Two-Dimensional Longest Common Extension Queries in Compact Space

Arnab Ganguly
University of Wisconsin, Whitewater, WI, USA

Daniel Gibney
University of Texas at Dallas, TX, USA

Rahul Shah
Louisiana State University, Baton Rouge, LA, USA

Sharma V. Thankachan
North Carolina State University, Raleigh, NC, USA

Abstract

For a length n text over an alphabet of size σ, we can encode the suffix tree data structure in $\mathcal{O}(n \log \sigma)$ bits of space. It supports suffix array (SA), inverse suffix array (ISA), and longest common extension (LCE) queries in $\mathcal{O}(\log_\sigma^\epsilon n)$ time, which enables efficient pattern matching; here $\epsilon > 0$ is an arbitrarily small constant. Further improvements are possible for LCE queries, where $\mathcal{O}(1)$ time queries can be achieved using an index of space $\mathcal{O}(n \log \sigma)$ bits. However, compactly indexing a two-dimensional text (i.e., an $n \times n$ matrix) has been a major open problem. We show progress in this direction by first presenting an $\mathcal{O}(n^2 \log \sigma)$-bit structure supporting LCE queries in near $\mathcal{O}((\log_\sigma n)^{2/3})$ time. We then present an $\mathcal{O}(n^2 \log \sigma + n^2 \log \log n)$-bit structure supporting ISA queries in near $\mathcal{O}(\log n \cdot (\log_\sigma n)^{2/3})$ time. Within a similar space, achieving SA queries in poly-logarithmic (even strongly sub-linear) time is a significant challenge. However, our $\mathcal{O}(n^2 \log \sigma + n^2 \log \log n)$-bit structure can support SA queries in $\mathcal{O}(n^2/(\sigma \log n)^c)$ time, where c is an arbitrarily large constant, which enables pattern matching in time faster than what is possible without preprocessing.

We then design a repetition-aware data structure. The δ_{2D} compressibility measure for two-dimensional texts was recently introduced by Carfagna and Manzini [SPIRE 2023]. The measure ranges from 1 to n^2, with smaller δ_{2D} indicating a highly compressible two-dimensional text. The current data structure utilizing δ_{2D} allows only element access. We obtain the first structure based on δ_{2D} for LCE queries. It takes $\tilde{\mathcal{O}}(n^{5/3} + n^{8/5}\delta_{2D}^{1/5})$ space and answers queries in $\mathcal{O}(\log n)$ time.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String matching, text indexing, two-dimensional text

Digital Object Identifier 10.4230/LIPIcs.STACS.2025.38

Funding Supported by the US National Science Foundation (NSF) under Grant Numbers 2315822 (S Thankachan) and 2137057 (R Shah).

1 Introduction

A two-dimensional text $T[0\,..\,n)[0\,..\,n)$ can be viewed as an $n \times n$ matrix, where each entry is a character from an alphabet set Σ of size σ. Data structures for two-dimensional texts have been studied for decades. In particular, there has been extensive work on generalizing suffix trees [16, 17, 23] and suffix arrays [16, 22] to 2D text. These data structures, although capable of answering most queries in optimal (or near optimal) time, require $\mathcal{O}(n^2)$ words, or $\mathcal{O}(n^2 \log n)$ bits, of space.

On the other hand, in the case of 1D texts of length n, there exist data structures with the same functionality as suffix trees/arrays but requiring only $\mathcal{O}(n \log \sigma)$ bits of space [18, 32], or even smaller in the case where the text is compressible [11, 21]. This is true even for

© Arnab Ganguly, Daniel Gibney, Rahul Shah, and Sharma V. Thankachan;
licensed under Creative Commons License CC-BY 4.0

42nd International Symposium on Theoretical Aspects of Computer Science (STACS 2025).
Editors: Olaf Beyersdorff, Michał Pilipczuk, Elaine Pimentel, and Nguyễn Kim Thắng;
Article No. 38; pp. 38:1–38:17

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

some variants of suffix trees, such as parameterized [14, 13] and order-isomorphic [12] suffix trees [33]. The query times of these space-efficient versions are often polylogarithmic, with the exception of LCE queries, for which Kempa and Kociumaka demonstrated that the query time can be made constant [19]. For 2D texts, the only results known in this direction include a data structure by Patel and Shah that uses $\mathcal{O}(n^2 \log \sigma + n^2 \log\log n)$ bits and supports inverse suffix array (ISA) queries in $\mathcal{O}(\log^4 n/(\log\log n)^3)$ time [28]. In this work, we make further progress in this direction. In particular, we focus on space-efficient data structures for longest common extension (LCE) queries in the 2D setting. The problem is formally defined as follows:

▶ **Problem 1** (2D LCE). *Preprocess a 2D text $T[0\mathinner{.\,.}n)[0\mathinner{.\,.}n)$ over an alphabet Σ of size σ into a data structure that can answer 2D LCE queries efficiently. A 2D LCE query consists of points (i_1, j_1), (i_2, j_2) and asks to return the largest L such that $T[i_1\mathinner{.\,.}i_1 + L)[j_1\mathinner{.\,.}j_1 + L)$ and $T[i_2\mathinner{.\,.}i_2 + L)[j_2\mathinner{.\,.}j_2 + L)$ are matching square submatrices of T.*

A 2D suffix tree of size $\mathcal{O}(n^2 \log n)$ bits can answer LCE queries in constant time. Our first result is an LCE data structure that occupies $\mathcal{O}(n^2 \log \sigma)$ bits of space.

▶ **Theorem 1.** *By maintaining an $\mathcal{O}(n^2 \log \sigma)$-bit data structure, we can answer 2D LCE queries in $\mathcal{O}((\log_\sigma n)^{2/3} \cdot (\log\log_\sigma n)^{5/3})$ time.*

Turning now to highly compressible 2D texts, we consider repetition-aware compression measures. The δ measure is an important and well-studied compressibility measure for 1D text [26]. Only recently has it been extended to 2D text by Carfagna and Manzini with a δ_{2D}-measure [5]. They demonstrate that the data structure of Brisaboa et al. [3] occupies $\mathcal{O}((\delta_{2D} + \sqrt{n\delta_{2D}}) \log \frac{n \log \sigma}{\sqrt{\delta_{2D}} \log n})$ space. However, this data structure only supports access to the elements of T. We provide the first repetition-aware data structure supporting the more advanced LCE queries. Note that the measure δ_{2D} ranges from 1 to n^2, with a smaller δ_{2D} value implying higher compressibility.

▶ **Theorem 2.** *By maintaining an $\mathcal{O}((n^{5/3} + n^{8/5}\delta_{2D}^{1/5}) \log \beta)$ word data structure, we can answer 2D LCE queries in $\mathcal{O}(1 + \log \beta)$ time, where β is always $\mathcal{O}(n)$ and goes to $\mathcal{O}(1)$ as δ_{2D} approaches n^2. In particular,*

$$\beta = \begin{cases} n & \text{if } \delta_{2D} < n^{9/5} \\ n^{9/5}/\delta_{2D}^{9/10} & \text{if } \delta_{2D} \geq n^{9/5}. \end{cases}$$

When $\delta_{2D} = \Theta(n^2)$, our data structure takes $\mathcal{O}(n^2)$ words of space and answers LCE queries in $\mathcal{O}(1)$ time. When $\delta_{2D} = o(n^2)$, the space becomes $o(n^2)$ and LCE queries are answered in logarithmic time. Our approach builds off many of the same techniques as our compact index but also introduces a matrix representation of the leaves of a truncated suffix tree. We call this a *macro-matrix.* We prove that if the original 2D text is compressible, then this macro-matrix remains compressible for appropriately chosen parameters. This is then combined with the data structure of Brisaboa et al. [3] to achieve Theorem 2.

As the first steps towards obtaining the other functionalities of the suffix tree, we apply our 2D LCE query structure from Theorem 1 to get the following results. Definitions of suffix array (SA) and inverse suffix array (ISA) are deferred to Section 1.1.

The following theorem significantly improves on the results by Patel and Shah [28].

▶ **Theorem 3** (2D ISA queries). *By maintaining an $\mathcal{O}(n^2 \log \sigma + n^2 \log\log n)$-bit data structure, we can answer inverse suffix array queries in $\mathcal{O}(\log n \cdot (\log_\sigma n)^{2/3} \cdot (\log\log_\sigma n)^{5/3})$ time.*

We also provide the first known results regarding a nearly compact index for 2D suffix array queries.

▶ **Theorem 4** (2D SA queries). *By maintaining an $\mathcal{O}(n^2 \log \sigma + n^2 \log\log n)$-bit data structure, we can answer suffix array (SA) queries in $\mathcal{O}(n^2/(\sigma \log n)^c)$ time, where c is an arbitrarily large constant fixed at the time of construction.*

A fundamental problem is to find all submatrices of T that match with a given square pattern $P[0\,..\,m)[0\,..\,m)$. After building the 2D suffix tree, given P as a query, the number of occurrences of P (denoted by *occ*) can be obtained in $\mathcal{O}(m^2)$ time, and all occurrences can be reported in $\mathcal{O}(m^2 + occ)$ time. Our result, which uses a smaller index, is the following.

▶ **Theorem 5** (PM queries). *By maintaining an $\mathcal{O}(n^2 \log \sigma + n^2 \log\log n)$-bit data structure, we can count the occurrences of an $m \times m$ query pattern in time $\mathcal{O}(m^2 + n^2/(\sigma \log n)^c)$ and report all occurrences in time $\mathcal{O}(m^2 + occ + n^2/(\sigma \log n)^c)$, where c is an arbitrarily large constant fixed at the time of construction.*

Although the time complexities in Theorems 4 and 5 are far from satisfactory, these are the first results demonstrating subquadratic query times in compact space are possible for 2D SA and PM queries.

1.1 Preliminaries

Notation and Strings. We denote the interval i, $i+1$, ..., j with $[i\,..\,j]$ and the interval i, $i+1$, ..., $j-1$, with $[i\,..\,j)$. For a string S of length n we use $S[i]$ to refer to i^{th} character, $i \in [0\,..\,n)$. We use $S_1 \cdot S_2$ to denote the concatenation of two strings S_1 and S_2. For notation, $S[i\,..\,j] = S[i] \cdot S[i+1] \cdot \ldots \cdot S[j]$, $S[i\,..\,j) = S[i] \cdot S[i+1] \cdot \cdots \cdot S[j-1]$, and $S[i\,..] = S[i\,..\,n)$. Arrays and strings are zero-indexed throughout this work.

For a single string $S[0\,..\,n)$ and $i, j \in [0\,..\,n)$, $\mathrm{LCE}(i,j)$ is defined as the length of the longest common prefix of $S[i\,..]$ and $S[j\,..]$. In the case of two strings, $S_1[0\,..\,n_1)$ and $S_2[0\,..\,n_2)$, we overload the notation so that for $i \in [0\,..\,n_1)$, $j \in [0\,..\,n_2)$, $\mathrm{LCE}(i,j)$ is the length of the longest common prefix of $S_1[i\,..]$ and $S_2[j\,..]$. For a given string S, the suffix tree [34] is a compact trie of all suffixes of S with leaves ordered according to the lexicographic rank of the corresponding suffixes. The classical suffix tree takes $\mathcal{O}(n)$ words of space and can be constructed in $\mathcal{O}(n)$ time for polynomially sized integer alphabets [9]. The suffix array $\mathrm{SA}[0\,..\,n)$ of a string $S[0\,..\,n)$ is the unique array such that $S[\mathrm{SA}[i]\,..]$ is the ith smallest suffix lexicographically. The inverse suffix array $\mathrm{ISA}[0\,..\,n)$ is the unique array such that $\mathrm{ISA}[\mathrm{SA}[i]] = i$, or equivalently, $\mathrm{ISA}[i]$ gives the lexicographic rank of $S[i\,..]$. The suffix tree can answer LCE queries in $\mathcal{O}(1)$ time. We call a compact trie with lexicographically ordered leaves for a subset of suffixes a *sparse suffix tree*. Observe that the number of nodes in a sparse suffix tree remains proportional to the number of suffixes it is built from.

We will utilize the following result by Kempa and Kociumaka, which provides an LCE data structure smaller than a classical suffix tree.

▶ **Lemma 6** ([19]). *1D LCE queries on a text $S[0\,..\,n)$ over an alphabet set $\Sigma = [0\,..\,\sigma)$ can be answered in $\mathcal{O}(1)$ time by maintaining a data structure of size $\mathcal{O}(n \log \sigma)$ bits.*

The next result by Bille et al. allows for a trade-off between space and query time. We will utilize it in Section 2.2.

▶ **Lemma 7** ([1]). *Suppose we have the text $S[0\,..\,n)$ as read-only, such that we can determine the lexicographic order of any of its two characters in constant time. Then we can answer 1D LCE queries on S in time $\mathcal{O}(\tau)$ by maintaining an $\mathcal{O}(n/\tau)$ words of space auxiliary structure, where $1 \le \tau \le n$ is any parameter fixed at the time of construction.*

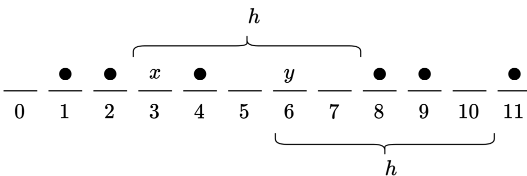

Figure 1 An example d-cover for $n = 12$ and $d = 7$. Here the difference cover used is $\mathcal{D} = \{1, 2, 4\}$, resulting in a d-cover $\mathcal{C} = \{1, 2, 4, 8, 9, 11\}$ (elements indicated with '•') and a lookup table $A = [1, 1, 2, 1, 4, 4, 2]$. For the positions $x = 3$ and $y = 6$, we have $h = A[(6-3) \mod 7] - 3 \mod 7 \equiv 5$. Observe that $3 + 5, 6 + 5 \in \mathcal{C}$.

d-Covers. A d*-cover* of an interval $[0\,..\,n)$ is a subset of positions, denoted by $\mathcal{C}$, such that for any $x \in [0\,..\,n-d)$ and $y \in [0\,..\,n-d)$ there exists $h \in [0\,..\,d)$ where $x + h, y + h \in \mathcal{C}$. It was shown by Burkhardt and Kärkkäinen that there exists a d-cover of size $\mathcal{O}(n/\sqrt{d})$ that can be computed in $\mathcal{O}(n/\sqrt{d})$ time [4]. d-Covers have been used previously for LCE queries in the 1D case by Gawrychowski et al. [15] and Bille et al. [2]. Since we need a small data structure that lets us find an h value as described above in constant time, we briefly outline the construction given in [4].

A *difference cover* modulo d is a subset $\mathcal{D} \subseteq \{0, 1, \ldots, d-1\}$ where for all $w \in \{0, 1, \ldots, d-1\}$ there exist $u, v \in \mathcal{D}$ such that $w \equiv u - v \mod d$. Colbourn and Ling showed there exists $\mathcal{D}$ such that $|\mathcal{D}| = \Theta(\sqrt{d})$ [8]. A d-cover $\mathcal{C}$ is constructed from a difference cover $\mathcal{D}$ as follows: For $j \in [0\,..\,n)$, if $(j \mod d) \in \mathcal{D}$, then j is added to $\mathcal{C}$. We also build a look-up table A of size d such that for all $i \in \{0, 1, \ldots, d-1\}$ both $A[i]$ and $(A[i] + i) \mod d$ are in $\mathcal{D}$. This is always possible, thanks to the definition of the difference cover. See Figure 1.

▶ **Lemma 8** ([4])**.** *For a d-cover $\mathcal{C}$ of an interval $[0\,..\,n)$, there exists a data structure of size $\mathcal{O}(d)$ that given $x, y \in [0\,..\,n-d)$, outputs an $h \in [0\,..\,d)$ such that $x + h, y + h \in \mathcal{C}$ in $\mathcal{O}(1)$ time.*

Proof. We maintain the $\mathcal{O}(d)$ space look-up table A as described above. We assume without loss of generality, $y \geq x$. Let $h := (A[(y - x) \mod d] - x) \mod d$. Observe that

$$x + h \equiv A[(y - x) \mod d] \mod d.$$

Hence, $(x + h \mod d) \in \mathcal{D}$ and $x + h \in \mathcal{C}$. Also,

$$y + h \equiv A[(y - x) \mod d] + (y - x) \mod d.$$

Hence, $(y + h \mod d) \in \mathcal{D}$ and $y + h \in \mathcal{C}$. ◀

2D Suffix Trees and 2D Suffix Arrays. We utilize the generalization of suffix trees to 2D texts presented by Giancarlo [16]. This suffix tree is created from the *Lstrings* of the 2D text T. LStrings are over an alphabet $\cup_{i=1}^{n} \Sigma^{2i-1}$. For a position $(i, j) \in [0\,..\,n)^2$ the suffix $T[i\,..][j\,..]$ is $a_0 \cdot a_1 \cdot \ldots \cdot a_l$ where $l = n - \max(i, j)$ and $a_0 = T[i][j]$ and $a_k = T[i + k][j\,..\,j + k) \cdot T[i\,..\,i + k][j + k]$ for $k > 0$. See Figure 2. The characters are maintained implicitly as references to T, resulting in the 2D suffix tree over all suffixes $T[i\,..][j\,..]$, $(i, j) \in [0, n)^2$ occuping $\mathcal{O}(n^2)$ words of space. Once constructed, the 2D suffix tree allows us to find the LCE of two positions in $\mathcal{O}(1)$ time through a lowest common ancestor (LCA) query. The 2D suffix tree also enables pattern matching in optimal $\mathcal{O}(m^2 + occ)$ time.

T	0	1	2	3	4
0	a	a	b	b	\$
1	a	b	b	c	\$
2	b	b	a	a	\$
3	b	c	a	b	\$
4	\$	\$	\$	\$	\$

suffix starting at $(0,0)$: $a \cdot aab \cdot bbbba \cdot bcabcab \cdot \$\$\$\$\$\$\$\$\$$
suffix starting at $(0,1)$: $a \cdot bbb \cdot babca \cdot cab\$\$\$\$$
suffix starting at $(1,0)$: $a \cdot bbb \cdot bcbaa \cdot \$\$\$cab\$$

Figure 2 An example 2D text and the suffixes starting at positions $(0,0)$, $(0,1)$, $(1,0)$. The "$\cdot$" denotes concatenation, and consecutive symbols without "$\cdot$" between them are treated as a single character.

The order between characters a and a' of Lstrings is defined as the lexicographic order induced by the base alphabet Σ. The lexicographic order of two Lstrings (and corresponding submatrices) is induced by the order of their characters. We additionally assume that the bottom row and rightmost column of T consist of only a \$ symbol, which is the smallest in the alphabet order and occurs nowhere else in T.

The suffix array $\mathrm{SA}[0\,..\,n^2)$ of a 2D text $T[0\,..\,n)[0\,..\,n)$ is an array containing 2D points such that if $(i,j) = \mathrm{SA}[h]$, then $T[i\,..][j\,..]$ is the hth smallest suffix lexicographically. The inverse suffix array maps each $(i,j) \in [0\,..\,n)^2$ to its position in SA, i.e. $\mathrm{ISA}[\mathrm{SA}[h]] = h$.

The δ_{2D} Measure and 2D Block Trees. The δ measure is a well-studied compressibility measure for 1D texts [7, 20, 24, 25, 30]. It is defined as $\delta(T) = \max_{1\le t\le n} d_t(T)/t$ where $d_t(T)$ denotes the number of distinct length t substrings of $T[0,n)$.

Carfagna and Manzini recently generalized the δ measure to 2D texts [5, 6]. Letting $d_t(T)$ denote the number of distinct $t \times t$ submatrices of $T[0\,..\,n)[0\,..\,n)$, $\delta_{2D}(T) = \max_{1\le t\le n} d_t(T)/t^2$. Observe that $\delta_{2D}(T)$ can range between 1, e.g., in case where all elements of T are the same character, and n^2, i.e., the case where all elements of T are distinct. Carfagna and Manzini showed that the 2D *block tree* data structure of Brisaboa, et al. [3] occupies $\mathcal{O}((\delta_{2D}(T) + \sqrt{n\delta_{2D}(T)})\log \frac{n\log\sigma}{\sqrt{\delta_{2D}}\log n})$ words of space and provides access to any entry of T in $\mathcal{O}(1 + \log \frac{n\log\sigma}{\sqrt{\delta_{2D}}\log n})$ time. A further generalization of the δ measure to 2D allowing for non-square matrices was introduced by Romana et al. and related to other potential 2D compressibility measures [31]. In this work, we will only consider the δ_{2D} measure based on square submatrices. We hereafter refer to δ_{2D} as δ and omit the text T when it is clear from context.

2 Compact Data Structures for 2D LCE Queries

We start with some definitions. Let R_i denote the ith row and C_j denote the jth column of our 2D text T, where $0 \le i,j < n$. Specifically, $R_i[0\,..\,n)$ (resp., $C_j[0\,..\,n)$) is a text of length n over the alphabet Σ, such that its kth character is $T[i][k]$ (resp., $T[k][j]$), where $k \in [0\,..\,n)$.

We define a set of sampled positions on the diagonals of T, that is $T[n-1][0]$, $T[n-2][0] \cdot T[n-1][1]$, $\ldots$, $T[0][n-2] \cdot T[1][n-1]$, $T[0][n-1]$, using d-cover with $d = \Theta(\log^2_\sigma n)$. This is obtained by taking a d-cover $\mathcal{C}$ for $[0\,..\,n)$ and using it to define sample positions starting from the top left of each diagonal. Formally, the sample positions are

$$\mathcal{C}_D = \{(i,j) \mid i,j \in [0\,..\,n), \min(i,j) \in \mathcal{C}\}.$$

See Figure 3.

Figure 3 An example 7-cover $\mathcal{C} = \{1, 2, 4\}$ used for the diagonal sample positions of a 7×7 text. Note that this $d = 7$ value is for illustrative purposes. Sample positions are indicated with a "●".

We maintain a sparse suffix tree over the suffixes starting from these sampled positions. As this is a compact trie with $|\mathcal{C}_D| = \mathcal{O}(n^2/\sqrt{d})$ leaves, the space required for this sparse suffix tree is $\mathcal{O}(n^2/\sqrt{d})$ words. By our above choice of d, this is $\mathcal{O}(n^2 \log \sigma)$ bits. Using this sparse suffix tree, we can obtain LCE for any two sampled positions in $\mathcal{O}(1)$ time.

Additionally, we maintain the data structure from Lemma 6 for the concatenation of columns $C_0, \ldots, C_{n-1}$ and rows $R_0, \ldots, R_{n-1}$, which adds another $\mathcal{O}(n^2 \log \sigma)$ bits. This allows us to find the LCE between $R_i[x \mathinner{.\,.}]$ and $R_j[y \mathinner{.\,.}]$ (or $C_i[x \mathinner{.\,.}]$ and $C_j[y \mathinner{.\,.}]$) in $\mathcal{O}(1)$ time. We can take a minimum between the LCE value and $\min(n - x, n - y)$ to avoid common prefixes crossing row or column boundaries.

In what follows, we first present a simple preliminary solution. We then develop these ideas further with two refinements that lead us to Theorem 1. The components defined above (sparse suffix tree from diagonal samples and LCE data structures for concatenated rows and columns) are used in all three solutions.

2.1 Achieving $\mathcal{O}(\log_\sigma^2 n)$ Query Time

To answer an LCE query (i_1, j_1), (i_2, j_2), we use the look-up structure discussed in Lemma 8 to obtain an $h \in [0 \mathinner{.\,.} d)$ such that $(i_1{+}h, j_1{+}h)$ and $(i_2{+}h, j_2{+}h)$ are sampled diagonal positions. For convenience, in the case where no such h in the look-up structure exists, because either (i_1, j_1) or (i_2, j_2) is near the boundary of T, we consider h as being one less than the minimum diagonal offset to a boundary of T. We first obtain $\mathrm{LCE}((i_1 + h, j_1 + h), (i_2 + h, j_2 + h))$ in $\mathcal{O}(1)$ time. Next, for $k \in [0 \mathinner{.\,.} h)$, we compute the LCEs between $R_{i_1+k}[j_1 \mathinner{.\,.}]$ and $R_{i_2+k}[j_2 \mathinner{.\,.}]$, and between $C_{j_1+k}[i_1 \mathinner{.\,.}]$ and $C_{j_2+k}[i_2 \mathinner{.\,.}]$. While iterating from $k = 1$ to $k = h - 1$, if for some k either the LCE between $R_{i_1+k}[j_1 \mathinner{.\,.}]$ and $R_{i_2+k}[j_2 \mathinner{.\,.}]$ or between $C_{j_1+k}[i_1 \mathinner{.\,.}]$ and $C_{j_2+k}[i_2 \mathinner{.\,.}]$ becomes less than k, we output $k - 1$. Otherwise, we output the minimum over $h + \mathrm{LCE}((i_1 + h, j_1 + h), (i_2 + h, j_2 + h))$ and all of the LCE values computed for the rows and columns specified above.

Only one constant time query for a diagonal sampled position is required, and the number of 1D LCE queries needed is at most $2d$. Since $d = \Theta(\log_\sigma^2 n)$ and each 1D LCE query takes $\mathcal{O}(1)$ time, the total time is $\mathcal{O}(\log_\sigma^2 n)$.

2.2 Achieving $\mathcal{O}(\log_\sigma n \cdot (\log \log_\sigma n)^2)$ Query Time

First, we define $R_{i,t}$ and $C_{j,t}$. These are texts of length n over an alphabet Σ^t, such that $0 \le i, j$ and $i + t - 1, j + t - 1 < n$. The kth character of $R_{i,t}$ and $C_{j,t}$ are length t strings over Σ defined as follows:

$$R_{i,t}[k] = R_i[k] \cdot R_{i+1}[k] \cdots R_{i+t-1}[k]$$

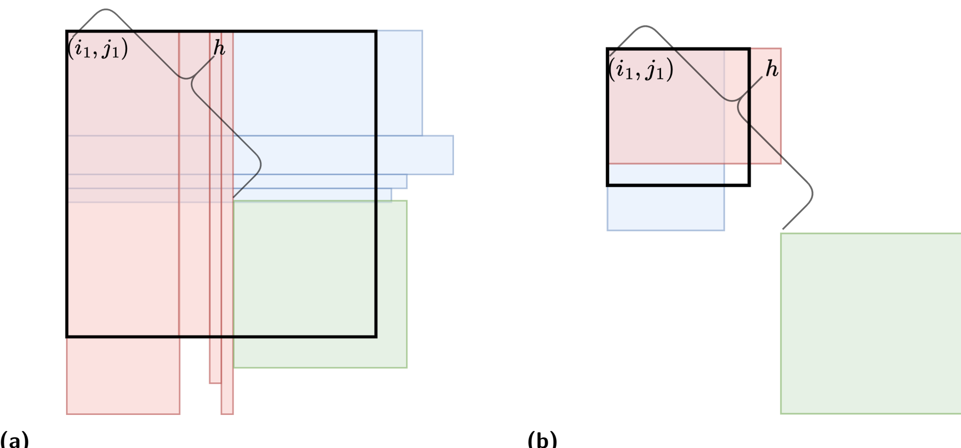

Figure 4 The two cases considered when querying. The actual LCE is shown as the black square. $\text{LCE}((i_1+h, j_1+h), (i_2+h, j_2+h))$ is shown with a green square. The LCE of slabs are shown in red and blue. Further binary search is necessary in Case (b).

$$C_{j,t}[k] = C_j[k] \cdot C_{j+1}[k] \cdots C_{j+t-1}[k].$$

We call these *meta characters*. We also call $R_{i,t}$ and $C_{j,t}$ *slabs* of length t. Applying the structure from Lemma 6 over the concatenation of rows and the concatenation of columns, we can compare two meta characters in $\mathcal{O}(1)$ time.

Data Structure. In addition to the previous components, we maintain the structure from Lemma 7 over the text obtained by concatenating $R_{i,t}$ for $i \in [0\,..\,n)$ and $t = 1, 2, 4, 8, \ldots, \min(n-i, 2^{\lceil \log d \rceil})$. We also maintain the structure from Lemma 7 over the text obtained by concatenating $C_{j,t}$ for $j \in [0\,..\,n)$ and $t = 1, 2, 4, 8, \ldots, \min(n-j, 2^{\lceil \log d \rceil})$. We leave the parameter τ appearing in Lemma 7 to be optimized later.

Querying. Given an LCE query (i_1, j_1), (i_2, j_2), we first find an $h \in [0\,..\,d)$ such that (i_1+h, j_1+h) and (i_2+h, j_2+h) are sampled positions. We then decompose the interval $[i_1\,..\,i_1+h)$ and $[j_1\,..\,j_1+h)$ into $\mathcal{O}(\log d)$ slabs that have lengths that are powers of two. We perform an LCE query for each corresponding slab for both rows and columns. A minimum is taken over all these LCE values and $h + \text{LCE}((i_1+h, j_1+h), (i_2+h, j_2+h))$. Denote this minimum with m. There are two possible cases.

- $m > h$. See Figure 4a. In this case, m is reported as the result.
- $m \le h$. See Figure 4b. In this case, we still need to find the largest value y such that the minimum LCE of the slabs covering $C_{j_1}[i_1\,..]$, …, $C_{j_1+y}[i_1\,..]$ (with slabs covering $C_{j_2}[i_2\,..]$,…, $C_{j_2+y}[i_2\,..]$, respectively) and $R_{i_1}[j_1\,..]$, …, $R_{i_1+y}[j_1]$ (with slabs covering $R_{i_2}[j_2\,..]$, …, $R_{i_2+y}[j_2\,..]$, respectively) is at least y. To accomplish this, we perform a modified binary search while keeping track of the minimum LCE values for both the column and row slabs. The only difference compared to standard binary search is that rather than always dividing the current range under consideration in half, we consider the power of two closest to half the size of the current range. This is done to ensure that we always use slabs for which we have prepared LCE data structures.

Analysis. Letting $T(l)$ be the number of LCE queries on slabs for the binary search on a range of length l, the resulting recurrence is

$$T(l) \le T(2^{\lceil \log l/2 \rceil}) + 1 = \mathcal{O}(\log l).$$

Hence, $T(h) = \mathcal{O}(\log d)$. We now fix $\tau = \log_\sigma n \cdot \log\log_\sigma n$. Since each LCE query on a slab takes $\mathcal{O}(\tau)$ time, the overall query time is $\tau \cdot \log d = \mathcal{O}(\log_\sigma n \cdot (\log\log_\sigma n)^2)$, where we used that $d = \Theta(\log_\sigma^2 n)$. The total added space relative to the previous solution is $\mathcal{O}(\log d \cdot n^2/\tau)$ words. Using our definitions of d and τ, the space remains $\mathcal{O}(n^2 \log \sigma)$ bits.

2.3 Achieving $\mathcal{O}(\log_\sigma^{2/3} n \cdot (\log\log_\sigma n)^{5/3})$ Query Time

Data Structure. Let x be a parameter to be defined later. In addition to the previously defined diagonal sample positions, we now define sample positions for the rows and columns using an x-cover, denoted by $\mathcal{X}$. We maintain the structure in Lemma 7 (with parameter τ left open for optimizing later) over the text obtained by concatenating slabs $R_{i,t}$ for $t = 1, 2, 4, 8, \dots, \min(n-i, 2^{\lceil \log d \rceil})$, whenever $i \in \mathcal{X}$. We do the same for slabs $R_{i,t}$ for $t = 1, 2, 4, 8, \dots, \min(2^{\lceil \log d \rceil})$ whenever $i + t - 1 \in \mathcal{X}$ and $i \ge 0$. Similarly, we maintain the structure from Lemma 7 for the concatenation of $C_{j,t}$ for $t = 1, 2, 4, 8, \dots, \min(2^{\lceil \log d \rceil})$ for $j \in \mathcal{X}$. We do the same for $C_{j,t}$ for $t = 1, 2, 4, 8, \dots, \min(2^{\lceil \log d \rceil})$ whenever $j + t - 1 \in \mathcal{X}$ and $j \ge 0$. Note that these slabs do not need to be explicitly constructed and can be simulated directly using the input text.

Querying. Given a query (i_1, j_1), (i_2, j_2), we first find $h \in [0\,..\,d)$ such that $(i_1 + h, j_1 + h)$ and $(i_2 + h, j_2 + h)$ are diagonal sample positions. Let find $y \in [0\,..\,x)$ such that $i_1 + y$ and $i_2 + y$ are in $\mathcal{X}$. We find the LCEs of columns $C_{i_1}[j_1\,..]$, …, $C_{i_1+y-1}[j_1\,..]$ with $C_{i_2}[j_2\,..]$, …, $C_{i_2+y-1}[j_2\,..]$, respectively. We next find $y' \in [0\,..\,x)$ such that $i_1 + h - 1 - y'$ and $i_2 + h - 1 - y'$ are in $\mathcal{X}$. We then find the LCEs of columns $C_{i_1+h-y'}[j_1\,..]$, … $C_{i_1+h-1}[j_1\,..]$, with $C_{i_2+h-y'}[j_2\,..]$, …, $C_{i_2+h-1}[j_2\,..]$, respectively. We then take the largest power of two, say 2^a, such that $i_1 + y + 2^a \le i_1 + h - 1 - y'$, and obtain the LCE of the slab $C_{i_1+y,2^a}[j_1\,..]$ with $C_{i_2+y,2^a}[j_2\,..]$. We also obtain the LCE of the slabs $C_{i_1+h-y'-2^a,2^a}[j_1\,..]$ and $C_{i_2+h-y'-2^a,2^a}[j_2\,..]$. We perform a symmetric procedure on the rows. A minimum is taken among all of these LCE values as well as $h + \mathrm{LCE}((i_1+h, j_1+h), (i_2+h, j_2+h))$. Let m denote this minimum. We consider two cases like in Section 2.2.

- $m > h$. In this case, m is reported as the result.
- $m \le h$. As in Section 2.2, we want to output the largest value y such that the minimum LCE of the slabs covering $C_{j_1}[i_1\,..]$, …, $C_{j_1+y}[i_1\,..]$ (with LCE relative to slabs covering $C_{j_2}[i_2\,..]$,…, $C_{j_2+y}[i_2\,..]$) and $R_{i_1}[j_1\,..]$, …, $R_{i_1+y}[j_1]$ (with LCE relative to slabs covering $R_{i_2}[j_2\,..]$, …, $R_{i_2+y}[j_2\,..]$) is at least y. The modification to the binary search algorithm from Section 2.2 is that we intermix at most x single row/column evaluations to reach the next position in $\mathcal{X}$. After this position in $\mathcal{X}$ is reached, the power of two that most evenly splits the remaining range can be used.

Analysis. We claim that answering a query requires $\mathcal{O}(x \cdot \log d)$ number of LCE queries for single rows/columns and $\mathcal{O}(\log d)$ number of LCE queries on slabs. To see this, let $S(l)$ be the number of single row/column LCE queries on a range of length l, and $T(l)$ be the number of slab LCE queries. Then we have

$$S(l) \le S(2^{\lceil \log l/2 \rceil}) + \mathcal{O}(x) = \mathcal{O}(x \log l)$$

$$T(l) \le T(2^{\lceil \log l/2 \rceil}) + 1 = \mathcal{O}(\log l).$$

Hence, $S(h) = \mathcal{O}(x \log d)$ and $T(h) = \mathcal{O}(\log d)$. Each single row/column LCE query takes $\mathcal{O}(1)$ time and each LCE query on a slab takes $\mathcal{O}(\tau)$ time. As a result, the total query time is $\mathcal{O}(x \cdot \log d + \log d \cdot \tau)$. To optimize, we keep $d = \Theta(\log_\sigma^2 n)$ and now fix $x = \tau = (\log_\sigma^{2/3} n \cdot (\log\log_\sigma n)^{2/3})$ and obtain the query time of $\mathcal{O}(\log_\sigma^{2/3} n \cdot (\log\log_\sigma n)^{5/3})$.

The (extra) space is $\mathcal{O}(\log d \cdot n^2/(\sqrt{x} \cdot \tau))$ words. This is because we take $\mathcal{O}(\log d)$ larger slabs for each column/row sample position, creating an overall string of length $\mathcal{O}(\log d \cdot n^2/\sqrt{x})$. The LCE structure from Lemma 7, then occupies $\mathcal{O}(\log d \cdot n^2/(\sqrt{x} \cdot \tau))$ words. With the above choice of x and τ, the total space is $\mathcal{O}(n^2 \log \sigma)$ bits. This completes the proof of Theorem 1.

3 Repetition-Aware LCE Data Structure

Overview. We use a parameter τ that we will optimize over later. We aim to use a truncated suffix tree in conjunction with a sparse suffix tree on sampled positions from a τ-cover to efficiently perform LCE queries. If we truncate the 2D suffix tree at a string depth of τ, then the δ measure provides an upper bound of $\tau^2\delta$ on the number of leaves at depth τ. As we argue, one can also upper bound the number of additional leaves in the truncated suffix tree in terms of τ and n.

The first challenge in using the above ideas is that, for these LCE queries from sampled positions to provide information on the overall LCE result, the matching submatrices starting at sampled positions should overlap. This is accomplished by using a string depth of 2τ for the truncated suffix tree while still using a τ-cover. The second challenge is that given our LCE query, we need to know which leaves to consider in the truncated suffix tree. Moreover, we should accomplish this in $o(n^2)$ space when δ is small. To this end, we introduce the notion of a *macro-matrix* M, which stores the leaf in the truncated suffix tree to examine for a specified position in T. We then relate the δ measure of this macro-matrix to the δ measure of the matrix T. This relationship enables us to use the 2D block tree data structure of Brisboa et al. [3] on M, which occupies sublinear space for compressible matrices and supports efficient access to the elements of M.

3.1 Data Structures

Truncated Suffix Tree. We first construct a 2D suffix tree of T truncated at a string depth of 2τ. Call this $\mathcal{T}_{\leq 2\tau}$. We use ℓ_1, ℓ_2, ... to denote the leaves of $\mathcal{T}_{\leq 2\tau}$.

Compressed Representation of Macro-Matrix. We next define the *macro-matrix*. The elements of a macro-matrix are essentially meta symbols, where two meta-symbols are the same if and only if the corresponding $2\tau \times 2\tau$ square substrings are identical. Formally, the macro-matrix M is the matrix obtained as follows: for $i, j \in [0\,..\,n)$,

- if there exists a $2\tau \times 2\tau$ matrix with upper left corner (i, j), i.e., $i, j \leq n - 2\tau$, then we make $M[i][j] = \ell$ where ℓ is a pointer to the leaf of $\mathcal{T}_{\leq 2\tau}$ corresponding to $T[i\,..\,i + 2\tau - 1][j\,..\,j + 2\tau - 1]$;
- if $i > n - 2\tau$ or $j > n - 2\tau$, then let $M[i][j] = \ell$ where ℓ is a pointer to the leaf in $\mathcal{T}_{\leq 2\tau}$ corresponding to the $(n - \max(i, j)) \times (n - \max(i, j))$ matrix with upper left corner (i, j).

See Figure 5. We then construct the 2D block tree of M, denoted as $\mathsf{BT}(M)$.

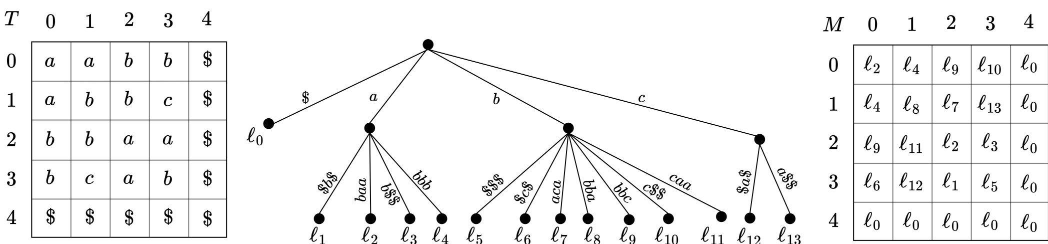

Figure 5 An example 2D text T, a truncated suffix tree with $\tau = 1$, i.e., truncated at a string depth of $2\tau = 2$, and the resulting macro-matrix M.

Sparse Suffix Tree. We define sample positions based on a τ-cover $\mathcal{C}$ of $[0..n)$. These consist of sample positions for the rows,

$$\mathcal{C}_R = \{(i,j) \mid i \in \mathcal{C}, j \in [0..n)\}$$

for the columns,

$$\mathcal{C}_C = \{(i,j) \mid i \in [0..n), j \in \mathcal{C}\}$$

and for the diagonals,

$$\mathcal{C}_D = \{(i,j) \mid i,j \in [0..n), \min(i,j) \in \mathcal{C}\}.$$

Let $\mathcal{C}' = \mathcal{C}_R \cup \mathcal{C}_C \cup \mathcal{C}_D$. Observe that $|\mathcal{C}'| = \Theta(n^2/\sqrt{\tau})$. We build a sparse suffix tree over the suffixes starting at sampled positions in $\mathcal{C}'$, denoted as $\mathcal{T}_s$. We also maintain the lookup data structure from Lemma 8. As before, this allows us to find in $\mathcal{O}(1)$ time equally far sampled positions at most τ away from the queried positions in each row, column, and diagonal.

3.2 Querying

Given LCE query (i_1, j_1), (i_2, j_2), we first use $\mathsf{BT}(M)$ to get the corresponding values in M. Say these correspond to the leaves ℓ_1 and ℓ_2 in $\mathcal{T}_{\leq 2\tau}$ respectively. If $\ell_1 \neq \ell_2$, then the string depth of the LCA of ℓ_1 and ℓ_2 gives us the LCE of (i_1, j_1), (i_2, j_2).

If $\ell_1 = \ell_2$ then we use the lookup data structure from Lemma 8 to find:

- $h_1 \in [0..\tau)$ such that $(i_1 + h_1, j_1)$ and $(i_2 + h_1, j_2)$ are sampled positions. We then use an $\mathcal{O}(1)$ time query on $\mathcal{T}_s$ to get the LCE of $(i_1 + h_1, j_1)$ and $(i_2 + h_1, j_2)$. Denote this LCE value as L_1.
- $h_2 \in [0..\tau)$ such that $(i_1 + h_2, j_1 + h_2)$ and $(i_2 + h_2, j_2 + h_2)$ are sampled positions. We use an $\mathcal{O}(1)$ time query on $\mathcal{T}_s$ to get the LCE of $(i_1 + h_2, j_1 + h_2)$ and $(i_2 + h_2, j_2 + h_2)$. Denote this LCE value as L_2.
- $h_3 \in [0..\tau)$ such that $(i_1, j_1 + h_3)$ and $(i_2, j_2 + h_3)$ are sampled positions. We use an $\mathcal{O}(1)$ time query on $\mathcal{T}_s$ to get the LCE of $(i_1, j_1 + h_3)$ and $(i_2, j_2 + h_3)$. Denote this LCE value as L_3.

We report $\min(h_1 + L_1, h_2 + L_2, h_3 + L_3)$ as the solution.

3.3 Correctness

The first lemma is immediate.

▶ **Lemma 9.** *When* $\ell_1 \neq \ell_2$, $\mathrm{LCE}((i_1, j_1), (i_2, j_2))$ *is the string depth of* $\mathrm{LCA}(\ell_1, \ell_2)$.

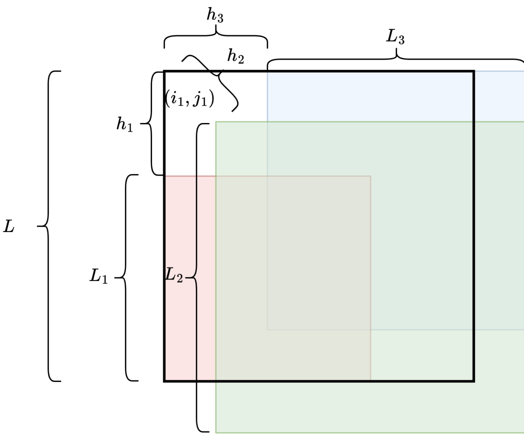

Figure 6 The solution LCE L is shown as the black square. Submatrix T_1 matrix in red, submatrix T_2 in green, and submatrix T_3 matrix in blue.

▶ **Lemma 10.** *When* $\ell_1 = \ell_2$, $\mathrm{LCE}((i_1, j_1), (i_2, j_2)) = \min(h_1 + L_1, h_2 + L_2, h_3 + L_3)$.

Proof. Define $L := \mathrm{LCE}((i_1, j_1), (i_2, j_2))$. First, we show that $L \le \min(h_1 + L_1, h_2 + L_2, h_3 + L_3)$. Starting from $(i_1, j_1 + h_1)$ there exists a matching submatrix (with respect to position $(i_2, j_2 + h_1)$) of size at least $L - h_1$, thus we have that $L_1 \ge L - h_1$. Hence, $L_1 + h_1 \ge L$. A similar argument holds for h_2 and h_3.

Next, we show $L \not< \min(h_1 + L_1, h_2 + L_2, h_3 + L_3)$.

- We denote the submatrix $T[i_1 + h_1 \mathinner{.\,.} i_1 + h_1 + L_1)[j_1 \mathinner{.\,.} j_1 + L_1)$ as T_1.
- We denote the submatrix $T[i_1 + h_2 \mathinner{.\,.} i_1 + h_2 + L_2)[j_1 + h_2 \mathinner{.\,.} j_1 + h_2 + L_2)$ as T_2.
- We denote the submatrix $T[i_1 \mathinner{.\,.} i_1 + L_3)[j_1 + h_3 \mathinner{.\,.} j_1 + h_3 + L_3)$ as T_3

See Figure 6.

Observe that $h_1, h_2, h_3 \le \tau - 1$ and since $L \ge 2\tau$, we have $L_1, L_2, L_3 \ge \tau$. Submatrix T_2 has lower left corner in column $j_1 + h_2 \le j_1 + L_1 - 1$ and in row $i_1 + h_2 + L_2 - 1 \ge i_1 + h_1$ making it overlap with T_1. Also, T_2 has upper right corner in column $j_1 + h_2 + L_2 - 1 \ge j_1 + h_3$ and row $i_1 + h_2 \le i_1 + h_3 + L_3 - 1$. Hence, T_2 overlaps with T_3 as well.

Now, suppose for the sake of contradiction that $h_1 + L_1, h_2 + L_2, h_3 + L_3 > L$. For any positions in row $x = i_1 + L$ and column y where $j_1 \le y \le j_1 + L$ we have

$$i_1 \le x = i_1 + L \le i_1 + h_1 + L_1 - 1,\ i_1 + h_2 + L_2 - 1$$

and

$$j_1 \le y \le j_1 + L \le j_1 + h_1 + L_1 - 1,\ j_1 + h_2 + L_2 - 1.$$

Similarly, for any position in column $y = j_1 + L$ and row x where $i_1 \le x \le i_1 + L$ we have

$$j_1 \le y = j_1 + L \le j_1 + h_2 + L_2 - 1,\ j_1 + h_3 + L_3 - 1$$

and

$$i_1 \le x \le i_1 + L \le i_1 + h_2 + L_2 - 1,\ i_1 + h_3 + L_3 - 1.$$

Based on the above inequalities and the fact that submatrices T_1, T_2, and T_3 overlap, this implies that the matching submatrices with upper left corners (i_1, j_1) and (i_2, j_2) can be extended further by at least one row and column. This contradicts the definition of L. ◀

3.4 Analysis and Optimization

3.4.1 Space Analysis

Space for τ-Cover lookup structure and Sparse Suffix Tree. According to Lemma 8, the lookup structure requires $\mathcal{O}(\tau)$ space. Since $|\mathcal{C}'| = \mathcal{O}(n^2/\sqrt{\tau})$, we have that the sparse suffix tree $\mathcal{T}_s$ uses $\mathcal{O}(n^2/\sqrt{\tau})$ space.

Space for $\mathcal{T}_{\leq 2\tau}$. The space for the truncated suffix tree $\mathcal{T}_{\leq 2\tau}$ is bounded by the number of distinct $2\tau \times 2\tau$ submatrices of T, denoted $d_{2\tau}(T)$, plus the number of distinct matrices of size less than 2τ that can not be further extended down and to the right (due to a boundary of T). There are at most $\mathcal{O}(\tau n)$ of the latter since, for every length from 1 to 2τ, at most $2n$ submatrices cannot be further extended. By the definition of δ, $d_{2\tau}(T) \leq 4\tau^2\delta(T)$, making the space for $\mathcal{T}_{\leq 2\tau}$ bound by $\mathcal{O}(\tau^2\delta(T) + \tau n)$.

Space for Macro-Matrix. The space for $\mathsf{BT}(M)$ depends on $\delta(M)$. We prove the following lemma relating $\delta(T)$ and $\delta(M)$.

▶ **Lemma 11.** *$\delta(M) = \Omega(\max(1, \delta(T)/\tau^2 - n/\tau))$ and $\delta(M) = \mathcal{O}(\tau^2\delta(T) + \tau n)$.*

Proof. First, the lower bound. Observe that for an arbitrary $t \in [2\tau\,..\,n]$, two matching $t \times t$ submatrices in T cause two matching $(t - 2\tau + 1) \times (t - 2\tau + 1)$ submatrices in M (with the same upper left corners as the corresponding submatrices in T). In this way, every distinct $t \times t$ submatrix in T maps to one distinct $(t - 2\tau + 1) \times (t - 2\tau + 1)$ submatrix in M, and we have $d_t(T) \leq d_{(t-2\tau+1)}(M)$. Then for $t \geq 2\tau$, we have

$$\frac{d_t(T)}{t^2} \leq \frac{d_{(t-2\tau+1)}(M)}{t^2} \leq \frac{(t-2\tau+1)^2\delta(M)}{t^2} \leq \delta(M) \tag{1}$$

implying $d_t(T) \leq t^2\delta(M)$ for $t \geq 2\tau$.

Next, consider $t \in [1\,..\,2\tau)$. Note that the number of distinct $t \times t$ submatrices in T is almost upper bounded by the number of distinct $(t + 2\tau) \times (t + 2\tau)$ submatrices in T, except that some of the distinct matrices with sizes between $t \times t$ and $(t + 2\tau) \times (t + 2\tau)$ may be prevented from being extended due to the right and bottom boundaries of T. The number of such submatrices is bounded by $2n(t + 2\tau - t) = \mathcal{O}(\tau n)$. Hence, for $t < 2\tau$,

$$d_t(T) \leq d_{(t+2\tau)}(T) + \mathcal{O}(\tau n)$$

Applying Inequality (1), we can then write

$$\frac{d_t(T)}{t^2} \leq \frac{d_{(t+2\tau)}(T)}{t^2} + \frac{\mathcal{O}(\tau n)}{t^2} \leq \frac{(t+2\tau)^2}{t^2}\delta(M) + \mathcal{O}(\tau n) = \mathcal{O}(\tau^2\delta(M) + \tau n).$$

Taking the maximum over both cases, yields that $\delta(T) = \mathcal{O}(\tau^2\delta(M) + \tau n)$.

For the upper bound, we claim that, for an arbitrary $t \in [1\,..\,n]$,

$$d_t(M) \leq d_{(t+2\tau-1)}(T) + \mathcal{O}(\tau n),$$

where we take $d_{(t+2\tau-1)}(T) = 0$ if $t + 2\tau - 1 > n$. The above inequality follows from the fact that every distinct $(t + 2\tau - 1) \times (t + 2\tau - 1)$ submatrix in T maps to one distinct $t \times t$ submatrix in M. What remains to be counted for $d_t(M)$ are distinct $t \times t$ submatrices in M that are not resulting from some $(t + 2\tau - 1) \times (t + 2\tau - 1)$ submatrix in T. That is, submatrices on the bottom and/or right boundary. Again, the number of such $t \times t$ submatrices is bounded by $2n((t + 2\tau - 1) - t) = \mathcal{O}(\tau n)$.

To complete the proof, we have the bound

$$\delta(M) = \max_t \frac{d_t(M)}{t^2} \leq \max_t \frac{d_{(t+2\tau-1)}(T) + \mathcal{O}(\tau n)}{t^2}$$
$$\leq \max_t \frac{(t+2\tau-1)^2}{t^2}\delta(T) + \mathcal{O}(\tau n) = \mathcal{O}(\tau^2\delta(T) + \tau n). \quad ◀$$

Let σ' be the alphabet size of the macro-matrix M. The space for the block tree $\mathsf{BT}(M)$ is

$$\mathcal{O}\left((\delta(M) + \sqrt{n\delta(M)})\log\left(\frac{n\log\sigma'}{\sqrt{\delta(M)}\log n}\right)\right).$$

Applying that $\sigma' \leq n^2$ and Lemma 11, this space is bound by

$$\mathcal{O}\left((\tau^2\delta(T) + \tau\sqrt{n\delta(T)} + \tau n)\log\left(\frac{n}{\sqrt{\max(1, \frac{\delta(T)}{\tau^2} - \frac{n}{\tau})}}\right)\right).$$

Total Space. Summing the total data structure sizes, the combined space is

$$\mathcal{O}\left((\tau^2\delta(T) \ + \tau\sqrt{n\delta(T)} + \tau n)\log\left(\frac{n}{\sqrt{\max(1, \frac{\delta(T)}{\tau^2} - \frac{n}{\tau})}}\right) + \frac{n^2}{\sqrt{\tau}} + \tau\right).$$

3.4.2 Optimizing

We consider two cases based on $\delta(T)$, which we now denote as just δ. If $\delta > n^{1/3}$, we set $\tau = \lceil n^{4/5}/(2\delta^{2/5})\rceil$ and let $\beta = n/\sqrt{\max(1, 4\delta^{9/5}/n^{8/5} - 2n^{1/5}\delta^{2/5})}$. The space is (up to constant factors)

$$\left(n^{8/5}\delta^{1/5} + n^{13/10}\delta^{1/10} + \frac{n^{9/5}}{\delta^{2/5}}\right)\log\beta + n^{8/5}\delta^{1/5} + \frac{n^{4/5}}{\delta^{2/5}} = \mathcal{O}\left(n^{8/5}\delta^{1/5}\cdot\log\beta\right).$$

Observe that as δ approaches n^2, β approaches $\mathcal{O}(1)$.
If $\delta \leq n^{1/3}$, we make $\tau = n^{2/3}$. The resulting space complexity is

$$(n^{4/3}\delta + n^{7/6}\sqrt{\delta} + n^{5/3})\log\beta + n^{5/3} + n^{2/3} = \mathcal{O}\left(n^{5/3}\log\beta\right).$$

For this case, the argument of the logarithm β is $\mathcal{O}(n)$. One can also readily check that β as defined above is bound by the expression for β appearing in Theorem 2.

3.4.3 Query Time

The query time is dominated by the access to $\mathsf{BT}(M)$, which takes $1+\log\frac{n}{\sqrt{\delta(M)}} = \mathcal{O}(1+\log\beta)$ time, where β is defined as above. The remaining queries take $\mathcal{O}(1)$ time. This completes the proof of Theorem 2.

4 Applications

We next demonstrate some applications of Theorem 1 by proving Theorems 3, 4, 5.

4.1 ISA Queries

We maintain a sampled suffix array. Specifically, we sample the suffix array values for every $(\log_\sigma n)$ leaf of the suffix tree. The space required for this is $\mathcal{O}(n^2 \log \sigma)$ bits. Additionally, for each text position, we maintain how far away its predecessor sampled leaf is relative to its leaf in the suffix tree. This requires $\mathcal{O}(\log \log_\sigma n)$ bits per entry. The resulting total space is $\mathcal{O}(n^2 \log \sigma + n^2 \log \log n)$ bits.

To find the ISA value of a text position (i, j), we perform a binary search on the sampled leaves to find the lexicographic predecessor of (i, j) within the sampled set. Once the predecessor is found, we add the offset associated with (i, j). This gives us the suffix array position associated with (i, j), i.e., its ISA value. The binary search requires $\mathcal{O}(\log n)$ number of LCE queries. Each LCE query takes $\mathcal{O}(\log_\sigma^{2/3} n \cdot (\log \log_\sigma n)^{5/3})$ time, resulting in an overall time complexity of $\mathcal{O}(\log n \cdot \log_\sigma^{2/3} n \cdot (\log \log_\sigma n)^{5/3})$.

4.2 SA queries

Let τ be a parameter. We divide the leaves of the suffix tree into contiguous blocks of size $\lceil n^2/\tau \rceil$ (except for perhaps the last block, which can be smaller). There are $\Theta(\tau)$ blocks. We associate each position in T with the block in which its leaf lies in the suffix tree. This information is stored as follows: consider a binary array B_b associated with each block b. Each binary array is of length n^2 and represents a linearization of T. For a block B_b, we consider a 1 in a position if the corresponding suffix tree leaf is in block b and 0 otherwise. Note that there are at most $m := \lceil n^2/\tau \rceil$ 1's in B_b. We build a data structure representing B_b using $m \log \frac{n^2}{m} + \mathcal{O}(m)$ bits of space, or equivalently, $n^2/\tau \cdot \log \tau + \mathcal{O}(n^2/\tau)$ bits of space, such that select queries can be answered in constant time [29]. The total space for select data structures over all $\Theta(\tau)$ bit vectors, is $n^2 \log \tau + \mathcal{O}(n^2) = \mathcal{O}(n^2 \log \tau)$ bits. We also maintain the ISA data structure described previously.

Given an SA query for index i, we first identify which block i is in. Say this is block b. We use select queries to iterate through the text positions contained in block b. For each text position iterated over, we perform an ISA query and check whether its ISA value equals i.

The space required for the ISA data structure is $\mathcal{O}(n^2 \log \sigma + n^2 \log \log n)$ bits. The space for the select data structures is $\mathcal{O}(n^2 \log \tau)$ bits. The query time is $\mathcal{O}(n^2/\tau \cdot \log n \cdot \log_\sigma^{2/3} n \cdot (\log \log_\sigma n)^{5/3})$. We obtain Theorem 4 by making $\tau = (\sigma \log n)^c$, where c is an arbitrarily large constant that can absorb the additional logarithmic factors in the query time.

4.3 Pattern Matching

Counting. In addition to the previous structures, we maintain the LCE data structure from Lemma 6 over the rows and columns. First, a binary search is done to find the leaf for the lexicographically smallest suffix with P as a prefix (if one exists). We start by using an SA query to obtain $\text{SA}[\lfloor n^2/2 \rfloor]$. Using that we have read access to the original text, we match characters in P in Lstring order to the submatrix starting at $\text{SA}[\lfloor n^2/2 \rfloor]$ until we reach our first mismatch. At this point, we know our lexicographical order relative to our current leaf. When we transition to a new leaf in the binary search, we perform an SA query followed by LCE queries with the position from the preceding leaf. This avoids repeatedly iterating over characters in P. Assuming the LCE query is at least the length already matched, we continue matching from the last matched position. A similar binary search finds the lexicographically largest suffix with P as a prefix. We return the suffix range length.

The total number of LCE and SA queries performed is $\mathcal{O}(\log n)$. The time is dominated by the SA queries, which require $\mathcal{O}(n^2/(\sigma \log n)^c)$ time.

Reporting. We start with the suffix range obtained previously, say $[x\,..\,y]$. We use the same blocking scheme for the suffix leaves described for SA queries, also using constant time select data structures. We first identify the block that x lies in, say B_b. We use the select data structure to iterate through all of the text positions corresponding to suffixes in block b. For each position, we perform an ISA query and check whether its position in the suffix array is at least x. If it is, we output it. We perform a similar procedure for the block containing y, now checking if the position in the suffix array is at most y. For the remaining blocks, those completely contained in $[x\,..\,y]$, we use their select data structures to output all occurrences with suffixes in that block.

The space complexity is the same as the SA data structure. For the query time, each block has size $\mathcal{O}(n^2/\tau)$, and with $\tau = (\sigma \log n)^c$, the time spent on the blocks containing x and y is absorbed by $n^2/(\sigma \log n)^c$ already appearing due to SA queries.

5 Open Problems

We leave open many directions for potential improvement, for example:

- Can we design a data structure with faster SA query time that uses $\mathcal{O}(n^2 \log \sigma + n^2 \log \log n)$ bits of space (or better)? This seems significantly harder than ISA queries. Suffix array sampling, like in the FM-index [10], is not immediately adaptable.
- Can we design a data structure in repetition-aware compressed space that supports ISA, SA, or pattern-matching queries? Also, can the space for a data structure for LCE queries be improved? Grammar-based compression has proven useful for repetition-aware compressed data structures supporting LCE queries in the 1D case, particularly run-length straight-line programs (RL-SLP). For 1D text, it is possible to construct RL-SLPs with size close to the δ measure [25], which can be used for LCE [27] and pattern matching queries [24]. Although Romana et al. [31] introduce a version of RL-SLP for 2D text, it is open how such a RL-SLP could be utilized for LCE queries and other types of queries, e.g., SA and pattern matching queries.

References

1. Philip Bille, Inge Li Gørtz, Mathias Bæk Tejs Knudsen, Moshe Lewenstein, and Hjalte Wedel Vildhøj. Longest common extensions in sublinear space. In Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, editors, *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 65–76. Springer, 2015. doi:10.1007/978-3-319-19929-0_6.
2. Philip Bille, Inge Li Gørtz, Benjamin Sach, and Hjalte Wedel Vildhøj. Time-space trade-offs for longest common extensions. *J. Discrete Algorithms*, 25:42–50, 2014. doi:10.1016/J.JDA.2013.06.003.
3. Nieves R. Brisaboa, Travis Gagie, Adrián Gómez-Brandón, and Gonzalo Navarro. Two-dimensional block trees. *Comput. J.*, 67(1):391–406, 2024. doi:10.1093/COMJNL/BXAC182.
4. Stefan Burkhardt and Juha Kärkkäinen. Fast lightweight suffix array construction and checking. In Ricardo A. Baeza-Yates, Edgar Chávez, and Maxime Crochemore, editors, *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2003. doi:10.1007/3-540-44888-8_5.
5. Lorenzo Carfagna and Giovanni Manzini. Compressibility measures for two-dimensional data. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 102–113. Springer, 2023. doi:10.1007/978-3-031-43980-3_9.

6. Lorenzo Carfagna and Giovanni Manzini. The landscape of compressibility measures for two-dimensional data. *IEEE Access*, 12:87268–87283, 2024. doi:10.1109/ACCESS.2024.3417621.
7. Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17(1):8:1–8:39, 2021. doi:10.1145/3426473.
8. Charles J. Colbourn and Alan C. H. Ling. Quorums from difference covers. *Inf. Process. Lett.*, 75(1-2):9–12, 2000. doi:10.1016/S0020-0190(00)00080-6.
9. Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.
10. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.
11. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.
12. Arnab Ganguly, Dhrumil Patel, Rahul Shah, and Sharma V. Thankachan. LF successor: Compact space indexing for order-isomorphic pattern matching. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 71:1–71:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICALP.2021.71.
13. Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pbwt: Achieving succinct data structures for parameterized pattern matching and related problems. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 397–407. SIAM, 2017. doi:10.1137/1.9781611974782.25.
14. Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Fully functional parameterized suffix trees in compact space. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 65:1–65:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ICALP.2022.65.
15. Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster longest common extension queries in strings over general alphabets. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 5:1–5:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.5.
16. 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.
17. Gaston H Gonnet. *Efficient searching of text and pictures.* UW Centre for the New Oxford English Dictionary, 1990.
18. 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. doi:10.1137/S0097539702402354.
19. Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 756–767. ACM, 2019. doi:10.1145/3313276.3316368.
20. Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. *Commun. ACM*, 65(6):91–98, 2022. doi:10.1145/3531445.

21 Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1877–1886. IEEE, 2023. doi:10.1109/FOCS57990.2023.00114.

22 Dong Kyue Kim, Yoo Ah Kim, and Kunsoo Park. Generalizations of suffix arrays to multi-dimensional matrices. *Theor. Comput. Sci.*, 302(1-3):223–238, 2003. doi:10.1016/S0304-3975(02)00828-9.

23 Dong Kyue Kim, Joong Chae Na, Jeong Seop Sim, and Kunsoo Park. Linear-time construction of two-dimensional suffix trees. *Algorithmica*, 59(2):269–297, 2011. doi:10.1007/S00453-009-9350-Z.

24 Tomasz Kociumaka, Gonzalo Navarro, and Francisco Olivares. Near-optimal search time in δ-optimal space, and vice versa. *Algorithmica*, 86(4):1031–1056, 2024. doi:10.1007/S00453-023-01186-0.

25 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Trans. Inf. Theory*, 69(4):2074–2092, 2023. doi:10.1109/TIT.2022.3224382.

26 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.

27 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In Piotr Faliszewski, Anca Muscholl, and Rolf Niedermeier, editors, *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.MFCS.2016.72.

28 Dhrumil Patel and Rahul Shah. Inverse suffix array queries for 2-dimensional pattern matching in near-compact space. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan*, volume 212 of *LIPIcs*, pages 60:1–60:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ISAAC.2021.60.

29 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

30 Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam D. Smith. Sublinear algorithms for approximating string compressibility. *Algorithmica*, 65(3):685–709, 2013. doi:10.1007/S00453-012-9618-6.

31 Giuseppe Romana, Marinella Sciortino, and Cristian Urbina. Exploring repetitiveness measures for two-dimensional strings, 2024. doi:10.48550/arXiv.2404.07030.

32 Kunihiko Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 225–232. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545410.

33 Sharma V. Thankachan. Compact text indexing for advanced pattern matching problems: Parameterized, order-isomorphic, 2d, etc. (invited talk). In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 3:1–3:3. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CPM.2022.3.

34 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.