Simple Order-Isomorphic Matching Index with Expected Compact Space

Sung-Hwan Kim[1]
Pusan National University, Busan, South Korea

Hwan-Gue Cho[2]
Pusan National University, Busan, South Korea

Abstract

In this paper, we present a novel indexing method for the order-isomorphic pattern matching problem (also known as order-preserving pattern matching, or consecutive permutation matching), in which two equal-length strings are defined to match when $X[i] < X[j]$ iff $Y[i] < Y[j]$ for $0 \le i, j < |X|$. We observe an interesting relation between the order-isomorphic matching and the insertion process of a binary search tree, based on which we propose a data structure which not only has a concise structure comprised of only two wavelet trees but also provides a surprisingly simple searching algorithm. In the average case analysis, the proposed method requires $\mathcal{O}(R(T))$ bits, and it is capable of answering a count query in $\mathcal{O}(R(P))$ time, and reporting an occurrence in $\mathcal{O}(\lg |T|)$ time, where T and P are the text and the pattern string, respectively; for a string X, $R(X)$ is the total time taken for the construction of the binary search tree by successively inserting the keys $X[|X|-1], \cdots, X[0]$ at the root, and its expected value is $\mathcal{O}(|X| \lg \sigma)$ where σ is the alphabet size. Furthermore, the proposed method can be viewed as a generalization of some other methods including several heuristics and restricted versions described in previous studies in the literature.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Compact Data Structure, String Matching, Order-Preserving Matching, Suffix Array, FM-index, Binary Search Tree

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.61

Acknowledgements The authors would like to thank anonymous reviewers for their helpful comments.

1 Motivation

In string matching, it is asked to find all the substrings of a given text string that match a pattern string. Instead of the standard setting where two strings are said to match if they are exactly the same, we can define a match of two strings in a different way depending on the target application. For example, parameterized string matching [2] and structural pattern matching [23] define the matching using bijective functions on the alphabet Σ, satisfying certain conditions to address pattern matching problems on program source codes and RNA sequences, respectively. In another variant of the matching problem [21], Cartesian tree is used to determine the matching of strings. Especially, in order-isomorphic pattern matching [19, 15], which is of interest in this paper, the relative ordering of characters is used; more formally, two equal-length strings X and Y are said to match, which we denote by $X =_o Y$, if $X[i] < X[j] \Leftrightarrow Y[i] < Y[j]$ for $0 \le i, j < |X|$.

Despite the standard string matching, for which there are many space-efficient data structures such as [9] that require only a small amount of space, indexing strings for variant problems in a compact space has been considered quite challenging. Until Ganguly et al. [13]

[1] Currently working at Ca' Foscari University of Venice, Italy.
[2] Corresponding author

presented the first succinct index for the parameterized string matching problem, it had not been revealed whether there exists any data structure that is capable of efficiently processing string matching queries for such problems using asymptotically less than $\Theta(n \lg n)$ bits; only the indexing methods that exploit the conventional suffix tree and suffix array had been usually considered. After this breakthrough method was successfully invented, several succinct and compact data structures [12, 11, 16, 18, 17] for these problems have started to be actively developed.

One of the main concerns in developing a space-efficient index for such problems is suffix representation. Conventionally, we transform the suffixes into a certain form so that indexing these encoded suffixes enables us to search for pattern strings efficiently. Since the encoded suffixes should have several required properties to be effectively indexed, finding an appropriate representation is an important goal. Let $E(\cdot)$ be such an encoding function. When exploiting the conventional suffix tree and suffix array, it is sufficient to hold the property that $E(X \circ Y)$ has a prefix $E(X)$ for any strings X and Y where $\circ$ is the concatenation operator. However, when developing an index that is more space-efficient, we need more than that. Many space-efficient indexes rely on the compact representation of the relation between adjacent suffixes. For a text string $T[0..n-1]$, adjacent suffixes $T[i..n-1]$ and $T[i+1..n-1]$ are associated using this relation, which refers to the so-called *LF-mapping* or Ψ *function* in the literature, and they are used to access the sampled suffix array stored in a compact space. To do this, we must have a space-efficient way to characterize the operation that prepends a character; i.e. it is necessary to compactly represent the relation between the encoded strings $E(X)$ and $E(x \circ X)$ for any character x and string X.

It is more complicated for the order-isomorphic matching. As mentioned in the literature [11, 16], there are possibly many positions in which encoded characters differ when comparing $E(X)$ with $E(x \circ X)$. Moreover, an encoded character that has been already changed possibly changes again after performing prepending operations successively. This is quite different from much simpler problems such as parameterized string matching [2, 13], in which at most one character in an encoded string can change after prepending a character to the original string; and once an encoded character changes, it never changes anymore. Due to this complicated nature of the order-isomorphic matching, the ordering of the encoded strings is severely jumbled by prepending characters. Consequently, it is sophisticated to characterize the relation between adjacent suffixes, which makes its indexing problem much challenging.

For this reason, there has been a limited progress on developing indexes for the order-isomorphic matching that are space-efficient than the suffix tree-based method [6, 7]. Some methods used a filtering technique, in which two strings have the same signature only if they are order-isomorphic; e.g. up-down signature [3] and the rank information within a window of a restricted length [8]. However, these techniques may produce false positives in the filtering step, so the verification step must follow, which possibly involves substantial costs. It was also shown that one can report an occurrence of a pattern, if any, using an $\mathcal{O}(n \lg \lg n)$-bit data structure when the pattern length is restricted within a logarithmic size [10]. However, it had been an open problem whether there exists a compact index that does not restrict the searching capabilities until Ganguly et al. [11] presented the first $\mathcal{O}(n \lg \sigma)$-bit compact index for this problem, where n and σ is the text length and the alphabet size. The idea is to sample not only the suffix array but also the LF-mapping function, and the notion of LF-successor has been introduced to support this two-level sampling method. The underlying observation is the changing positions on the encoded string can be characterized by the rightmost position in which the change is made. They showed that this property can be used to represent the relations between suffixes whose LF-mappings are adjacent. However, representing these relations were still rather sophisticated, which involved the classification of suffixes into four types according to the behavior of their LF-mapping, and an intricate structure including the topology of the suffix tree and a number of its subgraphs as well as many auxiliary bitvectors and wavelet trees storing the required information.

1.1 Contributions

In this paper, we develop a novel index with a concise structure and a simpler searching algorithm. More specifically, the main contributions can be summarized as follows.

1. **Novel suffix representation (Section 2):** We show how the order-isomorphic pattern matching can be described in terms of binary search trees, based on which we propose a novel suffix representation for this problem. With the new representation, each encoded character is a set, thereby not entirely replaced with another one but an integer is added to it when it changes by a prepending operation. This resolves many complications that arise in representing the relations between adjacent suffixes compactly.
2. **Simplicity (Sections 3 and 4):** The proposed data structure has a simple structure, which basically consists of only a pair of wavelet trees [9, 20] (plus a sampled suffix array if reporting each occurrence is needed); each wavelet tree can be implemented as a single bitvector. The searching algorithm is also surprisingly simple; the process can be done just by (properly) walking down on one wavelet tree, and then tracing up on the other wavelet tree.
3. **Expected compact space (Section 5):** The proposed data structure requires $\mathcal{O}(R(T))$ bits, where $R(T)$ is the time taken by root insertion of the characters of the input string T into a binary search tree in the backward fashion, which is $\mathcal{O}(n \lg \sigma)$ on average [24] where n and σ is the text length and the alphabet size, respectively. It has the same bound as that of the previous work in the average case analysis. Since the deviation of $R(T)$ seems small [24], many input strings are likely to be indexed within this bound.
4. **Fully-searchable index:** As far as we know, for this problem, this is the first fully-searchable index that does not require to keep the text string separately. Although adding the text string itself does not increase the space requirement asymptotically, we believe the proposed method can be a hint towards a succinct self-index for this problem.
5. **Extensibility and generalizability (Section 6):** The proposed method also has interesting connections with other related methods and problems. It can be viewed as a generalized method from existing methods addressing this problem including: (i) window-based order-isomorphic testing [8], (ii) filtering with up-down signatures [3], and (iii) indexing for length-restricted queries [10]. We can also effectively derive index structures for many variations of the order-isomorphic matching problem.

2 Binary Search Tree and Order-isomorphic Matching

In this section, we observe the relation between binary search trees and order-isomorphic matching. For brevity, until Section 5, we assume that all the characters of the string are distinct. We show that order-isomorphic matching of two strings can be represented as a match of their corresponding binary search trees. In this perspective, prepending a character at the beginning of a string is equivalent to inserting a node into the corresponding binary search tree as the root node. We establish some notations for characterizing a root insertion operation on a binary search tree (see Figure 1).

2.1 Binary Search Tree Generated from a String

Let $X[0..|X|-1]$ be a string. Consider a binary search tree $\mathsf{BST}(X)$ constructed by inserting a node with key $X[i]$ and value i for $0 \le i < |X|$ in order; in other words, for each insertion $i = 0, \cdots, |X|-1$, we perform the conventional search for key $X[i]$ on the current binary search tree until we reach a leaf node, then we insert the new node having key $X[i]$ and value i as a child. For a node v of $\mathsf{BST}(X)$, we denote its value, left child and right child by $\mathsf{value}(v)$, $\mathsf{left}(v)$, and $\mathsf{right}(v)$, respectively. We also denote the subtree rooted at node v by $\mathsf{subtree}(v)$.

Consider two strings X and Y, and their corresponding binary search trees $\mathsf{BST}(X)$ and $\mathsf{BST}(Y)$. We say two binary search trees $\mathsf{BST}(X)$ and $\mathsf{BST}(Y)$ are value-identical, denoted by $\mathsf{BST}(X) =_v \mathsf{BST}(Y)$, if:

1. the number of nodes (tree sizes) are equal,
2. $\mathsf{value}(r_X) = \mathsf{value}(r_Y)$,
3. $\mathsf{subtree}(\mathsf{left}(r_X)) =_v \mathsf{subtree}(\mathsf{left}(r_Y))$, and
4. $\mathsf{subtree}(\mathsf{right}(r_X)) =_v \mathsf{subtree}(\mathsf{right}(r_Y))$.

where r_X and r_Y are the root node of $\mathsf{BST}(X)$ and $\mathsf{BST}(Y)$, respectively. To be well-defined, two empty binary search trees are also considered to be value-identical.

It is easy to see that X and Y are an order-isomorphic match iff $\mathsf{BST}(X)$ and $\mathsf{BST}(Y)$ are value-identical.

▶ **Lemma 1.** *$X =_o Y$ iff $\mathsf{BST}(X) =_v \mathsf{BST}(Y)$.*

Proof. We prove by induction. If $|X| = |Y| = 1$, it is trivial. Let us assume that $X =_o Y$ and $\mathsf{BST}(X) =_v \mathsf{BST}(Y)$. Consider two strings $X \circ x$ and $Y \circ y$ for some characters x and y. We can have $X \circ x =_o Y \circ y \Leftrightarrow \mathsf{BST}(X \circ x) =_v \mathsf{BST}(Y \circ y)$ immediately from the observation that, when a node with key x is inserted into a binary search tree $\mathsf{BST}(X)$ as its leaf node to obtain $\mathsf{BST}(X \circ x)$, the locus of the new leaf node is uniquely determined by the rank of x among $\{X[0], \cdots, X[|X|-1], x\}$. ◀

2.2 Root Insertion and Prepending operation

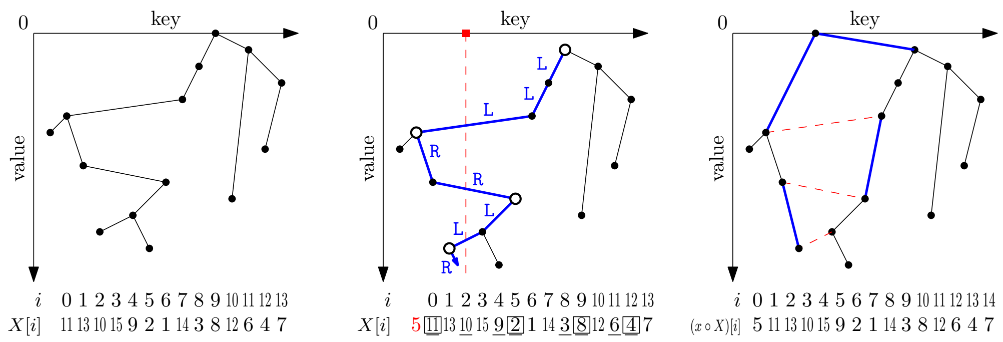
Let X be a string and x be a character. Consider a binary search tree $\mathsf{BST}(X)$. Suppose we prepend x at the beginning of X, and we want to have $\mathsf{BST}(x \circ X)$. We can observe that $\mathsf{BST}(x \circ X)$ can be obtained from $\mathsf{BST}(X)$ as follows:

1. Increment $\mathsf{value}(v)$ by 1 for every node v.
2. Insert a new node with key x and value 0 as the root node (as described in [24]).

Because the increment operation is applied to all nodes equally, the change can be characterized by how the root insertion is performed. Let us define $\mathsf{vpath}(x, X)$ to be the sequence of values of the nodes visited by searching x on $\mathsf{BST}(X)$; in other words, $\mathsf{vpath}(x, X)$ indicates the positions of the characters of X that correspond to the visited nodes when searching with the key x. We also define $\mathsf{branch}(x, X)$ to be a string over $\{\mathtt{L}, \mathtt{R}\}$ that indicates the branch taken at each node. More specifically, for $0 \le i < |\mathsf{vpath}(x, X)|$,

$$\mathsf{branch}(x, X) = \begin{cases} \mathtt{L} & \text{if } x < X[\mathsf{vpath}(x, X)[i]] \\ \mathtt{R} & \text{if } x > X[\mathsf{vpath}(x, X)[i]] \end{cases} \tag{1}$$

Here, note that $X[\mathsf{vpath}(x, X)[i]]$ is the key of the node having the value $\mathsf{vpath}(x, X)[i]$.


0
key
value
L
L
L
R
R
L
L
R
i 0 1 2 3 4 5 6 7 8 9 10 11 12 13
X[i] 11 13 10 15 9 2 1 14 3 8 12 6 4 7
i 0 1 2 3 4 5 6 7 8 9 10 11 12 13
X[i] 5 11 13 10 15 9 2 1 14 3 8 12 6 4 7
i 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
(x ∘ X)[i] 5 11 13 10 15 9 2 1 14 3 8 12 6 4 7

(a) BST(X) (b) Prepending $x = 5$ (c) BST$(x \circ X)$

Figure 1 Examples of a string X =11 13 10 15 9 2 1 14 3 8 12 6 4 7, and its corresponding binary search tree. (a) BST(X). Each dot is a node. x-axis represents the key, and y-axis represents the value of the nodes. (b) A character $x = 5$ is being prepended. Red dashed line indicates $x = 5$, and blue thick edges are the paths visited by searching for 5 on the tree. vpath(x, X)=0 2 4 5 8 9 11 12 (positions in which $X[i]$ are underlined), branch(x, X) = LLLRRLLR, and turnpoint$(x, X) = \{0, 5, 9, 12\}$ (positions in which $X[i]$ are boxed). (c) BST$(x \circ X)$. The red dashed edges are removed ones, blue thick edges are newly established edges.

We also define turnpoint(x, X) to be the set of vertices at which the branching direction is switched. If the left branch (L) is taken at the root node, the root node is also included.

$$\mathsf{turnpoint}(x, X) = \{\mathsf{vpath}(x, X)[i] : 0 \le i < p, \mathsf{branch}(x, X)[i-1] \neq \mathsf{branch}(x, X)[i]\} \quad (2)$$

where $p = |\mathsf{branch}(x, X)|$ and for convenience we assume $\mathsf{branch}(x, X)[-1] = \mathtt{R}$.

3 Data Structure

In this section, we describe how to organize and implement the proposed data structure. We transform the suffixes into a certain form based on their corresponding binary search trees, after which we sort these encoded suffixes. After computing two arrays F and L of binary strings containing information required for the searching tasks, we build wavelet trees [9, 20] on them.

3.1 Suffix Representation

First, we define an encoding function $E(X)$ to transform a string into a certain form in which an order-isomorphic match can be easily determined. We use the fact that (i) two strings are order-isomorphic iff their corresponding binary search trees are value-identical, and (ii) two value-identical binary search trees are still value-identical after performing a root insertion with the same branching sequence.

We define $E(X)$ as a length-$|X|$ string over $2^{\mathbb{N}\cup\{\infty\}}$ where $\mathbb{N}$ is the set of natural numbers; i.e. each character of $E(X)$ is a set. If X is an empty string, $E(X)$ is also an empty string. If X is a non-empty string, then $E(X)$ can be defined recursively as follows: for $0 \le i \le |X|$,

$$E(x \circ X)[i] = \begin{cases} \{\infty\} & \text{if } i = 0, \\ E(X)[i-1] \cup \{i\} & \text{if } i - 1 \in \mathsf{turnpoint}(x, X) \\ E(X)[i-1] & \text{otherwise.} \end{cases} \quad (3)$$

We transform all the suffixes $T[i..n-1]$ of a given text string $T[0..n-1]$ into its encoded form $E(T[i..n-1])$ for $0 \le i \le n$; here, we define $T[n..n-1]$ to be an empty string. Before sorting them, we need to define the ordering on sets because characters of the encoded suffixes are sets. We compare two sets by treating each set as a string that consists of the elements in the ascending order: i.e., for two sets A and B, we define $A < B$ iff:

1. $\min A < \min B$, or
2. $\min A = \min B$ and $A - \{\min A\} < B - \{\min B\}$.

Now we can sort all the suffixes to determine the lexicographic ranks of (encoded) suffixes; note that each encoded suffix is a string of sets. Let $\mathsf{SA}[0..n]$ be an integer array such that $\mathsf{SA}[i] = j$ iff there are i encoded suffixes that are lexicographically smaller than $E(T[j..n-1])$: i.e. $\mathsf{SA}[i] = j \Leftrightarrow i = |\{0 \le k \le n \mid E(T[k..n-1]) < E(T[j..n-1])\}|$. This array is the so-called *suffix array*. Because $\mathsf{SA}[0..n]$ is a permutation of $0, \cdots, n$, we can also define its inverse $\mathsf{SA}^{-1}[0..n]$ such that $\mathsf{SA}^{-1}[\mathsf{SA}[i]] = i$ for $0 \le i \le n$.

3.2 Associating Adjacent Suffixes Using F and L Arrays

In this subsection, we consider the relation between two adjacent (encoded) suffixes $E(T[i..n-1])$ and $E(T[i+1..n-1])$. In the literature of compact indexing, it is common to use the so-called *LF-mapping*, which is defined as: for $0 \le i \le n$,

$$\mathsf{LF}(i) = \mathsf{SA}^{-1}[(\mathsf{SA}[i] + n) \mod (n+1)] \tag{4}$$

It maps the lexicographical rank of a suffix $E(T[\mathsf{SA}[i]..n-1])$ into the lexicographical rank of its positionally previous suffix $E(T[\mathsf{SA}[i]-1..n-1])$ for $0 < i \le n$. For the rank of the suffix $E(T[0..n-1])$ starting at position 0, it maps to the rank of the empty suffix $E(T[n..n-1])$. Our goal is to implement $\mathsf{LF}(i)$ in a compact space, which will be done here by defining two length-$(n+1)$ arrays F and L of bitstrings.

To do this, we define an array $C[0..n]$ of bitstrings such that $C[n] = 1$, $C[n-1] = 001$, and, for $0 \le i < n-1$,

$$C[i] = (00)^{l_0} 01 (00)^{l_1} 01 \cdots (00)^{l_{k-1}} 1 \tag{5}$$

where $\mathsf{branch}(T[i], T[i+1..n-1]) = \mathtt{R}^{l_0}\mathtt{L}^{1+l_1}\mathtt{R}^{1+l_2}\cdots d^{1+l_{k-1}}$; d is either $\mathtt{R}$ or $\mathtt{L}$, and $l_0, \cdots, l_{k-1} \ge 0$. In other words, $C[i]$ is a bitstring that indicates the branching direction when we insert a node with key $T[i]$ into the binary search tree $\mathsf{BST}(T[i+1..n-1])$ as the root node. It is obvious that the sum of the length of bitstrings $C[i]$ over $0 \le i \le n$ is bounded by $\mathcal{O}(R(T))$ where $R(T)$ is the total time taken by performing the root insertion of a node with key $T[n-1], \cdots, T[0]$ in order, starting with an empty binary search tree.

▶ **Lemma 2.** *For a string $T[0..n-1]$,*

$$\sum_{0 \le i \le n} |C[i]| = \mathcal{O}(R(T)) \tag{6}$$

Proof. Immediate from the fact that the length of $C[i]$ is proportional to the length of $\mathsf{branch}(\cdot,\cdot)$. ◀

The bits at even positions $(0, 2, 4, \cdots)$ indicate the length of the code; 1-bit at an even position means the code ends. The bits at odd positions represent the unary code of $l_0, l_1, \cdots, l_{k-1}$ (except that 1-bit does not follow 0-bits for the last run) which characterize the prepending character $T[i]$ for a suffix $T[i+1..n-1]$ in terms of its encoded form.

i	$\mathsf{SA}[i]$	$\mathsf{LF}[i]$	$F[i]$	$L[i]$	$E(T[\mathsf{SA}[i]..n-1])$
0	12	1	1	001	empty string
1	11	11	001	001	$\{\infty\}$
2	7	8	011	00011	$\{\infty\}$ $\{1,\infty\}$ $\{1,\infty\}$ $\{1,\infty\}$ $\{\infty\}$
3	8	2	011	011	$\{\infty\}$ $\{1,\infty\}$ $\{1,\infty\}$ $\{\infty\}$
4	1	12	0101001	0000011	$\{\infty\}$ $\{1,\infty\}$ $\{2,\infty\}$ $\{1,2,\infty\}$ $\{3,\infty\}$ $\{1,2,\infty\}\cdots$
5	9	3	01001	011	$\{\infty\}$ $\{1,\infty\}$ $\{\infty\}$
6	5	10	011	0001001	$\{\infty\}$ $\{1,\infty\}$ $\{\infty\}$ $\{1,2,\infty\}$ $\{1,\infty\}$ $\{1,\infty\}$ $\cdots$
7	3	9	011	0001011	$\{\infty\}$ $\{1,\infty\}$ $\{\infty\}$ $\{1,2,\infty\}$ $\{\infty\}$ $\{1,2,\infty\}\cdots$
8	6	6	00011	011	$\{\infty\}$ $\{\infty\}$ $\{1,2,\infty\}$ $\{1,\infty\}$ $\{1,\infty\}$ $\{\infty\}$
9	2	4	0001011	0101001	$\{\infty\}$ $\{\infty\}$ $\{1,2,\infty\}$ $\{3,\infty\}$ $\{1,2,\infty\}$ $\{\infty\}$ $\cdots$
10	4	7	0001001	011	$\{\infty\}$ $\{\infty\}$ $\{1,2,\infty\}$ $\{\infty\}$ $\{1,2,\infty\}$ $\{1,\infty\}$ $\cdots$
11	10	5	001	01001	$\{\infty\}$ $\{\infty\}$
12	0	0	0000011	1	$\{\infty\}$ $\{\infty\}$ $\{1,\infty\}$ $\{2,3,\infty\}$ $\{1,2,\infty\}$ $\{3,\infty\}$ $\cdots$

Figure 2 Example of sorted (encoded) suffixes and related information for T =5 3 4 1 6 2 8 7 9 10 12 11.

Now we define a length-$(n+1)$ array $F[0..n]$ as a permuted array of $C[0..n]$ via $\mathsf{SA}[0..n]$: for $0 \le i \le n$,

$$F[i] = C[\mathsf{SA}[i]] \tag{7}$$

We also define another length-$(n+1)$ array $L[0..n]$ such that: for $0 \le i \le n$,

$$L[i] = F[\mathsf{LF}(i)] \tag{8}$$

See Figure 2 for an example of how the (encoded) suffixes are sorted and how these arrays are computed.

3.3 Operations on F and L

Obviously, $\{C[i]\}$ is a prefix code; i.e. there exists no $0 \le i, j \le n$ such that $C[i]$ is a prefix of $C[j]$ unless $C[i] = C[j]$. Therefore, we can build a wavelet tree with a prefix-coding (as described in Section 3.2 in [20]) for F (and L, respectively).

For the completeness, we describe the detail. Let us consider the wavelet tree WT_L of $L[0..n]$. We define it recursively. For $0 \le i \le n$, we write $L[i][0]$ into a single bitmap B of length $n+1$. B is stored in the root node. We divide $I = \{0, \cdots, n\}$ into two disjoint subsets $I_0 = \{0 \le i \le n : L[i][0] = 0\}$, and $I_1 = \{0 \le i \le n : L[i][0] = 1\}$. Let L_0 be a sequence of bitmaps that can be obtained by writing for $L[i][1..]$ for $i \in I_0$ in order. Similarly, we define L_1 using L and I_1. Then we construct the wavelet tree for L_0 (and L_1) recursively, and make the tree as the left (and right, resp.) subtree of the current node. The recursion is repeated until all the bits are processed.

For a node w of the wavelet tree WT_L for L, a bit $b \in \{0,1\}$, and an integer $0 \le i, j \le n$, we define the following operation:

- $\mathsf{WT}_L.\mathsf{down}_w(b, i, j)$ returns a triplet (w', i', j') of a node w', integers i' and j' such that:
  - If $b = 0$, w' is the left child node of w, otherwise w' is the right child of w,
  - $i' = B_w.\mathsf{rank}(b, i-1)$, and
  - $j' = B_w.\mathsf{rank}(b, j) - 1$,

  where B_w is the bitmap of node w, and $B_w.\mathsf{rank}(b, i)$ is the number of b-bits in $B_w[0..i]$.

We construct the wavelet tree WT_F for F in the same way except, at this time, we build a select dictionary on the bitmap of each node. For a node w of the wavelet tree WT_F of F, and an integer $0 \leq i, j \leq n$, we define the following operation:

- $\mathsf{WT}_F.\mathsf{up}_w(i, j)$ returns a triplet (w', i', j') of a node w', integers i' and j' such that:
  - w' is the parent node of w,
  - $i' = B_{w'}.\mathsf{select}(b, i + 1)$, and
  - $j' = B_{w'}.\mathsf{select}(b, j + 1)$,

  where $B_{w'}$ is the bitmap of node w' where $b = 0$ if w is the left child of w', $b = 1$ otherwise; and $B_{w'}.\mathsf{select}(b, i)$ is the position of the i-th occurrence of b-bit in $B_{w'}$ (note: defined for $i \geq 1$, and $i = 1$ indicates the leftmost occurrence).

Note that L is a permuted array of F. Therefore, the topology of their wavelet trees are the same, and there is a one-to-one correspondence between the nodes of WT_L and the nodes of WT_F. For a node w in WT_L, we denote by $\mathsf{paired}(w)$ its associated node w' in WT_F.

Note also that the wavelet trees WT_L (and WT_F) can be emulated with a single bitmap equipped with the rank (and select) dictionary. This can be done by concatenating all the bitmaps in the level order. When we go down in WT_L, the bitmap boundary at the next level can be computed by counting the number of 0- or 1-bits within the boundary at the current level and node. As described later, going up in WT_F will be always followed by going down in WT_L, so we can reused the bitmap boundaries.

▶ **Lemma 3.** *The wavelet trees* WT_L *and* WT_F *require* $\mathcal{O}(R(T))$ *bits supporting the operations described above in* $\mathcal{O}(1)$ *time.*

Proof. It is well-known that rank and select operations on a length-n bitmap can be performed in $\mathcal{O}(1)$ time using $n + o(n)$ bits [4, 14]. The total number of bits stored in arrays F and L is $\mathcal{O}(R(T))$ according to Lemma 2. ◀

4 Searching Algorithm

In this section, we present the searching algorithm on the proposed data structure implemented in the previous section. As other searching methods based on suffix arrays, our searching algorithm represents the occurrences of a pattern string as an interval (p_s, p_e) on the suffix array. For a pattern string $P[0..m-1]$, we call a pair (p_s, p_e) of integers the *suffix range* for P if $p_s \leq i \leq p_e \Leftrightarrow P =_o T[\mathsf{SA}[i]..\mathsf{SA}[i] + m - 1]$. We give a simple algorithm to compute the suffix range for a pattern string. The number of occurrences can be immediately obtained by $p_e - p_s + 1$ once the suffix range (p_s, p_e) is computed. To report each occurrences in $\mathcal{O}(\lg n)$ time, we also present a new method to sample the suffix array.

4.1 Computing the Suffix Range

Given a length-m pattern, its suffix range is computed in the backward fashion. We start from the last character of the pattern. The encoded string of any length-1 string is always $\{\infty\}$, thus we have its suffix range as $(1, n)$ where n is the text length. We also have the binary search tree with a single node at this moment. For each character to be prepended, we perform the root insertion accordingly to obtain the bitstring representation of the branching directions in a similar way described in Equation 5. Then we walk down on WT_L according to the computed bitstring, followed by jumping to the corresponding node on WT_F, and tracing up to the root of the wavelet tree, which completes the updated suffix range. Figure 3 describes an example of an iteration that updates the suffix range by prepending a character at the currently searched pattern, Algorithm 1 describes the procedure to compute the suffix range of a given pattern.

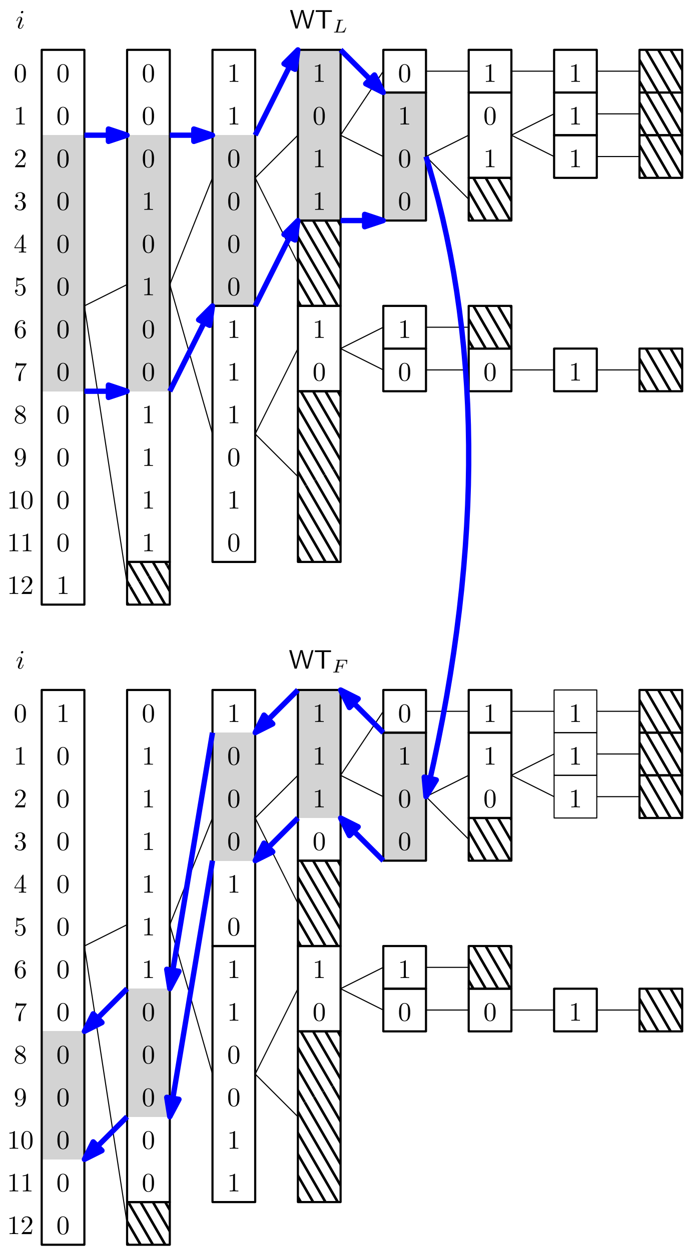


Figure 3 An iteration for searching for the pattern P =3 1 4 on the text string T described in Figure 2. For each wavelet tree, upward (downward) branches correspond to 0-bits (1-bits, resp.). Given a suffix range $(p_s, p_e) = (2, 7)$ for $P[1..2] = 14$, $P[0] = 3$ is to be prepended. Since $\mathsf{branch}(P[0], P[1..2]) = \texttt{RL}$, $c = \texttt{00011}$. We walk down on WT_L according to $c[0..|c| - 2]$, then jump to the corresponding locus on WT_F, then trace up to the root. After the iteration the suffix range is updated to $(8, 10)$. Note that $\mathsf{SA}[8] = 6$, $\mathsf{SA}[9] = 2$, $\mathsf{SA}[10] = 4$, and $T[6..8]$ =8 7 9, $T[2..4]$ =4 1 6, and $T[4..6]$ =6 2 8, which are all order-isomorphic to $P[0..2]$=3 1 4.

Algorithm 1 Computing the suffix range (p_s, p_e) of $P[0..m-1]$.

```
procedure computeSuffixRange($P[0..m-1]$: a non-empty string)
    $(p_s, p_e) \leftarrow (1, n)$    ▷ The suffix range for a length-1 string $P[m-1]$ is always $(1, n)$.
    for $i = m-2, \cdots, 0$ do
        $c \leftarrow$ bitstring as defined in Eq. (5) w.r.t. $P[i]$ and $P[i+1..m-1]$.
        $w \leftarrow$ the root of $\mathsf{WT}_L$
        for $j = 0, \cdots, |c|-2$ do    ▷ Process the code except the 1-bit at the end
            $(w, p_s, p_e) \leftarrow \mathsf{WT}_L.\mathsf{down}_w(c[j], p_s, p_e)$
        end for
        $w' \leftarrow \mathsf{paired}(w)$    ▷ Jump to $\mathsf{WT}_F$
        for $j = |c|-2, \cdots, 0$ do    ▷ Trace up until reaching the root node
            $(w', p_s, p_e) \leftarrow \mathsf{WT}_F.\mathsf{up}_{w'}(p_s, p_e)$
        end for
    end for
    return $(p_s, p_e)$
end procedure
```

It is easy to show its running time. It is proportional to the sum of the code length over all iterations, thereby proportional to the time required for performing the root insertion of nodes keyed by $P[m-1], \cdots, P[0]$ to the empty binary search tree.

▶ **Lemma 4.** *Algorithm 1 runs in $\mathcal{O}(R(P))$ time.*

Proof. The time taken for a single iteration for i is proportional to $|c|$ because other basic operations take $\mathcal{O}(1)$ time. $|c|$ is proportional to the length of the path retrieved in the binary search tree. Therefore, the total time is $\mathcal{O}(R(P))$ by its definition. ◀

Now we show the correctness of this algorithm. Suppose we have a suffix range (p_s, p_e) and a bitstring c that characterizes the next character $P[i]$ to be prepended to the currently searched pattern $P[i+1..m-1]$. Among the suffixes within the current suffix range (for $P[i+1..m-1]$), we need to determine which suffix should be included in the updated suffix range (for $P[i..m-1]$). Let us define $\mathbf{b}_j = \mathsf{branch}(T[\mathsf{SA}[\mathsf{LF}(j)]], T[\mathsf{SA}[j]..n-1])$ for $p_s \leq j \leq p_e$, and let $\mathbf{b}_P = \mathsf{branch}(P[i], P[i+1..m-1])$. If $\mathsf{LF}(j)$ is included in the update suffix range, then $\mathbf{b}_j$ must have a prefix $\mathbf{b}_P$, which is equivalent to that $L[j]$ has a prefix $c[0..|c|-2]$. Otherwise, the root insertion for the suffix would take a different branching direction within the common prefix region, which result in being excluded from the updated suffix range. As a result, when we walk down in WT_L according to $c[0..|c|-2]$, only the suffixes that are to be included in the updated suffix range remain.

The remaining task is to show that the suffixes within the interval (p_s, p_e) at w' after executing Line 9 correspond to the correct target suffixes. In order to show this, we establish the following lemma, which claims that the ordering of two (encoded) suffixes inverts via LF-mapping iff the larger suffix has at least one turn points (1) beyond the common turn points, (2) within the common prefix, (3) before any non-common turn point for the smaller suffix within the common prefix.

▶ **Lemma 5.** *Let i and j be integers such that $0 \leq i < j \leq n$. Let $T_i = T[\mathsf{SA}[i]..n-1]$ and $t_i = T[\mathsf{SA}[\mathsf{LF}(i)]]$ be the suffix whose rank is i and its previous character on T. Similarly, we define $T_j = T[\mathsf{SA}[j]..n-1]$ and $t_j = T[\mathsf{SA}[\mathsf{LF}(j)]]$. Consider $\tau_i = \mathsf{turnpoint}(t_i, T_i)$ and $\tau_j = \mathsf{turnpoint}(t_j, T_j)$. Let h_T be the length of the longest common prefix of $E(T_i)$ and $E(T_j)$, and let h_τ be the length of the longest prefix of τ_i and τ_j. Then $\mathsf{LF}(i) > \mathsf{LF}(j)$ iff:*

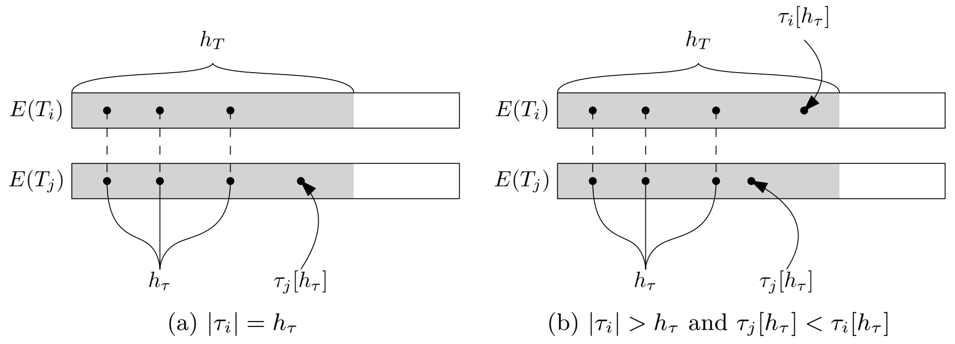


Figure 4 Example of order-inverted cases described in Lemma 5. Suffixes (T_i and T_j), the length of the common prefix of the encoded suffixes (h_T), turn points (τ_i and τ_j), and the length of the common prefix of the turn point sequences (h_τ) are defined as in the lemma. Shaded area indicates the common prefix of the two encoded suffixes. Dots indicate turn points where the changes are made when the corresponding characters are prepended. In order to invert the ordering of the two suffixes via LF-mapping, there must be a turn point for T_j (indicated by $\tau_j[h_\tau]$) between the position of the last common turn points and the earliest position of the end of the longest common prefix and the next non-common turn point for suffix T_i.

1. $|\tau_j| > h_\tau$,
2. $\tau_j[h_\tau] < h_T$, *and*
3. $|\tau_i| = h_\tau$ *or* ($|\tau_i| > h_\tau$ *and* $\tau_j[h_\tau] < \tau_i[h_\tau]$).

Proof. ($\Rightarrow$) We prove by contrapositive. Note that the changes are made only at turn points. For each case, when we assume it is false, then the changes made in $E(T_j)$ are either (i) also made in $E(T_i)$ at the same position or (ii) out of the common prefix, so the ordering is not affected.

1. Let us assume that $|\tau_j| \leq h_\tau$. Then, for all positions $q \in \tau_j$ in which $E(T_j)[q]$ changes, $E(T_i)[q]$ also changes. Therefore $E(t_j \circ T_j)[q+1] = E(t_i \circ T_i)[q+1]$ for such q, which means the ordering is not inverted.
2. Let us assume that $\tau_j[h_\tau] \geq h_T$. Then the first position only $E(T_j)$ changes is out of the common prefix. If $\tau_j[h_\tau] > h_T$, the ordering remains the same because it is determined at position h_T. Let us assume $\tau_j[h_\tau] = h_T$. We have $E(t_i \circ T_i)[h_T+1] = E(T_i)[h_T] < E(T_j)[h_T]$. Because $h_T + 1 \notin E(T_i)[h_T]$, we still have $E(T_i)[h_T] < E(T_j)[h_T] \cup \{h_T+1\} = E(t_j \circ T_j)[h_T+1]$.
3. Let us assume that $|\tau_i| \neq h_\tau$ and ($|\tau_i| \leq h_\tau$ or $\tau_j[h_\tau] \geq \tau_i[h_\tau]$). Since h_τ cannot be greater than $|\tau_i|$, we can rewrite it as: $|\tau_i| > h_\tau$ and ($|\tau_i| = h_\tau$ or $\tau_j[h_\tau] \geq \tau_i[h_\tau]$)$\Leftrightarrow$$|\tau_i| > h_\tau$ and $\tau_j[h_\tau] \geq \tau_i[h_\tau]$. However if $\tau_j[h_\tau] \geq \tau_i[h_\tau]$, the position on $E(T_i)$ where a change is made is earlier than that on $E(T_j)$, therefore $E(T_j)$ cannot become smaller after prepending the corresponding character.

($\Leftarrow$) We can easily see that, for $0 \leq q \leq \tau_j[h_\tau]$, $E(t_i \circ T_i)[q] = E(t_i \circ T_i)[q]$. Note that, for any q, $E(T_j)[q] > E(T_j)[q] \cup \{q+1\}$ because each encoded character, which is a set, contains ∞. And we have $E(t_i \circ T_i)[\tau_j[h_\tau]+1] = E(T_i)[\tau_j[h_\tau]] = E(T_j)[\tau_j[h_\tau]] > E(T_j)[\tau_j[h_\tau]] \cup \{\tau_j[h_\tau]+1\} = E(t_j \circ T_j)[\tau_j[h_\tau]+1]$. See Figure 4. Therefore $E(t_i \circ T_i) > E(t_j \circ T_j)$, which means $\mathsf{LF}(i) > \mathsf{LF}(j)$. ◀

Let us look at the interval (p_s, p_e) at node w of WT_L. The suffixes corresponding to indices $j < p_s$ or $p_e < j$ do not have a common prefix long enough to invert the lexicographical ordering. Therefore, by Lemma 5, the ordering does not change. Each suffix on the left (and right, resp.) side of the interval (p_s, p_e) at node w of WT_L corresponds to a suffix on the left (and right, resp.) side of the interval (p_s, p_e) at node $w' = \mathsf{paired}(w)$ of WT_F. Therefore, the interval at node w' of WT_F that correspond to the interval (p_s, p_e) at node w is still (p_s, p_e). Although the lexicographical ranks of the suffixes within the interval are possibly jumbled, all of them are the suffixes that need to be included in the updated suffix range.

Finally, by the similar reason, during tracing up to the root node of WT_F the interval length does not change. Because, otherwise it means the lexicographical rank is inverted by the turn point out of the common prefix.

4.2 Reporting the Occurrences: Suffix Sampling along Codes

In order to reduce the space requirement to store the suffix array, the existing succinct and compact indexes usually use the sampling method. The standard sampling method is to sample the entries whose value is a multiple of δ. Then we can retrieve the suffix array value by calling $\mathsf{LF}(i)$ at most δ times until reaching any sampled entry. However, this method is no longer efficient for our method. Note that we can compute $\mathsf{LF}(i)$ by walking down from position i in the root of WT_L until we reach the leaf, then tracing up to the root of WT_F; we can easily see its correctness by showing, for any $0 \le i < j \le n$ such that $L[i] = L[j]$, $\mathsf{LF}(i) < \mathsf{LF}(j)$ as a corollary of Lemma 5 because these two suffixes share all the turn points within the common prefix of their encoded strings. This operation takes $\Theta(|L[i]|)$ time, and the length $|L[i]|$ of the bitstring $L[i]$ can be $\Theta(n)$ in the worst case.

Rather, we propose to sample the suffix array based on the actual time taken during the computation of $\mathsf{LF}(i)$. Consider the computation of $\mathsf{LF}(\cdots(\mathsf{LF}(0))) = \mathsf{LF}^n(0)$. Starting at the position 0 at the root node of WT_L, and calling $\mathsf{LF}(\cdot)$ is equivalent to walking down to a leaf node, jumping to WT_F, followed by tracing up in WT_F. We repeat it until we arrive back the starting position, which eventually forms a cycle along the bits of the wavelet tree. We sample the suffix array entries for every δ-th bit along this cycle (see Figure 5). Note that the same suffix array value can be sampled multiple times when the length of its corresponding L-value (bitstring) is longer than δ. Note also that each wavelet tree can be linearized as a single bitvector, so we can store the sampled values from each wavelet tree using an array with a bitvector indicating marked locus. When we retrieve a suffix array value, we repeatedly call $\mathsf{LF}(\cdot)$ until we arrive any locus at which the suffix array is sampled. Then we can get the desired value by adding the retrieved sampled value by the number of jumps from WT_L to WT_F performed during the $\mathsf{LF}(\cdot)$ calls.

It is easy to see that $\mathcal{O}(\delta)$ navigating operations on the wavelet trees are needed until any of marked locus is met. By setting $\delta = \Theta(\lg n)$, we can have the following result.

▶ **Lemma 6.** *For $0 \le i \le n$, $\mathsf{SA}[i]$ can be computed in $\mathcal{O}(\lg n)$ time, and the required space is $\mathcal{O}(R(T))$ bits.*

Proof. The total number of bits to mark the sampled positions is $\mathcal{O}(R(T))$. By setting $\delta = \Theta(\lg n)$, the number of sampled entries of the suffix tree is $\mathcal{O}(R(T)/\lg n)$ and each entry requires $\lceil \lg n \rceil$ bits to be stored. The space required to store the sampled entries of the suffix array is $\mathcal{O}(R(T)/\lg n) \cdot \lceil \lg n \rceil = \mathcal{O}(R(T))$ bits. ◀

5 Non-distinct Case

In this section, we consider the case in which the characters of strings are drawn from an alphabet of size $\sigma(< n)$ so that the same character can appear more than once in a string. We can deal with non-distinct characters by making a slight modifications in order to consider

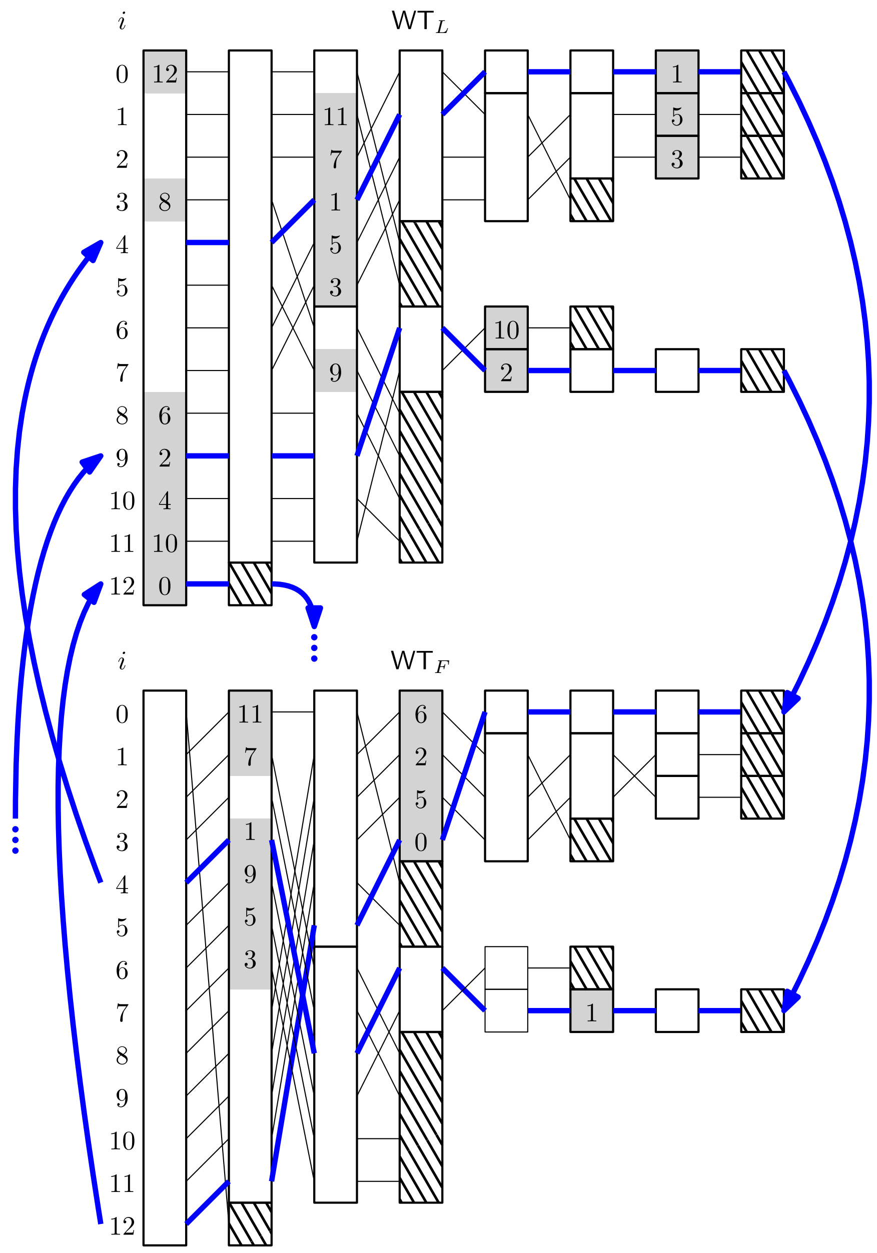


Figure 5 Suffix array sampling along the cycle formed by the visited order of bits by successive calls of LF-mapping ($\delta = 4$). Blue thick lines indicate a part of this cycle. Shaded cells are the marked locus where the sampled suffix array values are stored.

the equality case. Each node has another pointer, which we call a *middle pointer*, in addition to pointers to its left and right child. This middle pointer forms a linked list connecting the nodes with the same key. Then the following modifications are applied:

1. Root insertion: if we find a node with the same key x, we cut the links to its left and right child, if any. Then we remove the existing node, after which the new node is inserted as the root node and links are reestablished properly. Then the removed node is connected to the new node via the middle pointer.
2. Checking $\mathsf{BST}(X) =_v \mathsf{BST}(Y)$: for checking value-identical BSTs, we also consider the node connected via middle pointers.
3. $\mathsf{branch}(x, X)$ in Eq. (1): we add the case of $x = X[\mathsf{vpath}(x, X)[i]]$, in which the corresponding value is $\mathtt{E}$.
4. $C[i]$ in Eq. (5): we replace it with the following to deal with the case of $\mathtt{E}$. More specifically, for every three bits, the code ends if the first bit is 1, otherwise the following two bits represent the content of $\mathsf{branch}(x, X)$; 00: same direction, 01: direction switched, 11: equality.

$$C[i] = \begin{cases} (000)^{l_0}001(000)^{l_1}001\cdots(000)^{l_1}001\cdots(000)^{l_{k-1}}1 & \mathsf{branch}(\cdot) \text{ does not end with } \mathtt{E}. \\ (000)^{l_0}001(000)^{l_1}001\cdots(000)^{l_1}001\cdots(000)^{l_{k-2}}011\ 1 & \mathsf{branch}(\cdot) \text{ ends with } \mathtt{E}. \end{cases} \tag{9}$$

5. $E(x \circ X)$ in Eq. (3): we define it as a string of multisets, and we add the equality case in which we insert the corresponding index twice as follows.

$$E(x \circ X)[i] = \begin{cases} \cdots \\ E(X)[i-1] \cup \{i, i\} & \text{if } x = X[i-1] \\ \cdots \end{cases} \tag{10}$$

After applying the above modifications, we can reuse the searching algorithm. Note that $\mathtt{E}$ can appear only at the end of $\mathsf{branch}(\cdot,\cdot)$ so it does not involve many complications and the useful properties still hold.

For the time and space complexity analysis, we need to know the expected value of $R(X)$, the time taken by performing root insertion with keys $X[|X|-1], \cdots, X[0]$ in order. Note that we do not follow the middle pointer in this insertion process; middle pointers are (conceptually) used only when comparing trees. Hence, we can consider only the nodes connected via pointers to left and right subtrees, so the number of nodes of the binary search tree that are of concern in this insertion process is bounded by σ. Therefore, averaging over all possible binary search trees with σ nodes, we can see that the expected length of the path from the root to a node is $\mathcal{O}(\lg \sigma)$ as in [24, 22]. Therefore, the expected value of $R(X)$ is $\mathcal{O}(|X| \lg \sigma)$. Combining it with the previous results, we can have the following theorem.

▶ **Theorem 7.** *There exists a data structure that uses $\mathcal{O}(n \lg \sigma)$ bits on average, and is capable of counting the number of occurrences in $\mathcal{O}(m \lg \sigma)$ time on average, and reporting each occurrence in $\mathcal{O}(\lg n)$ time.*

Proof. Immediate from Lemmas 3, 4 and 6, and the fact that, for a string X randomly drawn over an alphabet of size σ, the expected value of $R(X)$ is $\mathcal{O}(|X| \lg \sigma)$. ◀

6 Connections to Other Methods

The encoded string is represented based on the retrieved path through the corresponding binary search tree. Deriving a variant by adding some restrictions to the binary search tree can naturally come up. For example, we can restrict the number of nodes by keeping the binary search tree have nodes having a value at most τ where τ is a parameter. In this case, we remove the (leaf) node with a value greater than τ, if any, after inserting a new node. This method is equivalent to checking the order-isomorphism with a size-τ window, as similar to [8]. If $\tau = 1$, it is identical to a heuristic filtering method that uses the up-down signature used in [3]. If we set $\tau = \mathcal{O}(\lg^c n)$ for some constant c, then we can have the required space to be $\mathcal{O}(n \lg\lg n)$ bits on average, which is related to the space requirement of the data structure for order-isomorphic matching with length-restricted patterns described in [10], which $\mathcal{O}(n \lg\lg n)$ in bits (in the worst case). Similarly, we can restrict other measures; e.g. the number of nodes and the maximum depth of the tree. We can also restrict the number of turn points used to encode the suffixes. If we restrict the number of turn points to be 1, the matching problem becomes analogous to Cartesian tree matching [21] because the first turn point indicates the leftmost element that is greater than the prepending character. The proposed method can also be viewed as an extension of pointer sequence matching [16]. We can view each node has pointers to its corresponding turn points. From this perspective, restricting the number of nodes of the binary search tree can be viewed as restricting the length of the pointers; and the restricting the number of turn points is equivalent to restricting the out-degree of a node in the pointer sequence viewpoint.

7 Conclusion

Developing a space-efficient index for order-isomorphic matching has been considered challenging. In this paper, we have presented a new method to index a string regarding this problem. The new suffix representation has been introduced, based on which two arrays L and F are computed using the sorted suffixes and the relation among them. The proposed searching algorithm is quite simple as it just traverses down through the wavelet tree for L and then traces up along the wavelet tree for F. We also presented a new suffix sampling method based on the time taken during the computation of $\mathsf{LF}(\cdot)$, which also resolves an open problem in [17], which asked if reporting an occurrence can be improved when L-values are of variable lengths, possibly as long as $\Theta(n)$. We leave the following open problems:

1. **Suffix array construction:** it has not been shown whether we can efficiently sort the suffixes that are encoded as described in this paper. Even if we use a generalized fast indexing method such as [5, 1], it is not trivial because a single character of an encoded suffix can be a set of $\Theta(n)$ integers in the worst case, so comparing characters may take too much time.
2. **Reducing the space requirement further:** although we achieved the compactness in the average case analysis, the space requirement can be $\mathcal{O}(n\sigma)$ in the worst case. Using a different representation for $C[i]$'s may allow to reduce the worst case space complexity; however, further investigation and analysis are needed. Ultimately, it is interesting to know whether there exists any $n \lg \sigma + o(n \lg \sigma)$-bit index.
3. **Discovering new string matching problems:** in this paper, we define the matching using the binary search tree. In a similar fashion, we can discover new string matching problems using appropriate discrete structures. For example, we can think of a trajectory matching problem, in which two trajectories are defined to be a match if their corresponding triangulations constructed by inserting the points in order are equivalent.

References

1 Amihood Amir and Eitan Kondratovsky. Sufficient Conditions for Efficient Indexing Under Different Matchings. In *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 6:1–6:12, 2019. doi:10.4230/LIPIcs.CPM.2019.6.

2 Brenda S. Baker. Parameterized Pattern Matching: Algorithm and Applications. *Journal of Computer and System Sciences*, 52:28–42, 1996. doi:10.1006/jcss.1996.0003.

3 Tamanna Chhabra, M. Oğuzhan Külekci, and Jorma Tarhio. Alternative Algorithms for Order-Preserving Matching. In *Proceedings of the Prague Stringology Conference*, pages 36–46, 2015.

4 David Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

5 Richard Cole and Ramesh Hariharan. Faster Suffix Tree Construction with Missing Suffix Links. In *Proceedings of the 32nd Annual Symposium on Theory of Computing (STOC)*, pages 407–415, 2000. doi:10.1145/335305.335352.

6 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, and Solon P. Pissis. Order-Preserving Incomplete Suffix Trees and Order-Preserving Indexes. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 84–95, 2013. doi:10.1007/978-3-319-02432-5_13.

7 Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving Indexing. *Theoretical Computer Science*, 628:122–135, 2016. doi:10.1016/j.tcs.2015.06.050.

8 Gianni Decaroli, Travis Gagie, and Giovanni Manzini. A Compact Index for Order-preserving Pattern Matching. *Software: Practice and Experience*, 46(6):1041–1051, 2019.

9 Paolo Ferragina and Giovanni Manzini. Opportunistic Data Structures with Applications. In *Proceedings of the 41st Annual Symposium on Foundation of Computer Science (FOCS)*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

10 Travis Gagie, Giovanni Manzini, and Rossano Venturini. An Encoding for Order-Preserving Matching. In *Proceedings of the 25th European Symposium on Algorithms (ESA)*, pages 38:1–38:15, 2017. doi:10.4230/LIPIcs.ESA.2017.38.

11 Arnab Ganguly, Dhrumil Patel, Rahul Shah, and Sharma V. Thankachan. LF Successor: Compact Space Indexing for Order-Isomorphic Pattern Matching. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 71:1–71:19, 2021. doi:10.4230/LIPIcs.ICALP.2021.71.

12 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Structural Pattern Matching - Succinctly. In *Proceedings of the 28th International Symposium on Algorithms and Computation (ISAAC)*, pages 35:1–35:13, 2017. doi:10.4230/LIPIcs.ISAAC.2017.35.

13 Arnab Ganguly, Rahul Shah, and SharmaV. Thankachan. pBWT: Achieving Succinct Data Structures for Parameterized Pattern Matching and Related Problems. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.

14 G. Jacobson. Space-efficient Static Trees and Graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

15 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving Matching. *Theoretical Computer Science*, 525:68–79, 2014. doi:10.1016/j.tcs.2013.10.006.

16 Sung-Hwan Kim and Hwan-Gue Cho. Indexing Isodirectional Pointer Sequences. In *Proceedings of the 31st International Symposium on Algorithms and Computation (ISAAC)*, pages 35:1–35:15, 2020. doi:10.4230/LIPIcs.ISAAC.2020.35.

17 Sung-Hwan Kim and Hwan-Gue Cho. A Compact Index for Cartesian Tree Matching. In *Proceedings of the 32nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 18:1–18:19, 2021. doi:10.4230/LIPIcs.CPM.2021.18.

18 Sung-Hwan Kim and Hwan-Gue Cho. Simpler FM-index for Parameterized String Matching. *Information Processing Letters*, page 106026, 2021. doi:10.1016/j.ipl.2020.106026.

19 M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A Linear Time Algorithm for Consecutive Permutation Pattern Matching. *Information Processing Letters*, 113(12):430–433, 2013. doi:10.1016/j.ipl.2013.03.015.

20 Gonzalo Navarro. Wavelet Trees for All. *Journal of Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.

21 Sung Gwan Park, Amihood Amir, Gad M. Landau, and Kunsoo Park. Cartesian Tree Matching and Indexing. In *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 16:1–14, 2019. doi:10.4230/LIPIcs.CPM.2019.16.

22 Bruce Reed. The Height of a Random Binary Search Tree. *Journal of the ACM*, 50(3):306–332, 2003. doi:10.1145/765568.765571.

23 Tetsuo Shibuya. Generalization of a Suffix Tree for RNA Structural Pattern Matching. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 393–406, 2000. doi:10.5555/645900.672451.

24 C. J. Stephenson. A Method for Constructing Binary Search Trees by Making Insertions at the Root. *International Journal of Computer & Information Science*, 9:15–29, 1980. doi:10.1007/BF00995807.