Nearly Optimal Internal Dictionary Matching

Jingbang Chen[1]
The Chinese University of Hong Kong, Shenzhen, China

Jiangqi Dai
Massachusetts Institute of Technology, Cambridge, MA, USA

Qiuyang Mang
The Chinese University of Hong Kong, Shenzhen, China

Qingyu Shi
Feicheng Hailiang Foreign Language School, China

Tingqiang Xu
Tsinghua University, Beijing, China

Abstract

We study the *internal dictionary matching* (IDM) problem where a dictionary $\mathcal{D}$ containing d substrings of a text T over a linearly sortable alphabet is given, and each query concerns the occurrences of patterns in $\mathcal{D}$ in another substring of T. We propose a novel $O(n)$-sized data structure named *Basic Substring Structure* (BASS) where n is the length of the text T. With BASS, we are able to handle *all types of queries* in the IDM problem in nearly optimal query and preprocessing time. Specifically, our results include:

- The first algorithm that answers the CountDistinct query in $\tilde{O}(1)$ time with $\tilde{O}(n+d)$ preprocessing, where we need to compute the number of distinct patterns that exist in $T[l,r]$. Previously, the best result was $\tilde{O}(m)$ time per query after $\tilde{O}(n^2/m+d)$ or $\tilde{O}(nd/m+d)$ preprocessing, where m is a chosen parameter.
- Faster algorithms for two other types of internal queries. We improve the runtime for **(1)** *Occurrence counting* (Count) queries to $O(\log n/\log\log n)$ time per query with $O(n+d\sqrt{\log n})$ preprocessing from $O(\log^2 n/\log\log n)$ time per query with $O(n\log n/\log\log n+d\log^{3/2} n)$ preprocessing. **(2)** *Distinct pattern reporting* (ReportDistinct) queries to $O(1+|\text{output}|)$ time per query from $O(\log n+|\text{output}|)$ per query.

In addition, we match the optimal runtime in the remaining two types of queries, *pattern existence* (Exists), and *occurrence reporting* (Report). We also show that BASS is more generally applicable to other internal query problems.

2012 ACM Subject Classification Theory of computation → Pattern matching; Theory of computation → Data structures design and analysis

Keywords and phrases Internal dictionary matching, pattern matching, string algorithms, data structures, substring queries

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.58

Related Version *Full Version*: https://arxiv.org/abs/2312.11873

1 Introduction

The *pattern matching*, as well as the *dictionary matching* problems, have been extensively studied for decades, where we need to preprocess a dictionary $\mathcal{D}$ containing one or multiple patterns to answer their occurrences in each query text. In addition to classical methods [38, 2], there are also studies on developing approximate algorithms [9, 44, 13, 22], and algorithms in the dynamic dictionary setting [5, 6, 48, 14, 34].

[1] Part of this work was done while at Georgia Tech and the University of Waterloo.

In this paper, we consider the *internal* setting where each pattern or query is represented by a substring of a given large text T. Such an "internal" concept is not new in string algorithms. For example, the famous suffix array algorithm [42, 31, 36, 43] sorts all suffixes of T, which helps to answer multiple types of queries such as the longest common prefix (LCP) of two suffixes. In this case, all related texts (suffixes) are internal with respect to T. Computing the LCP of any two suffixes can be easily generalized to computing the LCP of any two substrings (fragments), known as the *longest common extension* (LCE) queries introduced by Landau and Vishkin [41], which could be one of the earliest internal queries.

In recent years, the *internal pattern matching* (IPM) problem has been widely studied, which requires us to compute the occurrences of a substring of T in another substring of T. Recent research has developed algorithms that run in sublogarithmic time and occupy near-linear space [37, 40, 39]. Benefit from the IPM algorithms, many internal queries have been studied, such as shortest unique substrings [1], shortest absent string [8], BWT substring compression [7], circular pattern matching [35], etc. Moreover, the IPM queries have been used in some other problems, including approximate pattern matching [20, 21], approximate circular pattern matching [18, 19], RNA folding [27], and computing string covers [46].

Similar to dictionary matching, a natural generalization of the IPM problem is the newly proposed *internal dictionary matching* (IDM) problem, where we consider multiple patterns. It was first introduced and studied by Charalampopoulos et al. [16, 17]. In such a setting, given a large text T of length n in advance, we consider a dictionary $\mathcal{D}$ consisting of substrings of T. We denote the number of patterns as d. The following types of internal queries are studied in the IDM problem:

- $\textsc{Exists}(l, r)$: Decide whether at least one pattern from $\mathcal{D}$ occurs in $T[l, r]$.
- $\textsc{Report}(l, r)$: Report all occurrences of all patterns from $\mathcal{D}$ in $T[l, r]$.
- $\textsc{ReportDistinct}(l, r)$: Report all distinct patterns from $\mathcal{D}$ in $T[l, r]$.
- $\textsc{Count}(l, r)$: Count the number of all occurrences of all patterns from $\mathcal{D}$ in $T[l, r]$.
- $\textsc{CountDistinct}(l, r)$: Count the number of distinct patterns from $\mathcal{D}$ in $T[l, r]$.

Compared to the classical dictionary matching problem, the IDM problem is especially applicable to any internal dictionary whose total length of all patterns (substrings) L is much larger than the number of patterns d, such as the dictionary of all palindrome substrings, all square substrings, and all non-primitive substrings. Combined with algorithms that efficiently extract the endpoints of patterns [32, 24, 10], an IDM algorithm can answer queries regarding such an internal dictionary in time and space *not* depending on L. IDM algorithms for the palindromes case [47] and the squares case [17] have already been developed. Same as IPM, IDM also has many practical applications. For example, in bioinformatics, the multi-pattern matching query has many deployed applications, most of which are under the internal setting and can be reduced to IDM [50, 51].

Previous work [16] shows that the product of the time to process an update to $\mathcal{D}$ and the time to answer an internal query cannot be $O(n^{1-\epsilon})$ for any constant $\epsilon > 0$, unless the Online Boolean Matrix-Vector Multiplication conjecture [33] is false. This conditional lower bound demonstrates the hardness of efficient algorithms in the dynamic dictionary case. In this paper, we focus mainly on the case of a static dictionary.

1.1 Our Results

In this paper, we propose a new $O(n)$-sized data structure named *Basic Substring Structure (BASS)* that organizes T's substrings in a novel way (Section 3). BASS builds upon the idea of equivalence classes of substrings, an approach that has been explored in previous work, such as the seminal work by Blumer et al. [12, 25], which introduced the concept of

Table 1 Algorithmic Results.

	Our results			Previous results	
	Preprocessing	**Query time**	**Section**	**Preprocessing**	**Query time**
Exists	$O(n+d)$	$O(1)$	A.1	$O(n+d)$	$O(1)$
Report	$O(n+d)$	[1]$O(1+x)$	A.1	$O(n+d)$	[1]$O(1+x)$
Count	$O(n+d\sqrt{\log n})$	$O(\frac{\log n}{\log\log n})$	3.4	$O(\frac{n\log n}{\log\log n}+d\log^{3/2} n)$	$O(\frac{\log^2 n}{\log\log n})$
ReportDistinct	$O(n\log n+d)$	[1]$O(1+x)$	A.2	$O(n\log n+d)$	[1]$O(\log n+x)$
CountDistinct	$O(n\log^2 n+d\sqrt{\log n})$	$O(\log n)$	4	[2]$\tilde{O}(n^2/m+d)$	[2]$\tilde{O}(m)$
				[2]$\tilde{O}(nd/m+d)$	[2]$\tilde{O}(m)$
				[3]$O(n\log^{1+\epsilon} n+d\log^{3/2} n)$	[3]$O(\frac{\log^2 n}{\log\log n})$

[1] x denotes the length of output.
[2] m can be arbitrary positive integer
[3] 2-approximate

equivalence classes in the context of the compact acyclic word graph (CDAWG). While the equivalence class concept is indeed not new, the key contribution of this paper lies in the grid structure that organizes these equivalence classes in a two-dimensional grid. This novel structure enables the efficient handling of substring queries, something that earlier works did not fully address. In contrast to traditional representations like the CDAWG, which represent equivalence classes in a graph-like structure, the grid-based approach in BASS allows for more efficient indexing and querying of patterns.

BASS is the first data structure that is able to handle *all* types of queries of the IDM problem in nearly optimal query and preprocessing time. This contrasts with previous works where they treated different queries with tailored approaches. Table 1 gives an overview of our BASS-based algorithms on different query types compared to previous results. All algorithms take $\tilde{O}(n+d)$[2] space.

1.1.1 Counting Distinct Patterns

By introducing an extra parameter m, the previous best result answers CountDistinct queries in $\tilde{O}(m)$ time after $\tilde{O}(n^2/m+d)$ or $\tilde{O}(nd/m+d)$ preprocessing [17]. The question of whether any data structure answers CountDistinct queries exactly in $\tilde{O}(1)$ time each after $\tilde{O}(n+d)$ preprocessing was open. As one of the main contributions, we answer this question affirmatively:

▶ **Theorem 1** (CountDistinct). *CountDistinct(i,j) can be answered in $O(\log n)$ time with a data structure of $O(n\log^2 n+d)$ size that can be constructed in $O(n\log^2 n+d\sqrt{\log n})$ time.*

Similar to [16], for the case of a dynamic dictionary, where patterns can be added or deleted, we can construct algorithms that process updates in $\tilde{O}(n^\alpha)$ time and answer queries CountDistinct(l,r) in $\tilde{O}(n^{1-\alpha})$ time for any $0<\alpha<1$. For this type of query, this is the first result that matches the known conditional lower bound up to subpolynomial factors.

[2] The $\tilde{O}(\cdot)$ notation suppresses $\log^{O(1)} n$ factors for inputs of size n.

1.1.2 Other Internal Queries

As in previous work [16], Exists, Report, ReportDistinct, Count can be answered in $\tilde{O}(1+|\text{output}|)$ time after $\tilde{O}(n+d)$ preprocessing of T. Our BASS-based algorithms match all previous results and achieve some improvements.

For the Count query, previous works take $O(\frac{\log^2 n}{\log\log n})$ per query after $O(\frac{n\log n}{\log\log n} + d\log^{3/2} n)$ preprocessing. Our result improves the run time for both query and preprocessing:

▶ **Theorem 2** (Count). *$\textsc{Count}(l,r)$ can be answered in $O(\frac{\log n}{\log\log n})$ time with a data structure of size $O(n+d)$ that can be constructed in $O(n+d\sqrt{\log n})$ time.*

Also, we prove that there is an unconditional lower bound on $\textsc{Count}(l,r)$, stating that $O(\frac{\log n}{\log\log n})$ query time is optimal for all data structures of size $\tilde{O}(n)$ [45]. Thus, our query time for $\textsc{Count}(l,r)$ queries is optimal.

For ReportDistinct queries that output all distinct patterns that occurred, using the same preprocessing time, our query time improves to $O(1+|\text{output}|)$ from $O(\log n+|\text{output}|)$:

▶ **Theorem 3** (ReportDistinct). *$\textsc{ReportDistinct}(l,r)$ can be answered in $O(1+x)$ time with a data structure of size $O(n+d)$ that can be constructed in $O(n\log n+d)$ time, where x denotes the length of output.*

For Exists, Report queries, although taking different approaches, both our algorithm and previous works need $O(n+d)$ preprocessing time and $O(1+|\text{output}|)$ query time (Theorem 33, Theorem 34). In addition, BASS can find further applications, particularly for various kinds of internal queries. We discuss the *range longest common substring (RLCS)* problem as a showcase. Specifically, given two strings S and T, each query $\text{LCS}(l,r)$ requires computing the longest common substring between S and $T[l,r]$. With BASS, we can construct a data structure with $O(n)$ preprocessing time and constant query time, achieving optimality (Theorem 38) and improving the previous best result of $O(\log n)$ query time [4].

2 Preliminaries

We first give some common definitions and notations for strings, aligned with [23]. Let $T=T_1T_2T_3\cdots T_n$ be a string of length $|T|=n$ over a linearly sortable alphabet Σ. We refer to elements in Σ as letters or characters. For any two indexes $1\le l\le r\le n$, the string constructed by concatenating $T_l,T_{l+1},\cdots,T_r$ is denoted as $T[l,r]$.

Substrings, prefixes, and suffixes

For two strings T and t, t is a substring of T if and only if there exist some indexes $1\le l\le r\le|T|$ such that $T[l,r]=t$. We can also say that t occurs in T or T contains t. A prefix of T is a substring of the form $T[1,i]$ for some $1\le i\le|T|$. Similarly, a suffix of T is a substring of the form $T[i,|T|]$.

Occurrence sets

Given a string T, for any pattern t, $\texttt{occ}_T(t)$ is defined as the set of distinct index pairs (l,r) $(1\le l\le r\le n)$ such that $T[l,r]=t$, indicating the occurrences of t in T. If the dictionary string is not ambiguous, we can also write $\texttt{occ}(t)$.

A **trie** is a tree data structure where each edge is labeled by a single character, with paths from the root to specific nodes forming complete words. For a set S of strings, we denote the trie built by S as $\mathcal{E}(S)$. Also, we denote $\texttt{str}'(u)$ as the string formed by the path from the root to u. If $\texttt{str}'(u)\in S$, we call u a terminal node.

For a string T over a linearly sortable alphabet Σ, the **suffix tree** $\mathcal{T}(T)$ is a compact trie built by all suffixes of T. If a node in trie has only one child and is not a terminal node, the node is compressed. Thus, edges are labeled with character sequences instead of a single character. We denote $\texttt{str(u)}$ as the set of substrings $\texttt{str}'(v)$ for all v that is compressed into u. Thus, the union of all $\texttt{str}(u)$ is all different substrings of T, and we say a node u recognizes a substring t if $t \in \texttt{str}(u)$. Also, it is proved that substrings in $\texttt{str}(u)$ have the same set of start positions. Additionally, for non-root node u, we denote $\texttt{parent}(u)$ to be the parent of u, and by definition, any string in $\texttt{str}(\texttt{parent}(u))$ is a prefix of all strings in $\texttt{str}(u)$. Lastly, we denote $\texttt{maxstr}(u)$ as longest string stored in $\texttt{str}(u)$ and $\texttt{len}(u)$ as the length of $\texttt{maxstr}(u)$.

▶ **Theorem 4** ([29]). *For a given string T over a linearly sortable alphabet Σ, the size of suffix tree $\mathcal{T}(T)$ is $O(|T|)$, and it can be constructed in $O(|T|)$ time.*

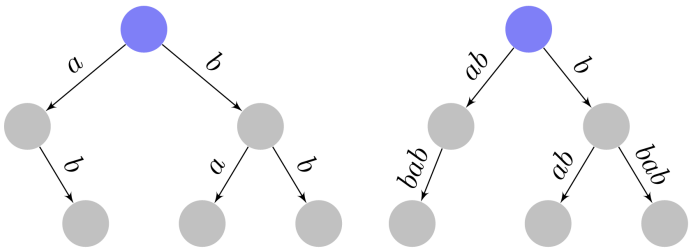

Figure 1 Two illustrative examples: **(a)** $\mathcal{E}(\{\texttt{ab}, \texttt{ba}, \texttt{bb}\})$; **(b)** $\mathcal{T}(\texttt{abbab})$.

3 Basic Substring Structure

3.1 Grid

The *Basic Substring Structure* (BASS) of a string T, denoted by $\texttt{BASS}(T)$, is based on a two-dimensional grid G which maps every substring $T[l, r]$ to a point (l, r). Each row i corresponds to all the substrings of T ending at position i, and each column j corresponds to all the substrings of T starting at position j. Figure 2 (a) gives an example with $T = \texttt{abbab}$.

Recall that $\textsc{Count}(l, r)$ computes the number of all occurrences of all patterns in $T[l, r]$. For every substring $T[l, r]$, we mark its corresponding point (l, r) if and only if $T[l, r] \in \mathcal{D}$. The answer to $\textsc{Count}(l, r)$ is equivalent to the number of marked points within the rectangle $[l, \infty] \times [-\infty, r]$. However, since there could be up to $O(nd)$ marked points, it is inefficient to preprocess them one by one.

3.2 Equivalence Class

Now, we show how these points can be efficiently stored. To do this, we need to introduce the concept of equivalence classes in the BASS. We begin by defining an extension operation for any substring t:

▶ **Definition 5** (Extension). *For any substring t of string T, $\texttt{ext}(t)$ is defined as the longest string s such that **(i)** t is a substring of s, and **(ii)** $|\texttt{occ}(t)| = |\texttt{occ}(s)|$.*

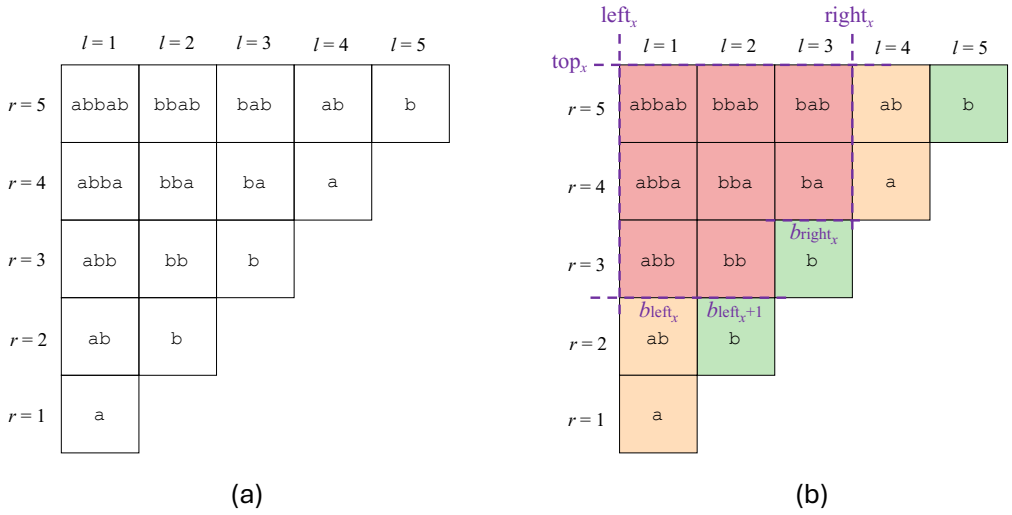

Figure 2 **(a)** The grid of the string `abbab`. **(b)** The blocks correspond to each equivalence class of the string, where each color corresponds to the blocks of an equivalence class.

The extension operation is well defined because, for any substring t, there is only one longest such string s. The proof of this uniqueness can be found in the full version.

For any two substrings t_1 and t_2 of T, we define them as equivalent if and only if $\texttt{ext}(t_1) = \texttt{ext}(t_2)$. That is to say, substrings s that share the same value of $\texttt{ext}(s)$ are classified into the same equivalence class.

▶ Remark 6. The concept of extension is commonly referred to in the literature as maximal repeats, and the equivalence class defined here corresponds exactly to the nodes of the compact acyclic word graph (CDAWG) [12, 25]. However, in the subsequent discussion, we explore the substrings within a single equivalence class in more detail, uncovering a more refined structure that extends the CDAWG.

Now, we will map the equivalence classes to the grid G. To illustrate, we assign a unique color to each equivalence class C and color each cell (i, j) as the color of the equivalence class of $T[i, j]$. Then, we call each maximal connected component with a single color in the grid as a *block*, which exhibits a staircase-like shape as shown in Figure 2 (b). More formally, we have the following lemma.

▶ **Lemma 7.** *For any block B, it can be described by using the following parameters:*

- *Three integers $\texttt{left}_B, \texttt{right}_B, \texttt{top}_B$;*
- *A non-decreasing integer sequence $b_{\texttt{left}_B} \le b_{\texttt{left}_B+1} \le \ldots \le b_{\texttt{right}_B}$;*

Specifically, we have $B = \{(i, j) \mid \texttt{left}_B \le i \le \texttt{right}_B, b_i \le j \le \texttt{top}_B\}$.

Taking the block colored in red in Figure 2 (b) as an example, we have $\texttt{left}_B = 1$, $\texttt{right}_B = 3$ and $\texttt{top}_B = 5$. For each column $i \in [1, 3]$, we have $b_1 = 3, b_2 = 3, b_3 = 4$. These parameters precisely describe the shape and position of the block. Furthermore, as shown in Lemma 8, all blocks from the same equivalence class have the same shape. Therefore, we only preprocess one corresponding block for each equivalence class.

▶ **Lemma 8.** *Given a string T and its grid, for each equivalence class C and pair of blocks $B_x \in C, B_y \in C$, the following satisfies:*

- $\texttt{right}_{B_x} - \texttt{left}_{B_x} = \texttt{right}_{B_y} - \texttt{left}_{B_y}$
- $\forall 0 \le i \le \texttt{right}_{B_x} - \texttt{left}_{B_x}$, $\texttt{top}_{B_x} - b_{\texttt{left}_{B_x}+i} = \texttt{top}_{B_y} - b_{\texttt{left}_{B_y}+i}$

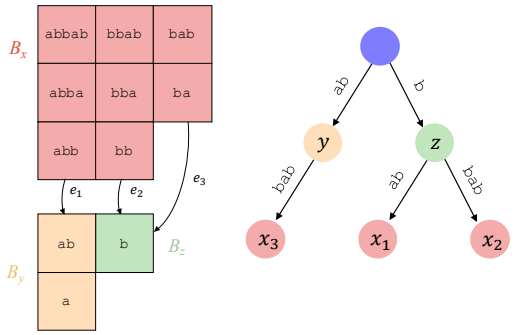

Figure 3 Blocks and $\texttt{PreTree}(T)$.

Table 2 Nodes and relations in Figure 3.

node	column	$\texttt{str}(\cdot)$
x_1	3_{rd} column of B_x	{abbab, abba, abb}
x_2	2_{nd} column of B_x	{bbab, bba, bb}
x_3	1_{st} column of B_x	{bab, ba}
y	1_{st} row of B_y	{ab, a}
z	1_{st} row of B_z	{b}

	from	to	$\texttt{PreTree}(T)$
e_1	1_{st} row of B_x	1_{st} row of B_y	x_3 to y
e_2	2_{nd} row of B_x	2_{nd} row of B_y	x_2 to z
e_3	1_{st} row of B_y	1_{st} row of B_z	x_1 to z

3.3 Relationships between Blocks

There are further relationships between neighboring blocks. Here, we denote two blocks as neighbors if they share a common edge on any row or column in the grid. If we treat each equivalence class as a node and establish edges between neighboring classes, the resulting graph would become the CDAWG. However, the key discovery in this paper is not just the use of equivalence classes but the grid structure itself and its connection to the suffix trees, which has not been explored in previous works. The grid structure provides a powerful framework for organizing and querying these classes.

Recall the definition of Suffix Trees. Given a text T, let $\texttt{PreTree}(T)$ be the suffix tree $\mathcal{T}(T)$, we have the following lemma.

▶ **Lemma 9.** *For an equivalence class C, we consider any block B of C on the grid. For any column i ($\texttt{left}_B \le i \le \texttt{right}_B$), there exists a node $v_{B,i}$ on $\texttt{PreTree}(T)$ such that $\texttt{str}(v_{B,i}) = \{T[i,j] \mid b_i \le j \le \texttt{top}_B\}$.*

The notation $v_{B,i}$ will be used throughout the paper. The following corollary follows:

▶ **Corollary 10.** *Consider two neighboring blocks B_1 and B_2 that share a common edge in column i. Assuming B_1 is on the top of B_2, we have $\texttt{parent}(v_{B_1,i}) = v_{B_2,i}$ on $\texttt{PreTree}(T)$.*

By Corollary 10, the relationship between blocks sharing a column indicates an ancestor-descendant relationship on $\texttt{PreTree}(T)$. Furthermore, if they are adjacent, they have a parent relationship. We further map such relationships onto the grid by adding directed edges. In Figure 3, the blocks B_x, B_y and B_z correspond to the nodes $\{x_1, x_2, x_3\}$, $\{y\}$ and $\{z\}$, respectively. The solid black arrows represent the edges in $\texttt{PreTree}(T)$. Table 2 shows detailed information about these nodes and edges.

Similarly, we build one other tree $\texttt{SufTree}(T)$ to help formalize the relationship between blocks sharing columns, which maintains substrings with common starting positions. Specifically, we build a Suffix Tree of $\mathsf{rev}(T)$. Then, for each node u, we reverse every string in $\texttt{str}(u)$. We denote the tree as $\texttt{SufTree}(T)$. The $\texttt{SufTree}(T)$ with $T = \texttt{abbab}$ is shown in Figure 4. For example, $\texttt{str}(x_2) = \{\texttt{abba}, \texttt{bba}, \texttt{ba}\}$ (after reversion).

▶ **Lemma 11.** *For an equivalence class C, we consider any block B of C in the grid. For any row j ($b_{\texttt{left}_B} \le j \le \texttt{top}_B$), its right boundary r_j is the maximum $i \in [\texttt{left}_B, \texttt{right}_B]$ such that $j \ge b_i$. Furthermore, there exists a node $u_{B,j}$ on $\texttt{SufTree}(T)$ such that $\texttt{str}(u_{B,j}) = \{T[i,j] \mid \texttt{left}_B \le i \le r_j\}$.*

The proof is similar to Lemma 9's. The notation $u_{B,j}$ will also be used throughout the paper. We have the following corollary similar to Corollary 10.

▶ **Corollary 12.** *Consider two neighboring blocks B_1 and B_2 that share a common edge in row j. Assuming B_1 is to the left of B_2, we have $\mathit{parent}(u_{B_1,j}) = u_{B_2,j}$ on $\mathit{SufTree}(T)$.*

By Corollary 12, the relationship between blocks sharing a row indicates an ancestor-descendant relationship on $\texttt{SufTree}(T)$. Furthermore, if they are adjacent, they have a parent relationship. We further map such relation onto the grid by adding directed edges, shown in Figure 4. By Corollary 12 and Corollary 10, we have the following lemma.

▶ **Lemma 13.** *Given a string T and all distinct blocks $B_1, B_2, \ldots, B_k$ in its grid, the sums of the length of the left boundary and the top boundary of all distinct blocks are both $O(n)$. Formally, $\sum_{i=1}^{k}(\mathit{right}_{B_k} - \mathit{left}_{B_k} + 1) = O(n)$ and $\sum_{i=1}^{k}(\mathit{top}_{B_k} - b_{\mathit{left}_{B_k}} + 1) = O(n)$.*

The proofs of Lemmas 7–9 and 13 can be found in the full version.

Now, we are ready to formally introduce the *Basic Substring Structure* $\texttt{BASS}(T)$ of a given text T, which consists of the following components:

- Two modified suffix trees $\texttt{SufTree}(T)$, $\texttt{PreTree}(T)$ and their mapping edges on G.
- $\texttt{left}_B, \texttt{right}_B, \texttt{top}_B$, and $b_{\texttt{left}_B} \le b_{\texttt{left}_B+1} \le \ldots \le b_{\texttt{right}_B}$ for a representative block B in each equivalence class;
- A data structure that locates a given substring $T[l, r]$'s equivalence class and its position within the block in $O(1)$ time.

By Lemma 13, $\texttt{BASS}(T)$ takes up to $O(n)$ memory.

▶ **Lemma 14.** *Given a string T, $\texttt{BASS}(T)$ can be constructed in $O(n)$ time.*

Proof. By [26], $\texttt{PreTree}(T)$ and $\texttt{SufTree}(T)$ can be constructed in $O(n)$. By Lemma 11 and Lemma 9, for each node u in $\texttt{PreTree}(T)$ and $\texttt{SufTree}(T)$, all the strings in $\texttt{str}(u)$ belong to a column (row) of a same equivalence class C. Once we can determine the equivalence class that each node's string belongs to, we can compute the shapes of all blocks in the grid G, i.e., $\texttt{left}_B, \texttt{right}_B, \texttt{top}_B$, and $b_{\texttt{left}_B} \le b_{\texttt{left}_B+1} \le \ldots \le b_{\texttt{right}_B}$, in $O(n)$ time.

To achieve this, we define the representative string $\texttt{rep}(C)$ as the longest string in each equivalence class C, which is unique. Note that, $T[l, r]$ is a representative string if and only if the following satisfies: **(1)** $|\texttt{occ}(T[l, r])| \neq |\texttt{occ}(T[l-1, r])|$ or $l = 1$; **(2)** $|\texttt{occ}(T[l, r])| \neq |\texttt{occ}(T[l, r+1])|$ or $r = n$.

This implies $\texttt{rep}(C)$ can only be among the longest strings of $\texttt{SufTree}(T)$'s node ($\texttt{PreTree}(T)$'s node). Thus, we have

$$\{\texttt{rep}(C) \mid C \in T\text{'s equivalence classes}\} \subseteq \{\texttt{maxstr}(u) \mid u \in \texttt{SufTree}(T)\}$$
$$\{\texttt{rep}(C) \mid C \in T\text{'s equivalence classes}\} \subseteq \{\texttt{maxstr}(u) \mid u \in \texttt{PreTree}(T)\}$$

In addition, computing the number of occurrences of a string can be done in $O(1)$ time on the suffix tree with $O(n)$ preprocessing time. [11] shows that it is possible to locate the node corresponding to any substrings $T[l, r]$ on the suffix tree in $O(1)$ time by using $O(n)$ preprocessing time and space. Therefore, we can take $O(n)$ time to check this for all $\texttt{maxstr}(u)$ from $\texttt{SufTree}(T)$ ($\texttt{PreTree}(T)$), and thus locate all representative strings in the suffix trees.

To infer the equivalence class of each node u in $\texttt{PreTree}(T)$ (or $\texttt{SufTree}(T)$), we first check if $\texttt{maxstr}(u)$ is a representative string. If it is, then u belongs to the equivalence class of $\texttt{maxstr}(u)$. If it is not, we locate $\texttt{maxstr}(u)$ in the opposite suffix tree ($\texttt{SufTree}(T)$ for $\texttt{PreTree}(T)$, and vice versa), denoting it as node v. Since $\texttt{maxstr}(v)$ must be a representative string, u then belongs to the equivalence class of $\texttt{maxstr}(v)$.

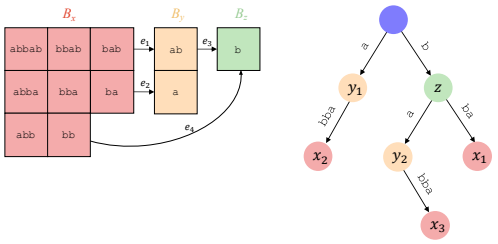

Table 3 Nodes and relations in Figure 4.

node	row	$\texttt{str}(\cdot)$
x_1	1_{st} row of B_x	{abb, bb}
x_2	2_{nd} row of B_x	{abba, bba, ba}
x_3	3_{rd} row of B_x	{abbab, bbab, bab}
y_1	1_{st} row of B_y	{a}
y_2	2_{nd} row of B_y	{ab}
z	1_{st} row of B_z	{b}

	from	to	$\texttt{SufTree}(T)$
e_1	1_{st} row of B_x	1_{st} row of B_y	x_3 to y_2
e_2	2_{nd} row of B_x	2_{nd} row of B_y	x_2 to y_1
e_3	1_{st} row of B_y	1_{st} row of B_z	y_2 to z
e_4	3_{rd} row of B_x	1_{st} row of B_z	x_1 to z

Figure 4 Blocks and $\texttt{SufTree}(T)$.

To implement the data structure that locates a given substring $T[l, r]$, one can first identify its position in $\texttt{PreTree}(T)$, then determine the corresponding equivalence class and the specific row to which it belongs, and ultimately return the desired grid cell. ◀

3.4 Optimal Algorithm for Occurrence Counting

After building $\texttt{BASS}(T)$, we can mark only $O(d)$ points to answer Count query, optimized from the algorithm mentioned in Section 3.1. Recall the naive algorithm to answer $\textsc{Count}(l, r)$. We have $\textsc{Count}(l, r) = \sum_{i=l}^{r} \sum_{j=i}^{r} \mathbb{1}\{T[i, j] \in \mathcal{D}\}$. We use partial sums on BASS to optimize the above algorithm. Define

$$\texttt{PreCount}(T[i, j]) = \sum_{k=i}^{j} \mathbb{1}\{T[i, k] \in \mathcal{D}\},\ \texttt{SubCount}(T[i, j]) = \sum_{k=i}^{j} \texttt{PreCount}(T[k, j]).$$

We can see that $\textsc{Count}(l, r)$ is equal to $\texttt{SubCount}(T[l, r])$. However, storing these values for all substrings $T[i, j]$ is expensive, so we store the following in the data structure:

1. $\texttt{SubCount}(T[i, j])$ for strings located at the left border of each distinct block B.
2. Partial sums of $\texttt{PreCount}(\texttt{maxstr}(\texttt{parent}(v_{B,i})))$ for $\texttt{left}_B \le i \le \texttt{right}_B$ for each distinct block B.
3. A static 2D range counting data structure built on pattern grid cells in B.

Queries can be answered through the following lemma:

▶ **Lemma 15.** *Let B be a block containing the substring $T[l, r]$. Then*

$$\textit{Count}(l, r) = \textit{SubCount}(\textit{maxstr}(\textit{parent}(u_{B,r}))) + \sum_{l \le i \le r, T[i,r] \in B} \textit{PreCount}(\textit{maxstr}(\textit{parent}(v_{B,i}))) + C,$$

where $u_{B,r}$ is the node of $\texttt{SufTree}(T)$ corresponding to row r of B, and C is the answer to a 2D range counting $[l, \infty] \times [-\infty, r]$ only considering the pattern grid cells in the block B.

The first term is already stored in the data structure, as $\texttt{maxstr}(\texttt{parent}(u_{B,r}))$ is on the left border. The second term can be computed in constant time by the partial sums. The last term is handled by the static 2D range counting data structure.

Now we show how to preprocess these values efficiently. First, locating patterns in BASS costs $O(d)$ time. Then, we adopt the following result from [15] to construct the 2D range counting data structure:

▶ **Theorem 16** ([15]). *There is a data structure that can preprocess n points in the plane in $O(n\sqrt{\log n})$ time, using $O(n)$ words of space, and count the number of points dominated by a query point in $O(\log n/\log\log n)$ time.*

To preprocess the remaining information, we need to preprocess $\texttt{PreCount}(\texttt{maxstr}(v))$ for all nodes $v \in \texttt{PreTree}(T)$, then do a partial sum, and compute $\texttt{SubCount}(\texttt{maxstr}(u))$ for all node $u \in \texttt{SufTree}(T)$. First, $\texttt{PreCount}(\texttt{maxstr}(v))$ is the number of patterns that is a prefix of $\texttt{maxstr}(v)$, so it can be computed by getting the number of patterns located on each node of $\texttt{PreTree}(T)$ and summing up these values along the edges between the root and v. Computing $\texttt{SubCount}(\texttt{maxstr}(u))$ is a bit more difficult. We also use the equation of Lemma 15. The first two terms can be computed in $O(1)$ time after taking a partial sum on $\texttt{PreCount}$. For the last term C, notice that all $\texttt{maxstr}(u)$ in a block B have the same and smallest l-coordinate, $\texttt{left}_B$. Thus, the answer to the 2D range counting query is the number of patterns on or below the line of u in block B. Consequently, after preprocessing the number of patterns on each node of $\texttt{SufTree}(T)$, values of $\texttt{SubCount}(\texttt{maxstr}(u))$ in a block B can be computed in time linear to the height of the block. Thus, preprocessing costs $O(n + d\sqrt{\log n})$ time in total. Putting pieces together, we have the following theorem:

▶ **Theorem 2** (Count). *$\textsc{Count}(l, r)$ can be answered in $O(\frac{\log n}{\log\log n})$ time with a data structure of size $O(n + d)$ that can be constructed in $O(n + d\sqrt{\log n})$ time.*

Moreover, we can further prove that our algorithm is equivalent to 2D range counting.

▶ **Theorem 17.** *If $\textsc{Count}(l, r)$ can be answered in $O(Q(n, d))$ time after $O(T(n, d))$ time preprocessing, then for every 2D range counting instance with n points whose coordinates are in $\{1, 2, \ldots, n\}$, there is a data structure that can answer queries in $O(1 + Q(2n, n))$ time after $O(n + T(2n, n))$ time preprocessing.*

On the other hand, the optimal query time complexity for the 2D range counting problem is $O(\frac{\log n}{\log\log n})$ when the data structure has a size of $\tilde{O}(n)$ [45]. Consequently, our query time for $\textsc{Count}(l, r)$ queries is optimal. Moreover, the preprocessing time of $O(d\sqrt{\log n})$ is currently the state-of-the-art in this field.
The proofs of Lemma 15 and Theorem 17 can be found in the full version.

4 On Querying Distinct Patterns

In this section, we discuss the CountDistinct queries, answering the open question from [16] affirmatively. To answer CountDistinct queries, we convert them into a data structure problem using BASS and Count queries in Section 3.4. Our main result is as follows:

▶ **Theorem 1** (CountDistinct). *$\textsc{CountDistinct}(i, j)$ can be answered in $O(\log n)$ time with a data structure of $O(n\log^2 n + d)$ size that can be constructed in $O(n\log^2 n + d\sqrt{\log n})$ time.*

4.1 Problem Decomposition

Given a text T, we consider $\texttt{PreTree}(T)$ of $\texttt{BASS}(T)$. For each node u in $\texttt{PreTree}(T)$, define $f_u = \texttt{PreCount}(\texttt{maxstr}(u))$, counting the number of unique patterns in $\mathcal{D}$ that can be recognized by u itself and the ancestors of u in $\texttt{PreTree}(T)$. We also define a trie $\texttt{Trie}(T[l,r])$ built on all suffixes of $T[l,r]$. For a node u of the trie, let $\texttt{str}'(u)$ be the string formed by the path from the root to u. We define $f'_u = \texttt{PreCount}(\texttt{str}'(u))$. Now, we have the following lemma that calculates $\textsc{CountDistinct}(l,r)$ on $\texttt{Trie}(T[l,r])$:

▶ **Lemma 18.**

$$\begin{aligned}\textsc{CountDistinct}(l,r) = &\underbrace{\sum_{u \text{ is a leaf node of } \texttt{Trie}(T[l,r])} f'_u}_{A_1} \\ &- \underbrace{\sum_{u \text{ is a non-leaf node of } \texttt{Trie}(T[l,r])} (\texttt{cnt}'_u - 1) f'_u}_{A_2} \\ &= A_1 - A_2,\end{aligned}$$

where $\texttt{cnt}'_u$ *represents the number of children of* u *in* $\texttt{Trie}(T[l,r])$.

We have divided the CountDistinct queries into two parts, A_1 and A_2. Now, we further explain what they are. To assist, we define the following key position:

▶ **Definition 19** (Key Position). *Given a string* T *and a query* $\textsc{CountDistinct}(l,r)$*, the key position* $\texttt{pos}$ *is defined as the index in* $[l,r]$ *such that* $T[\texttt{pos},r]$ *is the shortest suffix that appears exactly once in* $T[l,r]$.

With the definition of the key position, we can show that A_1 can be computed as follows:

▶ **Lemma 20.** $A_1 = \textsc{Count}(l,r) - \textsc{Count}(\texttt{pos}+1, r)$.

Then, A_2 can be defined on $\texttt{PreTree}(T)$ with the following lemma:

▶ **Lemma 21.** $A_2 = \sum_{u \in \texttt{PreTree}(T), \texttt{cnt}_u \geq 1} (\texttt{cnt}_u - 1) f_u$*, where* $\texttt{cnt}_u$ *represents the number of children of* u *in* $\texttt{PreTree}(T)$ *that recognizes at least one substring in* $T[l,r]$.

Combining Lemma 18 with Lemma 20 and 21, we get rid of the auxiliary trie and obtain:

▶ **Lemma 22.**

$$\textsc{CountDistinct}(l,r) = \textsc{Count}(l,r) - \textsc{Count}(\texttt{pos}+1,r) - \sum_{u \in \texttt{PreTree}(T), \texttt{cnt}_u \geq 1} (\texttt{cnt}_u - 1) f_u,$$

where $\texttt{cnt}_u$ *represents the number of children of* u *in* $\texttt{PreTree}(T)$ *that recognize at least one substring in* $T[l,r]$.

To efficiently answer $\textsc{CountDistinct}(l,r)$, we need: **(1)** Find the key position $\texttt{pos}$. **(2)** Compute $\textsc{Count}(l,r)$ and $\textsc{Count}(\texttt{pos}+1,r)$ by the algorithm in Section 3.4. **(3)** Compute $A_2 = \sum_{u \in \texttt{PreTree}(T), \texttt{cnt}_u \geq 1} (\texttt{cnt}_u - 1) f_u$. In Section 4.2, we discuss the solutions to **(1)**. **(3)** is discussed in detail in Section 4.3.
The proofs of Lemmas 18, 20, and 21 can be found in the full version.

4.2 Finding Key Positions

Recall the definition of key positions: $T[\texttt{pos}, r]$ occurs exactly once in $T[l, r]$, which means that the previous occurrence of $T[\texttt{pos}, r]$ has an end position smaller than $l + r - \texttt{pos}$. Thus, we define $a_0^{(r)}(t)$ to be the end position of the last occurrence of the string t in $T[1, r]$ (0 if no occurrence). By definition, $\texttt{pos}$ is the largest number that has $a_0^{(r-1)}(T[\texttt{pos}, r]) < l + r - \texttt{pos}$. Consider each node u in $\texttt{SufTree}(T)$. Since all strings in $\texttt{str}(u)$ share the same end positions, these strings have the same value of $a_0^{(r)}(t)$ for every fixed r. So we can regard $a_0^{(r)}(t)$ as a variable attached to each node of $\texttt{SufTree}(T)$, and we define $a^{(r)}(u) = a_0^{(r)}(\texttt{maxstr}(u))$.

Now we show how to find the key position for a query CountDistinct(l, r) if $a^{(r)}(u)$ is maintained for all $0 \le r \le n$ and $u \in \texttt{SufTree}(T)$. Let x_r be the node that $T[1, r] \in \texttt{str}(x_r)$.

▶ **Lemma 23.** *Let $y \in \texttt{SufTree}(T)$ be the lowest-depth node on the path from the root to x_r that satisfies $a^{(r-1)}(y) - \texttt{len}(y) + 1 < l$. We have $\texttt{pos} = \min\{r + l - a^{(r-1)}(y) - 1, r - \texttt{len}(\texttt{parent}(y))\}$, where we define $\texttt{len}(\texttt{parent}(y)) = 0$ when y is the root of $\texttt{SufTree}(T)$.*

4.2.1 Data Structure

Now we introduce a data structure that supports finding the node y stated in Lemma 23 for a given query range $[l, r]$. We focus on the difference between $a^{(r-1)}$ and $a^{(r)}$. The changes take place in strings having a new occurrence, that is, they are a suffix of $T[1, r]$. We have

$$a^{(r)}(u) = \begin{cases} r & u \text{ is on the path from the root to } x_r \\ a^{(r-1)}(u) & \text{otherwise.} \end{cases}$$

Then, we define the following events to characterize the difference between $a^{(r-1)}$ and $a^{(r)}$.

▶ **Definition 24.** *An event (i, j, u, v) is defined as follows:*

- *u and v are vertices in $\texttt{SufTree}(T)$ satisfying $u = v$ or v is u's ancestor.*
- *$a^{(i-1)}(x) = j, a^{(i)}(x) = i$ for all vertices x on the path from u to v.*

Then, the following lemma shows that all these changes (for $r = 1, 2, \ldots, n$) can be represented by a near-linear number of events. We utilize the *Link-Cut Tree* [49] here.

▶ **Lemma 25.** *There exists sets of events $E_1, E_2, \ldots, E_n$ satisfying:*

- *The paths from u to v of all events $(i, j, u, v) \in E_r$ do not intersect and their union form the path from x_r to the root.*
- *$a^{(r)}$ can be obtained from $a^{(r-1)}$ by changing j to i for all nodes on the path from u to v for all events (i, j, u, v) in E_r.*
- *$\sum_{i=1}^{n} |E_i| = O(n \log n)$ and $E_1, E_2, \ldots, E_n$ can be constructed in $O(n \log n)$ time.*

These sets of events give us enough information to recover all the values of $a^{(r)}(u)$. When querying the key position, we only care about the values of $a^{(r-1)}(u)$ for u on the path from the root to x_r, and these values are stored by those j in events $(i, j, u, v) \in E_r$. We now show how to find the key position by Lemma 23 when given E_r. The procedure has two steps. The first step is to find the node y with the smallest depth that satisfies $a^{(r-1)}(y) - \texttt{len}(y) + 1 < l$. Note that the value of $a^{(r-1)}(y)$ decreases on the path from the root to x_r because every time the longer substring occurs, the shorter substring also occurs. Thus, $a^{(r-1)}(y) - \texttt{len}(y) + 1$ also decreases on the path from the root to x_r. Consequently, we can use a binary search on E_r to locate y. The second step is to find $\texttt{pos}$ when given y, which is a constant-time evaluation. Overall, combined with Lemma 23, we have the following lemma:

▶ **Lemma 26.** *There exists a data structure of $O(n \log n)$ size that finds the key position* `pos` *of* CountDistinct(l, r) *in $O(\log n)$ time with a precomputation of $O(n \log n)$ time.*

The proofs of Lemmas 23 and 25 can be found in the full version.

4.3 Computing A_2

Recall that $A_2 = \sum_{u \in \texttt{PreTree}(T), \texttt{cnt}_u > 1} (\texttt{cnt}_u - 1) f_u$, since f_u is fixed by the dictionary $\mathcal{D}$ and does not depend on the query range $[l, r]$, the challenge of computing A_2 is to maintain the values of $\texttt{cnt}_u$ for all nodes in $\texttt{PreTree}(T)$ and different query ranges. Similarly, we make use of the sets of events $E_1, E_2, \ldots, E_n$ defined in Lemma 25, but on $\texttt{PreTree}(T)$. Specifically, since the substrings in $\texttt{str}(v)$ for $v \in \texttt{PreTree}(T)$ have the same starting positions, we define $b^{(l)}(v)$ to be the start position of the first occurrence of $\texttt{maxstr}(v)$ in $T[l, n]$ ($n + 1$ if no occurrence). And, the difference between $b^{(l+1)}$ and $b^{(l)}$ is described by the event set F_l. Formally, we define the events again:

▶ **Definition 27.** *An event (i, j, u, v) is defined as follows:*

- *u and v are vertices in $\texttt{PreTree}(T)$ satisfying $u = v$ or v is u's ancestor.*
- *$b^{(i+1)}(x) = j, b^{(i)}(x) = i$ for all vertices x on the path from u to v.*

And after defining x_l to be the node such that $T[l, n] \in \texttt{str}(x_l)$, we present the following:

▶ **Lemma 28.** *There exist sets of events $F_1, F_2, \ldots, F_n$ satisfying:*

- *The paths from u to v of all events $(i, j, u, v) \in F_l$ do not intersect and their union form the path from x_l to the root.*
- *$b^{(l)}$ can be obtained from $b^{(l+1)}$ by changing j to i for all nodes on the path from u to v for all events (i, j, u, v) in F_l.*
- *$\sum_{i=1}^{n} |F_i| = O(n \log n)$ and $F_1, F_2, \ldots, F_n$ can be constructed in $O(n \log n)$ time.*

Let $A_2(l, r)$ represent the value of A_2 when the query range is $[l, r]$. Assuming that we already know the value of $A_2(l + 1, r)$, we consider how to derive the value of $A_2(l, r)$. The difference comes from the substrings that occur in $T[l, r]$ but not in $T[l + 1, r]$, changing the value of `cnt`. Assume that we have already maintained the values of `cnt` for every node in $\texttt{PreTree}(T)$ when the query range is $[l + 1, r]$. For an event $(i, j, u, v) \in F_l$, we consider how it changes the values of `cnt`. During the event, we will change the value of $b^{(l+1)}(x)$ from j to i for all vertices x on the path from u to v, meaning that some of the vertices x may recognize substrings in $T[l, r]$ while they recognize no substrings in $T[l + 1, r]$. This possibly changes the values of $\texttt{cnt}_x$ for the vertices x in the path from $\texttt{parent}(u)$ to $\texttt{parent}(v)$.

▶ **Lemma 29.** *For each vertex x on the path from u to v, $\texttt{cnt}_{parent(x)}$ increases by one if and only if $r \in [i + \texttt{len}(\texttt{parent}(x)), j + \texttt{len}(\texttt{parent}(x)))$.*

Now we know that when moving the query range from $[l+1, r]$ to $[l, r]$, some of the values of `cnt` increase by 1. Recall that $A_2 = \sum_{u \in \texttt{PreTree}(T), \texttt{cnt}_u > 1} (\texttt{cnt}_u - 1) f_u$. Thus, $A_2(l, r)$ is equal to $A_2(l + 1, r)$ plus the sum of $f_{\texttt{parent}(x)}$ for those x whose $\texttt{cnt}_{\texttt{parent}(x)}$ increased and with $\texttt{cnt}_{\texttt{parent}(x)} \geq 2$. By the following lemma, we show that, for a single event, although multiple $\texttt{cnt}_{\texttt{parent}(x)}$ increased, only one $\texttt{cnt}_{\texttt{parent}(x)}$ can have $\texttt{cnt}_{\texttt{parent}(x)} \geq 2$.

▶ **Lemma 30.** *Given an event $(i, j, u, v) \in F_l$, let x, y be any pair of vertices on the path from u to v that satisfies y is the parent of x. For all $r \in [i + \texttt{len}(y), j + \texttt{len}(y))$, $\texttt{cnt}_y = 1$.*

Thus, the only possible $\texttt{cnt}_{\texttt{parent}(x)} \geq 2$ is at the top of the path, the node $\texttt{parent}(v)$. The following lemma describes when $\texttt{cnt}_{\texttt{parent}(v)} \geq 2$ is true:

▶ **Lemma 31.** *Given an event $(i, j_0, u_0, v_0) \in F_l$ and another event $(i, j_1, u_1, v_1) \in F_l$ where $u_1 = \texttt{parent}(v_0)$, $\texttt{cnt}_{u_1}$ increased to $\texttt{cnt}_{u_1} \geq 2$ if and only if $r \in [j_1 + \texttt{len}(u_1), j_0 + \texttt{len}(u_1))$.*

4.3.1 Final Construction

Now we conclude the formula for A_2. For $l = n, n-1, \ldots, 1$, we perform the following:

1. Sort events in F_l by the order of the paths from x_l to the root.
2. Let $A_2(l, r) \leftarrow A_2(l+1, r)$ for all $l+1 \leq r \leq n$ and $A_2(l, l) \leftarrow 0$.
3. For all adjacent events $(i, j_0, u_0, v_0) \in F_l$ and $(i, j_1, u_1, v_1) \in F_l$ where $u_1 = \texttt{parent}(v_0)$, let $A_2(l, r) \leftarrow A_2(l, r) + f_{u_1}$ for all $r \in [j_1 + \texttt{len}(u_1), j_0 + \texttt{len}(u_1))$.

By previous lemmas, we prove that the correct value of $A_2(l, r)$ is obtained after this procedure. Since the total number of events is $O(n \log n)$, the above process can be maintained by $O(n \log n)$ range additions with persistent data structures [28]: With a persistent segment tree, we answer each A_2 query in $O(\log n)$ time. We have Lemma 32 for computing A_2.

▶ **Lemma 32.** *There exists a data structure of $O(n \log^2 n)$ size that computes the value of A_2 of CountDistinct(l, r) in $O(\log n)$ time with a precomputation of $O(n \log^2 n)$ time.*

The proofs of Lemmas 29–31 can be found in the full version.

4.4 Conclusion

Applying Lemma 26 and Lemma 32 to the algorithm scheme discussed in Section 4.1, we obtain the results of Theorem 1. In the study by Charalampopoulos et al. [16], the authors examined the specific scenario where the dictionary comprises square strings. They demonstrated an algorithm that achieves a query time of $O(\log n)$ with space and preprocessing time complexities of $O(n \log^2 n)$. Our algorithm matches this. Furthermore, when the dictionary consists of all the distinct substrings of T, computing A_1 is straightforward since the result for Count(l, r) queries is given by $\frac{(r-l+1)(r-l+2)}{2}$. In addition, A_2 can be obtained by the algorithm above since $f_u = \texttt{len}(u)$ is easy to compute. Consequently, CountDistinct(l, r) queries can be resolved in $O(\log n)$ time, utilizing $O(n \log^2 n)$ space and preprocessing time.

References

1 Paniz Abedin, Arnab Ganguly, Solon P Pissis, and Sharma V Thankachan. Efficient data structures for range shortest unique substring queries. *Algorithms*, 13(11):276, 2020. doi:10.3390/A13110276.

2 Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

3 S. Alstrup, G. Stolting Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 198–207, 2000. doi:10.1109/SFCS.2000.892088.

4 Amihood Amir, Panagiotis Charalampopoulos, Solon P Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/S00453-020-00744-0.

5 Amihood Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994. doi:10.1016/S0022-0000(05)80047-9.

6 Amihood Amir, Martin Farach, Ramana M Idury, Johannes A Lapoutre, and Alejandro A Schaffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995. doi:10.1006/INCO.1995.1090.

7 Maxim Babenko, Pawel Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 572–591. SIAM, 2014.

8 Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, and Solon P Pissis. Internal shortest absent word queries in constant time and linear space. *Theoretical Computer Science*, 922:271–282, 2022. doi:10.1016/J.TCS.2022.04.029.

9 Ricardo Baeza-Yates and Gonzalo Navarro. Fast approximate string matching in a dictionary. In *Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No. 98EX207)*, pages 14–22. IEEE, 1998. doi:10.1109/SPIRE.1998.712978.

10 Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing All Distinct Squares in Linear Time for Integer Alphabets. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.22.

11 Djamal Belazzougui, Dmitry Kosolobov, Simon J Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In *32nd Annual Symposium on Combinatorial Pattern Matching*, 2021.

12 Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM (JACM)*, 34(3):578–595, 1987. doi:10.1145/28869.28873.

13 Leonid Boytsov. Indexing methods for approximate dictionary searching: Comparative analysis. *Journal of Experimental Algorithmics (JEA)*, 16:1–1, 2011. doi:10.1145/1963190.1963191.

14 Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *Proceedings of the sixteenth annual ACM-SIAM symposium on discrete algorithms*. Society for Industrial and Applied Mathematics., 2005.

15 Timothy M Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 161–173. SIAM, 2010.

16 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Counting Distinct Patterns in Internal Dictionary Matching. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CPM.2020.8.

17 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/S00453-021-00821-Y.

18 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Circular pattern matching with k mismatches. *Journal of Computer and System Sciences*, 115:73–85, 2021. doi:10.1016/J.JCSS.2020.07.003.

19 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P Pissis, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. Approximate circular pattern matching. In *ESA 2022-30th Annual European Symposium on Algorithms*, 2022.

20 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 978–989. IEEE, 2020. doi:10.1109/FOCS46700.2020.00095.

21 Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster pattern matching under edit distance: A reduction to dynamic puzzle matching and the seaweed monoid of permutation matrices. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 698–707. IEEE, 2022. doi:10.1109/FOCS54457.2022.00072.

22 Aleksander Cislak and Szymon Grabowski. A practical index for approximate dictionary matching with few mismatches. *COMPUTING AND INFORMATICS*, 36(5):1088–1106, December 2017. doi:10.4149/CAI_2017_5_1088.

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

24 Maxime Crochemore, Costas S Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. doi:10.1016/J.TCS.2013.11.018.

25 Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Annual Symposium on Combinatorial Pattern Matching*, pages 116–129. Springer, 1997. doi:10.1007/3-540-63220-4_55.

26 Maxime Crochemore and Renaud Vérin. On compact directed acyclic word graphs. *Structures in Logic and Computer Science: A Selection of Essays in Honor of A. Ehrenfeucht*, pages 192–211, 1997. doi:10.1007/3-540-63246-8_12.

27 Debarati Das, Tomasz Kociumaka, and Barna Saha. Improved Approximation Algorithms for Dyck Edit Distance and RNA Folding. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*, volume 229 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 49:1–49:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.49.

28 James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, 1986. doi:10.1145/12130.12142.

29 Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM (JACM)*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

30 Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *Annual Symposium on Combinatorial Pattern Matching*, pages 36–48. Springer, 2006. doi:10.1007/11780441_5.

31 Gaston H Gonnet, Ricardo A Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. *Information Retrieval: Data Structures & Algorithms*, 66:82, 1992.

32 Richard Groult, Élise Prieur, and Gwénaël Richomme. Counting distinct palindromes in a word in linear time. *Information Processing Letters*, 110(20):908–912, 2010. doi:10.1016/J.IPL.2010.07.018.

33 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

34 Wing-Kai Hon, Tak-Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Succinct index for dynamic dictionary matching. In *Algorithms and Computation: 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings 20*, pages 1034–1043. Springer, 2009. doi:10.1007/978-3-642-10631-6_104.

35 Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. Linear-Time Computation of Cyclic Roots and Cyclic Covers of a String. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, volume 259 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CPM.2023.15.

36 Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30–July 4, 2003 Proceedings 30*, pages 943–955. Springer, 2003. doi:10.1007/3-540-45061-0_73.

37 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/J.TCS.2013.10.010.

38 Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

39 Tomasz Kociumaka. *Efficient data structures for internal queries in texts*. PhD thesis, University of Warsaw, 2019.

40 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 532–551. SIAM, 2014.

41 Gad M Landau and Uzi Vishkin. Fast string matching with k differences. *Journal of Computer and System Sciences*, 37(1):63–78, 1988. doi:10.1016/0022-0000(88)90045-1.

42 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

43 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *2009 data compression conference*, pages 193–202. IEEE, 2009. doi:10.1109/DCC.2009.42.

44 Naoaki Okazaki and Jun'ichi Tsujii. Simple and efficient algorithm for approximate dictionary matching. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 851–859, 2010. URL: https://aclanthology.org/C10-1096/.

45 Mihai Patrascu. Lower bounds for 2-dimensional range counting. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 40–46, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250790.1250797.

46 Jakub Radoszewski and Juliusz Straszyński. Efficient Computation of 2-Covers of a String. In *28th Annual European Symposium on Algorithms (ESA 2020)*, volume 173 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 77:1–77:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.77.

47 Mikhail Rubinchik and Arseny M Shur. Counting palindromes in substrings. In *International Symposium on String Processing and Information Retrieval*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.

48 Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 320–328. IEEE, 1996.

49 Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, 1981.

50 Swati Tevatia and Rajesh Prasad. Multi-patterns parameterised matching with application to computational biology. *International Journal of Information and Communication Technology*, 7(4-5):469–480, 2015. doi:10.1504/IJICT.2015.070319.

51 Zhe Zhou, Tao Zhang, Sherman SM Chow, Yupeng Zhang, and Kehuan Zhang. Efficient authenticated multi-pattern matching. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 593–604, 2016.

A More Applications

A.1 Exists and Report

There are two more fundamental types of queries under the internal setting: Exists and Report. While previous results already approach nearly optimal efficiency, we provide alternative algorithms with our proposed BASS.

A.1.1 Exists

First, we consider Exists(l, r) queries, determining if there exists a pattern from $\mathcal{D}$ that occurs in $T[l, r]$. Our algorithm performance is formally stated as follows:

▶ **Theorem 33** (Exists). *Exists(l, r) can be answered in $O(1)$ time with a data structure of size $O(n + d)$ that can be constructed in $O(n + d)$ time.*

Our algorithm for EXISTS(l, r) queries works in a similar fashion to COUNT. Recall that in Section 3.4, we can compute

$$\sum_{i=l}^{r}\sum_{j=i}^{r}\mathbb{1}\{T[i,j]\in\mathcal{D}\wedge T[i,j]\notin B\}$$

in $O(1)$ time after $O(n+d)$ preprocessing, where B is the block to which $T[l,r]$ belongs. If this value is nonzero, the algorithm returns with positive validation; otherwise, EXISTS(i,j) = `True` if and only if there exists $[i,j]\subseteq[l,r]$ such that $T[i,j]\in\mathcal{D}$ and $T[i,j]\in B$. It suffices to show how the existence of such $[i,j]$ can be decided in $O(1)$ time given $[l,r]$.

To solve this, we consider each equivalence class in BASS(T) independently. Similar to Section 3.4, we only need to consider an arbitrary block of this class. By regarding the pattern cells in the block as m normalized points in the 2D plane, this problem is thus equivalent to the following:

▶ **Problem 1.** Given m points (x_i, y_i) on the 2D plane ($x_i, y_i \in [1, m]$), check whether the region $[l, +\infty)\times(-\infty, r]$ contains any point.

By precomputing the values of $\min_{(x_i,y_i):x_i\geq v} y_i, \forall v\in[1,m]$ in $O(m)$ time, each query to this problem can be answered in $O(1)$ time. Recall that the sum of m for all equivalence classes is $O(d)$. We thus prove Theorem 33.

A.1.2 Report

Then, we answer REPORT(l, r) queries, which reports all occurrences of all patterns of $\mathcal{D}$ in $T[l, r]$. Our algorithm performance is formally stated as follows:

▶ **Theorem 34** (Report). *REPORT(l, r) can be answered in $O(1+x)$ time with a data structure of size $O(n + d)$ that can be constructed in $O(n + d)$ time, where x denotes the length of output.*

Our algorithm for REPORT consists of two parts:

1. We compute a set $\mathcal{R}$ consisting of all indices $j\in[l,r]$ such that there exists a pattern in $T[l,r]$ with the end position j.
2. For each j in $\mathcal{R}$, we report a set $\mathcal{L}(j)$ of the start positions i such that $T[i,j]$ is a pattern.

Our goal is to finish the first part in $O(|\mathcal{R}|)$ time and the second part in $O(|\mathcal{R}| + \sum_{j\in\mathcal{R}}|\mathcal{L}(j)|)$ time. In this way, the total query time will be $O(x)$, and Theorem 34 is achieved.

A.1.2.1 Reporting $\mathcal{L}(j)$

We first consider the second part. Similar to Section 3.4, for each equivalence class C in BASS(T), it suffices to consider an arbitrary block B. For each block B considered, we precompute the following values for each row j of B: **(1)** all pattern grid cells $T[i,j]\in\mathcal{D}$ in row j of B in the increasing order of i and **(2)** the leftmost pattern grid cell $T[i,j]\in\mathcal{D}\setminus B$ that is in row j and to the right of B (Corollary 12).

By Lemma 13, the total number of rows over all distinct blocks is $O(n)$, so we can compute all the above information in $O(n + d)$ time. With these precomputed values, we can report the start positions in $\mathcal{L}(j)$ by first locating the block of $T[l,j]$ and then reporting the pattern grid cells in row l from left to right, within time $O(1)$ for each pattern. This exactly gives the $O(|\mathcal{R}| + \sum_{j\in\mathcal{R}}|\mathcal{L}(j)|)$ time complexity in total for the second part.

A.1.2.2 Finding $\mathcal{R}$

Now, we focus on how to find $\mathcal{R}$ efficiently. Recall that for each row j, $j \in \mathcal{R}$ if and only if there exists a start position $i (l \le i \le j)$ such that $T[i, j]$ is a pattern. To check this in $\texttt{BASS}(T)$, we may first locate the cell $T[l, j]$, and then see whether there exists a pattern grid cell (i, j) on the right of (l, j). Similar to the process of finding $\mathcal{L}(j)$s, for a fixed substring $T[l, j_1]$ that has a pattern as its suffix, we can maintain the preceding row $j_2 < j_1$ in the column l such that $T[l, j_2]$ also has a pattern as its suffix. However, this might take $\Theta(n^2)$ time and space.

To speed up the above process, we record the preceding equivalence class instead of the row. For each equivalence class C, we still consider only one of its blocks B similar to Section 3.4. We only maintain the "predecessor" in a different block for each column i in B, which only takes $O(n)$ space by Lemma 13. Formally, given a block B and a column i in B, we will maintain the top row j satisfying **(1)** j is in a block under B and **(2)** there exists a $k \in [i, j]$ such that $T[k, j] \in \mathcal{D}$. Dynamic programming on the grid can compute this in $O(n + d)$ time.

With this information, we can find the set of blocks that contain at least one row in $\mathcal{R}$. In addition, each row in $\mathcal{R}$ occurs in exactly one block of this set. This takes $O(|\mathcal{R}|)$ time by first locating the block that $T[l, r]$ belongs to and then iterating rows with the fixed column l. By doing so, it remains to find the rows r that belong to $\mathcal{R}$ for each of these blocks.

By Lemma 13, given a block B, we may maintain the rightmost pattern grid cell $T[i, j] \in \mathcal{D}$ for each row j $(l \le j \le r)$ in B in $O(n + d)$ time. Denote this by h_j, so that row j is in $\mathcal{R}$ if and only if $l \le h_j$. Now, it suffices to compute the set S_B of indices $j \in [b_l, \min(\texttt{top}_B, r)]$ that satisfies $h_j \ge l$ in $O(|S_B|)$ time. If we view (j, h_j) as points of a 2D plane, this is exactly the 3-sided range reporting problem. This is studied in [3], and is efficiently solved by *Range Minimum Query (RMQ)* algorithms [30] with the desired performance.

A.2 ReportDistinct

Compared to the Report query that finds all occurrences in a substring, ReportDistinct queries only report every pattern once. In [17], it achieves $O(\log n + x)$ query time with $O(n \log n + d)$ preprocessing, where x denotes the length of the output. With BASS, we improve the query time to $O(1 + x)$. Both previous works and our algorithm take $O(n + d)$ memory. The formal theorem is as follows:

▶ **Theorem 3** (ReportDistinct). *ReportDistinct(l, r) can be answered in $O(1 + x)$ time with a data structure of size $O(n + d)$ that can be constructed in $O(n \log n + d)$ time, where x denotes the length of output.*

Let D denote the set of patterns that occur in $T[l, r]$. The key idea is to find the patterns in D that are not suffixes of any longer pattern in D. We denote this set as S. With S, we can locate and report other patterns in $\texttt{SufTree}(T)$ in $O(1 + x)$ time by the following lemma.

▶ **Lemma 35.** *ReportDistinct(l, r) queries can be answered in $O(1 + x)$ time using S, where $x = |D|$.*

Proof. First, we directly report all the patterns in S. For each $t \in D \setminus S$, let v be the node in $\texttt{SufTree}(T)$ recognizing t. Since t is a suffix of some $t' \in S$, v must be an ancestor of the node u recognizing t'. Furthermore, for any node v that is an ancestor of a node u recognizing a substring $t' \in S$, all patterns in $\texttt{str}(v)$ should be reported.

Based on that, starting from the nodes recognizing substrings in S, we can iteratively "lift" each node to its deepest ancestor containing a non-reported pattern, until no node can be lifted. This process ensures that every pattern in D is reported exactly once, achieving $O(1+x)$ time complexity. ◀

We determine S by computing the following two subsets of D. For each $i \in [l,r]$, let p_i be the longest pattern suffix of $T[l,i]$, recognized by node $u_i \in \texttt{SufTree}(T)$. Let v_i denote the node recognizing $T[l,i]$. Then, the first subset is:

$$S_1 = \{p_i \mid u_i = v_i, i \in [l,r]\}.$$

Let q_i be the longest pattern suffix of $T[l,i]$ recognized at an ancestor of v_i. Then, we define the second subset as:

$$S_2 = \{q_i \mid q_i \text{ occurs exactly once in } T[l,i], i \in [l,r]\}.$$

It suffices to find S_1 and S_2 efficiently due to the following lemma:

▶ **Lemma 36.** $S \subseteq S_1 \cup S_2$.

Proof. We first define $S_2' = \{q_i \mid i \in [l,r]\}$. Then,

$$S \subseteq \{p_i \mid i \in [l,r]\} = \{p_i \mid u_i = v_i, i \in [l,r]\} \cup \{p_i \mid u_i \neq v_i, i \in [l,r]\} \subseteq S_1 \cup S_2'.$$

We have $S_2 \subseteq S_2'$, thus it suffices to prove that $(S_2' \setminus S_2) \cap S = \emptyset$. Consider $q_i \in S_2' \setminus S_2$. By definition, q_i occurs at least twice in $T[l,i]$. Let its first occurrence in $T[l,i]$ correspond to the suffix of $T[l,i']$, where $l \leq i' < i$. This implies that q_i is either equals $q_{i'}$ or is a proper suffix of $q_{i'}$.

- Case 1: If $q_i = q_{i'}$, then $q_{i'}$ occurs exactly once in $T[l,i']$, forcing $q_i \in S_2$. This contradicts $q_i \in S_2' \setminus S_2$.
- Case 2: If q_i is a proper suffix of $q_{i'}$, then $q_i \notin S$, as elements of S cannot be proper suffixes of other patterns. ◀

Finding S_1

Let $\mathcal{Q} = \{i \mid u_i = v_i, i \in [l,r]\}$. $\mathcal{Q}$ resembles the set $\mathcal{R}$ in $\textsc{Report}(l,r)$ queries, but $\mathcal{Q}$ includes indices i only if the pattern suffix of $T[l,i]$ is in the same block as $T[l,i]$ in BASS. Thus, by precomputing the rightmost occurrence of patterns in each row of every block, $\mathcal{Q}$ can be efficiently determined using a 3-sided range reporting algorithm [3].

Once Q is identified, we compute S_1', an extension of S_1, as follows: S_1' contains all patterns that are suffixes of $T[l,i]$ and lie in the same block as $T[l,i]$ for all $i \in \mathcal{Q}$. We have $S_1 \subseteq S_1'$, and since each row in the BASS corresponds to distinct strings, patterns in S_1' are unique. To construct S_1', we iterate over patterns in row i (for each $i \in \mathcal{Q}$) from right to left, terminating when a pattern is no longer a suffix of $T[l,i]$. This process runs in $O(|S_1'|)$ time, ensuring the total cost remains within $O(1+x)$, where x is the output size.

Finding S_2

For each node u in $\texttt{SufTree}(T)$, we define z_u as the longest pattern that is a suffix of $\texttt{maxstr}(\texttt{parent}(u))$ (ϵ if not exist). Furthermore, we define w_u as the node in $\texttt{SufTree}(T)$ that recognizes z_u. Since q_i is not recognized by the same node as $T[l,i]$, q_i depends only on v_i – the node that recognizes $T[l,i]$, and satisfies $q_i = z_{v_i}$. Despite that the indices l, i may

be different, whether z_{v_i} appears in $T[l,i]$ once only depends on the string $T[l,i]$ itself. And, for strings recognized by the same node $u \in \texttt{SufTree}(T)$, the longer the string, the higher the chance that z_{v_i} has another occurrence. Thus, for each node $u \in \texttt{SufTree}(T)$, there is a boundary in its BASS row, where only the substrings on its right contain z_u exactly once.

To determine the boundary, we use the array $a^{(r)}(\cdot)$ from Section 4.2. Specifically, z_u appears once in a string $T[j,k] \in \texttt{str}(u)$ if and only if $a^{(k-1)}(w_u) - |z_u| + 1 < j$. This inequality checks whether the previous occurrence of z_u starts before j. We can check for an arbitrary occurrence of strings in $\texttt{str}(u)$. The value of $a^{(k-1)}(w_u)$ can be retrieved from the event set E_{k-1} from Lemma 25, yielding an $O(n \log n)$ preprocessing time.

Finally, indices i contributing to S_2 correspond to the positions where $T[l,i]$ lies to the right of the precomputed boundary for that row. Using 3-sided range reporting (similar to $\mathcal{R}$ in $\textsc{Report}(l,r)$ queries and $\mathcal{Q}$ in the previous S_1 part), S_2 is determined in $O(|S_2|)$ time.

▶ Remark 37. By maintaining only the most recent event set E_r during preprocessing (instead of all historical sets) and immediately dealing with the correlated queries, the memory usage can be further improved to $O(n)$.

A.3 Range Longest Common Substring

Consider the following internal query problem related to the longest common substrings (LCS).

▶ **Problem 2.** Given two strings S and T, each query $\text{LCS}(l,r)$ requires to compute the longest common substring between S and $T[l,r]$.

By using BASS technique, we have the following result.

▶ **Theorem 38** (Range LCS). *$LCS(l,r)$ can be answered in $O(1)$ time with a data structure of $O(|S|+|T|)$ size that can be constructed in $O(|S|+|T|)$ time.*

This improves the previous best result of $O(\log n)$ query time [4] and achieves optimal performance.

We start presenting our data structure by introducing an auxiliary string A. Let $A = Sc_1Sc_2T$ be a string formed by concatenating S, c_1, S, c_2, T, where c_1, c_2 are distinct characters that do not exist in either S or T. We construct $\texttt{BASS}(A)$. Given that c_1 (or c_2) occurs only once in A, every string containing c_1 or c_2 should belong to the same block as the entire string A. Consequently, every substring of S, which occurs in A at least twice, belongs to a block different from A. Equivalently, we have the following lemma:

▶ **Lemma 39.** *Given an equivalence class C of $\texttt{BASS}(A)$, either all strings $t \in C$ occur in S, or none of the strings $t \in C$ occur in S.*

Now, we have the following, where B is the block that $A[l,r]$ belongs to:

$$\text{LCS}(l,r) = \begin{cases} r-l+1 & A[l,r] \text{ occurs in } S \\ \max\limits_{\substack{l \le a \le b \le r \\ A[a,b] \in S \text{ and } A[a,b] \notin B}} \{b-a+1\} & \text{otherwise} \end{cases}$$

One observation is that the optimal (a,b) above must be on some blocks' top or left border. During the preprocessing phase, we store the answer $\text{LCS}(l,r)$ for every (l,r) that is on the border of a block. Since we only need to store the answer for one arbitrary block for each equivalence class, the total information we need is $O(|A|)$.

What remains is that given the information of the blocks to the down or right of B, after some preprocessing within time linear in the perimeter of B, we need to compute $\mathrm{LCS}(l, r)$ in $O(1)$ time. This would give a unified approach for answering queries in constant time, as well as precomputing the required information in linear time.

The answers on the bottom or right border are easy to retrieve. And, for a substring t on the top or left border of B, the answer is the maximum of the answer of the substrings that occurs in t and on the bottom or right border of B. Since the block is staircase-shaped by Lemma 7, if we regard all substrings on the bottom or right border as a sequence, we only need to take the maximum of the answer inside an interval.

As we know that range maximum queries can be answered in constant time by a $O(n)$ time constructible data structure, our algorithm can be done in linear time. When query, similarly, we only need to answer a range maximum query within that block, costing only constant time.

In [4], another problem is also proposed: Given two strings, S and T, of lengths up to n, the task is to construct a data structure that computes the LCS between any prefix or suffix of S and any prefix or suffix of T. By employing BASS, we can develop a data structure of size $O(n \log \log n)$, which answers such queries in $O(\log \log n)$, outperforming the previous best-known result. We believe BASS offers deeper insight into the relationships among substrings and will lead to further applications.