



# Breaking a Barrier in Constructing Compact Indexes for Parameterized Pattern Matching

Kento Iseri 

Kyushu Institute of Technology, Japan

Tomohiro I  


Kyushu Institute of Technology, Japan

Diptarama Hendrian  



Tokyo Medical and Dental University, Japan

Dominik Köppl  

University of Yamanashi, Japan

Ryo Yoshinaka  

Tohoku University, Sendai, Japan

Ayumi Shinohara  

Tohoku University, Sendai, Japan

---

## Abstract

A parameterized string (p-string) is a string over an alphabet  $(\Sigma_s \cup \Sigma_p)$ , where  $\Sigma_s$  and  $\Sigma_p$  are disjoint alphabets for static symbols (s-symbols) and for parameter symbols (p-symbols), respectively. Two p-strings  $x$  and  $y$  are said to parameterized match (p-match) if and only if  $x$  can be transformed into  $y$  by applying a bijection on  $\Sigma_p$  to every occurrence of p-symbols in  $x$ . The indexing problem for p-matching is to preprocess a p-string  $T$  of length  $n$  so that we can efficiently find the occurrences of substrings of  $T$  that p-match with a given pattern. Let  $\sigma_s$  and respectively  $\sigma_p$  be the numbers of distinct s-symbols and p-symbols that appear in  $T$  and  $\sigma = \sigma_s + \sigma_p$ . Extending the Burrows-Wheeler Transform (BWT) based index for exact string pattern matching, Ganguly et al. [SODA 2017] proposed parameterized BWTs (pBWTs) to design the first compact index for p-matching, and posed an open problem on how to construct the pBWT-based index in compact space, i.e., in  $O(n \lg |\Sigma_s \cup \Sigma_p|)$  bits of space. Hashimoto et al. [SPIRE 2022] showed how to construct the pBWT for  $T$ , under the assumption that  $\Sigma_s \cup \Sigma_p = [0..O(\sigma)]$ , in  $O(n \lg \sigma)$  bits of space and  $O(n \frac{\sigma_p \lg n}{\lg \lg n})$  time in an online manner while reading the symbols of  $T$  from right to left. In this paper, we refine Hashimoto et al.'s algorithm to work in  $O(n \lg \sigma)$  bits of space and  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time in a more general assumption that  $\Sigma_s \cup \Sigma_p = [0..n^{O(1)}]$ . Our result has an immediate application to constructing parameterized suffix arrays in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of working space. We also show that our data structure can support backward search, a core procedure of BWT-based indexes, at any stage of the online construction, making it the first compact index for p-matching that can be constructed in compact space and even in an online manner.

**2012 ACM Subject Classification** Theory of computation  $\rightarrow$  Pattern matching

**Keywords and phrases** Index for parameterized pattern matching, Parameterized Burrows-Wheeler Transform, Online construction

**Digital Object Identifier** 10.4230/LIPIcs.ICALP.2024.89

**Category** Track A: Algorithms, Complexity and Games

**Related Version** *Full Version*: <https://arxiv.org/abs/2308.05977>

**Funding** *Tomohiro I*: KAKENHI (Grant Numbers 19K20213, 22K11907).

*Dominik Köppl*: KAKENHI (Grant Number 23H04378).

*Ryo Yoshinaka*: KAKENHI (Grant Numbers 18K11150, 20H05703, 23K11325, 24K14827).

*Ayumi Shinohara*: KAKENHI (Grant Number 21K11745).



© Kento Iseri, Tomohiro I, Diptarama Hendrian, Dominik Köppl, Ryo Yoshinaka, and Ayumi Shinohara;

licensed under Creative Commons License CC-BY 4.0

51st International Colloquium on Automata, Languages, and Programming (ICALP 2024).

Editors: Karl Bringmann, Martin Grohe, Gabriele Puppis, and Ola Svensson;

Article No. 89; pp. 89:1–89:19



Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



## 1 Introduction

A *parameterized string* (*p-string*) is a string over an alphabet  $(\Sigma_s \cup \Sigma_p)$ , where  $\Sigma_s$  and  $\Sigma_p$  are disjoint alphabets for *static symbols* (*s-symbols*) and for *parameter symbols* (*p-symbols*), respectively. Two p-strings  $x$  and  $y$  are said to *parameterized match* (*p-match*) if and only if  $x$  can be transformed into  $y$  by applying a bijection on  $\Sigma_p$  to every occurrence of p-symbols in  $x$ . For example with  $\Sigma_s = \{a, b\}$  and  $\Sigma_p = \{X, Y, Z\}$ , two p-strings  $aXYbZXaY$  and  $aZYbXZaY$  p-match because  $aXYbZXaY$  can be transformed into  $aZYbXZaY$  by replacing  $X, Y$  and  $Z$  with  $Z, Y$  and  $X$ , respectively. The concept of p-matching was introduced by Baker aiming at software maintenance and plagiarism detection [1, 2, 3], and has been extensively studied in the last decades (see a recent survey [28] and references therein).

The indexing problem for p-matching is to preprocess a p-string  $T$  of length  $n$  so that we can efficiently find the occurrences of substrings of  $T$  that p-match with a given pattern. Solutions proposed for this problem adapt and extend indexes initially devised for exact string pattern matching, e.g., parameterized suffix trees [1, 25, 2, 3], parameterized suffix arrays [8, 20, 4, 12], parameterized suffix trays [14], parameterized DAWGs [31], parameterized position heaps [9, 11, 13] and parameterized Burrows-Wheeler transforms (pBWTs) based indexes [16, 24, 18].

Among these indexes, pBWT-based indexes are the most space economic, consuming  $n \lg |\Sigma_s \cup \Sigma_p| + O(n)$  bits [16] or  $2n \lg |\Sigma_s \cup \Sigma_p| + 2n + o(n)$  bits with a simplified version proposed in [24]. Let  $\sigma_s$  and respectively  $\sigma_p$  be the numbers of distinct s-symbols and p-symbols that appear in  $T$  and  $\sigma = \sigma_s + \sigma_p$ . The pBWT-based index of  $T$  can be constructed via the parameterized suffix tree of  $T$  for which  $O(n(\lg \sigma_s + \lg \sigma_p))$ -time or randomized  $O(n)$ -time construction algorithms are known [25, 7, 26], but the intermediate memory footprint of  $O(n \lg n)$  bits could be intolerable when it is significantly larger than the resulting index. Hashimoto et al. [19] showed how to compute the pBWT of [24] for  $T$ , under the assumption that  $\Sigma_s \cup \Sigma_p = [0..O(\sigma)]$ , in  $O(n \lg \sigma)$  bits and  $O(n \frac{\sigma_p \lg n}{\lg \lg n})$  time in an online manner while reading the symbols of  $T$  from right to left. Here we note that the work of [19] lacks details in terms of pBWT-based index construction because any pBWT-based index to date [16, 24, 18] requires additional data structures other than the pBWT, and the pBWTs alone does not seem to be enough to support p-matching queries efficiently.

In this paper, we refine the algorithm of [19] to work in  $O(n \lg \sigma)$  bits and  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time in a more general assumption that  $\Sigma_s \cup \Sigma_p = [0..n^{O(1)}]$ . While working in compact space, i.e.,  $O(n \lg \sigma)$  bits, it achieves  $o(n \sigma_p)$  time when  $\sigma_p = \omega(\lg n)$ . This is of great interest because the time complexity of  $o(n \sigma_p)$  has not been achieved in the construction for p-matching indexes even in the offline setting unless we resort to a fast construction algorithm for parameterized suffix trees using  $O(n \lg n)$  bits. In particular, the currently best worst-case result for the direct construction of parameterized suffix arrays is  $O(n \sigma_p)$  time and  $O(n \lg n)$  bits of working space [12]. Since our online-built data structure for  $T$  can be used to compute the parameterized suffix array of  $T$  in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time, we obtain a new way to construct parameterized suffix arrays in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of working space.

We also show that our data structure can support backward search, a core procedure of BWT-based indexes, at any stage of the online construction, making it the first compact index for p-matching that can be constructed in compact space and even in an online manner. This cannot likely be achieved with the previous work [19] due to the lack of support for 2D range counting queries in the data structure it uses.

Our computational assumptions are as follows:

- We assume a standard Word-RAM model with word size  $\Omega(\lg n)$ .
- Each symbol in  $(\Sigma_s \cup \Sigma_p)$  is represented by  $O(\lg n)$  bits, namely, a symbol is from the universe  $[0..n^{O(1)}]$ .
- We can check membership for a given symbol ( $\in \Sigma_s \cup \Sigma_p$ ) in  $\Sigma_s$  and  $\Sigma_p$  in  $O(1)$  time, e.g., by having some flag bits or thresholds separating both alphabet sets.
- The order of two s-symbols can be determined in  $O(1)$  time based on their bit representations.

An index of a p-string  $T$  for p-matching is to support, given a pattern  $w$ ,

1. the *counting query* that asks to compute the number of occurrences of substrings in  $T$  that p-match with  $w$  and
2. the *locating query* that asks to compute the positions of these counted occurrences in  $T$ .

The number of occurrences returned for a locating query of  $w$  is the answer to the counting query of  $w$ . Since these occurrences can be at arbitrary positions of  $T$  in general, the time complexity for the locating query depends usually on the number of these occurrences. In contrast, most indexes based on the BWT can answer counting queries in time independent to this number, by leveraging the so-called *backward search*. By using backward search, our time complexities for both queries resemble those of other BWT-based indexes, with some additional logarithmic terms. In detail, our main result is as follows:

► **Theorem 1.** *For a p-string  $T$  of length  $n$  over an alphabet  $(\Sigma_s \cup \Sigma_p)$  of size  $n^{O(1)}$ , an index of  $T$  for p-matching can be constructed online in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of space, where  $\sigma_s$  and respectively  $\sigma_p$  are the numbers of distinct s-symbols and p-symbols used in the p-string and  $\sigma = \sigma_s + \sigma_p$ . At any stage of the online construction, it can support the counting queries in  $O(m \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time, where  $m$  is the length of a given pattern for queries. By building an additional data structure of  $O(\frac{n}{\Delta} \lg n)$  bits of space for a chosen parameter  $\Delta \in \{1, 2, \dots, n\}$  the locating queries can be supported in  $O(m \frac{\lg \sigma_p \lg n}{\lg \lg n} + \text{occ} \frac{\Delta \lg n}{\lg \lg n})$  time, where  $\text{occ}$  is the number of occurrences to be reported.*

We also obtain the following result for constructing the parameterized suffix array:

► **Theorem 2.** *For a p-string  $T$  of length  $n$  over an alphabet  $(\Sigma_s \cup \Sigma_p)$  of size  $n^{O(1)}$ , the parameterized suffix array of  $T$  can be constructed in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of space, where  $\sigma_s$  and respectively  $\sigma_p$  are the numbers of distinct s-symbols and p-symbols used in the p-string and  $\sigma = \sigma_s + \sigma_p$ .*

## 2 Preliminaries

### 2.1 Basic notations and tools

We denote with  $\lg = \log_2$  the logarithm with base two. An integer interval  $\{i, i + 1, \dots, j\}$  is denoted by  $[i..j]$ , where  $[i..j]$  represents the empty interval if  $i > j$ .

Let  $\Sigma$  be an ordered finite *alphabet*. An element of  $\Sigma^*$  is called a *string* over  $\Sigma$ . The length of a string  $w$  is denoted by  $|w|$ . The empty string  $\varepsilon$  is the string of length 0, that is,  $|\varepsilon| = 0$ . Let  $\Sigma^+ = \Sigma^* - \{\varepsilon\}$  and  $\Sigma^k = \{x \in \Sigma^* \mid |x| = k\}$  for any non-negative integer  $k$ . The concatenation of two strings  $x$  and  $y$  is denoted by  $x \cdot y$  or simply  $xy$ . When a string  $w$  is represented by the concatenation of strings  $x$ ,  $y$  and  $z$  (i.e.,  $w = xyz$ ), then  $x$ ,  $y$  and  $z$  are called a *prefix*, *substring*, and *suffix* of  $w$ , respectively. A substring  $x$  of  $w$  is called *proper* if  $x \neq w$ .

The  $i$ -th symbol of a string  $w$  is denoted by  $w[i]$  for  $1 \leq i \leq |w|$ , and the substring of a string  $w$  that begins at position  $i$  and ends at position  $j$  is denoted by  $w[i..j]$  for  $1 \leq i \leq j \leq |w|$ , i.e.,  $w[i..j] = w[i]w[i+1]\cdots w[j]$ . For convenience, let  $w[i..j] = \varepsilon$  if  $j < i$ ; further let  $w[..i] = w[1..i]$  and  $w[i..] = w[i..|w|]$  denote abbreviations for the prefix of length  $i$  and the suffix starting at position  $i$ , respectively. For two strings  $x$  and  $y$ , let  $\text{lcp}(x, y)$  denote the length of the longest common prefix between  $x$  and  $y$ . We consider the lexicographic order over  $\Sigma^*$  by extending the strict total order  $<$  defined on  $\Sigma$ :  $x$  is lexicographically smaller than  $y$  (denoted as  $x < y$ ) if and only if either  $x$  is a proper prefix of  $y$  or  $x[\text{lcp}(x, y) + 1] < y[\text{lcp}(x, y) + 1]$  holds. In this paper, we will ignore the former case since we mainly consider the lexicographic order between distinct strings that have a sentinel (end-marker) at the end of the strings so that  $x$  cannot be a proper prefix of  $y$ .

For any string  $w$ , character  $c$ , and position  $i$  ( $1 \leq i \leq |w|$ ),  $\text{rank}_c(w, i)$  returns the number of occurrences of  $c$  in  $w[..i]$  and  $\text{select}_c(w, i)$  returns the  $i$ -th occurrence of  $c$  in  $w$ . For  $1 \leq i \leq j \leq |w|$ , a *range minimum query*  $\text{RmQ}_w(i, j)$  asks for  $\arg \min_{i \leq k \leq j} \{w[k]\}$ . We also consider *find previous/next queries*  $\text{FPQ}_p(w, i)$  and  $\text{FNQ}_p(w, i)$ , where  $p$  is a predicate either in the form of “ $c$ ” (equal to  $c$ ), “ $< c$ ” (less than  $c$ ) or “ $\geq c$ ” (larger than or equal to  $c$ ):  $\text{FPQ}_p(w, i)$  returns the largest position  $j \leq i$  at which  $w[j]$  satisfies the predicate  $p$ . Symmetrically,  $\text{FNQ}_p(w, i)$  returns the smallest position  $j \geq i$  at which  $w[j]$  satisfies the predicate  $p$ . For example with the integer string  $w = [2, 5, 10, 6, 8, 3, 14, 5]$ ,  $\text{FNQ}_5(w, 4) = 8$ ,  $\text{FNQ}_6(w, 4) = 4$ ,  $\text{FPQ}_5(w, 4) = 2$ ,  $\text{FNQ}_{<5}(w, 4) = 6$ ,  $\text{FPQ}_{<5}(w, 4) = 1$ ,  $\text{FNQ}_{\geq 9}(w, 4) = 7$  and  $\text{FPQ}_{\geq 9}(w, 4) = 3$ .

If the answer of  $\text{select}_c(w, i)$ ,  $\text{FPQ}_p(w, i)$  or  $\text{FNQ}_p(w, i)$  does not exist, it is just ignored. To handle this case of non-existence, we would use them in an expression with  $\min$  or  $\max$ : For example,  $\max\{1, \text{FPQ}_p(w, i)\}$  returns 1 if  $\text{FPQ}_p(w, i)$  does not exist.

Dynamic strings should support insertion/deletion of a symbol to/from any position as well as fast random access. We use the following result:

► **Lemma 3** ([29]). *A dynamic string of length  $n$  over an alphabet  $[0..U]$  can be implemented while supporting random access, insertion, deletion, rank and select queries in  $(n + o(n)) \lg U$  bits of space and  $O(\frac{\lg n}{\lg \lg n})$  query and update times.*

Dynamic binary strings equipped with rank and select queries can be used as a building block for the dynamic wavelet matrix [6] of a string over an alphabet  $[0..U]$  to support queries beyond rank and select. The idea is that each of the other queries can be simulated by performing one of the building block queries on every level of the wavelet matrix, which has  $\lceil \lg U \rceil$  levels, cf. [32, Section 6.2].

► **Lemma 4.** *A dynamic string of length  $n$  over an alphabet  $[0..U]$  with  $U = O(n)$  can be implemented while supporting random access, insertion, deletion, rank, select, RmQ, FPQ and FNQ queries in  $(n + o(n)) \lceil \lg U \rceil$  bits of space and  $O(\frac{\lg U \lg n}{\lg \lg n})$  query and update times.*

## 2.2 Parameterized strings

Let  $\Sigma_s$  and  $\Sigma_p$  denote two disjoint sets of symbols. We call a symbol in  $\Sigma_s$  a *static symbol* (*s-symbol*) and a symbol in  $\Sigma_p$  a *parameter symbol* (*p-symbol*). A *parameterized string* (*p-string*) is a string over  $(\Sigma_s \cup \Sigma_p)$ . Let  $\$$  be the smallest s-symbol, which will be used as an end-marker of p-strings. Let  $\infty$  represent a symbol that is larger than any integer, and let  $\mathbf{N}_\infty = \mathbf{N}_+ \cup \{\infty\}$  be the set of positive integers  $\mathbf{N}_+$  including infinity ( $\infty$ ). Logically we assume that  $\mathbf{N}_\infty \cap \Sigma_s = \emptyset$  and  $(\mathbf{N}_\infty \cup \Sigma_s)$  is an ordered alphabet such that all s-symbols are smaller than any element in  $\mathbf{N}_\infty$ . For practical implementations, we require that s-symbols and integers can be distinguished in constant time (e.g., by shifting the ranges of the domains). Also, the conceptual symbol  $\infty$  can be treated as the finite value  $\sigma_p + 1$ .

For any p-string  $w$  the *p-encoded string*  $\langle w \rangle$  of  $w$ , also proposed as  $\text{prev}_\infty(w)$  in [24], is the string in  $(\mathbf{N}_\infty \cup \Sigma_s)^{|w|}$  such that

$$\langle w \rangle[i] = \begin{cases} w[i] & \text{if } w[i] \in \Sigma_s, \\ \infty & \text{if } w[i] \in \Sigma_p \text{ and } w[i] \text{ does not appear in } w[..i-1], \\ i-j & \text{otherwise,} \end{cases}$$

where  $j$  is the largest position in  $[1..i-1]$  with  $w[i] = w[j]$ . To put in words, we transformed each occurrence of a p-symbol into the distance to the previous occurrence of the same p-symbol, or  $\infty$  if it is the leftmost occurrence. Two p-strings  $x$  and  $y$  p-match if and only if  $\langle x \rangle = \langle y \rangle$ . On the one hand, the transformation from  $w$  to  $\langle w \rangle$  is *prefix-consistent*, i.e.,  $\langle w \rangle = \langle wc \rangle[..|w|]$  for any symbol  $c \in (\Sigma_s \cup \Sigma_p)$ . On the other hand,  $\langle w \rangle$  and  $\langle cw \rangle[2..]$  differ if and only if  $c \in \Sigma_p$  occurs in  $w$ . If it is the case, the leftmost occurrence  $h$  of  $c$  in  $w$  is the unique position such that  $\langle w \rangle$  and  $\langle cw \rangle[2..]$  differ with  $\langle w \rangle[h] = \infty$  and  $\langle cw \rangle[2..][h] = \langle cw \rangle[h+1] = h$ , i.e.,  $h = \text{select}_c(w, 1)$  and  $h+1 = \text{select}_c(cw, 2)$ .

For any p-string  $w$ , let  $|w|_p$  denote the number of distinct p-symbols in  $w$ , i.e.,  $|w|_p = \text{rank}_\infty(\langle w \rangle, |w|)$ . We define a function  $\pi$  that maps a non-empty p-string  $w \in (\Sigma_s \cup \Sigma_p)^+$  to an element in  $(\Sigma_s \cup [1..|w|_p])$  such that  $\pi(w)$  is  $w[1]$  if  $w[1]$  is an s-symbol; otherwise  $\pi(w)$  is the number of *distinct* p-symbols in  $w[..h+1]$ , where  $h+1$  is either the position of the second occurrence of  $w[1]$  in  $w$  or  $|w|$  if  $w[1]$  is unique in  $w$ . More formally,

$$\pi(w) = \begin{cases} w[1] & \text{if } w[1] \in \Sigma_s, \\ |w[..h+1]|_p & \text{otherwise,} \end{cases}$$

where  $h+1 = \min\{|w|, \text{select}_{w[1]}(w, 2)\}$ . In the second case,  $\pi(w)$  is considered to represent the rank of p-symbol  $w[1]$  when p-symbols are sorted in increasing order of the leftmost positions they appear in  $w[2..]$ , considering the rank of p-symbols not in  $w[2..]$  to be  $|w|_p$ . If  $\text{select}_{w[1]}(w, 2)$  exists, it holds that  $h = \text{select}_\infty(\langle w[2..] \rangle, \pi(w))$ . For convenience, we extend the domain of  $\pi$  to handle the empty string with  $\pi(\varepsilon) = \$$ .

For two p-strings  $x$  and  $y$ ,  $\text{lcp}^\infty(\langle x \rangle, \langle y \rangle)$  denotes the number of  $\infty$ 's in the longest common prefix of  $\langle x \rangle$  and  $\langle y \rangle$ .

Our algorithm heavily relies on the properties of the p-string encoding and  $\pi$ . For any p-strings  $x$  and  $y$ , Table 1 shows a complete list of cases for  $\text{lcp}(\langle x \rangle, \langle y \rangle)$ ,  $\text{lcp}^\infty(\langle x \rangle, \langle y \rangle)$  and the lexicographic order between  $\langle x \rangle$  and  $\langle y \rangle$ . The correctness immediately follows from the definition of the p-string encoding and  $\pi$  (see Figure 1 for illustrations). It is worth noting that Case (B3) is the only case in Cases (B1)-(B4) where we have  $\langle y \rangle < \langle x \rangle$ , i.e., the lexicographic order is changed after extension.

By Table 1, we have the following corollaries:

- **Corollary 5.** For any p-strings  $x$  and  $y$ ,  $\text{lcp}^\infty(\langle x \rangle, \langle y \rangle) \leq \text{lcp}^\infty(\langle x[2..] \rangle, \langle y[2..] \rangle) + 1$ .
- **Corollary 6.** For any p-strings  $x$  and  $y$  with  $\pi(x) = \pi(y)$ ,  $\langle x \rangle < \langle y \rangle$  if and only if  $\langle x[2..] \rangle < \langle y[2..] \rangle$ .
- **Corollary 7.** For any p-strings  $x$  and  $y$  (whether  $\langle x[2..] \rangle < \langle y[2..] \rangle$  or  $\langle x[2..] \rangle > \langle y[2..] \rangle$ ) with  $\pi(x) \leq \text{lcp}^\infty(\langle x[2..] \rangle, \langle y[2..] \rangle)$  and  $\pi(x) < \pi(y)$ , it holds that  $\langle x \rangle < \langle y \rangle$ . Note that  $\pi(x)$  and/or  $\pi(y)$  can be s-symbols.

■ **Table 1** All cases for  $\text{lcp}(\langle x \rangle, \langle y \rangle)$ ,  $\text{lcp}^\infty(\langle x \rangle, \langle y \rangle)$  and the lexicographic order between  $\langle x \rangle$  and  $\langle y \rangle$  for p-strings  $x$  and  $y$  over  $(\Sigma_s \cup \Sigma_p)$  with  $\lambda = \text{lcp}(\langle x[2..] \rangle, \langle y[2..] \rangle) < \min\{|x|, |y|\}$ ,  $e = \text{lcp}^\infty(\langle x[2..] \rangle, \langle y[2..] \rangle)$  and  $\langle x[2..] \rangle < \langle y[2..] \rangle$ . On the one hand, a case starting with letter A assumes that at least one of  $\pi(x)$  and  $\pi(y)$  is in  $\Sigma_s$ , while on the other hand, a case starting with letter B assumes that none of  $\pi(x)$  and  $\pi(y)$  is in  $\Sigma_s$ . We let  $h = \text{select}_{x[1]}(x, 2) - 1$  in Case (B2) and  $h' = \text{select}_{y[1]}(y, 2) - 1$  in Case (B3), both of which always exist because the conditions of Cases (B2) and (B3) imply that  $\pi(x) \neq \infty$  and  $\pi(y) \neq \infty$ , respectively.

cases	additional conditions	$\text{lcp}(\langle x \rangle, \langle y \rangle)$	$\text{lcp}^\infty(\langle x \rangle, \langle y \rangle)$	lexicographic order
(A1)	$\pi(x) \neq \pi(y)$	0	0	$\langle x \rangle < \langle y \rangle$ iff $\pi(x) < \pi(y)$
(A2)	$\pi(x) = \pi(y)$	$\lambda + 1$	$e$	$\langle x \rangle < \langle y \rangle$
(B1)	$\pi(x) = \pi(y) \leq e$	$\lambda + 1$	$e$	$\langle x \rangle < \langle y \rangle$
(B2)	$\pi(x) \leq e$ and $\pi(x) < \pi(y)$	$h$	$\pi(x)$	$\langle x \rangle < \langle y \rangle$
(B3)	$\pi(y) \leq e$ and $\pi(y) < \pi(x)$	$h'$	$\pi(y)$	$\langle y \rangle < \langle x \rangle$
(B4)	$e < \min\{\pi(x), \pi(y)\}$	$\lambda + 1$	$e + 1$	$\langle x \rangle < \langle y \rangle$

■ **Table 2** An example of  $R_T^{-1}(i)$ ,  $\text{LCP}_T^\infty$ ,  $L_T$  and  $F_T$  for a p-string  $T = \text{XYaZYXaZXZa\$}$  with  $\Sigma_s = \{\text{a}\}$  and  $\Sigma_p = \{\text{X, Y, Z}\}$ .

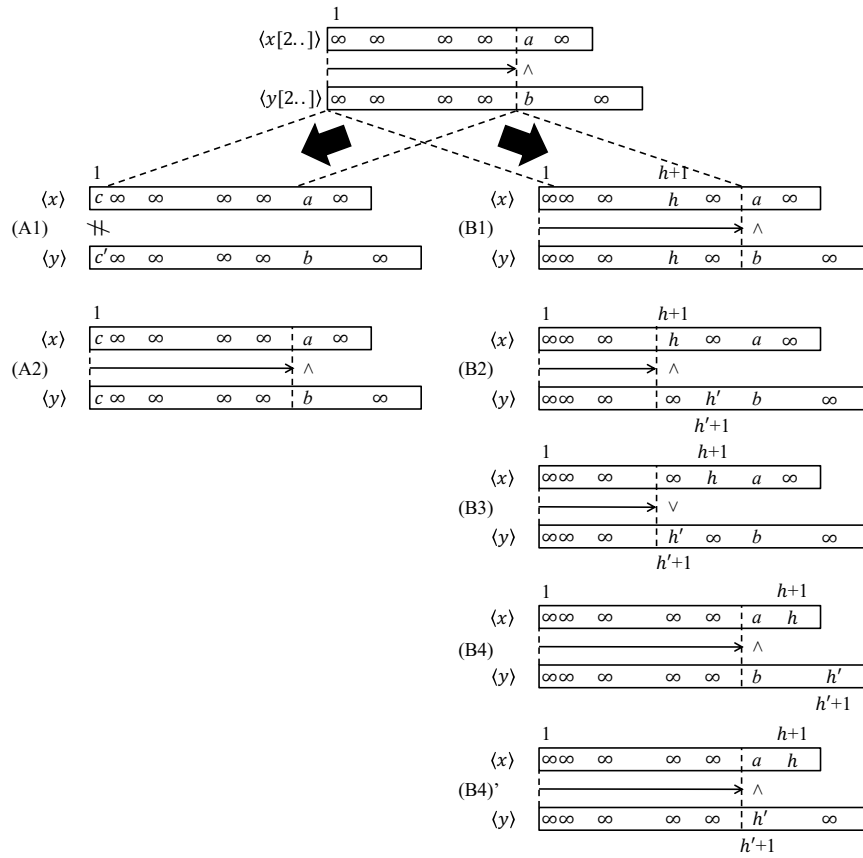
$i$	$T[i..]$	$\langle T[i..] \rangle$	$R_T^{-1}(i)$	$\text{LCP}_T^\infty[i]$	$L_T[i]$	$F_T[i]$	$\langle T[R_T^{-1}(i)..] \rangle$
1	XYaZYXaZXZa\$	xxax35a432a\$	12	0	a	\$	\$
2	YaZYXaZXZa\$	ax3xa432a\$	11	0	1	a	a\$
3	aZYXaZXZa\$	axxa432a\$	7	0	2	a	axx2a\$
4	ZYXaZXZa\$	xxa432a\$	3	2	2	a	axxa432a\$
5	YXaZXZa\$	xxax32a\$	10	0	2	1	xa\$
6	XaZXZa\$	ax32a\$	6	1	3	2	ax32a\$
7	aZXZa\$	ax2a\$	2	2	3	2	ax3xa432a\$
8	ZXZa\$	xx2a\$	9	1	2	2	xxa\$
9	XZa\$	xxa\$	5	2	3	3	xxax32a\$
10	Za\$	xa\$	1	3	\$	3	xxax35a432a\$
11	a\$	a\$	8	2	a	2	xx2a\$
12	\$	\$	4	2	a	3	xxxa432a\$

Let  $T$  be a p-string that has the smallest s-symbol  $\$$  as its end-marker, i.e.,  $T[|T|] = \$$  and  $\$$  does not appear anywhere else in  $T$ . The suffix rank function  $R_T : [1..|T|] \rightarrow [1..|T|]$  for  $T$  maps a position  $i$  ( $1 \leq i \leq |T|$ ) to the lexicographic rank of  $\langle T[i..] \rangle$  in  $\{\langle T[j..] \rangle \mid 1 \leq j \leq |T|\}$ . Its inverse function  $R_T^{-1}(i)$  returns the starting position of the lexicographically  $i$ -th p-encoded suffix of  $T$ .<sup>1</sup>

The *parameterized Burrows-Wheeler Transform* (*pBWT*) of  $T$  is the string  $L_T$  of length  $|T|$  over  $(\Sigma_s \cup [1..|T|_p])$  such that  $L_T[i] = \pi(T[R_T^{-1}(i) - 1..])$ , where we assume that  $T[0..] = \$$ . Another string  $F_T$  of length  $|T|$  is defined as  $F_T[i] = \pi(T[R_T^{-1}(i)..])$ .<sup>2</sup> Since  $\{T[R_T^{-1}(i)..] \mid 1 \leq i \leq |T|\} = \{T[R_T^{-1}(i) - 1..] \mid 1 \leq i \leq |T|\}$  is equivalent to the set of all non-empty suffixes of  $T$ ,  $F_T$  is a permutation of  $L_T$ .

<sup>1</sup>  $R_T^{-1}$  and  $R_T$  are essentially equivalent to parameterized suffix arrays and inverse parameterized suffix arrays, respectively.

<sup>2</sup> Previous studies [16, 24, 19] define pBWTs based on sorted cyclic rotations, but our suffix-based definition is more suitable for online construction to prevent unnecessary updates on  $F_T$  and  $L_T$ .



■ **Figure 1** Illustrations for the cases of Table 1. The two bold diagonal arrows on the top separate the cases starting with letter A (left side) from the others, starting with B (right side). Each horizontal right-facing arrow represents the longest common prefix of two p-encoded strings, and the lexicographic order between them is determined by the following p-encoded symbols. Particularly, we let  $a = \langle x[2..] \rangle[\lambda + 1]$  and  $b = \langle y[2..] \rangle[\lambda + 1]$ , where  $\lambda = \text{lcp}(\langle x[2..] \rangle, \langle y[2..] \rangle)$ . Since  $a < b$ , it holds that  $a \leq \lambda$  while  $b \leq \lambda$  or  $b = \infty$ . For Case (B1),  $h = \text{select}_{x[1]}(x, 2) - 1 = \text{select}_{y[1]}(y, 2) - 1$ . For Case (B2)-(B4) and (B4)',  $h = \text{select}_{x[1]}(x, 2) - 1$  and  $h' = \text{select}_{y[1]}(y, 2) - 1$ , some of which are not necessarily defined (when  $\pi(x)$  or  $\pi(y)$  is  $\infty$ ) but assumed to be present in illustrations. Case (B4)' illustrates the case with  $b = \infty$  and  $h' = \lambda + 1$ , which is included in Case (B4).

The so-called LF-mapping  $\text{LF}_T$  maps a position  $i$  to  $R_T(R_T^{-1}(i) - 1)$  if  $R_T^{-1}(i) > 1$ , and otherwise  $R_T(|T|) = 1$ . By definition and Corollary 6, we have:

► **Corollary 8.** For any p-string  $T$  and any integers  $i, j$  with  $1 \leq i < j \leq |T|$ ,  $\text{LF}_T(i) < \text{LF}_T(j)$  if  $\text{L}_T[i] = \text{L}_T[j]$ .

Thanks to Corollary 8, it holds that  $\text{LF}_T(i) = \text{select}_c(F_T, \text{rank}_c(\text{L}_T, i))$ , where  $c = \text{L}_T[i]$ . The inverse function  $\text{FL}_T$  of  $\text{LF}_T$  can be computed by  $\text{FL}_T(i) = \text{select}_c(\text{L}_T, \text{rank}_c(F_T, i))$ , where  $c = F_T[i]$ .

Let  $\text{LCP}_T^\infty$  be the string of length  $|T|$  such that  $\text{LCP}_T^\infty[1] = 0$  and  $\text{LCP}_T^\infty[i] = \text{lcp}^\infty(\langle T[R_T^{-1}(i-1)..] \rangle, \langle T[R_T^{-1}(i)..] \rangle)$  for every  $1 < i \leq |T|$ . An example of all explained arrays is given in Table 2.

### 3 Online construction algorithm

To construct our index for p-matching online, we maintain  $F_T$ ,  $L_T$ , and  $LCP_T^\infty$  with dynamic data structures while prepending a symbol to the current p-string  $T$ . The details of the data structures will be presented in Subsection 3.3. In what follows, we focus on a single step of updating  $T$  to  $\hat{T} = cT$  for some symbol  $c$  in  $\Sigma_s \cup \Sigma_p$ . Note that  $F_T$ ,  $L_T$  and  $LCP_T^\infty$  are strongly related to the sorted p-encoded suffixes of a p-string and  $\hat{T} = cT$  is the only suffix that was not in the suffixes of  $T$ . Let  $k = R_T(1)$  and  $\hat{k} = R_{\hat{T}}(1)$ . In order to deal with the new emerging suffix  $\hat{T}$ , we compute the lexicographic rank  $\hat{k}$  of  $\langle \hat{T} \rangle$  among the non-empty p-encoded suffixes of  $\hat{T}$ . Then  $F_{\hat{T}}$  and  $L_{\hat{T}}$  can be obtained by replacing  $\$$  in  $L_T$  at  $k$  by  $\pi(\hat{T})$  and inserting  $\$$  and  $\pi(\hat{T})$  into the  $\hat{k}$ -th position of  $L_T$  and  $F_T$ , respectively. In Subsection 3.1, we propose our algorithm to compute  $\hat{k}$ . For updating  $LCP^\infty$ , we have to compute the  $\text{lcp}^\infty$ -values for  $\langle \hat{T} \rangle$  with its lexicographically adjacent p-encoded suffixes, which will be treated in Subsection 3.2.

#### 3.1 How to compute $\hat{k}$

Unlike previous work [19] that computes  $\hat{k}$  by counting the number of p-encoded suffixes that are lexicographically smaller than  $\langle \hat{T} \rangle$ , we get  $\hat{k}$  indirectly by computing the rank of a lexicographically closest (smaller or larger) p-encoded suffix to  $\langle \hat{T} \rangle$ . The lexicographically smaller (resp. larger) closest element in  $\{\langle T[i..] \rangle \mid 1 \leq i \leq |T|\}$  to  $\langle \hat{T} \rangle$  is called the *p-pred* (resp. *p-succ*) of  $\langle \hat{T} \rangle$ . If the lexicographic rank of the p-pred (resp. p-succ) of  $\langle \hat{T} \rangle$  is  $k_-$  (resp.  $k_+$ ), then it holds that  $\hat{k} = k_+ = k_- + 1$ .

We start with the easy case that the prepended symbol  $c$  is an s-symbol.

► **Lemma 9.** *Let  $\hat{T} = cT$  be a p-string with  $c \in \Sigma_s$ . If  $p := \text{FPQ}_c(L_T, k)$  exists, the rank  $k_-$  of the p-pred of  $\hat{T}$  is  $\text{LF}_T(p)$ . Otherwise,  $k_- = \text{select}_b(F_T, \text{rank}_b(F_T, |T|))$ , where  $b$  is the largest s-symbol that appears in  $T$  and is smaller than  $c$ .*

**Proof.** By Case (A2) of Table 1, the lexicographic order of p-encoded suffixes starting with  $c$  does not change by removing their first characters, which are all  $c$ . If  $p$  exists,  $\langle T[R_T^{-1}(p)..] \rangle$  is the lexicographically smaller closest p-encoded suffix to  $\langle T \rangle$  that is preceded by  $c$ . Hence,  $\langle T[R_T^{-1}(\text{LF}_T(p))..] \rangle = \langle c(T[R_T^{-1}(p)..]) \rangle$  is the p-pred of  $\langle cT \rangle = \langle \hat{T} \rangle$ , which means that  $k_- = \text{LF}_T(p)$ .

If  $p$  does not exist, it implies that  $\langle \hat{T} \rangle$  is the lexicographically smallest p-encoded suffix that starts with  $c$ . Since  $\langle \hat{T} \rangle$  lexicographically comes right after the p-encoded suffixes starting with an s-symbol smaller than  $c$ ,  $k_-$  is the last occurrence of  $b$  in  $F_T$ , that is,  $k_- = \text{select}_b(F_T, \text{rank}_b(F_T, |T|))$ . ◀

In the rest of this subsection, we consider the case that  $c$  is a p-symbol. If  $T$  contains no p-symbol, it is clear that  $k_- = |T|$ . Hence, in what follows, we assume that there is a p-symbol in  $T$ .

Since  $\langle \hat{T} \rangle$  has the longest  $\text{lcp}$ -value with its p-pred or p-succ among all the suffixes of  $T$ , we search for such p-encoded suffixes of  $T$  using the following lemmas to leverage the information stored in  $LCP_T^\infty$ .

► **Lemma 10.** *Given two positions  $i$  and  $j$  with  $1 \leq i < j \leq |T|$ ,*

$$\text{lcp}^\infty(\langle T[R_T^{-1}(i)..] \rangle, \langle T[R_T^{-1}(j)..] \rangle) = \text{RmQ}_{LCP_T^\infty}(i + 1, j).$$



■ **Algorithm 1** Algorithm to compute the maximal interval  $[l..r]$  such that  $\text{lcp}^\infty(\langle T[\mathbf{R}_T^{-1}(i)..] \rangle, \langle T[\mathbf{R}_T^{-1}(j)..] \rangle) \geq e$  for any  $j \in [l..r]$ . It returns  $[i..i]$  if  $e > |T[\mathbf{R}_T^{-1}(i)..]|_p$ .

---

```

1 Function GetMI( $i, e$ ):
2    $l \leftarrow \max\{1, \text{FPQ}_{<e}(\text{LCP}_T^\infty, i)\};$ 
3    $r \leftarrow \min\{|T|, \text{FNQ}_{<e}(\text{LCP}_T^\infty, i + 1) - 1\};$ 
4   return  $[l..r]$ ;

```

---

**Proof.** By Lemma 1 of [22],  $\text{lcp}(x, z) = \min\{\text{lcp}(x, y), \text{lcp}(y, z)\}$  for any strings  $x < y < z$ , and thus,  $\text{lcp}^\infty(x, z) = \min\{\text{lcp}^\infty(x, y), \text{lcp}^\infty(y, z)\}$ . Since  $\text{LCP}_T^\infty$  holds the  $\text{lcp}^\infty$ -values of lexicographically adjacent p-encoded suffixes, we get  $\text{lcp}^\infty(\langle T[\mathbf{R}_T^{-1}(i)..] \rangle, \langle T[\mathbf{R}_T^{-1}(j)..] \rangle) = \min\{\text{LCP}_T^\infty[g]\}_{g=i+1}^j = \text{RmQ}_{\text{LCP}_T^\infty}(i+1, j)$  by applying the previous argument successively. ◀

► **Lemma 11.** For given  $i, e \in [1..n]$ , if  $e \leq |T[\mathbf{R}_T^{-1}(i)..]|_p$ , then Algorithm 1 computes the maximal interval  $[l..r]$  such that  $\text{lcp}^\infty(\langle T[\mathbf{R}_T^{-1}(i)..] \rangle, \langle T[\mathbf{R}_T^{-1}(j)..] \rangle) \geq e$  for any  $j \in [l..r]$ . If  $e > |T[\mathbf{R}_T^{-1}(i)..]|_p$ , then Algorithm 1 returns  $[i..i]$ .

► **Lemma 12.** Algorithm 2 correctly returns  $\hat{k}$ .

**Proof.**

**Outline.** Let  $h_i = \text{select}_\infty(\langle T \rangle, i)$  for any  $1 \leq i \leq \min\{|T|_p, \pi(\hat{T})\}$ , and  $h_i = |T| + 1$  for any  $i > \min\{|T|_p, \pi(\hat{T})\}$ . Also let  $\lambda = \max\{\text{lcp}(\langle \hat{T} \rangle, \langle T[i..] \rangle) \mid 1 \leq i \leq |T|\}$ . Although Algorithm 2 does not intend to compute the exact value of  $\lambda$ , it checks if  $\lambda$  falls in  $[h_e..h_{e+1}]$  in decreasing order of  $e$  starting from  $\min\{\pi(\hat{T}), \max\{\text{LCP}_T^\infty[k], \text{LCP}_T^\infty[k+1]\}\}$ . One of the necessary conditions to have  $\text{lcp}(\langle \hat{T} \rangle, \langle T[i..] \rangle) > h_e$  is that  $\text{lcp}(\langle T \rangle, \langle T[i+1..] \rangle) \geq h_e$ , or equivalently  $\text{lcp}^\infty(\langle T \rangle, \langle T[i+1..] \rangle) \geq e$ . Line 2 computes the maximal interval  $[l..r]$  that represents the ranks of the p-encoded suffixes having an  $\text{lcp}^\infty$ -value larger than or equal to  $e$ . The basic idea is to find a p-encoded suffix in  $\{\langle T[\mathbf{R}_T^{-1}(p)..] \rangle\}_{p=l}^r$  that comes closest to  $\langle \hat{T} \rangle$  when extended by adding its preceding symbol. Here let us call  $\langle T[\mathbf{R}_T^{-1}(p) - 1..] \rangle$  the *extended suffix* of  $\langle T[\mathbf{R}_T^{-1}(p)..] \rangle$ . When Algorithm 2 decreases  $e$  to the value with  $\lambda \in [h_e + 1..h_{e+1}]$ ,  $\hat{k}$  is returned in one of the if-then-blocks at Lines 4, 5, 8 and 13.

If  $\text{lcp}(\langle \hat{T} \rangle, \langle T[i..] \rangle) = h_{\hat{e}}$  for an integer  $\hat{e}$ , there are two possible scenarios (see Figure 2 for an illustration):

(H1)  $\text{lcp}^\infty(\langle T \rangle, \langle T[i+1..] \rangle) \geq \hat{e}$  and either  $\pi(\hat{T}) > \pi(T[i..]) = \hat{e}$  or  $\pi(T[i..]) > \pi(\hat{T}) = \hat{e}$ , and  
(H2)  $\text{lcp}(\langle T \rangle, \langle T[i+1..] \rangle) = h_{\hat{e}} - 1$  and both  $\pi(\hat{T})$  and  $\pi(T[i..])$  are at least  $\hat{e}$ .

Case (H1) is processed in one of the if-then-blocks at Lines 6 and 18 when  $e = \hat{e}$ . while Case (H2) at Lines 4, 5, 8 and 13 when  $e = \hat{e} - 1$ . Note that p-encoded suffix of Case (H1) is never farther from  $\langle \hat{T} \rangle$  than that of Case (H2) because the lexicographic order between  $\langle \hat{T} \rangle$  and  $\langle T[i..] \rangle$  is determined by  $\infty$  and  $h_{\hat{e}}$  at  $h_{\hat{e}} + 1$  in Case (H1), while it is by  $\infty$  and something smaller than  $h_{\hat{e}}$  in Case (H2). Since Algorithm 2 processes Case (H1) first, it guarantees that the algorithm finds the closer one first.

In what follows, we delve into the details of each code block.

**If-then-block at Line 3.** The case with  $e = \pi(\hat{T})$  is treated differently than other cases in the if-then-block at Line 3 since  $h_{\pi(\hat{T})}$  is the unique position where  $\langle T \rangle[h_{\pi(\hat{T})}] = \infty$  turns into  $\langle \hat{T} \rangle[h_{\pi(\hat{T})} + 1] = h_{\pi(\hat{T})}$ . For a p-encoded suffix  $\langle T[\mathbf{R}_T^{-1}(q')..] \rangle \in \{\langle T[\mathbf{R}_T^{-1}(p)..] \rangle\}_{p=l}^r$ , having  $\text{L}_T[q'] = \pi(\hat{T})$  is necessary and sufficient for its extended suffix  $\langle T[\mathbf{R}_T^{-1}(q') - 1..] \rangle$  to have an  $\text{lcp}$ -value larger than  $h_{\pi(\hat{T})}$  with  $\langle \hat{T} \rangle$ . By Corollary 6, p-encoded suffixes satisfying this condition must preserve their lexicographic order after extension, and hence, it is enough to search for the closest one ( $q \leftarrow \text{FPQ}_e(\text{L}_T, k)$  or  $q \leftarrow \text{FNQ}_e(\text{L}_T, k)$ ) to  $\langle T \rangle$  and compute the rank of its

extended suffix by  $\text{LF}_T(q)$ . If Lines 4 and 5 do not return a value, we know that  $\lambda \leq h_{\pi(\hat{T})}$ . The if-block at Line 6 checks if there exists a p-encoded suffix  $\langle T[i+1..] \rangle$  that satisfies the condition of Case (H1) to be  $\text{lcp}(\langle \hat{T} \rangle, \langle T[i..] \rangle) = h_{\pi(\hat{T})}$ . It is enough to find one  $\langle T[\text{R}_T^{-1}(q)..] \rangle$  with  $\text{L}_T[q] > \pi(\hat{T})$  because it is necessary and sufficient to have  $\text{lcp}(\langle \hat{T} \rangle, \langle T[\text{R}_T^{-1}(q) - 1..] \rangle) = h_{\pi(\hat{T})}$  and  $\langle \hat{T} \rangle[h_{\pi(\hat{T})} + 1] = \infty \neq h_{\pi(\hat{T})} = \langle T[\text{R}_T^{-1}(q) - 1..] \rangle[h_{\pi(\hat{T})} + 1]$ . Note that there could be two or more p-encoded suffixes that satisfy the condition and their lexicographic order may change by extension. In the then-block at Line 6, the algorithm computes the rank of the lexicographically smallest p-encoded suffix that has an  $\text{lcp}^\infty$ -value larger than  $\pi(\hat{T})$  with  $\langle T[\text{R}_T^{-1}(q) - 1..] \rangle = \langle T[\text{R}_T^{-1}(\text{LF}_T(q))..] \rangle$ , which is the p-succ of  $\langle \hat{T} \rangle$  in this case.

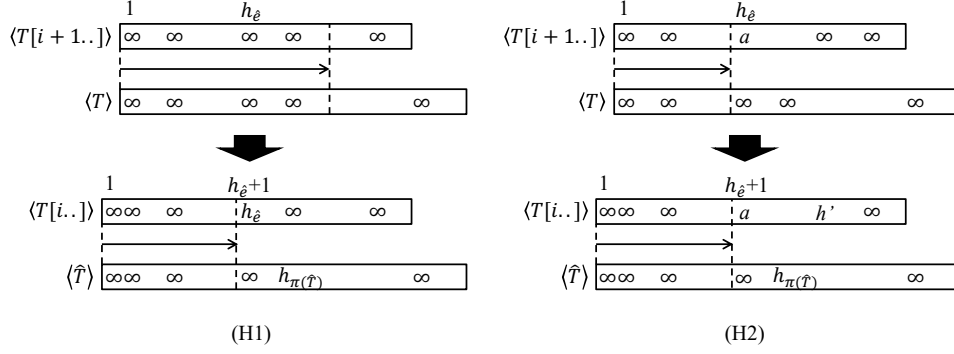
**Precondition to enter Line 7.** The case with  $e \neq \pi(\hat{T})$  is processed in the else-block at Line 7. Here, it is good to keep in mind that when we enter this else-block,  $\text{lcp}^\infty(\langle T \rangle, \langle T[i..] \rangle) \leq e$  or  $\pi(T[i - 1..]) \leq e$  holds for any proper suffix  $T[i..]$  of  $T$ , since otherwise  $\hat{k}$  must be reported in a previous round of the foreach loop.

**If-then-block at Line 8.** When the if-condition at Line 8 holds,  $\langle T[\text{R}_T^{-1}(q)..] \rangle$  is the lexicographically smaller closest p-encoded suffix to  $\langle T \rangle$  such that  $\text{lcp}^\infty(\langle \hat{T} \rangle, \langle T[\text{R}_T^{-1}(q) - 1..] \rangle) \geq e + 1$ , or equivalently  $\text{lcp}(\langle \hat{T} \rangle, \langle T[\text{R}_T^{-1}(q) - 1..] \rangle) > h_e$ . Note that  $\langle T[\text{R}_T^{-1}(q) - 1..] \rangle < \langle \hat{T} \rangle$  must hold, since otherwise,  $T[\text{R}_T^{-1}(q) - 1..]$  and  $\langle \hat{T} \rangle$  would fall into Case (B3) with  $\text{lcp}^\infty(\langle T[\text{R}_T^{-1}(q) - 1..] \rangle, \langle \hat{T} \rangle) = \pi(\hat{T})$ , and  $\hat{k}$  should be reported at Line 6 in a previous round. For any p-encoded suffix in  $\{\langle T[\text{R}_T^{-1}(p)..] \rangle\}_{p=q+1}^{k-1}$  its extended suffix is lexicographically smaller than  $\langle T[\text{R}_T^{-1}(q) - 1..] \rangle$  due to Corollary 7, and never closer to  $\langle \hat{T} \rangle$  than  $\langle T[\text{R}_T^{-1}(q) - 1..] \rangle$ . If  $|T[\text{R}_T^{-1}(q)..]_p| \geq e + 1$ , the interval  $[l'..r']$  computed at Line 9 is the maximal interval such that every p-encoded suffix in  $\{\langle T[\text{R}_T^{-1}(p)..] \rangle\}_{p=l'}^{r'}$  shares the common prefix of length  $h' := \text{select}_\infty(\langle T[\text{R}_T^{-1}(q)..] \rangle, e + 1)$  with  $\langle T[\text{R}_T^{-1}(q)..] \rangle$ . In the case with  $|T[\text{R}_T^{-1}(q)..]_p| = e$ ,  $\text{GetMI}(q, e + 1)$  returns  $[q..q]$  and let us define  $h'$  to be  $|T[\text{R}_T^{-1}(q)..]_p|$ .

Since any  $\langle T[i..] \rangle \in \{\langle T[\text{R}_T^{-1}(p)..] \rangle\}_{p=1}^{l'-1}$  has an lcp-value smaller than  $h'$  with  $\langle T[\text{R}_T^{-1}(q)..] \rangle$ , it follows from Table 1 that  $\langle T[i - 1..] \rangle < \langle T[\text{R}_T^{-1}(q) - 1..] \rangle$ . Also, for any  $\langle T[i..] \rangle \in \{\langle T[\text{R}_T^{-1}(p)..] \rangle\}_{p=k+1}^{l'}$ , the aforementioned precondition to enter the else-block at Line 7 implies that  $\langle T[\text{R}_T^{-1}(q) - 1..] \rangle < \langle T[i - 1..] \rangle < \langle \hat{T} \rangle$  cannot happen: If  $\text{lcp}^\infty(\langle T \rangle, \langle T[i..] \rangle) < e$  or  $\pi(T[i - 1..]) \leq e$ , then  $\text{lcp}(\langle \hat{T} \rangle, \langle T[i - 1..] \rangle) \leq h_e < \text{lcp}(\langle \hat{T} \rangle, \langle T[\text{R}_T^{-1}(q) - 1..] \rangle)$  leads to the conclusion. For the remaining case with  $\text{lcp}^\infty(\langle T \rangle, \langle T[i..] \rangle) = e$  and  $\pi(T[i - 1..]) > e$ , it holds that  $\langle \hat{T} \rangle < \langle T[i - 1..] \rangle$  due to Case (B4) of Table 1.

In the previous paragraph we have confirmed that the lcp-value between  $\langle \hat{T} \rangle$  and its p-pred is at most  $h'$ , which implies that the p-pred is the largest p-encoded suffix that is prefixed by  $x := \langle T[\text{R}_T^{-1}(q) - 1..] \rangle[h']$ . If  $q' \leftarrow \text{FPQ}_{\geq e+2}(\text{L}_T, r')$  computed at Line 10 is in  $[l'..r']$ ,  $\langle T[\text{R}_T^{-1}(q') - 1..] \rangle = \langle T[\text{LF}_T(q')..] \rangle$  is prefixed by  $x \cdot \infty$  and the p-pred is the largest p-encoded suffix that is prefixed by  $x \cdot \infty$ , which can be computed by  $\max \text{GetMI}(\text{LF}_T(q'), e + 2)$  because  $\langle T[\text{R}_T^{-1}(q') - 1..] \rangle[h' + 1] = \langle T[\text{R}_T^{-1}(\text{LF}_T(q')..] \rangle[h' + 1] = x \cdot \infty$  contains exactly  $e + 2$   $\infty$ 's. If  $q' \notin [l'..r']$ , the p-pred is the largest p-encoded suffix that is prefixed by  $x \cdot h'$  (or  $x \cdot \$$  for the case with  $|T[\text{R}_T^{-1}(q)..]_p| = e$ ), which is  $\langle T[\text{R}_T^{-1}(\text{LF}_T(q))..] \rangle$ .

**If-then-block at Line 13.** When the if-condition at Line 13 holds,  $\langle T[\text{R}_T^{-1}(q)..] \rangle$  is the lexicographically larger closest p-encoded suffix to  $\langle T \rangle$  such that  $\text{lcp}^\infty(\langle \hat{T} \rangle, \langle T[\text{R}_T^{-1}(q) - 1..] \rangle) \geq e + 1$ , or equivalently  $\text{lcp}(\langle \hat{T} \rangle, \langle T[\text{R}_T^{-1}(q) - 1..] \rangle) > h_e$ . Note that  $\langle \hat{T} \rangle < \langle T[\text{R}_T^{-1}(q) - 1..] \rangle$  must hold, since otherwise,  $\langle \hat{T} \rangle$  and  $T[\text{R}_T^{-1}(q) - 1..]$  would fall into Case (B3) with  $\text{lcp}(\langle \hat{T} \rangle, \langle T[\text{R}_T^{-1}(q) - 1..] \rangle) = h_{\hat{e}}$  for some  $\hat{e} > e$ , and  $\hat{k}$  should be reported at Line 18 of a previous round. For any p-encoded suffix in  $\{\langle T[\text{R}_T^{-1}(p)..] \rangle\}_{p=k+1}^{q-1}$  its extended suffix is lexicographically smaller than  $\langle \hat{T} \rangle$  due to Corollary 7, and never come lexicographically between  $\langle \hat{T} \rangle$  and  $\langle T[\text{R}_T^{-1}(q) - 1..] \rangle$ . If  $|T[\text{R}_T^{-1}(q)..]_p| \geq e + 1$ , the interval  $[l'..r']$  computed



■ **Figure 2** Illustration for Cases (H1) and (H2) in which  $\text{lcp}(\langle \hat{T} \rangle, \langle T[i..] \rangle) = h_{\hat{e}}$  for an integer  $\hat{e}$ . The left part shows one of the situations for Case (H1) where  $\text{lcp}^\infty(\langle T \rangle, \langle T[i+1..] \rangle) \geq \hat{e}$  and  $\pi(\hat{T}) > \pi(T[i..]) = \hat{e}$ . The right part shows one of the situations for Case (H2) where  $\text{lcp}(\langle T \rangle, \langle T[i+1..] \rangle) = h_{\hat{e}} - 1$ ,  $\pi(\hat{T}) > h_{\hat{e}}$  and  $\pi(T[i..]) > h_{\hat{e}}$ .

at Line 14 is the maximal interval such that every p-encoded suffix in  $\{\langle T[\mathbb{R}_T^{-1}(p)..] \rangle\}_{p=l'}$  shares the common prefix of length  $h' := \text{select}_\infty(\langle T[\mathbb{R}_T^{-1}(q)..] \rangle, e+1)$  with  $\langle T[\mathbb{R}_T^{-1}(q)..] \rangle$ . In the case with  $|T[\mathbb{R}_T^{-1}(q)..]_p| = e$ ,  $\text{GetMI}(q, e+1)$  returns  $[q..q]$  and let us define  $h'$  to be  $|T[\mathbb{R}_T^{-1}(q)..]|$ .

For any  $\langle T[i..] \rangle \in \{\langle T[\mathbb{R}_T^{-1}(p)..] \rangle\}_{p=1}^{k-1}$ , the aforementioned precondition to enter the else-block at Line 7 implies that  $\langle T[i-1..] \rangle < \langle \hat{T} \rangle$  because Case (B3) of Table 1 cannot hold under the condition of  $\text{lcp}^\infty(\langle T \rangle, \langle T[i..] \rangle) \leq e$  or  $\pi(T[i-1..]) \leq e$ . For any  $\langle T[i..] \rangle \in \{\langle T[\mathbb{R}_T^{-1}(p)..] \rangle\}_{p=r'+1}^{|T|}$ , we show that  $\langle \hat{T} \rangle < \langle T[i-1..] \rangle < \langle T[\mathbb{R}_T^{-1}(q)-1..] \rangle$  cannot happen: Note that  $\text{lcp}^\infty(\langle T[\mathbb{R}_T^{-1}(q)..] \rangle, \langle T[i..] \rangle) \leq e$  by definition, and  $\text{lcp}^\infty(\langle T[\mathbb{R}_T^{-1}(q)..] \rangle, \langle T[i..] \rangle) < e$  implies  $\text{lcp}^\infty(\langle T \rangle, \langle T[i..] \rangle) = \text{lcp}^\infty(\langle T[\mathbb{R}_T^{-1}(q)..] \rangle, \langle T[i..] \rangle)$ . If  $\text{lcp}^\infty(\langle T[i..] \rangle, \langle T[\mathbb{R}_T^{-1}(q)..] \rangle) < e$  or  $\pi(T[i-1..]) \leq e$ ,  $\text{lcp}(\langle \hat{T} \rangle, \langle T[i-1..] \rangle) \leq h_e < \text{lcp}(\langle \hat{T} \rangle, \langle T[\mathbb{R}_T^{-1}(q)-1..] \rangle)$  leads to the conclusion. For the remaining case with  $\text{lcp}^\infty(\langle T[i..] \rangle, \langle T[\mathbb{R}_T^{-1}(q)..] \rangle) = e$  and  $\pi(T[i-1..]) > e$ , it holds that  $\langle T[\mathbb{R}_T^{-1}(q)-1..] \rangle < \langle T[i-1..] \rangle$  due to Case (B4) of Table 1.

In the previous paragraph we have confirmed that the lcp-value between  $\langle \hat{T} \rangle$  and its p-succ is at most  $h'$ , which implies that the p-succ is the smallest p-encoded suffix that is prefixed by  $x := \langle T[\mathbb{R}_T^{-1}(q)-1..] \rangle[.h']$ . If  $q' \leftarrow \text{FNQ}_{e+1}(\mathbb{L}_T, l')$  computed at Line 15 is in  $[l'..r']$ , the p-succ is the smallest p-encoded suffix that is prefixed by  $x \cdot h'$  (or  $x \cdot \$$  for the case with  $|T[\mathbb{R}_T^{-1}(q)..]_p| = e$ ), which is  $\langle T[\mathbb{R}_T^{-1}(\mathbb{L}_T(q'))..] \rangle$ . If  $q' \notin [l'..r']$ , the p-succ is the smallest p-encoded suffix that is prefixed by  $x \cdot \infty$ , which can be computed by  $\min \text{GetMI}(\mathbb{L}_T(q), e+2)$  because  $\langle T[\mathbb{R}_T^{-1}(q)-1..] \rangle[.h'+1] = \langle T[\mathbb{R}_T^{-1}(\mathbb{L}_T(q))..] \rangle[.h'+1] = x \cdot \infty$  contains exactly  $e+2$   $\infty$ 's.

**If-then-block at Line 18.** When we enter the if-then-block at Line 18, it is guaranteed that  $\lambda \leq h_e$ . In order to check if there exists a p-encoded suffix  $\langle T[i+1..] \rangle$  that satisfies the condition of Case (H1) to be  $\text{lcp}(\langle \hat{T} \rangle, \langle T[i..] \rangle) = h_e$ , the algorithm computes  $q \leftarrow \text{FPQ}_e(\mathbb{L}_T, r)$ . If  $q \in [l..r]$ ,  $\langle T[\mathbb{R}_T^{-1}(q)..] \rangle$  is the lexicographically largest p-encoded suffix that satisfies the condition, and by Corollary 6, its extended suffix  $\langle T[\mathbb{R}_T^{-1}(q)-1..] \rangle$  must be the largest one to have  $\text{lcp}(\langle \hat{T} \rangle, \langle T[\mathbb{R}_T^{-1}(q)-1..] \rangle) = h_e$ . Therefore,  $\langle T[\mathbb{R}_T^{-1}(q)-1..] \rangle$  is the p-pred of  $\langle \hat{T} \rangle$ , and  $\hat{k} = 1 + \mathbb{L}_T(q)$ . ◀

■ **Algorithm 2** Algorithm to compute  $\hat{k}$ .

---

```

1 foreach  $e \leftarrow \min\{\pi(\hat{T}), \max\{\text{LCP}_T^\infty[k], \text{LCP}_T^\infty[k+1]\}\}$  down to 1 do
2    $[l..r] \leftarrow \text{GetMI}(k, e)$ ;
3   if  $e = \pi(\hat{T})$  then
4     if  $(q \leftarrow \text{FPQ}_e(\text{L}_T, k)) \in [l..r]$  then return  $1 + \text{LF}_T(q)$  ;
5     if  $(q \leftarrow \text{FNQ}_e(\text{L}_T, k)) \in [l..r]$  then return  $\text{LF}_T(q)$  ;
6     if  $(q \leftarrow \text{FNQ}_{\geq e+1}(\text{L}_T, l)) \in [l..r]$  then return  $\min \text{GetMI}(\text{LF}_T(q), e+1)$  ;
7   else
8     if  $(q \leftarrow \text{FPQ}_{\geq e+1}(\text{L}_T, k)) \in [l..r]$  then
9        $[l'..r'] \leftarrow \text{GetMI}(q, e+1)$ ;
10       $q' \leftarrow \text{FPQ}_{\geq e+2}(\text{L}_T, r')$ ;
11      if  $q' \in [l'..r']$  then return  $1 + \max \text{GetMI}(\text{LF}_T(q'), e+2)$  ;
12      else return  $1 + \text{LF}_T(q)$  ;
13     if  $(q \leftarrow \text{FNQ}_{\geq e+1}(\text{L}_T, k)) \in [l..r]$  then
14        $[l'..r'] \leftarrow \text{GetMI}(q, e+1)$ ;
15        $q' \leftarrow \text{FNQ}_{e+1}(\text{L}_T, l')$ ;
16       if  $q' \in [l'..r']$  then return  $\text{LF}_T(q')$  ;
17       else return  $\min \text{GetMI}(\text{LF}_T(q), e+2)$  ;
18     if  $(q \leftarrow \text{FPQ}_e(\text{L}_T, r)) \in [l..r]$  then return  $1 + \text{LF}_T(q)$  ;

```

---

### 3.2 How to maintain $\text{LCP}_T^\infty$

Suppose that we have  $k = R_T(1)$ ,  $\hat{k} = R_{\hat{T}}(1)$ ,  $\text{L}_T$ ,  $\text{F}_T$ . We show how to compute the  $\text{lcp}^\infty$ -values of  $\langle \hat{T} \rangle$  with its p-pred  $\langle T[R_T^{-1}(\hat{k}) - 1..] \rangle$  and p-succ  $\langle T[R_T^{-1}(\hat{k})..] \rangle$  to maintain  $\text{LCP}_T^\infty$ .

We focus on  $\text{lcp}^\infty(\langle \hat{T} \rangle, \langle T[R_T^{-1}(\hat{k})..] \rangle)$  because the other one can be treated similarly. We apply Table 1 by setting  $x = \hat{T}$  and  $y = T[R_T^{-1}(\hat{k})..]$  if  $k < \text{FL}_T(\hat{k})$  (otherwise we swap their roles for  $x$  and  $y$ ). In order to get  $\text{lcp}^\infty(\langle x \rangle, \langle y \rangle)$ , all we need are  $\pi(x) = \pi(\hat{T})$ ,  $\pi(y) = \text{F}[\hat{k}]$  and  $e = \text{lcp}^\infty(\langle x[2..] \rangle, \langle y[2..] \rangle)$ . For the computation of  $e$  we use Lemma 10, i.e.,  $e = \text{lcp}^\infty(\langle x[2..] \rangle, \langle y[2..] \rangle) = \text{RmQLCP}_T^\infty(k+1, \text{FL}_T(\hat{k}))$ .

### 3.3 Dynamic data structures and analysis

Let  $\sigma_s$  and respectively  $\sigma_p := |T|_p$  be the numbers of distinct s-symbols and p-symbols that appear in  $T$  and  $\sigma = \sigma_s + \sigma_p$ . We consider constructing  $\text{F}_T$ ,  $\text{L}_T$  and  $\text{LCP}_T^\infty$  online for a p-string  $T$  of length  $n$  over an alphabet  $(\Sigma_s \cup \Sigma_p)$  of size  $n^{O(1)}$ . To this end, we introduce data structures for implementing our algorithm, which we presented in the previous subsections.

To obtain our claimed space bounds of  $O(n \lg \sigma)$  bits, we maintain a naming function that maps the set of distinct s-symbols indexed in the pBWT from  $\Sigma_s$  to the range of ranks  $[1..\sigma_s]$ . By doing so, we can represent and store each s-symbol in the pBWT by its rank. Thus, each s-symbol in  $\text{F}_T$  and  $\text{L}_T$  consumes  $O(\lg \sigma_s)$  bits instead of  $O(\lg n)$  bits.

In the following we present two alternative implementations for the naming function. The first one imposes a new order on the s-symbols such that the computed  $\text{F}_T$  and  $\text{L}_T$  arrays may arrange s-symbols differently than the standard pBWT built on the plain s-symbols. The second one keeps the order of the s-symbols, but needs an additional scan of the input text  $T$  to determine the order *prior to* the computation of the pBWT.

1. Our first approach updates the naming function in the same online fashion as computing the pBWT. To this end, we represent the naming function by a dynamic associative array using  $O(\sigma_s \lg n) = O(n \lg \sigma_s)$  bits of space and taking  $O(\frac{\lg \sigma_s}{\lg \lg \sigma_s})$  operation time like [21]. Setting initially  $\sigma_s = 0$ , we update  $\sigma_s$  and the naming function whenever we read a new s-symbol. In detail, when we read a new s-symbol with integer representation  $x$  with a yet-undefined rank (meaning it is not yet stored in the associative array), we increment  $\sigma_s$  by one, and insert the integer key  $x$  associated with the value  $\sigma_s$  into the associative array. The assignment of the ranks is done in a *first come first served* order, meaning that the order of two s-symbols is determined by whose rightmost occurrence in the text has the larger text position (since we build the pBWT online reading symbols from the right end). Changing the alphabet order neither invalidates the LF-mapping nor the backward search.<sup>3</sup> However, there is a subtle caveat in that this approach needs to know  $\sigma_s$  in advance (or at least a close upper bound  $\sigma'_s$  with  $\sigma'_s = c\sigma$  for a constant  $c \geq 1$ ). Otherwise, we need to spend some extra time on reconstructing all dynamic data structures working with s-symbols. That is because, for steady increases of  $\sigma_s$ , there is point where we no longer can represent a s-symbol rank in just  $\lceil \lg \sigma_s \rceil$  bits, but need  $1 + \lceil \lg \sigma_s \rceil$  bits instead. Instead of rebuilding all data structures storing s-symbol ranks on every increase of  $\lceil \lg \sigma_s \rceil$ , we initially accommodate each s-symbol with  $2 \lceil \lg \sigma_s \rceil$  bits, and double this space whenever necessary.<sup>4</sup> By doing so, the number of bits per s-symbol increases from constant to  $\Theta(\lg \sigma_s)$  exponentially, and thus the number of total rebuilding steps is bounded by  $O(\lg \lg \sigma_s)$ , where  $\sigma_s$  denotes the final number of distinct s-symbols indexed by the pBWT. Thus the final construction time stated in Theorem 1 becomes  $O(n \frac{(\lg \sigma_p + \lg \lg \sigma_s) \lg n}{\lg \lg n})$ , based on the fact that querying or updating the dynamic data structures representing  $F_T$  and  $L_T$  needs  $O(\frac{\lg n}{\lg \lg n})$  time per entry, as we will later see in Lemma 13.

For computing  $b$  in Lemma 9 for a given s-symbol  $c$ , we proceed as follows. Given  $c$  has been assigned the rank  $r$ , then  $b = r - 1$ . Otherwise,  $c$  has not been ranked, and thus  $b = \sigma_s$  since  $c$  will receive a rank larger than all other s-symbols indexed in the pBWT.

2. However, if this imposed order is not desirable in some setting, it is possible to assign ranks reflecting the initial order of  $\Sigma_s$ . For that, we note that the aforementioned implementation [21] also supports a sorted traversal of the keys. Thus, it is sufficient to (a) build this associative array while scanning the entire text  $T$ , (b) reassign each key a new rank determined by a sorted traversal of the associative array, and finally (c) start the pBWT computation. This, of course, needs to read  $T$  twice instead of once.

Unfortunately, since we keep the initial alphabet order of the s-symbols, determining  $b$  in Lemma 9 becomes nontrivial. For computing the value of  $b$ , we maintain the set of s-symbols used in the currently computed pBWT by a dynamic fusion tree [34] taking  $O(\sigma_s \lg n) = O(n \lg \sigma_s)$  bits. The fusion tree allows us then to compute  $b$  in  $O(\frac{\lg \sigma_s}{\lg \lg \sigma_s})$  time.

Next, we maintain  $F_T$  by a dynamic string of Lemma 3 supporting random access, insertion, rank and select queries in  $O(\frac{\lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of space. For  $LCP_T^\infty$  we maintain a dynamic string of Lemma 4 to support random access, insertion, RmQ, FPQ and FNQ queries in  $O(\frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma_p)$  bits of space.

<sup>3</sup> Changing the alphabet order for optimizing the compressibility of the BWT is actually an actively researched topic [5].

<sup>4</sup> While this seems like a standard trick for amortizing the costs of dynamic arrays, the amortization argument does not hold here in general because the cost parameter  $\sigma_s$  and the array length  $n$  can be independent. For instance, imagine that we first read  $n/2$  times the same s-symbol from the input, and then start to read only distinct s-symbols.

If we build a dynamic string of Lemma 4 for  $L_T$ , the query time would be  $O(\frac{\lg \sigma \lg n}{\lg \lg n})$ . Since our algorithm does not use RmQ, FPQ and FNQ queries for s-symbols, we can reduce the query time to  $O(\frac{\lg \sigma_p \lg n}{\lg \lg n})$  as follows. We represent  $L_T$  with one level of a wavelet tree, where a bit vector partitions the alphabet into  $\Sigma_s$  and  $[1..|T|_p]$  and thus has pointers to  $X_T$  and  $Y_T$  storing respectively the sequence over  $\Sigma_s$  and that over  $[1..|T|_p]$  of  $L_T$ . We represent the former and the latter by the data structures described in Lemmas 3 and 4, respectively, since we only need the aforementioned queries such as RmQ on  $Y_T$ . Then, queries on  $L_T$  can be answered in  $O(\frac{\lg \sigma_p \lg n}{\lg \lg n})$  time using  $O(n \lg \sigma)$  bits of space.

In addition to these dynamic strings for  $F_T$ ,  $L_T$  and  $LCP_T^\infty$ , we consider maintaining another dynamic string  $Z_T$ , a string that is obtained by extracting the leftmost occurrence of every p-symbol in  $T$ . Note that  $|Z_T| = \sigma_p \leq n$ . A dynamic string  $Z_T$  of Lemma 3 enables us to compute  $\pi(cT)$  for a p-symbol  $c$  by  $\pi(cT) = \min\{\infty, \text{select}_c(Z_T, 1)\}$  in  $O(\frac{\lg \sigma_p}{\lg \lg \sigma_p})$  time and  $O(\sigma_p \lg \sigma_p)$  bits of space.

We are now ready to prove the following lemma.

► **Lemma 13.**  $F_T$ ,  $L_T$  and  $LCP_T^\infty$  for a p-string of length  $n$  over an alphabet  $(\Sigma_s \cup \Sigma_p)$  of size  $n^{O(1)}$  can be constructed online in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of space, where  $\sigma_s$  and respectively  $\sigma_p$  are the numbers of distinct s-symbols and p-symbols used in the p-string and  $\sigma = \sigma_s + \sigma_p$ .

**Proof.** We maintain the dynamic data structures of  $O(n \lg \sigma)$  bits described in this subsection while prepending a symbol to the current p-string. For a single step of updating  $T$  to  $\hat{T} = cT$  with  $c \in (\Sigma_s \cup \Sigma_p)$ , we compute  $\hat{k} = R_{\hat{T}}(1)$  as described in Subsection 3.1 and obtain  $F_{\hat{T}}$  and  $L_{\hat{T}}$  by replacing  $\$$  in  $L_T$  at  $k = R_T(1)$  by  $\pi(\hat{T})$  and inserting  $\$$  and  $\pi(\hat{T})$  into the  $\hat{k}$ -th position of  $L_T$  and  $F_T$ , respectively.  $LCP_T^\infty$  is updated as described in Subsection 3.2.

If  $c \in \Sigma_s$ , the computation of  $\hat{k}$  based on Lemma 9 requires a constant number of queries. If  $c \in \Sigma_p$ , Algorithm 2 computes  $\hat{k}$  invoking  $O(2 + e - \hat{e})$  queries, where  $e = \max\{LCP_T^\infty[k], LCP_T^\infty[k+1]\}$  and  $\hat{e} = \max\{LCP_{\hat{T}}^\infty[\hat{k}], LCP_{\hat{T}}^\infty[\hat{k}+1]\}$ . The value  $e$  can be seen as a potential held by the current string  $T$ , which upper bounds the number of queries. The number of queries in a single step can be  $O(\sigma_p)$  in the worst case when  $e$  and  $\hat{e}$  are close to  $\sigma_p$  and respectively 0, but this will reduce the potential for later steps, which allows us to give an amortized analysis. Since a single step increases the potential at most 1 by Corollary 5, the total number of queries can be bounded by  $O(n)$ .

Since we invoke  $O(n)$  queries that take  $O(\frac{\lg \sigma_p \lg n}{\lg \lg n})$  time each, the overall time complexity is  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$ . ◀

## 4 Extendable compact index for p-matching

In this section, we show that  $L_T$ ,  $F_T$  and  $LCP_T^\infty$  can serve as an index for p-matching.

First we show that we can support backward search, a core procedure of BWT-based indexes, with the data structures for  $L_T$ ,  $F_T$  and  $LCP_T^\infty$  described in Subsection 3.3. For any p-string  $w$ , let  $w$ -interval be the maximal interval  $[l..r]$  such that  $\langle T[R_T^{-1}(p)..] \rangle$  is prefixed by  $\langle w \rangle$  for any  $p \in [l..r]$ . We show the next lemma for a single step of the backward search, which computes  $cw$ -interval from  $w$ -interval.

► **Lemma 14.** Suppose that we have data structures for  $L_T$ ,  $F_T$  and  $LCP_T^\infty$  described in Subsection 3.3. Given  $w$ -interval  $[l..r]$  and  $c \in (\Sigma_s \cup \Sigma_p)$ , we can compute  $cw$ -interval  $[l'..r']$  in  $O(\frac{\lg \sigma_p \lg n}{\lg \lg n})$  time.

**Proof.** We show that we can compute  $cw$ -interval from  $w$ -interval using a constant number of queries supported on  $L_T$ ,  $F_T$  and  $\langle \hat{T} \rangle$ , which takes  $O(\frac{\lg \sigma_p \lg n}{\lg \lg n})$  time each.

- When  $c$  is in  $\Sigma_s$ : A p-encoded suffix  $\langle T[\mathbf{R}_T^{-1}(p) - 1..] \rangle = \langle T[\mathbf{R}_T^{-1}(\mathbf{LF}_T(p))..] \rangle$  is prefixed by  $\langle cw \rangle$  if and only if  $\langle T[\mathbf{R}_T^{-1}(p)..] \rangle$  is prefixed by  $\langle w \rangle$  and  $\mathbf{L}_T[p] = c$ . In other words,  $\mathbf{LF}_T(p) \in [l'..r']$  if and only if  $p \in [l..r]$  and  $\mathbf{L}_T[p] = c$ . Then it holds that  $l' = \mathbf{LF}_T(\mathbf{FNQ}_c(\mathbf{L}_T, l))$  and  $r' = \mathbf{LF}_T(\mathbf{FPQ}_c(\mathbf{L}_T, r))$  due to Corollary 6.
- When  $c$  is a p-symbol that appears in  $w$ : Similar to the previous case,  $\mathbf{LF}_T(p) \in [l'..r']$  if and only if  $p \in [l..r]$  and  $\mathbf{L}_T[p] = \pi(cw)$ . Then it holds that  $l' = \mathbf{LF}_T(\mathbf{FNQ}_{\pi(cw)}(\mathbf{L}_T, l))$  and  $r' = \mathbf{LF}_T(\mathbf{FPQ}_{\pi(cw)}(\mathbf{L}_T, r))$  due to Corollary 6.
- When  $c$  is a p-symbol that does not appear in  $w$ : Let  $e = |w|_p$ . Since  $p \in [l..r]$  and  $\mathbf{L}_T[p] > e$  are necessary and sufficient conditions for  $\mathbf{LF}_T(p)$  to be in  $[l'..r']$ , we can compute  $r' - l' + 1$ , the width of  $[l'..r']$ , by counting the number of positions  $p$  such that  $\mathbf{L}_T[p] > e$  with  $p \in [l..r]$ . This can be done with 2D range counting queries, which can also be supported with the wavelet matrix of Lemma 4. If  $s = \mathbf{FNQ}_{\geq e+1}(\mathbf{L}_T, l)$  is in  $[l..r]$ , it holds that  $r' - l' + 1 \neq 0$  and  $\mathbf{LF}_T(s) \in [l'..r']$ . Note that  $\mathbf{LF}_T(s)$  is not necessarily  $l'$  because p-encoded suffixes  $\langle T[\mathbf{R}_T^{-1}(p)..] \rangle$  with  $\mathbf{L}_T[p] > e$  in  $[l..r]$  do not necessarily preserve the lexicographic order when they are extended by one symbol to the left, making it non-straightforward to identify the position  $l'$ .

To tackle this problem, we consider

$$[l_e..r_e] = \mathbf{GetMI}(s, e) \text{ and } [l'_{e+1}..r'_{e+1}] = \mathbf{GetMI}(\mathbf{LF}_T(s), e + 1),$$

and show that  $l' = l'_{e+1} + x$ , where  $x$  is the number of positions  $p$  in  $[l_e..l-1]$  with  $\mathbf{L}_T[p] > e$ . Observe that  $[l..r] \subseteq [l_e..r_e]$  and  $[l'..r'] \subseteq [l'_{e+1}..r'_{e+1}]$  by definition, and that  $\mathbf{LF}_T(p) \in [l'_{e+1}..r'_{e+1}]$  if and only if  $p \in [l_e..r_e]$  and  $\mathbf{L}_T[p] > e$  (see Figure 3 for an illustration). Also, it holds that  $\langle T[\mathbf{R}_T^{-1}(\mathbf{LF}_T(p))..] \rangle < \langle T[\mathbf{R}_T^{-1}(\mathbf{LF}_T(q))..] \rangle$  for any  $p \in [l_e..l-1]$  and  $q \in [l..r]$  satisfying  $\mathbf{L}_T[p] > e$  and  $\mathbf{L}_T[q] > e$  because  $\mathbf{lcp}^\infty(\langle T[\mathbf{R}_T^{-1}(p)..] \rangle, \langle T[\mathbf{R}_T^{-1}(q)..] \rangle) = e$ , and they fall into Case (B4) of Table 1. Similarly for any  $p \in [l..r]$  and  $q \in [r+1..r_e]$  satisfying  $\mathbf{L}_T[p] > e$  and  $\mathbf{L}_T[q] > e$ , we have  $\langle T[\mathbf{R}_T^{-1}(\mathbf{LF}_T(p))..] \rangle < \langle T[\mathbf{R}_T^{-1}(\mathbf{LF}_T(q))..] \rangle$ . Hence,  $l' = l'_{e+1} + x$  holds.

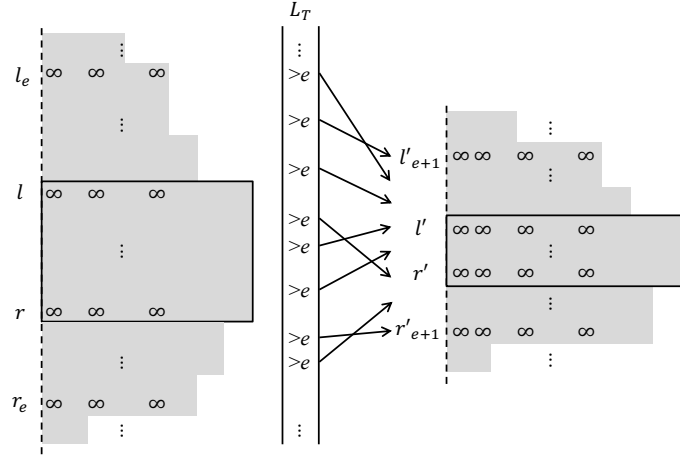
This concludes the proof.  $\blacktriangleleft$

We are now ready to prove Theorem 1, which we restate here:

► **Theorem 1.** *For a p-string  $T$  of length  $n$  over an alphabet  $(\Sigma_s \cup \Sigma_p)$  of size  $n^{O(1)}$ , an index of  $T$  for p-matching can be constructed online in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of space, where  $\sigma_s$  and respectively  $\sigma_p$  are the numbers of distinct s-symbols and p-symbols used in the p-string and  $\sigma = \sigma_s + \sigma_p$ . At any stage of the online construction, it can support the counting queries in  $O(m \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time, where  $m$  is the length of a given pattern for queries. By building an additional data structure of  $O(\frac{n}{\Delta} \lg n)$  bits of space for a chosen parameter  $\Delta \in \{1, 2, \dots, n\}$  the locating queries can be supported in  $O(m \frac{\lg \sigma_p \lg n}{\lg \lg n} + \text{occ} \frac{\Delta \lg n}{\lg \lg n})$  time, where  $\text{occ}$  is the number of occurrences to be reported.*

**Proof of Theorem 1.** If we only need counting queries, Lemmas 13 and 14 are enough: While we build  $\mathbf{L}_T$ ,  $\mathbf{F}_T$  and  $\mathbf{LCP}_T^\infty$  online, we can compute  $w$ -interval  $[l..r]$  for a given pattern  $w$  of length  $m$  using Lemma 14 successively  $m$  times, spending  $O(m \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time in total.

Since  $\{\mathbf{R}_T^{-1}(i) \mid i \in [l..r]\}$  is the set of occurrences of  $w$  in  $T$ , we consider how to access  $\mathbf{R}_T^{-1}(i)$  in  $O(\frac{\Delta \lg n}{\lg \lg n})$  time to support locating queries. As is common in BWT-based indexes, we employ a sampling technique (e.g., see [10]): For every  $\Theta(\Delta)$  text positions we store the values so that if we apply LF/FL-mapping to  $i$  successively at most  $\Theta(\Delta)$  times we hit one of the sampled text positions. A minor remark is that since our online construction proceeds from right to left, it is convenient to start sampling from the right-end of  $T$  and store the distance to the right-end instead of the text position counted from the left-end of  $T$ .



■ **Figure 3** Illustration for the computation of  $cw$ -interval  $[l'..r']$  from  $w$ -interval  $[l..r]$  for the case when  $c$  is a  $p$ -symbol that does not appear in  $w$  with  $e = |w|_p = 3$ . The left (resp. right) part shows sorted  $p$ -encoded suffixes around  $[l..r]$  (resp.  $[l'..r']$ ) with grayed areas representing the longest common prefix between  $w$  (resp.  $cw$ ) and each  $p$ -encoded suffix. The figure illustrates that each position  $p \in [l_e..l-1]$  with  $L_T[p] > e$  is mapped to  $[l'_{e+1}..l'-1]$  by the LF-mapping.

During the online construction of the data structures for  $L_T$ ,  $F_T$  and  $LCP_T^\infty$ , we additionally maintain a dynamic bit vector of length  $n$  and a dynamic integer string  $V_T$  of length  $O(n/\Delta)$ , which marks the sampled positions and stores sampled values, respectively. We implement  $V_T$  with the dynamic string of Lemma 3 in  $O(\frac{n}{\Delta} \lg n)$  bits with  $O(\frac{\lg n}{\lg \lg n})$  query times. In order to support LF/FL-mapping in  $O(\frac{\lg n}{\lg \lg n})$  time, we also maintain  $L_T$  by an instance of the dynamic string of Lemma 3. Using these data structures, we can access  $R_T^{-1}(i)$  in  $O(\frac{\Delta \lg n}{\lg \lg n})$  time as we use LF/FL-mapping at most  $O(\Delta)$  times. This leads to the claimed time bound for locating queries. ◀

## 5 Constructing parameterized suffix arrays in compact space

The parameterized suffix array of a  $p$ -string  $T$  of length  $n$  is the  $n$ -length integer array  $\text{PSA}_T$  with  $\text{PSA}_T[i] = R_T^{-1}(i)$  for any  $1 \leq i \leq n$ . We now prove Theorem 2:

▶ **Theorem 2.** *For a  $p$ -string  $T$  of length  $n$  over an alphabet  $(\Sigma_s \cup \Sigma_p)$  of size  $n^{O(1)}$ , the parameterized suffix array of  $T$  can be constructed in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of space, where  $\sigma_s$  and respectively  $\sigma_p$  are the numbers of distinct  $s$ -symbols and  $p$ -symbols used in the  $p$ -string and  $\sigma = \sigma_s + \sigma_p$ .*

**Proof of Theorem 2.** Using Lemma 13, we can build  $F_T$  and  $L_T$  in  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time and  $O(n \lg \sigma)$  bits of working space, which supports the LF-mapping in  $O(\frac{\lg \sigma_p \lg n}{\lg \lg n})$  time. After reserving  $n \lg n$  bits space for the array  $\text{PSA}_T$ , we fill  $\text{PSA}_T$  in decreasing order of its values starting from  $\text{PSA}_T[1] = n$ . By definition of the LF-mapping, given a position  $i$  with  $\text{PSA}_T[i] = x > 1$ , the position  $i'$  with  $\text{PSA}_T[i'] = x - 1$  can be computed by  $i' = \text{LF}_T(i)$ . Therefore, all values of  $\text{PSA}_T$  can be filled with  $n$  applications of the LF-mappings, leading to  $O(n \frac{\lg \sigma_p \lg n}{\lg \lg n})$  time in total. ◀



## 6 Concluding remarks

We have proposed a construction of a BWT-based compact index for p-matching, which works in compact space and in an online manner. BWT-based indexes have been proposed for other generalized pattern matching like structural pattern matching [17], order preserving matching [15], Cartesian tree matching [23], palindrome pattern matching [30] and circular parameterized pattern matching [33]. Generalized pattern matching listed above has a common feature that their underlying equivalent relations are substring consistent [27]. Since previous work has shown that similar techniques can often be applied to this class of generalized pattern matching, it is of great interest to see if the techniques presented in this paper can also be used for constructing other BWT-based indexes.

---

### References

- 1 Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proc. 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 71–80. ACM, 1993. doi:10.1145/167088.167115.
- 2 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996. doi:10.1006/jcss.1996.0003.
- 3 Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997. doi:10.1137/S0097539793246707.
- 4 Richard Beal and Donald A. Adjeroh. p-suffix sorting as arithmetic coding. *J. Discrete Algorithms*, 16:151–169, 2012. doi:10.1016/j.jda.2012.05.001.
- 5 Jason W. Bentley, Daniel Gibney, and Sharma V. Thankachan. On the complexity of BWT-runs minimization via alphabet reordering. In *Proc. ESA*, volume 173 of *LIPICs*, pages 15:1–15:13, 2020. doi:10.4230/LIPICs.ESA.2020.15.
- 6 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015. doi:10.1016/j.is.2014.06.002.
- 7 Richard Cole and Ramesh Hariharan. Faster suffix tree construction with missing suffix links. *SIAM J. Comput.*, 33(1):26–42, 2003. doi:10.1137/S0097539701424465.
- 8 Satoshi Deguchi, Fumihito Higashijima, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Parameterized suffix arrays for binary strings. In *Proc. Prague Stringology Conference (PSC) 2008*, pages 84–94, 2008. URL: <http://www.stringology.org/event/2008/p08.html>.
- 9 Diptarama, Takashi Katsura, Yuhei Otomo, Kazuyuki Narisawa, and Ayumi Shinohara. Position heaps for parameterized strings. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM) 2017*, volume 78 of *LIPICs*, pages 8:1–8:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.8.
- 10 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS) 2000*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.
- 11 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Right-to-left online construction of parameterized position heaps. In Jan Holub and Jan Zdárek, editors, *Proc. Prague Stringology Conference (PSC) 2018*, pages 91–102. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2018. URL: <http://www.stringology.org/event/2018/p09.html>.
- 12 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Direct linear time construction of parameterized suffix and LCP arrays for constant alphabets. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE) 2019*, volume 11811 of *Lecture Notes in Computer Science*, pages 382–391. Springer, 2019. doi:10.1007/978-3-030-32686-9\_27.

- 13 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. The parameterized position heap of a trie. In Pinar Heggernes, editor, *Proc. 11th International Conference on Algorithms and Complexity (CIAC) 2019*, volume 11485 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2019. doi:10.1007/978-3-030-17402-6\_20.
- 14 Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. The parameterized suffix tray. In Tiziana Calamoneri and Federico Corò, editors, *Proc. 12th International Conference on Algorithms and Complexity (CIAC) 2021*, volume 12701 of *Lecture Notes in Computer Science*, pages 258–270. Springer, 2021. doi:10.1007/978-3-030-75242-2\_18.
- 15 Travis Gagie, Giovanni Manzini, and Rossano Venturini. An encoding for order-preserving matching. In *Proc. 25th Annual European Symposium on Algorithms (ESA) 2017*, pages 38:1–38:15, 2017. doi:10.4230/LIPIcs.ESA.2017.38.
- 16 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pBWT: Achieving succinct data structures for parameterized pattern matching and related problems. In *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) 2017*, pages 397–407, 2017. doi:10.1137/1.9781611974782.25.
- 17 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Structural pattern matching - succinctly. In *Proc. 28th International Symposium on Algorithms and Computation (ISAAC) 2017*, pages 35:1–35:13, 2017. doi:10.4230/LIPIcs.ISAAC.2017.35.
- 18 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. Fully functional parameterized suffix trees in compact space. In Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff, editors, *Proc. 49th International Colloquium on Automata, Languages, and Programming, (ICALP) 2022*, volume 229 of *LIPIcs*, pages 65:1–65:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ICALP.2022.65.
- 19 Daiki Hashimoto, Diptarama Hendrian, Dominik Köppl, Ryo Yoshinaka, and Ayumi Shinohara. Computing the parameterized Burrows-Wheeler transform online. In Diego Arroyuelo and Barbara Poblete, editors, *Proc. 29th International Symposium on String Processing and Information Retrieval (SPIRE) 2022*, volume 13617 of *Lecture Notes in Computer Science*, pages 70–85. Springer, 2022. doi:10.1007/978-3-031-20643-6\_6.
- 20 Tomohiro I, Satoshi Deguchi, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Lightweight parameterized suffix array construction. In *Proc. 20th International Workshop on Combinatorial Algorithms (IWOCA) 2009*, pages 312–323, 2009. doi:10.1007/978-3-642-10217-2\_31.
- 21 Tomohiro I and Dominik Köppl. Load-balancing succinct B trees, 2021. arXiv:2104.08751. doi:10.48550/arXiv.2104.08751.
- 22 Robert W. Irving and Lorna Love. The suffix binary search tree and suffix AVL tree. *J. Discrete Algorithms*, 1(5-6):387–408, 2003. doi:10.1016/S1570-8667(03)00034-0.
- 23 Sung-Hwan Kim and Hwan-Gue Cho. A compact index for cartesian tree matching. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *Proc. 32nd Annual Symposium on Combinatorial Pattern Matching (CPM) 2021*, volume 191 of *LIPIcs*, pages 18:1–18:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.18.
- 24 Sung-Hwan Kim and Hwan-Gue Cho. Simpler FM-index for parameterized string matching. *Inf. Process. Lett.*, 165:106026, 2021. doi:10.1016/j.ipl.2020.106026.
- 25 S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *Proc. 36th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 631–637. IEEE Computer Society, 1995. doi:10.1109/SFCS.1995.492664.
- 26 Taehyung Lee, Joong Chae Na, and Kunsoo Park. On-line construction of parameterized suffix trees for large alphabets. *Inf. Process. Lett.*, 111(5):201–207, 2011. doi:10.1016/j.ipl.2010.11.017.
- 27 Yoshiaki Matsuoka, Takahiro Aoki, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theor. Comput. Sci.*, 656:225–233, 2016. doi:10.1016/j.tcs.2016.02.017.

- 28 Juan Mendivelso, Sharma V. Thankachan, and Yoan J. Pinzón. A brief history of parameterized matching problems. *Discret. Appl. Math.*, 274:103–115, 2020. doi:10.1016/j.dam.2018.07.017.
- 29 J. Ian Munro and Yakov Nekrich. Compressed data structures for dynamic sequences. In *Proc. 23rd Annual European Symposium on Algorithms (ESA) 2015*, pages 891–902, 2015. doi:10.1007/978-3-662-48350-3\_74.
- 30 Shinya Nagashita and Tomohiro I. PalFM-Index: FM-index for palindrome pattern matching. In Laurent Bulteau and Zsuzsanna Lipták, editors, *Proc. 34th Annual Symposium on Combinatorial Pattern Matching (CPM) 2023*, volume 259 of *LIPICs*, pages 23:1–23:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICs.CPM.2023.23.
- 31 Katsuhito Nakashima, Noriki Fujisato, Diptarama Hendrian, Yuto Nakashima, Ryo Yoshinaka, Shunsuke Inenaga, Hideo Bannai, Ayumi Shinohara, and Masayuki Takeda. Parameterized DAWGs: Efficient constructions and bidirectional pattern searches. *Theor. Comput. Sci.*, 933:21–42, 2022. doi:10.1016/j.tcs.2022.09.008.
- 32 Gonzalo Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014. doi:10.1016/j.jda.2013.07.004.
- 33 Eric M. Osterkamp and Dominik Köppl. Extending the parameterized Burrows–Wheeler transform. In *Proc. DCC*, pages 143–152, March 2024.
- 34 Mihai Patrascu and Mikkel Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 166–175, 2014. doi:10.1109/FOCS.2014.26.