A Compact Index for Cartesian Tree Matching

Sung-Hwan Kim
Pusan National University, South Korea

Hwan-Gue Cho
Pusan National University, South Korea

Abstract

Cartesian tree matching is a recently introduced string matching problem in which two strings match if their corresponding Cartesian trees are the same. It is considered appropriate to find patterns regarding their shapes especially in numerical time series data. While many related problems have been addressed, developing a compact index has received relatively less attention. In this paper, we present a $3n + o(n)$-bit index that can count the number of occurrences of a Cartesian tree pattern in $\mathcal{O}(m)$ time where n and m are the text and pattern length. To the best of our knowledge, this work is the first $\mathcal{O}(n)$-bit compact data structure for indexing for this problem.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases String Matching, Suffix Array, FM-index, Compact Index, Cartesian Tree Matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.18

Acknowledgements We would like to thank anonymous reviewers for their valuable and constructive comments.

1 Motivation

String matching is an important and fundamental problem that is widely applied in various applications of computer science and engineering. Many practical problems involve a large volume of sequential data and hence string matching problems in finding particular patterns therein. Owing to many reasons such as noise in data, and a variety forms of isomorphism, many variants of the string matching problem have been introduced from various perspectives and different application domains, e.g., parameterized string matching [3] for program source codes, δ-approximate [7] and order preserving matching [22] for numerical time series, structural pattern matching for biological sequences [29], as well as general approximate matching problems such as jumbled pattern matching with Parikh vectors [1] and pattern matching with mismatches [5] and gaps [25].

Developing an indexing method for these approximate string matching problems is important particularly when either the text or the patterns are provided in advance and the other is given online. We are interested in indexing the provided text before pattern queries are given. By indexing the text appropriately, pattern search can be performed efficiently during the query time compared with scanning the text repeatedly for every single pattern. However, indexing strings for approximate string matching problems is usually challenging because of their complex nature. Whereas many problems involve significant space and time complexities [5, 25, 1], some problems may involve efficient data structures [3, 14, 8, 13].

Cartesian tree matching is a variant of the string matching problem that was recently introduced by Park *et al.* [28]. In this problem, two strings over a totally ordered set $\mathbb{U}$ match if their corresponding Cartesian trees are identical. It is considered appropriate to find patterns regarding their shapes and suitable for time series data. Many interesting properties of Cartesian trees have attracted a lot of interest regarding this matching problem, and researches have been conducted in many perspectives such as multiple pattern matching [18], dictionary matching [19], cover [21], longest common substring [10], and indeterminate matching [15] since the problem was introduced. Meanwhile, the development of a compact index for this problem has received relatively less attention. In the original paper [28], the authors show that the suffix tree for Cartesian tree matching can be built efficiently using the suffix tree construction algorithm for generalized matching problems [6]. Some researchers have addressed this problem as a special case of their indexing framework [2, 23]. However, all these methods are from the standpoint of generalized indexing methods, and the index size is $\mathcal{O}(n \lg n)$ in bits. Because a Cartesian tree of size n can be represented in $2n - o(n)$ bits, it is distant from the optimal space.

The main challenge of the compact data structure for Cartesian tree matching is how we achieve space compactness within $\mathcal{O}(n)$ bits while keeping the search time bounded within a time linear to the pattern length. As mentioned in [28], an the unary code can be used, which allows us to achieve $2n$ bits to represent the text string. However, one single character can be of length $\mathcal{O}(n)$ in its unary code in the worst case. This would cause $\mathcal{O}(n)$ time required to search for a pattern, thereby preventing the index from performing searches efficiently.

In this paper, we present a $3n + o(n)$-bit data structure[1] that supports a counting query in $\mathcal{O}(m)$ time where n and m are the text and pattern lengths, respectively. We introduce a novel concept of the *trimmed LF-mapping*, which allows us to develop a data structure supporting time-efficient queries for counting the number of occurrences within the space bound. To the best of our knowledge, this is the first $\mathcal{O}(n)$-bit index introduced for this problem.

The main theorem of this paper is as follows.

▶ **Theorem 1.** *There exists a $3n + o(n)$-bit data structure that can count the number of occurrences of a Cartesian tree pattern in $\mathcal{O}(m)$ time where n and m are the text and pattern length, respectively.*

The rest of the paper is organized as follows. In Section 2, we establish some notations used in the paper and we give a brief review on backgrounds including succinct bitvectors, which is a building block of the proposed data structure. We define the encoding scheme used to transform the suffixes of the text string in Section 3. In Section 4, we describe the underlying information used in the proposed data structure, and we give a conceptual description of how the searching procedure is performed. In Section 5, we propose a space-efficient representation of the structure described in the previous section. Then we present the searching algorithm on the proposed data structure in Section 6, and we conclude the paper in Section 7.

2 Preliminaries

Notation. By T, P, X and Y, we denote strings over a totally ordered set $\mathbb{U}$; especially, T and P are called text and pattern string. We assume that every element of these strings is distinct in each of them; if not, ties can be broken by position. We use 0-based index for strings and arrays; $T[0]$ indicates the first character. $|T|$ is the length of T. $T[i..j]$ is a substring $T[i] \circ T[i+1] \circ \cdots \circ T[j]$ of T where $\circ$ is the concatenation operator. We define $T[i..j]$ for $i > j$ as an empty string ϵ. For $T[0..i]$ and $T[i..|T|-1]$, we may use $T[..i]$ and $T[i..]$ for brevity. $\mathsf{lcp}(X, Y)$ is the length of the longest common prefix of X and Y. For an integer array A and a property $P(\cdot)$, we denote by $\langle A[i] \rangle_{i|P(A[i])}$, its subsequence obtained by concatenating $A[i]$'s such that $P(A[i])$ holds.

[1] Our data structure can be called an *encoding structure* in the sense that the text string T is not necessary during the pattern matching process.

$E(\cdot)$ is the encoding function described in Section 3.1 that we use to transform suffixes for indexing. We use $\oplus$ to represent a prepending operation as described in Section 3.1. SA is the suffix array and LF is the so-called LF-mapping that represent the correspondence between adjacent suffixes in terms of the lexicographical rank; these are defined in Section 3.2 L and F are integer arrays described in Section 4.1. $B_k^{(L)}$ and $B_k^{(F)}$ are bitvectors described in Section 5.1. On these bitvectors we use several navigating operations such as $\mathsf{down}_k(\cdot)$, $\mathsf{up}_k(\cdot)$, $\mathsf{map}_k(\cdot)$ as defined in Algorithms 1, with which we also define the trimmed LF-mapping $\mathsf{tLF}_k(\cdot)$ in Algorithm 2.

Cartesian Tree. Given a string X over $\mathbb{U}$, its Cartesian tree is a binary tree that is defined as follows. The element with the smallest value becomes the root. The left (and right) subtree is constructed recursively with the elements on the left (right) side. The Cartesian tree of a string of length n can be constructed in $\mathcal{O}(n)$ time using a stack-based algorithm. There are several representation of Cartesian trees regarding our work. These representation use the relation between the element corresponding to each iteration and the elements popped from the stack at the iteration during the construction of the Cartesian tree. Cartesian signature [9] uses the number of pop operations performed on the stack at corresponding iteration, and the parent-distance representation [28] uses the positional distance between the current element and the popped element.

Bitvectors. Bitvectors are basic building blocks of the proposed data structure in this paper. A data structure that supports the following queries in $\mathcal{O}(1)$ time for a length-n bit vector B can be stored in $n + o(n)$ bits [16]:

- Accessing $B[i]$.
- $B.\mathsf{rank}_x(i)$: the number of occurrences of x in $B[0..i-1]$.
- $B.\mathsf{select}_x(i) = j$ such that $B.\mathsf{rank}_x(j) = i-1$ and $B[j] = x$.

For convenience, we define $B.\mathsf{select}_x(0) = -1$. We also define rank and select queries on integer arrays and strings. Although we do not use them in the final searching algorithm, it is useful for describing how the proposed method works.

3 Encoding and Sorting Suffixes

For many variants of the string matching problem, such as parameterized string matching [3, 14], structural pattern matching [13], and order-preserving matching [8], and Cartesian tree matching [28], it is conventional to use an encoding scheme that transforms strings in a certain form such that two strings match iff their corresponding encoded strings are exactly the same. After encoding the suffixes of the text provided, we can build a data structure for standard string matching such as a suffix tree and a suffix array on the encoded suffixes to enable an efficient pattern search.

In this section, we present the encoding scheme that transforms the suffixes of the given text string for Cartesian tree matching problem, and define the suffix array on them, which will be used in the rest of the paper.

3.1 Modified Parent-Distance Representation

The encoding scheme we use in this paper is similar to that in the original paper [28], which is called *parent-distance representation.* In this representation, each element has at most one parent. Let X be a string. For each $0 \le i < |X|$, $X[i]$ does not have a parent if it is the smallest one among $X[0], \cdots, X[i]$. If there is an element $X[j]$ that is smaller than $X[i]$ for some $0 \le j < i$, $X[i]$ points to the rightmost one among such elements as its parent.

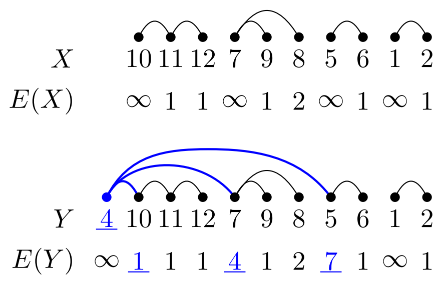


Figure 1 Illustration of changes in encoded strings when a character is prepended. For some k, the first k ∞'s are substituted with integers according to their positions after prepending a character. Arcs represent parents. In this case, we represent it as $E(Y) = 3 \oplus E(X)$.

The relation between an element and its parent is represented as the distance between their positions. For the elements that do not have a parent, 0 is used in the original paper. In this paper, we use ∞ instead of 0, which means the element having no parent will be represented as the greatest symbol in its encoded form. More formally, the encoding scheme is defined as follows:

▶ **Definition 2** (Encoding). *For a string X over $\mathbb{U}$, its encoded string $E(X)$ is defined as follows. For $0 \le i \le |X| - 1$,*

$$E(X)[i] = i - \max \pi_X(i) \tag{1}$$

where $\pi_X(i) = \{j \mid 0 \le j < i \text{ and } X[j] < X[i]\} \cup \{-\infty\}$. For convenience, we also define the encoded string of the empty string to be the empty string: i.e. $E(\epsilon) = \epsilon$.

As shown in [28], we can compute the (modified) parent-distance representation of a string in linear time.

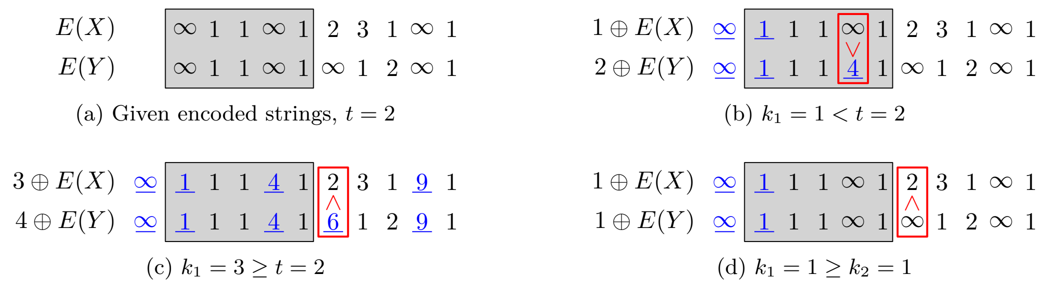
Our index will use a backward searching mechanism as FM-index families do. In order to devise a backward searching algorithm, we need to find out the relation between adjacent suffixes $T[i-1..]$ and $T[i..]$ in terms of their encoded strings. Note that $T[i-1..]$ is a string that can be obtained by prepending a character $T[i-1]$ at the beginning of $T[i..]$.

Let X and Y be strings over $\mathbb{U}$ such that $Y = x \circ X$ for some $x \in \mathbb{U}$; Y is a string that can be obtained by prepending a character x at the beginning of X. We want to observe their differences in terms of their encoded strings. This will be used to proceed from a suffix $T[i..]$ to its previous suffix $T[i-1..]$ during the searching process.

Figure 1 shows an example. When we prepend a character x to a string X, we can observe that, for some k, the first k ∞'s are substituted by some integers in its encoded form; and the integers with which ∞'s are substituted are determined by their positions. Then a single ∞ is prepended at the beginning, which completes $E(x \circ X) = E(Y)$. In other words, the operation of prepending a character can be characterized by an integer indicating the number of ∞'s to be substituted. In the rest of the paper, we represent the relation between two encoded strings such that $Y = x \circ X$ as $E(Y) = k \oplus E(X)$.

For an encoded string $E(X)$ and an integer $0 \le k \le E(X).\mathsf{rank}_\infty(|X|)$, $k \oplus E(X)$ is an encoded string defined as:

$$(k \oplus E(X))[i] = \begin{cases} \infty & \text{if } i = 0, \\ i & \text{if } E(X)[i-1] = \infty \text{ and } i - 1 \le E(X).\mathsf{select}_\infty(k), \\ E(X)[i-1] & \text{otherwise .} \end{cases} \tag{2}$$



$E(X)$ ∞ 1 1 ∞ 1 2 3 1 ∞ 1
$E(Y)$ ∞ 1 1 ∞ 1 ∞ 1 2 ∞ 1
(a) Given encoded strings, $t = 2$
$1 \oplus E(X)$ ∞ 1 1 1 ∞ 1 2 3 1 ∞ 1
$2 \oplus E(Y)$ ∞ 1 1 1 4 1 ∞ 1 2 ∞ 1
(b) $k_1 = 1 < t = 2$
$3 \oplus E(X)$ ∞ 1 1 1 4 1 2 3 1 9 1
$4 \oplus E(Y)$ ∞ 1 1 1 4 1 6 1 2 9 1
(c) $k_1 = 3 \geq t = 2$
$1 \oplus E(X)$ ∞ 1 1 1 ∞ 1 2 3 1 ∞ 1
$1 \oplus E(Y)$ ∞ 1 1 1 ∞ 1 ∞ 1 2 ∞ 1
(d) $k_1 = 1 \geq k_2 = 1$

Figure 2 Illustration of changes in lexicographical order when a character is prepended. $E(X)$ and $E(Y)$ are converted into $k_1 \oplus E(X)$ and $k_2 \oplus E(Y)$ due to a prepended character. t is the number of ∞'s within the longest common prefix (indicated with shaded boxes) of $E(X)$ and $E(Y)$. Underlined blue characters: changed elements after prepending a character; Red thick boxes: The position in which the order of two encoded strings is determined.

Given two encoded strings $E(X)$ and $E(Y)$, prepending characters at their beginning possibly changes their lexicographical order. Figure 2 decribes how the order can change by prepending different characters. From this observation, we establish an important lemma about the lexicographical order after prepending single characters at the beginning of these strings, which is frequently used throughout the paper.

▶ **Lemma 3.** *Let X and Y be strings over $\mathbb{U}$ such that $E(X) < E(Y)$, and let $t = E(X).\mathsf{rank}_\infty(l)$ where $l = \mathsf{lcp}(E(X), E(Y))$. For integers $0 \leq k_1 \leq E(X).\mathsf{rank}_\infty(|X|)$ and $0 \leq k_2 \leq E(Y).\mathsf{rank}_\infty(|Y|)$, $k_1 \oplus E(X) < k_2 \oplus E(Y)$ if and only if $k_1 \geq t$ or $k_1 \geq k_2$.*

Proof. **(⇒)** We prove by contrapositive. Let us assume that $k_1 < t$ and $k_1 < k_2$. Since $k_1 < t$, the (k_1+1)-th ∞ is within the longest common prefix of $E(X)$ and $E(Y)$. Thus $E(X).\mathsf{select}_\infty(k_1+1) = E(Y).\mathsf{select}_\infty(k_1+1)$. Let $i = E(X).\mathsf{select}_\infty(k_1+1)$. Then $(k_1 \oplus E(X))[i+1] = \infty > i+1 = (k_2 \oplus E(Y))[i+1]$. Since $(k \oplus E(X))[..i] = (k \oplus E(Y))[..i]$, this implies $k_1 \oplus E(X) > k_2 \oplus E(Y)$.

(⇐) We have two cases: (i) $k_1 \geq t$, and (ii) $k_1 \geq k_2$.

- **Case 1 ($k_1 \geq t$):** If $k_2 < t$, the (k_2+1)-th ∞ of $E(X)$ and $E(Y)$ is within their longest common prefix. Let $i = E(X).\mathsf{select}_\infty(k_2+1)$. The (k_2+1)-th ∞ of $E(X)$ is to be substituted with $i+1$, while the (k_2+1)-th ∞ of $E(Y)$ remains the same. Thus $(k_1 \oplus E(X))[i+1] = i+1 < \infty = (k_2 \oplus E(Y))[i+1]$, which implies $k_1 \oplus E(X) < k_2 \oplus E(Y)$.
  If $k_2 \geq t$, all ∞'s within the longest common prefix are to be substituted according to their positions, and the lexicographical order of $k_1 \oplus E(X)$ and $k_2 \oplus E(Y)$ is determined by $(k_1 \oplus E(X))[l+1]$ and $(k_2 \oplus E(Y))[l+1]$ where $l = \mathsf{lcp}(E(X), E(Y))$. If $E(Y)[l] \neq \infty$, $(k_1 \oplus E(X))[l+1] = E(X)[l] < E(Y)[l] = (k_2 \oplus E(Y))[l+1]$. If $E(Y)[l] = \infty$, $(k_1 \oplus E(X))[l+1] = E(X)[l] \leq l < l+1 = (k_2 \oplus E(Y))[l+1]$. In both cases, we have $k_1 \oplus E(X) < k_2 \oplus E(Y)$.
- **Case 2 ($k_1 \geq k_2$):** Since we have proved the case of $k_1 \geq t$, we can assume that $t > k_1 \geq k_2$. Thus all ∞'s being substituted with integers are within their longest common prefix. Let $i = E(X).\mathsf{select}_\infty(k_2+1)$ and $l = \mathsf{lcp}(E(X), E(Y))$. If $k_1 > k_2$, $(k_1 \oplus E(X))[..i] = (k_2 \oplus E(Y))[..i]$, and $(k_1 \oplus E(X))[i+1] = i+1 < \infty = E(Y)[i] = (k_2 \oplus E(Y))[i+1]$. If $k_1 = k_2$, $(k_1 \oplus E(X))[..l] = (k_2 \oplus E(Y))[..l]$, and $(k_1 \oplus E(X))[..l+1] = E(X)[l] < E(Y)[l] = (k_2 \oplus E(Y))[..l+1]$. Hence $k_1 \oplus E(X) < k_2 \oplus E(Y)$. ◀

3.2 Sorting Suffixes

In this subsection, we define the suffix array of a given text string $T[0..n-1]$. Let $\mathcal{S}(T)$ be the set of encoded suffixes:

$$\mathcal{S}(T) = \{E(T[i..]) \mid 0 \le i \le n\} \tag{3}$$

Note that we include $T[n..] = \epsilon$, so the size of $\mathcal{S}(T)$ is $n+1$. This acts like the unique termination symbol at the end of the text, which is a standard assumption in many indexing methods. The suffix array SA is an array of length $n+1$ that stores the encoded suffixes in the lexicographically sorted order. Each encoded suffix is represented by its starting position on the text; i.e. if $E(T[j..])$ is the $(i+1)$-th smallest string among the encoded suffixes, we define $\mathsf{SA}[i] = j$: For $0 \le i \le n$,

$$\mathsf{SA}[i] = j \text{ iff } i = \left|\{X \in \mathcal{S}(T) \mid X < E(T[j..])\}\right| \tag{4}$$

Since the suffix array is a permutation of $\langle 0, \cdots, n\rangle$, we can also define its inverse: $\mathsf{SA}^{-1}[i] = j$ iff $\mathsf{SA}[j] = i$.

3.3 Suffix Range

Remember that P matches $T[j..j+|P|-1]$ if and only if $E(P)$ is a prefix of $E(T[j..])$. Because we have sorted the encoded suffixes, if $E(P)$ is a prefix of encoded suffixes $E(T[j_1..]), E(T[j_2..]), \cdots, E(T[j_k..])$, then these encoded suffixes are consecutive in their sorted order on the suffix array. Therefore, we can use two integers $0 \le p_s \le p_e \le n$ such that $p_s \le \mathsf{SA}^{-1}[j_l] \le p_e$ for $1 \le l \le k$ to represent these encoded suffixes.

▶ **Definition 4** (Suffix range). *For an encoded pattern $E(P)$, an integer pair (p_s, p_e) is called the suffix range of $E(P)$ if $p_s \le i \le p_e \Leftrightarrow E(T[\mathsf{SA}[i]..])[0..|P|-1] = E(P)$ for all $0 \le i \le n$.*

4 Basic Idea on Updating Suffix Ranges

The proposed data structure performs a backward search that processes the pattern in the right-to-left direction. Assuming that the pattern $P[0..m-1]$ is not an empty string, it starts with the suffix range of $E(P[m-1..]) = \infty$. Then it repeatedly updates the suffix range by prepending $P[i]$ at the beginning of the currently searched pattern from $m-2$ to 0. At each iteration, we compute the suffix range of $E(P[i..m-1])$, hence we can obtain the desired suffix range of the entire pattern in the end.

Each iteration of this procedure can be described as follows. Let P be the currently searched pattern. When we prepend a character at the beginning of the currently searched pattern, this updated pattern can be written as $k \oplus E(P)$. Let (p_s, p_e) be the suffix range of an encoded pattern $E(P)$. What we want is to update the suffix range (p_s, p_e) into the suffix range (p'_s, p'_e) of $k \oplus E(P)$. It can be seen as two stages:

1. Identifying the target suffixes: from the set of indices $\mathcal{I} = \{i \mid p_s \le i \le p_e\}$, we identify the set of indices $\mathcal{I}' = \{i \mid p_s \le i \le p_e \text{ and } E(T[j-1..])[0..|P|] = k \oplus E(P) \text{ where } j = \mathsf{SA}[i]\} \subset \mathcal{I}$ whose corresponding suffixes are to be included in the updated suffix range after prepending their corresponding character $T[j-1]$.
2. Mapping $E(T[\mathsf{SA}[i]..])$ to its previous suffix $E(T[\mathsf{SA}[i]-1..])$ for each target suffix: we compute the set $\mathcal{I}'' = \{\mathsf{SA}^{-1}[\mathsf{SA}[i]-1] \mid i \in \mathcal{I}'\}$.

i	$\mathsf{SA}[i]$	$\mathsf{LF}[i]$	$F[i]$	$L[i]$	$E(T[\mathsf{SA}[i]..])$
0	15	1	-1	0	ϵ
1	14	8	0	0	∞
2	4	9	2	0	∞ 1 1 2 3 5 1 ∞ 1 2 3
3	0	0	1	-1	∞ 1 1 2 ∞ 1 1 2 3 5 1 ∞ 1 2 3
4	11	10	$\boxed{3}$	0	∞ 1 2 3
5	5	2	$\boxed{3}$	2	∞ 1 2 3 ∞ 1 ∞ 1 2 3
6	1	3	$\boxed{2}$	1	∞ 1 2 ∞ 1 1 2 3 5 1 ∞ 1 2 3
7	9	11	$\boxed{1}$	0	∞ 1 ∞ 1 2 3
8	13	12	0	0	∞ ∞
9	3	13	0	0	∞ ∞ 1 1 2 3 5 1 ∞ 1 2 3
10	10	7	0	$\boxed{1}$	∞ ∞ 1 2 3
11	8	14	0	0	∞ ∞ 1 ∞ 1 2 3
12	12	4	0	$\boxed{3}$	∞ ∞ ∞
13	2	6	0	$\boxed{2}$	∞ ∞ ∞ 1 1 2 3 5 1 ∞ 1 2 3
14	7	15	0	0	∞ ∞ ∞ 1 ∞ 1 2 3
15	6	5	0	$\boxed{3}$	∞ ∞ ∞ ∞ 1 ∞ 1 2 3

Figure 3 Underlying information of the proposed index for sequence $T = 4\ 6\ 9\ 8\ 2\ 10\ 15\ 14\ 12\ 3\ 13\ 1\ 11\ 7\ 5$. Examples of suffix ranges are indicated by boxes in column $E(T[\mathsf{SA}[i]..])$. Shaded, red (with thick borders), and blue (with thin borders) boxes indicate the suffix ranges of $P = 4\ 2$, $P' = 3\ 4\ 2$, and $P'' = 1\ 4\ 2$, respectively. Boxes in columns $F[i]$ and $L[i]$ indicate the entries used to obtain the suffix ranges of P' and P'' from that of P.

Computing the new indices in the stage 2 ($\mathsf{SA}^{-1}[\mathsf{SA}[i]-1]$) is the so-called *LF-mapping*, which is the core operation of many compact string matching indexes that use the backward searching mechanism. In this section, we define the LF-mapping along with its representation with two integer arrays. Then we also present how to identify the target suffixes using these arrays. Note that this representation using two integer arrays in this section is a conceptual representation; the space-efficient representation that is actually used in the proposed data structure is described in Section 5.

4.1 LF-mapping with Two Integer Arrays

In the context of backward searching methods such as FM-index, LF-mapping is a function indicating the correspondence between two adjacent suffixes $E(T[j..])$ and $E(T[j-1..])$ in terms of indices on the suffix array.

For an integer $0 \le i \le n$, let $T[j..]$ be a suffix such that $\mathsf{SA}[i] = j$. Let $j' = j + n \mod (n+1)$. The correspondence between $T[j..]$ and $T[j'..]$ in the sorted suffixes are stored in the array LF. More specifically, $\mathsf{LF}[i]$ indicates the lexicographical rank of $T[j'..]$.

$$\mathsf{LF}[i] = \mathsf{SA}^{-1}[\mathsf{SA}[i] + n \mod (n+1)] \tag{5}$$

We define two integer arrays L and F of length $n+1$, which contain the essential underlying information of the proposed data structure. Let us consider a particular suffix $T[\mathsf{SA}[i]..]$. When we prepend a character $T[\mathsf{SA}[i]-1]$ at the beginning of $T[\mathsf{SA}[i]..]$, the corresponding suffix is $E(T[\mathsf{SA}[i]-1..]) = E(T[\mathsf{SA}[i]-1] \circ T[\mathsf{SA}[i]..])$. We can uniquely determine an integer k such that $E(T[\mathsf{SA}[i]-1..]) = k \oplus E(T[\mathsf{SA}[i]..])$. This k is the value related to the Cartesian tree signature [9], which is also mentioned as an alternative representation for the matching problem in [28]. It can be computed during the construction of the Cartesian tree or its parent-distance representation. We define the array L to store such k's for each of the suffixes.

$$L[i] = \begin{cases} -1 & \text{if } \mathsf{SA}[i] = 0, \\ K(\mathsf{SA}[i] - 1) & \text{otherwise.} \end{cases}$$
$$\text{where } K(j) \text{ is the integer } k \text{ such that } E(T[j..]) = k \oplus E(T[j+1..]). \tag{6}$$

We can see that for the suffixes having the same $L[i]$ values, their LF-mapping is order-preserving.

▶ **Lemma 5.** *For $0 \leq i < j \leq n$ such that $L[i] = L[j]$, $\mathsf{LF}[i] < \mathsf{LF}[j]$.*

Proof. Note that $\mathsf{LF}[i] < \mathsf{LF}[j]$ iff $L[i] \oplus E(T[\mathsf{SA}[i]..]) < L[j] \oplus E(T[\mathsf{SA}[j]..])$. Applying Lemma 3 with $X = T[\mathsf{SA}[i]..]$, $Y = T[\mathsf{SA}[j]..]$, $k_1 = L[i]$, and $k_2 = L[j]$, we obtain $L[i] \oplus E(T[\mathsf{SA}[i]..]) < L[j] \oplus E(T[\mathsf{SA}[j]..])$ because $i < j \Leftrightarrow E(T[\mathsf{SA}[i]..]) < E(T[\mathsf{SA}[j]..])$ and $L[i] = L[j]$. ◀

Using this order-preserving property, we can represent the correspondence between two adjacent suffixes $E(T[\mathsf{SA}[\mathsf{LF}[i]]..])$ and $E(T[\mathsf{SA}[i]..])$ using their associated k-values described above. As the L array represents k-values for each suffix $E(T[\mathsf{SA}[i]..])$, we write these k-values for their associated suffixes $E(T[\mathsf{SA}[\mathsf{LF}[i]]..])$ to make another array F as follows:

$$F[\mathsf{LF}[i]] = L[i] \tag{7}$$

Using the arrays L and F, we can conceptually compute $\mathsf{LF}[i]$ as follows. Let $x = L[i]$. We can compute the number $c = L.\mathsf{rank}_{L[i]}(i+1)$ of occurrences $L[i]$ in $L[0..i]$. We find $0 \leq j \leq n$ such that the number of occurrences of x in $F[0..j]$ is c and $F[j] = x$: i.e. $j = F.\mathsf{select}_{L[i]}(c)$ is the position of the c-th occurrence of x on F. Then we have $j = \mathsf{LF}[i]$.

4.2 Identifying Target Suffixes

To devise a backward searching algorithm, we need to compute the suffix range (p'_s, p'_e) of $k \oplus E(P)$ from the suffix range (p_s, p_e) of $E(P)$. As we have mentioned at the beginning of this section, this update procedure consists of two stages, which is identifying the target suffixes according to k followed by applying LF-mapping for each of the identified target suffixes. In this subsection, we present how to identify the target suffixes using L array during the suffix range update.

We have two cases: (i) there are ∞'s remaining in $(k \oplus E(P))[1..]$, and (ii) all ∞'s in $E(P)$ are to be substituted into integers so there is no ∞ in $(k \oplus E(P))[1..]$. In the first case, we know that the same number of ∞'s are to be substituted in the target suffix after applying the LF-mapping. On the other hand, in the second case, the number of ∞'s that are to be substituted in the target suffix is not fixed, but k is the lower bound of the number of substituted ∞'s.

▶ **Lemma 6.** *For an encoded string $E(P)$ and an integer k such that $0 \leq k \leq E(P).\mathsf{rank}_\infty(|P|)$, let (p_s, p_e) and (p'_s, p'_e) be the suffix ranges of $E(P)$ and $k \oplus E(P)$, respectively. For $p_s \leq i \leq p_e$, $p'_s \leq \mathsf{LF}[i] \leq p'_e$ if and only if*

$$\begin{cases} L[i] = k & \text{if } k < E(P).\mathsf{rank}_\infty(|P|), \\ L[i] \geq k & \text{if } k = E(P).\mathsf{rank}_\infty(|P|). \end{cases} \tag{8}$$

Proof. ($\Leftarrow$) We can rewrite the right-hand of the if-and-only-if statement as: **(i)** $L[i] = k$ or **(ii)**$L[i] > k$ and $k = E(P).\mathsf{rank}_\infty(|P|)$. (Case 1: $L[i] = k$) Let Q be a string such that $E(Q) = E(P) \circ (\infty)^n$. Then the fact that $p_s \leq i \leq p_e$ is equivalent to $E(P) \leq E(T[\mathsf{SA}[i]..]) < E(Q)$. Also, $k \oplus E(P) \leq k \oplus E(T[\mathsf{SA}[i]..]) < k \oplus E(Q)$. Since $L[i] = k$, $k \oplus E(P) \leq L[i] \oplus E(T[\mathsf{SA}[i]..]) = E(T[\mathsf{SA}[\mathsf{LF}[i]]..]) < k \oplus E(Q)$ by Lemma 3. (Case 2: $L[i] > k$ and $k = E(P).\mathsf{rank}_\infty(|P|)$) Note that $E(T[\mathsf{SA}[i]..]).\mathsf{select}_\infty(k') \geq |P|$ for all $k' > k$. Hence, $(L[i] \oplus E(T[\mathsf{SA}[i]..]))[0..|P|] = k \oplus (E(T[\mathsf{SA}[i]..])[0..|P|-1]) = k \oplus E(P)$.

($\Rightarrow$) We prove by contrapositive. The negation of the right-hand part of the if-and-only-if statement can be rewritten as follows: **(i)** $L[i] < k$ or **(ii)**$L[i] > k$ and $k < E(P).\mathsf{rank}_\infty(|P|)$. Note that because i is within the suffix range of $E(P)$, $E(T[\mathsf{SA}[i]..]).\mathsf{rank}_\infty(|T[\mathsf{SA}[i]..]|) \geq E(P).\mathsf{rank}_\infty(|P|) \geq k$; moreover, $E(T[\mathsf{SA}[i]..]).\mathsf{select}_\infty(j) = E(P).\mathsf{select}_\infty(j)$ for $1 \leq j \leq E(P).\mathsf{rank}_\infty(|P|)$. (Case 1: $L[i] < k$): Let $j = E(T[\mathsf{SA}[i]..]).\mathsf{select}_\infty(L[i]+1)+1 = E(P).\mathsf{select}_\infty(L[i]+1)+1$. Then we have $(L[i] \oplus E(T[\mathsf{SA}[i]..])[j] = \infty \neq j = (k \oplus E(P))[j]$. (Case 2: $L[i] > k$ and $k < E(P).\mathsf{rank}_\infty(|P|)$) Let $j = E(T[\mathsf{SA}[i]..]).\mathsf{select}_\infty(k+1)+1$. $(L[i] \oplus E(T[\mathsf{SA}[i]..])[j] = j \neq \infty = E(P)[j-1] = (k \oplus E(P))[j]$. ◀

5 $3n + o(n)$-bit Representation

In this section, we present how to represent two arrays L and F in a space-efficient way. After we present a $6n + o(n)$-bit representation, we show how to reduce the space occupancy into $3n + o(n)$ bits by representing $3n + 2$ bits among them within $\mathcal{O}(\lg n) = o(n)$ bits.

5.1 Representation of L and F with Unary Coding

In this subsection, we present how to efficiently represent the two arrays L and F. First, we define subsequences of L and F for $k \geq -1$:

$$L_k = \langle L[i] \rangle_{i | L[i] \geq k} \tag{9}$$

Similarly,

$$F_k = \langle F[i] \rangle_{i | F[i] \geq k} \tag{10}$$

We also define bitvectors corresponding to L_k and F_k:

$$B_k^{(L)}[i] = 0 \text{ iff } L_k[i] = k, \text{ otherwise } 1. \tag{11}$$

and

$$B_k^{(F)}[i] = 0 \text{ iff } F_k[i] = k, \text{ otherwise } 1. \tag{12}$$

Note that the number of bits at level $k+1$ is the number of 1-bits at level k. The i-th 1-bit at level k is associated with i-th bit at level $k+1$. Using this correspondence between bits on consecutive levels, we can build tree-like structures on L and F. This is similar to the (pointer-less) wavelet tree [17, 27] with a non-standard shape. Wavelet tree with different shapes have been used to compress the space occupancy and speed up the query time [12], in which Huffman prefix tree is typically used where each element can be seen to be represented as its Huffman code. For our data structure, it can be seen as representing each element x of L and F as the unary code of $x+1$: i.e. a bit string $1^{x+1}0$.

Note the sum of $L[i]$ over $0 \leq i \leq n$ such that $L[i] \geq 0$ is less than n and there is the unique i such that $L[i] = -1$. Thus the sum of $L[i]+1$ over all $0 \leq i \leq n$ is less than $2n$. The total number of 0's is $n+1$. The number of bits for representing L and F is at most

$(2n-1)+(n+1)=3n$ each; thus the total number of bits over the entire data structure is at most $6n$. We build the rank and select dictionaries on these bitvectors, which would occupy $6n+o(n)$ bits in total.

▶ Remark. One might also find that it is similar to Direct Addressable Code (DAC,[4]) with block size $b=1$. Each chunk $C_{i,j}$ of DAC consists of two parts: a single flag bit ($B_i[j]$), and a b-bit block of codes ($A_{i,j}$). In this case, every block $A_{i,j}$ has a single bit, B_i indicates whether this block is the highest one. Each element of L (or F) can be represented across as many levels as its value. The bitvector B_i of DAC is actually the same as the bits that comprise our data structure. Perhaps we may use this observation to extend our data structure into other string matching problems where L and F values can be represented with a variable number of integers.

We define the operations that are used to navigate across bitvectors in Algorithm 1. By $\mathsf{down}_k(i)$ we can move from a position on $B^{(L)}_{-1}$ to its corresponding position on $B^{(L)}_k$. Similarly, $\mathsf{up}_k(i)$ computes the corresponding position on $B^{(F)}_{-1}$ to a position on $B^{(F)}_k$. We can use $\mathsf{map}_k(i)$ to jump from $B^{(L)}_k$ to $B^{(F)}_k$; 0's (resp. 1's) on $B^{(L)}_k$ correspond to 0's (resp. 1's) on $B^{(F)}_k$ in order.

Algorithm 1 Navigating operations on $B^{(L)}_k$'s and $B^{(F)}_k$'s.

```
1  function $\mathsf{down}_k(i)$:
2      for $l = -1$ To $k-1$ do
3          $i \leftarrow B^{(L)}_l.\mathsf{rank}_1(i)$
4      end
5      return $i$

6  function $\mathsf{up}_k(i)$:
7      for $l = k$ To $0$ do
8          $i \leftarrow B^{(F)}_{l-1}.\mathsf{select}_1(i+1)$
9      end
10     return $i$

11 function $\mathsf{map}_k(i)$:
12     $x \leftarrow B^{(L)}_k[i]$
13     return $B^{(F)}_k.\mathsf{select}_x(B^{(L)}_k.\mathsf{rank}_x(i)+1)$
```

5.2 Trimmed LF-mapping

The unary representation of L and F can reduce the required space into $\mathcal{O}(n)$ in bits. However, accessing a single element $L[i]$ takes $\Theta(L[i])$ time and $L[i] \leq n$. As a result, computing the LF-mapping may take $\Theta(n)$ time in the worst case, which would prevent us from achieving $\mathcal{O}(m)$ query time. In this section, we introduce the concept of *trimmed LF-mapping*, which enables us to update the suffix range efficiently. Although an individual computation of the trimmed LF-mapping of a particular suffix may produce an incorrect mapping, it effectively works for the simultaneous mapping of target suffixes during the suffix range updates. The trimmed-mapping function $\mathsf{tLF}(i)$ is defined as follows.

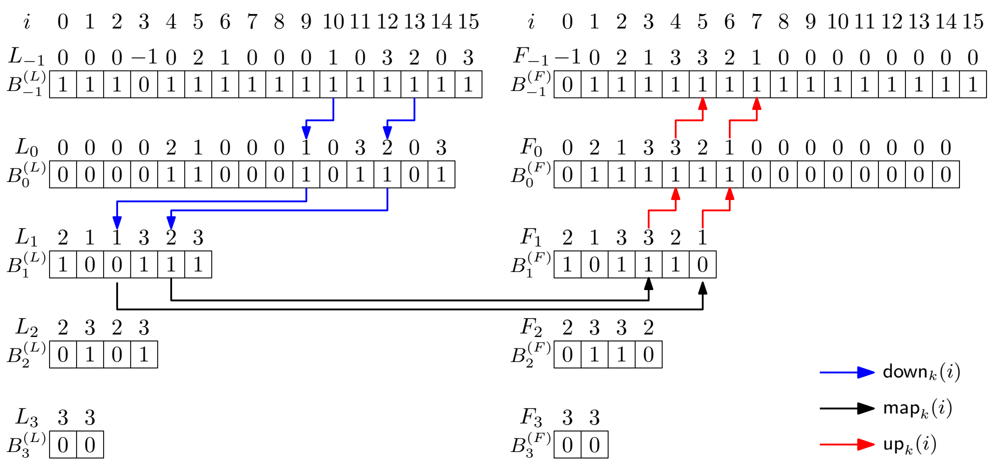

i 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
L_{-1} 0 0 0 −1 0 2 1 0 0 0 1 0 3 2 0 3
$B_{-1}^{(L)}$ 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
L_0 0 0 0 0 2 1 0 0 0 1 0 3 2 0 3
$B_0^{(L)}$ 0 0 0 0 1 1 0 0 0 1 0 1 1 0 1
L_1 2 1 1 3 2 3
$B_1^{(L)}$ 1 0 0 1 1 1
L_2 2 3 2 3
$B_2^{(L)}$ 0 1 0 1
L_3 3 3
$B_3^{(L)}$ 0 0
i 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
F_{-1} −1 0 2 1 3 3 2 1 0 0 0 0 0 0 0 0
$B_{-1}^{(F)}$ 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
F_0 0 2 1 3 3 2 1 0 0 0 0 0 0 0 0
$B_0^{(F)}$ 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0
F_1 2 1 3 3 2 1
$B_1^{(F)}$ 1 0 1 1 1 0
F_2 2 3 3 2
$B_2^{(F)}$ 0 1 1 0
F_3 3 3
$B_3^{(F)}$ 0 0
$\mathsf{down}_k(i)$
$\mathsf{map}_k(i)$
$\mathsf{up}_k(i)$

Figure 4 Computing $\mathsf{tLF}_k(i)$, which consists of $\mathsf{down}_k(\cdot)$, $\mathsf{map}_k(\cdot)$, and $\mathsf{up}_k(\cdot)$. If we arrive at a 0-bit after executing $\mathsf{down}_k(i)$, $\mathsf{tLF}_k(i) = \mathsf{LF}[i]$. If not, $\mathsf{tLF}_k(\cdot)$ may be jumbled; but it does not actually matter for updating suffix ranges. In this example, $\mathsf{tLF}_1(10)$ and $\mathsf{tLF}_1(13)$ are depicted.

Algorithm 2 Computing Trimmed LF-mapping.

```
1 function $\mathsf{tLF}_k(i)$:
2     $i \leftarrow \mathsf{down}_k(i)$
3     $i \leftarrow \mathsf{map}_k(i)$
4     $i \leftarrow \mathsf{up}_k(i)$
5     return $i$
```

An example of $\mathsf{tLF}_k(\cdot)$ is illustrated in Figure 4, which computes $\mathsf{tLF}_1(10)$ and $\mathsf{tLF}_1(13)$. Let us explain the computation of $\mathsf{tLF}_1(10)$, which is described as the outer path in Figure 4. After performing $\mathsf{down}_1(10)$, we arrive at position 2 at level 1. By $\mathsf{map}_1(2)$ we move to its corresponding position on the F-side tree, which is position 5 at level $B_1^{(F)}$. Then we obtain $\mathsf{tLF}_1(10) = 7$ by performing $\mathsf{up}_1(5)$. Similarly, we can compute $\mathsf{tLF}_1(13) = 5$ by performing $\mathsf{down}_k(\cdot)$, $\mathsf{map}_k(\cdot)$, $\mathsf{up}_k(\cdot)$ in order, each step of which we arrive at positions 4, 3 and 5 respectively.

Notice the different behavior of $\mathsf{map}_k(i)$ depending on the value $B_k^{(L)}(i)$, in the perspective of the correspondence that $\mathsf{map}_k(\cdot)$ makes between B_k and L_k. Suppose we arrive at position i at level k. If $B_k^{(L)}[i] = 0$, we have $L_k[i] = F_k[j]$ where $j = \mathsf{map}_k(i)$. We also observe that the mapping between L_k and F_k using 0-bits is order-preserving. From this observation, we can obtain the position of the l-th occurrence of x on F by computing $\mathsf{tLF}_k(i)$ where i is the position of the l-th occurrence of x on L if we can reach a 0-bit at the deepest level during the computation. $\mathsf{tLF}_k(\cdot)$ can be used to compute LF directly. If we compute $\mathsf{tLF}_{L[i]}(i)$, we reach a 0 after calling $\mathsf{down}_{L[i]}(i)$, which means we reach the end of the unary code of a particular element. Conceptually, $\mathsf{tLF}_{L[i]}(i)$ is equivalent to perform a rank query on L for $x = L[i]$, followed by performing select query for the corresponding occurrence of x on F. Thus we can successfully compute $\mathsf{LF}[i]$.

▶ **Lemma 7.** *For* $0 \leq i \leq n$, *we can compute* $\mathsf{LF}[i]$ *in* $\Theta(L[i])$ *time; more specifically,* $\mathsf{tLF}_{L[i]}(i) = \mathsf{LF}[i]$

Proof. Let $i' = \mathsf{down}_{L[i]}(i)$ and $i'' = \mathsf{map}_{L[i]}(i')$. Clearly, $L_{L[i]}[i'] = F_{L[i]}[i''] = L[i]$ and $L_{L[i]}.\mathsf{rank}_{L[i]}(i') = F_{L[i]}.\mathsf{rank}_{L[i]}(i'')$ because they are order-preserving. Let $i^* = \mathsf{up}_{L[i]}(i'')$. Then we have $L[i] = F[i^*]$ and $L.\mathsf{rank}_{L[i]}(i) = F.\mathsf{rank}_{L[i]}(i^*)$. Therefore $\mathsf{LF}[i] = i^* = \mathsf{tLF}_{L[i]}(i)$ by Lemma 5. $\mathsf{down}_{L[i]}(i)$ and $\mathsf{up}_{L[i]}(i'')$ take $\Theta(L[i])$ time each, and $\mathsf{map}_{L[i]}(i')$ takes $\mathcal{O}(1)$ time. Therefore, computing $\mathsf{tLF}_{L[i]}(i)$ takes $\Theta(L[i])$ time. ◀

On the other hand, when $B_k^{(L)}[i] = 1$, $L_k[i]$ is not necessarily the same as $F_k[j]$ where $j = \mathsf{map}_k(i)$. However, we can see that the l-th occurrence of an element on L that is not less than k is associated with the l-th occurrence of an element on F that is not less than k by this mapping at level k. Although $\mathsf{map}_k(i)$ may not give the exact value of $\mathsf{LF}[i]$, we can effectively use this to update a suffix range, which will be described in Section 6.

5.3 Representing $B_k^{(L)}$ and $B_k^{(F)}$ Compactly

The total number of bits to represent all of $B_k^{(L)}$'s and $B_k^{(F)}$'s in their raw form is at most $6n$ bits. We can reduce the required number of bits by representing certain levels of bitvectors compactly. Note that bit vectors at level -1 consist of $n+1$ bits, and there is only one 0-bit in each of $B_{-1}^{(L)}$ and $B_{-1}^{(F)}$. Thus we can represent them using $\mathcal{O}(\lg n)$ bits by representing the position of the unique 0-bit using a single integer. We can also make an observation that $B_0^{(F)}$ is a form of 01^p0^{n-p-1} where p is the number of occurrences of 0's on F. Therefore $B_0^{(F)}$ can be represented by a pair of integers that represent the interval in which 1-bits are located. Using this representation, the bitvectors can be represented in up to $3n + o(n)$ bits in total.

▶ **Lemma 8.** *$B_{-1}^{(L)}$, $B_{-1}^{(F)}$ and $B_0^{(F)}$ can be stored in $\mathcal{O}(\lg n)$ bits while supporting* rank *and* select *queries in $\mathcal{O}(1)$ time.*

Proof. It is trivial for $B_{-1}^{(L)}$ and $B_{-1}^{(F)}$ because the number of 1-bits is 1 so a single integer occupying $\mathcal{O}(\lg n)$ bits can represent these bit vectors. For $B_0^{(F)}$, we claim that $B_0^{(F)}$ has a form of 01^p0^{n-p-1} for some p. Note that $E(T[\mathsf{SA}[0]..]) = E(T[|T|..]) = \epsilon$ is the smallest encoded suffix, and $E(T[|T|-1..]) = \infty$ is the smallest among non-empty encoded suffixes because, for $1 \le i \le n$, $E(T[\mathsf{SA}[i]..])$ have a common prefix ∞. It is clear that $L[0] = 0 = F[\mathsf{LF}[0]] = F[1]$. Since $F[0] = -1$, $F[1]$ is the first occurrence of 0, thus we have $B_0^{(F)}[0] = 0$. Because the number of ∞'s in the longest common prefix of any two non-empty encoded suffixes is at least 1, $L[i] \oplus E(T[\mathsf{SA}[i]..]) < L[j] \oplus E(T[\mathsf{SA}[j]..])$ if $L[i] > 0$ and $L[j] = 0$ for any $0 < i, j \le n$ by Lemma 3. Therefore, we have $\mathsf{LF}[i] < \mathsf{LF}[j]$ for such i and j. By the definition of F, $F[\mathsf{LF}[i]] \ge 1$ and $F[\mathsf{LF}[j]] = 0$. ◀

6 Searching Algorithm with Trimmed LF-mapping

In this section, we devise a searching algorithm to compute the suffix range of a pattern P in $\mathcal{O}(|P|)$ time using the data structure described in the earlier section. Algorithm 3 shows the procedure to compute the suffix range of a given pattern. Starting with $E(P[|P|-1..]) = \infty$, it prepends $P[m-2], \cdots, P[0]$ at each iteration. Examples of updating a suffix range within an individual iteration are depicted in Figure 5.

6.1 Identifying the Target Suffixes

We need to identify the target suffixes in order to update the suffix range correctly. As we have discussed in Section 4.2, given a currently searched encoded pattern $E(P)$ and an integer k, we need to identify the suffixes $T[\mathsf{SA}[i]..]$'s such that $L[i] = k$ if $k < E(P).\mathsf{rank}_\infty(|P|)$,

Algorithm 3 Compute the suffix range of a pattern.

```
function SuffixRange($P[0..m-1]$):
    Compute $E(P)$
    Set $K[i] \leftarrow 0$ for $0 \leq i \leq m-2$
    for $i$=0 To $m-1$ do
        if $E(P)[i] \neq \infty$ then
            $K[i - E(P)[i]] \leftarrow K[i - E(P)[i]] + 1$
    end
    $p_s \leftarrow 1$ // the start position of the current suffix range.
    $p_e \leftarrow n$ // the end position of the current suffix range.
    $t \leftarrow 1$ // the number of $\infty$'s in the currently searched pattern.
    for $i$=2 To $m$ do
        $k \leftarrow K[m-i]$ // the number of $\infty$'s to be substituted.
        $p_s \leftarrow \mathsf{down}_k(p_s)$
        $p_e \leftarrow \mathsf{down}_k(p_e + 1) - 1$
        if $k < t$ then
            // Equivalent to $\mathsf{map}_k(\cdot)$ for the left- and rightmost 0's.
            $p_s \leftarrow B_k^{(F)}.\mathsf{select}_0(B_k^{(L)}.\mathsf{rank}_0(p_s) + 1)$
            $p_e \leftarrow B_k^{(F)}.\mathsf{select}_0(B_k^{(L)}.\mathsf{rank}_0(p_e + 1))$
        $p_s \leftarrow \mathsf{up}_k(p_s)$
        $p_e \leftarrow \mathsf{up}_k(p_e)$
        $t \leftarrow t - k + 1$
    end
    return $(p_s, p_e)$
```

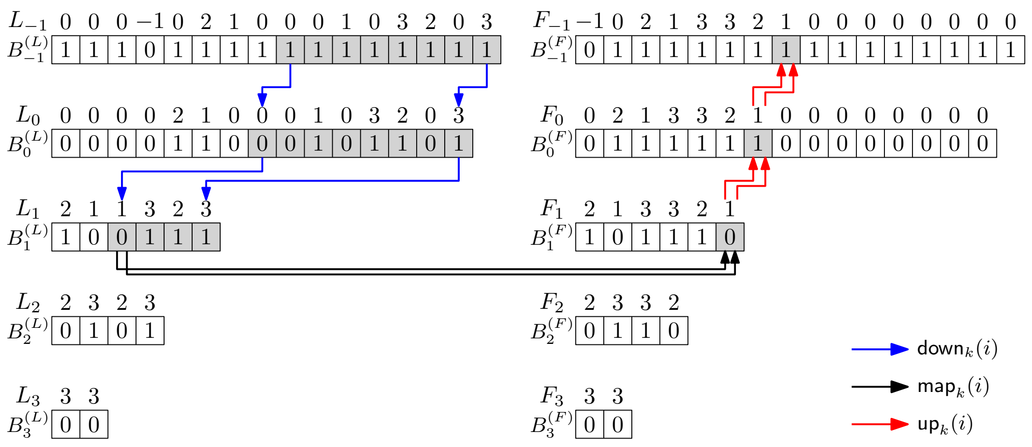
$L[i] \geq k$ if $k = E(P).\mathsf{rank}_\infty(|P|)$. This task can be done by computing the interval $[i_s, i_e]$ at level k that corresponds to the current suffix range $[p_s, p_e]$ (i.e. the interval on level -1) where $i_s = \mathsf{down}_k(p_s)$ and $i_e = \mathsf{down}_k(p_e + 1) - 1$. This is because L_k is a subsequence of L, whose elements are equal to or greater than k.

If $k < E(P).\mathsf{rank}_\infty(|P|)$, the target suffixes correspond to the elements that are equal to k, which can be determined by positions $\{i_s \leq i \leq i_e \mid B_k^{(L)}[i] = 0\}$. For example, in Figure 5-(a), the corresponding interval at level $k = 1$ of the suffix range $[8, 15]$ is $[2, 5]$. We can have $\{2\}$ as the set of positions whose value of the bitvector $B^{(L)}$ is 0. This position 2 at level 1 corresponds to the position 10 at level -1 that represent the entire elements of L. It means that $E(T[\mathsf{SA}[10]..])$ is the only target suffix, and we need to compute its LF-mapping to obtain the updated suffix range. We have $\mathsf{LF}[10] = 7$, and the updated suffix range is $[7, 7]$, which exactly matches the one obtained by the proposed algorithm.

If $k = E(P).\mathsf{rank}_\infty(|P|)$, all the elements within the interval $i_s \leq i \leq i_e$ correspond to the target suffixes. For example, in Figure 5-(b), we have $[i_s, i_e] = [1, 3]$. The positions at level -1 that corresponds to positions 1,2, and 3 at level $k = 2$, are 12, 13, and 15, respectively. Since $\mathsf{LF}[12] = 4$, $\mathsf{LF}[13] = 6$, and $\mathsf{LF}[15] = 5$, the updated suffix range is $[4, 6]$.

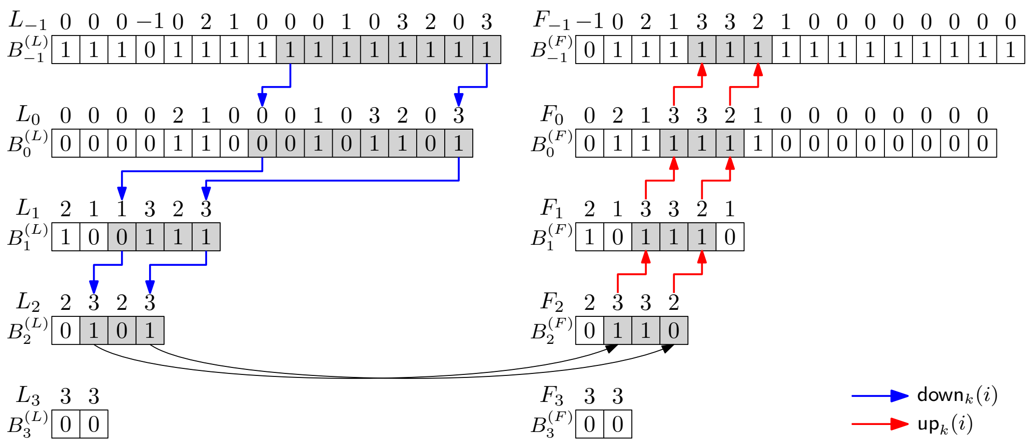
6.2 Computing the LF-mapping of the Target Suffixes

We have shown that the target suffixes can be correctly identified after calling $\mathsf{down}_k(\cdot)$. Thus the correctness of the algorithm relies on the correctness of Lines 15–17, which compute the positions on $B_k^{(F)}$ that correspond to the positions on $B_k^{(L)}$. If this mapping can identify the target suffixes in terms of an interval on F_k, then computing $\mathsf{up}_k(\cdot)$ will give the desired suffix range that is correctly updated with respect to the given $E(P)$ and k.



L_{-1} 0 0 0 −1 0 2 1 0 0 0 1 0 3 2 0 3
$B_{-1}^{(L)}$ 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
F_{-1} −1 0 2 1 3 3 2 1 0 0 0 0 0 0 0 0
$B_{-1}^{(F)}$ 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
L_0 0 0 0 0 2 1 0 0 0 1 0 3 2 0 3
$B_0^{(L)}$ 0 0 0 0 1 1 0 0 0 1 0 1 1 0 1
F_0 0 2 1 3 3 2 1 0 0 0 0 0 0 0 0
$B_0^{(F)}$ 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0
L_1 2 1 1 3 2 3
$B_1^{(L)}$ 1 0 0 1 1 1
F_1 2 1 3 3 2 1
$B_1^{(F)}$ 1 0 1 1 1 0
L_2 2 3 2 3
$B_2^{(L)}$ 0 1 0 1
F_2 2 3 3 2
$B_2^{(F)}$ 0 1 1 0
L_3 3 3
$B_3^{(L)}$ 0 0
F_3 3 3
$B_3^{(F)}$ 0 0
down$_k(i)$
map$_k(i)$
up$_k(i)$

(a) Case 1: $k = 1 < E(P).\mathsf{rank}_\infty(|P|) = 2$



L_{-1} 0 0 0 −1 0 2 1 0 0 0 1 0 3 2 0 3
$B_{-1}^{(L)}$ 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
F_{-1} −1 0 2 1 3 3 2 1 0 0 0 0 0 0 0 0
$B_{-1}^{(F)}$ 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
L_0 0 0 0 0 2 1 0 0 0 1 0 3 2 0 3
$B_0^{(L)}$ 0 0 0 0 1 1 0 0 0 1 0 1 1 0 1
F_0 0 2 1 3 3 2 1 0 0 0 0 0 0 0 0
$B_0^{(F)}$ 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0
L_1 2 1 1 3 2 3
$B_1^{(L)}$ 1 0 0 1 1 1
F_1 2 1 3 3 2 1
$B_1^{(F)}$ 1 0 1 1 1 0
L_2 2 3 2 3
$B_2^{(L)}$ 0 1 0 1
F_2 2 3 3 2
$B_2^{(F)}$ 0 1 1 0
L_3 3 3
$B_3^{(L)}$ 0 0
F_3 3 3
$B_3^{(F)}$ 0 0
down$_k(i)$
up$_k(i)$

(b) Case 2: $k = E(P).\mathsf{rank}_\infty(|P|) = 2$

Figure 5 An iteration of Algorithm 3 to update a suffix range. Given a suffix range $[8, 15]$ for the currently searched pattern $E(P) = \infty\infty$, it computes the suffix range for $k \oplus E(P)$.

It is obvious for the case $k < E(P).\mathsf{rank}_\infty(|P|)$. Because 0-bits (and their corresponding $L[i]$-values) in both bitvectors $B_k^{(L)}$ (and L_k) and $B_k^{(F)}$ (and F_k) are associated in order, it is sufficient to find the positions on $B_k^{(F)}$ that correspond to the first and last occurrence of 0-bit within the interval on $B_k^{(L)}$. We can determine the (relative) positions of the first and last 0-bits within the interval on $B_k^{(L)}$ using $B_k^{(L)}.\mathsf{rank}(\cdot)$ queries, and perform the mapping into their corresponding positions using $B_k^{(F)}.\mathsf{select}(\cdot)$ queries. This can be done by performing $\mathsf{map}_k(\cdot)$ at the leftmost and rightmost 0's in the interval.

It is not trivial if we are in the case $k = E(P).\mathsf{rank}_\infty(|P|)$. We observe that $\mathsf{up}_k(i) < \mathsf{up}_k(j)$ for $0 \leq i < j < |B_k^{(F)}|$ because $\mathsf{up}_k(\cdot)$ is order-preserving. Suppose we can compute the number of suffixes $E(T[\mathsf{SA}[i]..])$'s such that $L[i] \geq k$ and $\mathsf{LF}[i] < p'_s$ where (p'_s, p'_e) is the suffix range of $k \oplus E(P)$, say a to denote this number. Let b be the number of the target suffixes. Then the suffixes that belong to the updated suffix range (p'_s, p'_e) must correspond to the interval $[a, a + b - 1]$ on $B_k^{(F)}$. We claim the following:

▶ **Lemma 9.** *Let (p_s, p_e) be the suffix range of an encoded pattern $E(P)$, and let i and j be integers such that $L[i], L[j] \geq E(P).\mathsf{rank}_\infty(|P|)$. If $i < p_s \leq j$ or $i \leq p_e < j$, then $\mathsf{LF}[i] < \mathsf{LF}[j]$.*

Proof. Let $l = \mathsf{lcp}(E(T[\mathsf{SA}[i]..]), E(T[\mathsf{SA}[j]..]))$ and $t = E(T[\mathsf{SA}[i]..]).\mathsf{rank}_\infty(l)$. Clearly, $t \leq E(P).\mathsf{rank}_\infty(|P|)$ because one of $E(T[\mathsf{SA}[i]..])$ and $E(T[\mathsf{SA}[j]..])$ has $E(P)$ as its prefix while the other does not. We also have $E(T[\mathsf{SA}[i]..]) < E(T[\mathsf{SA}[j]..])$ since $i < j$. By Lemma 3, $L[i] \oplus E(T[\mathsf{SA}[i]..]) < L[j] \oplus E(T[\mathsf{SA}[j]..])$ ◀

Note that it does not matter if $\mathsf{tLF}_k(\cdot)$ is not order-preserving within $p_s \leq i, j \leq p_e$. Among the suffixes $E(T[\mathsf{SA}[i]..])$ such that $L[i] \geq k$, the suffixes that are smaller than $k \oplus E(P)$ will be mapped into those that are smaller than $k \oplus E(P)$ after applying the LF-mapping function, and the larger suffixes remain larger. Let $\mathcal{I}$ be the interval at level k that correspond to a suffix range (p_s, p_e). Then computing the set $\{\mathsf{map}_k(i) \mid i \in \mathcal{I}\}$ is identical to $\mathcal{I}$ itself, which proves the correctness of the mapped interval on $B_k^{(F)}$ in this case.

6.3 Time Complexity

The search time of Algorithm 3 depends on how many times $\mathsf{down}_k(\cdot)$ and $\mathsf{up}_k(\cdot)$ are called over the iterations. Note that each of $\mathsf{down}_k(\cdot)$ and $\mathsf{up}_k(\cdot)$ takes $\mathcal{O}(k)$ time. We have $\sum_i k_i < |P|$ because k_i is the number of ∞'s to be substituted at iteration i. Once an ∞ is substituted with an integer, it remains the same until the end of the search process. The total number of substitution is bounded by the pattern length $|P|$, thus the search time is also bounded by $\mathcal{O}(|P|)$ time. This completes the proof of the main theorem.

7 Conclusion and Open Problems

In this paper, we have proposed a $3n + o(n)$-bit index for the Cartesian tree matching problem. We achieved this space bound by representing the correspondence between adjacent encoded suffixes using unary code. To bound the search time within linear time, we introduced the concept of a *trimmed LF-mapping function*. The trimmed LF-mapping function has special properties which enable us to compute the updated suffix ranges efficiently.

We also have open problems that should be addressed in the future work as follows:

- **Can we achieve $2n + o(n)$ bits?** We need to achieve $2n + o(n)$ bits to make it *succinct* regarding the entropy bound of Cartesian trees. We do not think that $B_k^{(L)}$ can be further reduced, but we think $B_k^{(F)}$ can be represented in a space-efficient way. Nevertheless, it seems quite challenging to represent $B_k^{(F)}$ within $o(n)$ bits to achieve succinctness.
- **Can we efficiently locate the occurrences?** The standard method that uses a sampled suffix array may take a long time especially when a pattern is long and it is also very frequent, because the time complexity has an $\mathcal{O}(|P| \cdot occ)$ term, which would result in $\mathcal{O}(n^2)$ time for locating occurrences. Although it might be a rare case in practice, the algorithm should be improved in order to bound the worst case complexity.
- **Can we apply the trimmed LF-mapping to other indexing problems?** We may apply the trimmed LF-mapping when L-values or pattern lengths are bounded by $\mathcal{O}(1)$. We can also extend it into some other matching problems where the prepending operation to a suffix cannot be represented in a single value.

References

1 Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 114–125, 2014. doi:10.1007/978-3-662-43948-7_10.

2 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.

3 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.

4 Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing and Management*, 49(1):392–404, 2013. doi:10.1016/j.ipm.2012.08.003.

5 Vincent Cohen-Addad, Laurent Feuilloley, and Tatiana Starikovskaya. Lower bounds for text indexing with mismatches and differences. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1146–1164, 2019. doi:10.1137/1.9781611975482.70.

6 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.

7 Maxime Crochemore, Costas S. Iliopoulos, Thierry Lecroq, and Y. J. Pinzon. Approximate string matching in musical sequences. In *Proceedings of the Prague Stringology Conference*, pages 1–11, 2001.

8 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.

9 Erik D. Demaine, Gad M. Landau, and Oren Weimann. On cartesian trees and range minimum queries. *Algorithmica*, 68:610–625, 2014. doi:10.1007/s00453-012-9683-x.

10 Simone Faro, Thierry Lecroq, and Kunsoo Park. Fast practical computation of the longest common cartesian substring of two strings. In *Proceedings of the Prague Stringology Conference*, pages 48–60, 2020.

11 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.

12 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffery Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006. doi:10.1145/1198513.1198521.

13 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 35:1–13, 2017. doi:10.4230/LIPIcs.ISAAC.2017.35.

14 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.

15 Paweĺ Gawrychowski, Samah Ghazawi, and Gad M. Landau. On indeterminate string matching. In *Proceedings of the 31th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 14:1–14, 2020. doi:10.4230/LIPIcs.CPM.2020.14.

16 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.

17 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.

18 Geonmo Gu, Siwoo Song, Simone Faro, Thierry Lecroq, and Kunsoo Park. Fast multiple pattern cartesian tree matching. In *Proceedings of the 14th International Workshop on Algorithms and Computations (WALCOM)*, pages 107–119, 2020. doi:10.1007/978-3-030-39881-1_10.

19 Diptarama Hendrian. Generalized dictionary matching under substring consistent equivalence relations. In *Proceedings of the 14th International Workshop on Algorithms and Computations (WALCOM)*, pages 120–132, 2020. `doi:10.1007/978-3-030-39881-1_11`.

20 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`.

21 Natsumi Kikuchi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. Computing covers under substring consistent equivalence relations. In *Proceedings of the 27th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 131–146, 2020. `doi:10.1007/978-3-030-59212-7_10`.

22 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`.

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

24 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`.

25 Moshe Lewenstein. Indexing with gaps. In *Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 135–143, 2011. `doi:10.1007/978-3-642-24583-1_14`.

26 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`.

27 Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. `doi:10.1016/j.jda.2013.07.004`.

28 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`.

29 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:10.1007/s00453-003-1067-9`.

A Further Discussion

In this section, we discuss several considerations that can be addressed in the context of developing the compact index of Cartesian tree matching and its extension, including the problems mentioned in Section 7.

A.1 Reducing the Required Number of Bits Further

Although we achieved $3n + o(n)$ bits by representing $B^{(L)}_{-1}$, $B^{(F)}_{-1}$ and $B^{(F)}_{0}$ in $\mathcal{O}(\lg n)$ bits, we did not use any techniques to reduce the space occupancy for the other bitvectors. A straightforward method to reduce the space occupancy further is to use compressed bit vectors. We may also apply compression boosting techniques such as level-wise [11] and block-wise compression [26, 20] to bitvectors.

We can also notice that there must be some redundancies in bitvectors, because $B^{(F)}_{k}$ is not an arbitrary permutation of $B^{(L)}_{F}$. Although we used the fact that $B^{(F)}_{0}$ has a certain form, perhaps the other bitvectors $B^{(F)}_{k}$'s can be represented more efficiently rather than independently of $B^{(L)}_{k}$'s. We can make several observations regarding bitvectors and operations related to them, some of which are as follows.

▶ **Proposition 10.** *For $k \geq -1$ and $0 \leq i, j < |B_k^{(L)}|$, the following facts hold:*

1. *If $B_k^{(L)}[i] = 0$, then $\mathsf{map}_k(i) \geq i$.*
2. *If $B_k^{(L)}[i] = 1$, then $\mathsf{map}_k(i) \leq i$.*
3. *If $i < j$ and $B_k^{(L)}[i] = 1$ and $B_k^{(L)}[j] = 0$, then $\mathsf{map}_k(i) < \mathsf{map}_k(j)$.*

Proof. (1) For $0 \leq j < |B_k^{(L)}|$, let $f(j) = F_k.\mathsf{select}_x(L_k.\mathsf{rank}_x(j) + 1)$ where $x = L_k[j]$. Suppose there exists $0 \leq i < |B_k^{(L)}|$ such that $B_k^{(L)}[i] = 0$ and $i' = \mathsf{map}_k(i) < i$. Because the number of occurrences of k in $L_k[0..i]$ and $F_k[0..i']$ is the same, there must exist $0 \leq j < i$ such that $L[j] \neq k$ and $f(j) > i$. When we trace up to F_{-1}, which is order-preserving, this implies $\mathsf{LF}[i] < \mathsf{LF}[j]$. However, since we have $j < i$ and $L[j] > k = L[i]$, $E(T[\mathsf{SA}[\mathsf{LF}[j]]..]) = L[j] \oplus E(T[\mathsf{SA}[j]..]) < L[i] \oplus E(T[\mathsf{SA}[i]..]) = E(T[\mathsf{SA}[\mathsf{LF}[i]]..])$. This implies $\mathsf{LF}[j] < \mathsf{LF}[i]$. Contradiction.

(2) Note that $B_k^{(F)}.\mathsf{rank}_0(i+1) + B_k^{(F)}.\mathsf{rank}_1(i+1) = B_k^{(L)}.\mathsf{rank}_0(i+1) + B_k^{(L)}.\mathsf{rank}_1(i+1) = i$. Suppose there exists $\mathsf{map}_k(i) > i$ for i such that $B_k^{(L)}[i] = 1$. Then $B_k^{(F)}.\mathsf{rank}_1(i+1) < B_k^{(L)}.\mathsf{rank}_1(i+1)$. Since $\mathsf{map}_k(j) \geq j$ for $0 \leq j < |B_k^{(L)}|$ such that $B_k^{(L)}[j] = 0$, we also have $B_k^{(F)}.\mathsf{rank}_0(i+1) \leq B_k^{(L)}.\mathsf{rank}_0(i+1)$. Therefore $B_k^{(F)}.\mathsf{rank}_0(i+1) + B_k^{(F)}.\mathsf{rank}_1(i+1) < B_k^{(L)}.\mathsf{rank}_0(i+1) + B_k^{(L)}.\mathsf{rank}_1(i+1)$. Contradiction.

(3) Immediate from $\mathsf{map}_k(i) \leq i < j \leq \mathsf{map}_k(j)$. ◀

We believe that we can reduce the number of bits required to implement $B_k^{(F)}$'s if we can find more properties on the relation between $B_k^{(L)}$ and $B_k^{(F)}$. However, it is not trivial if we can break it down into $o(n)$ bits. One possibility to achieve $o(n)$ bits is the use of sparse bitvector if we can make $B_k^{(F)}$'s significantly imbalanced in terms of the number of 0- and 1-bits.

A.2 Locating Occurrences

To locate the occurrences, we can sample entries of the suffix array $\mathsf{SA}[i]$ as we conventionally do for other problems. More specifically, we build an array $\hat{\mathsf{SA}}$ storing $\mathsf{SA}[i]$ if it is divisible by δ.

$$\hat{\mathsf{SA}} = \langle \mathsf{SA}[i] \rangle_{i | \mathsf{SA}[i] = 0 \mod \delta} \tag{13}$$

Using a bitvector M of length $n+1$, we mark the sampled entry; i.e., $M[i] = 1$ iff $\mathsf{SA}[i] = 0 \mod \delta$. Algorithm 4 traces from a suffix backward until it meets the sampled entry of the suffix array.

Algorithm 4 Locating Occurrences.

```
function $Locate(i, t = E(P).\mathsf{rank}_\infty(|P|))$:
    $R \leftarrow \{\}$
    for $j = 0$ To $\delta - 1$ do
        if $i = 0$ then break
        if $M[i] = 1$ then $R \leftarrow R \cup \{\hat{\mathsf{SA}}[M.\mathsf{rank}_1(i)] + j\}$
        $k \leftarrow \min\{t, L[i]\}$
        $i \leftarrow \mathsf{tLF}_k(i)$
        $t \leftarrow t - k + 1$
    end
    return $R$
```

Unlike the conventional technique of the sampled suffix array, tracing from a single suffix does not necessarily report one entry in the sampled suffix array because $\mathsf{tLF}_k(i)$ does not guarantee to be the same as $\mathsf{LF}[i]$; during the iterations of the algorithm, they may be jumbled and it possibly jumps into another suffix other than its previous one. Nevertheless, reported positions are correct if we collect the reported positions over all suffixes in the suffix range. Note that at the end of each iteration j of the outer loop, $E(T[\mathsf{SA}[i]..])$ has the form of $k_1 \oplus (\cdots (k_j \oplus E(P)) \cdots)$, thus even if it jumps into another suffix, it is still correct one.

The time for locating all the occurrence is $\mathcal{O}((|P|+\delta)\cdot occ)$. The time taken for computing k in Line 6 can be done in $\mathcal{O}(\min\{t, L[i]\})$ time, because we can repeatedly move into the next level using $B_k^{(L)}.\mathsf{rank}_1(\cdot)$ until either we arrive at level t or we reach a 0-bit at level $L[i]$. A single computation of $\mathsf{tLF}_k(i)$ may take $\mathcal{O}(L[i])$ time, but it is bounded by $\mathcal{O}(|P|+\delta)$ because the sum of all k's over all iterations is the number of ∞'s that are substituted. There are at most $|P|$ ∞'s remaining in the searched part, and there are possibly $\delta - 1$ more ∞'s to be substituted during the iterations.

A.3 Extensions

We may apply the trimmed LF-mapping when L-values or pattern lengths are bounded by $\mathcal{O}(1)$. For example, the indexing methods for the parameterized string matching [14, 24] have a $\log\sigma$ factor in their space complexity, which comes from the fact that $L[i]$ is bounded by σ. If they are bounded by some constant c, we may adopt this technique with unary coded L-values to achieve $\mathcal{O}(n)$ bits and $\mathcal{O}(m)$ search time.

We can extend the concept of the trimmed LF-mapping introduced in this paper for other matching problems where the prepending operation to a suffix cannot be represented in a single value. The resulting structure would look like the combination of wavelet trees [17, 27] and DAC [4] with a larger size of blocks. For example, in [23], the number of pointers starting at a particular position is constrained to be at most 1. The main reason of this constraint was that it may cause an unbounded time complexity if there are multiple pointers having the same starting position. We can use the mechanism of the trimmed LF-mapping to relax this constraint.

A.4 Construction Time

In [28], it is shown that the suffix tree for the Cartesian tree matching problem can be built in randomized $\mathcal{O}(n)$ time or deterministic $\mathcal{O}(n \lg n)$ time based on the suffix tree construction algorithm based on the character oracles [6]. Once the suffix tree is constructed, we can compute the suffix array in $\mathcal{O}(n)$ time using the tree traversal. The array SA^{-1}, LF, L, and F can also be computed in order, each of which takes $\mathcal{O}(n)$ time; note that we can precompute $K(\cdot)$ required to compute L in $\mathcal{O}(n)$ time using Lines 2–7 of Algorithm 3. The remaining task is to build the corresponding bitvectors, which does not exceed this time bound.