Repetition Aware Text Indexing for Matching Patterns with Wildcards

Daniel Gibney[1]
University of Texas at Dallas, Richardson, TX, USA

Jackson Huffstutler
University of Texas at Dallas, Richardson, TX, USA

Mano Prakash Parthasarathi
North Carolina State University, Raleigh, NC, USA

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

Abstract

We study the problem of indexing a text $T[1\,..\,n]$ to support pattern matching with wildcards. The input of a query is a pattern $P[1\,..\,m]$ containing $h \in [0, k]$ wildcard (a.k.a. don't care) characters and the output is the set of occurrences of P in T (i.e., starting positions of substrings of T that matches P), where $k = o(\log n)$ is fixed at index construction. A classic solution by Cole et al. [STOC 2004] provides an index with space complexity $O(n \cdot (c \log n)^k / k!))$ and query time $O(m + 2^h \log\log n + \mathsf{occ})$, where $c > 1$ is a constant, and occ denotes the number of occurrences of P in T. We introduce a new data structure that significantly reduces space usage for highly repetitive texts while maintaining efficient query processing. Its space (in words) and query time are as follows:

$$O\left(\delta \log(n/\delta) \cdot c^k \left(1 + \frac{\log^k(\delta \log n)}{k!}\right)\right) \quad \textit{and} \quad O((m + 2^h + \mathsf{occ}) \log n))$$

The parameter δ, known as *substring complexity*, is a recently introduced measure of repetitiveness that serves as a unifying and lower-bounding metric for several popular measures, including the number of phrases in the LZ77 factorization (denoted by z) and the number of runs in the Burrows-Wheeler Transform (denoted by r). Moreover, $O(\delta \log(n/\delta))$ represents the optimal space required to encode the data in terms of n and δ, helping us see how close our space is to the minimum required. In another trade-off, we match the query time of Cole et al.'s index using $O(n + \delta \log(n/\delta) \cdot (c \log \delta)^{k+\epsilon}/k!)$ space, where $\epsilon > 0$ is an arbitrarily small constant. We also demonstrate how these techniques can be applied to a more general indexing problem, where the query pattern includes k-gaps (a gap can be interpreted as a contiguous sequence of wildcard characters).

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Pattern Matching, Text Indexing, Wildcard Matching

Digital Object Identifier 10.4230/LIPIcs.ICALP.2025.88

Category Track A: Algorithms, Complexity and Games

Funding *Mano Prakash Parthasarathi*: U.S. National Science Foundation (NSF) award CCF-2316691.
Sharma V. Thankachan: U.S. National Science Foundation (NSF) award CCF-2316691.

1 Introduction and Related Work

Efficient indexing of string or textual data is crucial in fields such as computational biology and information retrieval, enabling fast and accurate search queries. A fundamental task in this context is to preprocess a long string $T[1\,..\,n]$ (the text) into a data structure that

[1] Corresponding author

allows efficient retrieval of *occurrences* of a short string $P[1\,..\,m]$ (the pattern) when queried. An occurrence of P refers to the starting position of a substring in T that matches P either exactly or approximately, depending on the matching criteria. We denote the number of occurrences as occ. In the exact match setting, the well-known suffix tree [51] structure supports such queries in $O(m + \mathsf{occ})$ time. WLOG, we assume that the characters in T are from an integer alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$ and $\sigma \leq n$.

A major drawback of suffix trees (and even closely related suffix arrays) is their space usage, which is $O(n)$ words (equivalently, $O(n \log n)$ bits). This can be significantly larger than the space required to store the text itself, which is $n \log \sigma$ bits. In the early 2000s, a key research challenge was designing indexes that required space closer to that of the text. This led to major breakthroughs, such as compressed suffix arrays/trees [17, 46, 47], FM-index [12] and their refinements. See [40] for an extensive survey on this topic. These innovations enabled encodings that use close to $n \log \sigma$ bits – or even approach entropy-compressed space while supporting suffix tree/array operations efficiently. While these findings are significant, they face major challenges when applied to many modern datasets, such as large collections of DNA sequences or versioned texts. These data sets tend to be highly repetitive, i.e., contain many long, repeated substrings. Although large in size, repetitiveness makes them highly compressible, but statistical entropy-based compressors (and data structures of that space) are often less effective at capturing such redundancy. Instead, dictionary-based compressors perform better in these scenarios [37].

Two popular dictionary-based compression methods are Lempel-Ziv (LZ77) [53] and the (run-length-encoded) Burrows-Wheeler Transform (BWT) [7], which use $O(z)$ and $O(r)$ space, respectively. Here, z denotes the number of phrases in the LZ77 factorization of the text, while r represents the number of runs in the text's BWT. A major recent breakthrough in this area is the introduction of two new measures of repetitiveness – the *string attractor* [25] and the *δ-measure* (a.k.a. substring complexity) [27, 43] - which led to the discovery of the asymptotic relationship between several seemingly distinct measures. A string attractor Γ of $T[1\,..\,n]$ is a subset of the text's positions such that for every substring $T[i\,..\,j]$ there exists $i', j' \in [1, n]$ and $x \in \Gamma$ where $T[i'\,..\,j'] = T[i\,..\,j]$ and $x \in [i', j']$. In linear time, we can compute a string attractor of size z or r; however, finding the smallest string attractor (its size is denoted by γ) is NP-hard. The δ-measure is equal to $\max_t d_t(T)/t$ where $d_t(T)$ is number of distinct substrings of T of length t, which can be easily computed in $O(n)$ time using a suffix tree. An important and unifying result in the field of text compression is that $\delta \leq \gamma \leq z, r = O(\delta \log \delta \cdot \max(1, \log \frac{n}{\delta \log \delta}))$ [23]; also $z = O(\delta \log(n/\delta))$ [43]. We can also bound $\bar{r}$, the number of runs in the BWT of text's reverse by $O(\delta \log \delta \cdot \max(1, \log \frac{n}{\delta \log \delta}))$ [23] since δ (and γ) are invariant under the reversal of the text. In short, the compressibility measures mentioned are always within poly-logarithmic factors of one another and δ lower bounds all others. It is also known that the optimal space required to encode the data in terms of n and δ is $O(\delta \log(n/\delta))$ [28]. This raises an important question:

Can we index the text in (close to) optimal repetition-aware space of $O(\delta \log(n/\delta))$ words and support various pattern matching queries efficiently?

For exact pattern matching, this question has been answered positively; there exists such an optimal space index that supports queries in $O(m + (1 + \mathsf{occ}) \log^\epsilon n)$ time [26], where $\epsilon > 0$ is an arbitrarily small constant. The authors also showed how to achieve $O(m + \mathsf{occ})$ query time using $\log^\epsilon(n/\delta)$ factor more space. The recently introduced structure (called δ-SA) encodes suffix array and inverse suffix array in optimal space and supports random access in poly-logarithmic time [24]. Another structure encodes the suffix tree in $O(r \log(n/r))$ space and supports all basic operations in $O(\log(n/r))$ time [14]. We refer to [38] for a detailed survey on this topic. In summary, data structures for the exact pattern-matching problem have kept pace with the data in terms of space as repetitiveness has increased. However, this adaptation is still in its early stages for more advanced problems; See [1, 16, 39, 36, 45, 44, 50] for a list of problems, where limited progress has been achieved. To this end, we revisit two fundamental problems, which we define formally below.

▶ **Problem 1** (k-Wildcard Indexing). *Given a text $T[1\,..\,n]$ and an integer $k = o(\log n)$, design a data structure (called an index) that supports the following query efficiently.*

- *Input: A pattern $P[1\,..\,m]$ with $h \le k$ wildcards. A wildcard is a character, denoted by #, that matches any character.*
- *Output: The set of occurrences of P in T (denote the output size by* occ*).*

The alphabet set of T is $\Sigma = \{1, 2, \ldots, \sigma\}$ and $\sigma \le n$. The alphabet set of P is $\Sigma \cup \{\#\}$.

This problem can be solved algorithmically in $O(n \log m)$ time for any k using the Fast Fourier Transform (FFT). The first non-trivial indexing solution was given by Cole et al. [10]. Their $O(n(c \log n)^k/k!)$ space structure, known as the k-wildcard errata trie, can answer queries in time $O(m + 2^h \log\log n + occ)$, where $c > 1$ is a fixed constant. There have been several attempts to generalize/improve this classic result for constant k as follows.

- $O(n \log n \log_\beta^{k-1} m)$ space and $O(m + \beta^j \log\log n + \mathsf{occ})$ query time [6] for any $\beta \in [2, \sigma]$.
- $O(n \log^{k-1+\epsilon} n)$ space and $O(m + 2^h \log\log n + \mathsf{occ})$ query time [32].
- $O(n \log^{k-1} n \log \sigma)$ space and $O(m + 2^h \log n + \mathsf{occ})$ query time [32]. We note that this time can be improved to $O(m + 2^h \log\log n + \mathsf{occ})$.

There are also some linear (or slightly better) space indexes that can handle queries with an *unbounded* number h of wildcards. However, they generally have a factor σ^h in the query time, making them comparable to others only when the alphabet size is constant [6, 33, 31]. This includes an index achieving $O(m + \mathsf{occ})$ query time using $O(n(\sigma^k \log\log n)^k)$ space [6]. We present the first set of repetition-aware solutions as summarized below.

▶ **Theorem 1.** *There exists a solution for Problem 1 with space*

$$O\left(\delta \log(n/\delta) \cdot c^k \left(1 + \frac{\log^k(\delta \log n)}{k!}\right)\right) \text{ words and query time } O((m + 2^h + \mathsf{occ}) \log n)).$$

Here δ is the substring complexity and $c > 1$ is a fixed constant.

The query time is improved in the next result, matching that of Cole et al.'s index.

▶ **Theorem 2.** *There exists a solution for Problem 1 with space*

$$O\left(\delta \log(n/\delta) \cdot \frac{(c \log \delta)^{k+\epsilon}}{k!} + \frac{n}{(\log_\sigma n)^{1-\epsilon}}\right) \text{ words and query time } O(m + 2^h \log\log n + \mathsf{occ}).$$

Here δ is the substring complexity, $c > 1$ is a fixed constant and $\epsilon > 0$ is an arbitrarily small constant. The query time can be made $O(m + \mathsf{occ})$ time by maintaining an additional structure of space $O(\delta \log(n/\delta) \cdot (c^k \log\log n)^{k+2}/k!)$.

We also show how our techniques can be extended to design a repetition-aware index for a closely related problem known as *indexing with gaps* [30].

2 Preliminaries

Notation. For a length n string $T[1..n]$, we denote the i^{th} character by $T[i]$. We use "$\cdot$" to denote concatenation between two strings. A *substring* of T starting at position i and ending at position j of T is denoted by $T[i..j] = T[i] \cdot T[i+1] \cdots T[j]$. If $i > j$, then $T[i..j]$ is the empty string. We also use the notation $T[i..j) = T[i] \cdots T[j-1]$ and $T(i..j] = T[i+1] \cdots T[j]$. We use T^R to denote the reverse of the string T. A *suffix* is a string of the form $T[i..n]$, which we denote as $T[i..]$. A suffix $T[i..]$ is called a *proper suffix* if $i > 1$. A *prefix* is a string of the form $T[1..i]$, which we denote as $T[..i]$. A prefix $T[..i]$ is called a *proper prefix* if $i < n$.

Compact Tries and Suffix Trees. Let $\mathcal{S}$ be a set of strings over Σ. We use $\mathcal{T}(\mathcal{S})$ to denote a compact trie constructed over $\mathcal{S}$ after appending each string in $\mathcal{S}$ with a special character $\$ \notin \Sigma$. The character $\$$ is lexicographically smaller than all other characters in Σ. The number of leaves in $\mathcal{T}(\mathcal{S})$ will be exactly $|\mathcal{S}|$. The leaves of $\mathcal{T}(\mathcal{S})$, denoted in left-to-right order are $\ell_1, \ell_2, \ldots, \ell_{|\mathcal{S}|}$ such that ℓ_i corresponds to i^{th} string in $\mathcal{S}$ when $\mathcal{S}$ is sorted in ascending lexicographic order. For a node u (either implicit or explicit[2]) in $\mathcal{T}(\mathcal{S})$, we use $\mathrm{str}(u)$ to denote the string obtained by concatenating edge labels on the root-to-u path and string depth $\mathrm{strlen}(u)$ to denote its length.

Given a pattern $P[1..m]$, and a compact trie $\mathcal{T}$, if $\mathrm{str}(u) = P$ for some node u (either implicit or explicit) in $\mathcal{T}$, then u is called the *locus* of P. Given a locus, the collection of leaves under that locus forms a contiguous range $\ell_a, \ell_{a+1}, \ldots, \ell_b$ and we call $[a, b]$ the *range of the locus*. Observe that P is a prefix of $\mathrm{str}(\ell_x)$ for all $x \in [a, b]$.

The suffix tree [51] of a string $T[1..n]$, denoted as $\mathrm{ST}(T)$, is a compact trie built over all suffixes of T. The suffix tree can be constructed in $O(n)$ time for polynomially-sized integer alphabets [11]. The suffix array $\mathrm{SA}[1..n]$ is an array such that $T[\mathrm{SA}[i]..]$ is the i^{th} suffix when sorted in ascending lexicographic order. The inverse suffix array $\mathrm{ISA}[1..n]$ is an array such that $\mathrm{ISA}[\mathrm{SA}[i]] = i$, or equivalently $\mathrm{ISA}[i]$ is the lexicographic rank of the suffix $T[i..]$. The *longest common extension* of two suffixes $T[i..]$ and $T[j..]$, denoted by $\mathrm{LCE}(i, j)$ is the length of their longest common prefix. LCE queries can be answered in $O(1)$ time using an $O(n)$ space structure [18].

There exists a combination of compact data structures of total space $O(n \log \sigma \cdot \log_\sigma^\epsilon n)$ bits, equivalently $O(n/\log_\sigma^{1-\epsilon} n)$ words, and support all suffix tree functionalities with no slowdown in performance [17, 47]. There also exist indexes of space $O(n \log \sigma)$ bits that supports SA and ISA queries in time $O(\log_\sigma^\epsilon n)$ time [17] and LCE queries in $O(1)$ time [22].

Burrows-Wheeler Transform. The Burrows-Wheeler Transform (BWT) [7] of a text $T[1..n]$, denoted by $\mathrm{BWT}[1..n+1]$ is a permutation of all characters in $T \cdot \$$, such that $\mathrm{BWT}[i]$ is the last character of i^{th} string in the set $\{T[i..] \cdot \$ \cdot T[..i) \mid i \in [1, n]\} \cup \{\$ \cdot T\}$ when sorted lexicographically. The BWT can be computed in linear time. The BWT has been a popular form of compression used in several previous text indexes [3, 34, 35] including the well known FM-index [12]. The popularity of the BWT is due to the following favorable properties: First, the BWT groups characters sharing similar contexts, resulting in a small run-length encoded form when the text is highly repetitive. The letter r denotes the number of maximal unary substrings (called runs) in the BWT. Second, the BWT is amenable to efficient pattern matching and string query procedures.

[2] Branching nodes and leaves are called explicit nodes, and positions on edges are called implicit nodes.

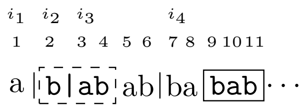

Figure 1 LZ77 -factorization of `abababbabab` (The dashed and the solid boxes capture a primary and a secondary occurence of `bab` respectively).

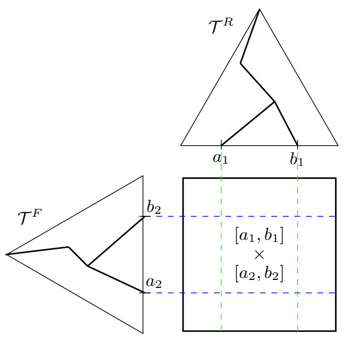

Figure 2 Reduction of finding primary occurrences to 2D range queries.

LZ77 and String Attractor. Lempel-Ziv (LZ77) [53] is another popular type of repetition-based compression. LZ77 partitions the string greedily from left-to-right into what are called *LZ factors* or *phrases*, $T[i_1 \mathinner{.\,.} i_2)$, $T[i_2 \mathinner{.\,.} i_3)$, ... Let z denote the number of factors. The following conditions are satisfied: (i) $T = T[i_1 \mathinner{.\,.} i_2) \cdots T[i_z \mathinner{.\,.}]$ and (ii) $T[i_j \mathinner{.\,.} i_{j+1})$ is either the first occurrence of a character or is the longest substring starting at position i_j that has an occurrence starting at some position $< i_j$. The positions $i_1, \ldots, i_z$ are called phrase boundaries and we can compute them in linear time [49]. Let $T[i \mathinner{.\,.} j] = S$, we say i is a *primary occurrence* of S if a phrase boundary exists in the range $[i, j]$. Every occurrence of S not containing a phrase boundary is called a *secondary occurrence*. It can easily be shown that every *distinct substring* of T has a primary occurrence [21], which makes $\{i_1, \ldots, i_z\}$ a *string attractor*. See Figure 1 for an example.

Repetition-Aware Suffix Trees/Arrays and the r-index. By indexing the text in space $O(r \log(n/r))$, we can support SA, ISA, and LCE operations in $O(\log(n/r))$ time [14]. There also exists an optimal $O(\delta \log(n/\delta))$ space index (called δ-SA) that supports SA and ISA operations in $O(\log^{4+\epsilon} n)$ time and LCE operation in time $O(\log n)$ [24]. The r-index is another important structure of space $O(r)$ words [14]; see [41] for its refinements. It supports efficient pattern matching, but it cannot support any of the basic operations mentioned earlier. Fortunately, the limited operations supported by r-index are sufficient for our purpose.

Orthogonal Range Queries. The task here is to preprocess a set of d-dimensional points $\mathcal{P}$. For a query $[a_1, b_1] \times \cdots \times [a_d, b_d]$, the output is the subset of points in $\mathcal{P} \cap [a_1, b_1] \times \cdots \times [a_d, b_d]$. Over a set of N 2-dimensional points in an $N \times N$ grid, we can maintain an $O(N)$ space structure and support queries in $O((1+t) \log^{\varepsilon} N)$ time [8], where $\varepsilon > 0$ is an arbitrarily small constant, (or) an $O(N \log^{\varepsilon} N)$ space and support queries in $O(\log \log N + t)$ time, where t denotes the output size [9].

A standard technique in compressed text indexing, and one that we expand on here, is the use of orthogonal range queries to report primary occurrences [13, 29, 38]. For a simple example, assume that we have the phrases $T = T[i_1 \mathinner{.\,.} i_2) \cdot T[i_2 \mathinner{.\,.} i_3) \cdots T[i_s \mathinner{.\,.} n]$. We build a compact trie $\mathcal{T}^F$ over the suffixes $T[i_1 \mathinner{.\,.}], \ldots, T[i_j \mathinner{.\,.}]$, maintaining for each leaf its respective i_j value. We also build a compact trie $\mathcal{T}^F$ over the reversed prefixes $T[..i_2)^R, \ldots, T[\mathinner{.\,.} i_s)^R$, maintaining for each leaf its respective i_j value. Next for each i_j, we create a 2D point (x, y) where x is the leaf index in $\mathcal{T}^R$ corresponding to $T[\mathinner{.\,.} i_j)^R$ and y is the leaf index in $\mathcal{T}^F$ corresponding to $T[i_j \mathinner{.\,.}]$ and associate the value i_j with (x, y). Construct a 2D range query structure over this set of points.

Given a query pattern P, for each $x \in [0, m)$, we find the locus of $P[..x]^R$ in $\mathcal{T}^R$, let its range be $[a_1, b_1]$. We also find the locus of $P[x+1..]$ in $\mathcal{T}^F$, let its range be $[a_2, b_2]$. Observe that query $[a_1, b_1] \times [a_2, b_2]$ returns points for phrase boundaries aligning with position $x+1$ in a primary occurrence of P. See Figure 2.

Heavy Path Decomposition. For a general tree $\mathcal{T}$ and node u in $\mathcal{T}$, we denote the subtree rooted at u as subtree(u). The size of the subtree(u), denoted as $|\text{subtree}(u)|$, is defined as the number of leaves in subtree(u). We next describe heavy path decomposition [48]. Given any tree $\mathcal{T}$, we categorize its nodes into light and heavy: we take the root as a light node. Exactly one child of every (non-leaf) node is heavy, specifically the one with maximum subtree size. When there is a tie, we pick the leftmost child as heavy. Any path starting from a light node and recursively following its heavy child, and ending at a leaf node is a heavy path. Note that each node belongs to precisely one heavy path.

▶ **Property 1.** *Each root-to-leaf path in a tree of size n contains at most $\log n$ light nodes.*

Property 1 can be shown by observing that for any node u in $\mathcal{T}$ with light child v, $|\text{subtree}(v)| \le |\text{subtree}(u)|/2$. Letting L be the set of light nodes in $\mathcal{T}$, Property 1 further implies that $\sum_{u \in L} |\text{subtree}(u)| \le n \log n$.

3 Our Solution

As the first building block, we maintain a *base structure* that supports suffix array, inverse suffix array, and LCE queries for the text and its reverse. This can be suffix trees in some (compressed) form. We use S_{DS} to denote the space of the base structure and t_Q to denote its query time. We perform the analysis with a general base structure and specify specific trade-offs in Section 5. Our approach then builds two versions of Cole et al.'s *k-wildcard errata structure* [10] over a subset of substrings based on a parse of the text. Given a query pattern $P[1..m]$, we first find the primary occurrences of P. This is done using a 2D-range query structure constructed over a set of points created with the leaves of both k-wildcard errata structures. The secondary occurrences are obtained using the base structure. We start with a basic subroutine used throughout.

3.1 LCP Queries

We consider a compact trie $\mathcal{T}$ built from a subset of q substrings of T. With every leaf of $\mathcal{T}$ we associate a starting text position of the corresponding substring. We seek to answer queries of the following form: Given i, j and a node u (either implicit or explicit) in $\mathcal{T}$, find the locus of $T[i..j']$ in subtree(u), where $T[i..j']$ is the longest prefix of $T[i..j]$ for which such a locus exists. We call this type of query a *unrooted LCP query*. If u is restricted to always be the root $\mathcal{T}$, we call this a *rooted LCP query.*

▶ **Lemma 3.** *Assuming the availability of our base structure of space S_{DS}, which supports basic suffix tree operations (*SA, ISA, *and* LCE *queries) in t_Q time, we have the following results. An unrooted LCP query on a compact trie $\mathcal{T}$ constructed from q substrings of T can be answered in $O(t_Q + \log\log n)$ time. The (additional) space used by the data structure is $O(q \log q)$. Moreover, for rooted LCP queries, the (additional) space can be made $O(q)$ with the same query time complexity.*

Proof. As preprocessing for unrooted LCP queries we do the following. Let u_r be the root of $\mathcal{T}$. For every *light* node u in $\mathcal{T}$, we consider all substrings corresponding to the leaves in its subtree, that is, strings obtained by concatentating edges labels on u_r-to-leaf paths for leaves

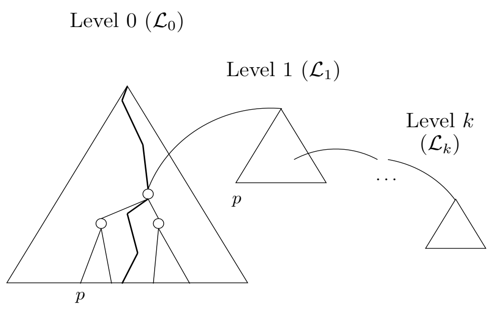
Level 0 ($\mathcal{L}_0$)
Level 1 ($\mathcal{L}_1$)
Level k ($\mathcal{L}_k$)
p
p
...

Figure 3 Visualization of Cole's k-wildcard errata structure.

in subtree(u). Suppose this is the set $\{T[i_1 \mathinner{.\,.} i'_1], T[i_2 \mathinner{.\,.} i'_2] \dots\}$. For each u we create a y-fast trie [52] over the values ISA$[i_1 + \text{strlen}(u)]$, ISA$[i_2 + \text{strlen}(u)]$, ... that supports predecessor search in $O(\log\log n)$ time. Note that by Property 1 of the heavy path decomposition, the number of values over all light nodes, as well as the total space is $O(q \log q)$.

Given a query consisting of positions i, j (specifying an arbitrary substring $T[i \mathinner{.\,.} j]$) and a locus u in $\mathcal{T}$, we first perform an LCE$(i, b + \text{strlen}(u))$ where b is the text position of the leaf on the heavy path containing u. If the entirety of $T[i \mathinner{.\,.} j]$ is matched along this heavy path (LCE$(i, b + \text{strlen}(u)) \geq j - i + 1$), or if the matching ends prematurely at an implicit node of the heavy path, we are done. Otherwise, we obtain a node v on the heavy path from which we have to explore v's light children. We next obtain the child w of v that needs to be traversed; note that w must be a light child of v. We query the base structure to obtain the value $x :=$ ISA$[i + \text{strlen}(w) - \text{strlen}(u)]$. We then perform a predecessor search for x over the values stored for w. The result of the predecessor search gives us at most two leaves corresponding to the predecessor and successor of x in terms of ISA value. We then perform LCE queries with these neighboring leaves and obtain the result.

For rooted LCP queries, we apply the same preprocessing technique but only at the root of $\mathcal{T}$. Queries are answered in the same way as unrooted LCP queries. ◀

3.2 Cole et al.'s k-Wildcard Errata Structure

The Structure. Cole et al.'s structure [10] consists of a collection of compact tries, which are further categorized into levels (from level 0 to k). We use $\mathcal{L}_h$ to denote the set of tries at level h for $h \in [0, k]$. In $\mathcal{L}_0$ we have a single trie. In Cole et al.'s framework, this level 0 trie is taken as the complete suffix tree ST(T). In this work, we will be more general, allowing it to be an arbitrary trie constructed from q substrings of T. In $\mathcal{L}_1$, each trie corresponds to an internal node in the level 0 trie. A level 1 trie for an internal node u of the level 0 trie is constructed as follows: we collect all strings corresponding to leaves in the subtree of u except those in the subtree of the heavy child of u, remove their first strlen$(u) + 1$ characters, and create a compact trie. Suppose that a substring $T[j \mathinner{.\,.} j']$ used in level 1 is obtained from $T[i \mathinner{.\,.} j']$ after removing the first strlen$(u) + 1$ characters, then we say that the substring $T[j \mathinner{.\,.} j']$ in the level 1 trie corresponds to the *original position* i. Continuing, each trie in $\mathcal{L}_h$, $h \in [1, k]$, corresponds to a unique internal node in some trie in level $(h - 1)$. A level h trie for a node u in a level $(h - 1)$ trie is constructed over all substrings for leaves in the subtree of u except those in the heavy child of u, and with their first strlen$(u) + 1$ characters removed. We similarly maintain the correspondence with the *original position* in $\mathcal{L}_0$ for each substring in $\mathcal{L}_h$. We use $\mathcal{L}_{\leq k}$ to denote the collection of all k levels. See Figure 3.

It follows directly from Property 1 of heavy path decomposition that the overall space is $O(q(\log q)^k)$. With a tighter analysis, one can show that the space is $O(q2^k(\log q + k)^k/k!)$. As the details of this analysis are important to our solution, and the statement of Lemma 4 differs from that made in [10], we present the argument in full.

▶ **Lemma 4.** *Let q be the number of leaves in $\mathcal{L}_0$. Also, let i be the original position for a leaf in $\mathcal{L}_0$. Then i is the original position for at most $2^k(k + \log q)^k/k!$ leaves in $\mathcal{L}_{\leq k}$.*

Proof. We fix a particular leaf ℓ in $\mathcal{L}_0$. Suppose the light nodes on the root-to-ℓ path in $\mathcal{L}_0$ are $r = u_0$, u_1, …,u_x and the respective parents v_1, …, v_x (excluding v_0 since u_0 is the root). There are two cases: if ℓ contributes to u_0's level 1 trie (that is, ℓ is in the subtree of a light child of u_0), then the size of v_1 is at most $q/2$, the size of v_2 is at most $q/4$, and so on. Otherwise, the size of v_1 is at most q, the size v_2 is at most $q/2$, and so on. Let the inductive hypothesis be that in a $(k-1)$-level structure for q substrings that the leaf ℓ occurs at most $2^{k-1}(k-1+\log q)^{k-1}/(k-1)!$. Based on the observation regarding subtree sizes, for a k-level structure, we have the number of leaves with original position i is bounded by

$$1 + \frac{2^{k-1}(k-1+\log(q))^{k-1}}{(k-1)!} + \frac{2^{k-1}(k-1+\log(q/2))^{k-1}}{(k-1)!} + \cdots + \frac{2^{k-1}(k-1)^{k-1}}{(k-1)!}$$
$$= 1 + \frac{2^{k-1}}{(k-1)!}\left((k-1)^{k-1} + k^{k-1} + \cdots + (k-1+\log q)^{k-1}\right)$$

$$\leq 1 + \frac{2^{k-1}}{(k-1)!}\int_0^{k+\log q} x^{k-1}dx = 1 + \frac{2^{k-1}(k+\log q)^k}{k!} \leq \frac{2^k(k+\log q)^k}{k!} \quad \text{for } k \geq 1. \quad ◀$$

From Lemma 4, the total space needed for q substrings is $O(q \cdot 2^k(\log q + k)^k/k!)$.

Querying. Querying is carried out by starting at the root of the level 0 trie and matching until the first wildcard character # is reached. If the locus is an implicit node, then we consider the match to continue on the implicit node's edge. If the locus of the last character before the first # is an explicit non-leaf node u, then we consider the match as continuing both to the heavy child of u and the level 1 trie for u. In both cases, we match the # by skipping the next character. In the case of the level 1 trie for u, this is accomplished by skipping the # and starting from the root of the level 1 trie.

The same procedure is recursively applied for the level 1 trie when the next wildcard is reached. Assuming P has $h \leq k$ wildcards, this causes at most 2^h bifurcations to occur. If a mismatch occurs at any point during a branch of this search, we stop exploring that branch further. Note that if $P = P_0 \cdot \#P_1 \cdots \# \cdot P_h$ and the search is performed character-by-character then the total time required is $|P_0| + 2|P_1| + \cdots + 2^{h-1}|P_h| = O(2^h m)$. However, by preprocessing all pattern segments P_i so that they are known substrings of T, we can apply Lemma 3 and find all pattern loci in $O(2^h(t_Q + \log\log n))$ time. Note that by using the unrooted LCP query structure for tries in levels $\mathcal{L}_0$, …, $\mathcal{L}_{k-1}$ rooted LCP query structure for tries in $\mathcal{L}_k$, the space complexity is unaltered.

3.3 Our Data Structure

We maintain a *base structure* (as described in the beginning of Section 3) for T and for T^R. We assume that we are given a string attractor $\Gamma = \{i_1, \ldots, i_s\}$ of size s such that $1 = i_1 < i_2 < \cdots < i_s$. This gives a factorization $T = T[i_1 \mathinner{..} i_2) \cdots T[i_s \mathinner{..}]$ with s phrases.

Forward Sparse Structure. The construction of our structure is similar to that of Cole et al.'s, but instead of having the full suffix tree as the level 0, we keep a trie of only those suffixes corresponding to the phrase boundaries. We assign a global index to the leaves across all tries. The only condition we ensure is that all leaves in the same trie should get contiguous global indices in the left-to-right order. We call the resulting structure $\mathcal{L}^F_{\leq k}$. Following the same analysis as presented in Section 3.2, the space required for $\mathcal{L}^F_{\leq k}$ is $O(s2^k(\log s + k)^k/k!)$. We next equip all tries in $\mathcal{L}^F_0, \ldots, \mathcal{L}^F_{k-1}$ with the unrooted LCP query structure described in Lemma 3 and the tries in $\mathcal{L}^F_k$ with the rooted LCP query structure. This does not change the space complexity.

Reverse Sparse Structure. Next, we collect all phrases. We create a compact trie from the collection of *reversed phrases*[3]. For each leaf in the level 0 trie, we maintain the phrase boundary in T that immediately proceeds it; that is, if the reversed phrase is $T[i_j \mathinner{.\,.} i_{j+1})^R$, we associate the original position i_{j+1} with the corresponding leaf. With this trie as level 0, we then create Cole et al.'s k-level structure. We also give a global index labeling to the leaves as before. We call the resulting structure $\mathcal{L}^R_{\leq k}$. As before, we equip the tries in $\mathcal{L}^R_{\leq k-1}$ with the unrooted LCP query data structures from Lemma 3 and the rooted LCP structure on $\mathcal{L}^R_k$. The total space for this structure is $O(s2^k(\log s + k)^k/k!)$.

Orthogonal Range Query Structure. Our next component is a 2D orthogonal range query structure. The 2D points (x, y) are created as follows: let y be the global index of a leaf in a level α trie of $\mathcal{L}^F_{\leq k}$ with original position i, and x is the global index of a leaf in a level β trie in $\mathcal{L}^R_{\leq k}$ with original position i. We create a point for all such x and y where the inequality $\alpha + \beta \leq k$ holds. We associate the original position i with that point.

3.4 Space Analysis

Applying Lemma 4, the number of leaves with original position i at level α of $\mathcal{L}^F_{\leq k}$ is bounded by $2^\alpha(\log s+\alpha)^\alpha/\alpha!$. Similarly, the number of leaves assigned original position i at a level β in $\mathcal{L}^R_{\leq k}$ is also bounded by $2^\beta(\log s+\beta)^\beta/\beta!$. As a result, the number of 2D points corresponding to a specific original position i is bound by

$$\begin{aligned}
\sum_{\alpha+\beta\leq k} \frac{2^\alpha(\log s+\alpha)^\alpha}{\alpha!} \cdot \frac{2^\beta(\log s+\beta)^\beta}{\beta!} &\leq \sum_{j=0}^{k}\sum_{\alpha=0}^{j} \frac{2^\alpha(\log s+k)^\alpha}{\alpha!} \cdot \frac{2^{j-\alpha}(\log s+k)^{j-\alpha}}{(j-\alpha)!} \\
&= \sum_{j=0}^{k} 2^j(\log s+k)^j \sum_{\alpha=0}^{j} \frac{1}{\alpha!(j-\alpha)!} \\
&= \sum_{j=0}^{k} 2^j(\log s+k)^j \frac{1}{j!} \cdot \sum_{\alpha=0}^{j} \binom{j}{\alpha} \\
&= \sum_{j=0}^{k} \frac{4^j(\log s+k)^j}{j!}
\end{aligned}$$

where the last equality applies that $\sum_{\alpha=0}^{j}\binom{j}{\alpha} = 2^j$. Next, we consider the inequality

$$\frac{4^{j+1}(\log s+k)^{j+1}}{(j+1)!} \geq \frac{4^j(\log s+k)^j}{j!}.$$

[3] In contrast to the forward sparse structure, we do not use the entire prefix ending at a phrase boundary.

This holds as long as $j+1 \leq 4(\log s + k)$, which is true, since $j+1 \leq k+1 \leq 4(\log s + k)$. Thus, we can write

$$\sum_{j=0}^{k} \frac{4^j(\log s + k)^j}{j!} \leq 1 + k \cdot \frac{4^k(\log s + k)^k}{k!} = O\left(\frac{(4e^{1/e})^k(\log s + k)^k}{k!}\right)$$

where we used in the last inequality that $k \leq e^{k/e}$. Letting $C = 4e^{1/e}$, which is constant, the total number of 2D points is $N = O(s \cdot C^k(\log s + k)^k/k!)$. It is now convenient to split our analysis into two cases, the case where $\log s$ dominates k and vice versa:

- If $\log s$ dominates k, then the space complexity is

$$\frac{C^k(\log s + k)^k}{k!} \leq \frac{(2C)^k(\log s)^k}{k!}$$

- If k dominates $\log s$, then the space complexity is

$$\frac{C^k(\log s + k)^k}{k!} \leq \frac{(2C)^k k^k}{k!} \sim \frac{(2C)^k k^k}{\sqrt{2\pi k}\left(\frac{k}{e}\right)^k} \leq (2Ce)^k$$

Hence, we can say

$$\frac{C^k(\log s + k)^k}{k!} \leq \frac{(2C)^k(\log s)^k}{k!} + (2Ce)^k \leq (2Ce)^k\left(\frac{(\log s)^k}{k!} + 1\right).$$

We will first consider using a linear-space 2D range query data structure over these N points. Note that the space required by the forward sparse structure and the reverse sparse structure is also bound by the same expression. The resulting overall space complexity is therefore $S_{\mathsf{DS}} + O\left(s \cdot c^k\left(\frac{(\log s)^k}{k!} + 1\right)\right)$ for some constant c.

3.5 Querying Algorithm

We first present a simpler querying procedure and then show how it can be improved upon. Assume the query is $P = P_0 \cdot \# \cdot P_1 \cdot \# \cdots \# \cdot P_h$ with $h \leq k$. We begin by searching for each segment P_i (resp., P_i^R) in the *base structure.* If some segment does not exist in T, then there are no occurrences of P. Otherwise, this enables us to apply Lemma 3.

Finding Primary Occurrences. Next, we obtain all primary occurrences of P. Let occ_p denote the number of primary occurrences of P. To accomplish finding all primary occurrences, we consider each split of P, that is, $P[..x]$ and $P[x+1..]$ for $x \in [0, m)$. For a given split $P[..x]$ and $P[x+1..]$, we match $P[..x]^R$ in the $\mathcal{L}^R_{\leq k}$ and $P[x+1..]$ in $\mathcal{L}^F_{\leq k}$. In more detail, suppose we have β wildcards in the reverse part and α wildcards in the forward part. Then, based on the analysis of the querying procedure in Section 3.2, there will be at most 2^β bifurcation loci in the search in $\mathcal{L}^R_{\leq k}$ and 2^α loci in $\mathcal{L}^F_{\leq k}$. Applying Lemma 3, we spend $O(t_Q + \log\log n)$ time per bifurcation. For a given split of P, we now have the loci in all tries of $\mathcal{L}^R_{\leq k}$ where the matching of $P[..x]^R$ ends. Denote this set of loci as L_R. We also have the loci in all tries of $\mathcal{L}^F_{\leq k}$ where the matching of $P[x+1..]$ ends. Denote this set of loci as L_F. We next consider each pair of locus, $l_1 \in L_R$ and $l_2 \in L_F$. For l_1 (resp., l_2) we obtain a contiguous range of global indices $[a_1, b_1]$ (resp., $[a_2, b_2]$) corresponding to the leaves in the subtree of l_1 (resp., l_2). For all such pairs of ranges, we make the 2D range query $[a_1, b_1] \times [a_2, b_2]$ and from each reported point's value and offset x, we obtain a primary occurrence.

Finding Secondary Occurrences. From the set of primary occurrences, we first extract *one occurrence per each distinct substring* of T that matches with P. To do this, we find the ISA values of all primary occurrences first, sort them in ascending order of their ISA values, then scan this list in the left to right order and retain only those entries with LCE value less than m with its previous element. Then, for each occurrence i in the reduced set, we can find all other occurrences of $T[i\,..\,i+m)$ in T as follows: First: initialize $j = \text{ISA}[i]+1$, while $\text{LCE}(i, \text{SA}[j]) \geq m$, report $\text{SA}[j]$ and increment j. Second: initialize $j = \text{ISA}[i]-1$, while $\text{LCE}(i, \text{SA}[j]) \geq m$, report $\text{SA}[j]$ and decrement j. This part of the query algorithm takes $O(\mathsf{occ}_p \cdot \log \mathsf{occ}_p + \mathsf{occ} \cdot t_Q)$ time.

Time Complexity. Regarding the query time complexity, as there are $2^\alpha + 2^\beta \leq 2^{h+1}$ bifurcations loci per pattern split and m pattern splits, the total time for finding all pattern loci (using Lemma 3) is bound by $O(m2^h(t_Q + \log\log n))$. For orthogonal range queries, the number of pairs of ranges we have to consider for a given pattern split is bound by $2^\alpha \cdot 2^\beta = 2^h$. Each query takes $O((1+\mathsf{occ}_j^x)\cdot \log^\varepsilon N)$ time, where N is the number of 2D points and occ_j^x refers to the number of occurrences reported for the j^{th} range query (the j^{th} locus pair) with pattern split x. Since all loci used for the orthogonal range query are for distinct substrings in T, all reported occurrences for two different orthogonal range queries are distinct, and $\sum_j \mathsf{occ}_j^x = \mathsf{occ}^x$, where occ^x is the total number of occurrences reported for a split x. Summing over all orthogonal range queries for a given split, we have the time used per split x bounded by

$$\sum_j (1+\mathsf{occ}_j^x)\cdot \log^\varepsilon N \leq (2^h + \mathsf{occ}^x)\cdot \log^\varepsilon N$$

We claim $\sum_{x=1}^m \mathsf{occ}^x = \mathsf{occ}_p$. It is easy to see that every primary occurrence will be reported. Moreover, each primary occurrence will be reported only once, because $\mathcal{L}^R_{\leq k}$ is built over the set of phrases (rather than the entire prefix ending prior to the start of a phrase). Due to this, even if a primary occurrence contains multiple phrase boundaries, it is still only reported once. In particular, it gets reported for the split aligned with its leftmost phrase boundary.

To obtain the time used by orthogonal range queries for finding all primary occurrences, we sum over all x, resulting in the expression

$$\sum_{x=1}^m \left(2^h + \mathsf{occ}^x\right)\cdot \log^\varepsilon N = (2^h m + \mathsf{occ}_p)\cdot \log^\varepsilon N.$$

Note that N can be loosely bounded by $2^{O(k+\log s)}$. This is because the maximum value of the function $f(k) = (\log s)^k/k!$ is $f(\log s) = (\log s)^{\log s}/(\log s)! \sim \frac{1}{\sqrt{2\pi\cdot \log s}} e^{\log s} = 2^{O(\log s)}$. Therefore, $\log N = O(k + \log s) \subseteq O(\log n)$.

The total time is $O(2^h m(t_Q + \log n) + \mathsf{occ}_p \cdot (\log \mathsf{occ}_p + \log^\varepsilon n) + \mathsf{occ}\cdot t_Q)$, in addition to the time for finding one occurrence of all segments of P.

We summarize the results of Section 3 in Lemma 5.

▶ **Lemma 5.** *For problem 1, there exists an* $S_{\mathsf{DS}} + O\left(s \cdot c^k \left(\frac{(\log s)^k}{k!} + 1\right)\right)$ *space data structure, where s is the size of text's string attractor and $c > 1$ is a fixed constant. The query time is* $O(2^h m(t_Q + \log n) + \mathsf{occ}_p \cdot (\log \mathsf{occ}_p + \log^\varepsilon n) + \mathsf{occ} \cdot t_Q)$*, in addition to the time for initial pattern search.*

4 An Improved Solution

4.1 Our Data Structure

Our next aim is to replace the product $2^h m$ in the time complexity with $2^h + m$. The idea is to handle long patterns ($m > \tau$) and short patterns ($m \leq \tau$) separately, where τ is a parameter we specify later.

Long Patterns. For long patterns, we include some additional suffixes and phrases (extended slightly more to the right and left) within our construction. Specifically, for each phrase $T[i_j \mathinner{.\,.} i_{j+1})$ for $j \in [1, s]$, we include all of the suffixes $T[i_j + t \mathinner{.\,.}]$ for $t \in (-\tau, \tau)$ in $\mathcal{L}_0^F$ and $T[i_{j-1} \mathinner{.\,.} i_j + t)^R$ in $\mathcal{L}_0^R$. We then build $\mathcal{L}_{\leq k}^F$ and $\mathcal{L}_{\leq k}^R$ based of their respective level 0 trie. We also create a set of 2D points, again with (x, y) values where x is global index of a leaf in $\mathcal{L}_{\leq k}^R$, and y a global index of a leaf in $\mathcal{L}_{\leq k}^F$, and x and y both have original position $i_j + t$ for some $j \in [1 \mathinner{.\,.} s]$ and $t \in (-\tau, \tau)$. The linear space 2D orthogonal range query structure used in Section 3.3 is constructed over these points.

Short Patterns. To handle short patterns, we collect the substrings $T[i_j - \tau \mathinner{.\,.} i_j + \tau]$ for each $j \in [1, s]$ (also record one of its occurrence in T). After appending each such substring with a \$ (a delimiter), we concatenate them and obtain a new string T'. We build the k wildcard errata structure of Cole et al. [10] for T'.

4.2 Space Analysis

For the long pattern data structure, the number of substrings used in level 0 of both $\mathcal{L}_{\leq k}^R$ and $\mathcal{L}_{\leq k}^F$ now becomes $O(s\tau)$. Following Lemma 4, the space for both $\mathcal{L}_{\leq k}^F$ and $\mathcal{L}_{\leq k}^R$ becomes at most $2s\tau \cdot 2^k(\log(2s\tau) + k)^k/k!$. Following the same analysis presented in Section 3.4, the number of 2D points created and the final asymptotic space complexity is $O(s\tau \cdot c^k(1 + \log^k(s\tau)/k!))$ for a constant c. For short patterns, the length of T' is $O(s\tau)$. The k-wildcard errata structure over T' requires $O(s\tau \cdot c^k(1 + \log^k(s\tau)/k!))$.

4.3 Querying Algorithm

As before, we first find occurrences of all pattern segments using the base structure. This enables us to apply Lemma 3. We first consider the long pattern case where $m > \tau$. Now, for finding primary occurrences, rather than trying every split $x \in [0, m)$ as was done in Section 3.5, we split only at positions $x \in \{1, 1 + \tau, 1 + 2\tau, 1 + 3\tau \dots\}$. For a split of P at x, we find the loci of $P[\mathinner{.\,.} x]^R$ in $\mathcal{L}_{\leq k}^R$ and $P[x + 1 \mathinner{.\,.}]$ in $\mathcal{L}_{\leq k}^F$. For each pair of loci, we obtain two ranges of global leaf labels and perform a range query as before. From the original index associated with a reported point we can obtain the corresponding primary occurrence. Secondary occurrences are then reported using the base structure.

For the short pattern case where $m \leq \tau$, we first obtain the primary occurrences by querying the k-wildcard errata trie over T'. Secondary occurrences are obtained using the base structure.

4.4 Time Complexity

In the case of long patterns, we consider $O(m/\tau)$ different splits of the pattern. For a split, locating all of the loci in $\mathcal{L}_{\leq k}^R$ and $\mathcal{L}_{\leq k}^F$ requires $O(2^h(t_Q + \log\log n))$ time. Two key observations of our sampling method are that (i) every primary occurrence has a split that

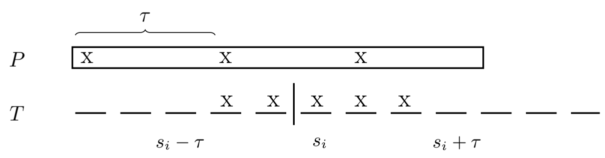

Figure 4 Sampling method for long patterns with $\tau = 3$ and phrase boundary s_i. The x's in P indicate the splits considered, and the x's in T indicate the sampled suffix positions used.

aligns with a sampled position (ii) each primary occurrence will be reported at most twice. See Figure 4. This follows from the splits considered being τ indexes apart in P, making it so at most two will be within the same τ sized window around a phrase boundary. For a split at x and the j^{th} pair of loci, one in $\mathcal{L}^R_{\leq k}$ and $\mathcal{L}^F_{\leq k}$, let occ^x_j be the number of 2D points reported for a range query. Combining the above to facts, for a particular split x, the time complexity for range queries is $\sum_j (1 + \mathsf{occ}^x_j) \cdot \log n = (2^h + \mathsf{occ}^x) \cdot \log n$. Summing over all $O(m/\tau)$ splits, the time complexity for all range queries $O((2^h m/\tau + \mathsf{occ}_p) \cdot \log n)$. Finally, using the base structure to obtain the secondary occurrences once again requires $O(\mathsf{occ}_p \cdot \log n + \mathsf{occ} \cdot t_Q)$ time. We conclude that the total time taken for long patterns is the time for locating all segments of P in addition to

$$O(\frac{m}{\tau} 2^h \cdot (t_Q + \log\log n + \log^\varepsilon n) + \mathsf{occ}_p \cdot (\log \mathsf{occ}_p + \log^\varepsilon n) + \mathsf{occ} \cdot t_Q).$$

For short patterns, each locus is found in $O(\log\log n)$ time. The resulting time complexity for finding all occurrences is the time for locating all segments of P in addition to $O(2^h \log\log n + \mathsf{occ}_p \cdot \log \mathsf{occ}_p + \mathsf{occ} \cdot t_Q)$.

▶ **Lemma 6.** *For problem 1, there exists an $S_{\mathsf{DS}} + O\left(s\tau c^k \left(\frac{(\log s\tau)^k}{k!} + 1\right)\right)$ space data structure, where s is the size of text's string attractor, $c > 1$ is a fixed constant and $\tau \geq 1$ is a parameter. The query time, in addition to the time for initial pattern search is $O((1 + \frac{m}{\tau})2^h \cdot (t_Q + \log^\varepsilon n) + \mathsf{occ}_p \cdot (\log \mathsf{occ}_p + \log^\varepsilon n) + \mathsf{occ} \cdot t_Q)$.*

5 Trade-Offs for Time and Space

We now look at some possible trade-offs based on the base structure and the choice of τ.

5.1 Fully Functional Suffix Trees in Repetition-Aware Space

We first consider using fully functional compressed suffix tree in [14] for the text and its reverse. The space is $O(r \log \frac{n}{r} + \overline{r} \log \frac{n}{\overline{r}})$, where r and $\overline{r}$ are the number of runs in $\mathrm{BWT}(T)$ and $\mathrm{BWT}(T^R)$ respectively. Making $\tau = 2^k$, the additional space complexity becomes

$$s\tau \cdot c^k \left(\frac{\log^k(s\tau)}{k!} + 1\right) \leq s \cdot (2c)^k \left(\frac{(\log s + k)^k}{k!} + 1\right) \leq s \cdot C^k \left(\frac{\log^k s}{k!} + 1\right).$$

Here $C > c$ is a large enough constant. For query times, we have $t_Q = O(\log n)$. The suffix ranges of all pattern segments can be found in $O(m)$ time, and one occurrence of each segment is reported in $O(\log n)$ time. This gives an $O(r \log \frac{n}{r} + \overline{r} \log \frac{n}{\overline{r}} + s(c^k(\frac{(\log s)^k}{k!} + 1)))$ space index with query time $O((m + 2^h + \mathsf{occ}) \log n)$. We improve this result next.

5.2 The r-Index and Optimal LCE Structure (Completing Theorem 1)

Here we replace the fully functional suffix trees in the base structure with two r-indexes [14] (one for T and one for T^R), which brings down the space to $O(r + \bar{r})$. Although, the r-index supports only limited functionalities, we will see that they suffice for our purposes. We additionally maintain δ-SA [24] for T and T^R in $O(\delta \log(n/\delta))$ space, for supporting LCE queries in time time $O(\log n)$. In what follows, we examine the steps where the base structure is queried, and observe how our new base structure suffices.

- For each segment of the pattern P_i, we need to find an occurrence (text position of P_i and its inverse suffix array position). This can be done with the $O(r)$ space r-index. Moreover, the *backward search* procedure utilized by the r-index provides us with a corresponding values (i.e., position in the text and its ISA) for every suffix of the pattern segment $P_i[j\,..]$; we refer to the *toehold lemma* in [14], also see [42]. The same can be accomplished for the reverse of the pattern segment with the r-index for the reversed text. The total time for this step is $O(m \log\log n)$.
- For rooted and unrooted LCP queries, we used ISA value of the current suffix (or prefix) of a pattern segment as well as LCE queries. As observed above, the needed ISA values of suffixes and prefixes of pattern segments can be obtained while performing the backward search procedure. For LCE queries, we use the structure of Kempa et al. [24].
- The next step utilizing the base structure is finding the secondary occurrences from the primary occurrences. The first step in our previous algorithm was to find a subset of primary occurrences, such that for each each distinct substring of T matching P, we have exactly one occurrence in that set. Previously, this procedure required ISA queries on arbitrary positions, which cannot be supported by the r-index. Therefore, we follow an alternative procedure, which rely on the fact that r-index can support operations $\phi(p) = \mathrm{SA}[\mathrm{ISA}[p] - 1]$, $\phi^{-1}(p) = \mathrm{SA}[\mathrm{ISA}[p] + 1]$, $\mathrm{LCE}(p, \phi^{-1}(p))$, and $\mathrm{LCE}(p, \phi(p))$ for any given p in $O(\log\log n)$ time [14]. Therefore, by applying ϕ iteratively, we can compute $\mathrm{SA}[\mathrm{ISA}[p]-1], \mathrm{SA}[\mathrm{ISA}[p]-2], \ldots, \mathrm{SA}[\mathrm{ISA}[p]-t]$ for any t in $O(t \log\log n)$ time. Similarly, by iteratively applying ϕ^{-1}, we can find $\mathrm{SA}[\mathrm{ISA}[p]+1], \mathrm{SA}[\mathrm{ISA}[p]+2], \ldots, \mathrm{SA}[\mathrm{ISA}[p]+t]$ for any t in $O(t \log\log n)$ time.

  We now describe the procedure. Let OCC_p be the non-reduced set of a primary occurrences. Also, let OCC be a set, which is initially empty; we maintain its elements in the form of a balanced binary search tree [2], supporting membership queries in logarithmic time. We process each $p \in \mathsf{OCC}_p$ in any order as follows. If $p \in \mathsf{OCC}$ then skip p, otherwise,

  1. Initialize $p' = \phi(p)$.
     While $\mathrm{LCE}(\phi^{-1}(p'), p') \geq m$, add p' to OCC and make $p' \leftarrow \phi(p')$.
  2. Initialize $p' = \phi^{-1}(p)$.
     While $\mathrm{LCE}(\phi(p'), p') \geq m$, add p' to OCC and make $p' \leftarrow \phi^{-1}(p')$.

  Lastly, we output the set OCC.

The total query time is $O((m + 2^h) \cdot \log n + \mathsf{occ}_p \cdot \log^{\varepsilon} n + \mathsf{occ} \cdot (\log\log n + \log \mathsf{occ}))$. We summarize the results of this approach in Theorem 7.

▶ **Theorem 7.** *There exists a solution for Problem 1 with space*

$$O\left(r + \bar{r} + \delta \log(n/\delta) + s \cdot c^k \left(1 + \frac{\log^k s}{k!}\right)\right) \text{ words,}$$

and query time $O((m + 2^h) \cdot \log n + \mathsf{occ} \cdot (\log^{\varepsilon} n + \log \mathsf{occ})) \subseteq O((m + 2^h + \mathsf{occ}) \cdot \log n)$, where δ is the substring complexity, s is the size of a known string attractor, r (resp., $\bar{r}$) denotes the number of runs in the BWT of T (resp., T^R), and $c > 1$ is a fixed constant.

By replacing s with $z = O(\delta \log(n/\delta))$ [43] and $r + \bar{r}$ with $O(\delta \log \delta \log(n/\delta))$ [23] in the space complexity in Theorem 7, we obtain the result in Theorem 1 for $k \geq 1$. Recall that for $k = 0$, an $O(\delta \log(n/\delta))$ space index with query time $O(m + (1 + \mathsf{occ}) \log^{\epsilon} n)$ is already known [26]. By combining both cases, we obtain the result in Theorem 1.

5.3 Matching Cole et al.'s Query Time (Completing Theorem 2)

If we are willing to utilize more space, an improved query time is possible. To that end, for the base structure, we use text indexes that support all queries in $O(1)$ time, just like (uncompressed) suffix trees, but in space just $O(n \log \sigma \cdot \log_{\sigma}^{\epsilon} n)$ bits, or, equivalently, $O(n/(\log_{\sigma} n)^{1-\epsilon})$ words [17, 47]. We also maintain the version of the FM-index in [4] of space $O(n \log \sigma)$ bits, which can be utilized to find the suffix ranges of all segments of P in $O(m)$ time. For 2D range queries, we use the *super-linear* space structure of Chan et al. [9]. Over a set of N points, it occupies $O(N \log^{\varepsilon} N)$ space and answers queries in time $O(\log \log N + t)$ where t is the output size. We make $\tau = 2^k \log \log n$. Note that since $k = O(\log n)$,

$$\log \log N = \log \log \left(s\tau \frac{(\log s\tau + k)^k}{k!} \right) = O(\log \log n).$$

The space complexity of our structure becomes $O\left(\frac{n}{(\log_{\sigma} n)^{1-\epsilon}} + s\tau \cdot c^k \left(\frac{(\log \tau s)^{k+\varepsilon}}{k!} + 1\right)\right)$ for some constant c. We again substitute in that $s \leq z = O(\delta \log(n/\delta))$ and simplify to get the space bound shown in Theorem 2.

We now analyze the query complexity. The first step of finding an occurrence of each segment of the pattern can be done in $O(m)$ total time. Regarding long pattern queries, for a particular split x, the time is bound by $\sum_j (\log \log n + \mathsf{occ}_j^x) = 2^h \log \log n + \mathsf{occ}^x$ where occ_j^x and occ_x are defined as in Section 4.4. Summing over all $O(m/\tau)$ splits, we can bound the time complexity of computing the primary occurrences by $O((2^h m \log \log n)/\tau + \mathsf{occ}_p) \subseteq O(m + \mathsf{occ}_p)$. To further obtain the secondary occurrences, we follow the same procedure as in Section 5.2, however we use a bit string $B[1\,..\,n]$ to support constant time membership queries on the set OCC (i.e., $B[i] = 1$ iff $i \in \mathsf{OCC}$). Thus, the overall query complexity for long patterns is $O(m + \mathsf{occ})$. For short pattern queries, the time complexity is $O(m + 2^h \log \log n + \mathsf{occ})$. This completes the proof of the first part of Theorem 2.

5.3.1 Achieving $O(m + \mathsf{occ})$ Query Time

Our goal in this subsection is to achieve $O(m + \mathsf{occ})$ query time. When $m \geq \tau = 2^k \log \log n$ or $\mathsf{occ} \geq \tau$, this has already been accomplished using Theorem 2. Therefore, we consider designing an auxiliary structure for queries with $m, \mathsf{occ} < \tau$. We first take all substrings of length τ containing a phrase boundary. There are at most $O(s\tau)$ such substrings. For each *distinct* substring constructed above, we consider all possible ways of substituting up to k number of $\star$ characters ($\star$ is a special symbol) into the substring. For a given string, there at most $\sum_{h=1}^{k} \binom{\tau}{h} \leq k\binom{\tau}{k}$ ways to do this, where we used that $k \leq \tau/2$. Thus, the total number of strings generated in this way is bounded by

$$s\tau k \binom{\tau}{k} \leq s\frac{k\tau^{k+1}}{k!} = s\frac{k 2^{k(k+1)} (\log \log n)^{k+1}}{k!} = O\left(s\frac{c^{k^2} (\log \log n)^{k+1}}{k!} \right)$$

for some constant c. We remove duplicate strings and build a compact trie $\mathcal{T}$ over the resulting set. We associate with each node u in $\mathcal{T}$ the first τ positions in T matching $\mathrm{str}(u)$ (treating $\star$ as a wildcard). The total additional space becomes $O\left(s\frac{c^{k^2} (\log \log n)^{k+2}}{k!}\right)$ for some constant c. Finally, we fix $s = z$, where $z = O(\delta \log(n/\delta))$ as earlier.

Given a query pattern P, we replace every $\#$ with $\star$ and find its locus in $\mathcal{T}$. Then, we report all associated occurrences if the number of associated occurrences is less than τ. This takes $O(m + \mathsf{occ})$ time. If the number of associated occurrences is greater or equal to τ, we use the previous data structure.

This completes the second part of Theorem 2.

5.4 A Simple, Space Optimal Solution

The suffix tree of T can be used to answer the query in $O(m\sigma^h + \mathsf{occ})$ time for any h, by exhaustively exploring all possible substitutions of each wildcard with actual characters. This can be improved to $O(m + \sigma^h \log n + \mathsf{occ})$ by leveraging the fact that an unrooted LCE query on the suffix tree can be resolved via binary search, requiring $O(\log n)$ basic operations (i.e., SA, ISA, and LCE queries). The query procedure can be simulated on the $O(r \log(n/r))$ space repetition-aware suffix tree [14] in time $O(m + (\sigma^h \log n + \mathsf{occ}) \log(n/r))$. Alternatively, we can use the δ-suffix array [24] of T and the structure in [26] for initial pattern search. The combined space is $O(\delta \log(n/\delta))$ words and the query time is $O(m + (\sigma^h \log n + \mathsf{occ}) \log^{4+\epsilon} n)$.

6 Indexing for Pattern Matching with Gaps

We now consider a generalization of Problem 1, where the query consists of a pattern P with h gaps and we want to find its occurrences in the text. A maximal contiguous sequence of wildcards is defined as a single gap, and its size is the number of wildcards. The maximum number of gaps allowed and the maximum size of a gap are fixed at construction. A formal definition is below.

▶ **Problem 2** (k-Gapped Indexing)**.** *Given a text $T[1\,..\,n]$ and integers k and G, design a data structure that supports the following query efficiently.*

- *Input: Strings $P_0, P_1, \ldots P_h$ of total length m and integers $g_1, g_2, \ldots g_h$, where $h \leq k$ and $g_i \in [1, G]$ for all $i \in [1, h]$.*
- *Output: The set of occurrences of the gapped-pattern $P_0 \cdot \#^{g_1} \cdot P_1 \cdot \#^{g_2} \cdots \#^{g_h} \cdot P_h$ in T.*

A solution by Lewenstein [30] offers $O(nG^{2k} \log^k n)$ space and $O(m + 2^h \log\log n + \mathsf{occ})$ query time. To obtain a repetition-aware solution, one could replace each gap g_i with g_i number of wildcard characters and apply our solution from Section 3. However, this would require a structure with a space complexity exponential in kG. On a related note, see [5, 15, 19, 20] for some interesting solutions for the special case where $k = 1$.

Our Data Structure. We start with the sparse suffix trees for $\mathcal{L}_0^F$ and $\mathcal{L}_0^R$ obtained in the same way as in Section 3.3. To construct $\mathcal{L}_1^F$, we consider a particular explicit node u in $\mathcal{L}_0^F$. For each $g \in \{1, \ldots, G\}$, consider every light node v that (i) branches off the heavy path containing u and (ii) has depth satisfying $\text{strlen}(u) \leq \text{strlen}(v) \leq \text{strlen}(u) + g$. For every substring represented by a leaf in the subtree of such a node v, remove its first $\text{strlen}(u) + g$ characters. These are combined into a single level 1 trie for u and g. The same procedure is performed recursively until all k levels are constructed in $\mathcal{L}_{\leq k}^F$. The same procedure is also used to construct $\mathcal{L}_{\leq k}^R$ from $\mathcal{L}_0^R$. Again, a linear space 2D range query structure is constructed based on original positions, as was done in Section 3.3.

Querying Algorithm and Time Complexity. To answer a query $P = P_0 \cdot \#^{g_1} \cdot P_1 \#^{g_2} \cdots \#^{g_h} \cdot P_h$, we try each split within each segment one by one, i.e., splits of the form $(P_0 \cdot \#^{g_1} \cdot P_1 \cdot \#^{g_2} \cdots \#^{g_j} \cdot P_j[..x))^R$ and $P_j[x..] \cdot \#^{g_{j+1}} \cdots \#^{g_h} \cdot P_h$. When a gap g_i is reached in either a trie in $\mathcal{L}^R_{\leq k}$ or $\mathcal{L}^F_{\leq k}$, if the locus is on an edge (u, v), we subtract the remaining edge label length from the gap size, say l. We then continue from node v, taking both the heavy path within the same trie and the associated trie for v for the gap $g_i - l$. The query time complexity remains the same as in k-wildcard problem, that is $O((m + 2^h + \mathsf{occ}) \log n)$.

Space Analysis. To analyze the space required by this structure, consider that in our solution to Problem 1 , a 2^k arose in the analysis due to a potential bifurcation at every node. Instead, we now take at most G associated side tries for each node, causing us to replace that 2^k with a G^k. Following a similar analysis as Lemma 4, we arrive at a structure of size $O(s(c \cdot G)^k((\log s)^k/k! + 1))$ for some constant c, in addition to the size of the base structure. The key advantage of this space complexity is that it has ceased to be exponential in G. Using the same base structure from Section 5.2 and $s = z$, we achieve Theorem 8.

▶ **Theorem 8.** *There exists a solution for Problem 2 with space*

$$O\left(\delta \log(n/\delta) \cdot (c \cdot G)^k \left(1 + \frac{\log^k(\delta \log n)}{k!}\right)\right) \text{ words and query time } O((m+2^h+\mathsf{occ}) \log n).$$

Here δ is the substring complexity of the text and $c > 1$ is a fixed constant.

References

1 Paniz Abedin, Oliver A. Chubet, Daniel Gibney, and Sharma V. Thankachan. Contextual pattern matching in less space. In *Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023*, pages 160–167. IEEE, 2023. doi:10.1109/DCC55655.2023.00024.

2 Georgii M Adel'son-Vel'skii. An algorithm for the organization of information. *Soviet Math.*, 3:1259–1263, 1962.

3 Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *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 26–39. Springer, 2015. doi:10.1007/978-3-319-19929-0_3.

4 Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Trans. Algorithms*, 10(4):23:1–23:19, 2014. doi:10.1145/2635816.

5 Philip Bille, Inge Li Gørtz, Moshe Lewenstein, Solon P. Pissis, Eva Rotenberg, and Teresa Anna Steiner. Gapped string indexing in subquadratic space and sublinear query time. In *41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024*, volume 289 of *LIPIcs*, pages 16:1–16:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.STACS.2024.16.

6 Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory Comput. Syst.*, 55(1):41–60, 2014. doi:10.1007/S00224-013-9498-4.

7 Michael Burrows and D J Wheeler. A block-sorting lossless data compression algorithm. In , 1994. URL: https://api.semanticscholar.org/CorpusID:2167441.

8 Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.

9 Timothy M. Chan and Konstantinos Tsakalidis. Dynamic orthogonal range searching on the RAM, revisited. *J. Comput. Geom.*, 9(2):45–66, 2018. doi:10.20382/JOCG.V9I2A5.

10 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.

11 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.

12 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.

13 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *LATIN 2014: Theoretical Informatics - 11th Latin American Symposium, Montevideo, Uruguay, March 31 - April 4, 2014. Proceedings*, volume 8392 of *Lecture Notes in Computer Science*, pages 731–742. Springer, 2014. doi:10.1007/978-3-642-54423-1_63.

14 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.

15 Arnab Ganguly, Daniel Gibney, Paul Macnichol, and Sharma V. Thankachan. Bounded-ratio gapped string indexing. In *String Processing and Information Retrieval - 31st International Symposium, SPIRE 2024, Puerto Vallarta, Mexico, September 23-25, 2024, Proceedings*, volume 14899 of *Lecture Notes in Computer Science*, pages 118–126. Springer, 2024. doi:10.1007/978-3-031-72200-4_9.

16 Daniel Gibney, Paul Macnichol, and Sharma V. Thankachan. Non-overlapping indexing in BWT-runs bounded space. In *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 260–270. Springer, 2023. doi:10.1007/978-3-031-43980-3_21.

17 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.

18 Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984. doi:10.1137/0213024.

19 Md Helal Hossen, Daniel Gibney, and Sharma V Thankachan. Text indexing for faster gapped pattern matching. *Algorithms*, 17(12), 2024.

20 Costas S. Iliopoulos and M. Sohel Rahman. Indexing factors with gaps. *Algorithmica*, 55(1):60–70, 2009. doi:10.1007/S00453-007-9141-3.

21 Juha Kärkkäinen and Esko Ukkonen. Lempel-ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.

22 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *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.

23 Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. *Commun. ACM*, 65(6):91–98, 2022. doi:10.1145/3531445.

24 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.

25 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.

26 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`.

27 Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive measure of repetitiveness. In *LATIN 2020: Theoretical Informatics - 14th Latin American Symposium, São Paulo, Brazil, January 5-8, 2021, Proceedings*, volume 12118 of *Lecture Notes in Computer Science*, pages 207–219. Springer, 2020. `doi:10.1007/978-3-030-61792-9_17`.

28 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`.

29 Sebastian Kreft and Gonzalo Navarro. Self-indexing based on LZ77. In *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings*, volume 6661 of *Lecture Notes in Computer Science*, pages 41–54. Springer, 2011. `doi:10.1007/978-3-642-21458-5_6`.

30 Moshe Lewenstein. Indexing with gaps. In *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings*, volume 7024 of *Lecture Notes in Computer Science*, pages 135–143. Springer, 2011. `doi:10.1007/978-3-642-24583-1_14`.

31 Moshe Lewenstein, J. Ian Munro, Yakov Nekrich, and Sharma V. Thankachan. Document retrieval with one wildcard. In *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*, volume 8635 of *Lecture Notes in Computer Science*, pages 529–540. Springer, 2014. `doi:10.1007/978-3-662-44465-8_45`.

32 Moshe Lewenstein, J. Ian Munro, Venkatesh Raman, and Sharma V. Thankachan. Less space: Indexing for queries with wildcards. *Theor. Comput. Sci.*, 557:120–127, 2014. `doi:10.1016/J.TCS.2014.09.003`.

33 Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France*, volume 25 of *LIPIcs*, pages 506–517. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. `doi:10.4230/LIPICS.STACS.2014.506`.

34 Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinform.*, 25(14):1754–1760, 2009. `doi:10.1093/BIOINFORMATICS/BTP324`.

35 Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *J. Comput. Biol.*, 17(3):281–308, 2010. `doi:10.1089/CMB.2009.0169`.

36 César Martínez-Guardiola, Nathaniel K. Brown, Fernando Silva-Coira, Dominik Köppl, Travis Gagie, and Susana Ladra. Augmented thresholds for MONI. In *Data Compression Conference, DCC 2023, Snowbird, UT, USA, March 21-24, 2023*, pages 268–277. IEEE, 2023. `doi:10.1109/DCC55655.2023.00035`.

37 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`.

38 Gonzalo Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2022. `doi:10.1145/3432999`.

39 Gonzalo Navarro. Computing mems and relatives on repetitive text collections. *ACM Trans. Algorithms*, 21(1):12:1–12:33, 2025. `doi:10.1145/3701561`.

40 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2, 2007. `doi:10.1145/1216370.1216372`.

41 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 101:1–101:15.

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICALP.2021.101.

42 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018. doi:10.1007/S00453-017-0327-Z.

43 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.

44 Massimiliano Rossi, Marco Oliva, Paola Bonizzoni, Ben Langmead, Travis Gagie, and Christina Boucher. Finding maximal exact matches using the r-index. *J. Comput. Biol.*, 29(2):188–194, 2022. doi:10.1089/CMB.2021.0445.

45 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A pangenomic index for finding maximal exact matches. *J. Comput. Biol.*, 29(2):169–187, 2022. doi:10.1089/CMB.2021.0290.

46 Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Trans. Algorithms*, 7(4):53:1–53:34, 2011. doi:10.1145/2000807.2000821.

47 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

48 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

49 James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

50 Igor Tatarnikov, Ardavan Shahrabi Farahani, Sana Kashgouli, and Travis Gagie. MONI can find k-MEMs. In *34th Annual Symposium on Combinatorial Pattern Matching, CPM 2023, June 26-28, 2023, Marne-la-Vallée, France*, volume 259 of *LIPIcs*, pages 26:1–26:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.CPM.2023.26.

51 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.

52 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

53 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.