1]$ respectively. For simplicity, let $\mathsf{ISA}^{\mathsf{L}}[i, j] = r, \mathsf{ISA}^{\mathsf{H}}[i-1, j-1] = h, \mathsf{ISA}^{\mathsf{V}}[i, j-1] = v, \mathsf{ISA}^{\mathsf{L}}[i-1, j-1] = \mathsf{LFISA}^{\mathsf{L}}(i, j, \mathsf{ISA}^{\mathsf{L}}[i, j]) = s$.

As inverse suffix array values are related to the leaves of the 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 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 in the latter subsections.

Given (h, v, r), 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,

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

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

Lemma 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 as. Hence, we call the above query as *lowest marked ancestor query*.

▶ **Lemma 7.** *The marked node v^{L}_{max} in ST^{L} is the lowest marked ancestor (or cover) of the leaf ℓ_s.*

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

Furthermore, as the $\mathsf{string}(p^{\mathsf{H}})$, $\mathsf{string}(p^{\mathsf{V}})$ and $\mathsf{string}(p^{\mathsf{L}})$ are the horizontal, vertical and square components of the $\mathsf{string}(v^{\mathsf{L}})$ respectively (as per 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, the triplet node 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. The same is true for $\forall v^{\mathsf{L}} \in A$.

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 $v^{\mathsf{L}} \in A$, it is the *lowest marked ancestor or cover* of ℓ_s. ◀

Now as we are interested in obtaining cover 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 8. The details concerning this reduction is discussed in the next subsection. Furthermore, after finding the cover v^{L}_{max}, in order to uniquely go to the correct leaf ℓ_s we store additional augmenting information [discussed in latter subsection]. This shows the computation of an $\mathsf{LFISA}^{\mathsf{L}}$ operation. The time or query complexity of such an operation is discussed in the Section 7.

6.2.1 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 *3D stabbing-max* query. In [14], the authors proves the following theorem,

▶ **Theorem 8** ([14]). *Given a set I of n 3D rectangles in $\mathbb{R}^3$, where each rectangle rec has a weight $w(rec)$ associated to it, finding a rectangle with maximum weight containing (or stabbed by) the 3D query point q can be done in $O((\frac{\log n}{\log \log n})^3)$ time using a data structure occupying $O(n(\frac{\log n}{\log \log n})^2)$ space.*

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

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

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

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

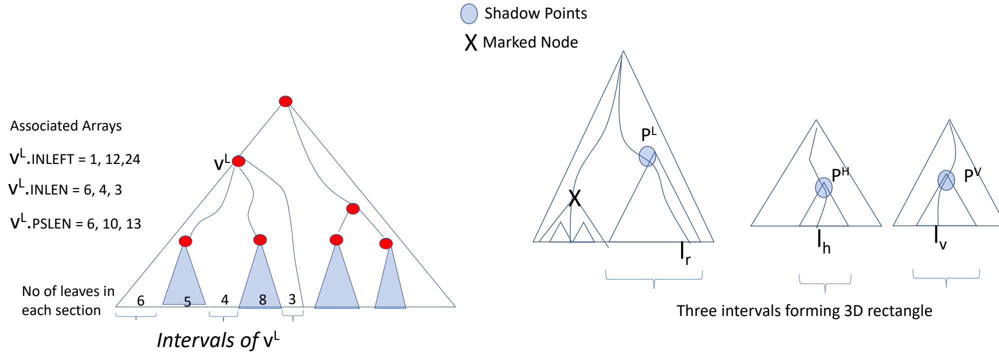


Figure 2 For a particular marked node v^{L} in ST^{L} (shown in red color), the array INLEFT corresponding to v^{L} stores the start of its associated intervals. Likewise, the array INLEN stores the length of such intervals and the array PSLEN is the prefix-sum array of INLEN. The points $(p^{\mathsf{L}}, p^{\mathsf{H}}, p^{\mathsf{V}})$ are the shadow points of v^{L} (shown as X in the figure shown on the right side).

6.2.2 Augmenting Information for getting ℓ_s from its Cover

In this section, we explain the procedure of obtaining the correct leaf ℓ_s from its cover v^{L} by storing the leaf's rank q (say). Here, we define the task for this section as: Given q and v^{L}, find the qth leftmost leaf in $\mathsf{coveredby}(v^{\mathsf{L}})$ (See Definition 6 of $\mathsf{coveredby}(v^{\mathsf{L}})$).

Now the challenge lies due to the fact that v^{L} may have multiple marked nodes in its subtree and due to that there may be leaves in its subtree whose lowest marked ancestor or cover is not v^{L}. 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 it 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 denotes covered intervals.

Lemma 9 proves that the total number of such intervals is bounded by $O(\sigma)$, i.e. $k = O(\sigma)$. Additionally, it establishes that the total number of leaves for which v^{L} is the cover is bounded by $\sum_{a=1}^{k} |I_a| = O(k \log^3 N) = O(\sigma \log^3 N)$ where $|I|$ is the length of the interval I. Furthermore, as there is a one-to-one correspondence between each marked node and a rectangle as shown before, we store the augmenting information for each rectangle rather than storing it explicitly for the marked node. Let the rectangle associated with v^{L} be denoted as *rec*. Therefore, let $CI_{rec} = CI$.

▶ **Lemma 9.** *The total number of intervals in CI_{rec} is bounded by $O(\sigma)$ and the total number of leaves for which any marked node (here v^{L}) is the cover is bounded by $O(\sigma \log^3 N)$.*

Proof. Let c^{L} be one of the child nodes of v^{L}. 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} 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} is the lowest common ancestor (lca) of its leftmost and rightmost leaves (See marking scheme for details). But this is not possible as for v^{L} to be the lca, the leftmost and rightmost leaves of that grouping interval should exist on two separate downward branches of v^{L}. 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}.

This means I_i and I_j are consecutive intervals. Therefore, the subtree of a child node of v^{L} overlaps with at most 2 consecutive intervals in CI_{rec}. Furthermore, there are at most σ child nodes of v^{L}. Hence, the total number of intervals in CI_{rec} is bounded by $O(\sigma)$.

Secondly, there are at most $O(\sigma)$ grouping intervals under the subtree of v^{L} for which v^{L} is the lca of its leftmost and rightmost leaves, as each grouping interval need to span over two separate downward branches of v^{L} for v^{L} 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} is the lowest marked ancestor or the cover is bounded by the same factor i.e. $O(\sigma \log^3 N)$. ◀

As the set of leaves for which v^{L} is the cover, is divided into contiguous intervals of leaves (as shown above), to go from the cover v^{L} to the output leaf ℓ_s, first we store some information to retrieve which interval that leaf belongs to and then where exactly that leaf is inside that interval.

For each marked node (here v^{L} or its associated rec) firstly we store 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 2 for example). Now as 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 $\mathsf{miniISA}^{\mathsf{L}}[\cdot,\cdot]$, where we store just 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 bounded by $O(\sigma \log^3 N)$ (Lemma 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 $s = \mathsf{INLEFT}_{rec}[e] + (q - \mathsf{PSLEN}_{rec}[e])$.

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

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

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

```
1: h = ST^H.inverse(i, j)
2: v = ST^V.inverse(i, j)
3: s = ISA^L[i, j]
4: rec = 3d_stabbing_max(h, v, s)
5: q = miniISA^L[i, j]
6: e = binary_search(PSLEN_rec, q)
7: s = INLEFT_rec[e] + (q − PSLEN_rec[e])
8: return s
```

7 Space and Time Complexity Analysis

7.1 Space Complexity

After the end of the construction phase, we have three suffix trees in our index based on three different types of suffix definitions, along with some auxiliary structures that are actually stored. The horizontal and vertical suffix trees i.e. ST^{H} and ST^{V} are stored as compact suffix trees (See Theorem 10 and 11 in Appendix) 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 (See Theorem 10 in Appendix). This compressed topology provides navigational functionalities, and overall it occupies $4N + o(N)$ bits of space.

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

Each 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 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 for all the auxiliary structures is bounded by $O(N/\log^2 N) = o(N)$ bits.

If there are t rectangles, the data structure for stabbing-max query takes $O(t(\log t/\log\log t)^2)$ [14] which is $O(t \log^2 t)$ space. By taking $t = O(N/\log^3 N)$, we get that stabbing-max data structure takes $O(N/\log N)$ words of space which is bounded by $O(N)$ 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 any entry in $\mathsf{miniISA}^{\mathsf{L}}$ writes a position of the desired leaf among at most $\sigma \log^3 N$ leaves which have the same lowest marked node. Thus, in total we get $O(N \log \sigma + N \log\log N)$ bits 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, in total it takes $O(N)$-bits of space.

After summing up all five parts that are considered, we get $O(N \log \sigma) + o(N) + O(N) + O(N \log \sigma + N \log\log N) + O(N)$ bits. This simplifies to $O(N \log \sigma + N \log\log N)$ bits as claimed in Theorem 1.

7.2 Time Complexity

For the time complexity of query evaluation, as a key component, we first focus on computing $\mathsf{LFISA}^{\mathsf{L}}$-mapping operation. We follow the pseudocode step by step for this. The first two steps take $t_{inverse}$ as given by CST which is $O(\log^{\epsilon} n)$ (See Theorem 11 in Appendix). The third step is constant time since the value is provided as a part of the function. The main 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, our dominating and main query bound for $\mathsf{LFISA}^{\mathsf{L}}$-mapping operation is $O((\log N/\log\log N)^3)$. Finally, considering that our query algorithm for $\mathsf{ISA}^{\mathsf{L}}$ can have at most $\log N$ applications of $\mathsf{LFISA}^{\mathsf{L}}$, we get our query-time bound as $O(\log^4 N/(\log\log N)^3)$ (as claimed in Theorem 1).

8 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 which 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.

References

1. Jeffrey Scott Vitter Ankur Gupta, Roberto Grossi. Entropy-compressed indexes for multidimensional pattern matching. In *DIMACS working group on Burrows-Wheeler Transform*, 2004.
2. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation (now part of Hewlett-Packard, Palo Alto, CA), 1994.
3. Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.
4. 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"*. doi:10.1145/1082036.1082039.
5. Raffaele Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM J. Comput.*, 24(3):520–562, 1995. doi:10.1137/S0097539792231982.
6. 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.
7. 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.* doi:10.1137/S0097539702402354.
8. Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park. Constructing suffix arrays for multidimensional 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. doi:10.1007/BFb0030786.
9. Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.
10. 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. doi:10.1109/DCC.2008.47.
11. Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.
12. Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.
13. Gonzalo Navarro. *Compact Data Structures – A Practical Approach*. Cambridge University Press, 2016. URL: http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/compact-data-structures-practical-approach?format=HB.
14. 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. doi:10.1007/978-3-642-25591-5_19.

15 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/s00224-006-1198-x.

16 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

17 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. doi:10.1109/SWAT.1973.13.

A Appendix

A fully functional compact/compressed suffix tree is realized using three components, 1) its *compressed tree topology* that supports navigational functionalities [See Theorem 10] 2) the *compressed suffix array* [See Theorem 11] and 3) auxiliary data structures that supports *longest common prefix* (LCP) information.

▶ **Theorem 10** (Fully-Functional Succinct Suffix Tree [15]). *The topology of a suffix tree can be encoded in $4N + o(N)$ bits to support the following operations in $O(1)$ time.*

- pre-order(u)/post-order(u)*: pre-order/post-order rank of node u*
- parent(u)*: parent of node u*
- nodeDepth(u)*: number of edges on the path from the root to u*
- child(u, q)*: qth leftmost child of node u*
- sibRank(u)*: number of children of* parent(u) *to the left of u*
- lca(u, v)*: lowest common ancestor (LCA) of two nodes u and v*
- lleaf(u)/rleaf(u)*: leftmost/rightmost leaf in the subtree of u*
- levelAncestor(u, d)*: ancestor of u such that* nodeDepth$(u) = d$

▶ **Theorem 11** (Compressed Suffix Array [15]). *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.*

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