# Compressing Suffix Trees by Path Decompositions

Ruben Becker[1], Davide Cenzato[1], Travis Gagie[2], Ragnar Groot Koerkamp[3], Sung-Hwan Kim[1], Giovanni Manzini[4], and Nicola Prezza[1] ⋆

[1] Ca' Foscari University of Venice, Italy,
{rubensimon.becker, davide.cenzato, sunghwan.kim, nicola.prezza}@unive.it
[2] Dalhousie University, Halifax, Nova Scotia, Canada Travis.Gagie@dal.ca
[3] ETH Zurich, Switzerland ragnar.grootkoerkamp@gmail.com
[4] University of Pisa, Italy giovanni.manzini@unipi.it

**Abstract.** The suffix tree — the path-compressed trie of the string's suffixes — 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, research in the subsequent 50 years showed first how to reduce its size by a multiplicative constant through Suffix Arrays, and then how to support suffix tree functionality in space proportional to the size of the *compressed* string, see Grossi and Vitter (STOC 2000). Modern compressed indices such as the $r$-index of Gagie et al. (SODA 2018, JACM 2020) support suffix tree operations in $O(r\log(n/r))$ space and pattern matching queries in $O(r)$ space, where the number $r$ of runs in the Burrows-Wheeler transform of the indexed string $\mathcal{T}$ (of length $n$) is a strong and universal compressibility measure capturing the string's repetitiveness. These advances, however, came with a price: such indices are orders of magnitude slower than classic suffix trees and Suffix Arrays due to poor cache locality at query time.

In this paper, we solve the long-standing problem of designing small *and* I/O-efficient compressed indexes. Classic suffix trees represent unary suffix trie paths with pairs of pointers to $\mathcal{T}$, which must be available in the form of some random access oracle at query time. In our approach, instead, we (i) sort the suffix tree's leaves according to a more general priority function $\pi$ (generalizing suffix sorting), (ii) we build a *suffix tree path decomposition* prioritizing the smallest (according to $\pi$) paths in such an order, and (iii) we path-compress the decomposition's paths as pointers to a small subset of the string's suffixes. At this point, we show that the colexicographically-sorted array of those pointers can be used to obtain a new elegant, simple, and remarkably I/O-efficient compressed suffix tree. For instance, by taking $\pi$ to be the lexicographic rank of $\mathcal{T}$'s suffixes, we can compress the suffix tree topology in $O(r)$ space on top of a $n\log\sigma + O(\log n)$-bits text representation while essentially matching the query I/O complexity of Weiner and McCreight's suffix tree. Another solution is obtained by taking $\pi$ to be the colexicographic rank of $\mathcal{T}$'s prefixes and using a fully-compressed random access oracle. The resulting self-index allows us to locate all occurrences of a given query pattern in *less space* and *orders of magnitude faster* than the $r$-index. Due to the considerable interest that the $r$-index has gained since its introduction, we expect our new solution to have a profound impact in the field. More in general, we provide a mechanism for locating all pattern occurrences for a wide class of functions $\pi$, each yielding a new reachable repetitiveness measure (the number of samples in the corresponding suffix tree path decomposition).

⋆ *Ruben Becker, Davide Cenzato, Sung-Hwan Kim, 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.

# 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 [47] and revisited (in their modern form) in 1976 by McCreight [35] 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 $\sigma$ 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 [20, 33, 34], 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) [21], FM-index [15], run-length compressed Suffix Array [32], run-length FM-index [31], $r$-index [19], Lempel-Ziv-based [27] and grammar-based [12] indexes (and their variants), and the more recent $\delta$-SA [24]. See the survey of Navarro [38] 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 [18,19] and its improvements [5,40,50] stood out for its 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 [37] covers the subject in detail), culminating in recent breakthroughs [23,25] 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 [30], normalized substring complexity [26], 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, 14, 43, 44, 45].

## 1.1 Are compressed data structures just a theoretical tool?

Modern compressed indexes such as the $\delta$-SA of Kempa and Kociumaka [24] 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 [11]), but rely on complex data structures that, again, make them orders of magnitude slower than simple suffix trees in practice. The $r$-index (in its improved version [40]) 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 (apart from the problem of searching the bit-packed query pattern — partially solved in [19]), 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 [15]) triggers I/O operations (that is, likely cache misses). While this was later mitigated by the *move* structure of Nishimoto and Tabei [40], that solution still triggers $O(m)$ cache misses. This does not happen with the suffix tree:

*Remark 1.* The suffix tree of Weiner [47] and McCreight [35] 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 (i.e. each suffix tree edge is represented as a pair of pointers to the text) makes it possible to compare (a substring of) the pattern with the label of an edge of length $\ell$ with $O(\ell/B + 1)$ I/O complexity. As a result, each of the $d$ edge traversals triggers a I/O operation in the worst case[5]. 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 Section 4.2). 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. While Remark 1 reflects the original suffix tree design [35, 47] (locating pattern occurrences by subtree navigation), it is worth noticing that augmenting the suffix tree with the Suffix Array SA [20, 34], the *occ* term gets reduced to $occ/B$ since the *occ* pattern occurrences occur contiguously in SA. In this paper, we stick with the original suffix tree design of Remark 1 since we are mostly interested in matching the $m/B$ term, which becomes dominant as the pattern length increases asymptotically (*occ*, on the other hand, does not increase when appending new characters to a given query pattern).

The performance gap in the I/O model between compressed indexes and the suffix tree shows up spectacularly in practice: a simple experiment (see Section 4.2) 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 for all the existing compressed indexes using a space close to that of the $r$-index. This triggers a natural question:

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

### 1.2 Contributions and overview of our techniques

The main contribution of our paper is to answer the above question positively. We show that compressed indexes can be *faster* (and much smaller) than classic (uncompressed) indexes, while at the same time being *smaller* and *orders of magnitude faster* than modern compressed indexes such as the $r$-index. Our main theoretical result states that we can compress the suffix tree topology to $O(r)$ words on top of a (possibly, compressed) text oracle while still being able to answer most suffix tree navigation queries efficiently:

[5] While this is true in general, a more accurate choice of path compression pointers (based on heavy paths) can reduce the term $d$ to $\log n$ in the worst case. Typical suffix tree construction algorithms, however, do not provide such a guarantee (and, in practice, an arbitrary choice of the pointers leads to good performance anyways).

**Theorem 1.** *Let $\mathcal{T}$ be a text of length $n$ over an alphabet of size $\sigma$. 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 lexicographically-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 lexicographic successor of $a$ 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.*

*If $t$ is the I/O complexity of* lce*/random access queries and $h$ is the I/O complexity of fingerprinting, then all the above statements are still valid by replacing "$O(\dots)$ time" with "$O(\dots)$ I/O complexity".*

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 [41], supporting lce with $O(t) = O(\log n)$ I/O complexity, fingerprinting with $O(h) = O(1)$ I/O complexity, and extraction of $\ell$ contiguous characters with $O(1 + \ell/B)$ I/O complexity on polynomial alphabets, we obtain the following corollary:

**Corollary 1.** *Let $\mathcal{T}$ be a text of length $n$ over alphabet of size $\sigma \leq n^{O(1)}$. The topology of $\mathcal{T}$'s suffix tree can be compressed in $O(r)$ words on top of a text representation [41] 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 [19]). Furthermore, we show that the term $d \log n$ can be replaced by $d \log m$ with a different technique (Theorem 3; this is important since in typical applications, $m \ll n$ holds). Importantly, our result does not rely on a particular text oracle and can benefit from the vast amount of literature existing on *compressed* random access and lce oracles [4, 6, 28, 29] (to cite a few; see also the survey of Navarro [37]). 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.

Differently from the long line of research on *compressed self-indexes* [15, 21] (integrating index and text in the same compressed data structure), we obtain this result from separating the indexing functionality (suffix tree topology) from the text oracle. This is exactly what happens in classic suffix trees and has the additional advantage of allowing us to play with the text oracle (e.g., by choosing a different compression method depending on the dataset at hand).

**Related work.** In their paper introducing the $r$-index, Gagie et al. [19] showed that full suffix tree functionality can be supported in $O(r\log(n/r))$ words of space and logarithmic time for most operations. By augmenting their representation with an I/O-efficient text oracle (e.g., a plain text representation), one obtains pattern matching queries with a similar I/O complexity of our Corollary 1, which however uses just $O(r)$ words on top of the text oracle. Additionally, the structure of Gagie et al. relies on a complex machinery built on top of a grammar-compressed Suffix Array and, to the best of our knowledge, has never been implemented (nor do we believe it would be practical since the Suffix Array does not compress as well as the text — read also below).

The *move* structure of Nishimoto and Tabei [40] managed to reduce the number of I/O operations of the $r$-index by a $\log\log n$ factor (essentially replacing predecessor queries with pointers), even though this solution still requires $O(m + occ)$ I/O operations in the worst case. Practical implementations [5, 50] are one order of magnitude faster than the $r$-index, albeit several times larger.

Puglisi and Zhukova [42] showed that relative Lempel-Ziv compression (yielding a very cache-efficient compressed random access data structure) applied to a transformation of the Suffix Array does indeed yield good query-time memory locality in practice (even though with no formal guarantees), but at the price of blowing up the space of the $r$-index by one order of magnitude due to the fact that the Suffix Array does not compress as well as the text.

More recently, Cenzato et al. [9] adopted a new way of approaching the problem: they showed that, rather than trying to compress the full Suffix Array, a tiny subset of the Prefix Array (the *suffixient array*, an integer array of length $O(r)$ [39]) is sufficient to solve pattern matching queries, provided that (compressed) random access is available on the text. Thanks to good cache-locality at query time, their approach simultaneously (i) uses *less* space than the $r$-index and orders of magnitude less space than the Suffix Array, and (ii) solves *find* queries (that is, returns *one* pattern occurrence) faster than the Suffix Array and *orders of magnitude faster* than the $r$-index. In Appendix A we briefly summarize the ideas behind suffixient arrays.

**Overview of our techniques.** Even though suffixient arrays do reduce the term $m$ in the query complexity to $O(1+m/B)$, (i) they allow locating only *one* pattern occurrence; (ii) even worse, the returned occurrence depends on implementation details of the index and cannot be chosen by the user; (iii) as we show in this paper, the smallest suffixient set size $\chi$ can be twice as large as $r$. Our results stem from the observation that the issues (i-ii) are, in fact, connected. As a byproduct, we also solve issue (iii) by providing a Prefix Array sample of size bounded by $r$ that is still sufficient to perform pattern matching. 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 4), we obtain a compressed index that returns 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 $\pi$, (ii) we build a *suffix tree path decomposition* (STPD for brevity) prioritizing the leftmost paths (i.e., smaller $\pi$) in such an order, 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.

This procedure can be formalized precisely as follows. In Section 3 we observe that each order-preserving STPD (i.e., an STPD using an order-preserving $\pi$) is associated with a Longest Previous Factor array $\mathrm{LPF}_\pi$ storing in entry $\mathrm{LPF}_\pi[i]$ the length $k$ of the longest factor $\mathcal{T}[j,j+k-1] = \mathcal{T}[i,i+k-1]$ occurring in a position $j$ with $\pi(j) < \pi(i)$. This array generalizes the well-known Permuted Longest Common Prefix Array PLCP (obtained by taking $\pi = \mathrm{ISA}$, the Inverse Suffix Array), and the Longest Previous Factor array LPF (obtained by taking $\pi = id$, the identity permutation). Both $\pi = \mathrm{ISA}$ and $\pi = id$ can be easily shown to be order-preserving. Then, the positions at the beginning of each STPD path — that is, the integers contained in PDA — are $\{i + \mathrm{LPF}_\pi[i] \ : \ i \in [n]\}$. The size of such a set is precisely the number of *irreducible* values in $\mathrm{LPF}_\pi$, that is, positions $i$ such that either $i = 1$ or $\mathrm{LPF}_\pi[i] \neq \mathrm{LPF}_\pi[i-1]-1$ (a terminology borrowed from the literature on the PLCP array). This is interesting because the number of irreducible values in $\mathrm{LPF}_\pi$ is known to be at most $r$ when $\pi = \mathrm{ISA}$ (and in practice consistently smaller than that, as we show experimentally). More in general, by taking $\pi$ to be any order-preserving permutation, we obtain a new compressibility measure ($|\mathrm{PDA}_\pi|$) generalizing the number of irreducible PLCP values. In Theorem 2 we show that such repetitiveness measure is reachable, meaning that $O(|\mathrm{PDA}_\pi|)$ words of space are sufficient to compress the text. We provide a general mechanism of space $O(|\mathrm{PDA}_\pi|)$ (on top of the text oracle) for locating all pattern occurrences on any order-preserving $\pi$. When $\pi$ is the lexicographic rank of suffixes ($\pi = \mathrm{ISA}$) or the colexicographic rank of prefixes ($\pi = \mathrm{IPA}$, the Inverse Prefix Array), we describe more efficient (both in theory and practice) strategies for locating all pattern occurrences. We give more details below.

*Lexicographic rank.* We start in Section 3.1 with perhaps the most natural order-preserving permutation $\pi$: the lexicographic rank among the text's suffixes (see Figure 1). We prove that the corresponding PDA array — called $\texttt{st-lex}^-$ — has size $|\texttt{st-lex}^-| \le r$. In contrast, we show that (i) the smallest suffixient set [9] has size $\chi \le r + w - 1$, where $w \le r$ is the number of leaves in the Weiner link tree, (ii) show that this upper bound is tight, and (iii) provide an infinite string family on which $|\texttt{st-lex}^-| \le r \le \chi/2 - 1$ holds (Corollary 5). This proves a separation between $r$ and $\chi$ and proves that our new sampling of the Prefix Array is superior to suffixient sets as a function of $r$. At this point, we consider the dual order-preserving permutation $\bar{\pi}(i) = n - \pi(i) + 1$, yielding an STPD that always chooses the lexicographically-largest suffix tree paths. We prove that the same bound holds on the size of the corresponding array: $|\texttt{st-lex}^+| \le r$. We then show that the colexicographically-sorted array $\texttt{st-lex} = \{j - 1 \ : \ j \in \texttt{st-lex}^- \cup \texttt{st-lex}^+ \cup \{n-1\}\}$, in addition to longest common extension queries on the text, supports several suffix tree operations, including descending to children. Intuitively, we design a constant-size representation $R_u$ of any suffix tree node $u = \mathrm{locus}(\alpha)$ (i.e. $u$ is the node reached by reading $\alpha \in \Sigma^*$ from the suffix tree root) formed by the colexicographic range of $\alpha$ in $\texttt{st-lex}$, the lexicographically smallest and largest text suffixes being prefixed by $\alpha$, and the node's string depth $|\alpha|$. Given a character $c \in \Sigma$, the representation of $u$ combined with $\texttt{st-lex}$ and Range Minimum/Maximum Query (RMQ) data structures of $O(|\texttt{st-lex}|)$ bits allow us to find the lexicographically smallest and largest text suffixes being

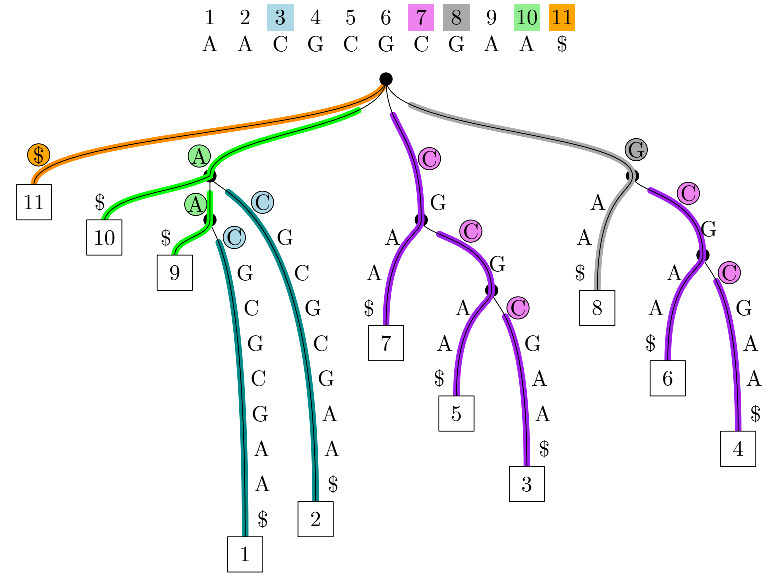


**Fig. 1.** Overview of our technique. We sort $\mathcal{T}$'s suffixes $\mathcal{T}[i, n]$ (equivalently, suffix tree leaves) by increasing $\pi(i)$. In this example, $\pi$ corresponds to the standard lexicographic order of the text's suffixes (but $\pi$ can be more general). This induces a *suffix 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 $|\text{PDA}|$ is bounded by universal compressibility measures and that it can be used to support basic suffix tree navigation and pattern matching operations.

prefixed by $\alpha \cdot c$. Then, the longest common prefix between those two suffixes (found by a longest common extension query) is precisely the length of the suffix tree edge reached by reading $\alpha \cdot c$ from the root. This allows us to find the remaining characters $\beta \in \Sigma^*$ labeling that edge, meaning that the end of the edges is reached by reading $\alpha \cdot c \cdot \beta$ from the root. In turn, this allows us to reconstruct the representation of node $child(x, c)$ via binary search on `st-lex` and longest common extension queries.

The procedure sketched above is already sufficient to locate the suffix tree locus of pattern $P$ and, by navigating the subtree rooted on that node, reporting all the $occ$ occurrences of $P$ in logarithmic time each, $O(occ \log n)$ in total. We reduce this time to $O(\log\log(n/r) + occ)$ as follows. Getting the leftmost/rightmost leaves of the subtree rooted in a given node and returning the beginning $i$ of the suffix $\mathcal{T}[i, n]$ corresponding to a given leaf are easily supported in constant time each thanks to the particular node representation that we adopt. Finally, leaf pointers (connecting leaves in lexicographic order) are supported in the claimed running time using techniques borrowed from the $r$-index [19] (the so-called $\phi$-function, optimized with the *move* technique of Nishimoto and Tabei [40]), adding further $O(r)$ memory words of space.

*General locating mechanism.* We proceed in Section 3.2 by presenting a general technique to locate all pattern occurrences on *any* order-preserving STPD. Our technique is based on the observation that array $\text{LPF}_\pi$ induces an *overlapping bidirectional parse* (i.e., a text factorization generalizing

Lempel-Ziv '77 and allowing phrases to overlap) with $|\mathrm{PDA}_\pi|$ phrases. We prove that these factorizations have the following remarkable property: any given pattern $P$ that is a substring of $T$ has exactly one text occurrence $\mathcal{T}[i,j]$ not entirely contained in a single phrase — that is, a *primary occurrence*. All the remaining occurrences of $P$ are entirely contained inside a phrase — we call these *secondary occurrences* — and can be located from the primary occurrence by resorting to two-dimensional orthogonal point enclosure in $O(\log n)$ time each. To locate the primary occurrence, we present an algorithm working on any order-preserving STPD and returning the pattern occurrence $P[i,j]$ minimizing $\pi(i)$.

*Colexicographic rank.* In Section 3.3 we then observe that on a particular order-preserving permutation $\pi$, the above general locating mechanism gets simplified: this happens when $\pi$ is the colexicographic rank of the text's prefixes (that is, $\pi = \mathrm{IPA}$). We show that the corresponding PDA array — deemed $\texttt{st-colex}^-$ — has size $|\texttt{st-colex}^-| \le \bar{r}$, where $\bar{r}$ is the number of runs in the Burrows-Wheeler transform of $\mathcal{T}$ reversed (note the symmetry with $|\texttt{st-lex}^-| \le r$). This STPD possesses a feature that makes it appealing for a practical implementation: the image $\pi(\texttt{st-colex}^-)$ of $\texttt{st-colex}^-$ through $\pi$ is increasing. Ultimately, this implies that we do not need Range Minimum queries (constant-time in theory, but slow in practice) to identify samples minimizing $\pi$. This simplified version of the algorithm of Section 3.2 allows us to locate the colexicographically-smallest text prefix being suffixed by the query pattern $P$. At this point, the locate mechanism of the $r$-index allows us to locate the remaining *occ* occurrences. Due to a smaller space usage with respect to the compressed suffix tree of Section 3.1 (the constant hidden in the $O(\bar{r})$ space usage is close to 3 in practice), our experimental results on *locate* queries use an optimized implementation of this index. Experimentally, we show that this index solves the long-standing locality problem of compressed indexes, being at the same time *smaller* than the $r$-index and *one to two orders of magnitude faster*. Due to the considerable interest [1, 2, 14, 43, 44, 45, 50] that the $r$-index has gained since its introduction in applications related with computational pangenomics (that is, indexing repetitive genomic collections), we expect our new solution to have a profound impact in the field.

*Text position order.* Of interest is also the order-preserving identity permutation $\pi(i) = i$. The corresponding path decomposition array $\texttt{st-pos}^-$ allows computing the *leftmost* pattern occurrence in the text. The size $|\texttt{st-pos}^-|$ of this STPD array is the number of irreducible values in the Longest Previous Factor Array (LPF), a new interesting repetitiveness measure that we analyze both in theory and experimentally. Letting $p = |\texttt{st-pos}^-|$, we show that $p$ words of space are close to optimal in the worst case to compress strings of length $n$ with $|\texttt{st-pos}^-| = p$. Together with the fact that $O(p)$ words of space are sufficient to compress the text (Theorem 2), this indicates that $|\texttt{st-pos}^-|$ is a strong compressibility measure, a fact that we confirm in section 4 by showing that $|\texttt{st-pos}^-|$ is consistently smaller than $r$ in practice. We then briefly discuss applications of this STPD and observe that it is tightly connected with (1) the celebrated Ukkonen's suffix tree construction algorithm, and (2) the PPM* (prediction by partial matching) compression algorithm.

## 2 Preliminaries

Let $\Sigma$ be a finite integer alphabet of size $\sigma$ endorsed with a total order $<$, which we call the *lexicographic order*. A *string* of length $n$ over $\Sigma$ is a sequence $\mathcal{S} = \mathcal{S}[1]\mathcal{S}[2]\cdots\mathcal{S}[n] \in \Sigma^n$. The *reverse* $\mathcal{S}^{rev}$ of $\mathcal{S}$ is $\mathcal{S}^{rev} = \mathcal{S}[n]\mathcal{S}[n-1]\cdots\mathcal{S}[1]$. With $\Sigma^*$ we denote the set of strings of arbitrary length, i.e., $\Sigma^* = \bigcup_{n\ge 0}\Sigma^n$. A *text* is a string $\mathcal{T} \in \Sigma^n$ such that symbol $\mathcal{T}[n] = \$ \in \Sigma$ appears only in $\mathcal{T}[n]$ and is lexicographically-smaller than all other alphabet symbols, i.e., $\$ < c$ for all $c \in \Sigma \setminus \{\$\}$.

With $[i, j]$ we denote the integer interval $\{i, i+1, \ldots, j\}$ and with $[i]$ the interval $[1, i]$. For an arbitrary string $\mathcal{S} \in \Sigma^*$ and an interval $[i, j]$, we let $\mathcal{S}[i, j] = \mathcal{S}[i]\mathcal{S}[i+1] \ldots \mathcal{S}[j]$. A string $\alpha$ is a *substring* of a string $\mathcal{S}$, if there exists an interval $[i, j]$ such that $\alpha = \mathcal{S}[i, j]$. If $i = 1$ $(j = n)$, we call $\mathcal{S}[i, j]$ a *prefix* (*suffix*) of $\mathcal{S}$. The string $\mathcal{S}[i, j]$ is called a *proper prefix* (*proper suffix*) if it is a prefix (suffix) and in addition $\mathcal{S}[i, j] \neq \mathcal{S}$. To improve readability, we use symbols $\mathcal{S}$ and $\mathcal{T}$ to indicate the main string/text subject of our lemmas and theorems (usually, this is the string/text being indexed), and Greek letters $\alpha, \beta, \ldots$ for their substrings.

We use the same notation that is used for strings for indexing arrays, i.e., if $A \in U^n$ is an array of $n$ elements over a universe $U$, then $A[i, j] = A[i] \ldots A[j]$. For a function $f$ with domain $U$, we use $f(A)$ to denote the array $f(A[1]) \ldots f(A[n])$.

**Definition 1.** *A substring $\alpha$ of $\mathcal{S}$ is said to be* right-maximal *if (1) it is a suffix of $\mathcal{S}$ or (2) there exist distinct $a, b \in \Sigma$ such that $\alpha a$ and $\alpha b$ are substrings of $\mathcal{S}$.*

We extend the *lexicographic order* from $\Sigma$ to $\Sigma^*$ as follows, denoting it with $<_{\text{lex}}$. For two strings (substrings) $\alpha$ and $\beta$, it holds that $\alpha <_{\text{lex}} \beta$ if $\alpha$ is a proper prefix of $\beta$, or if there exists $j$ such that $\alpha[i] = \beta[i]$ for all $i \in [j-1]$ and $\alpha[j] < \beta[j]$. The *co-lexicographic order* $<_{\text{colex}}$ is defined symmetrically: for two strings $\alpha$ and $\beta$, it holds that $\alpha <_{\text{colex}} \beta$ if $\alpha$ is a proper suffix of $\beta$, or if there exists $j$ such that $\alpha[|\alpha| - i + 1] = \beta[|\beta| - i + 1]$ for all $i \in [j-1]$ and $\alpha[|\alpha| - j + 1] < \beta[|\beta| - j + 1]$.

We proceed with the definition of the longest common prefix (suffix) functions.

**Definition 2** (lcp, lcs)**.** *The* longest common prefix *function* lcp *(the* longest common suffix *function* lcs*) is defined as the function that, for two strings $\alpha \in \Sigma^n$ and $\beta \in \Sigma^m$, returns the maximum integer $k = \text{lcp}(\alpha, \beta)$ $(k = \text{lcs}(\alpha, \beta))$ such that $\alpha[1, k] = \beta[1, k]$ $(\alpha[n-k+1, n] = \beta[m-k+1, m])$.*

Using these functions we now define longest common extension oracles.

**Definition 3** (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 $\mathcal{S}.\text{rlce}(i, j) = \text{lcp}(\mathcal{S}[i, n], \mathcal{S}[j, n])$ $(\mathcal{S}.\text{llce}(i, j) = \text{lcs}(\mathcal{S}[1, i], \mathcal{S}[1, j]))$. A* longest common extension (lce) *oracle is an oracle that supports both $\mathcal{S}$.*rlce *and $\mathcal{S}$.*llce *queries for the string $\mathcal{S}$.*

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

We review further basic concepts that we consider the expert reader to already be familiar with in Appendix B. This includes the Suffix Array, suffix tree and Burrows-Wheeler transform.

## 3 Order-preserving STPDs

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 property (but the technique can be made to work on any permutation: we will treat the general case in an extension of this article).

**Definition 4 (Order-preserving permutation).** *Let $\mathcal{S} \in \Sigma^n$ be a string and $\pi : [n] \to [n]$ be a permutation. The permutation $\pi$ 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. It is a simple exercise to 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). We will prove this formally in Lemmas 4 and 12.

*Suffix tree path decomposition.* In this paper, 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 $\pi$, 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 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 $k = \max_{j'<j} \mathrm{rlce}(\pi^{-1}(j'), \pi^{-1}(j))$ be the longest common prefix between the $j$-th suffix and all the previous suffixes in the order induced by $\pi$. 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 $\pi$. 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.

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

**Definition 5 (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 $\pi$ is the integer array that, for $i \in [n]$, is defined as:*

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

**Definition 6 (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 $\pi$ 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 $\mathrm{LPF}_\pi$ and $\mathrm{PDA}_\pi$ instead of $\mathrm{LPF}_{\mathcal{S},\pi}$ and $\mathrm{PDA}_{\mathcal{S},\pi}$, respectively. When also $\pi$ is clear from the context, we will write LPF and PDA.

*Example 1.* Consider the STPD of Figure 1. In this example, $\pi(i)$ is the rank of leaf $i$ in lexicographic order (in other words, $\pi = \text{ISA}$): $\pi = (4, 5, 8, 11, 7, 10, 6, 9, 3, 2, 1)$. Then, the corresponding LPF array corresponds to the PLCP array: $\text{LPF} = \text{PLCP} = [2, 1, 4, 3, 2, 1, 0, 0, 1, 0, 0]$. For instance, $\text{LPF}[1] = 2$ 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 $2 = \text{LPF}[1]$.

At this point, the sequence $i + \text{LPF}[i]$ 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: $\text{PDA} = [11, 10, 3, 7, 8]$ (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 $\pi$ and the corresponding array $\text{PDA}_\pi$ are said to be *order-preserving* if $\pi$ is order-preserving.

The expert reader might have noticed that, for the permutation $\pi$ used in Example 1 (lexicographic rank), the values in PDA are in a one-to-one correspondence with the *irreducible LCP values* (known to be at most $r$ in total, which gives a hint of how we will later prove $|\text{PDA}_\pi| \le r$ for this particular $\pi$). As a matter of fact, our technique generalizes the notion of *irreducible values* to any array $\text{LPF}_\pi$ such that $\pi$ is order-preserving. First, note that the array $\text{LPF}_\pi$ is almost nondecreasing:

**Lemma 1.** *For any string $\mathcal{S} \in \Sigma^n$, if $\pi$ is order-preserving then for every $i > 1$ it holds $\text{LPF}_\pi[i] \ge \text{LPF}_\pi[i-1] - 1$. In particular, the sequence $i + \text{LPF}_\pi[i]$, for $i = 1, \ldots, n$, is nondecreasing.*

*Proof.* If, for a contradiction, it were $k = \text{LPF}_\pi[i-1] \ge \text{LPF}_\pi[i] + 2 \ge 2$, then the position $j - 1$ with $\pi(j-1) < \pi(i-1)$ such that $\mathcal{S}[j-1, j+k-2] = \mathcal{S}[i-1, i+k-2]$ would satisfy $\mathcal{S}[j-1, j] = \mathcal{S}[i-1, i]$ hence, by the order-preserving property of $\pi$, $\pi(j) < \pi(i)$ would hold. But then, since $\mathcal{S}[j, j+k-2] = \mathcal{S}[i, i+k-2]$, we would have $\text{LPF}_\pi[i] \ge \text{rlce}(i, j) \ge k-1 \ge \text{LPF}_\pi[i]+1$, a contradiction. □

Values where the inequality of Lemma 1 is strict are of particular interest:

**Definition 7.** *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 $\text{LPF}_\pi$ position *if and only if either $i = 1$ or $\text{LPF}_\pi[i] \ne \text{LPF}_\pi[i-1] - 1$.*

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

*Remark 2.* For any order-preserving permutation $\pi : [n] \to [n]$, $\{x \in \text{PDA}_\pi\} = \{i + \text{LPF}_\pi[i] : i \text{ is an irreducible } \text{LPF}_\pi \text{ position}\}$, hence $|\text{PDA}_\pi|$ is equal to the number of irreducible $\text{LPF}_\pi$ positions (since $\text{PDA}_\pi$ contains distinct values).

The following lemma generalizes the well-known relation between irreducible PLCP values and the Burrows-Wheeler transform [22, Lemma 4] in more general terms.

**Lemma 2.** *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 $\text{LPF}_\pi$ position. Then, for every $j > 1$ with $\pi(j-1) < \pi(i-1)$ and $\text{rlce}(i, j) = \text{LPF}_\pi[i]$, it holds that $\mathcal{S}[i-1] \ne \mathcal{S}[j-1]$.*

*Proof.* Let $i > 1$ be an irreducible $\text{LPF}_\pi$ position. We analyze separately the cases (i) $\text{LPF}_\pi[i] = 0$ and (ii) $\text{LPF}_\pi[i] > 0$.

(i) If $\text{LPF}_\pi[i] = 0$ and $i > 1$ is irreducible, then by definition of irreducible position $\text{LPF}_\pi[i-1] \neq \text{LPF}_\pi[i] + 1$ holds. Combining this with Lemma 1 we obtain $\text{LPF}_\pi[i-1] < \text{LPF}_\pi[i] + 1 = 1$, hence $\text{LPF}_\pi[i-1] = 0$. Assume now, for a contradiction, that there exists $j > 1$ with $\pi(j-1) < \pi(i-1)$, $\text{rlce}(i,j) = \text{LPF}_\pi[i] = 0$, and $\mathcal{S}[i-1] = \mathcal{S}[j-1]$. In particular, this implies $\text{rlce}(i-1, j-1) = 1$. Then, $0 = \text{LPF}_\pi[i-1] \geq \text{rlce}(i-1, j-1) = 1$, a contradiction.

(ii) Let $\text{LPF}_\pi[i] > 0$ and $i > 1$ be irreducible. Let moreover $j > 1$ be such that $\text{rlce}(i,j) = \text{LPF}_\pi[i]$. Assume, for a contradiction, that $\mathcal{S}[i-1] = \mathcal{S}[j-1]$. Then, since $\pi(j-1) < \pi(i-1)$, $\text{LPF}_\pi[i-1] \geq \text{rlce}(i-1, j-1) = \text{LPF}_\pi[i] + 1$ holds. On the other hand, by Lemma 1 any order-preserving $\pi$ must satisfy $\text{LPF}_\pi[i-1] \leq \text{LPF}_\pi[i] + 1$. We conclude that $\text{LPF}_\pi[i-1] = \text{LPF}_\pi[i] + 1$, hence $i$ cannot be an irreducible position and we obtain a contradiction. □

Later we will use Remark 2 and Lemma 2 to bound the size of our compressed suffix tree.

Lemma 3 formalizes the following intuitive fact about order-preserving STPDs. Consider an STPD built on string $\mathcal{S}$ using order-preserving permutation $\pi$. If for a suffix tree node $u = \text{locus}(\alpha)$, 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 $t \in \text{PDA}_\pi$.

**Lemma 3.** *Let $\mathcal{S} \in \Sigma^n$ be a string and $\pi : [n] \to [n]$ be an order-preserving permutation. For any one-character right-extension $\alpha \cdot a$ of a right-maximal substring $\alpha$ of $\mathcal{S}$, define $\pi(\alpha \cdot a) = \min\{\pi(i-1) \ : \ \mathcal{S}[1,i] \text{ is suffixed by } \alpha \cdot a\}$.*

*Then, for any two distinct one-character right-extensions $\alpha \cdot a \neq \alpha \cdot b$ of $\alpha$ with $\pi(\alpha \cdot a) < \pi(\alpha \cdot b)$, there exists $t \in \text{PDA}_\pi$ such that $\mathcal{S}[1,t]$ is suffixed by $\alpha \cdot b$.*

*Proof.* Let $a, b \in \Sigma$ with $\pi(\alpha \cdot a) < \pi(\alpha \cdot b)$. Let $j', j$ be such that $\pi(j') = \pi(\alpha \cdot a) < \pi(\alpha \cdot b) = \pi(j)$. In particular, $\mathcal{S}[1, j'+1]$ is suffixed by $\alpha \cdot a$ and $\mathcal{S}[1, j+1]$ is suffixed by $\alpha \cdot b$. Then, by the order-preserving property of $\pi$ it holds that $\pi(j' - |\alpha| + 1) < \pi(j - |\alpha| + 1)$, hence $\text{LPF}_\pi[j - |\alpha| + 1] \geq \text{rlce}(j - |\alpha| + 1, j' - |\alpha| + 1) = |\alpha|$. On the other hand, by definition of $\pi(\alpha \cdot b) = \min\{\pi(i-1) \ : \ \mathcal{S}[1,i] \text{ is suffixed by } \alpha \cdot b\}$, we also have that $\text{LPF}_\pi[j - |\alpha| + 1] \leq |\alpha|$. We conclude $\text{LPF}_\pi[j - |\alpha| + 1] = |\alpha|$. But then, $(j - |\alpha| + 1) + \text{LPF}_\pi[j - |\alpha| + 1] = j + 1 \in \text{PDA}_\pi$. Since, as observed above, $\mathcal{S}[1, j+1]$ is suffixed by $\alpha \cdot b$, the claim follows by taking $t = j + 1$. □

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

**Theorem 2.** *Let $\mathcal{S} \in \Sigma^n$ be a string over alphabet of size $\sigma = |\Sigma|$ and $\pi : [n] \to [n]$ be an order-preserving permutation. Then, $\mathcal{S}$ can be compressed in $O(|\text{PDA}_\pi| \log(n\sigma))$ bits of space.*

*Proof.* Our compressed representation is as follows. For each irreducible $\text{LPF}_\pi$ position $i$, we store the quadruple $(i, \text{LPF}_\pi[i], s_i, \mathcal{S}[i + \text{LPF}_\pi[i]])$, where $s_i$ is the "source" of $i$, that is, any integer $s_i \in [n]$ such that $\pi(s_i) < \pi(i)$ and $\mathcal{S}[s_i, s_i + \text{LPF}_\pi[i] - 1] = \mathcal{S}[i, i + \text{LPF}_\pi[i] - 1]$ (if $\text{LPF}_\pi[i] = 0$, the latter condition is true for any $s_i \in [n]$). This set of quadruples takes $O(|\text{PDA}_\pi| \log(n\sigma))$ bits of space by Remark 2.

We show how to reconstruct any character $\mathcal{S}[j]$ given $j \in [n]$ and the above representation. We consider two cases.

(i) If $j = i + \mathrm{LPF}_\pi[i]$ for some quadruple $(i, \mathrm{LPF}_\pi[i], s_i, \mathcal{S}[i + \mathrm{LPF}_\pi[i]])$, then $\mathcal{S}[j] = \mathcal{S}[i + \mathrm{LPF}_\pi[i]]$ (explicitly stored) and we are done.

(ii) Otherwise, find any quadruple $(i, \mathrm{LPF}_\pi[i], s_i, \mathcal{S}[i + \mathrm{LPF}_\pi[i]])$ such that $i \le j < i + \mathrm{LPF}_\pi[i]$ (there must exist at least one such quadruple since $j$ is either irreducible or reducible). Let $j_1 = s_i + (j - i)$. By the definition of $s_i$, it holds that $\mathcal{S}[j] = \mathcal{S}[j_1]$. Moreover, since $\mathcal{S}[s_i, s_i + \mathrm{LPF}_\pi[i] - 1] = \mathcal{S}[i, i + \mathrm{LPF}_\pi[i] - 1]$, $\pi(s_i) < \pi(i)$, and by the order-preserving property of $\pi$, it follows that $\pi(j_1) < \pi(j)$. We repeat recursively the above procedure to extract $\mathcal{S}[j_1]$. Let $j, j_1, j_2, \ldots$ be the sequence of text positions, with $\mathcal{S}[j] = \mathcal{S}[j_1] = \mathcal{S}[j_2] = \ldots$, obtained by repeating recursively the above procedure. Since $\pi(j) > \pi(j_1) > \pi(j_2) > \ldots$, and since $\pi(q) \in [n]$ for all $q \in [n]$, it follows that the above recursive procedure must stop in case (i) after at most $n$ steps. □

We proceed as follows. First, we discuss the notable order-preserving permutation given by the lexicographic rank of the text's suffixes (Subsection 3.1). This permutation will enable us to support most suffix tree operations in $O(r)$ space on top of the text oracle. Then, in Subsection 3.2 we describe a general locating mechanism working on any order-preserving STPD. After that, we focus on another particular order-preserving permutation: the colexicographic rank of the text's prefixes (Subsection 3.3). This case is particularly interesting because it allows us to simplify the locating algorithm of Subsection 3.2, and will lead to a practical solution (used in our experiments in Section 4). Finally, in Subsection 3.4 we consider yet another remarkable order-preserving permutation: identity. This permutation will allow us to locate efficiently the leftmost and rightmost pattern occurrences.

### 3.1 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, $\pi$ will always be equal to ISA). We denote with $\texttt{st-lex}^- = \mathrm{PDA}_\pi$ the path decomposition array associated with this permutation $\pi$. 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 4.** *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 $|\texttt{st-lex}^-| \le r$ and $|\texttt{st-lex}^+| \le r$.*

*Proof.* For every $i, j \in [n-1]$ such that $\mathcal{T}[i, n] <_{\mathrm{lex}} \mathcal{T}[j, n]$ and $T[i] = T[j]$, it holds that $\mathcal{T}[i + 1, n] <_{\mathrm{lex}} \mathcal{T}[j + 1, n]$ by definition of the lexicographic order. This proves the order-preserving property for $\pi$ and $\bar{\pi}$.

By Remark 2, it holds that $|\texttt{st-lex}^-|$ is equal to the number of irreducible $\mathrm{LPF}_\pi$ positions. We bijectively map each irreducible $\mathrm{LPF}_\pi$ position $i$ to $\mathrm{BWT}[\mathrm{ISA}[i]]$ and show that $\mathrm{BWT}[\mathrm{ISA}[i]]$ is the beginning of an equal-letter BWT run. This will prove $|\texttt{st-lex}^-| \le r$. Symmetrically, to prove $|\texttt{st-lex}^+| \le r$ we bijectively map each irreducible $\mathrm{LPF}_{\bar{\pi}}$ position $i$ to $\mathrm{BWT}[\mathrm{ISA}[i]]$ and show that $\mathrm{BWT}[\mathrm{ISA}[i]]$ is the *end* of an equal-letter BWT run. Since this case is completely symmetric to the one above, we omit its proof. Let $i$ be an irreducible $\mathrm{LPF}_\pi$ position. We analyze the cases (i) $i = 1$ and (ii) $i > 1$ separately.

(i) If $i = 1$, then $\mathrm{BWT}[\mathrm{ISA}[i]] = \$$. Since the symbol \$ occurs only once in $\mathcal{T}$, $\mathrm{BWT}[\mathrm{ISA}[i]]$ is the beginning of an equal-letter BWT run.

(ii) If $i > 1$, then either $\mathrm{ISA}[i] = 1$ and therefore $\mathrm{BWT}[\mathrm{ISA}[i]] = \mathrm{BWT}[1]$ is the beginning of an equal-letter BWT run, or $\mathrm{ISA}[i] > 1$. In the latter case, let $j = \mathrm{SA}[\mathrm{ISA}[i] - 1]$ (in particular, $\mathrm{ISA}[j] = \mathrm{ISA}[i] - 1$). If $j = 1$, then $\mathrm{BWT}[\mathrm{ISA}[j]] = \$$. Since the symbol \$ occurs only once in $\mathcal{T}$, $\mathrm{BWT}[\mathrm{ISA}[i]]$ is the beginning of an equal-letter BWT run. In the following, we can therefore assume $i > 1$ and $j > 1$.

Then, by the definition of $\mathrm{LPF}_\pi = \mathrm{PLCP}$, it holds that $\mathrm{LPF}_\pi[i] = \mathrm{rlce}(i, j)$. We show that it must be $\mathrm{BWT}[\mathrm{ISA}[i]] = \mathcal{T}[i-1] \neq \mathcal{T}[j-1] = \mathrm{BWT}[\mathrm{ISA}[j]] = \mathrm{BWT}[\mathrm{ISA}[i] - 1]$, which proves the main claim. If, for contradiction, it was $\mathcal{T}[i-1] = \mathcal{T}[j-1]$, then $\mathrm{ISA}[j] < \mathrm{ISA}[i]$ would imply $\pi(j-1) = \mathrm{ISA}[j-1] < \mathrm{ISA}[i-1] = \pi(i-1)$. But then, Lemma 2 would imply $\mathcal{T}[i-1] \neq \mathcal{T}[j-1]$, a contradiction. □

**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 1 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 $\{j = (i-1) \ : \ i \in \texttt{st-lex}^- \cup \texttt{st-lex}^+ \cup \{n+1\}\} \subseteq \{0, \dots, n\}$ 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 4, it holds that $|\texttt{st-lex}| \leq 2r + 1$.

(2) Let $\# \notin \Sigma$ be a new character not appearing in $\Sigma$ (taken to be lexicographically larger than all the characters in $\Sigma$). 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 [36], 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 $\mathcal{FL} = \mathcal{FL}^{-1}$). $\mathcal{LF}$ and $\mathcal{FL}$ can be evaluated in $O(\log \sigma)$ time by running a constant number of rank and select operations on $\mathcal{L}$ and $\mathcal{F}$.

**Definition 8.** *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 3.* 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.

**Node representation.** Suffix tree navigation operations are supported on a particular representation of (explicit) suffix tree nodes that we describe next. First, Lemma 3 immediately implies:

**Corollary 2.** *Let $u$ be an explicit suffix tree node and $\alpha$ be such that $u = \mathrm{locus}(\alpha)$. Then, $\alpha$ suffixes $\mathcal{T}[1, j]$ for at least one value $j \in$ `st-lex`.*

Follows our representation of suffix tree nodes.

**Definition 9 (Suffix tree node representation).** *We represent suffix tree node $u = \mathrm{locus}(\alpha)$ with the tuple of integers*

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

- `st-lex`$[b, e]$ *is the colexicographic range of $\alpha$ in* `st-lex` *(by Corollary 2, $e \geq b$ always holds);*
- $\mathcal{T}[i_{min}, i_{min} + |\alpha| - 1] = \alpha$ *is the occurrence of $\alpha$ minimizing $\pi(i_{min})$, that is, $\mathcal{T}[i_{min}, n]$ is the lexicographically-smallest suffix prefixed by $\alpha$;*
- $\mathcal{T}[i_{max}, i_{max} + |\alpha| - 1] = \alpha$ *is the occurrence of $\alpha$ maximizing $\pi(i_{max})$, that is, $\mathcal{T}[i_{max}, n]$ is the lexicographically-largest suffix prefixed by $\alpha$; and*
- $|\alpha|$ *is the string depth of $u$.*

*Remark 4.* Observe that $n \in$ `st-lex`$[1, 2]$. In particular, if $1 \in$ `st-lex`$^{-} \cup$ `st-lex`$^{+}$ then $0 \in$ `st-lex`. Since 0 corresponds to the text's prefix $\mathcal{T}[1, 0]$ (the empty string), in this case `st-lex`$[1] = 0$ because the empty string is smaller than any other text prefix. In that case, `st-lex`$[2] = n$ because $\mathcal{T}[1, n]$ (ending with \$) is the second colexicographically-smallest sampled prefix. If, on the other hand, $1 \notin$ `st-lex`$^{-} \cup$ `st-lex`$^{+}$, then `st-lex`$[1] = n$.

Based on the above remark:

**Definition 10.** *We denote with $i^* \in \{1, 2\}$ the integer such that* `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 `st-lex` is `st-lex`$[i^*, i^*]$. This will be used later.

**Suffix tree operations.** Next, we show how to support a useful subset of suffix tree operations on our data structure. This will prove Theorem 1. 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 $u = \mathrm{locus}(\epsilon)$ be the suffix tree root. Operation *root*() returns $R_u = (1, |$`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, $i_{min} = \mathrm{SA}[1] = n$ and $i_{max} = \mathrm{SA}[n]$).

*String depth.* Let $R_u = (b, e, i_{min}, i_{max}, \ell)$. Then, *sdepth*$(R_u)$ simply returns $\ell$ 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 $\mathrm{rlce}(i_{min}, i'_{min}) \geq \ell$, that is, if and only if the string $\alpha = \mathcal{T}[i_{min}, i_{min}+\ell-1]$ with $u = \mathrm{locus}(\alpha)$ is a prefix of $\alpha' = \mathcal{T}[i'_{min}, i'_{min} + \ell' - 1]$ with $u' = \mathrm{locus}(\alpha')$. 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 10), if and only if the string $\alpha = \mathcal{T}[i_{min}, i_{min} + \ell - 1]$ with $u = \text{locus}(\alpha)$ ends with \$, that is, $i_{min} + \ell - 1 = n$.

*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 $R_{u'} = (b', e', i'_{min}, i'_{max}, \ell')$. The function *label*$(R_u, R_{u'})$ returns $(i'_{min} + \ell, i'_{min} + \ell' - 1)$ in $O(1)$ time.

*Next leaf.* Nishimoto and Tabei [40] 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[6] can be computed in $O(\log\log(n/r) + k)$ time with a data structure using $O(r)$ words of space.

**Definition 11 ($\phi$-function).** *Let $\bar{\phi} : [n] \rightarrow [n]$ be defined as*

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

Observe that, by the very 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 10 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 lexicographically-smallest label in $\mathcal{L}[b, e]$ (i.e., in out$(u)$). Operation *succ*$(R_u, a)$ returns the lexicographically-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 [36]. 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 3 implies the following corollary:

**Corollary 3.** *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 $\alpha$. All (and only) the letters labeling the outgoing edges from node $u$ appear in $\mathcal{L}[b, e] \setminus \{\#\}$, that is,* $\text{out}(u) = \{c \;:\; c \in \mathcal{L}[b, e] \wedge c \neq \#\}$.

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

*Proof.* Ignoring character #, by definition $\mathcal{L}[b,e]$ only contains characters following occurrences of $\alpha$ in $\mathcal{T}$, that is, characters in $\mathrm{out}(u)$.

We now show that every $c \in \mathrm{out}(u)$ belongs to $\mathcal{L}[b,e]$. Assume that $c \in \mathrm{out}(u)$ is not the lexicographically-smallest character in $\mathrm{out}(u)$ (the other case — $c$ is not the lexicographically-largest character in $\mathrm{out}(u)$ — is symmetric and the following proof adapts by replacing $\pi$ with $\bar{\pi}$). Let $a \neq c$ be the lexicographically-smallest character in $\mathrm{out}(u)$. Then, any text suffix being prefixed by $\alpha \cdot a$ is lexicographically smaller than any text suffix being prefixed by $\alpha \cdot c$, hence (since $\pi = \mathrm{ISA}$) it holds $\pi(\alpha \cdot a) < \pi(\alpha \cdot c)$ (see Lemma 3 for the definition of this overloading of $\pi$ to right-extensions of right-maximal strings). But then, Lemma 3 implies that there exists $t \in \mathrm{PDA}_\pi = \texttt{st-lex}^-$ with $\mathcal{T}[1,t]$ being suffixed by $\alpha \cdot c$, hence $t-1 \in \texttt{st-lex}[b,e]$ and therefore $c \in \mathcal{L}[b,e]$. □

Following Corollary 3, 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 $\mathit{child}(R_u, a) = (0,0,0,0,0)$, 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 $\mathit{child}(R_u, a)$.

Observe that $|\mathrm{out}(u)| \geq 2$, since $u$ is an explicit suffix tree node. We distinguish three cases. (i) $a$ is neither the lexicographically-largest nor the lexicographically-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 lexicographically-smallest label in $\mathcal{L}[b,e] \setminus \{\#\}$, i.e., $a = \min\{c \in \mathcal{L}[b,e] \setminus \{\#\}\}$; (iii) $a$ is the lexicographically-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 3 to the particular STPDs we are using in this section (i.e. those derived from $\pi = \mathrm{ISA}$ and $\bar{\pi}[i] = n - \mathrm{ISA}[i] + 1$):

**Corollary 4.** *Let $\alpha$ be right-maximal, and let $\texttt{st-lex}[b,e]$ be the range containing the text positions $j \in \texttt{st-lex}$ such that $\mathcal{T}[1,j]$ is suffixed by $\alpha$. Then:*

*(a) If $c \in \mathcal{L}[b,e]$ ($c \neq \#$) is not the lexicographically-smallest character in $\mathcal{L}[b,e] \setminus \{\#\}$ then $j \in \texttt{st-lex}[b,e]$, where $\mathcal{T}[j - |\alpha| + 1, n]$ is the lexicographically-smallest suffix being prefixed by $\alpha \cdot c = \mathcal{T}[j - |\alpha| + 1, j+1]$.*

*(b) If $c \in \mathcal{L}[b,e]$ ($c \neq \#$) is not the lexicographically-largest character in $\mathcal{L}[b,e] \setminus \{\#\}$ then $j \in \texttt{st-lex}[b,e]$, where $\mathcal{T}[j - |\alpha| + 1, n]$ is the lexicographically-largest suffix being prefixed by $\alpha \cdot c = \mathcal{T}[j - |\alpha| + 1, j+1]$.*

We now show how to compute $i'_{min}$ and $i'_{max}$ in cases (i-iii).

(i) By Corollary 4, $i''_{min}, i''_{max} \in \texttt{st-lex}[b,e]$, where $\mathcal{T}[i''_{min} - \ell + 1, n]$ and $\mathcal{T}[i''_{max} - \ell + 1, n]$ are the lexicographically-smallest and lexicographically-largest suffixes being prefixed by $\alpha \cdot a$, respectively. Observe that $i''_{min}$ and $i''_{max}$ are not necessarily distinct (they are equal precisely when descending to a leaf). We locate the leftmost $\mathcal{L}[L] = a$ and rightmost $\mathcal{L}[R] = a$ occurrences of letter $a$ in $\mathcal{L}[b,e]$ using rank and select operations on $\mathcal{L}$, in $O(\log\sigma)$ time. Then, we map all the occurrences of $a$ in $\mathcal{L}[b,e]$ to the corresponding $a$'s in $\mathcal{F}$ by applying $\mathcal{LF}$: those occurrences correspond to the range $\mathcal{F}[\mathcal{LF}(L), \mathcal{LF}(R)] = \mathcal{F}[b', e']$, hence $b'$ and $e'$ can be found in $O(\log\sigma)$ time using rank and select operations on $\mathcal{L}$ and $\mathcal{F}$. Observe that, by definition of the permutation $\texttt{st-lex}'$ of $\texttt{st-lex}$ (Definition 8), it holds that $i''_{min}, i''_{max} \in \texttt{st-lex}'[b', e']$. Then, our Range Minimum/Maximum data structures queried in range $\pi(\texttt{st-lex}')[b', e']$ will yield two positions $j_{min}, j_{max}$ such that $\texttt{st-lex}'[j_{min}] = i''_{min}$ and $\texttt{st-lex}'[j_{max}] = i''_{max}$. Although we are not storing $\texttt{st-lex}'$ explicitly (we could, since it would just take $|\texttt{st-lex}|$ further memory words), we can retrieve $i''_{min}$ and $i''_{max}$ by just applying $\mathcal{F}L$: by definition of $\texttt{st-lex}'$ (Definition 8), we have that $\texttt{st-lex}'[j] = \texttt{st-lex}[\mathcal{FL}(j)]$ for any $j \in [|\texttt{st-lex}|]$. Then, $i'_{min} = i''_{min} - \ell + 1$ and $i'_{max} = i''_{max} - \ell + 1$.

(ii) The letter $a$ is the lexicographically-smallest label in $\mathcal{L}[b, e] \setminus \{\#\}$. Since $|\text{out}(u)| \geq 2$ (because $u$ is an internal suffix tree node), we obtain that $a$ is not the lexicographically-largest label in $\mathcal{L}[b, e] \setminus \{\#\}$. Then, by Corollary 4, $i''_{max} \in \texttt{st-lex}[b, e]$, where $\mathcal{T}[i''_{max} - \ell + 1, n]$ is the lexicographically-largest suffix being prefixed by $\alpha \cdot a$. We can find $i''_{max}$ following the same procedure described in point (i) above (resorting to Range Maxima Queries). At this point, since $a$ is the lexicographically-smallest label in $\mathcal{L}[b, e] \setminus \{\#\}$ (equivalently, in $\text{out}(u)$) and $\mathcal{T}[i_{min}, n]$ is the lexicographically-smallest suffix being prefixed by $\alpha$, it also holds that $\mathcal{T}[i_{min}, n]$ is the lexicographically-smallest suffix being prefixed by $\alpha \cdot a$. But then, we are done: $i'_{min} = i_{min}$ and $i'_{max} = i''_{max} - \ell + 1$.

(iii) The letter $a$ is the lexicographically-largest label in $\mathcal{L}[b, e] \setminus \{\#\}$. This case is completely symmetric to case (ii) so we omit the details.

At this point, we know that $\mathcal{T}[i'_{min}, n]$ and $\mathcal{T}[i'_{max}, n]$ are the lexicographically-smallest and lexicographically-largest suffixes prefixed by $\alpha \cdot a$, respectively. Note that $i'_{min} = i'_{max}$ if and only if $\mathit{child}(R_u, a)$ is a leaf. In this case, we simply return $\mathit{child}(R_u, a) = (i^*, i^*, i'_{max}, i'_{max}, n - i'_{max} + 1)$.

In the following, we can therefore assume that $i'_{min} \neq i'_{max}$, hence $\mathit{child}(R_u, a)$ is not a leaf. Then, 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 $\alpha' = \mathcal{T}[i'_{min}, i'_{min} + \ell' - 1]$.

Binary-searching $\alpha'$ in $\texttt{st-lex}$ then yields the maximal range $\texttt{st-lex}[b'', e'']$ such that, for all $j \in [b'', e'']$, $\alpha'$ is a suffix of $\mathcal{T}[1, \texttt{st-lex}[j]]$. Using random access queries on $\mathcal{T}$ to guide binary search, this process requires in total $O(\ell' \log |\texttt{st-lex}|)$ random access queries. We can do better if the text oracle supports lce queries. Since $\alpha' = \mathcal{T}[i'_{min}, i'_{min} + \ell' - 1]$ is a substring of $\mathcal{T}$ itself, each binary search step can be implemented with one llce and one random access query (the latter, extracting one character). This yields $[b'', e'']$ in time $O(t \cdot \log |\texttt{st-lex}|) \subseteq O(t \cdot \log r)$.

Yet another solution uses *z-fast tries* [3,7], a machinery supporting *internal suffix searches* (that is, the queried string is guaranteed to suffix at least one string in the pre-processed set) in a set of $q$ strings each of length bounded by $n$. This solution uses space $O(q)$ and answers suffix-search queries in $O(\log n)$ steps, each requiring computing the fingerprint of a substring of the query $\alpha'$. In our case, the set of strings is $\{\mathcal{T}[1, \texttt{st-lex}[i]] \ : \ i \in [|\texttt{st-lex}|]\}$ and we have that $\alpha' = \mathcal{T}[i_{min}, i_{min} + \ell' - 1]$ indeed suffixes at least one of them by Corollary 2 and by the fact that $i'_{min} \neq i'_{max}$, hence $\alpha'$ is right-maximal. This solution finds $[b'', e'']$ in $O(h \log n)$ time (or I/O operations, if $h$ is the I/O complexity of fingerprinting).

We finally have all ingredients to return our result: $\mathit{child}(R_u, a) = (b'', e'', i'_{min}, i'_{max}, \ell')$. If the oracle only supports lce queries and random access in $O(t)$ time, then $\mathit{child}(R_u, a)$ is answered in $O(t \log r + \log \sigma)$ time. If the oracle also supports fingerprinting queries in $O(h)$ time, then the running time is $O(t + h \log n + \log \sigma)$. The latter is preferable if fingerprinting queries are answered faster than lce queries (e.g., with the text oracle of Prezza [41]).

*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 1.

**Proof of Corollary 1: pattern matching.** To prove Corollary 1, we plug the random access oracle [41] in Theorem 1. This oracle uses $n \log \sigma + O(\log n)$ bits of space and supports lce queries with $O(\log n)$ I/O complexity, fingerprinting with $O(1)$ I/O complexity, and extraction of $\ell$ consecutive characters with $O(1+\ell/B)$ I/O complexity[7] provided that the alphabet size is polynomial ($\sigma \leq n^{O(1)}$). Starting with $k=0$ and $R_u = \mathit{root}()$, suppose we have matched a right-maximal proper pattern's prefix $P[1,k]$ and found the representation $R_u$ of node $u = \text{locus}(P[1,k])$. To continue matching, we compute $R_v = \mathit{child}(R_u, P[k+1])$. If the child exists, we get the edge's text pointers $(i, i+q-1) = \mathit{label}(R_u, R_v)$ ($q$ is the edge's length) and match $P[k+1,\dots]$ with $\mathcal{T}[i,i+q-1]$ by random access, either reaching node $v$ (i.e. the end of the edge $(u,v)$) or reaching the end of $P$. In the former case, we repeat the above operations to continue matching $P$ on descendants of $v$. In the latter case, we are left to locate all the $\mathit{occ}$ pattern occurrences in the subtree rooted in $v$.

To locate the pattern's occurrences, we have two choices. The first (less efficient, corresponding to the original one of Weiner [47]) is to navigate the subtree (of size $O(\mathit{occ})$) rooted in $v$ using $\mathit{child}(\cdot,\cdot)$, $\mathit{first}(\cdot)$, and $\mathit{succ}(\cdot,\cdot)$ queries. For each reached leaf $u$ (we use $\mathit{isleaf}(R_u)$ to check whether we reached a leaf), we call $\mathit{locate}(R_u)$ to report the corresponding pattern occurrence. The I/O complexity of this navigation using the text oracle [41] is $O(\mathit{occ}\cdot(t + h\log n + \log\sigma)) = O(\mathit{occ}\cdot\log n)$.

A more efficient solution consists of navigating just the leaves using $\mathit{next}(\cdot)$. This can be done by computing $R_{left} = \mathit{lleaf}(R_v)$, $R_{right} = \mathit{rleaf}(R_v)$ and evaluating $\mathit{occ}$ consecutive applications of $\mathit{next}(\cdot)$ starting from $R_{left}$ until reaching $R_{right}$ (again, calling $\mathit{locate}(R_u)$ on each leaf $u$ to get a pattern occurrence). This solution locates the $\mathit{occ}$ pattern occurrences with $O(\log\log(n/r) + \mathit{occ})$ I/O complexity.

Finding the locus $v$ of pattern $P$ required calling a $\mathit{child}(\cdot,\cdot)$ operation for each of the $d$ traversed suffix tree edges, plus $O(d + m/B)$ I/O complexity for matching the pattern against the edges' labels. In total, finding $v$ has therefore $O(d\cdot(t + h\log n + \log\sigma) + d + m/B) = O(d\log n + m/B)$ I/O complexity. Taking into account the I/O complexity of locating the pattern's occurrences, this proves Corollary 1.

**`st-lex`$^-$ and `st-lex`$^+$ are superior to suffixient sets.** To conclude this subsection, we show that `st-lex`$^-$ and `st-lex`$^+$ are samplings of the Prefix Array that strictly improve suffixient sets [9]. We first show that the size $\chi$ of the smallest suffixient set satisfies $\chi \leq r + w - 1$, where $w \leq r - 1$ is the number of leaves in the Weiner link tree of $\mathcal{T}$ (Definition 12, Lemma 6). Then, we provide a small example with $r = w - 1$ that matches this upper bound. We use this example to describe an infinite family of strings where $r \leq \chi/2 - 1$ (Corollary 5). This proves the first known separation between $r$ and $\chi$ and, since in Lemma 4 we prove $|\texttt{st-lex}^-| \leq r$, it shows that $\chi$ can be twice as large as $|\texttt{st-lex}^-|$. Since in Section 3.3 we show that the pattern matching algorithm designed for suffixient sets [9] also works on order-preserving STPDs (with only small modifications), this proves the superiority of STPDs with respect to suffixient sets as Prefix Array samples supporting pattern matching queries.

**Definition 12 (Weiner link tree).** *Let $\mathcal{T}$ be any text. The* Weiner link tree $T^W = (V^\circ, E^f)$ *is the tree on the internal nodes $V^\circ$ of the suffix tree of $\mathcal{T}$ that contains as edges all reversed suffix links between them, i.e., $E^f := \{(\text{locus}(\alpha(v)[2,|\alpha(v)|]), v) : v \in V^\circ \wedge |\alpha(v)| > 0\}$.*

[7] Even if in [41] they only prove $O(\ell)$ random access time for a contiguous block of $\ell$ characters in the RAM model, it is not hard to see that their data structure triggers $O(\ell/B)$ I/O operations in the I/O model: the data structure consists in the Karp-Rabin fingerprints of a sample of the text's prefixes. To extract a character, the structure combines the fingerprints (adjacent in memory) of two consecutive prefixes. This locality property leads immediately to the claimed $O(\ell/B)$ complexity.

While the Weiner link tree is usually defined with labeled edges, below we will not need those labels so we omitted them from Definition 12.

We start with a simple observation, whose proof uses some concepts that will be reused later.

**Lemma 5.** *Denote the set of leaves of the Weiner link tree $T^W$ by $L$, and its cardinality by $w = |L|$. Then it holds that $w \leq r - 1$.*

*Proof.* Consider a leaf $v \in L$, that corresponds to an explicit node in the suffix tree with $\alpha = \alpha(v)$. Thus, $\alpha$ is right maximal, $|\mathrm{out}(v)| \geq 2$, and we let $a < b$ be the smallest two elements of $\mathrm{out}(v)$.

Let $c_a$ and $c_b$ be any characters such that $c_a\alpha a$ and $c_b\alpha b$ occur in $\$\mathcal{T}$ (observe that it may be $c_a = \$$ or $c_b = \$$). We now show that it must be that $c_a \neq c_b$. Assume, for a contradiction, that $c_a = c_b$. Then, it must be $c_a = c_b \neq \$$ since (by the definition above) $c_a = c_b = \$$ would mean that both $\alpha a$ and $\alpha b$ prefix $\mathcal{T}$. This proves $c_a = c_b \neq \$$. Then, $c_a = c_b \neq \$$ would imply that $c_a\alpha$ is right maximal. In turn, this would mean that $\mathrm{locus}(c_a\alpha)$ has a suffix link to $v = \mathrm{locus}(\alpha)$, contradicting $v$ being a leaf in the Weiner link tree.

Now, consider the BWT of $\mathcal{T}$. Let $i \in [n]$ be the integer such that $\mathcal{T}[\mathrm{SA}[i-1], n]$ is the lexicographically largest suffix of $\mathcal{T}$ being prefixed by $\alpha a$ and $\mathcal{T}[\mathrm{SA}[i], n]$ is the lexicographically smallest suffix being prefixed by $\alpha b$. Note that $i$ exists by our choice of $a$ and $b$, and that $i > 1$. Choosing $c_a = \mathrm{BWT}[i-1]$ and $c_b = \mathrm{BWT}[i]$ in the reasoning above, we obtain $\mathrm{BWT}[i-1] \neq \mathrm{BWT}[i]$. We call such an $i$ with $\mathrm{BWT}[i-1] \neq \mathrm{BWT}[i]$ a *run boundary* of the BWT. Then, $\alpha$ is injectively associated with this run boundary (that is, no other $\alpha(v') \neq \alpha(v)$ is associated with it), since the longest common prefix of the two lexicographically adjacent suffixes $\mathcal{T}[\mathrm{SA}[i-1], n]$ and $\mathcal{T}[\mathrm{SA}[i], n]$ is $\alpha$.

We conclude that the leaves of the Weiner link tree can be mapped injectively into run boundaries, hence $w \leq r - 1$ holds. □

**Lemma 6.** *For any text $\mathcal{T}$, it holds that $\chi \leq r + w - 1$, where $r$ is the number of runs in the BWT of $\mathcal{T}$ and $w$ is the number of leaves in the Weiner link tree of $\mathcal{T}$.*

*Proof.* Our goal is to construct a suffixient set of size bounded by $r + w - 1$. By [9], a set $S$ of text indices is suffixient, if it "covers" all right maximal extensions $\alpha(v)a$, where $v$ is an internal node in the suffix tree and $a \in \mathrm{out}(v)$, in the sense that there exists $i \in S$ such that $\alpha(v)a$ is a suffix of $\mathcal{T}[1, i]$. From a high-level point of view, our construction first chooses a set of text positions that cover all right maximal extensions at leaves in $T^W$ and then "propagate" these positions up the tree, choosing additional text positions when necessary. We now make this approach precise.

**Definition 13.** *Let $u \in V^\circ$ and $a \in \mathrm{out}(u)$. We define $i_{\alpha(u)a}$ to be the smallest integer such that $\alpha(u)a$ is a suffix of $\mathcal{T}[1, i_{\alpha(u)a}]$.*

**Definition 14.** *Let $u \in V^\circ$. Denote with $u_i$, for $i \in [k]$, the $k$ children of $u$ in the Weiner link tree, that is, the nodes $u_i \in V^\circ$ such that $\alpha(u_i) = c_i\alpha(u)$ for some distinct $c_1, \ldots, c_k \in \Sigma$ (if $u$ is a leaf in the Weiner link tree, then $k = 0$). We define $D(u) := \mathrm{out}(u) \setminus \bigcup_{i \in [k]} \mathrm{out}(u_i)$ (for leaves, this simplifies to $D(u) := \mathrm{out}(u)$).*

Let $I = \bigcup_{v \in V^\circ} \{i_{\alpha(v)a} : a \in D(v)\}$. We prove that $I$ is a suffixient set, and then show that $|I| \leq r + w - 1$, which will yield our claim.

To see that $I$ is suffixient, consider any right extension $\alpha a$ of any right maximal string $\alpha$. Let $\beta \in \Sigma^*$ be the longest string such that (a) $\beta\alpha$ is right maximal and (b) $\beta\alpha a$ occurs in $\mathcal{T}$. Then,

(i) $a \in \mathrm{out}(\mathrm{locus}(\beta\alpha))$ and (ii) if $\mathrm{locus}(\beta\alpha) \notin L$, then $a \notin \mathrm{out}(u)$, where $u$ is any child of $\mathrm{locus}(\beta\alpha)$ in the Weiner link tree. To see that (ii) is true, let $c \in \Sigma$ be such that $\alpha(u) = c\beta\alpha$; if it were $a \in \mathrm{out}(u)$, then $c\beta$ would be a sting longer than $\beta$ with properties (a) and (b), a contradiction. By definition of $D(u)$, properties (i) and (ii) imply $a \in D(\mathrm{locus}(\beta\alpha))$. But then, by definition of $I$ we have that $i_{\beta\alpha a} \in I$, hence $\alpha a$ is "covered" by $I$. This proves that $I$ is a suffixient set.

It now remains to show that $|I| \leq r + w - 1$. (1) Let $v = \mathrm{locus}(\alpha) \in L$ be a leaf in the Weiner link tree. Then, a character $a \in \mathrm{out}(v)$ corresponds to a right extension $\alpha a$ at $v$. As before, for each $a \in \mathrm{out}(v)$, there exists some character $c_a$ such that $c_a\alpha a$ occurs in the text (where again we may choose $c_a = \$$ when $\alpha a$ is a prefix). As in the proof of Lemma 5, we know that regardless of the choice of the $c_a$, they can never be identical for distinct $a, a' \in \mathrm{out}(v)$. Thus, $|\{c_a : a \in \mathrm{out}(v)\}| = |\mathrm{out}(v)|$ and there are $|\mathrm{out}(v)|$ distinct strings $c_a\alpha a$ that we can map to at least $|\mathrm{out}(v)| - 1$ run boundaries in the BWT that have LCP $\alpha$. It follows that we can assign the at most $|\mathrm{out}(v)|$ suffixient set samples that we create at the leaf $v$ as follows. We assign one of the samples to the leaf $v$ itself (counting towards the $w$ leaves) and (at most) $|\mathrm{out}(v)| - 1$ of them to (at least) $|\mathrm{out}(v)| - 1$ distinct BWT run boundaries.

(2) Now let $v = \mathrm{locus}(\alpha) \in V^\circ \backslash L$ be an internal node of the Weiner link tree. Let $u_i = \mathrm{locus}(c_i\alpha)$ for $i \in [k]$ ($k \geq 1$) be the internal nodes that link to $v$ via suffix links (that is, the children of $v$ in the Weiner link tree). Recall that $D(v)$ contains all and only the characters $a \in \mathrm{out}(v)$ such that $\alpha(v)a$ occurs in $\mathcal{T}$ and $\alpha(u_i)a = c_i\alpha(v)a$ does not occur in $\mathcal{T}$ for all $u_i$ defined as above. For each $a \in D(v)$, there again exists $c_a$ such that $c_a\alpha a$ occurs in the text (or $c_a = \$$ when $\alpha a$ is a prefix). Now, let $P(v) := \{c_a : a \in D(v)\}$, where again we have $|P(v)| = |D(v)|$ as assuming otherwise contradicts the fact that $a$ forms a right maximal extension at $v$ but not at any of its children in the Weiner link tree. It now follows that in the BWT, $\alpha$ is preceded by at least $|D(v)|$ different characters $c_a$, one corresponding to the string $c_a\alpha$ for each $a \in D(v)$. Furthermore, there is at least one more distinct character $c_i$ (say, $c_1$) preceding $\alpha$. These $|D(v)| + 1$ distinct characters induce at least $|D(v)|$ run boundaries with LCP $\alpha$, since distinct $c\alpha$ and $c'\alpha$ can not be succeeded by the same character. (The $c_a$ corresponding to $D(v)$ are succeeded by $a$ only, while $c_1$ is succeeded by a character in some $\mathrm{out}(u_i)$ that does not occur in $D(v)$.) Hence there are at least $|D(v)|$ run boundaries that the $|D(v)|$ suffixient set samples at $v$ can be assigned to. □

We proceed with some remarks. (1) Our bound of $r + w - 1$ is strictly better than the bound of $2r$ by Navarro et al. [39] as $w \leq r$ follows from our proof above that uniquely maps samples at leaves of the Weiner link tree to run boundaries. (2) There are examples where $r$ is asymptotically strictly larger than $w$. E.g., imagine appending the \$ character to the $k$'th Fibonacci word $F_k$ that is defined recursively via $F_1 = a$, $F_2 = ab$, and $F_k = F_{k-1}F_{k-2}$ for $k > 2$. We obtain a text of length $n$ that for even $k$ has $\Theta(k) = \Theta(\log n)$ BWT runs and $w = 3$. (3) The upper bound from the previous lemma is tightly matched in the example in Figure 2. That example leads to the following separation, stating that there is an infinite family of strings for which $\chi$ is twice as large as $r$ (and therefore as $|\texttt{st-lex}|$):

**Corollary 5 (Separation between $|\texttt{st-lex}|$, $r$, and $\chi$).** *For any $t \geq 1$, there exists a text of length $11t$ on an alphabet of cardinality $3t$ satisfying*

$$|\texttt{st-lex}| \leq r \leq \chi/2 + 1.$$

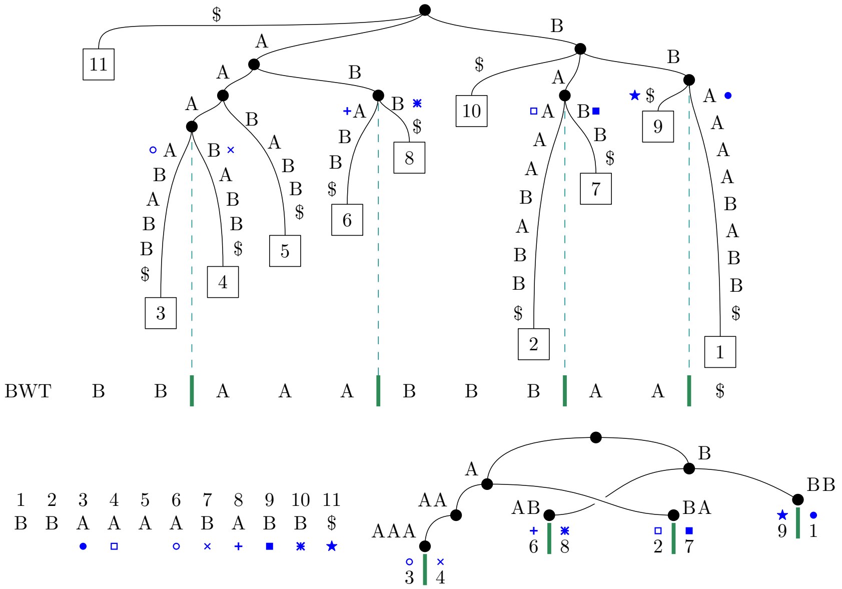
*Proof.* Let $\mathcal{T}$ be the text of the example in Figure 2. This text is of length $n = 11$ and furthermore $\chi(\mathcal{T}) = r(\mathcal{T}) + w(\mathcal{T}) - 1 = 2r(\mathcal{T}) - 2$. Consider now the string $\mathcal{S}_1 \in \{0, 1, 2\}^n$ such that $\mathcal{S}_1[i]$ is equal to 0 if $\mathcal{T}[i] = \$$, 1 if $\mathcal{T}[i] = A$, and 2 if $\mathcal{T}[i] = B$. Since this alphabet renaming does not change the alphabet's order ($0 < 1 < 2$), $\chi(\mathcal{S}_1) = r(\mathcal{S}_1) + w(\mathcal{S}_1) - 1 = 2r(\mathcal{S}_1) - 2$ holds also on $\mathcal{S}_1$. We note that $\chi$, $r$ and $w$ are defined for texts rather than strings, but the above renaming of \$ to 0 maintains the minimality of the terminating character (0) and thus $\chi$, $r$ and $w$ are well-defined for $\mathcal{S}_1$ as well. Now, fix an arbitrary $t \geq 1$ and, for $j \in [t]$ define $\mathcal{S}_j \in \mathbb{N}^n$ as follows: for any $j > 1$ and $i \in [n]$, let $\mathcal{S}_j[i] := \mathcal{S}_1[i] + 3(j-1)$. Individually inside each $\mathcal{S}_j$, this is just another alphabet renaming preserving the alphabet's order so it still holds that $\chi(\mathcal{S}_j) = r(\mathcal{S}_j) + w(\mathcal{S}_j) - 1 = 2r(\mathcal{S}_j) - 2$. We now concatenate these strings in order to obtain $\mathcal{S}' := \mathcal{S}_t \cdot \mathcal{S}_{t-1} \cdots \mathcal{S}_1$. We note again that $\mathcal{S}'$ ends with the unique lexicographically minimum character 0 and can thus be considered a text.

It is not hard to see that, since the alphabets of those strings are disjoint and all characters of $\mathcal{S}_j$ are smaller than those of $\mathcal{S}_{j+1}$, it holds that the characters of $\mathrm{BWT}(\mathcal{S}_j)$ form a contiguous substring in $\mathrm{BWT}(\mathcal{S}')$. This implies $r(\mathcal{S}') = t \cdot r(\mathcal{S}_1)$ or, equivalently, $r(\mathcal{S}_1) = r(\mathcal{S}')/t$.

Moreover, no substring of $\mathcal{S}'$ crossing two adjacent substrings $\mathcal{S}_{j+1}, \mathcal{S}_j$ is right-maximal (the alphabets of those strings being disjoint), which also implies $\chi(\mathcal{S}') = t \cdot \chi(\mathcal{S}_1)$. We obtain:

$$\chi(\mathcal{S}') = t \cdot \chi(\mathcal{S}_1) = t \cdot (2r(\mathcal{S}_1) - 2) = 2r(\mathcal{S}') - 2.$$

Since, by Theorem 4, on string $\mathcal{S}'$ it holds that $|\texttt{st-lex}| \leq r$, we finally obtain $\chi(\mathcal{S}') = 2r(\mathcal{S}') - 2 \geq 2|\texttt{st-lex}(\mathcal{S}')| - 2$. □



| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{T}[i]$ | B | B | A | A | A | A | B | A | B | B | \$ |
| sA | | | ● | □ | | ○ | × | + | ■ | ✱ | ★ |

**Fig. 2.** Example where $8 = \chi = r + w - 1 = 5 + 4 - 1$. On top: the suffix tree for the text $\mathcal{T} = BBAAAABABB\$$. Under the tree, $\mathrm{BWT}(\mathcal{T})[i]$ is shown, being aligned to the $i$-th lexicographically smallest suffix. On the bottom: the text with the suffixient set chosen by the described procedure and the Weiner link tree, in which each node corresponds to an internal node of the suffix tree. Strings indicate the root-to-node path to each node, and the edges correspond to suffix links through which the chosen positions are propagated in order to cover all outgoing edges of the suffix tree. Note that in this example all suffixient set samples are chosen at leaves of the Weiner link tree.

### 3.2 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' $\phi$ function (Definition 11, cases $\pi = \mathrm{ISA}$ and $\pi = \mathrm{IPA}$) to arbitrary order-preserving permutations. This increased generality with respect to Section 3.1, 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 $\pi = \mathrm{IPA}$ (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.

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

**Definition 15 (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* $\mathrm{LPF}_\pi[i] < |P|$. *All the other occurrences are called* secondary occurrences*.*

**Lemma 7.** *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 $\pi(i)$.*

*Proof.* Let $P \in \Sigma^+$ occur in $\mathcal{T}$. To prove the existence of a primary occurrence, suppose for a contradiction that there is no primary occurrence of $P$. Let $\mathcal{T}[i, i + |P| - 1] = P$ be the secondary occurrence minimizing $\pi(i)$. By definition of secondary occurrence we have $\mathrm{LPF}_\pi[i] \geq |P|$, hence, by definition of $\mathrm{LPF}_\pi$, there must exist $i' \in [n]$ such that $\pi(i') < \pi(i)$ and $\mathrm{rlce}(i', i) \geq |P|$ implying that $\mathcal{T}[i', i' + |P| - 1] = P$, which contradicts the minimality of $\pi(i)$.

To prove the uniqueness of the primary occurrence, suppose for a contradiction that there are at least two primary occurrences $i, i' \in [n]$ with $i \neq i'$. Since $\pi$ is a permutation, exactly one of (i) $\pi(i) < \pi(i')$ and (ii) $\pi(i) > \pi(i')$ holds. Without loss of generality, assume that $\pi(i) < \pi(i')$. Then by definition of $\mathrm{LPF}_\pi$, $\mathrm{LPF}_\pi[i'] \geq \mathrm{rlce}(i', i) \geq |P|$, a contradiction. This also shows, as a byproduct, that the unique primary occurrence $P = \mathcal{T}[i, i + |P| - 1]$ is the one minimizing $\pi(i)$ among all (primary and secondary) occurrences. □

**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 $\mathrm{PDA}_\pi$. We first provide an intuition through an example (Example 2). Our algorithm is formalized in Algorithm 1. Then, we prove the algorithm's complexity, correctness, and completeness.

*Example 2.* 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 $\epsilon$ (the empty string). To continue matching $P$, we need to change path. We binary search $\mathrm{PDA}_\pi = \texttt{st-lex}^-$ 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 $\pi(i) = \mathrm{ISA}[i]$. 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 $\mathrm{PDA}_\pi = \texttt{st-lex}^-$ 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 $\pi$, 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 [9] (see Appendix A) allows locating the primary pattern occurrence on any order-preserving STPD. The search algorithm is sketched in Example 2 and formalized in Algorithm 1. We need the following data structures:

1. A Range Minimum Data structure on $\pi(\mathrm{PDA}_\pi)$, requiring just $O(|\mathrm{PDA}_\pi|)$ bits and answering queries in constant time. In Algorithm 1, this data structure allows finding the arg min at Line 6 in constant time.

2. A data structure supporting suffix searches on the text's prefixes $\{\mathcal{T}[1, \mathrm{PDA}_\pi[t]] \;:\; t \in [|\mathrm{PDA}_\pi|]\}$. Given $(i,j,c) \in [n] \times [n] \times \Sigma$, $\mathrm{sufsearch}(i,j,c)$ returns the maximal range $[b,e] \subseteq [|\mathrm{PDA}_\pi|]$ such that $\mathcal{T}[1, \mathrm{PDA}_\pi[t]]$ 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.

3. 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 $i \leftarrow \pi^{-1}(1)$;
 2 $j \leftarrow 1$ ;        /* Invariant: $P[1, j-1] = \mathcal{T}[i-j+1, i-1]$ minimizes $\pi(i-j+1)$ */
 3 while $j \le m$ do
 4     if $\mathcal{T}[i] \neq P[j]$ then
 5         $[b, e] \leftarrow \mathrm{sufsearch}(i-j+1, i-1, P[j])$;
 6         $i' \leftarrow \arg\min_{i' \in [b,e]} \{\pi(\mathrm{PDA}_\pi[i'])\}$;
 7         $i \leftarrow \mathrm{PDA}_\pi[i']$;
 8     $i \leftarrow i+1$; $j \leftarrow j+1$;
 9 if $\mathcal{T}[i-m, i-1] = P$ 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 $\pi(\mathrm{PDA}_\pi)$.

A first simple implementation of $\mathrm{sufsearch}(i, j, c)$ uses binary search on $\mathrm{PDA}_\pi$ and random access on $\mathcal{T}$. This solution executes $O(\log|\mathrm{PDA}_\pi|) \subseteq O(\log n)$ random access queries, each requiring the extraction of $O(m)$ contiguous text characters. If the random access oracle supports the extraction of $\ell$ contiguous text characters with $O(1 + \ell/B)$ I/O complexity, then this solution has $O((1 + m/B)\log n)$ I/O complexity. Due to its simplicity, small space usage (only array $\mathrm{PDA}_\pi$ and the text oracle are required), and attractive I/O complexity, this is the solution we implemented in practice in the index tested in Section 4.

A second optimized implementation of sufsearch executes one llce and one random access query for every binary search step. This solution runs 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.

As done in Section 3.1, a faster solution can be obtained using z-fast tries [3, 7]. Since z-fast tries are guaranteed to answer correctly only internal suffix search queries (that is, the queried string suffixes at least one of the strings in the trie), we build a separate z-fast trie for every distinct character $c \in \Sigma$. The z-fast trie associated with character $c$ is built over text prefixes $\{\mathcal{T}[1, i-1] \ : \ i \in \mathrm{PDA}_\pi \wedge \mathcal{T}[i] = c\}$. We store the z-fast tries in a map associating each $c \in \Sigma$ to the corresponding z-fast trie and supporting constant-time lookup queries. At this point, query $\mathrm{sufsearch}(i-j+1, i-1, c)$ is solved by issuing an internal suffix search query $\mathcal{T}[i-j+1, i-1]$ on the z-fast trie associated with character $c$. In order to return the result $[b, e] = \mathrm{sufsearch}(i-j+1, i-1, c)$, we store along the z-fast trie for character $c$ the number $\Delta_c = |\{i \in \mathrm{PDA}_\pi \ : \ \mathcal{T}[i] < c\}|$ of sampled text prefixes ending with a character being smaller than $c$. Letting $[b', e']$ be the colexicographic range (retrieved with the z-fast trie for $c$) of $\mathcal{T}[i-j+1, i-1]$ among the text prefixes $\{\mathcal{T}[1, i-1] \ : \ i \in \mathrm{PDA}_\pi \wedge \mathcal{T}[i] = c\}$, then the result of $\mathrm{sufsearch}(i-j+1, i-1, c)$ is $[b, e] = [b' + \Delta_c, e' + \Delta_c]$.

This implementation of sufsearch runs in $O(h \log m)$ time (respectively, I/O complexity), where $h$ is the running time (respectively, I/O complexity) of fingerprinting queries on the text oracle.

Observe that Algorithm 1 calls sufsearch once for every STPD path crossed while reaching $\text{locus}(P)$ from the suffix tree root. The number of crossed paths is upper-bounded by the node depth $d$ of $\text{locus}(P)$ in the suffix tree of $\mathcal{T}$. In the end (Line 9), Algorithm 1 compares the pattern with a text substring of length $m$. Using the latter implementation of sufsearch (z-fast tries), we obtain:

**Lemma 8.** *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 $\ell$ 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* $\text{locus}(P)$ *in the suffix tree of $\mathcal{T}$.*

For example, using the text oracle of Prezza [41] the I/O complexity of Algorithm 1 becomes $O(d \log m + m/B)$.

**Lemma 9.** *Algorithm 1 is correct and complete.*

*Proof.* We prove that the following invariant is maintained by the `while` loop: 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 $\pi(i-j+1)$. At the beginning, the invariant is clearly true by the way we choose $i$ and $j$ (in particular, $P[1, j-1]$ is the empty string).

We show that the invariant is maintained after one execution of the `while` loop, assuming that $P$ occurs in $\mathcal{T}$. If the condition of the `if` statement in Line 4 fails, then $\mathcal{T}[i] = P[j]$, we just increment $i$ and $j$ in Line 8, and the invariant still holds by the order-preserving property of $\pi$. Otherwise, let $\mathcal{T}[i] \neq P[j]$. Then, since $P$ occurs in $\mathcal{T}$, string $\alpha = P[1, j-1]$ is right-maximal in $\mathcal{T}$: there exists a copy of $\alpha$ followed by $P[j]$ and one followed by $\mathcal{T}[i] \neq P[j]$. By the inductive hypothesis, $P[1, j-1] = \mathcal{T}[i-j+1, i-1]$ minimizes $\pi(i-j+1)$ across all occurrences of $P[1, j-1]$. By the order-preserving property of $\pi$, $\mathcal{T}[i-j+1, i-1]$ also minimizes $\pi(i-1)$ across all occurrences of $P[1, j-1]$. Then, this means that $\pi(\alpha \cdot \mathcal{T}[i]) < \pi(\alpha \cdot P[j])$ (see Lemma 3 for the definition of this extension of $\pi$ to right-extensions of right-maximal strings) so, by Lemma 3, we have that there exists $t \in \text{PDA}_\pi$ such that $\mathcal{T}[1, t]$ is suffixed by $\alpha \cdot P[j] = P[1, j]$. But then, $\text{sufsearch}(i-j+1, i-1, P[j])$ (implemented with z-fast tries) is guaranteed to return the (non-empty) range $\text{PDA}_\pi[b, e]$ containing all and only the text prefixes in $\text{PDA}_\pi$ being suffixed by $P[1, j]$. Among those, Lines 6-7 choose the one $i \in \text{PDA}_\pi[b, e]$ minimizing $\pi(i)$. At this point, by the order-preserving property of $\pi$, we have that $\mathcal{T}[i-j+1, i] = P[1, j]$ is the occurrence of $P[1, j]$ minimizing $\pi(i-j+1)$. In Line 8 we increment $i$ and $j$, hence the invariant is maintained. This proves completeness and correctness under the assumption that $P$ occurs in $\mathcal{T}$.

If, on the other hand, $P$ does not occur in $\mathcal{T}$, then the comparison in Line 9 detects this event and we correctly return `NOT_FOUND`. This completes the proof. □

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

**Lemma 10.** *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 $O(|\text{PDA}_\pi|)$ 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 16 (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* $\mathrm{LPF}_\pi$ *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 5.* 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 since $\mathrm{LPF}_\pi[i] = 0$ then $c = \mathcal{T}[i]$ is the occurrence of $c$ minimizing $\pi(i)$, hence by the order-preserving property of $\pi$, $\mathcal{T}[i', i' + m - 1] = P$ is the occurrence of $P$ minimizing $\pi(i')$.

*Remark 6.* 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 $\pi$. We formalize this fact as follows:

**Definition 17 (Phrase source).** *Let $\mathcal{T}[i, i + \mathrm{LPF}_\pi[i] - 1]$ be a type-2 phrase. Then, we define* $\mathrm{SRC}[i] = j$ *to be the position $j$ with $\pi(j) < \pi(i)$ such that $\mathcal{T}[i, i+\mathrm{LPF}_\pi[i]-1] = \mathcal{T}[j, j+\mathrm{LPF}_\pi[i]-1]$ (in case of ties, choose the leftmost such position $j$).*

We also use the fact that each phrase can be split into a prefix containing strictly decreasing $\mathrm{LPF}_\pi$ 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 18 (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 7.* Observe that all $\mathrm{LPF}_\pi$ positions in $[i+1, i+\ell_i-1]$ are reducible (that is, not irreducible).

See Figure 3 for a running example.

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 19 (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 $\ell_i$ 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 $i$.*

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{T}[i]$ | a | b | a | a | b | b | a | a | b | a | a | b | a | a | a | b | a | b | a | b | \$ |
| LPF$[i]$ | 0 | 0 | $\boxed{1}$ | $\boxed{2}$ | 1 | $\boxed{4}$ | 3 | $\boxed{5}$ | $\boxed{6}$ | 5 | 4 | 3 | 2 | $\boxed{4}$ | 3 | 2 | $\boxed{4}$ | 3 | 2 | 1 | 0 |
| SRC$[i]$ | – | – | 1 | 1 |  | 2 |  | 1 | 6 |  |  |  |  | 7 |  |  | 15 |  |  |  | – |
| $I_1$ | **3** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| $I_2$ | **4** | **5** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| $I_3$ |  | $\boxed{\mathbf{6}}$ | $\boxed{\mathbf{7}}$ | 8 | 9 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| $I_4$ | **8** | $\boxed{9}$ | $\boxed{10}$ | 11 | 12 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| $I_5$ |  |  |  |  |  | $\boxed{\mathbf{9}}$ | $\boxed{\mathbf{10}}$ | **11** | $\boxed{\mathbf{12}}$ | $\boxed{\mathbf{13}}$ | 14 |  |  |  |  |  |  |  |  |  |  |
| $I_6$ |  |  |  |  |  |  | **14** | **15** | $\boxed{\mathbf{16}}$ | $\boxed{17}$ |  |  |  |  |  |  |  |  |  |  |  |
| $I_7$ |  |  |  |  |  |  |  |  |  |  |  |  |  |  | **17** | $\boxed{\mathbf{18}}$ | $\boxed{\mathbf{19}}$ | **20** |  |  |  |

**Fig. 3.** 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). For example, consider such a position $i = 9$. Since $\mathrm{LPF}[9] = 6$, the corresponding phrase is $\mathcal{T}[9, 9 + 6 - 1] = \mathcal{T}[9, 14]$. The corresponding LPF subarray is $\mathrm{LPF}[9, 14] = (\underline{6}, \underline{5}, \underline{4}, \underline{3}, \underline{2}, 4)$, where we underlined the phrase's reducible prefix (Definition 18) of length $\ell_9 = 5$. From the 5th to the 11th line ($I_1$—$I_7$) 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 (red). For example, consider again phrase $\mathcal{T}[9, 14]$, and let $s_9 = \mathrm{SRC}[9] = 6$ be its source. Row $I_5$ shows the phrase's source $\mathcal{T}[s_9, s_9 + \mathrm{LPF}[9] - 1] = \mathcal{T}[6, 11]$, highlighting in blue the source of the phrase's reducible prefix and in red the remaining suffix. In lines $I_1$—$I_7$, we moreover show, using boxes, 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. By the way we design the rectangles indexed in our orthogonal point enclosure data structure (see Definition 19 and Lemma 11 below), only occurrences in a blue box trigger the location of further secondary occurrences. **Example of locate**. After locating the primary occurrence $\mathcal{T}[2, 3] = ba$, an orthogonal point enclosure query on point $(2, 3)$ yields rectangle $[2, 3] \times [2, 5]$, corresponding to the phrase starting in position 6 (and whose source is shown in line $I_3$). This leads us to discover the secondary occurrence $\mathcal{T}[6, 7]$. Observe that, even if $\mathcal{T}[2, 3]$ overlaps also the source of the phrase starting in position 8, point $(2, 3)$ is not contained in the phrase's rectangle $[1, 1] \times [1, 5]$ (in Line $I_4$, this fact can be visualized by noting that the box lies completely in the red area). This is correct, as otherwise following this source would lead us locating secondary occurrence $\mathcal{T}[9, 10]$, which will be located again later when issuing the orthogonal point enclosure query on point $(6, 7)$.

*Example 3.* Continuing the running example of Figure 3, consider again 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 $[s_9, s_9+\ell_9-1] \times [s_9, s_9+\mathrm{LPF}[9]-1] = [6,10] \times [6,11]$. In Figure 3, Line $I_5$, the first coordinate $[6,10]$ is highlighted in blue, while the second coordinate $[6,11]$ is the whole colored interval in Line $I_5$ (red and blue).

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

**Lemma 11 (Orthogonal point enclosure, [10, 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 defined in Definition 19. As noted above, $|\mathcal{R}| \le |\mathrm{PDA}_\pi|$. We build the data structure of Lemma 11 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 pattern 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 such 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 $Q$.

*Correctness.* To prove that we only report pattern occurrences, we prove inductively that the stack always contains only pattern occurrences. At the beginning, the stack contains the primary occurrence of $P$. Assume inductively that, before entering in Step 1 of the above algorithm, the stack contains only pattern occurrences. In Step 1, we pop $x$. Then, let $[s_i, s_i+\ell_i-1] \times [s_i, s_i+\mathrm{LPF}_\pi[i]-1]$ be a rectangle, labeled with position $i$, found in Step 2 by querying point $(x, x+m-1)$. In particular, $[x, x+m-1] \subseteq [s_i, s_i+\mathrm{LPF}_\pi[i]-1]$. Then, by the definition of SRC (Definition 17), it holds $\mathcal{T}[i, i+\mathrm{LPF}_\pi[i]-1] = \mathcal{T}[s_i, s_i+\mathrm{LPF}_\pi[i]-1]$, hence $[i+x-s_i, i+x-s_i+m-1] \subseteq [i, i+\mathrm{LPF}_\pi[i]-1]$, therefore $\mathcal{T}[i+x-s_i, i+x-s_i+m-1] = \mathcal{T}[x, x+m-1]$ is a pattern occurrence. This proves that in Step 2 we only push pattern occurrences in $Q$.

*Completeness.* We prove that every pattern occurrence at some point is pushed in the stack (and is therefore located). Assume, for a contradiction, that this is not true. Let $\mathcal{T}[j, j+m-1]$ be the secondary occurrence minimizing $\pi(j)$ that is never pushed on the stack. Then, there exists a type-2 phrase $\mathcal{T}[i, i+\mathrm{LPF}_\pi[i]-1]$ entirely containing $\mathcal{T}[j, j+m-1]$, i.e. $[j, j+m-1] \subseteq [i, i+\mathrm{LPF}_\pi[i]-1]$. Without loss of generality, let $i$ be the largest position with such a property. Let $\ell_i$ be the length of the phrase's reducible prefix. Observe that $\mathcal{T}[j', j'+m-1] = \mathcal{T}[j, j+m-1]$, where $j' = s_i + (j-i)$, is another pattern occurrence with $\pi(j') < \pi(j)$. We distinguish two cases.

(i) $j \in [i, i+\ell_i-1]$. Then, the point $(j', j'+m-1)$ belongs to the rectangle $[s_i, s_i+\ell_i-1] \times [s_i, s_i+\mathrm{LPF}_\pi[i]-1]$. This means that also pattern occurrence $\mathcal{T}[j', j'+m-1]$ is never pushed on the stack, otherwise at some point the algorithm would pop $j'$ from the stack and locate $j$ as well. This is a contradiction, since either $j'$ is the primary occurrence, or $\pi(j') < \pi(j)$ and we assumed that $\mathcal{T}[j, j+m-1]$ is the secondary occurrence minimizing $\pi(j)$ that is never pushed on the stack.

(ii) $j \in [i+\ell_i, i+\mathrm{LPF}_\pi[i]-1]$. Then, either $i+\ell_i = n+1$ (a contradiction with the fact that $j \geq i+\ell_i$) or, by Definition 18, position $i' = i+\ell_i$ is irreducible with $\mathrm{LPF}_\pi[i'] > 0$. Then, $\mathcal{T}[i', i'+\mathrm{LPF}_\pi[i']-1]$ is a phrase with $i' > i$ and, as observed in Remark 1, it must be $i'+\mathrm{LPF}_\pi[i']-1 \geq i+\mathrm{LPF}_\pi[i]-1 \geq j+m-1$. This means that $\mathcal{T}[i', i'+\mathrm{LPF}_\pi[i']-1]$ is a type-2 phrase with $i' > i$ that contains $\mathcal{T}[j, j+m-1]$ (i.e. $[j, j+m-1] \subseteq [i', i'+\mathrm{LPF}_\pi[i']-1]$), a contradiction since we assumed that $i$ was the largest such position.

*Complexity.* In order to prove Lemma 10, we are left to show that every occurrence is pushed at most once in the stack, that is, we never report an occurrence twice. Assume, for a contradiction, that secondary occurrence $\mathcal{T}[j, j+m-1]$ is pushed at least twice on the stack. Without loss of generality, we can assume that $\mathcal{T}[j, j+m-1]$ is the secondary occurrence pushed at least twice on the stack that minimizes $\pi(j)$. This can happen in two cases, treated below.

(i) There exists *one* irreducible $\mathrm{LPF}_\pi$ position $i$, associated with rectangle $[s_i, s_i+\ell_i-1] \times [s_i, s_i+\mathrm{LPF}_\pi[i]-1] \in \mathcal{R}$, that causes pushing $j$ twice on the stack. This can happen only in one situation: when position $j' = j-i+s_i$ (uniquely determined from $i$ and $j$) is pushed twice on the stack, therefore querying twice the orthogonal point enclosure data structure on point $(j', j'+m-1)$ leads to pushing twice $j = i+j'-s_i$ in $Q$. In such a case, however, note that $\pi(j') < \pi(j)$, hence $\mathcal{T}[j', j'+m-1]$ is a secondary occurrence pushed at least twice on the stack with $\pi(j') < \pi(j)$. This contradicts the assumption that $\mathcal{T}[j, j+m-1]$ is the occurrence with this property minimizing $\pi(j)$.

(ii) There exist (at least) *two distinct* irreducible $\mathrm{LPF}_\pi$ positions $i_1 \neq i_2$, associated with rectangles $[s_{i_t}, s_{i_t}+\ell_{i_t}-1] \times [s_{i_t}, s_{i_t}+\mathrm{LPF}_\pi[i_t]-1] \in \mathcal{R}$ for $t = 1, 2$, respectively, each causing to push $j$ on the stack. Similarly to the case above, this happens when both positions $j_t = j-i_t+s_{i_t}$ (for $t = 1, 2$) are pushed on the stack, therefore querying the orthogonal point enclosure data structure on point $(j_t, j_t+m-1)$ leads to pushing $j = i+j_t-s_{i_t}$ in $Q$ for $t = 1, 2$. In particular, this means that $j \in [i_1, i_1+\ell_{i_1}-1] \cap [i_2, i_2+\ell_{i_2}-1]$, that is, $j$ belongs to the reducible prefix of both phrases. Assume, without loss of generality, that $i_1 < i_2$ (the other case is symmetric). Then, $i_1 < i_2 \leq j \leq i_1+\ell_{i_1}-1$: in other words, $i_2 \in [i_1+1, i_1+\ell_{i_1}-1]$. This is a contradiction, because all $\mathrm{LPF}_\pi$ positions $[i_1, i_1+\ell_{i_1}-1]$ are reducible (Remark 7), while $i_2$ is irreducible.

*Putting everything together.* Combining Lemmas 8, 9, and 10 we obtain:

**Theorem 3.** *Let $\mathcal{T} \in \Sigma^n$ be a text and $\pi : [n] \rightarrow [n]$ 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 $\ell$ 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 [41], the I/O complexity of Theorem 3 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.

### 3.3 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 4 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 $\pi$. 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 12.** *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, $|\texttt{st-colex}^-| \le \bar{r}$ and $|\texttt{st-colex}^+| \le \bar{r}$ hold.*

*Proof.* For every $i, j \in [n-1]$ such that $\mathcal{T}[1,i] <_{\text{colex}} \mathcal{T}[1,j]$ and $T[i+1] = T[j+1]$, it holds that $\mathcal{T}[1,i+1] <_{\text{colex}} \mathcal{T}[1,j+1]$ by definition of the colexicographic order. This proves the order-preserving property for $\pi$ and $\bar{\pi}$.

In order to upper-bound $|\texttt{st-colex}^-|$ and $|\texttt{st-colex}^+|$, we rotate $\mathcal{T}$ to the right by one position, i.e. we replace $\mathcal{T}$ by $\mathcal{T}[n]\cdot\mathcal{T}[1,n-1] = \$\cdot\mathcal{T}[1,n-1]$. By Remark 8.4, this rotation does not change the number of coBWT equal-letter runs. The number of irreducible positions in $\mathrm{LPF}_\pi$ and $\mathrm{LPF}_{\bar{\pi}}$, on the other hand, increases at most by one: since \$ is the smallest alphabet character, the colexicographic order of prefixes does not change after the rotation. Then, $i$ was irreducible before the rotation if and only if $i+1$ is irreducible after the rotation. Additionally, $i = 1$ is always irreducible after the rotation. It follows that if we prove the bound for the rotated $\mathcal{T}$, then the bound also holds for the original one.

By Remark 2, $|\texttt{st-colex}^-|$ is equal to the number of irreducible $\mathrm{LPF}_\pi$ positions. Let $i' = i + \mathrm{LPF}_\pi[i] - 1$. We map bijectively each irreducible $\mathrm{LPF}_\pi$ position $i$ to $\mathrm{coBWT}[\mathrm{IPA}[i']]$ and show that $\mathrm{coBWT}[\mathrm{IPA}[i']]$ is the beginning of an equal-letter coBWT run. This will prove $|\texttt{st-colex}^-| \le \bar{r}$. Symmetrically, to prove $|\texttt{st-colex}^+| \le \bar{r}$ we map bijectively each irreducible $\mathrm{LPF}_{\bar{\pi}}$ position $i$ to $\mathrm{coBWT}[\mathrm{IPA}[i']]$ and show that $\mathrm{coBWT}[\mathrm{IPA}[i']]$ is the end of an equal-letter coBWT run. Since this case is completely symmetric to the one above, we omit its proof.

Let $i$ be an irreducible $\mathrm{LPF}_\pi$ position and let $i' = i + \mathrm{LPF}_\pi[i] - 1$ ($\le n$, by definition of $\mathrm{LPF}_\pi$). We analyze separately the cases (i) $i' = n$ and (ii) $i' < n$.

(i) If $i' = n$, then $\mathrm{coBWT}[\mathrm{IPA}[i']] = \$$. Since the symbol \$ occurs only once in $\mathcal{T}$, $\mathrm{coBWT}[\mathrm{IPA}[i']]$ is the beginning of an equal-letter BWT run.

(ii) If $i' < n$, then either $\mathrm{IPA}[i'] = 1$ and therefore $\mathrm{coBWT}[\mathrm{IPA}[i']] = \mathrm{coBWT}[1]$ is the beginning of an equal-letter coBWT run, or $\mathrm{IPA}[i'] > 1$. In the latter case, let $j' = \mathrm{PA}[\mathrm{IPA}[i'] - 1]$ (in particular, $\mathrm{IPA}[j'] = \mathrm{IPA}[i'] - 1$) and $j = j' - \mathrm{LPF}_\pi[i] + 1$. If $j' = n$, then $\mathrm{coBWT}[\mathrm{IPA}[i'] - 1] = \mathrm{coBWT}[\mathrm{IPA}[j']] = \$ \neq \mathrm{coBWT}[\mathrm{IPA}[i']]$, hence again we have that $\mathrm{coBWT}[\mathrm{IPA}[i']]$ is the beginning of an equal-letter BWT run. In the following, we can therefore assume both $i' < n$ and $j' < n$.

Observe that it must be the case that $\mathcal{T}[j, j'] = \mathcal{T}[i, i']$. To see this, let $t$ be such that $\pi(t) < \pi(i)$ and $\mathrm{LPF}_\pi[i] = \mathrm{rlce}(i, t)$. Let $t' = t + \mathrm{LPF}_\pi[i] - 1$. Then, by the order-preserving property of $\pi$, it must be $\pi(t') < \pi(i')$. Moreover, since $\mathrm{LPF}_\pi[i] = \mathrm{rlce}(i, t)$, then $\mathrm{llce}(i', t') \ge \mathrm{LPF}_\pi[i]$. But then, by definition of $j'$ and $\pi = \mathrm{IPA}$, it must be $\mathrm{llce}(i', j') \ge \mathrm{llce}(i', t') \ge \mathrm{LPF}_\pi[i]$, which implies $\mathcal{T}[j, j'] = \mathcal{T}[i, i']$.

Then, since $\pi(j') = \pi(i') - 1 < \pi(i')$, by the order-preserving property of $\pi$ it must be $\pi(j) < \pi(i)$. Finally, by the very definition of $\mathrm{LPF}_\pi$ it must be the case that $\mathrm{coBWT}[\mathrm{IPA}[i'] - 1] = \mathrm{coBWT}[\mathrm{IPA}[j']] = \mathcal{T}[j'+1] \neq \mathcal{T}[i'+1] = \mathrm{coBWT}[\mathrm{IPA}[i']]$, which proves the claim. In fact, if it were $\mathcal{T}[j'+1] = \mathcal{T}[i'+1]$ then we would obtain $\mathrm{LPF}_\pi[i] \ge \mathrm{rlce}(i, j) \ge \mathrm{LPF}_\pi[i] + 1$, a contradiction. □

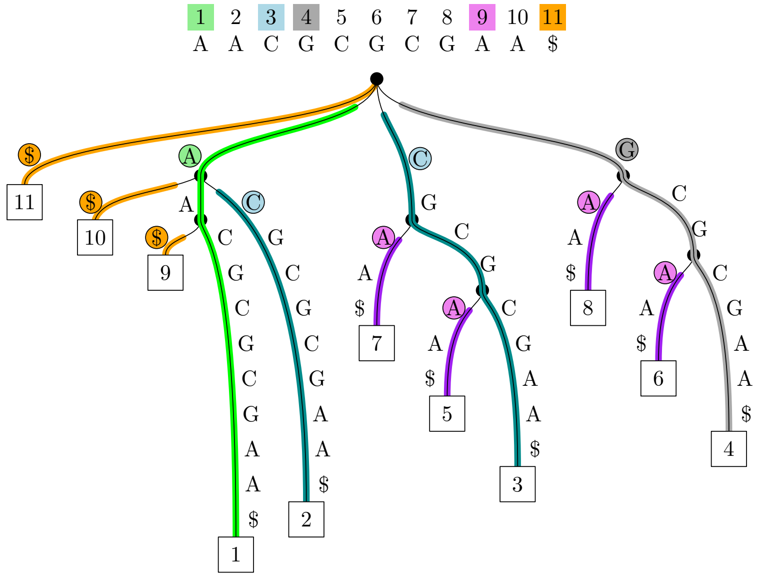


**Fig. 4.** Example of the STPD obtained by taking $\pi = \text{IPA}$ to be the colexicographic rank of the text's prefixes. For a better visualization, we do not sort suffix tree leaves according to $\pi$ as this would cause some edges to cross. Paths $\mathcal{T}[j,n]$ are colored according to the color of $j \in$ `st-colex`$^-$. To see how the paths are built, consider the Prefix Array $\text{PA} = [11, 1, 2, 10, 9, 3, 5, 7, 4, 6, 8]$ and the generalized Longest Previous Factor array $\text{LPF}_\pi = [0, 1, 0, 0, 4, 3, 2, 1, 2, 1, 0]$. PA tells us the order in which we have to imagine inserting the text's suffixes in the trie. We show how the process works for the first three suffixes in the order induced by $\pi$. The first suffix to be inserted is $\mathcal{T}[11, 11]$. Position $11 + \text{LPF}_\pi[11] = 11 + 0 = 11$ is orange, hence an orange path labeled $\mathcal{T}[11, 11]$ starts in the root. The next suffix is $\mathcal{T}[1, 11]$. Position $1 + \text{LPF}_\pi[1] = 1 + 0 = 1$ is green, hence a green path labeled $\mathcal{T}[1, 11]$ is inserted. The next suffix is $\mathcal{T}[2, 11]$. This suffix shares a common prefix of length $\text{LPF}_\pi[2] = 1$ with the previously-inserted suffix $\mathcal{T}[1, 11]$, hence position $2 + \text{LPF}_\pi[2] = 2 + 1 = 3$ is sampled. This position is blue, hence a blue path labeled $\mathcal{T}[3, 11]$ is inserted as a child of $\text{locus}(\mathcal{T}[2] = A)$.

We now show how to locate the pattern's occurrences with `st-colex`$^-$. In this section, among the various trade-offs summarized in Theorem 3 for general order-preserving permutations, we will analyze the one obtained by assuming a text oracle supporting the extraction of $\ell$ contiguous text characters with $O(1+\ell/B)$ I/O complexity as it models the scenario of our practical implementation discussed in Section 3.2.

**Finding the primary occurrence.** For the particular order-preserving permutation $\pi = \text{IPA}$ 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 $\pi = \text{IPA}$ (colexicographic rank) and $\text{PDA}_\pi =$ `st-colex`$^-$ is sorted colexicographically, observe that $\pi(\text{PDA}_\pi) = \pi($`st-colex`$^-)$ is increasing. This means that, in Line 6 of Algorithm 1, it always holds $\arg\min_{i' \in [b,e]}\{\pi(\text{PDA}_\pi[i'])\} = b$ and therefore we do not need a Range Minimum Data structure over $\pi($`st-colex`$^-)$.

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

**Lemma 13.** *Let $\mathcal{T} \in \Sigma^n$ be a text, and fix $\pi = \text{IPA}$. Suppose we have access to a text oracle supporting the extraction of $\ell$ contiguous characters with $O(1 + \ell/B)$ I/O complexity. Algorithm 1 requires just array* `st-colex`$^-$*, fitting in* $|$`st-colex`$^-| \le \bar{r}$ *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* $\mathcal{T}$.

**Locating the secondary occurrences.** As it turns out, the mechanism for locating secondary occurrences described in Section 3.2 in the particular cases of $\pi = \text{ISA}$ and $\pi = \text{IPA}$ is equivalent to the $\bar{\phi}$ function of Definition 11, with the only additional detail that for $\pi = \text{IPA}$ one should replace in Definition 11 SA and ISA with PA and IPA, respectively (below, with symbol $\bar{\phi}$ we denote this variant using PA and IPA).

Then, our locating mechanism essentially reduces to the one of the $r$-index [19], with only minor modifications that we describe below. The $r$-index [19] (of the reversed $\mathcal{T}$) locates pattern occurrences as follows. Let $\text{PA}[b, b + occ - 1]$ be the Prefix Array range of pattern $P$. The $r$-index (i) finds $\text{PA}[b]$ and $occ$ with a mechanism called the *toehold lemma*, and (ii) it applies $occ - 1$ times function $\bar{\phi}$ starting from $\text{PA}[b]$. This yields the sequence $\text{PA}[b], \text{PA}[b + 1], \ldots, \text{PA}[b + occ - 1]$, that is, the list of all pattern's occurrences. As shown by Nishimoto and Tabei [40], function $\bar{\phi}$ (in the variant using PA and IPA) can be stored in $O(\bar{r})$ memory words in such a way that the above $occ - 1$ successive evaluations of $\bar{\phi}$ take $O(occ + \log \log(n/\bar{r}))$ time (and I/O complexity).

Differently from the toehold lemma of the $r$-index, Algorithm 1 only allows to compute $\text{PA}[b]$ (not $occ$). To also compute $occ$ and output all pattern's occurrences we proceed as follows.

The simplest solution to locate all occurrences is to use both arrays $\texttt{st-colex}^-$ and $\texttt{st-colex}^+$ to compute $\text{PA}[b]$ and $\text{PA}[b + occ - 1]$, respectively, and then extract one by one $\text{PA}[b], \text{PA}[b + 1], \ldots, \text{PA}[b + occ - 1]$ by applying $occ - 1$ times function $\bar{\phi}$. This solution, however, requires also storing $\texttt{st-colex}^+$.

We can still locate efficiently all secondary occurrences using just $\texttt{st-colex}^-$, as follows. Let $\text{PA}[b, b + occ - 1]$ be the range of $P$ in the Prefix Array. Let $T = \lceil m/B \rceil$ be a *block size*. Partition the range $[b, b + occ - 1]$ into $q = \lfloor occ/T \rfloor$ equal-size blocks of size $T$, i.e. $[b + (i - 1)T, b + i \cdot T - 1]$ for $i = 1, \ldots, q$, plus (possibly) a last block of size $occ \mod T$. For $i = 1, \ldots, q$, reconstruct $\text{PA}[b + (i - 1)T, b + i \cdot T]$ by applying the $\bar{\phi}$ function $T$ times starting from $\text{PA}[b + (i - 1)T]$ (available from the previous iteration). Note that we do not know $q$ (since we do not know $occ$); in order to discover when we reach the last block of size $T$ (i.e. the $q$-th block), after reconstructing $\text{PA}[b + (i - 1)T, b + i \cdot T]$ we compare $\mathcal{T}[i', i' + m - 1] \overset{?}{=} P$ for $i' = \text{PA}[b + i \cdot T - 1]$ (i.e., the last Prefix Array entry in the block). Then, we know that $i \le q$ if and only if this equality test succeeds. Since we realize that we reached block number $i = q$ only when extracting the $(q + 1)$-th block of length $T$, this means that the total number of applications of function $\bar{\phi}$ is bounded by $occ + T \le occ + m/B + 1$ (constant time each, except the very first $\bar{\phi}(\text{PA}[b])$, costing $O(\log \log(n/\bar{r}))$).

As far as the $q + 1$ comparisons $\mathcal{T}[i', i' + m - 1] \overset{?}{=} P$ are concerned, if the random access oracle supports the extraction of $\ell$ contiguous characters of $\mathcal{T}$ with $O(1 + \ell/B)$ I/O complexity, then the I/O cost of these comparisons amounts to $O((1 + occ/T) \cdot (1 + m/B)) = O(1 + occ + m/B)$.

Note that, when incrementing the block number $i$, we can re-use the same memory ($T$ words) allocated for $\text{PA}[b + (i - 1)T, b + i \cdot T]$ in order to store the new block of $T$ Prefix Array entries. It follows that the above process uses in total $T \le 1 + m/B$ memory words of space.

If $rem = occ \mod T > 0$, we are left to show how to reconstruct the last block $\text{PA}[b + q \cdot T, b + q \cdot T + rem - 1] = \text{PA}[b', e']$, of length $rem$. The idea is to simply use binary search on $\text{PA}[b', b' + T - 1]$, which we already extracted in the $(q + 1)$-th iteration. In each of the $O(\log T)$ binary search steps, we compare $P$ with a substring of $\mathcal{T}$ of length $m$. If the random access oracle supports the extraction of $\ell$ contiguous characters of $\mathcal{T}$ with $O(1 + \ell/B)$ I/O complexity, finding this last (partial) block of $rem = occ \mod T$ pattern occurrences via binary search costs $O((1 + m/B) \cdot \log T) = O((1 + m/B)\lceil \log(1 + m/B)\rceil)$ I/O complexity using $T$ memory words of space.

**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 13, we obtain:

**Theorem 4.** *Let $\mathcal{T} \in \Sigma^n$ be a text. Assume we have access to a text oracle supporting the extraction of $\ell$ 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 $O\left((d \log \bar{r} + \log(1 + m/B)) \cdot (1 + m/B) + occ + \log\log(n/\bar{r})\right)$ I/O complexity 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 $|\texttt{st-colex}^-| \le \bar{r}$ words) and Nishimoto and Tabei's data structure [40] storing function $\bar{\phi}$ (less than $2\bar{r}$ words using the optimized implementation of [5]).

### 3.4 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 4. We call the corresponding path decomposition array $\texttt{st-pos}^-$. The corresponding STPD is almost equal to the one in Figure 4, with the only difference being that the path starting in the root is $\mathcal{T}[1, 11]$ (green) rather than $\mathcal{T}[11, 11]$ (orange). In this particular example, the two STPD obtained using $\pi = id$ and $\pi = \mathrm{IPA}$ are essentially equal because (as the reader can easily verify) the colexicographically-smallest occurrence of any substring $\alpha$ of $\mathcal{T}$ is also the leftmost one (but of course, this is not always the case for any text $\mathcal{T}$).

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[8], below we show that these measures are worst-case optimal, meaning that for every $p \ge 1$ we exhibit a family of strings with $p = \Theta(|\texttt{st-pos}^-|)$ requiring $O(p)$ words to be stored in the worst case.

Moreover, in Section 4 we show that in practice $|\texttt{st-pos}^-|$ and $|\texttt{st-pos}^+|$ are consistently smaller than $r$.

**Theorem 5.** *For any integers $n \ge 1$ and $p \in [n]$, there exists a string family $\mathcal{F}$ of cardinality $\binom{n}{p}$ such that every string $\mathcal{S} \in \mathcal{F}$ is over alphabet of cardinality $\sigma = p+1$ and satisfies $|\texttt{st-pos}^-(\mathcal{S})| = p + 1$ and $|\mathcal{S}| \le np$. 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 $|\texttt{st-pos}^+|$.*

[8] The techniques used by Kempa and Kociumaka [23] to prove that the number of irreducible PLCP values is bounded by $O(\delta \log^2 n)$, do not apply to array LPF.

*Proof.* 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, it is not hard to see that the LPF irreducible positions are 1, 2, and $i+1$ for all $i < n$ such that $\mathcal{S}[i] > 0$ (in total, $p+1$ irreducible positions). This proves $|\texttt{st-pos}^-(\mathcal{S})| = p+1$. Fix $n \geq 1$ and $p \in [n]$. Then, the family $\mathcal{F}_{n,p}$ of strings built as above contains $\binom{n}{p}$ elements and satisfies the conditions of the claim.

The proof for $\texttt{st-pos}^+$ is symmetric: just sort the set's elements in increasing order $\{x_1 < x_2 < \cdots < x_p\} \subseteq [n]$ and build $\mathcal{S}$ as above. Then, the irreducible positions are $i = 1, n$, and $i+1$ for all $i < n$ such that $\mathcal{S}[i] > 0$ (in total, $p+1$ irreducible positions). □

Theorem 5 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 [26] for the repetitiveness measure $\delta$. Since by Theorem 2 $O(|\texttt{st-pos}^-|\log(|\mathcal{S}|\sigma))$ bits (in the theorem above, $O(p\log(np\sigma)) = O(p\log(n\sigma))$ 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.

**Applications of `st-pos`.** By running Algorithm 1 on arrays $\texttt{st-pos}^-$ and $\texttt{st-pos}^+$, one can quickly locate the leftmost and rightmost occurrences of $P$ in $\mathcal{T}$. This is a problem that is not easily solvable with the $r$-index (unless using a heavy context-free grammar of $O(r\log(n/r))$ words supporting range minimum queries on the Suffix Array — see [19]). We briefly discuss an application of `st-pos` and observe interesting connections between this STPD, suffix tree construction, and the *Prediction by Partial Matching* compression algorithm.

*Taxonomic classification of DNA fragments.* The STPD discussed in this section finds an important application in taxonomic classification algorithms, which we plan to explore in a future publication. Imagine having a collection of genomes, organized in a phylogenetic tree. The taxonomic classifiers Kraken [49] and Kraken 2 [48] index the genomes in a given phylogenetic tree such that, given a DNA read (i.e., a short string over the alphabet $\{A, C, G, T\}$), they can map each $k$-mer (substring of length $k$) in that read to the root of the smallest subtree of the phylogenetic tree containing all the genomes containing that $k$-mer. Considering the roots for all the $k$-mers in the read, the tools then tries to predict from what part of the tree the read came from (with a smaller subtree corresponding to a more precise prediction). The mapping is fairly easy with $k$-mers but becomes more challenging with variable-length substrings such as maximal exact matches (MEMs), for example.

A first step to generalizing the strategy to variable-length strings has been taken in [17]. In that paper, the authors concatenate the genomes in leaf order and index the resulting string with a structure able to return the leftmost and rightmost occurrence of all the $k$-mers of an input string, where (differently from [48,49]) $k$ is provided at query time rather than index construction time. At this point, for each $k$-mer a constant-time lowest-common-ancestor (LCA) query on the (at most) two genomes (tree nodes) containing those occurrences, yields the smallest subtree containing the $k$-mer.

Using suffix tree path decompositions, we can easily extend this solution to any subset of strings of the query string (e.g. the set of al MEMs). Once we have a MEM, with $\texttt{st-pos}^-$ and $\texttt{st-pos}^+$ we can find the MEM's leftmost and rightmost occurrences, the genomes that contain those occurrences (with a simple predecessor data structure), and those genomes' LCA — which is the root of the smallest subtree of the phylogenetic tree containing all occurrences of the MEM. This means $\texttt{st-pos}^-$ and $\texttt{st-pos}^+$ finally offer the opportunity to generalize Kraken and Kraken

2's approach to taxonomic classification to work *efficiently* (fast query times and fully-compressed space) with arbitrary-length strings such as MEMs.

*Relation to Ukkonen's suffix tree construction algorithm.* We observe the following interesting relation between the irreducible values in LPF and Ukkonen's suffix tree construction algorithm [46]. Ukkonen's algorithm 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 per iteration like McCreight's algorithm [35], Ukkonen's algorithm inserts a variable number of suffixes per iteration.

Consider the beginning of the $i$-th iteration, for $i = 1, \ldots, n$. The algorithm maintains a variable $j \in [n]$, indicating that we have already inserted suffixes $\mathcal{T}[1, n], \cdots \mathcal{T}[j-1, n]$ and we are at the locus of $\mathcal{T}[j, i-1]$ on the suffix tree. We have two cases. (i) If there already exists a locus corresponding to $\mathcal{T}[j, i]$, i.e., if there exists an out-edge labeled with $\mathcal{T}[i]$ from the current locus, we just walk down the tree accordingly taking child $\mathcal{T}[i]$. (ii) If there is no locus for $\mathcal{T}[j, i]$ in the tree, then we create a path labeled with $\mathcal{T}[i, n]$ (if $\mathcal{T}[j, i-1]$ ends in the middle of the label of a suffix tree edge, we also create an internal node and a suffix link to this new internal node if necessary), meaning that we are inserting the leaf that corresponds to $\mathcal{T}[j, n]$, then move to the locus for $\mathcal{T}[j+1, i-1]$ (following a suffix link), increment $j$ by 1 and repeat (ii) until we fall into Case (i) or until $j > i$.

When inserting a suffix $\mathcal{T}[j, n]$ during the $i$-th iteration, we are on the locus of $\mathcal{T}[j, i-1]$, meaning that there exists a suffix $\mathcal{T}[j', n]$ with $j' < j$ that is prefixed by $\mathcal{T}[j, i-1]$, but none of the suffixes $\mathcal{T}[j', n]$ with $j' < j$ is prefixed by $\mathcal{T}[j, i]$, which implies that $\mathrm{LPF}[j] = (i-1) - j + 1 = i - j$. Note that if more than one suffix is inserted in the same iteration $i$, they must be consecutive in terms of starting positions, i.e., those are $\mathcal{T}[j, n], \mathcal{T}[j+1, n], \cdots, \mathcal{T}[j+k, n]$ for some $k \in [n]$. Note that such suffixes form a decreasing run in LPF since $\mathrm{LPF}[j + k'] = i - j - k'$ for $0 \le k' \le k$. Moreover, observe that $(j + k') + \mathrm{LPF}[j + k'] = i$ for $0 \le k' \le k$, which implies they all those suffixes contribute to a single element $i \in \texttt{st-pos}^-$. As a result, $\texttt{st-pos}^-$ can be interpreted as the set of iterations in Ukkonen's algorithm in which at least one leaf is inserted.

*Relation with PPM*.* To conclude the section, we observe that $\texttt{st-pos}^-$ is tightly connected with the *Prediction by Partial Matching* compression algorithm. This connection indicates why this STPD achieves very good compression in practice (see experimental results in Section 4).

Consider how we decompose the suffix tree for $\mathcal{T}[1, n]$ when building $\texttt{st-pos}^-$. If we choose a path from a node $u$ to a leaf labeled with SA entry $i$, then either

1. $u$ is the root, $i = 1$, the path's label is all of $\mathcal{T}$, and we add 1 to $\texttt{st-pos}^-$; or
2. $u$ is already in a path to a leaf labelled $h < i$.

We focus on the second case. Let $\ell > 0$ be the length of $u$'s path label,

$$\mathcal{T}[h, h + \ell - 1] = \mathcal{T}[i, i + \ell - 1]\,,$$

so we add $i + \ell$ to $\texttt{st-pos}^-$.

Notice $\mathcal{T}[i, i+\ell-1]$ occurs starting at position $h < i$ but $\mathcal{T}[i, i+\ell]$ is the first occurrence of that substring. Let $i' \le i < i + \ell$ be the minimum value such that $\mathcal{T}[i', i+\ell-1]$ occurs in $\mathcal{T}[1, i+\ell-2]$; then $\mathcal{T}[i', i+\ell]$ does not occur in $\mathcal{T}[1, i+\ell-1]$. Setting $j = i + \ell$, we obtain the following corollary:

**Corollary 6.**

$$\texttt{st-pos}^- \subseteq \left\{ j \;:\; \begin{array}{l} \textit{either } j = 1 \textit{ or, for the minimum value } i' < j \\ \textit{such that } \mathcal{T}[i', j-1] \textit{ occurs in } \mathcal{T}[1, j-2], \\ \mathcal{T}[i', j] \textit{ does not occur in } \mathcal{T}[1, j-1] \end{array} \right\}.$$

*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 $c$. Here we consider the simple version of PPM* lengths [13] shown in Algorithm 2. When all previous occurrences of $\alpha$ are never followed by $c$ (that is, the model cannot assign a nonzero probability for character $c$ given context $\alpha$), the algorithm outputs a special *escape* symbol $\perp$ followed by character $c$.

Consider the set of positions at which we emit copies of the escape symbol $\perp$ when encoding $\mathcal{T}$ with Algorithm 2. By inspection of the algorithm, that set is the same as the one on the right side of the inequality in Lemma 6. We conclude:

**Lemma 14.** $|\texttt{st-pos}^-|$ *is upper-bounded by the number of escape symbols* $\perp$ *output by the PPM* algorithm described in [13] (Algorithm 2).*

**Algorithm 2:** A simple version of PPM*

```
Encode $\mathcal{T}[1]$ as $\perp\mathcal{T}[1]$;
for $j = 2 \dots n$ do
    $i' \leftarrow \arg\min_{i'} \{i' \leq j : \mathcal{T}[i', j-1]$ occurs in $\mathcal{T}[1, j-2]\}$;
    if $\mathcal{T}[i', j]$ occurs in $\mathcal{T}[1, j-1]$ then
        Encode $\mathcal{T}[j]$ according to the distribution of characters immediately following occurrences of
          $\mathcal{T}[i', j-1]$;
    else
        Encode $\mathcal{T}[j]$ as $\perp\mathcal{T}[j]$;
```

## 4 Preliminary experimental results

We show some preliminary experimental results on repetitive genomic collections. In Subsection 4.1 we show that the number of irreducible LCP values is, in practice, much smaller than $r$. Subsection 4.2 is dedicated to showing that (as expected from Remark 1) the suffix tree has excellent locate performance compared to the $r$-index. In subsection 4.3 we show that, in addition to being very small, our index is also orders of magnitude faster than the $r$-index (especially as the pattern's length $m$ increases).

### 4.1 $|\mathbf{PDA}_\pi|$ in practice

We begin with a few numbers (Table 1) showing how $|\mathrm{PDA}_\pi|$ compares in practice with $n$, $r$, and $\bar{r}$, for $\pi = \mathrm{ISA}$ ($|\texttt{st-lex}^-|$), $\pi = \mathrm{IPA}$ ($|\texttt{st-colex}^-|$), and $\pi = id$ ($|\texttt{st-pos}^-|$). For this experiment, we used the corpus https://pizzachili.dcc.uchile.cl/repcorpus.html of repetitive DNA collections (this repository became a standard in the field of compressed data structures). While only being representative of a small set of data, we find it interesting to observe that on these datasets the number of irreducible LCP values ($|\texttt{st-lex}^-|$ and $|\texttt{st-colex}^-|$) is consistently smaller than the number of BWT runs ($r$ and $\bar{r}$). From a practical perspective, this means that it is preferable to use data structures whose space is proportional to $|\texttt{st-colex}^-|$ rather than $r$. The same holds true for the number of irreducible LPF values ($|\texttt{st-pos}^-|$).

|  | influenza | cere | escherichia |
|---|---|---|---|
| $n$ | 154808556 | 461286645 | 112689516 |
| $\lvert\texttt{st-lex}^-\rvert$ | 1815861 | 7455556 | 9817075 |
| $\lvert\texttt{st-colex}^-\rvert$ | 1805730 | 7455980 | 9825972 |
| $\lvert\texttt{st-pos}^-\rvert$ | 1928408 | 8954851 | 11677573 |
| $r$ | 3022822 | 11574641 | 15044487 |
| $\bar{r}$ | 3018825 | 11575583 | 15045278 |

**Table 1.** Experimental evaluation of Path Decomposition Array sizes on repetitive collections from http://pizzachili.dcc.uchile.cl/repcorpus.html.

### 4.2 I/O complexity of classic versus compressed indexes

We compared custom implementations of the suffix tree and Suffix Array against the original $r$-index's implementation (https://github.com/nicolaprezza/r-index) on a small dataset in order to confirm the caching effects discussed in Section 1. All the experiments in this and in the following section have been run 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.

Due to the large space usage of the suffix tree, in this experiment we tested a text of limited length $n = 10^8$ formed by variants of the SarsCoV2 genome downloaded from the Covid-19 data portal (www.covid19dataportal.org/). The goal of the experiment was to verify experimentally the size and I/O complexity of the suffix tree, Suffix Array, and $r$-index:

- Suffix tree (Remark 1): $O(n)$ words of space, $O(d + m/B)$ I/O complexity for finding one occurrence, $O(occ)$ I/O complexity to find the remaining $occ - 1$ occurrences.
- Suffix array: $n$ words of space on top of the text, $O((1 + m/B) \log n)$ I/O complexity for finding one occurrence, $O(occ/B)$ I/O complexity to find the remaining $occ - 1$ occurrences.
- $r$-index: $O(r)$ words of space, $\Omega(m)$ I/O complexity for finding one occurrence, $\Omega(occ)$ I/O complexity to find the remaining $occ - 1$ occurrences.

The experiment consisted in running locate queries on $10^5$ patterns of length $m = 100$, all extracted from random text positions (in particular, $occ \geq 1$ for every pattern). The average number of occurrences per pattern was 2896 (the dataset is very repetitive). For each pattern, we measured the running time of locating the first occurrence and that of locating all occurrences.

To no surprise, due to the highly repetitive nature of the dataset, the $r$-index's size was of just 0.9 MiB, orders of magnitude smaller than the Suffix Array (0.3 GiB) and of the suffix tree (9.9 GiB). As expected, however, the poor cache locality of pattern matching queries on the $r$-index makes it orders of magnitude slower than those two classic data structures. As far as locating the primary occurrence (finding the pattern's locus in the suffix tree) was concerned, the $r$-index took 572 $ns$ per pattern's character on average. This is not far from the 200 $ns$ per character of the Suffix Array (where each binary search step causes a cache miss — an effect that gets diluted as $m$ increases), but one order of magnitude larger than the 60 $ns$ per character of the suffix tree. This also indicates that the term $d$ (node depth of the pattern's locus in the suffix tree) has a negligible effect in practice.

As far as locating the remaining $occ - 1$ occurrences was concerned, the $r$-index took 105 $ns$ per occurrence on average to perform this task. This is five times larger than the 21 $ns$ per character of the suffix tree; while both trigger at least one cache miss for each located occurrence, the $r$-index executes a predecessor query per occurrence, which in practice translates to several cache misses[9]. As expected, the Suffix Array solved this task in 3.4 $ns$ per occurrence. This confirms its $O(occ/B)$ I/O complexity for outputting the occurrences after locating the Suffix Array range.

### 4.3 Comparing `st-colex`$^-$ with the state-of-the-art

Figures 5 and 6 show some preliminary results comparing our index based on `st-colex`$^-$ (Section 3.3) with the $r$-index [19], move-r [5], and the Suffix Array on the task of locating all occurrences of a set of patterns in a text of length $n \approx 10^9$. The suffix tree was excluded from this experiment due to its size (it did not fit in the RAM of our workstation). Next, we discuss some implementation choices made in our practical index. Then, we describe the details of the performed experiments. Finally, we conclude with some comments.

*Implementation details.* The index follows the design of Section 3.3, except for a few details and optimizations.

First, due to a smaller space usage and good performance in practice, we decided to use the original $r$-index $\bar{\phi}$-function implementation, locating each occurrence in $O(\log(n/r))$ time (rather than Nishimoto and Tabei's move structure [40]). This is reflected in the fact that, for large $occ$ (Figure 6, pattern length 30), we locate pattern occurrences slower than move-$r$ [5] (an optimized implementation of the move structure [40], which in turn improves the locating mechanism of the $r$-index), which however uses much more space. Importantly, the locating mechanism of move-$r$ can be seamlessly integrated in our index: this gives a space-time trade-off that we discuss below.

As random access oracle, we used an ad-hoc implementation of Relative Lempel Ziv [29], yielding excellent compression and random access cache locality on repetitive collections of genomes. While relative Lempel-Ziv cannot formally guarantee $O(\ell/B)$ I/O operations for extracting a contiguous subsequence of $\ell$ characters, in practice it gets close to this performance on repetitive collections.

Finally, we implemented two heuristics speeding up function sufsearch (see Algorithm 1): (i) we tabulate the results of short suffixes of length at most $k_{max}$ (where $k_{max}$ is chosen so that the table uses a negligible amount of space on top of the index itself), and (ii) we start Algorithm 1 from $j = k_{max} + 1$, choosing the value of $i$ returned by the table on $P[1, k_{max}]$. Heuristics (i) and (ii) allow avoiding binary search in most cases and dramatically speed up Algorithm 1 in practice.

*Datasets and experiments.* 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 from the text. For each pattern, we measured the resources (peak memory and running time per character/occurrence) used by all the indexes while finding one pattern occurrence (Figure 5) and locating all pattern occurrences (Figure 6).

*Comments.* On this dataset, our index is smaller than the $r$-index, about four times smaller than move-$r$, and orders of magnitude smaller than the Suffix Array.

As expected, Figure 5 (*find* queries) shows that our index is always orders of magnitude faster than the $r$-index on *find* queries, and even faster than the Suffix Array (the latter result is due to the optimizations above described). As expected, move-$r$ is only slightly slower than our index for $m = 30$, becoming orders of magnitude slower for longer patterns. This confirms the fact that

[9] This gap was recently closed by the move structure of Nishimoto and Tabei [40]. On this dataset, the move-$r$ data structure [5] was able to locate each occurrence in 14 $ns$ on average.

move-$r$ incurs $O(m)$ cache misses while solving *find* queries. With our index, on the other hand, the query time per character gets smaller as the pattern length increases. This is because, often, a short pattern prefix suffices to find an STPD path that continues with all the remaining pattern's characters (matched very quickly on our random access oracle).

Figure 6 (*locate* queries), on the other hand, indicates (as expected) that when $occ \gg m$ the performance of our index gets close to that of the $r$-index and worse than that of move-$r$. This is the case for $m = 30$, where each pattern occurred on average 300 times in the (repetitive) text. Longer patterns occurred less frequently[10] and, again, in those regimes our index outperforms all the others in both query time and space usage.

The locating mechanism of move-$r$ can be seamlessly integrated in our index (since it is just a way to speed up computing the $\bar{\phi}$ function of Definition 11, which stands at the core of our locating mechanism). This yields (at the same time) the same *find* performance of our index, the same *locate* time per occurrence of the move-$r$ data structure, and about the same space usage of the move-$r$ data structure (about four times larger than our index). Reducing the space of the move data structure of Nishimoto and Tabei [40] without sacrificing query times goes out of the scope of this article and will be covered in a journal extension. While we have partial results in this direction, however, this approach cannot break the $\Omega(occ)$ cache misses for locating the pattern's occurrences. Obtaining $O(occ/B)$ cache misses in a space comparable to that of our data structure is the ultimate (challenging) goal of this line of research.

[10] For $m = 100$, $occ$ was 18 on average. For $m = 1000$, $occ$ was 11 on average. For $m = 10000$, $occ$ was 4 on average.

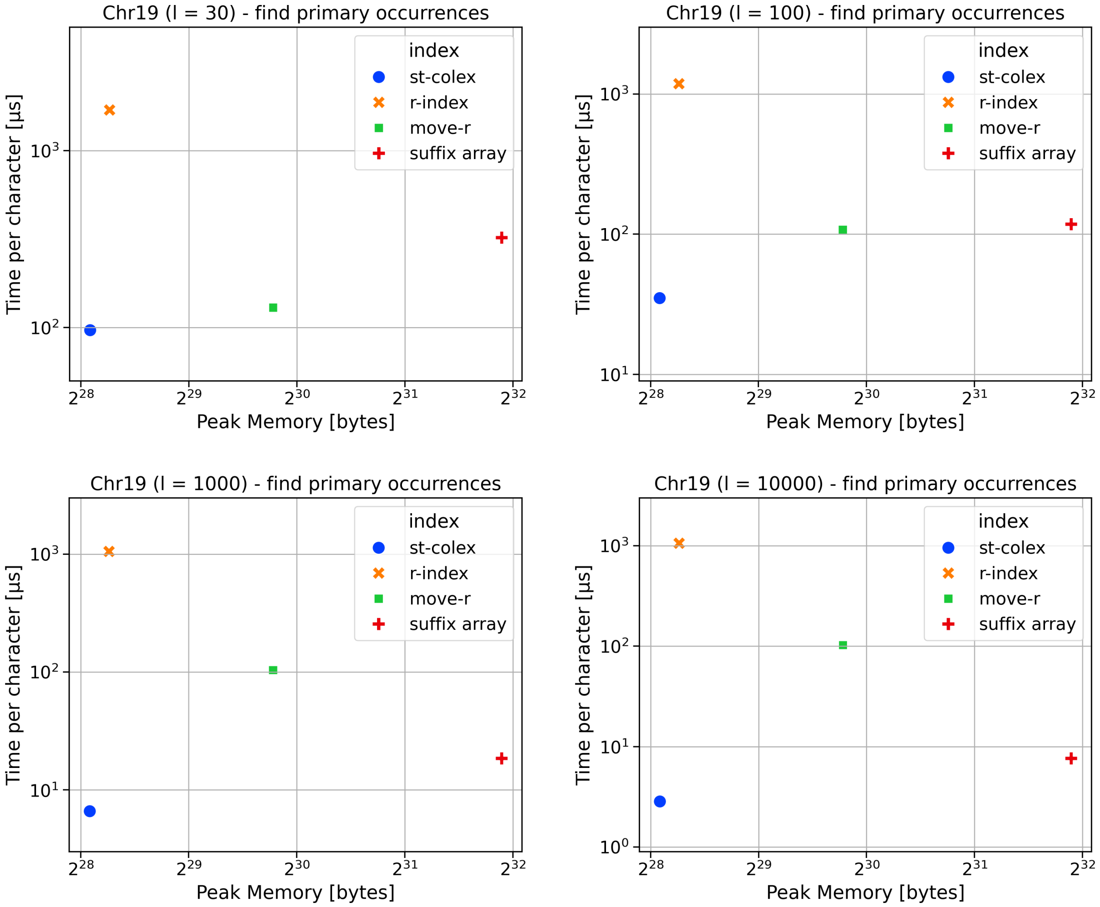


**Fig. 5.** Resources (time and working space) used by `st-colex`$^-$, $r$-index, move-$r$, and the Suffix Array to find primary occurrences of a set of $10^5$ patterns of length 30, 100, 1000, and 10000. Note that our index deviates from the usual Pareto curve (smaller space, larger query time) and always dominates by a wide margin all other solutions in both dimensions.

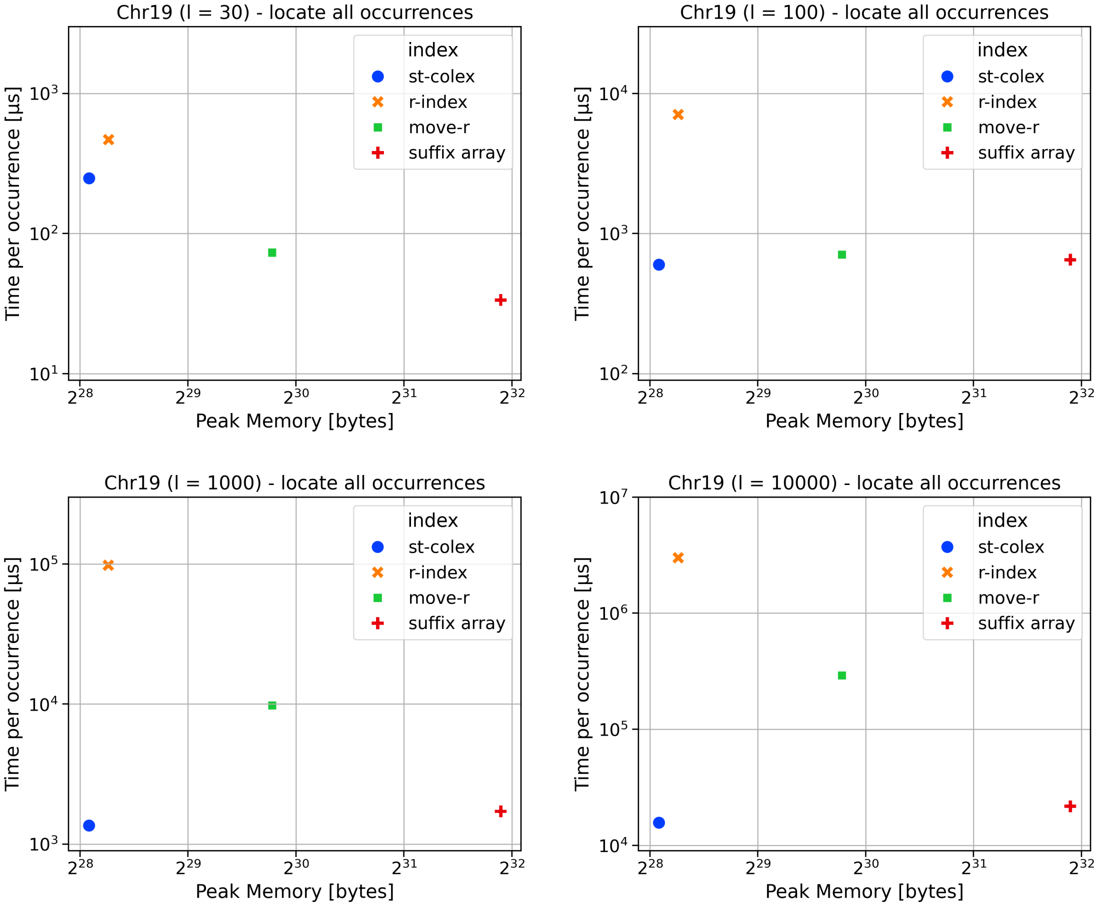


**Fig. 6.** Resources (time and working space) used by `st-colex`$^-$, $r$-index, move-$r$, and the Suffix Array to locate all occurrences of a set of $10^5$ patterns of length 30, 100, 1000, and 10000. Being based on computing the $\bar{\phi}$ function, our locate mechanism cannot escape the Pareto curve when *occ* is large (top left plot — about 300 occurrences per pattern on average). Nevertheless, our work shows for the first time that *all* occurrences can be located with a small sample of the Prefix Array (unlike suffixient arrays, see Appendix A). Escaping this Pareto curve (for large *occ*) is out of the scope of this article and will be covered in future research.

## 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. Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 785–794, 2009. URL: https://dl.acm.org/doi/10.5555/1496770.1496856.
4. 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.
5. 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, July 23-26, 2024, Vienna, Austria*, volume 301 of *LIPIcs*, pages 1:1–1:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.
6. Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44:513–539, 2015.
7. 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)*, volume 100 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FUN.2018.10.
8. Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
9. 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, 2025. URL: https://arxiv.org/abs/2407.18753.
10. Bernard Chazelle. Filtering search: a new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, 1986.
11. 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.
12. Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *International Symposium on String Processing and Information Retrieval*, pages 180–192. Springer, 2012.
13. John G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Comput. J.*, 40(2/3):67–75, 1997.
14. Davide Cozzi, Massimiliano Rossi, Simone Rubinacci, Travis Gagie, Dominik Köppl, Christina Boucher, and Paola Bonizzoni. $\mu$- PBWT: a lightweight r-indexing of the PBWT for storing and querying UK biobank data. *Bioinform.*, 39(9), 2023.
15. Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.
16. Johannes Fischer and Volker Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM Journal on Computing*, 40:465–492, 2011.
17. Travis Gagie, Sana Kashgouli, and Ben Langmead. KATKA: A kraken-like tool with k given at query time. In Diego Arroyuelo and Barbara Poblete, editors, *String Processing and Information Retrieval - 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, volume 13617 of *Lecture Notes in Computer Science*, pages 191–197. Springer, 2022.
18. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in bwt-runs bounded space, 2018.
19. Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1), January 2020.
20. Gaston H Gonnet, Ricardo A Baeza-Yates, and Tim Snider. New indices for text: Pat trees and pat arrays. *Information Retrieval: Data Structures & Algorithms*, 66:82, 1992.
21. Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the thirty-second annual ACM Symposium on Theory of Computing (STOC)*, pages 397 – 406, 2000.
22. Juha Kärkkäinen, Giovanni Manzini, and Simon J. Puglisi. Permuted longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2009.
23. Dominik Kempa and Tomasz Kociumaka. Resolution of the burrows-wheeler transform conjecture. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1013. IEEE, 2020.

24. Dominik Kempa and Tomasz Kociumaka. Collapsing the hierarchy of compressed data structures: Suffix arrays in optimal compressed space. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1877–1886. IEEE, 2023.
25. Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 827–840, 2018.
26. Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Toward a definitive compressibility measure for repetitive sequences. *IEEE Transactions on Information Theory*, 69(4):2074–2092, 2022.
27. Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.
28. Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013.
29. Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *International Symposium on String Processing and Information Retrieval*, pages 201–206. Springer, 2010.
30. Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Trans. Inf. Theory*, 22(1):75–81, 1976.
31. Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
32. Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
33. 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.
34. Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
35. Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
36. Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.
37. Gonzalo Navarro. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Computing Surveys*, 54(2):29:1–29:31, 2022.
38. Gonzalo Navarro. Indexing Highly Repetitive String Collections, Part II: Compressed Indexes. *ACM Computing Surveys*, 54(2):26:1–26:31, 2022.
39. Gonzalo Navarro, Giuseppe Romana, and Cristian Urbina. Smallest suffixient sets as a repetitiveness measure, 2025. URL: https://arxiv.org/abs/2506.05638.
40. 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, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 101:1–101:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
41. Nicola Prezza. Optimal substring equality queries with applications to sparse text indexing. *ACM Trans. Algorithms*, 17(1):7:1–7:23, 2021.
42. Simon J Puglisi and Bella Zhukova. Relative lempel-ziv compression of suffix arrays. In *International Symposium on String Processing and Information Retrieval*, pages 89–96. Springer, 2020.
43. 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.
44. 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.
45. Vikram S Shivakumar, Omar Y Ahmed, Sam Kovaka, Mohsen Zakeri, and Ben Langmead. Sigmoni: classification of nanopore signal with a compressed pangenome index. *Bioinformatics*, 40(Supplement_1):i287–i296, 2024.
46. Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
47. Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, pages 1–11. IEEE, 1973.
48. Derrick E Wood, Jennifer Lu, and Ben Langmead. Improved metagenomic analysis with kraken 2. *Genome biology*, 20:1–13, 2019.
49. Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15:1–12, 2014.
50. 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.

# A Suffixient Arrays

Like suffixient arrays [9], ours is a technique for sampling the Prefix Array while still maintaining search functionalities. The idea behind suffixient arrays is rather simple and it is based on the concept of *suffixient set*. See Figure 7 for a running example. A suffixient set is a set $S \subseteq [n]$ of text positions with the property that, for every string $\alpha$ labeling the path starting from the suffix tree root to the first character of every suffix tree edge, there exists a position $i \in S$ such that $\alpha$ is a suffix of $\mathcal{T}[1, i]$. In other words, for every one-character right-extension $\alpha$ of every right-maximal string $\alpha[1, |\alpha| - 1]$, there exists $i \in S$ such that $\alpha$ is a suffix of $\mathcal{T}[1, i]$. The suffixient array sA is simply a (not necessarily unique) suffixient set $S$ of smallest cardinality, sorted according to the co-lexicographic order of the corresponding text prefixes $\{\mathcal{T}[1, i] \; : \; i \in S\}$. As discussed in the caption of Figure 7 with an example, binary search on sA and random access on $\mathcal{T}$ suffice to locate one pattern occurrence.

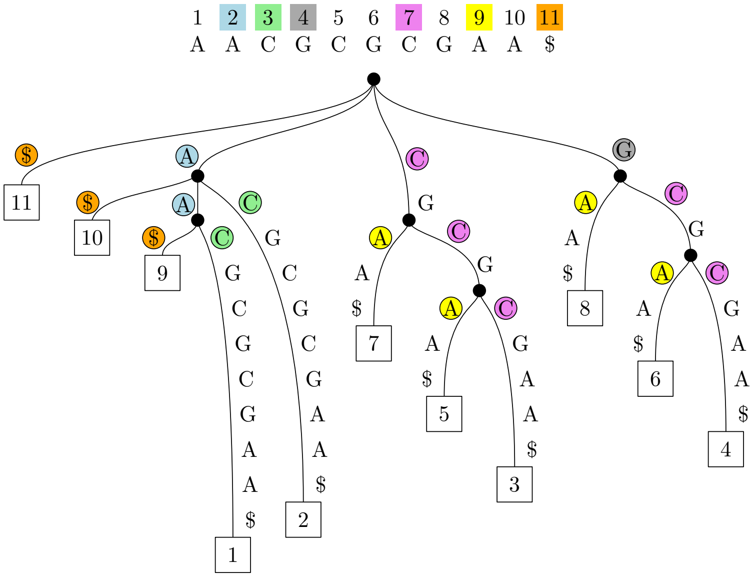


**Fig. 7.** Visualization of a smallest suffixient set for string AACGCGCGAA\$. The corresponding suffixient array is sA $= [11, 2, 9, 3, 7, 4]$ (assuming alphabet order $\$ < A < C < G$). We show how pattern matching works with an example. Imagine the task of matching pattern $P = CGCGA$ on $\mathcal{T}$, and assume that $P$ does occur in $\mathcal{T}$. Since the alphabet has cardinality at least 2, then the empty string $\epsilon$ is *right-maximal*, hence the pattern prefix $C$ is a one-character extension of a right-maximal string. By binary search on sA and random access on $\mathcal{T}$, we find all $i \in$ sA such that $C$ suffixes $\mathcal{T}[1, i]$: in this case, $\mathcal{T}[1, 3]$ and $\mathcal{T}[1, 7]$. Choose arbitrarily such a prefix, for instance $\mathcal{T}[1, 3]$ (this arbitrary choice does not affect correctness of the procedure). From this point, continue matching (by random access on $\mathcal{T}$) the remaining suffix $P[2, 5] = \underline{GCG}A$ with $\mathcal{T}$'s suffix following the match: $\mathcal{T}[4, 7] = \underline{GCG}CGAA\$$. As highlighted, three characters ($GCG$) match. At this point, we know that both $\mathcal{T}[3, 7] = CGCGC$ and $P[1, 5] = CGCGA$ occur in the text. But then, this means again that $P[1, 4] = CGCG$ is right-maximal, hence binary-searching sA with string $CGCGA$ will yield at least one prefix being suffixed by $CGCGA$: in this case, $\mathcal{T}[1, 9]$. Since we reached the end of $P$, we found an occurrence of $P$, that is: $P = \mathcal{T}[9 - m + 1, 9] = \mathcal{T}[5, 9]$.

While suffixient arrays sample every edge of the suffix tree (see Figure 7), our technique samples just a subset of the edges (that is, the first edges in every STPD path, see Figure 1). As we show in this paper, this has several benefits: it leads to a smaller sampling size (consistently smaller in practice), it allows us to locate the occurrence optimizing a user-defined function $\pi$, and ultimately it allows us to simulate suffix tree operations and locate all pattern occurrences.

## B Basic Concepts

We review more basic concepts. We start with the definition of the Suffix/Prefix Array and their inverses.

**Definition 20 (Suffix Array (SA) etc. [34]).** *Let $\mathcal{S}$ be a string of length $n$.*

- *The* Suffix Array SA *of $\mathcal{S}$ is the permutation of $[n]$ such that $\mathcal{S}[\mathrm{SA}[i], n] <_{lex} \mathcal{S}[\mathrm{SA}[j], n]$ holds for any $i, j \in [n]$ with $i < j$.*
- *The* Prefix Array PA *of $\mathcal{S}$ is the permutation of $[n]$ such that $\mathcal{S}[1, \mathrm{PA}[i]] <_{colex} \mathcal{S}[1, \mathrm{PA}[j]]$ holds for any $i, j \in [n]$ with $i < j$.*
- *The* Inverse Suffix Array ISA *of $\mathcal{S}$ is the permutation of $[n]$ such that $\mathrm{ISA}[i] = j$ if and only if $\mathrm{SA}[j] = i$.*
- *The* Inverse Prefix Array IPA *of $\mathcal{S}$ is the permutation of $[n]$ such that $\mathrm{IPA}[i] = j$ if and only if $\mathrm{PA}[j] = i$.*

We proceed with the definition of the suffix tree. In what follows, we fix a text $\mathcal{T}$ of length $n$.

**Definition 21 (Suffix trie and Suffix tree (ST) [47]).** *The* suffix tree *of $\mathcal{T}$ is an edge-labeled rooted tree with $n$ leaves numbered from 1 to $n$ such that (i) each edge is labeled with a non-empty substring of $\mathcal{T}$, (ii) each internal node has at least two outgoing edges, (iii) the labels of outgoing edges from the same node start with different characters, and (iv) the string obtained by concatenating the edge labels on the path from the root to the leaf node numbered $\mathrm{SA}[i]$ is $\mathcal{T}[\mathrm{SA}[i], n]$ where SA is the Suffix Array of $\mathcal{T}$.*

The *suffix trie* of $\mathcal{T}$ is the edge-labeled tree obtained from the suffix tree by replacing every edge labeled with a string $\alpha$ with a path of $|\alpha|$ edges labeled with $\alpha[1], \ldots, \alpha[|\alpha|]$. Hence the suffix trie is edge-labeled with characters, while the suffix tree is edge-labeled with strings. A node of degree 2 in the suffix trie that is not the root is an *implicit node*, all other nodes are *explicit nodes*. Note that explicit nodes are the nodes that are both in the suffix tree and the suffix trie, while a node that is introduced by the procedure of replacing edges with paths above is an implicit node. For an internal node $u$ in the suffix tree, we denote with $\mathrm{out}(u) \subseteq \Sigma$ the set of first characters of strings labeling outgoing edges of $u$.

**Definition 22 (Path Label, String Depth, Locus).** *(i) For a node $u$ in the suffix tree/trie of $\mathcal{T}$, we call $\alpha(u)$ the unique string obtained by concatenating the edge labels on the path from the root of the suffix tree to $u$ the* path label *of $u$. (ii) We call $\mathrm{sd}(u) = |\alpha(u)|$ the* string depth *of $u$. (iii) For a right-maximal substring $\alpha$ of $\mathcal{T}$, the* locus *of $\alpha$, denoted by $\mathrm{locus}(\alpha)$, is the unique suffix tree node $u$ for which $\alpha(u) = \alpha$.*

Observe that, for any $i \neq j$, the longest common extension $\mathcal{T}[i, i + \mathrm{rlce}(i, j) - 1]$ at $i, j$ is a right maximal substring whose locus is at string depth $\mathrm{rlce}(i, j)$ in the suffix tree.

We proceed with the definitions of the longest common Prefix Array, the permuted longest common Prefix Array, and the longest previous factor array. Note that LCP is the longest common Prefix Array for a text $\mathcal{T}$, while lcp was the function that for two strings returns the length of their longest common prefix.

**Definition 23 (LCP, PLCP, and LPF).** *Let $\mathcal{S}$ be a string of length $n$.*

- *The* longest common prefix *array* LCP *of $\mathcal{S}$ is the length-$(n-1)$ integer array such that* $\mathrm{LCP}[i] := \mathcal{S}.\mathrm{rlce}(\mathrm{SA}[i], \mathrm{SA}[i-1])$*, for all $i \in [2, n]$, where* SA *is the Suffix Array of $\mathcal{S}$.*
- *The* permuted longest common prefix *array* PLCP *of $\mathcal{S}$ is the length-$(n-1)$ integer array such that* $\mathrm{PLCP}[i] := \mathrm{LCP}[\mathrm{ISA}[i]]$*, for all $i \in [n-1]$, where* ISA *is the Inverse Suffix Array of $\mathcal{S}$.*
- *An* irreducible PLCP value *is a value* $\mathrm{PLCP}[i]$ *such that $i = 1$ or* $\mathrm{PLCP}[i] \neq \mathrm{PLCP}[i-1] - 1$*.*
- *The* longest previous factor *array* LPF *of $\mathcal{S}$ is the length-$n$ integer array such that* $\mathrm{LPF}[i] := \max\{\mathcal{S}.\mathrm{rlce}(i, j) : j < i\}$ *for all $i \in [n]$.*

We now define the Burrows-Wheeler transform. In order to do so we need to define the rotations of a string. For a string $\mathcal{S}$ of length $n$ and an integer $i \in [n]$, the $i$'th rotation of $\mathcal{S}$ is the string $\mathcal{S}^{\leftarrow i} := \mathcal{S}[i+1] \ldots \mathcal{S}[n]\mathcal{S}[1] \ldots \mathcal{S}[i]$.

**Definition 24 (Burrows-Wheeler Transform [8]).**

- *The* Burrows-Wheeler transform *(*co-Burrows-Wheeler transform*) of a string $\mathcal{S}$, denoted by* $\mathrm{BWT}(\mathcal{S})$ *(*$\mathrm{coBWT}(\mathcal{S})$*), is the permutation of the characters of $\mathcal{S}$ that is obtained by lexicographically sorting (co-lexicographically sorting) all the rotations of $\mathcal{S}$ and taking the last (first) characters of the strings in this sorted list.*
- *We define $r$ ($\bar{r}$) as the number of equal-letter runs in* $\mathrm{BWT}(\mathcal{S})$ *(*$\mathrm{coBWT}(\mathcal{S})$*), i.e., the number of maximal substrings of* $\mathrm{BWT}(\mathcal{S})$ *(*$\mathrm{coBWT}(\mathcal{S})$*) containing a single character.*

*We will omit $\mathcal{S}$ when clear from the context.*

*Remark 8.* The following properties concerning the SA, ISA, BWT, and coBWT of a string $\mathcal{S}$ are immediate from their definition. For $i, j \in [n]$,

1. $\mathrm{BWT}(\mathcal{S})[i] = \mathcal{S}[\mathrm{SA}[i] - 1]$, where $\mathcal{S}[0] := \mathcal{S}[n]$.
2. If $\mathrm{ISA}[i] < \mathrm{ISA}[j]$ and $\mathcal{S}[i-1] = \mathcal{S}[j-1]$, then $\mathrm{ISA}[i-1] < \mathrm{ISA}[j-1]$, where $\mathrm{ISA}[0] := \mathrm{ISA}[n]$.
3. $\mathrm{BWT}[\mathrm{ISA}[i]] = \mathcal{S}[i-1]$.
4. $\mathrm{BWT}(\mathcal{S}) = \mathrm{BWT}(\mathcal{S}^{\leftarrow i})$.
5. $\mathrm{coBWT}[i] = \mathcal{S}[\mathrm{PA}[i] + 1]$ where $\mathcal{S}[n+1] := \mathcal{S}[1]$.

We proceed with the definition of Range Minimum and Maximum queries.

**Definition 25.** *Given a list $L$ of $n$ integers, the* Range Minimum (Maximum) query *on $L$ with arguments $\ell, r \in [n]$, returns the index* $\mathrm{argmin}_{k \in [\ell, r]} L[k]$ *(*$\mathrm{argmax}_{k \in [\ell, r]} L[k]$*) of the minimum (maximum) element in $L[\ell, r]$.*

There exists a data structure that uses $2n + o(n)$ bits and supports Range Minimum (Maximum) queries in $O(1)$ time [16].