Indexing Isodirectional Pointer Sequences

Sung-Hwan Kim
Department of Electrical and Computer Engineering, Pusan National University, South Korea
sunghwan@pusan.ac.kr

Hwan-Gue Cho
Department of Electrical and Computer Engineering, Pusan National University, South Korea
hgcho@pusan.ac.kr

Abstract

Many sequential and temporal data have dependency relationships among their elements, which can be represented as a sequence of pointers. In this paper, we introduce a new string matching problem with a particular type of strings, which we call *isodirectional pointer sequence*, in which each entry has a pointer to another entry. The proposed problem is not only a formalization of real-world dependency matching problems, but also a generalization of variants of the string matching problem such as parameterized pattern matching and Cartesian tree matching. We present a $2n \lg \sigma + 2n + o(n)$-bit index that preprocesses the text $T[1:n]$ so as to count the number of occurrences of pattern $P[1:m]$ in $\mathcal{O}(m \lg \sigma)$ where σ is the number of distinct lengths of pointers in T. Our index is also easily implementable in practice because it consists of wavelet trees and range maximum query index, which are widely used building blocks in many other compact data structures. By compressing the wavelet trees, the index can also be stored into $2nH_0^*(T) + 2n + o(n)$ bits where $H_0^*(T)$ is the 0-th order empirical entropy of the distribution of pointer lengths of T.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String Matching, Suffix Array, FM-index, Wavelet Tree, Range Minimum Query, Parameterized String Matching, Cartesian Tree Matching

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.35

1 Motivation

Many sequential and temporal data have dependency relationships among their elements. For example, a web access log may contain a list of accessed web pages in temporal order along with their referrer from which page a user comes. Retweet network can be used to analyze how a single message has been diffused widely, which can be represented as a sequence of individual tweets in their posted order with links to previous tweets where they are retweeted. Hierarchical comment tree in online discussion is a sequence of comments, each of which has its parent comment to which the comment writer made a reply. In natural language processing, a dependency parser generates a tree on top of a sentence. Each word has a pointer to its parent, which represents semantic dependency between words.

In this context, we introduce a simple string matching problem for a specific class of strings, which we call *isodirectional pointer sequence*. In these particular sequences, each entry is a pointer to another entry, and the orientations are the same across all the pointers; either all leftwards, or all rightwards. Many types of dependency sequences can be represented as isodirectional pointer sequence, especially when they have some hierarchical properties. Sequences representing temporal causality are likely to have pointers refer to previously occurred events such as web access logs, retweet network and comment trees. Dependency parse trees of head-final languages such as Korean and Japanese are likely to have only pointers with the left-to-right orientation. Imagine we need to search on this kind of sequences of pointers whether there exists some consecutive entries having the same

© Sung-Hwan Kim and Hwan-Gue Cho;
licensed under Creative Commons License CC-BY

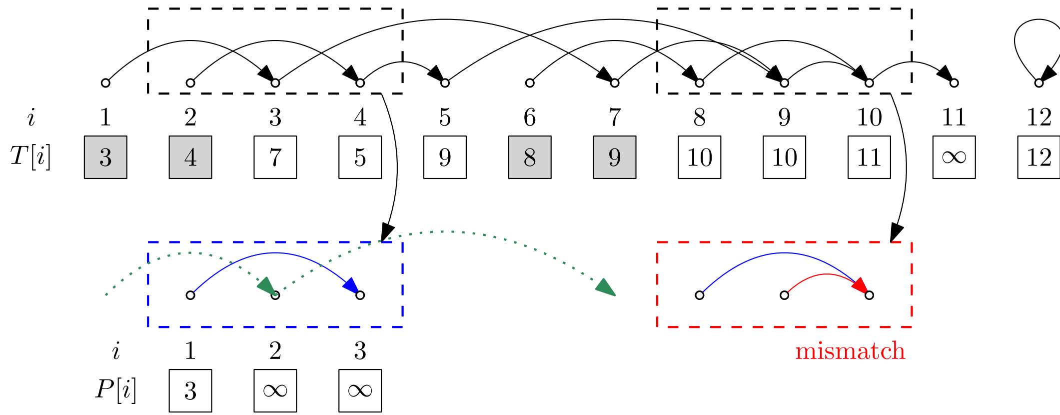


Figure 1 Illustration of the introduced matching problem. P matches to T at positions at 1, 2, 6, and 7. Pointers from/to outside do not count to determine the matching (indicated by green). But P does not match at position 8 because there is a pointer (indicated by red) from the second entry to the third entry while P does not have.

dependency relationship as a given pattern. More specifically, we want to build an index on a (relatively) long isodirectional pointer sequence so that later, we can perform pattern matching queries for short patterns efficiently.

To define the problem more formally, we first assume that every pointer is rightward without loss of generality, otherwise we can reverse the sequence. By the positional index of its target, we represent these pointers.

▶ **Definition 1** (Isodirectional Pointer Sequence). *A sequence $T[1:n]$ over $\Sigma = \{1, \cdots, n\} \cup \{\infty\}$ is an* isodirectional pointer sequence *if $T[i] \geq i$ for $1 \leq \forall i \leq n$.*

$T[i]$ means the i-th has a pointer to the $T[i]$-th entry. We allow self-referencing entries; that is those having $T[i] = i$. We also allow entries having no pointers, and denote by ∞.

When a pattern P of length m is given we say that it matches to the text sequence T at position i if, when we take m consecutive entries $T[i : i+m-1]$ from T, the dependency among these entries is exactly the same as that of the pattern. Because we consider the matching of consecutive entries this problem can be said a variant of string matching. Figure 1 illustrates the matching problem.

▶ **Definition 2** (String Matching). *For isodirectional pointer sequences $T[1:n]$ and $P[1:m]$, we say P matches to T at position i if, for $1 \leq \forall j \leq m$,*

$$\begin{cases} T[i+j-1] \geq i+m & \text{if } P[j] = \infty, \\ T[i+j-1] = i + P[j] - 1 & \text{otherwise.} \end{cases} \tag{1}$$

From a graphical perspective, we can represent an isodirectional pointer sequence T a labeled directed graphs $G_T = (V_T, E_T)$. Vertices are uniquely labeled with $\{1, \cdots, |V_T|\}$, and for any edge $(u, v) \in E_T$, $u \leq v$. Given a pattern $G_P = (V_P, E_P)$ we say P matches at vertex $u \in V_T$ if the subgraph induced by consecutive vertices $\{u, \cdots, u + |V_P| - 1\}$ is exactly the same as G_P when relabeling vertices with $\{1, \cdots, |V_P|\}$ in order.

This matching problem can also be considered as a generalization of other string matching variants allowing particular types of approximate matching such as parameterized string matching [2] and Cartesian tree matching [17]. In these problems, suffixes are transformed

by certain encoding schemes so that an encoded suffix is represented with dependency relationships among its entries. These problems can be viewed as special cases of the introduced problem, which means the result of this study can also be applied to them.

In this paper, we present an indexing method for isodirectional pointer sequences, with which we can count the number of occurrences of a length-m pattern in $\mathcal{O}(m \lg \sigma)$ where σ is the number of distinct lengths of the pointers. The proposed index occupies $2n \lg \sigma + 2n + o(n)$ bits where n is the text length, in addition to the sampled suffix array.

The rest of this paper is organized as follows. In Section 2, we establish some notation used throughout the paper, and we briefly review some data structures used for our work as building blocks. We first describe how to organize the proposed data structure in Section 3 and 4. Section 3 is dedicated to define the suffix array of isodirectional pointer sequences, and Section 4 describes the structural components of the proposed index with searching algorithms. Then we show the correctness of the proposed method in Section 5. In Section 6, we present indexes for parameterized string matching and Cartesian tree matching problem as special cases to demonstrate how our proposed method can be applied to other string matching problems. Finally we conclude the paper in Section 7 with giving some directions for the future work.

2 Preliminaries

2.1 Notation

By $T[1:n]$, we denote a string (sequence) of length n. $T[i]$ indicates the i-th entry and $T[i:j]$ is the substring $T[i]T[i+1]\cdots T[j]$ starting from position i to position j of T. For an isodirectional pointer sequence $T[1:n]$, we assume that $T[n]=n$ and it is the unique self-referencing entry as it acts like the termination symbol in the usual string matching problem. We say that the i-th entry refers to the j-th entry if $T[i]=j$, and in this case, we call the i-th entry a source entry and the j-th entry is the target entry of the i-th entry. If $T[i]=\infty$, we say that the i-th entry does not have a pointer. We define a set of integers $[n]=\{1,\cdots,n\}$. For any positive integer x, we define $\infty - x = \infty$.

2.2 Building Blocks

- **Wavelet tree** [11] of a string $T[1..n]$ over alphabet $\{1,\cdots,\sigma\}$ is a binary tree that a recursive structure in which nodes at the same level has a disjoint subsequence of the input text. At the root node, the bit vector $B[i]$ indicates whether $T[i]$ is less than x where x is the median among the symbols in that node. The subsequence induced by 0's is moved to its left child in order, and the remainder is moved to the right child. For each child node we perform this recursively until one symbol is left in a node.
  Wavelet tree can answer the following queries in $\mathcal{O}(\lg \sigma)$ time with $n \lg \sigma + o(n)$-bit space [16, 10].
  - $T[i]$: the value of the entry i of T.
  - $\texttt{select}(x,i)$: the position in which the i-th occurrence of x on T.
  - $\texttt{rank}(x,i)$: the number of occurrences of $x \in \Sigma$ in prefix $T[1..i]$.
  - $\texttt{rankrange}(x,[i,j])$: the number of occurrences of $T[k] \geq x$ for $i \leq k \leq j$.
- **Range Maximum (Minimum) Query** is also an important component of our index. For an integer array $L[1..n]$, a range maximum query (i,j) asks to find the index i^* such that $L[i^*] \geq L[i']$ for any $i \leq i' \leq j$. It can be answered in $\mathcal{O}(1)$ time with an index occupying $2n+o(n)$ bits [7].

3 Suffix Array for Isodirectional Pointer Sequences

In this section, we define the suffix array of isodirectional pointer sequences. More specifically, we define how to define, compare and sort suffixes of an isodirectional pointer sequence. As usual string matching problems [2, 17], we use an encoding scheme to define suffixes that will be used for suffix array construction and pattern matching. Because these encoded suffixes are not usual strings but sequences of integer sets, we define the relative order among integer set in order to enable the comparison of suffixes in lexicographical order. Then we define the suffix array and its related terms.

3.1 Encoding

In this subsection, we define the suffixes of an isodirectional pointer sequence $T[1:n]$, which will be used to construct the suffix array. Each entry of the suffixes is encoded with its corresponding source entries in terms of their positional distances. Note that each entry of an encoded suffix is an integer set because more than one entries may refer to the same entry.

▶ **Definition 3** (Encoded Suffix). *For an isodirectional pointer sequence $T[i:n]$, its encoded suffix $\mathbf{T}_i$ starting at position i is a sequence of integer sets defined as: For $1 \le j \le n-i+1$,*

$$\mathbf{T}_i[j] = \{T[k] - k : i \le k \le i+j-1 \text{ and } T[k] = i+j-1\} \tag{2}$$

The following lemma shows this particular encoding method corresponds to our matching problem. We can search for a pattern by searching the encoded suffixes starting having a particular prefix, which is an essential idea of indexing methods based on suffix trees and suffix arrays.

▶ **Lemma 4.** *For two isodirectional pointer sequence $T[1:n]$ and $P[1:m]$ ($m \le n$), let $\mathbf{T}_i$ and $\mathbf{P}_i$ be their encoded suffixes. P matches to T at position i if and only if $\mathbf{T}_i[1:m] = \mathbf{P}_1$.*

Proof. ($\Rightarrow$) For $1 \le j \le m$,

$$\mathbf{P}_1[j] = \{P[k] - k : 1 \le k \le j \text{ and } P[k] = j\}$$

Since we have $T[i+k-1] = i + P[k] - 1$ for $1 \le k \le j$ such that $P[k] = j \ne \infty$,

$$\begin{aligned} &= \{T[i+k-1] - (i+k-1) : 1 \le k \le j \text{ and } T[i+k-1] - (i-1) = j\} \\ &= \{T[i+k-1] - (i+k-1) : 1 \le k \le j \text{ and } T[i+k-1] = j+i-1\} \end{aligned}$$

Letting $k' = i+k-1$,

$$\begin{aligned} &= \{T[k'] - k' : 1 \le k' - (i-1) \le j \text{ and } T[k'] = j+i-1\} \\ &= \{T[k'] - k' : i \le k' \le j+i-1 \text{ and } T[k'] = j+i-1\} \\ &= \mathbf{T}_i[j] \end{aligned}$$

($\Leftarrow$) Assume that $\mathbf{T}_i[1:m] = \mathbf{P}_1$ but P does not match to T at position i. Then there must be $1 \le j \le m$ such that $T[i+j-1] < i+m$ and $T[i+j-1] \ne i+P[j]-1$. Let $k = T[i+j-1]-(i+j-1)$. Then $k \in \mathbf{T}_i[k+1]$. However, since $T[i+j-1] = k+(i+j-1) \ne i+P[j]-1$, we have $k \ne P[j]-j$. Hence, $k \notin \mathbf{P}_1[k+1]$. Therefore, $\mathbf{T}_i[k+1] \ne \mathbf{P}_1[k+1]$. Note that $k+1 = T[i+j-1]-(i+j-1)+1 < i+m-(i+j-1)+1 = m-j+2 \le m+1$. Therefore there exist $1 \le k' \le m$ such that $\mathbf{T}_i[k'] \ne \mathbf{P}_1[k']$. Contradiction. ◀

3.2 Comparison of Integer Sets

As the suffix array is defined using the lexicographical order of suffixes, we need to compare arbitrary two encoded suffixes. In lexicographical comparison of sequences, individual entries should be compared. Provided that an encoded suffix is a sequence of integer sets, we need to define the relative order among integer sets.

We define the relative order of two integer sets as recursively as follows:

▶ **Definition 5** (Order for Integer sets). *For integer sets $A \neq B$, we define:*

$$A < B \Leftrightarrow \begin{cases} B = \phi, \ \textit{or} \\ \min A < \min B, \ \textit{or} \\ A - \{\min A\} < B - \{\min B\}. \end{cases} \tag{3}$$

For more intuitive description, we can represent an integer set as a string by enumerating its elements in the increasing order and append a termination symbol at the end. The termination symbol is considered to be the greatest. For example, $A = \{1, 2\}$ can be represented as $12\diamond$, and $B = \{1, 2, 3\}$ as $123\diamond$, where $\diamond$ is the termination symbol. We have $A > B$ because $12\diamond > 123\diamond$; specifically, $A[3] = \diamond > 3 = B[3]$.

3.3 Suffix Array and LF-mapping

Now that we can compare arbitrary two encoded suffixes, we define the suffix array of an isodirectional pointer sequence by sorting the encoded suffixes. Because the suffix array SA is a one-to-one function from rank indices to position indices, we also define its inverse SA^{-1}.

▶ **Definition 6** (Suffix Array). *Suffix array $\mathsf{SA} : [n] \to [n]$ is a one-to-one function such that $\mathsf{SA}(i) = j$ if and only if $\mathbf{T}_j$ is the i-th smallest sequence among the encoded suffixes of T. Its inverse is denoted by SA^{-1}, and $\mathsf{SA}^{-1}(i) = j$ if and only if $\mathsf{SA}(j) = i$. For convenience, we also define $\mathsf{SA}^{-1}(0) = \mathsf{SA}^{-1}(n)$.*

The encoded suffixes sharing the same common prefix are consecutive in the suffix array because they are sorted. We define the suffix range to represent these consecutive suffixes corresponding to a particular pattern as a single interval.

▶ **Definition 7** (Suffix Range). *For two isodirectional pointer sequences T and P, let SA be the suffix array of T. A pair of integers (p_s, p_e) is the suffix range for P when, for $1 \leq \forall i \leq n$, P matches to T at position $\mathsf{SA}(i)$ if and only if $p_s \leq i \leq p_e$.*

Backward searching indexes such as FM-index [5], LF-mapping is a one-to-one function that connects positionally adjacent suffixes. If the rank of the suffix starting at position j on the text sequence T is i, its LF-mapping $\mathsf{LF}(i)$ indicates the rank of its positionally preceding suffix that starts at position $j - 1$ on T.

▶ **Definition 8** (LF-mapping). *LF-mapping $\mathsf{LF} : [n] \to [n]$ is a one-to-one function such that $\mathsf{LF}(i) = \mathsf{SA}^{-1}(\mathsf{SA}(i) - 1)$.*

4 Index Structure and Search Algorithm

Now we describe the structure of our proposed method. Given an isodirectional pointer sequence $T[1 : n]$, we build an index on T. The main objective of this data structure is to compute the suffix range (p_s, p_e) for a pattern $P[1 : m]$ efficiently. In this section, we focus on structural and procedural description of our method for the practical reasons, and the detailed theoretical justification will be described in the next section.

i	$\mathsf{SA}(i)$	$\mathsf{LF}(i)$	$F[i]$	$L[i]$	$\mathbf{T}_{\mathsf{SA}(i)}$											
1	12	2	0	∞	$\{0\}$											
2	11	3	∞	1	ϕ	$\{0\}$										
3	10	4	1	1	ϕ	$\{1\}$	$\{0\}$									
4	9	6	1	2	ϕ	$\{1\}$	$\{1\}$	$\{0\}$								
5	4	7	1	4	ϕ	$\{1\}$	ϕ	ϕ	$\{2\}$	$\{2,4\}$	$\{1,2\}$	$\{1\}$	$\{0\}$			
6	8	8	2	2	ϕ	ϕ	$\{1,2\}$	$\{1\}$	$\{0\}$							
7	3	9	4	2	ϕ	ϕ	$\{1\}$	ϕ	$\{4\}$	$\{2\}$	$\{2,4\}$	$\{1,2\}$	$\{1\}$	$\{0\}$		
8	7	10	2	2	ϕ	ϕ	$\{2\}$	$\{1,2\}$	$\{1\}$	$\{0\}$						
9	2	11	2	2	ϕ	ϕ	$\{2\}$	$\{1\}$	ϕ	$\{4\}$	$\{2\}$	$\{2,4\}$	$\{1,2\}$	$\{1\}$	$\{0\}$	
10	6	12	2	4	ϕ	ϕ	$\{2\}$	$\{2\}$	$\{1,2\}$	$\{1\}$	$\{0\}$					
11	1	1	2	0	ϕ	ϕ	$\{2\}$	$\{2\}$	$\{1\}$	ϕ	$\{4\}$	$\{2\}$	$\{2,4\}$	$\{1,2\}$	$\{1\}$	$\{0\}$
12	5	5	4	1	ϕ	ϕ	ϕ	$\{2\}$	$\{2,4\}$	$\{1,2\}$	$\{1\}$	$\{0\}$				

Figure 2 Example for $T = \langle 3, 4, 7, 5, 9, 8, 9, 10, 10, 11, \infty, 12 \rangle$ in Figure 1.

4.1 Structural Components

Our data structure consists of two wavelet trees and one range maximum query index. We define two arrays F and L of length n on which wavelet trees will be built. And we build the range maximum query index on an integer array M representing $\mathsf{LF}(\cdot)$ values. Clearly, the index occupies $2n \lg \sigma + 2n + o(n)$ bits where n is the text length, and σ is the number of distinct integers in arrays F and L. Specifically, as we will describe, σ is the number of distinct lengths of pointers of the input sequence T.

4.1.1 Wavelet Trees on F and L

We define two arrays F and L as: $F[i] = T[j] - j$ where $j = \mathsf{SA}(i)$, $L[i] = F[\mathsf{LF}(i)]$. $F[i]$ indicates which entry the first entry of the i-th smallest encoded suffix refers to. $L[i]$ represents the correspondence between positionally adjacent encoded suffixes.

We build wavelet trees on F and L. By WT_F and WT_L, we denote each of these wavelet trees respectively. The operations we use on these wavelet trees are listed as follows:

1. $\mathsf{WT}_L.\mathtt{access}(i) = L[i]$
2. $\mathsf{WT}_L.\mathtt{rank}(x, i) = |\{1 \leq k \leq i : L[k] = x\}|$
3. $\mathsf{WT}_L.\mathtt{rankrange}(x, [i, j]) = |\{i \leq k \leq j : L[k] \geq x\}|$
4. $\mathsf{WT}_F.\mathtt{select}(x, i) = j$ if and only if $F[j]$ is the i-th occurrence of x in F.

where i, j, x are integers.

4.1.2 Range Maximum Query on LF

Let M be an array of length n such that $M[i] = \mathsf{LF}(i)$. We build the range maximum query index on M, and this index supports the following operation:

- $\mathsf{RMQ}(i, j) = \arg\max_{i \leq k \leq j} \mathsf{LF}(k)$

This RMQ query finds the encoded suffix that will be mapped into largest encoded suffix via LF-mapping.

4.2 Computing LF(i)

Now we give an algorithm to compute LF(i). The basic idea is to group encoded suffixes into disjoint subsets. Let $\mathcal{L}_k$ be the encoded suffixes $\mathbf{T}_{\mathsf{SA}(i)}$ such that $L[i] = k$. Similarly, $\mathcal{F}_k$ be the encoded suffixes $\mathbf{T}_{\mathsf{SA}(i)}$ such that $F[i] = k$. The underlying observation is that suffixes in $\mathcal{L}_k$ are in one-to-one correspondence with suffixes $\mathcal{F}_k$. More specifically, the j-th smallest suffix in $\mathcal{L}_k$ corresponds to the j-th smallest suffix in $\mathcal{F}_k$.

The algorithm is shown in Algorithm 1. To compute LF(i), we determine j such that $\mathbf{T}_{\mathsf{SA}(i)}$ is the j-th smallest encoded suffix among encoded suffixes having the same L-value. This can be done by performing `rank` query on the wavelet tree built on L. Then we locate the j-th smallest encoded suffix among those having the same F-value via `select` query on F's wavelet tree.

Algorithm 1 Compute LF(i).

```
1: function LF(i)
2:     x ← WT_L.access(i)
3:     j ← WT_L.rank(x, i)
4:     k ← WT_F.select(x, j)
5:     return k
6: end function
```

4.3 Suffix Range Computation

Algorithm 2 shows the procedure to compute the suffix range of a given pattern of length m. Starting with the last position on the pattern, we iterate the loop m times At each iteration, we properly update the suffix range so that the encoded suffixes yield matching with the encoded suffix $\mathbf{P}_i$ of the pattern P. We apply different updating routines according to whether the currently processing entry has a pointer or not.

Algorithm 2 Compute the suffix range $[p_s, p_e]$ for a pattern P.

```
1: function SEARCH(P)                        ▷ P: an isodirectional pointer sequence
2:     (p_s, p_e) ← (1, n).
3:     for i = |P| to 1 do                    ▷ Processing the pattern backward
4:         x ← P[i] − i.
5:         if x = ∞ then                      ▷ The processing entry does not have a pointer
6:             c ← WT_L.rankrange(|P| − i + 1, [p_s, p_e])
7:             j ← RMQ(p_s, p_e)
8:             p_e ← LF(j)
9:             p_s ← p_e − c + 1
10:        else                               ▷ The processing entry has a valid pointer
11:            j_s ← WT_L.rank(x, p_s − 1) + 1
12:            j_e ← WT_L.rank(x, p_e)
13:            p_s ← WT_F.select(x, j_s)
14:            p_e ← WT_F.select(x, j_e)
15:        end if
16:    end for
17:    return (p_s, p_e)
18: end function
```

5 Proofs of Correctness

In this section, we prove the correctness of the proposed algorithms described in the earlier section.

5.1 Inducing Preceding Encoded Suffixes

First, we give an important observation from two consecutive encoded suffixes, which will be useful for the remainder of this section. When we consider two positionally adjacent suffixes $\mathbf{T}_{i-1}$ and $\mathbf{T}_i$, we can observe that at most one pointer is additionally established. As a result, $\mathbf{T}_{i-1}$ can simply be obtained from $\mathbf{T}_i$ by prepending ϕ at the beginning, and adding a single integer to a particular entry if needed.

▶ **Lemma 9.** *For an isodirectional pointer sequence $T[1:n]$, let $\{\mathbf{T}_i\}$ be its encoded suffixes. For any $1 < i \leq n$,*

$$\mathbf{T}_{i-1}[j] = \begin{cases} \mathbf{T}_i[j-1] \cup \{j-1\} & \text{if } j-1 = T[i-1]-(i-1) \\ \mathbf{T}_i[j-1] & \text{otherwise.} \end{cases} \tag{4}$$

Proof. For $1 < j \leq n-i+2$, we have

$$\begin{aligned} \mathbf{T}_{i-1}[j] &= \{T[k]-k : i-1 \leq k \leq (i-1)+j-1 \text{ and } T[k] = (i-1)+j-1\} \\ &= \{T[k]-k : i-1 \leq k \leq i+(j-1)-1 \text{ and } T[k] = i+j-2\} \\ &= \{T[i-1]-(i-1) : k = i-1 \text{ and } T[i-1] = (i-1)+(j-1)\} \\ &\cup \{T[k]-k : i \leq k \leq i+(j-1)-1 \text{ and } T[k] = i+(j-1)-1\} \\ &= \{j-1 : j-1 = T[i-1]-(i-1)\} \cup \mathbf{T}_i[j-1] \end{aligned}$$ ◀

Because there is at most one entry to be changed (except prepending ϕ at the beginning) we can say that there is unique entry where an integer is to be added when we compute its preceding encoded suffix from it. We call the position of this entry the *changing position*.

▶ **Definition 10** (Changing Position). *For an isodirectional pointer sequence $T[1:n]$, let $\{\mathbf{T}_i\}$ be its encoded suffixes. For $i > 1$, we say the changing position on $\mathbf{T}_i$ is $j-1$ if $T[i-1] = i+j-2$. If $T[i-1] = \infty$ we say $\mathbf{T}_i$ does not have any changing position.*

In particular to the text sequence where the suffix array and LF-mapping are available, we have the following corollary from Lemma 9.

▶ **Corollary 11.** *For any two consecutive encoded suffixes $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))}$ and $\mathbf{T}_{\mathsf{SA}(i)}$ of an isodirectional pointer sequence $T[1:n]$, for $1 \leq j \leq |\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))}|$,*

$$\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))}[j] = \begin{cases} S(j) \cup \{j-1\} & \text{if } j-1 = L[i] \\ S(j) & \text{otherwise.} \end{cases} \tag{5}$$

where $S(j) = \mathbf{T}_{\mathsf{SA}(i)}[j-1]$ if $j \geq 2$, $S(j) = \phi$ if $j = 1$.

Proof. We have $\mathsf{SA}(\mathsf{LF}(i)) = \mathsf{SA}(\mathsf{SA}^{-1}(\mathsf{SA}(i)-1))$. Thus, if $\mathsf{SA}(i) > 1$, $\mathsf{SA}(\mathsf{LF}(i)) = \mathsf{SA}(i)-1$, we apply Lemma 9 (noting that $L[i] = F[\mathsf{LF}(i)] = F[\mathsf{SA}^{-1}(\mathsf{SA}(i)-1)] = T[\mathsf{SA}(i)-1] - (\mathsf{SA}(i)-1)$). For i such that $\mathsf{SA}(i) = 1$, $\mathsf{SA}(\mathsf{LF}(i)) = n$ and $L[i] = 0$. Thus we have $|\mathbf{T}_n| = 1$, and $\mathbf{T}_n[1] = \{0\}$. Note that $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))}[1] = \phi$ unless $L[i] = 0$. ◀

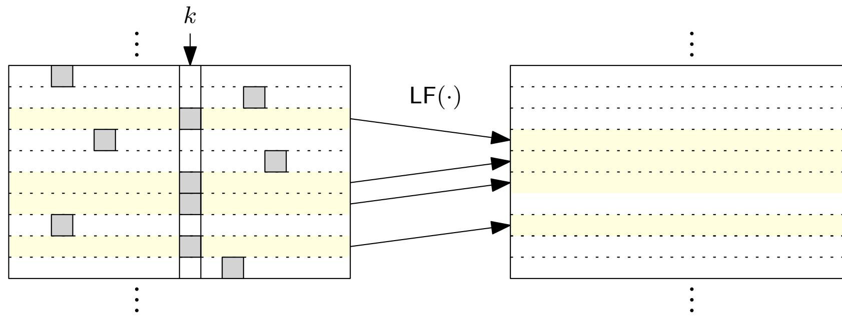


Figure 3 Illustration of Lemma 12. Encoded suffixes are listed in sorted order. Dark rectangles indicate $L[i]$'s of the encoded suffixes. For the suffixes having the same $L[i]$'s, the relative order is preserved after applying $\mathsf{LF}(\cdot)$.

5.2 Order Preserving Property and LF(·)

Now we prove the order preserving property $\mathsf{LF}(\cdot)$ within the set of the encoded suffixes $\mathbf{T}_i$ having the same $L[i]$-value as illustrated in Figure 3.

▶ **Lemma 12.** *For any integers i and j such that $1 \leq i < j \leq n$ and $L[i] = L[j]$, $\mathsf{LF}(i) < \mathsf{LF}(j)$.*

Proof. Since $L[i] = L[j]$, encoded suffixes $\mathbf{T}_{\mathsf{SA}(i)}$ and $\mathbf{T}_{\mathsf{SA}(j)}$ have the same changing position $L[i]$. Therefore the relative order of $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))}$ and $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}$ is determined by that of $\mathbf{T}_{\mathsf{SA}(i)}$ and $\mathbf{T}_{\mathsf{SA}(j)}$. Since $i < j$, $\mathbf{T}_{\mathsf{SA}(i)} < \mathbf{T}_{\mathsf{SA}(j)}$, thus we have $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))} < \mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}$, which leads to $\mathsf{LF}(i) < \mathsf{LF}(j)$. ◀

From this lemma, we can prove the correctness of Algorithm 1, which directly used this property.

▶ **Theorem 13.** *Algorithm 1 computes $\mathsf{LF}(i)$ correctly and runs in $\mathcal{O}(\lg \sigma)$ time where σ is the number of distinct integers in array F.*

Proof. By Lemma 12, if $\mathbf{T}_{\mathsf{SA}(i)}$ is the j-th smallest encoded suffix among $\{\mathbf{T}_{\mathsf{SA}(k)} : L[k] = L[i]\}$, the j-th smallest encoded suffix among $\{\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(k))} : F[k] = L[i]\}$ is $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))}$.

Noting that, L is a permutation of F, the alphabet sizes, say σ, of these two arrays are the same. Each operation on the wavelet trees takes at $\mathcal{O}(\lg \sigma)$ time, thus the algorithm runs in $\mathcal{O}(\lg \sigma)$ time. ◀

5.3 Correctness of Updated Suffix Range

In this section, we prove a lemma that is important to show that we correctly choose the encoded suffixes whose LF-mapping is to be included in the updated suffix range in Algorithm 2.

First. we introduce the notion of decoded sequence, which is interpreted as the inverse of the encoding scheme we used in this paper.

▶ **Definition 14** (Decoded Sequence). *For an isodirectional pointer sequence $P[1 : m]$, let $\{\mathbf{P}_i\}$ be its encoded suffixes. By $D(\mathbf{P}_i)$, we denote an isodirectional pointer sequence X such that its longest encoded suffix $\mathbf{X}_1 = D(\mathbf{P}_i)$.*

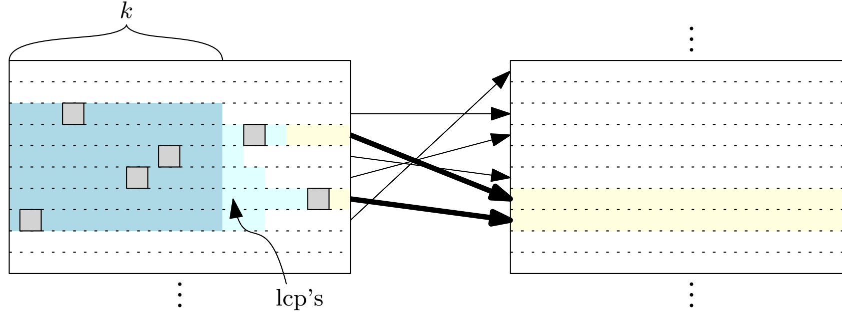


Figure 4 Illustration of Lemma 16. Light blue indicates the longest common prefixes between encoded suffixes and dark blue indicates a common prefix of encoded suffixes of length k, dark rectangles indicate changing positions. We can locate encoded suffixes having its changing position out of the current common prefix by finding the one whose LF-mapping is the largest.

The following lemma shows that, when we choose the encoded suffixes $\mathcal{S} = \{p_s \leq i \leq p_e : L[i]$ holds some certain condition$\}$ according to their $L[i]$-value, and apply LF-mapping $\{\mathsf{LF}(i) : i \in \mathcal{S}\}$ to them individually, we can obtain the correctly updated suffix range.

▶ **Lemma 15.** *For an isodirectional pointer sequence $P[1 : m]$ such that $P[i] \neq i$ for $1 \leq \forall i \leq m$, let $\{\mathbf{P}_i\}$ be its encoded suffixes. Let (p_i, q_i) be the suffix range for $D(\mathbf{P}_i)$. Then for $1 < i \leq m$ and $1 \leq j \leq n$, $p_{i-1} \leq \mathsf{LF}(j) \leq q_{i-1}$ if and only if $p_i \leq j \leq q_i$ and $L[j] > m - i + 1$ if $P[i-1] = \infty$, $L[j] = P[i-1] - (i-1)$, if $P[i-1] \neq \infty$.*

Proof. ($\Rightarrow$) Provided $p_{i-1} \leq \mathsf{LF}(j) \leq q_{i-1}$, we have $\mathbf{P}_{i-1} = \mathsf{T}_{\mathsf{SA}(\mathsf{LF}(j))}[1 : m - i + 2]$. To prove it by contradiction, we assume the negation of each of conditions. (i) First, let us assume $j < p_i$ or $q_i < j$, which means $\mathbf{P}_i[k] \neq \mathbf{T}_{\mathsf{SA}(j)}[k]$ for some $1 \leq k \leq m - i + 1$. Let x be any integer in the symmetric difference of $\mathbf{P}_i[k]$ and $\mathbf{T}_{\mathsf{SA}(j)}[k]$. Then $x < k$. Thus we cannot have $\mathbf{P}_{i-1}[k+1] = \mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}[k+1]$ only k can be added to these entry compared to $\mathbf{P}_i[k]$ and $\mathbf{T}_{\mathsf{SA}(j)}[k]$ by Lemma 9 and Corollary 11. (ii) Next, let us assume $L[j] \leq m - i + 1$ and $P[i-1] = \infty$. Then for some $1 < k \leq m - i + 1$, $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}[k+1] = \mathbf{T}_{\mathsf{SA}(j)}[k] \cup \{k\}$ while $\mathbf{P}_{i-1}[k+1] = \mathbf{P}_i[k]$ for all $1 < k \leq m - i + 1$. (iii) Finally, let us assume $L[j] \neq P[i-1] - (i-1)$ and $P[i-1] \neq \infty$. Then we have $\mathbf{P}_{i-1}[L[j]+1] = \mathbf{P}_i[L[j]]$. However, $\mathsf{T}_{\mathsf{SA}(j)}[L[j]] \cup \{L[j]\} = \mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}[L[j]+1] \neq \mathbf{P}_{i-1}[L[j]+1]$. Contradiction.

($\Leftarrow$) For $1 \leq k \leq m - i + 1$, $\mathbf{P}_i[k] = \mathbf{T}_{\mathsf{SA}(j)}[k]$. (i) If $P[i-1] = \infty$ and $L[j] > m - i + 1$, the changing position on $\mathbf{T}_{\mathsf{SA}(j)}$ is out of $m - 1 + 1$. Hence we have $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}[k+1] = \mathbf{T}_{\mathsf{SA}(j)}[k] = \mathbf{P}_i[k] = \mathbf{P}_{i-1}[k+1]$ for $1 < k \leq m - i + 1$, and $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}[1] = \mathbf{P}_{i-1}[1] = \phi$. (ii) If $P[i-1] \neq \infty$ and $L[j] = P[i-1] - (i-1)$, the changing position on $\mathbf{T}_{\mathsf{SA}(j)}$ (and $\mathbf{P}_i$) is $L[j]$ (and $P[i-1] - (i-1)$). Since $L[j] = P[i-1] - (i-1)$, the same entry is to be changed, which leads to $\mathbf{P}_{i-1} = \mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}[1 : m - i + 2]$. ◀

5.4 Lefter-Smaller Property within Common Prefix

While the method in the previous subsection correctly updates the suffix range according the given processing entry, we need to apply $\mathsf{LF}(\cdot)$ simultaneously across all the target suffixes for efficiency, rather than individually. Line 6–9 and Line 11–14 in Algorithm 2 perform this simultaneous update of the suffix range.

For the correctness of Line 6–9 in Algorithm 2 lies on the following lemma, which indicates that, when applying LF-mapping, an encoded suffix having a changing position within the currently searched prefix will always become smaller than the encoded suffixes that does not have a changing position within the same prefix (See Figure 4). Moreover, within the common prefix the lefter the changing position is the smaller its LF-mapping is.

▶ **Lemma 16.** *For any integers i, j, k such that $L[i] \leq k < L[j]$ and $k \leq \mathsf{lcp}(\mathbf{T}_{\mathsf{SA}(i)}, \mathbf{T}_{\mathsf{SA}(j)})$, $\mathsf{LF}(i) < \mathsf{LF}(j)$.*

Proof. Since position $L[i]$ is within the longest common prefix of $\mathbf{T}_{\mathsf{SA}(i)}$ and $\mathbf{T}_{\mathsf{SA}(j)}$, $\mathbf{T}_{\mathsf{SA}(i)}[L[i]] = \mathbf{T}_{\mathsf{SA}(j)}[L[i]]$. Note that, by Definition 3, for two integer sets X and $Y = X \cup \{y\}$ such that $y > \max X$, we defined their relative order as $X > Y$. By Corollary 11, $\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))}[L[i]+1] = \mathbf{T}_{\mathsf{SA}(i)}[L[i]] \cup \{L[i]\} < \mathbf{T}_{\mathsf{SA}(i)}[L[i]] = \mathbf{T}_{\mathsf{SA}(j)}[L[i]] = \mathbf{T}_{\mathsf{SA}(\mathsf{LF}(j))}[L[i]+1]$. Therefore $\mathsf{LF}(i) < \mathsf{LF}(j)$. ◀

As a result, when no additional pointer is to be established at the current iteration, we can locate the encoded suffix that will become the greatest encoded suffix among those in the updated suffix range by performing the range maximum query within the current suffix range. Combining it with the order preserving property, we can efficiently update the suffix range.

▶ **Theorem 17.** *Algorithm 2 computes the suffix range of $P[1:m]$ correctly and runs in $\mathcal{O}(m \lg \sigma)$ where σ is the number of distinct integers in array F.*

Proof. The invariant is (p_s, p_e) is the suffix range for $D(\mathbf{P}_i)$ at the end of each iteration. If $P[i] \neq \infty$ we can locate the encoded suffixes in the updated suffix range using Lemma 12 (Line 11–14). If $P[i] = \infty$, by Lemma 15, we can count the number $|\{\mathbf{T}_{\mathsf{SA}(\mathsf{LF}(i))} : p'_s \leq i \leq p'_e\}|$ of encoded suffixes in the updated suffix range (p'_s, p'_e) to be $c = |\{\mathbf{T}_{\mathsf{SA}(i)} : p_s \leq i \leq p_e \text{ and } L[i] \leq m - i + 1\}|$ in the current suffix range (p_s, p_e) (Line 6). By Lemma 16, we can find the right end p'_e of the updated suffix range (Line 7–8), then locate p'_s using p'_e and c (Line 9).

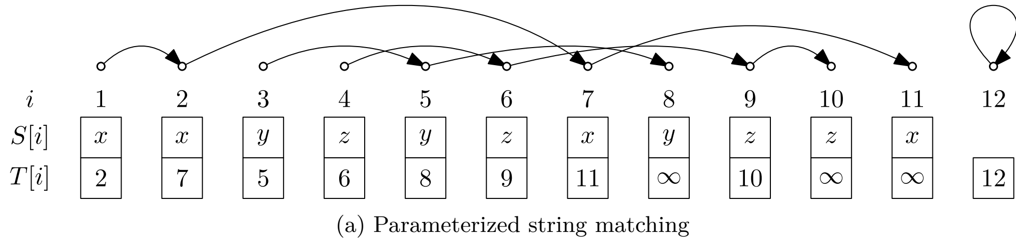
It takes $\mathcal{O}(\lg \sigma)$ time for each of wavelet tree queries, $\mathcal{O}(1)$ time for range maximum query, and the loop iterates m times, thus the algorithm runs in $\mathcal{O}(m \lg \sigma)$ time in total. ◀

6 Special Cases

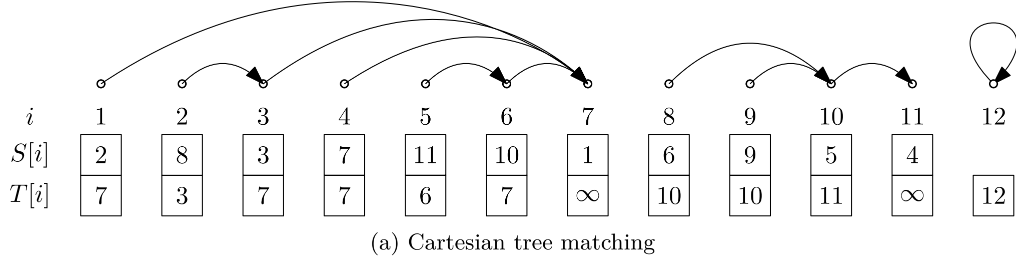
In this section, we present indexes for parameterized string matching [2] and Cartesian tree matching, which is a recently proposed string matching problem [17]. Although these indexes are not optimal solutions, they provide how the proposed method can be applied to other string matching problems.

6.1 Parameterized String Matching

In the original version of the parameterized string matching, the alphabet Σ over which strings are defined is the union of two subsets, static alphabet Σ_s and parameterized alphabet Σ_p. For sake of simplicity, we consider strings over only parameterized symbols; i.e. $\Sigma = \Sigma_p$. In this problem, two strings are defined as a match if there exists a one-to-one function $f : \Sigma \to \Sigma$ that converts one string into the other. For example $X = xyzxx$ matches to $Y = yzxyy$ because when we replace x with y, y with z, and z with x, we can transform X into Y completely.



(a) Parameterized string matching



(a) Cartesian tree matching

Figure 5 Isodirectional pointer sequence representation for parameterized string matching and Cartesian tree matching.

The traditional encoding scheme for this problem that has been used in the literature [2, 8] is to make a chain for each parameterized symbol. Let $S[1:n]$ be a parameterized string, then its corresponding isodirectional pointer sequence $T[1:n+1]$ is defined as follows. Figure 5-(a) shows an example.

$$T[i] = \begin{cases} \min_{i<j\leq n}\{j : S[j] = S[i]\} & \text{if such j exists,} \\ \infty & \text{otherwise.} \end{cases} \tag{6}$$

for $1 \leq i \leq n$, and $T[n+1] = n+1$.

Clearly, T is an isodirectional pointer sequence because $T[i] \geq i$ for all $1 \leq i \leq n+1$. Moreover, we can observe that taking a substring does not affect the pointers within the taken part in its isodirectional pointer sequence representation. In other words, when we append a character at the beginning (or at the end) of the underlying parameterized string, at most one pointer can be added from (or to) the newly added entry compared to the previous one, and no pointers become removed after appending. This exactly fits to matching of isodirectional pointer sequences. As a result, we can directly apply our method and build an $2n \lg \sigma + 2n + o(n)$-bit index for the parameterized matching.

Note that σ here is the number of distinct lengths of pointers, not the size of the parameterized alphabet $|\Sigma|$. In fact, we can reduce σ to $|\Sigma|$ by using the relative positions as in [8, 14]. Observe that a newly prepending pointer can refer to only one of the leftmost occurrences of parameterized symbols; in the graphical viewpoint, it can be linked to a node whose in-degree is 0. Thus the values of F and L are not necessarily arbitrary integers, but we can represent them with relative positions in terms of such entries, which makes the same index as that in [14] except cyclic shifts in the suffix enumeration.

6.2 Cartesian Tree Matching

We can also apply our method to Cartesian tree matching in a similar way. In Cartesian tree matching, two strings are said to match if their Cartesian tree is the same. Cartesian tree of a string is defined as: the entry with the minimum value becomes the root, and the substring on its left (right) side becomes its left (right, resp.) child, and construct recursively. For example, $X = 51432$ matches to $Y = 31542$ because they have the same Cartesian trees.

Let us rewrite the encoding scheme presented in its original work [17] in our language. In [17], each entry refers to some entry in its left side. In this paper, we assume the pointers to be rightwards, so we reverse the orientation. Let $S[1:n]$ be a string over the set of integers, and we assume that every $S[i]$ is distinct for simplicity. We define its corresponding pointer sequence $T[1:n+1]$ is defined as follows (see Figure 5-(b) for example):

$$T[i] = \begin{cases} \min_{i<j\leq n}\{j : S[j] < S[i]\} & \text{if such j exists,} \\ \infty & \text{otherwise.} \end{cases} \quad (7)$$

for $1 \leq i \leq n$, and $T[n+1] = n+1$.

Similar to the case of the parameterized string matching as we discuss above, we can also see that T is clearly an isodirectional pointer sequence, and a Cartesian tree match of integer sequences correspond to matching of their corresponding isodirectional pointer sequences. Therefore we can directly build an index using our method.

We can also observe the similarity between the parameterized string matching and Cartesian tree matching in converting the input string in its corresponding isodirectional pointer sequence. Each entry refers to its nearest (leftmost) entry that satisfies some conditions among its right side. Perhaps we can find other string matching problems to which our method can be applied regarding this characteristic.

7 Conclusions

In this paper, we introduced a string matching problem on a particular class of sequences, which we call *isodirectional pointer sequences*. This problem may capture some patterns describing consecutive temporal and sequential dependencies. Further, it can be considered as a generalization of existing variants of the string matching problems such as parameterized matching [2, 8] and Cartesian tree matching [17]. We presented a $2n \lg \sigma + 2n + o(n)$-bit index that can count the number of occurrences of a pattern in $\mathcal{O}(m \lg \sigma)$ time where n, m and σ are the text length, the pattern length, and the number of distinct lengths of pointers. Although we omitted the sampled suffix array part, we can directly add it as usual if we need to locate occurrences, which would need $2n + o(n)$ bits to locate each occurrence in $\mathcal{O}(\lg n \lg \sigma)$ time if we sample the suffix array for every $\lg n$ entries.

Our index consists of two wavelet trees and one range maximum query index. These data structures have been widely used in many compact and succinct data structures, thus we believe that the implementation of our index is quite simple. Because a wavelet tree over string $T[1:n]$ is compressible into $nH_0(T) + o(n)$ bit without sacrificing its time complexity [16], where $H_0(T)$ is T's 0-th order empirical entropy, our index can naturally be compressed into $2nH_0^*(T) + 2n + o(n)$ bits where $H_0^*(T)$ is the 0-th order empirical entropy of the distribution of pointer lengths of T. We can also apply other compression methods to them such as [6, 15, 12, 9].

While we did not consider any restriction about the pointer other than its orientation, there are possibly additional constraints. For example, in the parameterized string matching problem as we discussed in Section 6, we can observe that not only each entry can refer to at most one entry, but also it can be referred by at most one entry. With considering these constraints, we may be able to present tighter bounds for particular string matching problems.

We can also observe that our method cannot be directly applied to a certain kind of string matching problems such as order-preserving matching [13, 4]. In this problem, we can observe that appending a character at a string boundary completely affect the character

dependencies because it considers the relative order across the whole string despite that only the nearest character having a certain property is considered in parameterized matching or Cartesian tree matching. If we can find sufficient conditions for matching problems under which our method can be applied, it would be useful to give a deep understanding of the nature of the string matching variants as other attempts such as [3, 1] to generalization of string matching problems to pursue.

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 6:1–12, 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. Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. *SIAM Journal on Computing*, 33(1):26–42, 2003. doi:10.1137/S0097539701424465.
4. Maxime Crochemore, Costas S. Iliopulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving indexing. *Theoretical Computer Science*, 638:122–135, 2016. doi:10.1016/j.tcs.2015.06.050.
5. 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.
6. Paolo Ferragina and Giovanni Manzini. Compression boosting in optimal linear time using the burrows-wheeler transform. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 655–663, 2004. URL: https://dl.acm.org/doi/10.5555/982792.982892.
7. Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.
8. 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.
9. PawełGawrychowski, Seungbum Jo, Shay Mozes, and Oren Weimann. Compressed range minmum queries. *Theoretical Computer Science*, 812:39–48, 2020. doi:10.1016/j.tcs.2019.07.002.
10. Alexander Golynski, Roberto Grossi, Ankur Gupta, Rajeev Raman, and Satti Srinivasa Rao. On the size of succinct indices. In *Proceedings of the 15th European Symposium on Algorithms (ESA)*, pages 371–382, 2007. doi:10.1007/978-3-540-75520-3_34.
11. Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003. URL: https://dl.acm.org/doi/10.5555/644108.644250.
12. Juha Kärkkäinen and Simon J. Puglisi. Fixed block compression boosting in fm-indexes. In *Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 174–184, 2011. doi:10.1007/s00453-018-0475-9.
13. 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.
14. Sung-Hwan Kim and Hwan-Gue Cho. Simpler fm-index for parameterized string matching. *Information Processing Letters*, page 106026, 2020. (Online available). doi:10.1016/j.ipl.2020.106026.

15 Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 229–241, 2007. doi:10.1007/978-3-540-75530-2_21.
16 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.
17 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.