Practical Space-Efficient Index for Structural Pattern Matching

Sung-Hwan Kim(✉) and Hwan-Gue Cho

Pusan National University, 46241 Busan, South Korea
{sunghwan,hgcho}@pusan.ac.kr

Abstract. In structural pattern matching, two n-length strings X and Y over Σ are said to match, if there exists a one-to-one function $f : \Sigma \to \Sigma$ such that (i) for $0 \leq i < n$, $f(X[i]) = Y[i]$ and (ii) for any $x, y \in \Sigma$ whose complements are x' and y', respectively, if $f(x) = y$ then $f(x') = y'$. In this paper, we present a $2n \lg \sigma + 2n + o(n)$-bit index for this problem. Although it does not theoretically achieve the succinctness for a general alphabet, the proposed method is more practical and the space requirement can be smaller than the previous succinct solution especially when σ is small. A source code is available at: https://github.com/sunghwank/spmindex.

Keywords: Compact data structure · String matching · Suffix array · FM-index · LF-mapping

1 Motivation

Structural pattern matching was introduced by Shibuya [14,15] to address a string matching problem on RNA sequences regarding their secondary structure. In this problem, matching of two strings is defined differently from the standard string matching problem (see Sect. 2.1). An encoding method was used to transform the suffixes into a certain form so that indexing the encoded suffixes with a suffix tree can resolve the problem. In order to reduce the space requirement, which is excessively large for the suffix tree-based indexing methods, Beal and Adjeroh [1] proposed the use of a suffix array as well as its construction method. However, indexing an n-length string still requires $\Theta(n \lg n)$ space in bits, which is quite far from $n \lg \sigma$ bits, the space required to represent the string, where σ is the alphabet size.

Recently, Ganguly *et al.* [5] presented the first succinct data structure for this problem, which requires $n \lg \sigma + \mathcal{O}(n)$ bits of space. Although this method dramatically reduces the space requirement, it relies on several data structures that are theoretically optimal but hard to implement in practice, such as a multiary wavelet tree [3,7,8], and a fully-functional succinct tree supporting constant-time queries [12,13].

This paper is devoted to present a data structure, which is practically implementable as well as efficient in time and space. Comparison with the existing works is shown in Table 1. The proposed index uses $2n \lg \sigma + 2n + o(n)$ bits where

C. Bazgan and H. Fernau (Eds.): IWOCA 2022, LNCS 13270, pp. 369–382, 2022.
https://doi.org/10.1007/978-3-031-06678-8_27

Table 1. Comparison with other works.

Method	Space (in bits)	Query time (counting)
Suffix tree [14,15]	$\Theta(n \lg n)$	$\mathcal{O}(m)$
Suffix array [1]	$\Theta(n \lg n)$	$\mathcal{O}(m + \lg n)$
sBWT [5]	$n \lg \sigma + \mathcal{O}(n)$	$\mathcal{O}(m \lg \sigma)$
Proposed	$2n \lg \sigma + 2n + o(n)$	$\mathcal{O}(m \lg \sigma)$

n is the text length and σ is the alphabet size, and it can count the number of occurrences of an m-length pattern in $\mathcal{O}(m \lg \sigma)$ time. It is also practical in the sense that it uses bitvector dictionaries [2,9], wavelet trees [8] and range maximum query index [4], which have practical implementations available in public software libraries such as `sdsl-lite` [6].

As mentioned in [5], the main challenge in using the suffix-encoding method described in [14] for space-efficient indexing is that prepending a single character can affect more than one positions in its encoded string. In this paper, we address this issue by transforming a structural string (s-string) into a pointer sequence of double length so that a single prepending operation can affect at most one position, which is a different approach from that of [5]. We develop an index on this transformed pointer sequence using its space-efficient representation. As an overview, the proposed method can be described the following:

1. We represent an n-length s-string as a $(2n)$-length pointer sequence such that pointers at even (odd, resp.) positions refer to the next occurrence of the character (equal-group character, resp.).
2. We construct two suffix arrays by sorting the suffixes starting at even and odd positions separately.
3. The searching procedure is performed by navigating these two suffix arrays alternatingly; the so-called *LF-mapping* of a suffix at one suffix array is defined to be a suffix at the other suffix array.
4. The LF-mappings can be represented using four arrays L_{even}, F_{odd}, L_{odd}, and F_{even}, which can be stored in $2n \lg \sigma + 2n + o(n)$ bits in total; using these arrays, the LF-mappings can be simply computed in $\mathcal{O}(\lg \sigma)$ time.

The remainder of this paper is organized as follows. In Sect. 2, we briefly review some backgrounds needed to develop our proposed method. In Sect. 3, we present a pointer sequence representation for the structural pattern matching problem. Section 4 describes how to organize the proposed index structure, and the searching algorithm is presented in Sect. 5. Section 6 concludes the paper.

2 Preliminaries

2.1 Structural Pattern Matching

We describe the structure pattern matching introduced in [14]. In the original paper, a string consists of two types of characters: (i) static characters for

the exact matching, and (ii) parameterized characters for a more sophisticated matching. In this paper, we consider only parameterized characters for brevity. Nevertheless, we emphasize that our proposed method can easily be applied to the original problem.

Let $T[0..n-1]$ be an n-length structural string (s-string) over alphabet $\Sigma = \{0, \cdots, \sigma-1\}$. We use the 0-based index. We have a one-to-one function $\mathsf{compl} : \Sigma \to \Sigma$ that represent the association among characters in Σ. For each $x \in \Sigma$, x is associated with its complement $\mathsf{compl}(x) \in \Sigma$. And for any $x, y \in \Sigma$, if $\mathsf{compl}(x) = y$ then $\mathsf{compl}(y) = x$. For simplicity, we assume that the alphabet size $\sigma = |\Sigma|$ is a multiple of 2, and $x \neq \mathsf{compl}(x)$. To represent the relationship defined via $\mathsf{compl}(\cdot)$, we can also use a function $g : \Sigma \to \{0, \cdots, \sigma/2-1\}$ such that for $x, y \in \Sigma$ $g(x) = g(y)$ iff $x = y$ or $x = \mathsf{compl}(y)$. We say x and y such that $g(x) = g(y)$ are equal-group characters. Two s-strings X and Y are said to match if there exists a one-to-one function $f : \Sigma \to \Sigma$ such that (i) for $0 \leq i < n$, $f(X[i]) = Y[i]$ and (ii) for any $x, y \in \Sigma$, if $f(x) = y$ then $f(\mathsf{compl}(x)) = \mathsf{compl}(y)$. For example, let $\Sigma = \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$ and $g(\mathtt{w}) = g(\mathtt{x})$ and $g(\mathtt{y}) = g(\mathtt{z})$. Then `wyxxwyzw` matches `zwyyzwxz`, while it does not match `yxwwyxzy`.

2.2 Pointer Sequence Matching

In this subsection, we briefly review the pointer sequence matching described in [10]. Although the description below may be slightly different from that in the original paper in detail, the basic idea is essentially the same.

In the pointer sequence matching problem, a string is a sequence of pointers. Each pointer is either a null pointer or one refers to another element among those in its right-hand side. We represent a null pointer by a symbol ∞. We represent a pointer referring to another element by its length so that the element at position i refers to the element at position $i + X[i]$ if $X[i] \neq \infty$.

Definition 1 (Pointer sequence). *A sequence $X[0..n-1]$ of length n is a pointer sequence if, for $0 \leq i < n$, $X[i] \in \{1, \cdots, n-i-1\} \cup \{\infty\}$.*

With this representation, we say two equal-length pointer sequences *match* if they are exactly the same. To define a pattern matching problem on pointer sequences, we define a substring of a pointer sequence. Taking a substring from a pointer sequence not only copies the target part but also converts pointers going to the outside of the taken part into null pointers.

Definition 2 (Substring of a pointer sequence). *A substring $Y = X[i..j]$ of X from position i to position j is defined as follows: for $0 \leq k \leq j-i$.*

$$Y[k] = \begin{cases} X[i+k] & \text{if } X[i+k] \leq j-i-k, \\ \infty & \text{otherwise.} \end{cases} \tag{1}$$

For indexing a pointer sequence, we transform it into an encoded form, which is a sequence of sets. The encoded sequence $E(X)$ of an n-length pointer sequence X is defined as follows:

$$E(X)[i] = \{1 \leq j \leq i \mid X[i-j] = j\} \tag{2}$$

An element of an encoded sequence represents the set of elements pointing to it. To define the lexicographical order among encoded sequences, we define the ordering on their elements, which are sets. As we will see, an element of an encoded sequence handled in this paper is either the empty set $\emptyset$ or a singleton $\{x\}$ for some integer x. We define the ordering of sets (which are elements of encoded sequences) as follows: $A < B$ iff (i) $A \neq \emptyset$ and $B = \emptyset$ or (ii) $A = \{a\}$ and $B = \{b\}$ are singletons such that $a < b$.

We can index the set of encoded suffixes in $2n \lg n + 2n + o(n)$ bits although we do not use it directly in this paper. Rather, we develop a more space-efficient representation for the structural pattern matching problem.

Proposition 1 ([10]). *For an n-length pointer sequence, there exists a data structure that uses $2n \lg n + 2n + o(n)$ bits, and can count the number of occurrences of an m-length pattern in $\mathcal{O}(m \lg n)$ time.*

2.3 Building Blocks

The proposed index uses several well-known data structures as its building blocks.

Bitvector. For an n-length bitvector $A[0..n-1]$, a data structure that supports the following operations in $\mathcal{O}(1)$ time can be represented in $n + o(n)$ bits [2,9].

1. $A.\mathsf{rank}_x(i)$: the number of occurrences of x in $A[0..i]$.
2. $A.\mathsf{select}_x(j)$: the position of the j-th occurrence of x on A.

We also define $A.\mathsf{rank}_x(i,j) = A.\mathsf{rank}_x(j) - A.\mathsf{rank}_x(i-1)$.

Wavelet Tree. A wavelet tree of an n-length string $A[0..n-1]$ over an alphabet of size σ is a data structure that supports the following operations in $\mathcal{O}(\lg \sigma)$ time using $n \lg \sigma + o(n)$ bits [3,7,11].

1. $A(i)$: accessing $A[i]$.
2. $A.\mathsf{rank}_x(i,j)$: the number of occurrences of x in $A[i..j]$.
3. $A.\mathsf{rank\_ge}_x(i,j)$: the number of occurrences of characters that are greater than or equal to x in $A[i..j]$.
4. $A.\mathsf{select}_x(j)$: the position of the j-th occurrence of x on A.

Range Maximum Query. A range maximum query (i,j) on array $A[0..n-1]$ is to ask the index of the maximum element among $A[i..j]$, which can be performed in $\mathcal{O}(1)$ time with a $2n + o(n)$-bit data structure [4]:

1. $A.\mathsf{rMq}(i,j) = \arg\max_{i \leq k \leq j} A[k]$.

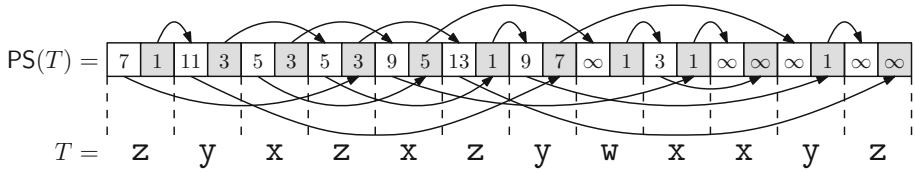

Fig. 1. Pointer sequence representation $\mathsf{PS}(T)$ of $T = \mathtt{zyxzxzywxxyz}$ where $\Sigma = \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$, $g(\mathtt{w}) = g(\mathtt{x})$ and $g(\mathtt{y}) = g(\mathtt{z})$. Each square represents an element of the pointer sequence. The integers inside the squares indicate the pointer lengths and ∞s indicate null pointers. White ones are pointers to its next occurrence, and shaded ones are pointers to the next occurrence of its equal-group character.

3 Pointer Sequence Representation

The basic idea of this paper is to resolve the structural pattern matching problem by solving the matching problem on pointer sequences. In this section, we present a pointer sequence representation for structural pattern matching, which will be used for developing an index structure. More specifically, we represent an n-length s-string as a $(2n)$-length pointer sequence. Each character of an s-string is corresponding to two pointers. One pointer points to the position of the nearest occurrence of the character at the current position, and the following pointer points to the position of the nearest occurrence of its equal-group character. Let $\nu(i)$ and $\mu(i)$ be the distance to the next occurrence of $T[i]$ and $T[i]$'s equal-group character, respectively. More formally, for $0 \leq i < n$,

$$\nu(i) = \min_{j>i}\{j - i : T[j] = T[i]\} \cup \{\infty\} \tag{3}$$

$$\mu(i) = \min_{j>i}\{j - i : g(T[j]) = g(T[i])\} \cup \{\infty\} \tag{4}$$

For an n-length s-string T, we define its pointer sequence representation $\mathsf{PS}(T)$ as follows: for $0 \leq i < 2n$,

$$\mathsf{PS}(T)[i] = \begin{cases} 2\nu(\frac{i}{2}) + 1 & \text{if } i = 0 \mod 2 \\ 2\mu(\frac{i-1}{2}) - 1 & \text{if } i = 1 \mod 2 \end{cases} \tag{5}$$

As an example, the pointer sequence representation of an s-string $T = \mathtt{zyxzxzywxxyz}$ with the complement relationship $g(\mathtt{w}) = g(\mathtt{x})$ and $g(\mathtt{y}) = g(\mathtt{z})$ is given in Fig. 1. It is easy to see that this pointer sequence representation can be used for solving the structural pattern matching problem.

Observation 1. *For s-strings $T, P \in \Sigma^*$, let $\mathsf{PS}(T)$ and $\mathsf{PS}(P)$ be their pointer sequence representations. For $0 \leq i \leq |T| - |P|$, P matches T at position i if and only if $\mathsf{PS}(P)$ matches $\mathsf{PS}(T)$ at position $2i$.*

One can directly apply the indexing method in [10] to this representation to obtain a $4n \lg n + \mathcal{O}(n)$-bit data structure that can compute the number of

occurrences in $\mathcal{O}(m \lg n)$ time. One of the goal of this paper is to reduce the space requirement into $\mathcal{O}(n \lg \sigma)$ bits. The $\lg n$ factor in the space requirement comes from the representation of the pointer length. In [10], the pointers are represented by their lengths, which is $\mathcal{O}(n)$. This is the alphabet size of the underlying sequence on which the wavelet trees are built, which results in the $\lg n$ factor. To reduce this into $\lg \sigma$, we need to represent these sequences in more compact values within a range of $\mathcal{O}(\sigma)$.

One may also notice that we do not consider occurrences of $\mathsf{PS}(P)$ at odd positions $2i + 1$ on $\mathsf{PS}(T)$, despite the fact that there may be (false positive) occurrences of $\mathsf{PS}(P)$ at odd positions even if P does not match T there. When we apply the method in [10], it is inevitable to involve an additional filtering method to remove these false positives, which produces a non-negligible overhead. We will address this problem in the next section by constructing suffix arrays for suffixes at even and odd positions separately.

4 Data Structure

In this section, we present a data structure for structural pattern matching. We build two suffix arrays using the corresponding pointer sequence, one for the suffixes starting at even positions (even suffixes), the other one for those starting at odd positions (odd suffixes). Then we define integer arrays that will be used for the searching algorithm we will describe in the next section.

4.1 Suffix Arrays

For the pointer sequence $\mathsf{PS}(T)$ of an n-length s-string T, let $\mathcal{S}_{even} = \{\mathsf{PS}(T)[2i..] : 0 \leq i \leq n\}$ be the set of the suffixes of $\mathsf{PS}(T)$ that start at even positions; note that $\mathcal{S}_{even}$ includes the empty string $\epsilon = \mathsf{PS}(T)[2n..]$, which acts as the termination symbol as usually assumed in many other string indexing methods. We define the suffix array SA_{even} for the suffixes $\mathcal{S}_{even}$ using their encoded form; i.e. $\mathsf{SA}_{even}(i) = j$ iff there are i encoded suffixes in $\mathcal{S}_{even}$ that are smaller than $E(\mathsf{PS}(T)[2j..])$. Similarly, we define SA_{odd} from the set $\mathcal{S}_{odd} = \{\mathsf{PS}(T)[2i + 1..] : 0 \leq i \leq n\}$ of suffixes of $\mathsf{PS}(T)$ that start at odd positions; note that $\mathcal{S}_{odd}$ also contains the empty string as $\mathcal{S}_{even}$ does. We also define the inverse function of the two suffix arrays such that $\mathsf{SA}^{-1}_{even}(\mathsf{SA}_{even}(i)) = i$ and $\mathsf{SA}^{-1}_{odd}(\mathsf{SA}_{odd}(i)) = i$ for $0 \leq i \leq n$.

Recall that the *LF-mapping* is the one-to-one function that maps a suffix starting at position j into its previous suffix starting at position $j-1$ in terms of their lexicographical ranks. Recall also that we defined suffix arrays separately for even and odd suffixes. The previous suffix of an even suffix is an odd suffix, and vice versa. Hence we have to define the LF-mappings to be functions from even suffixes to odd suffixes, and the other way around.

$$\mathsf{LF}_{eo}(i) = \mathsf{SA}^{-1}_{odd}(\mathsf{SA}_{even}(i) + n \mod (n+1)) \tag{6}$$

$$\mathsf{LF}_{oe}(i) = \mathsf{SA}^{-1}_{even}(\mathsf{SA}_{odd}(i)) \tag{7}$$

4.2 F and L Arrays

In this subsection, we define four integer arrays F_{even}, F_{odd}, L_{even}, and L_{odd}, which compactly store the information used to compute the LF-mappings and to update suffix ranges in the searching algorithm.

For $0 \leq i < n$, let us define $\nu(i)$ and $\mu(i)$ as Eqs. 3 and 4. We define two n-length arrays D_{even} and D_{odd} as follows. $D_{even}(i)$ indicates whether $\nu(i) = \mu(i)$ or not. D_{odd} indicates the number of distinct groups appearing between position i and $\min\{i + \mu(i), n\}$ (both exclusive). More formally, these two sequences are defined as follows.

$$D_{even}(i) = \begin{cases} 0 & \text{if } \nu(i) = \mu(i) \\ 1 & \text{otherwise.} \end{cases} \tag{8}$$

$$D_{odd}(i) = |\{g(T[j]) : i < j < \min\{i + \mu(i), n\}\}| \tag{9}$$

Now we define F_{even} and F_{odd}. Note that F_{even} represents the pointers at even positions of the pointer sequence representation and each element $F_{even}(i)$ corresponds to each entry $\mathsf{SA}_{even}(i)$ of the suffix array for even positions; similarly, F_{odd} represents the pointers at odd positions and corresponds to entries of SA_{odd}. F_{even} and F_{odd} are $(n+1)$-length arrays, which are defined as follows: $F_{even}(0) = F_{odd}(0) = -1$, and for $0 < i \leq n$,

$$F_{even}(i) = D_{even}(\mathsf{SA}_{even}(i)) \tag{10}$$

$$F_{odd}(i) = D_{odd}(\mathsf{SA}_{odd}(i)) \tag{11}$$

We also define L_{even} and L_{odd} as permuted arrays of F_{odd} and F_{even}, respectively, as follows: for $0 \leq i \leq n$,

$$L_{even}(i) = F_{odd}(\mathsf{LF}_{eo}(i)) \tag{12}$$

$$L_{odd}(i) = F_{even}(\mathsf{LF}_{oe}(i)) \tag{13}$$

4.3 Computing $\mathsf{LF}_{eo}(i)$ and $\mathsf{LF}_{oe}(i)$

In this subsection, we present how $\mathsf{LF}_{eo}(i)$ and $\mathsf{LF}_{oe}(i)$ can be computed using the four arrays L_{even}, F_{odd}, L_{odd}, and F_{even}. The following lemma shows the key property we can use for computing the LF-mappings using the correspondence between the arrays (Fig. 2).

Lemma 1. *For $0 \leq i < j \leq n$,*

1. *$L_{even}(i) = L_{even}(j)$ then $\mathsf{LF}_{eo}(i) < \mathsf{LF}_{eo}(j)$.*
2. *$L_{odd}(i) = L_{odd}(j)$ then $\mathsf{LF}_{oe}(i) < \mathsf{LF}_{oe}(j)$.*

i	SA	LF	F	L	
0	12	1	−1	0	empty string
1	11	2	0	0	ϕ ϕ
2	8	3	0	0	ϕ ϕ {1} {3} ϕ ϕ {1} ϕ
3	10	7	1	1	ϕ ϕ {1} ϕ
4	7	8	1	1	ϕ ϕ {1} ϕ {1} {3} ϕ ϕ {1} ϕ
5	5	9	1	1	ϕ ϕ {1} ϕ ϕ ϕ {1} ϕ {1} {3} {7} {9} {1} {13}
6	0	0	1	−1	ϕ ϕ {1} ϕ ϕ ϕ {3} {7} {3} {5} {3} {5} {1} {11} {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}
7	9	4	0	0	ϕ ϕ ϕ ϕ {1} ϕ
8	6	5	0	0	ϕ ϕ ϕ ϕ {1} ϕ {1} {3} {7} {9} {1} ϕ
9	4	10	1	1	ϕ ϕ ϕ ϕ {1} ϕ {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}
10	3	11	0	1	ϕ ϕ ϕ ϕ {3} {5} {1} ϕ {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}
11	2	12	0	1	ϕ ϕ ϕ ϕ {3} {5} {3} {5} {1} ϕ {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}
12	1	6	1	0	ϕ ϕ ϕ ϕ {3} ϕ {3} {5} {3} {5} {1} {11} {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}

(a) Sorted even suffixes with SA_{even}, LF_{eo}, F_{even}, and L_{even}

i	SA	LF	F	L	
0	12	0	−1	−1	empty string
1	11	1	0	0	ϕ
2	10	3	0	1	ϕ {1} ϕ
3	7	4	0	1	ϕ {1} ϕ {1} {3} ϕ ϕ {1} ϕ
4	8	2	0	0	ϕ {1} ϕ ϕ ϕ {1} ϕ
5	5	5	0	1	ϕ {1} ϕ ϕ ϕ {1} ϕ {1} {3} {7} {9} {1} ϕ
6	0	6	0	1	ϕ {1} ϕ ϕ ϕ {3} ϕ {3} {5} {3} {5} {1} {11} {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}
7	9	7	1	0	ϕ ϕ ϕ {1} ϕ
8	6	8	1	0	ϕ ϕ ϕ {1} ϕ {1} {3} {7} ϕ {1} ϕ
9	4	9	1	1	ϕ ϕ ϕ {1} ϕ {5} ϕ {1} ϕ {1} {3} {7} {9} {1} {13}
10	3	10	1	0	ϕ ϕ ϕ {3} ϕ {1} ϕ {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}
11	2	11	1	0	ϕ ϕ ϕ {3} ϕ {3} {5} {1} ϕ {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}
12	1	12	1	1	ϕ ϕ ϕ {3} ϕ {3} {5} {3} {5} {1} ϕ {5} ϕ {1} {9} {1} {3} {7} {9} {1} {13}

(b) Sorted odd suffixes with SA_{odd}, LF_{oe}, F_{odd}, and L_{odd}

Fig. 2. Sorted (encoded) suffixes for $T =$ `zyxzxzywxxyz` and the related information used in the proposed data structure (white: even positions, gray: odd positions). The searching algorithm navigates two suffix arrays alternatingly by updating suffix ranges iteratively.

Proof. Observe that prepending a pointer at the beginning of a pointer sequence affects at most one position in terms of its encoded form. More specifically, consider pointer sequences X and Y such that Y can be obtained by prepending a pointer of length l at the beginning of X. Then, for $0 \leq i < |Y|$,

$$E(Y)[i] = \begin{cases} \emptyset & \text{if } i = 0, \\ E(X)[i-1] \cup \{l\} & \text{if } i = l, \\ E(X)[i-1] & \text{otherwise.} \end{cases} \tag{14}$$

We call the position on X to which a new pointer to refer *a changing position*: i.e. it refers to position $l-1$ of X in the above equation. Consider two pointer sequences, and a pointer for each sequence is to be prepended. Their relative order changes by these new pointers only if the changing position of the lexicographically greater sequence is earlier and the changing position is within their common prefix. We claim that it is impossible for two suffixes having the same L-value. Based on this observation, we prove each proposition as follows.

1. Let $k = \mathsf{lcp}(E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{even}(i)..]), E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{even}(j)..])$ be the length of the longest common prefix of the (encoded) suffixes on SA_{even} whose ranks are i and j. Let $d = |\{g(T[\mathsf{SA}_{even}(i) + p]) : 0 \leq p < \lfloor \frac{k}{2} \rfloor\}$ be the number of distinct groups in this longest common prefix. If $L_{even}(i) < d$ the length of the newly prepended pointers are the same, which implies the relative order does not change because the changing positions of the encoded sequences are the same. If $L_{even}(i) > d$, the changing position is out of the longest common prefix, and the relative order is determined by $E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{even}(i)..])[k] < E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{even}(j)..])[k]$, which do not change. Before considering the case of $L_{even}(i) = d$, note that $E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{even}(i)..])[k] \leq \{k\} \neq \emptyset$ because $E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{even}(i)..])[k] < E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{even}(j)..])[k]$ and $\emptyset$ is considered to be the greatest. Therefore the changing position of suffix i is not k. Even if the changing position of suffix j is k, $\emptyset$ is substituted by $\{k+1\}$, which does not change the relative order.
2. Let $k = \mathsf{lcp}(E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{odd}(i) + 1..]), E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{odd}(j) + 1..])$ be the length of the longest common prefix of suffixes whose ranks are i and j, respectively. Note that, at this moment the group of the character is already determined (by the pointer at the odd position on the text pointer sequence), the pointer to be newly prepended here indicates which of the two characters in the group is actually prepended; therefore we have two candidates for the change position for each suffix. Let $c_i^{(1)}$ and $c_i^{(2)}$ be two candidate positions for suffix whose rank is i such that $c_i^{(1)} < c_i^{(2)}$. Similarly, we define $c_j^{(1)}$ and $c_j^{(2)}$ for suffix whose rank is j. Note that if $c_i^{(1)} < k$ or $c_j^{(1)} < k$, then $c_i^{(1)} = c_j^{(1)}$. Similarly, if $c_i^{(2)} < k$ or $c_j^{(2)} < k$, then $c_i^{(2)} = c_j^{(2)}$. Let l_i and l_j be the changing position of suffixes whose ranks are i and j, respectively. If $L_{odd}(i) = L_{odd}(j) = 0$, then $l_i = c_i^{(1)}$ and $l_j = c_j^{(1)}$. Similarly, if $L_{odd}(i) = L_{odd}(j) = 1$, $l_i = c_i^{(2)}$ and $l_j = c_j^{(2)}$. Therefore, if $l_i < k$ or $l_j < k$ then we have $l_i = l_j$. Let us assume that the relative order changes after prepending corresponding pointers to these suffixes. Then it must be both $l_j < k$ and $l_j < l_i$. However, if $l_j < k$, we have $l_i = l_j$. Contradiction. □

From the order-preserving property described in Lemma 1, we can simply compute $\mathsf{LF}_{eo}(i)$ and $\mathsf{LF}_{eo}(i)$ using the rank and select operations as follows.

Corollary 1. *For* $0 \leq i \leq n$,

1. $\mathsf{LF}_{eo}(i) = F_{odd}.\mathsf{select}_x(L_{even}.\mathsf{rank}_x(i))$ *where* $x = L_{even}(i)$.
2. $\mathsf{LF}_{oe}(i) = F_{even}.\mathsf{select}_x(L_{odd}.\mathsf{rank}_x(i))$ *where* $x = L_{odd}(i)$.

4.4 Implementation

In this subsection, we describe how the proposed data structure is organized. More specifically, we show the following lemma.

Lemma 2. *There exists a data structure that uses* $2n \lg \sigma + 2n + o(n)$ *bits and supports the following operations in* $\mathcal{O}(\lg \sigma)$ *time for any* $0 \leq i, j \leq n$, $a \in \{0, \cdots, \sigma - 1\}$, *and* $b \in \{0, 1\}$:

1. $L_{even}(i)$: *access to the value* $L_{even}(i)$.
2. $L_{odd}(i)$: *access to the value* $L_{odd}(i)$.
3. $L_{even}.\mathsf{rank}_a(i, j)$: *the number of occurrences of* a *in* $L_{even}[i..j]$.
4. $L_{even}.\mathsf{rank\_ge}_a(i, j)$: *the number of elements greater than or equal to* a *in* $L_{even}[i..j]$.
5. $L_{odd}.\mathsf{rank}_b(i, j)$: *the number of occurrences of* b *in* $L_{odd}[i..j]$.
6. $F_{even}.\mathsf{select}_b(i)$: *the position of the* i*-th occurrence of* b *in* F_{even}.
7. $F_{odd}.\mathsf{select}_a(i)$: *the position of the* i*-th occurrence of* a *in* F_{odd}.
8. $\mathsf{LF}_{eo}.\mathsf{rMq}(i, j)$: *the index of the maximum element among* $\mathsf{LF}_{eo}(i), \cdots, \mathsf{LF}_{eo}(j)$.

Proof. In short, we build wavelet trees on L_{even} and F_{odd}, and rank/select dictionaries for bitvectors on L_{odd} and F_{even}, and a range maximum query index on LF_{eo}. More specifically,

- We build wavelet trees on L_{even} and F_{odd}, which can support the operations related on these arrays in $\mathcal{O}(\lg \sigma)$ time. Note that the alphabet size of these arrays is $\sigma/2$, thereby each wavelet tree uses $n \lg(\sigma/2) + o(n) = n \lg \sigma - n \lg 2 + o(n) = n \lg \sigma - n + o(n)$ bits.
- For L_{odd} and F_{even}, we can observe that these arrays consist of 0 and 1 except the unique -1. Thus storing the index at which -1 appears using $\mathcal{O}(\lg \sigma)$ bits, we can consider them as bitvectors, which can support rank and select queries in $\mathcal{O}(1)$ time using $n + o(n)$ bits each.
- Range maximum query on LF_{eo} requires $2n + o(n)$ bits, and can answer to a range maximum query in $\mathcal{O}(1)$ time.

As a result, the total space requirement is $2n \lg \sigma + 2n + o(n)$ bits. □

5 Searching Algorithm

In this section, we present the searching algorithm on the proposed data structure. As other methods based on suffix arrays do, we represent the occurrences of a pattern as a contiguous interval on the suffix array, which is called a *suffix range*. Recall that we have two suffix arrays SA_{even} and SA_{odd}, and only suffix ranges on SA_{even} should be the final answer. To distinguish a suffix range on SA_{even} from one on SA_{odd}, we call a suffix range on SA_{even} *a real suffix range* and one on SA_{odd} *an imaginary suffix range*.

For a pattern P, its suffix range on SA_{even} is a pair of indices (p_s, p_e) such that for any $p_s \leq i \leq p_e$ $E(\mathsf{PS}(T)[2 \cdot \mathsf{SA}_{even}(i)..])$ has a prefix $E(\mathsf{PS}(P))$. Note that a character of an s-string is represented as two pointers. Let $x \in \Sigma$ be a character, suppose we are to prepend x to the beginning of P. Let l_1 and l_2 be the lengths of the first two pointers $\mathsf{PS}(xP)$. These are what we are about to prepend to $\mathsf{PS}(P)$ to compute the updated suffix range for xP. By prepending the latter pointer at the beginning of $\mathsf{PS}(P)$, we obtain an imaginary suffix range on SA_{odd}. Then, prepending the other pointer completes $\mathsf{PS}(xP)$, and we obtain a real suffix range on SA_{even} via a proper update procedure, which is the desired (real) suffix range for the updated pattern xP.

Our searching algorithm iteratively updates the suffix array starting from the suffix range $(p_s, p_e) = (0, n)$ on SA_{even} of the empty string, which represents all the suffixes starting at even positions. Each iteration we prepend each character of the pattern in the backward searching fashion. It is equivalent to prepend the corresponding two pointers in the pointer sequence representation. Thus the update procedure consists of two phases, in which we update the current (real) suffix range into an imaginary suffix range on SA_{odd}, followed by updating it into a real suffix range on SA_{even}. The algorithm for updating a suffix range is given in Algorithm 1.

For a suffix range (p_s, p_e) for a pointer sequence X, let l be the length of the pointer to be prepended to the beginning of X. Let (p'_s, p'_e) be the suffix range the pointer sequence after prepending the pointer. For an index $p_s \leq i \leq p_e$, we say i is a *target suffix* if $p'_s \leq \mathsf{LF}^*(i) \leq p'_e$ where $\mathsf{LF}^*(i) = \mathsf{LF}_{eo}(i)$ if $|X|$ is a multiple of 2, $\mathsf{LF}^*(i) = \mathsf{LF}_{oe}(i)$ otherwise. Now we can describe a suffix update procedure as (i) identifying the target suffixes within the current suffix range, followed by (ii) applying the LF-mapping to the identified suffixes. The remainder of the section is devoted to show the following theorem about the correctness of the algorithm.

Theorem 1. *Given a suffix range (p_s, p_e) for a pattern $P \in \Sigma^*$ and $x \in \Sigma$, Algorithm 1 correctly computes the updated suffix range (p'_s, p'_e) for pattern xP in $\mathcal{O}(\lg \sigma)$ time, provided i_g, i_c, and a can be computed in $\mathcal{O}(\lg \sigma)$ time.*

By Lemma 2, all the operations in Algorithm 1 related to the arrays take $\mathcal{O}(\lg \sigma)$ time. Considering a character $x \in \Sigma$ to be prepended to the beginning of the currently searched pattern P during the update procedure, we divide it into two cases: (i) P has x or $\mathsf{compl}(x)$, and (ii) P does not have any of them.

Algorithm 1. Update a suffix range.

```
 1: procedure UPDATE(P: current pattern s-string, (p_s, p_e): suffix range, x: character)
 2:     if either x or compl(x) appeared in P then
 3:         i_c ← min{0 ≤ j < |P| : P[i] = x} ∪ {|P|}.
 4:         i_g ← min{0 ≤ j < |P| : g(P[i]) = g(x)}.
 5:         a ← |{g(P[j]) : 0 ≤ j < i_g}|.
 6:         c ← L_even.rank_a(p_s, p_e).
 7:         p_e ← F_odd.select_a(L_even.rank_a(0, p_e)).
 8:         p_s ← p_e − c + 1.
 9:         b ← 0 if i_c = i_g, 1 otherwise.
10:         c ← L_odd.rank_b(p_s, p_e).
11:         p_e ← F_even.select_b(L_odd.rank_b(0, p_e)).
12:         p_s ← p_e − c + 1.
13:     else
14:         i* ← LF_eo.rMq(p_s, p_e).
15:         l ← L_even(i*).
16:         a ← |{g(P[j]) : 0 ≤ j < |P|}|.
17:         c ← L_even.rank_ge_a(p_s, p_e).
18:         p_e ← F_odd.select_l(L_even.rank_l(i*)).
19:         p_s ← p_e − c + 1.
20:     end if
21:     return (p_s, p_e).
22: end procedure
```

Case 1: At least one of x and compl(x) appear in P. Lines 3–12 handle this case. Let i_c be the position of the first occurrence of x on P; if x does not appear on P, then $i_c = |P|$. Let i_g be the position of first occurrence of x's equal-group character (either x or compl(x)), which must exist. Let a be the number of distinct groups in $P[0..i_g - 1]$. This value can be computed in $\mathcal{O}(\lg \sigma)$ time, if we keep a balanced binary tree keyed by positions of the first occurrences of each group, and the leftmost position of each character as similar to that described in [5]. Then the indices of the target suffixes on SA_{even} must be the suffixes having a as their L_{even}-values. The number of these suffixes can be counted by $c = L_{even}.\mathsf{rank}_a(p_s, p_e)$. And the last index of the suffix is located by $i_e = L_{even}.\mathsf{rank}_a(0, p_e)$. We can find the corresponding index $\mathsf{LF}_{eo}(i_e)$ on $\mathsf{select}_a(i_e)$, update p_e to it. Using the number c of target suffixes, we can update p_s to be $p_e - c + 1$.

Now (p_s, p_e) is an imaginary suffix range on SA_{odd}. Let b be a binary number such that $b = 0$ if $P[i_g] = x$ (i.e. $i_c = i_g$), $b = 1$ otherwise. Similarly, the target suffixes are those having b as their L_{odd}-values. We can update the suffix range correspondingly in a similar way.

Case 2: Neither x nor compl(x) appears in P. Lines 14–19 handle this case, which is a little more difficult. Let a be the number of distinct groups in P. The target suffixes are those within the current suffix range that have a L_{even} value of at least a. We can count the number c of these suffixes via $L_{even}.\mathsf{rank\_ge}_a(p_s, p_e)$. It is easy to see, for $p_s \le i, j \le p_e$ such that $L_{even}(i) <$

$a \leq L_{even}(j)$, $\mathsf{LF}_{eo}(i) < \mathsf{LF}_{eo}(j)$. This is because, for such suffix i, the changing position is one of the first $|P|$ positions, which is definitely within the common prefix of the encoded sequences. Since such suffix j has a changing position beyond that of the suffix i, the suffix i becomes smaller after prepending the corresponding pointer. Therefore $\mathsf{LF}_{eo}(i) < \mathsf{LF}_{eo}(j)$. As a result, we can find i^* such that $\mathsf{LF}_{eo}(i^*)$ is the updated p_e by performing the range maximum query on LF_{eo} with the current suffix range. After updating $p_e = \mathsf{LF}_{eo}(i^*)$ by $L_{even}.\mathsf{rank}(\cdot)$ followed by $F_{odd}.\mathsf{select}(\cdot)$, p_s can also be updated using the updated p_e and the number c of the target suffixes.

To update this imaginary suffix range into a real suffix range for the update pattern xP, we observe that target suffixes are all the suffixes within the current imaginary suffix range. This is because the group corresponding to the newly prepended character x is a new group that has not been appeared in P, every suffix within the current imaginary suffix range should be considered regardless of their L_{odd}-values. And surprisingly, for $0 \leq i, j, k \leq n$ such that $i < p_s \leq k \leq p_e < j$, $\mathsf{LF}_{oe}(i) < \mathsf{LF}_{oe}(k) < \mathsf{LF}_{oe}(j)$. The lengths of the pointers to be prepended is longer than the length of the longest common prefix of (encoded) suffixes i (or j) and k, which does not change the relative order. Therefore, we do not have to update p_s and p_e anymore, and (p_s, p_e) itself is also the desired real suffix range.

6 Conclusions

In this paper, we present a space-efficient index for the structural pattern matching problem. The data structure requires $2n \lg \sigma + 2n + o(n)$ bits and it can count the number of occurrences of an m-length pattern in $\mathcal{O}(m \lg \sigma)$ time. Due to the hidden constant factor in $\mathcal{O}$ term in the previous succinct index [5], our structure can become much smaller if σ is small enough. Further, our data structure consists of building blocks that are widely used and practically implementable in many other succinct and compact data structures. Adding the sampled suffix array, we can also report each occurrence by repeatedly calling the LF-functions until reaching the sampled entry.

In the future work, the construction algorithm should be addressed. Once the suffix array of the pointer sequence for a given text s-string is given, our data structure can efficiently constructed; however, the construction of such suffix array is an open problem; besides pointer sequences, suffix sorting algorithms for a s-string have not been well-studied as well. Separating suffixes is not only for the compact representation, but it also gives many ways to generalize this problem further. For example, we may think of a problem in which a multiple number of characters are grouped together, instead of complement pairs.

Acknowledgement. The authors would like to thank anonymous reviewers for their valuable comments and suggestions.

References

1. Beal, R., Adjeroh, D.: Efficient pattern matching for RNA secondary structures. Theoret. Comput. Sci. **592**, 59–71 (2015). https://doi.org/10.1016/j.tcs.2015.05.016
2. Clark, D.: Compact pat trees. Ph.D. thesis, University of Waterloo (1996)
3. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algorithms **3**(2), 20–es (2007). https://doi.org/10.1145/1240233.1240243
4. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12200-2_16
5. Ganguly, A., Shah, R., Thankachan, S.V.: Structural pattern matching - succinctly. In: Proceedings of the 28th International Symposium on Algorithms and Computation (ISAAC), pp. 35:1–35:13 (2017). https://doi.org/10.4230/LIPIcs.ISAAC.2017.35
6. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Proceedings of the 13th International Symposium on Experimental Algorithms (SEA), pp. 326–337 (2014). https://doi.org/10.1007/978-3-319-07959-2_28
7. Golynski, A., Grossi, R., Gupta, A., Raman, R., Rao, S.S.: On the size of succinct indices. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 371–382. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75520-3_34
8. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 841–850 (2003). https://doi.org/10.5555/644108.644250
9. Jacobson, G.: Space-efficient static trees and graphs. In: Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS), pp. 549–554 (1989). https://doi.org/10.1109/SFCS.1989.63533
10. Kim, S.H., Cho, H.G.: Indexing isodirectional pointer sequences. In: Proceedings of the 31st International Symposium on Algorithms and Computation (ISAAC), pp. 35:1–35:15 (2020). https://doi.org/10.4230/LIPIcs.ISAAC.2020.35
11. Navarro, G.: Wavelet trees for all. J. Discrete Algorithms **25**, 2–20 (2014). https://doi.org/10.1016/j.jda.2013.07.004
12. Navarro, G., Sadakane, K.: Fully functional static and dynamic succinct trees. ACM Trans. Algorithms **10**(3), 1–39 (2014). https://doi.org/10.1145/2601073
13. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 134–149 (2010). https://doi.org/10.5555/1873601.1873614
14. Shibuya, T.: Generalization of a suffix tree for RNA structural pattern matching. In: SWAT 2000. LNCS, vol. 1851, pp. 393–406. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44985-X_34
15. Shibuya, T.: Generalization of a suffix tree for RNA structural pattern matching. Algorithmica **39**, 1–19 (2004). https://doi.org/10.1007/s00453-003-1067-9