Compressing Suffix Trees by Path Decompositions

Ruben Becker
Ca' Foscari University of Venice, Italy

Davide Cenzato
Ca' Foscari University of Venice, Italy

Travis Gagie
Dalhousie University, Halifax, Canada

Ragnar Groot Koerkamp
ETH Zurich, Switzerland
Karlsruhe Institute of Technology, Germany

Sung-Hwan Kim
Ca' Foscari University of Venice, Italy
Sant'Anna School of Advanced Studies, Pisa, Italy

Giovanni Manzini
University of Pisa, Italy

Nicola Prezza
Ca' Foscari University of Venice, Italy

Abstract

The suffix tree is arguably the most fundamental data structure on strings: introduced by Weiner (SWAT 1973) and McCreight (JACM 1976), it allows solving a myriad of computational problems on strings in linear time. Motivated by its large space usage, subsequent research focused first on reducing its size by a constant factor via Suffix Arrays, and later on reaching space proportional to the size of the compressed string. Modern compressed indexes, such as the r-index (Gagie et al., JACM 2020), fit in space proportional to r, the number of runs in the Burrows-Wheeler transform (a strong and universal repetitiveness measure). These advances, however, came with a price: while modern compressed indexes boast optimal bounds in the RAM model, they are often orders of magnitude slower than uncompressed counterparts in practice due to catastrophic cache locality. This reality gap highlights that Big-O complexity in the RAM model has become a misleading predictor of real-world performance, leaving a critical question unanswered: can we design compressed indexes that are efficient in the I/O model of computation?

We answer this in the affirmative by introducing a new Suffix Array sampling technique based on particular path decompositions of the suffix tree. We prove that sorting the suffix tree leaves by specific priority functions induces a decomposition where the number of distinct paths (each corresponding to a string suffix) is bounded by r. This allows us to solve indexed pattern matching efficiently in the I/O model using a Suffix Array sample of size at most r, strictly improving upon the (tight) $2r$ bound of Suffixient Arrays, another recent compressed Suffix Array sampling technique.

Experiments confirm that this theoretical I/O efficiency translates to practice in pangenomic applications: our index locates pattern occurrences using less space and orders of magnitude less time than the r-index when performing pattern matching on repetitive DNA collections. Beyond this, our contributions are twofold: (i) unlike Suffixient Arrays, our technique supports most standard suffix tree operations in $O(r)$ space on top of the text while matching the I/O complexity of uncompressed suffix trees; and (ii) we establish a general framework where any valid path decomposition induces a Suffix Array sampling whose size is a new strong repetitiveness measure; we provide a universal mechanism for locating all pattern occurrences for each such path decomposition.

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

Keywords and phrases Text indexing, suffix tree, I/O-efficient, Compressed Data Structures

53rd International Colloquium on Automata, Languages, and Programming (ICALP 2026).
Editors: Sayan Bhattacharya, Danupon Nanongkai, Michael Benedikt, and Gabriele Puppis;
Article No. 24; pp. 24:1–24:25

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

Digital Object Identifier 10.4230/LIPIcs.ICALP.2026.24

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2506.14734 [4]

Supplementary Material *Software (Source Code)*: https://github.com/regindex/STPD-index [3] archived at swh:1:dir:d4f76a2be744c622f98d403443e3f4507ae9b536

Funding *Ruben Becker, Davide Cenzato, Sung-Hwan Kim, and Nicola Prezza*: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.
Ruben Becker and Giovanni Manzini: Funded by INdAM-GNCS (INdAM-GNCS Project CUP E53C25002010001).
Travis Gagie: Funded by NSERC (Discovery Grant RGPIN-07185-2020).

1 Introduction

In this paper, we describe a new elegant and very efficient paradigm to solve the well-studied problem of compressing suffix trees. Suffix trees were introduced in 1973 by Weiner [48] and revisited (in their modern form) in 1976 by McCreight [36] to solve string processing problems such as finding longest common substrings and matching patterns on indexed text in linear time. The latter problem (indexed string matching) asks to build a data structure on a text $\mathcal{T}$ of length n over alphabet of size σ so that later (at query time), given a string P (the pattern) of length $m \leq n$, the following can be returned: (1) one exact occurrence $\mathcal{T}[i, j] = P$ of P in $\mathcal{T}$ if any exists (*find* queries), (2) all exact occurrences of P in $\mathcal{T}$ (*locate* queries), or (3) the number of exact occurrences of P in $\mathcal{T}$ (*count* queries). While being extremely fast due to excellent query-time cache locality, suffix trees require a linear number of words to be stored in memory regardless of the input compressibility and are therefore not suitable to nowadays massive-data scenarios such as pan-genome indexing (where one aims at indexing terabytes of data in the form of repetitive collections of thousands of genomes). This problem was later mitigated by Suffix Arrays [22, 34, 35], which use only a constant fraction of the space of suffix trees while supporting (cache-efficiently) a subset of their functionality, still sufficient to support pattern matching queries.

Subsequent research succeeded (spectacularly) in reducing the space usage of suffix trees and Suffix Arrays to the bare minimum needed to store the *compressed text*. Notable contributions in this direction include the compressed Suffix Array (CSA) [23], FM-index [18], run-length compressed Suffix Array [33], run-length FM-index [32], r-index [21], Lempel-Ziv-based [29] and indexes based on straight-line programs (SLPs) [14] (and their variants), and the more recent δ-SA [26]. See the survey of Navarro [39] for an extensive treatment of the subject. While the first line of work (CSA and FM-index) focused on entropy compression, the subsequent works mentioned above switched to text compressors capable of exploiting the *repetitiveness* of the underlying text sequence (a source of redundancy that entropy compression is not able to exploit). Among those, the r-index [20, 21] and its improvements [7, 41, 51] stood out for its optimal linear-time pattern matching query time and its size – linear in the number r of equal-letter runs in the Burrows-Wheeler transform (BWT) of the text. These works on repetition-aware compressed text indexes spurred a very fruitful line of research on compressibility measures (the survey of Navarro [38] covers the subject in detail), culminating in recent breakthroughs [25, 27] which showed that r is a strong and

universal repetitiveness measure, being equivalent to all other known compressibility measures (such as the size of the Lempel-Ziv factorization [31], normalized substring complexity [28], and straight-line programs) up to a multiplicative polylogarithmic factor. Altogether, these results laid the theoretical foundations for subsequent works in computational pan-genomics that showed how the run-length encoded BWT can successfully be used to index very large collections of related genomes in compressed space [1, 2, 16, 44, 45, 46].

1.1 Are repetition-aware data structures just a theoretical tool?

Modern compressed indexes such as the δ-SA of Kempa and Kociumaka [26] support random access and pattern matching queries, but their time complexities depend on a high polynomial of the logarithm of the text's length, which makes them hardly practical. Indexes based on the Lempel-Ziv factorization or on grammar compression mitigate this problem (even reaching optimal search time [13]), but rely on complex and cache-inefficient data structures that, again, make them orders of magnitude slower than simple suffix trees in practice. The r-index (in its modern version [41]) uses $O(r)$ words of space and solves *find*, *locate*, and *count* queries in $O(m)$ time (assuming constant alphabet for simplicity), plus the number of occurrences to be reported (if any). While this is essentially the end of the story in the word RAM model, it does not take into account caching effects. As a matter of fact, each of the $O(m)$ steps of the *backward search* algorithm of the r-index and of its predecessor (the FM-index [18]) triggers I/O operations (that is, likely cache misses). While this issue was later partially addressed by the *move* structure of Nishimoto and Tabei [41], that solution still triggers $O(m)$ cache misses. This does not happen with the suffix tree:

▶ Remark 1. The suffix tree of Weiner [48] and McCreight [36] uses $O(n)$ words on top of the plain text ($n \log \sigma$ bits) and allows locating all the *occ* occurrences of any pattern P of length m with $O(d + m/B + occ)$ I/O complexity, where d is the node depth of P in the suffix tree and B is the number of integers fitting in an I/O block.

To see this, observe that path compression makes it possible to compare (a substring of) the pattern with the label of an edge of length ℓ with $O(\ell/B + 1)$ I/O complexity. In particular, each of the d edge traversals triggers at least one I/O operation in the worst case. The additive term d is negligible in practice as in most interesting scenarios the suffix tree tends to branch mostly in the highest levels (we investigate this effect in the full version of this paper [4]). For instance, if the text is uniform then $d \in \Theta(\log n)$ with high probability since the longest repeated substring's length is $\Theta(\log n)$ w.h.p.

The performance gap in the I/O model between compressed indexes and the suffix tree shows up dramatically in practice: a simple experiment (see [4]) shows that, while the r-index is orders of magnitude smaller than the suffix tree on very repetitive inputs, it also solves queries *orders of magnitude slower*. The same holds true for all the existing compressed indexes using a space close to that of the r-index. This reality gap highlights that Big-O complexity in the RAM model has become a misleading predictor of real-world performance in the context of compressed data structures, leaving a critical question unanswered:

Can compressed indexes be efficient in the I/O model of computation?

1.2 Our contributions

We answer this question in the affirmative by introducing a novel *compressed sampling* of the Prefix Array. In one configuration, our sampling requires at most r samples to resolve pattern matching queries via a simple binary search strategy, assuming random access to the text is

available. Since random access in compressed space is a well-studied problem [6, 8, 29, 30] (see also [38] for a survey on the topic), this requirement does not limit our strategy. On the contrary, unlike existing compressed indexes, our approach is flexible enough to utilize any random access compressed data structure.

By incorporating additional data structures, we achieve our main result: a compressed suffix tree topology occupying only $O(r)$ words of space on top of a (potentially compressed) text oracle that still efficiently supports most standard navigation queries:

▶ **Theorem 2.** *Let $\mathcal{T}$ be a text of length n over an integer alphabet of size σ. Assume we have access to an oracle supporting longest common extension (*lce*) and random access queries (extraction of one character) on $\mathcal{T}$ in $O(t)$ time. Then, there is a representation of $\mathcal{T}$'s suffix tree using $O(r)$ words on top of the text oracle and supporting these queries:*

- *root() in $O(1)$ time: the suffix tree root.*
- *child(u, a) in $O(t \log r + \log \sigma)$ time: the child of node u by letter a.*
- *first(u) in $O(\log \sigma)$ time: the alphabetically-smallest label among the outgoing edges of u.*
- *succ(u, a) in $O(\log \sigma)$ time: given node u and a character a labeling one of the outgoing edges of u, return the successor of a (in alphabetic order) among the characters labeling outgoing edges of u (return $\perp$ if no such label exists).*
- *label(u, v) in $O(1)$ time: given an edge (u, v), return $(i, j) \in [n]^2$ such that the edge's label is $\mathcal{T}[i, j]$.*
- *lleaf(u), rleaf(u) in $O(1)$ time: the leftmost/rightmost leaves of the subtree rooted in a given node u.*
- *next(u): if u is a leaf, return the next leaf in lexicographic order; we support following a sequence of k leaf pointers in $O(\log \log(n/r) + k)$ time.*
- *locate(u) in $O(1)$ time: the text position i of a given leaf (representing suffix $\mathcal{T}[i, n]$).*
- *sdepth(u) in $O(1)$ time: the string depth of node u.*
- *ancestor(u, v) in $O(t)$ time: whether u is an ancestor of v.*

If the text oracle also supports computing a collision-free (on text substrings) hash $\kappa(\mathcal{T}[i, j])$ of any text substring in $O(h)$ time (an operation we call fingerprinting*), then child(u, a) can be supported in $O(t + h \log n + \log \sigma)$ time within the same asymptotic space.*

The same asymptotic bounds apply to I/O complexity if t and h represent I/O costs rather than time.

A subset of the above queries suffices to navigate the suffix tree (a task at the core of several string-processing algorithms) and to solve pattern matching queries. For example, using the cache-efficient text representation of Prezza [42], supporting lce with $O(t) = O(\log n)$ I/O complexity, fingerprinting with $O(h) = O(1)$ I/O complexity, and extraction of ℓ contiguous characters with $O(1 + \ell/B)$ I/O complexity on polynomial alphabets, we obtain the following:

▶ **Corollary 3.** *Let $\mathcal{T}$ be a text of length n over an integer alphabet of size σ. The topology of $\mathcal{T}$'s suffix tree can be compressed in $O(r)$ words on top of a text representation [42] of $n \log \sigma + O(\log n)$ bits so that all the occ occurrences of any pattern P of length m can be located with $O(d \log n + m/B + occ)$ I/O complexity, where d is the node depth of P in the suffix tree and B is the number of integers fitting in an I/O block.*

That is, the same I/O complexity of Weiner and McCreight's suffix tree up to a logarithmic factor multiplying d (see Remark 1). At the same time, the space usage on top of the text is reduced from $O(n)$ to $O(r)$ words (r is orders of magnitude smaller than n on very repetitive inputs [21]). Furthermore, the term $d \log n$ can be replaced by $d \log m$ with a different technique (Theorem 40; this is important since in typical applications, $m \ll n$ holds).

Using up-to-date cache-efficient compressed data structures, we show experimentally that an optimized implementation of our fully-compressed index is simultaneously *smaller* and *orders of magnitude faster* than the r-index on the task of locating *all* pattern occurrences on a highly repetitive collection of genomes.

1.3 Improvements over related techniques

Suffixient Array. Our approach is most closely related to the *Suffixient Array* [10], another recent compressed sampling of the Prefix Array. We strictly improve upon the Suffixient Array in three dimensions:

- **Space Bounds.** The Suffixient Array requires up to $2r$ samples [40], a bound recently proven to be worst-case tight up to a small additive constant [17]. Our technique demonstrates that r samples are sufficient for pattern matching.
- **Functionality.** Unlike Suffixient Arrays, which are limited to pattern matching, our sampling retains the topological structure of the suffix tree. Consequently, we support a much broader range of suffix tree operations efficiently.
- **Locating Efficiency.** Suffixient Arrays return an *arbitrary* pattern occurrence, offering no control over which one is found. To locate *all* occurrences starting from an arbitrary position $SA[i]$, one requires bidirectional navigation (both $SA[i-1]$ and $SA[i+1]$), costing about $4r$ words of auxiliary space [21]. In contrast, one of the configurations of our index guarantees returning the *first* pattern occurrence in Suffix Array order. This allows us to locate all remaining occurrences using only unidirectional successor Suffix Array queries (i.e., retrieving $SA[i+1]$ given $SA[i]$) with $2r$ words of auxiliary space [21].

Compressed Suffix Trees. Gagie et al. [21] previously addressed compressed suffix trees in space bounded by a function of r, showing that full functionality is possible in $O(r \log(n/r))$ space with logarithmic operation time. When augmented with a I/O efficient text oracle, their theoretical I/O complexity for pattern matching matches our Corollary 3; however, their approach consumes significantly more space on top of the oracle ($O(r \log(n/r))$ vs $O(r)$). Furthermore, their structure relies on complex machinery atop a grammar-compressed Suffix Array; to our knowledge, it has never been implemented, likely due to the practical reality that Suffix Arrays do not compress as effectively as the text itself via grammar compression.

Other I/O Efficient Solutions. Other works in the literature addressed the I/O-efficiency of full-text indexes. Chien et al. [12] introduced the Geometric Burrows-Wheeler Transform, which connects range searching with text indexing to provide compressed indexing techniques and theoretical lower bounds within the I/O model. Their solution, however, achieves I/O-efficient performance only in the uncompressed setting. Similarly, Ferragina and Venturini [19] proposed a compressed version of the string B-tree, adapting classical external-memory string structures to use compressed space while remaining efficient across memory hierarchies. While their compressed cache-oblivious string B-Tree supports I/O-efficient prefix search and enumeration, its space for *locate* queries remains proportional to the number of suffixes.

As far as fully-compressed space is concerned, several recent works have targeted the I/O bottleneck of the r-index. The *Move* structure of Nishimoto and Tabei [41] reduces the I/O operations of the r-index by a $\log \log n$ factor by replacing predecessor queries with pointers. However, it retains a worst-case complexity of $O(m + occ)$ I/O operations. While practical implementations [7, 51] are roughly an order of magnitude faster than the standard r-index, they require significantly more space. Similarly, Puglisi and Zhukova [43] applied relative Lempel-Ziv compression to a transformed Suffix Array. While this yields good memory

locality and query speeds in practice, it lacks formal guarantees and increases the space usage of the r-index by an order of magnitude, again suffering from the poor compressibility of the Suffix Array relative to the text.

2 Order-preserving Suffix Tree Path Decompositions

Due to space limitations, we assume the reader to be already familiar with the following standard concepts: *right-maximal strings*, *lexicographic* and *colexicographic* order, *longest common suffix/prefix* function $\mathrm{lcp}(\alpha, \beta)$ / $\mathrm{lcs}(\alpha, \beta)$ between strings, *Suffix Array*, *suffix tree*, and *Burrows-Wheeler transform*. See the full version [4] for the formal definitions.

▶ **Definition 4** (rlce, llce, and lce)**.** *Let $\mathcal{S}$ be a string of length n. The* right (left) longest common extension *function $\mathcal{S}$.*rlce *($\mathcal{S}$.*llce*) is the function that, for two distinct integers $i, j \in [n]$, returns $\mathcal{S}.\mathrm{rlce}(i,j) = \mathrm{lcp}(\mathcal{S}[i,n], \mathcal{S}[j,n])$ ($\mathcal{S}.\mathrm{llce}(i,j) = \mathrm{lcs}(\mathcal{S}[1,i], \mathcal{S}[1,j])$). A* longest common extension (lce) *oracle is an oracle that supports both $\mathcal{S}$.*rlce *and $\mathcal{S}$.*llce *queries for the string $\mathcal{S}$.*

When clear from the context, we simply write rlce and llce instead of $\mathcal{S}$.rlce and $\mathcal{S}$.llce.

Our work can be interpreted as a generalization of suffix sorting: by sorting the text's suffixes according to any priority function (permutation) $\pi : [n] \to [n]$ satisfying a natural and desirable *order-preserving* property (see Definition 5), we obtain a compressed index that keeps track of the pattern occurrence $\mathcal{T}[i,j]$ minimizing $\pi(i)$ among all pattern occurrences. Figure 1 broadly introduces our solution. Intuitively, we (i) sort the suffix tree's leaves according to π, (ii) we build a *suffix tree path decomposition* (STPD for brevity) prioritizing the paths with smaller π, and (iii) we path-compress the STPD paths by just recording their starting position in the text. At this point, we show that the colexicographically-sorted *Path Decomposition Array* PDA of those positions (a sample of the Prefix Array) can be used to obtain a new elegant, simple, and remarkably efficient compressed suffix tree.

As introduced above, the starting point of our technique is the choice of a priority function (permutation) $\pi : [n] \to [n]$ that we use to generalize suffix sorting. In this paper, we focus on permutations possessing the following natural property.

▶ **Definition 5** (Order-preserving permutation)**.** *Let $\mathcal{S} \in \Sigma^n$ be a string and $\pi : [n] \to [n]$ be a permutation. The permutation π is said to be* order-preserving for $\mathcal{S}$ *if and only if $\pi(i) < \pi(j) \wedge \mathcal{S}[i, i+1] = \mathcal{S}[j, j+1]$ implies $\pi(i+1) < \pi(j+1)$ for all $i, j \in [n-1]$.*

The above property is sufficient and necessary to guarantee the following desirable universal minimization property: if $\mathcal{S}[i,j] = \mathcal{S}[i',j']$ are two pattern occurrences, then $\pi(i+k) < \pi(i'+k)$ either holds for all $k \in [0, j-i]$ or for none. In Lemmas 15 and 41 we will show that the lexicographic rank of suffixes, the colexicographic rank of prefixes, and the identity function are order-preserving permutations (but not the only ones).

Suffix tree path decomposition. A *suffix tree path decomposition* (STPD) is an edge-disjoint collection of node-to-leaf paths covering all the suffix tree's edges built as follows. Since we will never start a path on an implicit suffix tree node, we can equivalently reason about path decompositions of the *suffix trie*. We describe how to obtain the STPD associated with a given order-preserving permutation π, as we believe this will help the reader to better understand our technique. Then, we will make the construction fully formal. Let $\mathcal{T} \in \Sigma^n$ be a text. Imagine the process of inserting $\mathcal{T}$'s suffixes $\mathcal{T}[i,n]$ (for $i \in [n]$) in a trie in order of increasing $\pi(i)$. The path associated with the first suffix $\mathcal{T}[\pi^{-1}(1), n]$ in this order is the

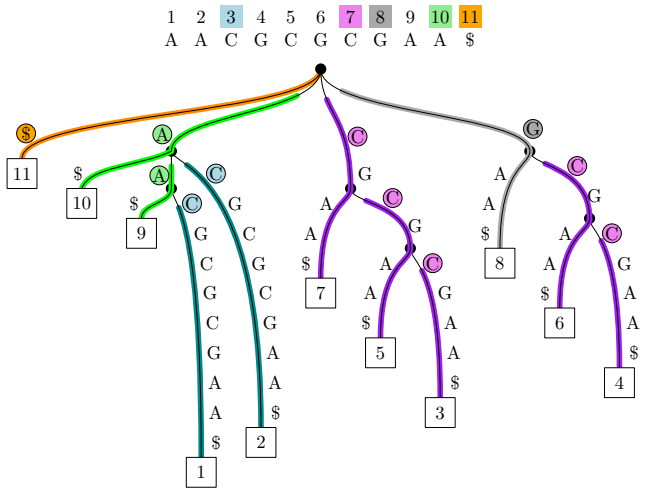

Figure 1 Overview of our technique. We sort $\mathcal{T}$'s suffixes $\mathcal{T}[i, n]$ (equivalently, suffix tree leaves) by increasing $\pi(i)$. In this example, π corresponds to the standard lexicographic order of the text's suffixes (but π can be more general). This induces a *suffix tree path decomposition* (an edge-disjoint set of node-to-leaf paths covering all edges) obtained by always following the leftmost path. At this point, we associate each path with the integer i such the path's label is $\mathcal{T}[i, n]$ (in the figure, we also color each path according to the color of the associated position i). In particular, the label from the root to the first path's edge is $\alpha_{i,k} = \mathcal{T}[i - k + 1, i]$, where k is the string depth of the first path's edge. Our indexing strategy essentially consists in implicitly colexicographically-sorting strings $\alpha_{i,k}$ in a subset of the Prefix Array called the *Path Decomposition Array* PDA: the colexicographically-sorted array containing (without duplicates) the first position of each path (in our example: PDA $= [11, 10, 3, 7, 8]$). Observe that, in this example, there are few (five) *distinct* path labels: indeed, we show that |PDA| is bounded by universal compressibility measures and that it can be used to support basic suffix tree navigation and pattern matching operations.

one starting in the root and continuing with characters $\mathcal{T}[\pi^{-1}(1), n]$. When inserting the j-th ($j > 1$) suffix $\mathcal{T}[\pi^{-1}(j), n]$, let $k = \max_{j'<j} \text{rlce}(\pi^{-1}(j'), \pi^{-1}(j))$ be the longest common prefix between the j-th suffix and all the previous suffixes in the order induced by π. The corresponding new path in the decomposition is the one starting in the suffix tree locus of string $\mathcal{T}[\pi^{-1}(j), \pi^{-1}(j) + k - 1]$ and labeled with string $\mathcal{T}[\pi^{-1}(j) + k, n]$. In other words, the path associated with $\mathcal{T}[\pi^{-1}(j), n]$ is its suffix that "diverges" from the trie containing the previous suffixes in the order induced by π. See Figure 1, where leaves (suffixes) $\mathcal{T}[i, n]$ are sorted left-to-right in order of increasing $\pi(i)$ where $\pi = ISA$.

The core of our indexing strategy is to store in a colexicographically-sorted array PDA all the *distinct* integers $\pi^{-1}(j) + k$ obtained in this process (that is, the starting positions of paths in $\mathcal{T}$). We now formalize this intuition. Due to space limitations, most proofs are deferred to the full version of this article [4].

LPF and PDA arrays. We first introduce the concept of the Generalized Longest Previous Factor Array LPF_π. Intuitively, this array stores the lengths of the longest common prefixes of a string's suffixes in the order induced by π. It generalizes the well-known *Permuted Longest Common Prefix* (PLCP) array (when taking $\pi = \text{ISA}$ to be the lexicographic rank of the text's suffixes) and the LPF array (when taking $\pi = id$ to be the identity function).

▶ **Definition 6** (Generalized Longest Previous Factor Array $\mathrm{LPF}_{\mathcal{S},\pi}$). *Let $\mathcal{S} \in \Sigma^n$ be a string and $\pi : [n] \to [n]$ be a permutation. The* generalized Longest Previous Factor array $\mathrm{LPF}_{\mathcal{S},\pi}[1,n]$ *associated with $\mathcal{S}$ and π is the integer array that, for $i \in [n]$, is defined as:*

$$\mathrm{LPF}_{\mathcal{S},\pi}[i] = \begin{cases} 0 & \text{if } \pi(i) = 1, \\ \max_{\pi(j)<\pi(i)} \mathrm{rlce}(j,i) & \text{otherwise.} \end{cases}$$

▶ **Definition 7** (Path Decomposition Array $\mathrm{PDA}_{\mathcal{S},\pi}$). *Let $\mathcal{S} \in \Sigma^n$ be a string and $\pi : [n] \to [n]$ be an order-preserving permutation. The* (suffix tree) Path Decomposition Array $\mathrm{PDA}_{\mathcal{S},\pi}$ *associated with $\mathcal{S}$ and π is the set $\{j = i + \mathrm{LPF}_{\mathcal{S},\pi}[i] \ : \ i \in [n]\}$ sorted in colexicographic order of the corresponding string's prefixes $\mathcal{S}[1,j]$.*

It will always be the case that the indexed string $\mathcal{S}$ is fixed in our discussion, so we will simply write LPF_π and PDA_π instead of $\mathrm{LPF}_{\mathcal{S},\pi}$ and $\mathrm{PDA}_{\mathcal{S},\pi}$, respectively. When also π is clear from the context, we will write LPF and PDA.

▶ **Example 8.** Consider the STPD of Figure 1. In this example, $\pi(i)$ is the rank of leaf i in lexicographic order (in other words, $\pi = \mathrm{ISA}$): $\pi = (4,5,8,11,7,10,6,9,3,2,1)$. Then, the LPF array corresponds to the PLCP array: $\mathrm{LPF} = \mathrm{PLCP} = [2,1,4,3,2,1,0,0,1,0,0]$. For instance, $\mathrm{LPF}[1] = 2$ because $\pi(1) = 4$ and the suffixes $\mathcal{T}[j,n]$ with $\pi(j) < \pi(1)$ are those starting in positions $j \in \{11,10,9\}$. Among those, the one with the longest common prefix with $\mathcal{T}[1,n]$ is $\mathcal{T}[9,n]$, and their longest common prefix is $2 = \mathrm{LPF}[1]$.

At this point, the sequence $i+\mathrm{LPF}[i]$ for $i = 1,\ldots,n$ is equal to $(3,3,7,7,7,7,7,8,10,10,11)$. The *Path Decomposition Array* is the array containing the distinct values in such a sequence, sorted colexicographically: $\mathrm{PDA} = [11,10,3,7,8]$ (that is, j precedes j' in the order if and only if $\mathcal{T}[1,j]$ is colexicographically smaller than $\mathcal{T}[1,j']$).

The STPD associated with π and the corresponding array PDA_π are said to be *order-preserving* if π is order-preserving.

The expert reader might have noticed that, for the permutation π used in Example 8, the values in PDA are in a one-to-one correspondence with the *irreducible LCP values*. As a matter of fact, our technique generalizes the notion of *irreducible values* to any array LPF_π such that π is order-preserving. First, note that the array LPF_π is almost nondecreasing:

▶ **Lemma 9.** *For any string $\mathcal{S} \in \Sigma^n$, if π is order-preserving then $\mathrm{LPF}_\pi[i] \geq \mathrm{LPF}_\pi[i-1] - 1$ for every $i > 1$. In particular, the sequence $i + \mathrm{LPF}_\pi[i]$, for $i = 1,\cdots,n$, is nondecreasing.*

Values where the inequality of Lemma 9 is strict are of particular interest:

▶ **Definition 10** (Irreducible LPF_π position). *Let $\mathcal{S} \in \Sigma^n$ be a string and $\pi : [n] \to [n]$ be an order-preserving permutation. We say that $i \in [n]$ is an* irreducible LPF_π position *if and only if either $i = 1$ or $\mathrm{LPF}_\pi[i] \neq \mathrm{LPF}_\pi[i-1] - 1$.*

It is a simple observation that $\mathrm{LPF}_\pi[i] = \mathrm{LPF}_\pi[i-1]-1$ is equivalent to $(i-1)+\mathrm{LPF}_\pi[i-1] = i + \mathrm{LPF}_\pi[i]$, from which we obtain:

▶ Remark 11. For any order-preserving permutation $\pi : [n] \to [n]$, $\{x \in \mathrm{PDA}_\pi\} = \{i + \mathrm{LPF}_\pi[i] \ : \ i \text{ is an irreducible } \mathrm{LPF}_\pi \text{ position}\}$, hence $|\mathrm{PDA}_\pi|$ is equal to the number of irreducible LPF_π positions (since PDA_π contains distinct values).

The following lemma generalizes the well-known relation between irreducible PLCP values and the Burrows-Wheeler transform [24, Lemma 4] in more general terms.

▶ **Lemma 12.** *Let $\mathcal{S} \in \Sigma^n$ be a string and $\pi : [n] \to [n]$ be an order-preserving permutation. Let moreover $i > 1$ be an irreducible LPF_π position. Then, for every $j > 1$ with $\pi(j-1) < \pi(i-1)$ and $\mathrm{rlce}(i,j) = \mathrm{LPF}_\pi[i]$, it holds that $\mathcal{S}[i-1] \neq \mathcal{S}[j-1]$.*

We will use Remark 11 and Lemma 12 to bound the size of our compressed suffix tree.

Lemma 13 formalizes the following intuitive fact about order-preserving STPDs. Consider an STPD built on string $\mathcal{S}$ using order-preserving permutation π. If for a suffix tree node $u = \mathrm{locus}(\alpha)$, the three nodes $\mathit{parent}(u)$, u, and $\mathit{child}(u,a)$ belong to the same STPD path (for some $a \in \Sigma$), then for any other outgoing label $b \in \Sigma$ of node u, string $\alpha \cdot b$ suffixes at least one sampled prefix $\mathcal{S}[1,t]$, for some $t \in \mathrm{PDA}_\pi$.

▶ **Lemma 13.** *Let $\mathcal{S} \in \Sigma^n$ be a string and $\pi : [n] \to [n]$ be an order-preserving permutation.*

For $\alpha \in \Sigma^*$ and $a \in \Sigma$ such that $\alpha \cdot a$ occurs in $\mathcal{S}$, let $\pi(\alpha \cdot a) = \min\{\pi(i-1) : \mathcal{S}[1,i]$ is suffixed by $\alpha \cdot a\}$. Then, for any characters $a, b \in \Sigma$ such that both $\alpha \cdot a$ and $\alpha \cdot b$ occur in $\mathcal{S}$ and $\pi(\alpha \cdot a) < \pi(\alpha \cdot b)$, there exists $t \in \mathrm{PDA}_\pi$ such that $\mathcal{S}[1,t]$ is suffixed by $\alpha \cdot b$.

We conclude with the following result, implying that $|\mathrm{PDA}_\pi|$ is a reachable compressibility measure for any order-preserving π.

▶ **Theorem 14.** *Let $\mathcal{S} \in \Sigma^n$ be a string over integer alphabet Σ and let $\pi : [n] \to [n]$ be an order-preserving permutation. Then, $\mathcal{S}$ can be compressed in $O(|\mathrm{PDA}_\pi| \log n)$ bits of space.*

Proof. (Sketch) For each irreducible LPF_π position i, we store a quadruple $(i, \mathrm{LPF}_\pi[i], s_i, \mathcal{S}[i+\mathrm{LPF}_\pi[i]])$, where $s_i \in [n]$ is the *source* of i, that is, any integer with $\pi(s_i) < \pi(i)$ and $\mathcal{S}[s_i, s_i+\mathrm{LPF}_\pi[i]-1] = \mathcal{S}[i, i+\mathrm{LPF}_\pi[i]-1]$. This set of quadruples takes $O(|\mathrm{PDA}_\pi| \log(n\sigma)) = O(|\mathrm{PDA}_\pi| \log n)$ bits of space by Remark 11, and we can reconstruct any character $\mathcal{S}[j]$ given $j \in [n]$ by recursively following the source position. ◀

We proceed as follows. First, we discuss the notable order-preserving permutation given by the lexicographic rank of the text's suffixes (Section 2.1). This permutation will enable us to support most suffix tree operations in $O(r)$ space on top of the text oracle. Then, in Section 2.2 we describe a general locating mechanism working on any order-preserving STPD. After that, we focus on the colexicographic rank of the text's prefixes (Section 2.3). This case is particularly interesting because it allows us to simplify the locating algorithm of Section 2.2, and will lead to a practical solution. Finally, in Section 2.4 we consider yet another remarkable order-preserving permutation: identity. This permutation will allow us to locate efficiently the leftmost and rightmost pattern occurrences.

2.1 Lexicographic rank (`st-lex`): suffix tree navigation

Figure 1 depicts the STPD obtained by choosing $\pi = \mathrm{ISA} = \mathrm{SA}^{-1}$ to be the Inverse Suffix Array (in this subsection, π will always be equal to ISA). We denote with $\texttt{st-lex}^- = \mathrm{PDA}_\pi$ the path decomposition array associated with this permutation π. Similarly, $\texttt{st-lex}^+$ denotes the path decomposition array associated with the dual permutation $\bar{\pi}(i) = n - \mathrm{ISA}[i] + 1$. The following properties hold:

▶ **Lemma 15.** *Let $\mathcal{T} \in \Sigma^n$ be a text. The permutations $\pi, \bar{\pi}$ defined as $\pi(i) = \mathrm{ISA}[i]$ and $\bar{\pi}(i) = n - \mathrm{ISA}[i] + 1$ for $i \in [n]$ are order-preserving for $\mathcal{T}$. Furthermore, it holds that $|\texttt{st-lex}^-| \leq r$ and $|\texttt{st-lex}^+| \leq r$.*

2.1.1 Data structure

Let $\mathcal{T} \in \Sigma^n$ be a text. We describe a data structure of $O(r)$ space supporting the suffix tree queries of Theorem 2 on top of any text oracle supporting Longest Common Extension (and, optionally, fingerprinting) queries on $\mathcal{T}$. We store the following components.

(1) The array $\texttt{st-lex}$ containing the integers $\{j = (i-1) \,:\, i \in \texttt{st-lex}^- \cup \texttt{st-lex}^+ \cup \{n+1\}\} \subseteq \{0, \ldots, n\}$ sorted increasingly according to the colexicographic order of the corresponding text prefixes $\mathcal{T}[1, j]$ (if $j = 0$, then $\mathcal{T}[1, j]$ is the empty string). Note that, by Lemma 15, it holds that $|\texttt{st-lex}| \leq 2r + 1$.

(2) Let $\# \notin \Sigma$ be a new character not appearing in Σ (taken to be larger than all the characters in Σ). We store a string $\mathcal{L}[1, |\texttt{st-lex}|] \in (\Sigma \cup \{\#\})^{|\texttt{st-lex}|}$ defined as:

$$\mathcal{L}[i] = \begin{cases} \# & \text{if } \texttt{st-lex}[i] = n, \\ \mathcal{T}[\texttt{st-lex}[i] + 1] & \text{otherwise.} \end{cases}$$

We store $\mathcal{L}$ with a wavelet tree [37], taking $O(|\texttt{st-lex}|)$ space and supporting rank and select operations in $O(\log \sigma)$ time.

(3) The string $\mathcal{F}[1, |\texttt{st-lex}|] \in (\Sigma \cup \{\#\})^{|\texttt{st-lex}|}$ obtained by sorting lexicographically the characters of $\mathcal{L}$. We store $\mathcal{F}$ with a wavelet tree supporting rank and select operations in $O(\log \sigma)$ time.

(4) Let $\mathcal{LF}$ and $\mathcal{FL}$ be the permutations of $[|\texttt{st-lex}|]$ defined as follows. For any integers i, j, k, if $\mathcal{L}[i] = a$ is the k-th occurrence of a in $\mathcal{L}$ and $\mathcal{F}[i] = a$ is the k-th occurrence of a in $\mathcal{F}$, then $\mathcal{LF}(i) = j$ and $\mathcal{FL}(j) = i$ (note that $\mathcal{FL} = \mathcal{LF}^{-1}$). $\mathcal{LF}$ and $\mathcal{FL}$ can be evaluated in $O(\log \sigma)$ time with $O(1)$ rank and select operations on $\mathcal{L}$ and $\mathcal{F}$.

▶ **Definition 16.** *We denote with* $\texttt{st-lex}'$ *the permutation of* $\texttt{st-lex}$ *defined as follows: for any* $j \in [|\texttt{st-lex}|]$, $\texttt{st-lex}'[j] = \texttt{st-lex}[\mathcal{FL}(j)]$ *or, equivalently (since* $\mathcal{FL}$ *and* $\mathcal{LF}$ *are inverse of each other),* $\texttt{st-lex}[j] = \texttt{st-lex}'[\mathcal{LF}(j)]$.

We store one Range Minimum and one Range Maximum data structure on the array $\pi(\texttt{st-lex}')$, supporting queries in constant time in $O(|\texttt{st-lex}|)$ bits of space (we do not need to store $\texttt{st-lex}'$ explicitly).

▶ Remark 17. While this is not fundamental for our discussion below, it may be helpful from an intuitive point of view to observe that string $\mathcal{L}$, once removed character # from it, is a subsequence of length $|\texttt{st-lex}| - 1$ of the colexicographic Burrows-Wheeler transform (BWT). Similarly, permutations $\mathcal{LF}$ and $\mathcal{FL}$ are the counterpart of functions LF and FL typically used with the BWT.

2.1.2 Node representation

Suffix tree navigation operations are supported on a particular representation of (explicit) suffix tree nodes that we describe next. First, Lemma 13 immediately implies:

▶ **Corollary 18.** *Let* u *be an explicit suffix tree node and* α *be such that* $u = \text{locus}(\alpha)$. *Then,* α *suffixes* $\mathcal{T}[1, j]$ *for at least one value* $j \in \texttt{st-lex}$.

In the Following, we describe our representation of suffix tree nodes.

▶ **Definition 19** (Suffix tree node representation)**.** *We represent suffix tree node* $u = \text{locus}(\alpha)$ *with the tuple of integers*

$R_u = (b, e, i_{min}, i_{max}, |\alpha|)$, *where*

- $\texttt{st-lex}[b, e]$ *is the colexicographic range of* α *in* $\texttt{st-lex}$ *(by Corollary 18,* $e \geq b$ *always holds);*
- $\mathcal{T}[i_{min}, i_{min} + |\alpha| - 1] = \alpha$ *is the occurrence of* α *minimizing* $\pi(i_{min})$*, that is,* $\mathcal{T}[i_{min}, n]$ *is the lexicographically-smallest suffix prefixed by* α*;*
- $\mathcal{T}[i_{max}, i_{max} + |\alpha| - 1] = \alpha$ *is the occurrence of* α *maximizing* $\pi(i_{max})$*, that is,* $\mathcal{T}[i_{max}, n]$ *is the lexicographically-largest suffix prefixed by* α*; and*
- $|\alpha|$ *is the string depth of* u*.*

▶ Remark 20. Observe that $n \in \texttt{st-lex}[1, 2]$. In particular, if $1 \in \texttt{st-lex}^- \cup \texttt{st-lex}^+$ then $0 \in \texttt{st-lex}$. Since 0 corresponds to the text's prefix $\mathcal{T}[1, 0]$ (the empty string), in this case $\texttt{st-lex}[1] = 0$ because the empty string is smaller than any other text prefix. In that case, $\texttt{st-lex}[2] = n$ because $\mathcal{T}[1, n]$ (ending with \$) is the second colexicographically-smallest sampled prefix. If, on the other hand, $1 \notin \texttt{st-lex}^- \cup \texttt{st-lex}^+$, then $\texttt{st-lex}[1] = n$.

Based on the above remark:

▶ **Definition 21.** *We denote with* $i^* \in \{1, 2\}$ *the integer such that* $\texttt{st-lex}[i^*] = n$*.*

Letting u be a leaf, observe that $\alpha(u)$ ends with character \$. It follows that the colexicographic range of $\alpha(u)$ in $\texttt{st-lex}$ is $\texttt{st-lex}[i^*, i^*]$. This will be used later.

2.1.3 Suffix tree operations

Next, we show how to support a useful subset of suffix tree operations on our data structure. This will prove Theorem 2. In the description below, suffix tree operations will take as input node representations (R_u) instead of nodes themselves (u). Recall that we are assuming we have access to a text oracle supporting longest common extension (lce) and random access queries on $\mathcal{T}$ in $O(t)$ time and fingerprinting queries in $O(h)$ time.

Root. Let $u = \text{locus}(\epsilon)$ be the suffix tree root. *root*() returns $R_u = (1, |\texttt{st-lex}|, i_{min}, i_{max}, 0)$ in $O(1)$ time, where $\mathcal{T}[i_{min}, n]$ is the lexicographically-smallest text suffix and $\mathcal{T}[i_{max}, n]$ is the lexicographically-largest text suffix (in other words, $i_{min} = \text{SA}[1] = n$ and $i_{max} = \text{SA}[n]$).

String depth. Let $R_u = (b, e, i_{min}, i_{max}, \ell)$. Then, *sdepth*($R_u$) returns ℓ in $O(1)$ time.

Ancestor Let $R_u = (b, e, i_{min}, i_{max}, \ell)$ and $R_{u'} = (b', e', i'_{min}, i'_{max}, \ell')$. If $\ell > \ell'$, u cannot be an ancestor of u' so *ancestor*($R_u, R_{u'}$) returns false. Otherwise, *ancestor*($R_u, R_{u'}$) returns true if and only if $\text{rlce}(i_{min}, i'_{min}) \geq \ell$, that is, if and only if the string $\alpha = \mathcal{T}[i_{min}, i_{min} + \ell - 1]$ with $u = \text{locus}(\alpha)$ is a prefix of $\alpha' = \mathcal{T}[i'_{min}, i'_{min} + \ell' - 1]$ with $u' = \text{locus}(\alpha')$. This operation runs in $O(t)$ time.

Is leaf. Let $R_u = (b, e, i_{min}, i_{max}, \ell)$. Then, *isleaf*($R_u$) returns true (in $O(1)$ time) if and only if $i_{min} = i_{max}$, if and only if $b = e = i^*$ (see Definition 21), if and only if the string $\alpha = \mathcal{T}[i_{min}, i_{min} + \ell - 1]$ with $u = \text{locus}(\alpha)$ ends with \$, that is, $i_{min} + \ell - 1 = n$ (equivalently, $\mathcal{T}[i_{min} + \ell - 1] = \$$).

Locate leaf. Let $R_u = (b, e, i_{min}, i_{max}, \ell)$. The output of *locate*(R_u) is defined only if *isleaf*(R_u) is true. In that case, *locate*(R_u) returns i_{min} ($= i_{max}$) in $O(1)$ time.

Leftmost/rightmost leaves. Let $R_u = (b, e, i_{min}, i_{max}, \ell)$. Then, in $O(1)$ time we can compute $lleaf(R_u) = (i^*, i^*, i_{min}, i_{min}, n - i_{min} + 1)$ and $rleaf(R_u) = (i^*, i^*, i_{max}, i_{max}, n - i_{max} + 1)$.

Edge label. Let (u, u') be a suffix tree edge, with $R_u = (b, e, i_{min}, i_{max}, \ell)$ and $R_{u'} = (b', e', i'_{min}, i'_{max}, \ell')$. Then, $label(R_u, R_{u'})$ returns $(i'_{min} + \ell, i'_{min} + \ell' - 1)$ in $O(1)$ time.

Next leaf. Nishimoto and Tabei [41] showed that, starting from $i \in [n]$, k consecutive applications $\bar{\phi}(i), \bar{\phi}^2(i), \ldots, \bar{\phi}^k(i)$ of the permutation defined below[1] can be computed in $O(\log\log(n/r) + k)$ time with a data structure using $O(r)$ words of space.

▶ **Definition 22** (ϕ-function). *Let* $\bar{\phi} : [n] \to [n]$ *be defined as*

$$\bar{\phi}(i) = \begin{cases} \text{SA}[\text{ISA}[i] + 1] & \text{if } \text{ISA}[i] < n, \\ \text{SA}[1] & \text{if } \text{ISA}[i] = n. \end{cases}$$

Observe that, by the definition of $\bar{\phi}$, leaf locus$(\mathcal{T}[\bar{\phi}(i), n])$ is the next leaf in lexicographic order after leaf locus$(\mathcal{T}[i, n])$ (unless the latter is the suffix tree's rightmost leaf).

Let u be a leaf, with $R_u = (i^*, i^*, i, i, n - i + 1)$ (see Definition 21 for the definition of i^*). Then, unless u is the rightmost leaf, $next(R_u) = (i^*, i^*, \bar{\phi}(i), \bar{\phi}(i), n - \bar{\phi}(i) + 1)$. It follows that the structure of Nishimoto and Tabei can be used to evaluate k consecutive applications of $next(\cdot)$ in $O(\log\log(n/r) + k)$ time.

Smallest children label and successor child. Let u be a node with $R_u = (b, e, i_{min}, i_{max}, \ell)$. Operation $first(R_u)$ returns the alphabetically-smallest label in $\mathcal{L}[b, e]$ (i.e., in out(u)). Operation $succ(R_u, a)$ returns the alphabetically-smallest label in $\mathcal{L}[b, e]$ (out(u)) being larger than a. Both queries reduce to an *orthogonal range successor* (also known as *range next value*) query in the range $\mathcal{L}[b, e]$, an operation that can be solved in $O(\log\sigma)$ time on wavelet trees [37]. In both queries, if $\# \in \mathcal{L}[b, e]$ then we simply ignore it.

Child by letter. The function $child(R_u, a)$ is the most technically-interesting operation. Let u be a node with $R_u = (b, e, i_{min}, i_{max}, \ell)$ and $a \in \Sigma$ be a letter. Let $\alpha = \mathcal{T}[i_{min}, i_{min} + \ell - 1]$ be such that $u = \text{locus}(\alpha)$. Lemma 13 implies the following corollary:

▶ **Corollary 23.** *Let* $u = \text{locus}(\alpha)$*, and let* `st-lex`$[b, e]$ *be the range containing the text positions* $j \in$ `st-lex` *such that* $\mathcal{T}[1, j]$ *is suffixed by* α*. All (and only) the letters labeling the outgoing edges from node* u *appear in* $\mathcal{L}[b, e] \setminus \{\#\}$*, that is,* $\text{out}(u) = \{c \ : \ c \in \mathcal{L}[b, e] \wedge c \neq \#\}$.

Following Corollary 23, if $a \notin \mathcal{L}[b, e]$ (a test taking $O(\log\sigma)$ time using rank and select operations on $\mathcal{L}$), then we can return $child(R_u, a) = (0, 0, 0, 0, 0)$, signaling that no outgoing edge from u is labeled with a.

Otherwise, we first show how to compute i'_{min} and i'_{max} such that $\mathcal{T}[i'_{min}, n]$ and $\mathcal{T}[i'_{max}, n]$ are the lexicographically-smallest and lexicographically-largest suffixes being prefixed by $\alpha \cdot a$, respectively. After that, we show how to use this information to compute $child(R_u, a)$.

Due to space constraints, we provide only the outlines here. The detailed description can be found in the full version [4].

[1] This permutation is usually denoted as ϕ^{-1}. Here we instead use the symbol $\bar{\phi}$.

Observe that $|\text{out}(u)| \geq 2$, since u is an explicit suffix tree node. We distinguish three cases. (i) a is neither the alphabetically-largest nor the alphabetically-smallest label in $\mathcal{L}[b,e] \setminus \{\#\}$, i.e. $a \neq \min\{c \in \mathcal{L}[b,e] \setminus \{\#\}\}$ and $a \neq \max\{c \in \mathcal{L}[b,e] \setminus \{\#\}\}$ hold; (ii) a is the alphabetically-smallest label in $\mathcal{L}[b,e] \setminus \{\#\}$, i.e., $a = \min\{c \in \mathcal{L}[b,e] \setminus \{\#\}\}$; (iii) a is the alphabetically-largest label in $\mathcal{L}[b,e] \setminus \{\#\}$, i.e., $a = \max\{c \in \mathcal{L}[b,e] \setminus \{\#\}\}$.

To simplify the discussion below, we rephrase Lemma 13 to the particular STPDs we are using in this section (i.e. those derived from $\pi = \text{ISA}$ and $\bar{\pi}[i] = n - \text{ISA}[i] + 1$):

▶ **Corollary 24.** *Let α be right-maximal, and let $\texttt{st-lex}[b,e]$ be the range containing the text positions $i \in \texttt{st-lex}$ such that $\mathcal{T}[1,i]$ is suffixed by α. Let moreover $C = \{c \in \Sigma : \alpha \cdot c$ occurs in $\mathcal{T}\}$ be the set of all the characters extending α in the text. Then:*

(a) *Let $c \in (C \setminus \min C)$, and let $\mathcal{T}[j - |\alpha| + 1, n]$ be the lexicographically-smallest suffix being prefixed by $\alpha \cdot c = \mathcal{T}[j - |\alpha| + 1, j + 1]$. Then, $j \in \texttt{st-lex}[b,e]$.*

(b) *Let $c \in (C \setminus \max C)$, and let $\mathcal{T}[j - |\alpha| + 1, n]$ be the lexicographically-largest suffix being prefixed by $\alpha \cdot c = \mathcal{T}[j - |\alpha| + 1, j + 1]$. Then, $j \in \texttt{st-lex}[b,e]$.*

We now show how to compute i'_{min} and i'_{max}. The three cases are considered as follows:

(i) By Corollary 24, $i''_{min}, i''_{max} \in \texttt{st-lex}[b,e]$, where $\mathcal{T}[i''_{min} - \ell + 1, n]$ and $\mathcal{T}[i''_{max} - \ell + 1, n]$ are the lexicographically smallest and largest suffixes being prefixed by $\alpha \cdot a$, respectively. We locate the leftmost and rightmost occurrences of letter a in $\mathcal{L}[b,e]$ using rank/select operations on $\mathcal{L}$, in $O(\log \sigma)$ time. We apply $\mathcal{LF}$ to locate corresponding a's in $\mathcal{F}$. Range minimum/maximum queries in range $\pi(\texttt{st-lex}')[b', e']$ followed by applying $\mathcal{FL}$, i''_{min} and i''_{max} can be retrieved. Then we obtain $i'_{min} = i''_{min} - \ell + 1$ and $i'_{max} = i''_{max} - \ell + 1$.

(ii) We can find i''_{max} following the same procedure described in (i) above (resorting to Range Maximum Queries). And since $\mathcal{T}[i_{min}, n]$ is the lexicographically-smallest suffix being prefixed by $\alpha \cdot a$, we have $i'_{min} = i_{min}$ and $i'_{max} = i''_{max} - \ell + 1$.

(iii) Symmetric to (ii).

If $i'_{min} = i'_{max}$, $\mathit{child}(R_u, a)$ is a leaf, thus we simply return the tuple $\mathit{child}(R_u, a) = (i^*, i^*, i'_{max}, i'_{max}, n - i'_{max} + 1)$. If $i'_{min} \neq i'_{max}$ (i.e., $\mathit{child}(R_u, a)$ is not a leaf), note that $\ell' = \text{rlce}(i'_{min}, i'_{max})$ is the string depth of $\mathit{child}(u, a)$. In particular, $\mathit{child}(u, a) = \text{locus}(\alpha')$, where $\alpha' = \mathcal{T}[i'_{min}, i'_{min} + \ell' - 1]$.

Simple binary search can compute $[b'', e'']$ such that for all $j \in [b'', e'']$, α' is a suffix of $\mathcal{T}[1, \texttt{st-lex}[j]]$, with $O(\ell' \log |\texttt{st-lex}|)$ random access queries. If the text oracle supports lce queries in $O(t)$ time, this process can be done in time $O(t \cdot \log |\texttt{st-lex}|) \subseteq O(t \cdot \log r)$. With *z-fast tries* [5, 9] on $\{\mathcal{T}[1, \texttt{st-lex}[i]] : i \in [|\texttt{st-lex}|]\}$, we can find $[b'', e'']$ in $O(h \log n)$ time (I/O operations, resp.) where $O(h)$ is the time (I/O complexity, resp.) for fingerprinting.

We finally have all ingredients to return our result: $\mathit{child}(R_u, a) = (b'', e'', i'_{min}, i'_{max}, \ell')$.

Putting everything together. To sum up, our data structure uses $O(|\texttt{st-lex}| + r) = O(r)$ space on top of the text oracle. Taking into account the complexity of all queries discussed above, this proves Theorem 2. By plugging the random access oracle in [42], we obtain:

▶ **Corollary 3.** *Let $\mathcal{T}$ be a text of length n over an integer alphabet of size σ. The topology of $\mathcal{T}$'s suffix tree can be compressed in $O(r)$ words on top of a text representation [42] of $n \log \sigma + O(\log n)$ bits so that all the occ occurrences of any pattern P of length m can be located with $O(d \log n + m/B + occ)$ I/O complexity, where d is the node depth of P in the suffix tree and B is the number of integers fitting in an I/O block.*

2.2 General locating mechanism for order-preserving STPDs

In this subsection we provide a general locating mechanism working on any order-preserving STPD. First, we show how to locate the pattern occurrence $P = \mathcal{T}[i, j]$ minimizing $\pi(i)$ among all pattern occurrences. We call this occurrence *primary*. Then we show that, starting from the (unique) primary occurrence, we can locate the remaining ones (called *secondary*) by resorting to orthogonal point enclosure. As a matter of fact, this technique generalizes the r-index' ϕ function (Definition 22, cases $\pi = \text{ISA}$ and $\pi = \text{IPA}$) to arbitrary order-preserving permutations. This increased generality with respect to Section 2.1, comes at the price of not being able to support suffix tree queries (only pattern matching).

After this section, we will tackle the particular case $\pi = \text{IPA}$ (colexicographic order of the text's prefixes), for which the locating algorithm that we describe here can be simplified. That particular case will lead to our optimized implementation able to beat the r-index both in query time (by orders of magnitude) and space usage.

For the remainder of the section, we assume $\mathcal{T} \in \Sigma^n$ is a text and π is any order-preserving permutation on $\mathcal{T}$. We start by classifying the occurrences of a pattern $P \in \Sigma^+$ as follows.

▶ **Definition 25** (Primary/Secondary occurrence)**.** *For a string $P \in \Sigma^+$, an occurrence $\mathcal{T}[i, i + |P| - 1] = P$ is said to be a* primary occurrence *if and only if* $\text{LPF}_\pi[i] < |P|$*. All the other occurrences are called* secondary occurrences*.*

▶ **Lemma 26.** *For any string $P \in \Sigma^+$ that occurs in $\mathcal{T}$ there exists exactly one primary occurrence $P = \mathcal{T}[i, i + |P| - 1]$. Furthermore, such occurrence is the one minimizing $\pi(i)$.*

2.2.1 Finding the primary occurrence

The idea to locate the primary pattern occurrence is simple and can be visualized as a process of walking along the paths of the STPD, starting from the root. Whenever we find a mismatch between the pattern P and the current STPD path, we change path by running a suffix search (e.g. binary search) on PDA_π. We first provide an intuition through an example (Example 27). Our algorithm is formalized in Algorithm 1. Then, we prove the algorithm's complexity, correctness, and completeness.

▶ **Example 27.** Consider the STPD of Figure 1, and let $P = CGCGAA$ be the query pattern. We start by matching P with the characters of the path $\mathcal{T}[11, 11] = \$$ starting at the root. The longest pattern prefix matching the path is ϵ (the empty string). To continue matching P, we need to change path. We binary search $\text{PDA}_\pi = \texttt{st-lex}^-$ looking for a sampled text prefix being suffixed by $\epsilon \cdot P[1] = C$ (i.e. the concatenation of the pattern's prefix matched so far and the first unmatched pattern's character). Two sampled prefixes (elements in $\texttt{st-lex}^-$) satisfy this requirement: $\mathcal{T}[1, 3]$ and $\mathcal{T}[1, 7]$. By definition of our STPD, among them we need to choose the one $\mathcal{T}[1, i]$ minimizing $\pi(i) = \text{ISA}[i]$. This choice can be performed in constant time with the aid of a Range Minimum Query data structure built on top of $\pi(\texttt{st-lex}^-)$. Such prefix minimizing $\pi(i)$ is $\mathcal{T}[1, 7]$. This means that we choose an STPD path labeled with $\mathcal{T}[7, 11]$. Observe that this also means that $\mathcal{T}[7]$ is the lexicographically-smallest occurrence of $P[1]$. We repeat the process, matching the remaining characters $P[1, 6]$ of P. As can be seen in Figure 1 characters $P[1, 2] = CG$ match (purple path starting below the root). Then, the path continues with A and the pattern with $P[3] = C$. As done above, we binary search $\text{PDA}_\pi = \texttt{st-lex}^-$ looking for a sampled text prefix being suffixed by $P[1, 2] \cdot P[3] = CGC$. Now, only one sampled prefix matches: $\mathcal{T}[1, 7]$, corresponding to an STPD path labeled with $\mathcal{T}[7, 11]$. Observe that this means that $\mathcal{T}[7]$ is the lexicographically-smallest occurrence of $P[3]$ being preceded by $P[1, 2]$; in

other words, $\mathcal{T}[7-2,7] = \mathcal{T}[5,7]$ is the lexicographically-smallest occurrence of $P[1,3]$. We therefore continue matching the remaining pattern's suffix $P[3,6]$ on $\mathcal{T}[7,11]$. This time, the whole pattern's suffix matches, hence we are done. Since the last binary search returned the sampled text's prefix $\mathcal{T}[1,7]$, and before running the search we already matched $P[1,2]$, we return pattern occurrence $\mathcal{T}[7-2,(7-2)+|P|-1] = \mathcal{T}[5,10]$ which, by construction of our STPD and by the order-preserving property of π, is the lexicographically-smallest one (in this example, also the only one).

Data structures and search algorithm. We show that, interestingly, a small modification of the search algorithm of Suffixient Arrays [10] (see Appendix A in the full version [4]) allows locating the primary pattern occurrence on any order-preserving STPD. The search algorithm is sketched in Example 27 and formalized in Algorithm 1. We need the following data structures:

- **(i)** A Range Minimum Data structure on $\pi(\text{PDA}_\pi)$, requiring just $O(|\text{PDA}_\pi|)$ bits and answering queries in constant time. In Algorithm 1, this data structure allows finding the arg min at Line 6 in constant time.
- **(ii)** A data structure supporting suffix searches on the text's prefixes $\{\mathcal{T}[1,\text{PDA}_\pi[t]] \ : \ t \in [|\text{PDA}_\pi|]\}$. Given $(i,j,c) \in [n] \times [n] \times \Sigma$, sufsearch$(i,j,c)$ returns the maximal range $[b,e] \subseteq [|\text{PDA}_\pi|]$ such that $\mathcal{T}[1,\text{PDA}_\pi[t]]$ is suffixed by $\mathcal{T}[i,j] \cdot c$ for all $t \in [b,e]$. This structure is used in Line 5 of Algorithm 1. We discuss different possible implementations for this structure below.
- **(iii)** A text oracle supporting random access on $\mathcal{T}$. To speed up sufsearch we may also require the oracle to support lce and/or fingerprinting queries on $\mathcal{T}$ (we obtain different performance depending on which queries are available, read below).

Algorithm 1 Locating the primary pattern occurrence.

Input: Pattern $P \in (\Sigma \setminus \{\$\})^m$.
Output: Primary occurrence of P, i.e. position $i \in [n]$ such that (1) $\mathcal{T}[i,i+m-1] = P$ and (2) i minimizes $\pi(i)$ among all the occurrences of P. If P does not occur in $\mathcal{T}$, return `NOT_FOUND`.

```
1  i ← π^{-1}(1);
2  j ← 1 ; /* Invariant:  P[1,j-1] = T[i-j+1,i-1] minimizes π(i-j+1) */
3  while j ≤ m do
4      if T[i] ≠ P[j] then
5          [b,e] ← sufsearch(i-j+1, i-1, P[j]);
6          i' ← arg min_{i'∈[b,e]}{π(PDA_π[i'])};
7          i ← PDA_π[i'];
8      i ← i+1; j ← j+1;
9  if T[i-m,i-1] = P then
10     return i-m;
11 else
12     return NOT_FOUND;
```

Complexity. As observed above, the arg min operation in Line 6 takes just $O(1)$ time using a Range Minimum data structure built over $\pi(\text{PDA}_\pi)$. For the analysis of sufsearch(i,j,c), we provide only the summarized result here due to space constraints. The detailed analysis is available in the full version [4]. If we use binary search on PDA_π and a random access

oracle on $\mathcal{T}$ that supports the extraction of ℓ contiguous text characters with $O(1+\ell/B)$ I/O complexity, then sufsearch(i,j,c) can be solved with $O((1+m/B)\log n)$ I/O complexity. If the random access oracle supports llce, this can be done in $O(t\cdot\log|\mathrm{PDA}_\pi|)\subseteq O(t\log n)$ time (respectively, I/O complexity), where t is the time complexity (respectively, I/O complexity) of llce and random access queries. With z-fast tries [5, 9] built on the text prefixes $\{\mathcal{T}[1,i-1]\ :\ i\in\mathrm{PDA}_\pi\wedge\mathcal{T}[i]=c\}$ for each $c\in\Sigma$, we obtain the following:

▶ **Lemma 28.** *Let $\mathcal{T}\in\Sigma^n$ be a text and $\pi:[n]\to[n]$ be an order-preserving permutation. Suppose we have access to an oracle supporting fingerprinting queries on $\mathcal{T}$ in $O(h)$ time (respectively, I/O complexity) and extraction of ℓ contiguous text characters in $e(\ell)$ time (respectively, I/O complexity). Then, Algorithm 1 runs in $O(d\cdot h\log m+e(m))$ time (respectively, I/O complexity), where d is the node depth of* locus(P) *in the suffix tree of $\mathcal{T}$.*

▶ **Lemma 29.** *Algorithm 1 is correct and complete.*

Proof (Sketch). The invariant maintained by the `while` loop is: if P occurs in $\mathcal{T}$, then $P[1,j-1]=\mathcal{T}[i-j+1,i-1]$ is the occurrence of $P[1,j-1]$ minimizing $\pi(i-j+1)$. ◀

2.2.2 Locating the secondary occurrences

We now describe how to locate the secondary occurrences. This subsection is devoted to proving the following lemma.

▶ **Lemma 30.** *Let $\mathcal{T}\in\Sigma^n$ be a text and $\pi:[n]\to[n]$ be an order-preserving permutation. Let $P\in\Sigma^m$ occur in $\mathcal{T}$. There exists a data structure that takes $O(|\mathrm{PDA}_\pi|)$ words of space and that, given the primary occurrence of P, finds all the $occ-1$ secondary occurrences in $O(occ\cdot\log|\mathrm{PDA}_\pi|)\subseteq O(occ\cdot\log n)$ time (equivalently, I/O complexity).*

We start by covering $\mathcal{T}$ with $|\mathrm{PDA}_\pi|$ (possibly overlapping) *phrases* (that is, substrings of $\mathcal{T}$), as follows:

▶ **Definition 31** (Phrase cover of $\mathcal{T}$)**.**
- (Type-1 phrases) *we associate phrase $\mathcal{T}[i]$ (one character) to each position $i\in[n]$ such that* $\mathrm{LPF}_\pi[i]=0$.
- (Type-2 phrases) *We associate phrase $\mathcal{T}[i,i+\mathrm{LPF}_\pi[i]-1]$ to each irreducible* LPF_π *position i such that* $\mathrm{LPF}_\pi[i]>0$.

Observe that there are at most $|\mathrm{PDA}_\pi|$ type-2 phrases. By definition, our cover of $\mathcal{T}$ into phrases satisfies the following properties:

▶ Remark 32. Any pattern occurrence $\mathcal{T}[i',i'+m-1]$ crossing a type-1 phrase $\mathcal{T}[i]$ (i.e. $i\in[i',i'+m-1]$) is primary. To see this observe that, due to $\mathrm{LPF}_\pi[i]=0$, we know that $c=\mathcal{T}[i]$ is the occurrence of c minimizing $\pi(i)$, hence by the order-preserving property of π, $\mathcal{T}[i',i'+m-1]=P$ is the occurrence of P minimizing $\pi(i')$.

▶ Remark 33. For each secondary occurrence $\mathcal{T}[i',i'+m-1]$, there exists a type-2 phrase $\mathcal{T}[i,i+\mathrm{LPF}_\pi[i]-1]$ containing it, i.e. $[i',i'+m-1]\subseteq[i,i+\mathrm{LPF}_\pi[i]-1]$. This immediately follows from the definition of secondary occurrence.

Observe that each type-2 phrase is copied from another text position with a smaller π. We formalize this fact as follows:

▶ **Definition 34** (Phrase source)**.** *Let $\mathcal{T}[i,i+\mathrm{LPF}_\pi[i]-1]$ be a type-2 phrase. We define* $\mathrm{SRC}[i]=j$ *to be any position j with $\pi(j)<\pi(i)$ and $\mathcal{T}[i,i+\mathrm{LPF}_\pi[i]-1]=\mathcal{T}[j,j+\mathrm{LPF}_\pi[i]-1]$.*

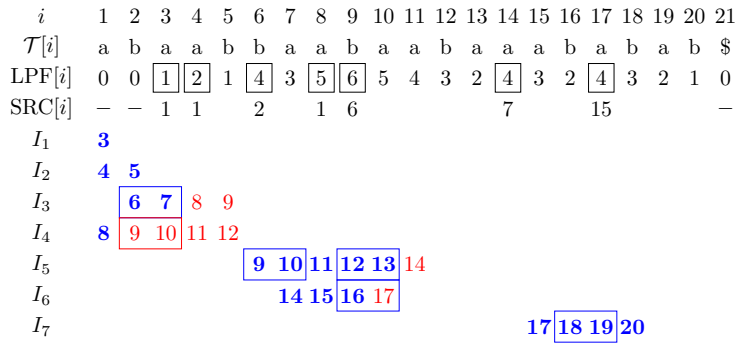

Figure 2 Consider this particular string $\mathcal{T}$ and take $\pi = id$ to be the identity function. Then, $\mathrm{LPF}_\pi = \mathrm{LPF}$ is the classic Longest Previous Factor array. In the third line, we put in boxes the irreducible LPF positions i such that $\mathrm{LPF}[i] > 0$; these are the beginnings of our phrases (7 phrases in total). In Lines $I_1 - I_7$ in the figure, we show each type-2 phrase's source as an interval containing the phrase's positions, colored according to its reducible prefix (in blue) and the remaining suffix (in red). Boxes show how the occurrences of string *ba* intersect the phrases' sources; the box is blue when the occurrence intersects (the source of) the phrase's reducible prefix, red otherwise. Only occurrences in a blue box trigger the location of further secondary occurrences.

We also use the fact that each phrase can be split into a prefix containing strictly decreasing LPF_π values and the remaining suffix. This will play a crucial role in our locating algorithm, as it will allow us to report each secondary occurrence exactly once.

▶ **Definition 35** (Reducible prefix). *Let $\mathcal{T}[i, i+\mathrm{LPF}_\pi[i]-1]$ be a type-2 phrase. The* reducible prefix *of $\mathcal{T}[i, i+\mathrm{LPF}_\pi[i]-1]$ is its longest prefix $\mathcal{T}[i, i+\ell_i-1]$, with $1 \le \ell_i \le \mathrm{LPF}_\pi[i]$, such that $\mathrm{LPF}_\pi[j] = \mathrm{LPF}_\pi[j-1]-1$ for each $j \in [i+1, i+\ell_i-1]$.*

▶ Remark 36. All LPF_π positions in $[i+1, i+\ell_i-1]$ are reducible (that is, not irreducible).

The idea to locate secondary occurrences, is to associate every type-2 phrase $\mathcal{T}[i, i+\mathrm{LPF}_\pi[i]-1]$ with a 2-dimensional rectangle whose coordinates reflect the source of the whole phrase and of its reducible prefix:

▶ **Definition 37** (Rectangle associated with a type-2 phrase). *Let $\mathcal{T}[i, i+\mathrm{LPF}_\pi[i]-1]$ be a type-2 phrase, $s_i = \mathrm{SRC}[i]$ be its source, and ℓ_i be the length of its reducible prefix. We associate with $\mathcal{T}[i, i+\mathrm{LPF}_\pi[i]-1]$ the 2-dimensional rectangle $[s_i, s_i+\ell_i-1] \times [s_i, s_i+\mathrm{LPF}_\pi[i]-1] \subseteq [n]^2$, and label it with position i.*

▶ **Example 38.** In the running example of Figure 2, consider the phrase $\mathcal{T}[9, 14]$, let $s_9 = \mathrm{SRC}[9] = 6$ be its source, and $\ell_9 = 5$ be the length of its reducible prefix. This phrase is associated with rectangle $[s_9, s_9+\ell_9-1] \times [s_9, s_9+\mathrm{LPF}[9]-1] = [6, 10] \times [6, 11]$.

Our locating algorithm relies on the following well-known data structure result on the *orthogonal point enclosure problem*:

▶ **Lemma 39** (Orthogonal point enclosure, [11, Theorem 6]). *Let $\mathcal{R}$ be a collection of axis-parallel two-dimensional rectangles in $[n]^2$. There exists an $O(|\mathcal{R}|)$-space data structure supporting the following query: given a point (x, y), find all rectangles $[a, b] \times [c, d] \in \mathcal{R}$ containing (x, y), i.e. such that $x \in [a, b]$ and $y \in [c, d]$. The query is answered in $O(\log |\mathcal{R}| + k)$ time, where k is the number of returned rectangles.*

Let $\mathcal{R}$ be the set of rectangles in Definition 37. As noted above, $|\mathcal{R}| \le |\mathrm{PDA}_\pi|$. We build the data structure of Lemma 39 on $\mathcal{R}$. The structure uses $O(|\mathrm{PDA}_\pi|)$ words of space and answers orthogonal point enclosure queries in $O(\log |\mathrm{PDA}_\pi| + k) \subseteq O(\log n + k)$ time.

Locating algorithm. Let $\mathcal{T}[j, j+m-1] = P$ be the primary occurrence of P, found with Algorithm 1. To locate the secondary occurrences, initialize a stack $Q \leftarrow \{j\}$. While Q is not empty:

1. Pop an element x from Q and report pattern occurrence x.
2. Locate all rectangles in $\mathcal{R}$ containing point $(x, x+m-1)$. For each retrieved rectangle $[s_i, s_i+\ell_i-1] \times [s_i, s_i+\mathrm{LPF}_\pi[i]-1]$ labeled with position i, push $i+x-s_i$ in Q.

Analysis. Due to space constraints, we defer the detailed analysis to the full version [4]. The time complexity can be derived from the observation that no occurrence is pushed more than once into the stack. Combining this with Lemmas 28, 29, and 30 we obtain:

▶ **Theorem 40.** *Let $\mathcal{T} \in \Sigma^n$ be a text and $\pi : [n] \to [n]$ be an order-preserving permutation. Suppose we have access to an oracle supporting fingerprinting queries on $\mathcal{T}$ in $O(h)$ time (respectively, I/O complexity) and the extraction of ℓ contiguous characters of $\mathcal{T}$ in $e(\ell)$ time (respectively, I/O complexity). Then, there exists a data structure taking $O(|\mathrm{PDA}_\pi|)$ words of space on top of the oracle and able to report the occ occurrences of any pattern $P \in \Sigma^m$ in $O(d \cdot h \log m + e(m) + occ \cdot \log |\mathrm{PDA}_\pi|) \subseteq O(d \cdot h \log m + e(m) + occ \cdot \log n)$ time (respectively, I/O complexity), where d is the node depth of $\mathrm{locus}(P)$ in the suffix tree of $\mathcal{T}$.*

For example, using the text oracle of Prezza [42], the I/O complexity of Theorem 40 becomes $O(d \log m + m/B + occ \cdot \log n)$. The resulting index uses $O(|\mathrm{PDA}_\pi|)$ memory words on top of the oracle of $n \log \sigma + O(\log n)$ bits.

2.3 Colexicographic rank (`st-colex`): smaller-space I/O-efficient pattern matching

We now move to the particular case $\pi = \mathrm{IPA}$, for which the algorithm described in the previous section can be simplified, leading to a very space-efficient and fast implementation.

Figure 3 depicts the STPD obtained by choosing $\pi = \mathrm{IPA}$ to be the Inverse Prefix Array. We denote with $\texttt{st-colex}^- = \mathrm{PDA}_\pi$ the path decomposition array associated with this permutation π. Similarly, $\texttt{st-colex}^+$ denotes the path decomposition array associated with the dual permutation $\bar{\pi}(i) = n - \mathrm{IPA}[i] + 1$. The following properties hold:

▶ **Lemma 41.** *Let $\mathcal{T} \in \Sigma^n$ be a text. The permutations $\pi, \bar{\pi}$ defined as $\pi(i) = \mathrm{IPA}[i]$ and $\bar{\pi}(i) = n - \mathrm{IPA}[i] + 1$ for $i \in [n]$ are order-preserving for $\mathcal{T}$. Furthermore, $|\texttt{st-colex}^-| \leq \bar{r}$ and $|\texttt{st-colex}^+| \leq \bar{r}$ hold.*

We now show how to locate the pattern's occurrences with $\texttt{st-colex}^-$. In this section, we assume a text oracle supporting the extraction of ℓ contiguous text characters with $O(1 + \ell/B)$ I/O complexity.

2.3.1 Finding the primary occurrence

For the particular order-preserving permutation $\pi = \mathrm{IPA}$ used in this section, the primary occurrence $P = \mathcal{T}[i, j]$ of P is the one for which the text prefix $\mathcal{T}[1, j]$ is the colexicographically-smallest being suffixed by P.

We use Algorithm 1 to locate the primary occurrence. Since $\pi = \mathrm{IPA}$ (colexicographic rank) and $\mathrm{PDA}_\pi = \texttt{st-colex}^-$ is sorted colexicographically, observe that $\pi(\mathrm{PDA}_\pi) = \pi(\texttt{st-colex}^-)$ is increasing. This means that, in Line 6 of Algorithm 1, it always holds $\arg\min_{i' \in [b,e]}\{\pi(\mathrm{PDA}_\pi[i'])\} = b$ and therefore we do not need a Range Minimum Data structure over $\pi(\texttt{st-colex}^-)$.

By implementing sufsearch() by binary search and random access, we obtain:

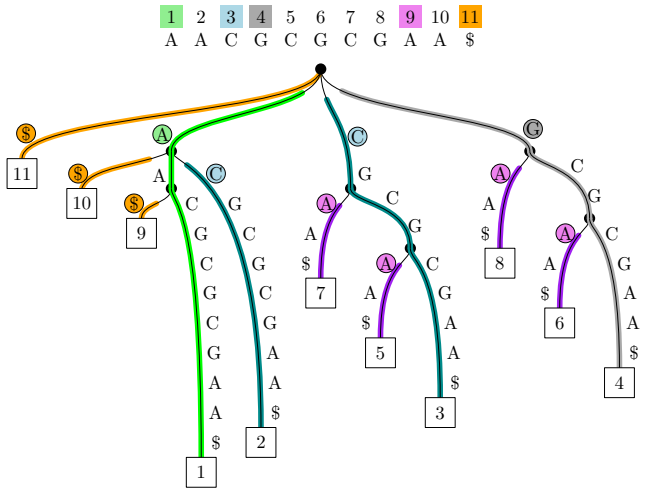

Figure 3 Example of the STPD obtained by taking $\pi = \text{IPA} = [2, 3, 6, 9, 7, 10, 8, 11, 5, 4, 1]$. Paths $\mathcal{T}[j, n]$ are colored according to the color of $j \in \texttt{st-colex}^-$.

▶ **Lemma 42.** *Let $\mathcal{T} \in \Sigma^n$ be a text, and fix $\pi = \text{IPA}$. Suppose we have a text oracle supporting extraction of ℓ contiguous characters with $O(1+\ell/B)$ I/O complexity. Algorithm 1 requires just array* $\texttt{st-colex}^-$*, fitting in $|\texttt{st-colex}^-| \leq \bar{r}$ words, on top of the text oracle and locates the primary occurrence of $P \in \Sigma^m$ with $O(d \log |\texttt{st-colex}^-| \cdot (1 + m/B)) \subseteq O(d \log \bar{r} \cdot (1 + m/B))$ I/O complexity, where d is the node depth of $\text{locus}(P)$ in the suffix tree of $\mathcal{T}$.*

2.3.2 Locating the secondary occurrences

Due to space constraints, here we describe only the outlines of how to locate secondary occurrences. The full description can be found in the full version [4]. Briefly speaking, we can exploit the locating mechanism of the r-index [21], with minor modifications. Differently from the toehold lemma of the r-index, Algorithm 1 only allows to compute $\text{PA}[b]$ (not occ). If we use both $\texttt{st-colex}^-$ and $\texttt{st-colex}^+$, we can apply $\bar{\phi}$ repeatedly starting from the primary occurrence for $\texttt{st-colex}^-$ until we reach that for $\texttt{st-colex}^+$, this will be exactly $occ - 1$ applications of $\bar{\phi}$. On the other hand, we can compute the number occurrences using just $\texttt{st-colex}^-$. We can find $i \geq 1$ such that $(i-1)\lceil m/B \rceil \leq occ < i\lceil m/B \rceil$ by applying $\bar{\phi}$ recursively and comparing the text with the pattern for every $\lceil m/B \rceil$ applications. Then, we perform binary search to find occ. As far as the I/O complexity is concerned, the cost of the first phase is $O((1 + occ/\lceil m/B \rceil) \cdot (1 + m/B)) \subseteq O(1 + occ + m/B)$, and that of the second binary search phase is $O((1 + m/B) \cdot \log \lceil m/B \rceil) \subseteq O((1 + m/B)\lceil \log(1 + m/B) \rceil)$.

2.3.3 Putting everything together

Locating the $occ - 1$ secondary occurrences of P costs $O((1 + m/B)\lceil \log(1 + m/B) \rceil + occ)$ I/O complexity and requires $O(1 + m/B)$ words of memory on top of the index at query time. Combining this with Lemma 42, we obtain:

▶ **Theorem 43.** *Let $\mathcal{T} \in \Sigma^n$ be a text. Assume we have access to a text oracle supporting the extraction of ℓ contiguous characters of $\mathcal{T}$ with $O(1 + \ell/B)$ I/O complexity. Our data structure locates all the occ occurrences of $P \in \Sigma^m$ with I/O complexity*

$$O\left((d \log \bar{r} + \log(1 + m/B)) \cdot (1 + m/B) + occ + \log \log(n/\bar{r})\right)$$

and uses $O(\bar{r})$ memory words on top of the oracle. At query time, $O(m/B)$ further memory words are used.

Observe that the extra $O(m/B)$ memory words of space are negligible; most compressed indexes explicitly store (and/or receive as input) the full pattern at query time anyways, in $O(m)$ words. The remaining $O(\bar{r})$ words of space are spent to store array $\texttt{st-colex}^-$ (at most $|\texttt{st-colex}^-| \leq \bar{r}$ words) and Nishimoto and Tabei's data structure [41] storing function $\bar{\phi}$ (less than $2\bar{r}$ words using the optimized implementation of [7]).

2.4 Identity (st-pos): leftmost pattern occurrence

Another notable STPD is obtained by using the identity function $\pi(i) = i$, trivially satisfying the order-preserving property of Definition 5. We call the corresponding path decomposition array $\texttt{st-pos}^-$. Similarly, $\texttt{st-pos}^+$ is the path decomposition array associated with the dual permutation $\bar{\pi}(i) = n - i + 1$.

The size of $\texttt{st-pos}^-$ is equal to the number of irreducible values in the Longest Previous Factor array LPF, a new interesting repetitiveness measure that, to the best of our knowledge, has never been studied before. While we could not prove a theoretical bound for $|\texttt{st-pos}^-|$ and $|\texttt{st-pos}^+|$ in term of known repetitiveness measures, below we show that these measures are worst-case optimal, meaning that for every $p \geq 1$ we exhibit a family of strings with $p = \Theta(|\texttt{st-pos}^-|)$ requiring $O(p)$ words to be stored in the worst case.

▶ **Theorem 44.** *For any integers $n \geq 1$ and $p \in [n]$, there exists a family $\mathcal{F}$ of $\binom{n}{p}$ strings over an alphabet of size $p+1$ such that for every string $\mathcal{S} \in \mathcal{F}$ it holds that $|\texttt{st-pos}^-(\mathcal{S})| = p+1$ and $|\mathcal{S}| \leq np$. In particular, no compressor can compress every individual $\mathcal{S} \in \mathcal{F}$ in asymptotically less than $\log_2 \binom{n}{p} = p\log(n/p) + \Theta(p)$ bits. The same holds for $|\texttt{st-pos}^+|$.*

Proof (Sketch). Consider any integer set $\{x_1 > x_2 > \cdots > x_p\} \subseteq [n]$. Encode the set as the string $\mathcal{S} = 0^{x_1} \cdot 1 \cdot 0^{x_2} \cdot 2 \cdots 0^{x_p} \cdot p$. Then $|\texttt{st-pos}^-(\mathcal{S})| = p + 1$. Fix $n \geq 1$ and $p \in [n]$. Then, the family $\mathcal{F}_{n,p}$ of such strings contains $\binom{n}{p}$ elements. The proof for $\texttt{st-pos}^+$ is symmetric. ◀

Theorem 44 proves that $p = \Theta(|\texttt{st-pos}^-|)$ words of space are essentially worst-case optimal as a function of n and p to compress the string (the same holds for $p = |\texttt{st-pos}^+|$). This result is similar to that obtained in [28] for the repetitiveness measure δ. Since by Theorem 14, $O(|\texttt{st-pos}^-| \log |\mathcal{S}|)$ bits (in the theorem above, $O(p\log(np)) = O(p \log n)$ bits) are also sufficient to compress $\mathcal{S}$ (the same holds for $|\texttt{st-pos}^+|$), this indicates that $|\texttt{st-pos}^-|$ and $|\texttt{st-pos}^+|$ are two meaningful repetitiveness measures.

We conclude this section with the following discussion. More detailed explanations can be found in the full version [4].

- **Application to taxonomic classification of DNA fragments.** The taxonomic classifiers Kraken [50] and Kraken 2 [49] index the genomes in a given phylogenetic tree such that, given a pattern, they can map each substring of fixed length k to the root of the smallest subtree of the phylogenetic tree containing all the genomes containing that substring. A bottleneck is finding the leftmost and rightmost such genomes, which can now be efficiently done using our $\texttt{st-pos}^-$ and $\texttt{st-pos}^+$.
- **Relation to Ukkonen's suffix tree construction algorithm.** Ukkonen's algorithm [47] builds the suffix tree by inserting the suffixes from the longest ($\mathcal{T}[1, n]$) to the shortest ($\mathcal{T}[n, n]$). Instead of inserting exactly one suffix (leaf) per iteration like McCreight's algorithm [36], it inserts a variable number (possibly none) of leaf nodes per iteration. $|\texttt{st-pos}^-|$ can be interpreted as the number of iterations that insert at least one leaf.

- **Relation with PPM*.** *Prediction by Partial Matching with unbounded context* (*PPM**) is a popular and effective compression algorithm that encodes each text's symbol $\mathcal{T}[j] = c$ according to the probability distribution of characters following the longest context $\mathcal{T}[i', j-1] = \alpha$ preceding it that already occurred before in the text followed by c. In the simple version of PPM* [15], when all previous occurrences of α are never followed by c, the algorithm outputs a special *escape* symbol $\perp$ followed by character c. Then, $|\texttt{st-pos}^-|$ is upper-bounded by the number of occurrences of $\perp$ output by this algorithm.

3 Preliminary experimental results

Comparison with other repetitiveness measures. Table 1 compares the sizes (number of samples) of the three notable suffix tree path decompositions discussed in this paper with existing repetitiveness measures on three repetitive datasets downloaded from `https://pizzachili.dcc.uchile.cl/repcorpus.html`. Interestingly, on these datasets our new repetitiveness measures are consistently smaller than the smallest suffixient set size (χ) and the number of runs in the Burrows-Wheeler transform of the text (r) and its reverse ($\bar{r}$).

Table 1 Experimental evaluation of Path Decomposition Array sizes on repetitive strings. All numbers are in millions.

Data	n	$\lvert\texttt{st-lex}^-\rvert$	$\lvert\texttt{st-colex}^-\rvert$	$\lvert\texttt{st-pos}^-\rvert$	χ	r	$\bar{r}$
Influenza	154.808	1.815	1.805	1.928	2.225	3.022	3.018
Cere	461.286	7.455	7.455	8.954	9.921	11.574	11.575
Escherichia	112.689	9.817	9.825	11.677	13.119	15.044	15.045

Compressed index. We implemented an optimized prototype of our $\texttt{st-colex}^-$ index (Section 2.3) in C++ to evaluate its performance against the state-of-the-art: the r`-index` [21], `move-`r [7], the `suffixient array` [10] and the `suffix array`. We used an ad-hoc implementation of Relative Lempel Ziv [30] for the random access oracle in our index. The evaluation focused on space usage and the efficiency of locating one or all pattern occurrences in repetitive genomic data. See the full version of this paper [4] for the detailed implementation choices and experimental results.

Experiments were conducted on an Intel(R) Xeon(R) W-2245 CPU @ 3.90GHz workstation with 8 cores and 128 gigabytes of RAM running Ubuntu 18.04 LTS 64-bit. The input text consisted of 19 variants of Human chromosome 19 (total length $n \approx 10^9$) downloaded from `https://github.com/koeppl/phoni`. We extracted 10^5 patterns of variable length ($m = 15, 150, 1500$) from random locations in the text. For each pattern, we measured the resources (peak memory and running time) used by all the indexes while finding one pattern occurrence and locating all pattern occurrences.

In our experiments, our `st-colex` index was always smaller than the r`-index`, about four times smaller than `move-`r, and orders of magnitude smaller than the `suffix array`. It was moreover always slightly smaller than the `suffixient array`. Our index was always orders of magnitude faster than the r`-index` on the task of locating *one pattern occurrence*, and even faster than the `suffix array`. It was also one order of magnitude faster than `move-`r on long patterns ($m \geq 150$), and slightly faster on small patterns. As far as locating *all pattern occurrences* was concerned, we observed similar performance as discussed above for long patterns ($m \geq 150$), due to the small number of reported occurrences per pattern ($occ \leq 17.2$ on average on those pattern lengths). On shorter patterns ($m = 15$), the number

of reported pattern occurrences was much larger ($occ = 6911.6$ on average), and, as expected, our index was slightly slower than `move-`r, slower than the `suffix array` by an order of magnitude, and slightly faster than the r`-index`.

References

1 Omar Ahmed, Massimiliano Rossi, Sam Kovaka, Michael C Schatz, Travis Gagie, Christina Boucher, and Ben Langmead. Pan-genomic matching statistics for targeted nanopore sequencing. *iScience*, 24(6), 2021.

2 Omar Y Ahmed, Massimiliano Rossi, Travis Gagie, Christina Boucher, and Ben Langmead. Spumoni 2: Improved classification using a pangenome index of minimizer digests. *Genome Biology*, 24(1):122, 2023.

3 Ruben Becker, Davide Cenzato, Travis Gagie, Ragnar Groot Koerkamp, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza. STPD-index. Software, Funded by the European Union (ERC, REGINDEX, 101039208), swhId: `swh:1:dir:d4f76a2be744c622f98d403443e3f4507ae9b536` (visited on 2026-06-19). URL: `https://github.com/regindex/STPD-index`, `doi:10.4230/artifacts.26761`.

4 Ruben Becker, Davide Cenzato, Travis Gagie, Sung-Hwan Kim, Ragnar Groot Koerkamp, Giovanni Manzini, and Nicola Prezza. Compressing suffix trees by path decompositions. *arXiv preprint arXiv.2506.14734*, 2025. `doi:10.48550/arXiv.2506.14734`.

5 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 785–794. SIAM, 2009. `doi:10.1137/1.9781611973068.86`.

6 Djamal Belazzougui, Manuel Cáceres, Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Gonzalo Navarro, Alberto Ordóñez Pereira, Simon J. Puglisi, and Yasuo Tabei. Block trees. *J. Comput. Syst. Sci.*, 117:1–22, 2021. `doi:10.1016/j.jcss.2020.11.002`.

7 Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: Optimizing the r-index. In Leo Liberti, editor, *22nd International Symposium on Experimental Algorithms, SEA 2024, Vienna, Austria, July 23-26, 2024*, volume 301 of *LIPIcs*, pages 1:1–1:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPIcs.SEA.2024.1`.

8 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015. `doi:10.1137/130936889`.

9 Paolo Boldi and Sebastiano Vigna. Kings, name days, lazy servants and magic. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms, FUN 2018, La Maddalena, Italy, June 13-15, 2018*, volume 100 of *LIPIcs*, pages 10:1–10:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.FUN.2018.10`.

10 Davide Cenzato, Lore Depuydt, Travis Gagie, Sung-Hwan Kim, Giovanni Manzini, Francisco Olivares, and Nicola Prezza. Suffixient arrays: A new efficient suffix array compression technique. *arXiv preprint*, 2024. `arXiv:2407.18753`.

11 Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986. `doi:10.1137/0215051`.

12 Yu-Feng Chien, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Geometric burrows-wheeler transform: Linking range searching and text indexing. In *Data Compression Conference (DCC 2008)*, pages 252–261, 2008. `doi:10.1109/DCC.2008.67`.

13 Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17(1):8:1–8:39, 2021. `doi:10.1145/3426473`.

14 Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, editors, *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, volume 7608 of *Lecture Notes in Computer Science*, pages 180–192. Springer, 2012. doi:10.1007/978-3-642-34109-0_19.

15 John G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Comput. J.*, 40(2/3):67–75, 1997.

16 Davide Cozzi, Massimiliano Rossi, Simone Rubinacci, Travis Gagie, Dominik Köppl, Christina Boucher, and Paola Bonizzoni. μ- PBWT: a lightweight r-indexing of the PBWT for storing and querying UK biobank data. *Bioinform.*, 39(9), 2023. doi:10.1093/bioinformatics/btad552.

17 Vinicius T. V. Date and Leandro M. Zatesko. On the near-tightness of $\chi \leq 2r$: A general σ-ary construction and a binary case via LFSRs. *arXiv preprint*, 2025. arXiv:2512.20598.

18 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, Redondo Beach, California, USA, November 12-14, 2000*, pages 390–398. IEEE Computer Society, 2000. doi:10.1109/SFCS.2000.892127.

19 Paolo Ferragina and Rossano Venturini. Compressed cache-oblivious string b-tree. *ACM Trans. Algorithms*, 12(4), August 2016. doi:10.1145/2903141.

20 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1459–1477. SIAM, 2018. doi:10.1137/1.9781611975031.96.

21 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.

22 Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. In William B. Frakes and Ricardo A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 66–82. Prentice-Hall, 1992.

23 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 397–406. ACM, 2000. doi:10.1145/335305.335351.

24 Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In Gregory Kucherov and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings*, volume 5577 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2009. doi:10.1007/978-3-642-02441-2_17.

25 Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 1002–1013. IEEE, 2020. doi:10.1109/FOCS46700.2020.00097.

26 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.

27 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: String attractors. In Ilias Diakonikolas, David Kempe, and Monika Henzinger, editors, *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.

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. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

30 Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In Edgar Chávez and Stefano Lonardi, editors, *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*, pages 201–206. Springer, 2010. doi:10.1007/978-3-642-16321-0_20.

31 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976. doi:10.1109/TIT.1976.1055501.

32 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.

33 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.

34 Udi Manber and Eugene W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993. doi:10.1137/0222058.

35 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 319–327. SIAM, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

36 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

37 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.

38 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.

39 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.

40 Gonzalo Navarro, Giuseppe Romana, and Cristian Urbina. Smallest suffixient sets as a repetitiveness measure. *arXiv preprint arXiv:2506.05638*, 2025. doi:10.48550/arXiv.2506.05638.

41 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on BWT-runs compressed indexes. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, Glasgow, Scotland (Virtual Conference), July 12-16, 2021*, volume 198 of *LIPIcs*, pages 101:1–101:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.101.

42 Nicola Prezza. Optimal substring equality queries with applications to sparse text indexing. *ACM Trans. Algorithms*, 17(1):7:1–7:23, 2021. doi:10.1145/3426870.

43 Simon J. Puglisi and Bella Zhukova. Relative Lempel-Ziv compression of suffix arrays. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 89–96. Springer, 2020. doi:10.1007/978-3-030-59212-7_7.

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 Vikram Shivakumar, Omar Y. Ahmed, Sam Kovaka, Mohsen Zakeri, and Ben Langmead. Sigmoni: Classification of nanopore signal with a compressed pangenome index. *Bioinform.*, 40(Supplement_1):i287–i296, 2024. doi:10.1093/bioinformatics/btae213.

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

48 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.

49 Derrick E Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with Kraken 2. *Genome Biology*, 20:1–13, 2019.

50 Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15:1–12, 2014.

51 Mohsen Zakeri, Nathaniel K Brown, Omar Y Ahmed, Travis Gagie, and Ben Langmead. Movi: a fast and cache-efficient full-text pangenome index. *iScience*, 27(12), 2024.