Theoretical Computer Science 437 (2012) 21–34

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Parameterized longest previous factor☆

Richard Beal*, Donald Adjeroh

West Virginia University, Lane Department of Computer Science and Electrical Engineering, Morgantown, WV 26506, USA

ARTICLE INFO

Article history:
Received 19 July 2011
Received in revised form 26 January 2012
Accepted 5 February 2012
Communicated by M. Crochemore

Keywords:
Parameterized suffix array
Parameterized longest common prefix
p-string
p-match
LPF
LCP

ABSTRACT

Given a string W, the longest previous factor (LPF) problem is to determine the maximum length of a previously occurring factor for each suffix occurring in W. The LPF problem is defined for traditional strings exclusively from the constant alphabet Σ. A parameterized string (p-string) is a string composed of symbols from a constant alphabet Σ and a parameter alphabet Π. We formulate the LPF problem in terms of p-strings by defining the parameterized longest previous factor (pLPF) problem. Subsequently, we present an expected linear time solution to construct the parameterized longest previous factor (*pLPF*) array. Given our pLPF solution, we show how to construct the *pLCP* (parameterized longest common prefix) array with the same general algorithm. We exploit the properties of the *pLPF* data structure to also construct the standard *LPF* (longest previous factor) and *LCP* (longest common prefix) arrays all in linear time. Further, we provide insight into the practicality of our construction algorithms.

1. Introduction

Given an n-length traditional string W from the alphabet Σ, the longest previous factor (LPF) problem is to determine the maximum length of a previously occurring factor for each suffix occurring in W. More formally, for any suffix u beginning at index i in the string W, the LPF problem is to identify the length of the longest factor between u and another suffix v at some position h before i in W, that is, $1 \leq h < i$. The LPF problem, introduced by Crochemore and Ilie [2], yields a data structure convenient for fundamental applications such as string compression [3] and detecting runs [4] within a string. In order to compute the *LPF* array, it is shown in [2] that the suffix array *SA* is useful to quickly identify the most lexicographically similar suffixes that constitute as previous factors for the chosen suffix in question. The use of *SA* expedites the work required to solve the LPF problem and likewise, is the *cornerstone* to solutions for many problems defined for traditional strings.

A generalization of traditional strings over an alphabet Σ is the parameterized string (p-string), introduced by Baker [5]. A p-string is a production of symbols from the alphabets Σ and Π, which represent the constant symbols and parameter symbols, respectively. The parameterized pattern matching (p-match) problem is to identify an equivalence between a pair of p-strings S and T when (1) the individual constant symbols match and (2) there exists a bijection between the parameter symbols of S and T. For example, the following p-strings that represent program statements $z = y * f / + +y;$ and $a = b * f / + +b;$ over the alphabet sets $\Sigma = \{*, /, +, =, ;\}$ and $\Pi = \{a, b, f, y, z\}$ satisfy both conditions and thus, the p-strings p-match. The motivation for addressing a problem in terms of p-strings is the range of problems that a single solution can address, including (1) exact pattern matching when $|\Pi| = 0$, (2) mapped matching (m-matching) when $|\Sigma| = 0$ [6], and clearly, (3) p-matching when $|\Sigma| > 0 \wedge |\Pi| > 0$. Prominent applications concerned with the p-match problem include detecting plagiarism in academia and industry, reporting similarities in biological sequences [7],

☆ A shorter version of this paper was presented at the International Workshop on Combinatorial Algorithms, Victoria, BC, Beal and Adjeroh 2011 [1]. This work was partly supported by grants from the National Historical Publications & Records Commission, and from WV-EPSCoR RCG.

* Corresponding author.
E-mail addresses: r.beal@computer.org (R. Beal), don@csee.wvu.edu (D. Adjeroh).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.02.004

discovering cloned code segments in a program [8], and even answering critical legal questions regarding the unauthorized use of intellectual property [9].

In this work, we introduce the parameterized longest previous factor (pLPF) for p-strings analogous to the LPF problem for traditional strings, which can similarly be used to study compression and duplication within p-strings. Given an n-length p-string T, the pLPF problem is to determine the length of the longest factor, or more specifically, the length of the longest parameterized suffix (p-suffix) v at some position h that matches with the p-suffix starting at i in T with $1 \leq h < i$. Our approach uses a parameterized suffix array (*pSA*) [10–13] for p-strings analogous to the traditional suffix array [14]. The major difficulty of the pLPF problem is that unlike traditional suffixes of a string, the p-suffixes are dynamic, varying with the starting position of the p-suffix. Thus, traditional LPF solutions cannot be directly applied to the pLPF problem.

Main Contributions: We define the parameterized longest previous factor (pLPF) problem to observe the longest previous factor (LPF) problem in terms of p-strings. Then, we present an expected linear time algorithm for constructing the *pLPF* (parameterized longest previous factor) data structure. Traditionally, the LPF problem is solved by using the longest common prefix (*LCP*) array. This was one approach used in [2]. In this work, we show how to go in the reverse direction: that is, given the pLPF solution, we now construct the *pLCP* (parameterized longest common prefix) array. Further, we identify how to exploit our algorithm for the pLPF problem to construct the *LPF* (longest previous factor) and *LCP* (longest common prefix) arrays. Our main results are formalized in the following:

Theorem **16**. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the* `compute_pLPF` *algorithm constructs the pLPF array in $O(\max\{n, m\gamma\})$ time, where m is the length of the longest p-match between a p-suffix at i and two defined p-suffixes in T and γ is dependent on the lexicographical orderings of specified p-suffixes in T.*

Corollary **17**. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the pLPF array can be constructed in $O(n)$ expected time.*

Theorem **18**. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the* `compute_pLPF` *algorithm can be used to construct the pLCP array in $O(\max\{n, m\phi\})$ time, where m is the length of the longest p-match between a p-suffix at i and two defined p-suffixes in T and ϕ is dependent on the lexicographical orderings of specified p-suffixes in T.*

Corollary **19**. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the pLCP array can be constructed in $O(n)$ expected time.*

Our algorithm `compute_pLPF` is the first solution that constructs the *pLPF* data structure. We further develop the algorithm `compute_pLCP` to construct the *pLCP* array, improving the original $O(n^2)$ construction time given in [10].

2. Background/related work

Baker [8] identifies three types of pattern matching: (1) exact matching, (2) parameterized matching (p-match), and (3) matching with modifications. The p-match generalizes exact matching with the parameterized string (p-string) composed of symbols from a constant symbol alphabet Σ and a parameter alphabet Π. A p-match exists between a pair of p-strings S and T of length n when (1) the constant symbols $\sigma \in \Sigma$ match and (2) there exists a bijection of parameter symbols $\pi \in \Pi$ between the pair of p-strings. The first p-match breakthroughs, namely, the `prev` encoding and the parameterized suffix tree (p-suffix tree) that demands the worst case construction time of $O(n(|\Pi| + \log(|\Pi| + |\Sigma|)))$, were introduced by Baker [5]. Additional improvements to the p-suffix tree were given in [15–17]. Like the traditional suffix tree [18–20], the p-suffix tree [5] implementation suffers from a large memory footprint. Other solutions that address the p-match problem without the space limitations of the p-suffix tree include the parameterized-KMP [6] and parameterized-BM [21], variants of traditional pattern matching approaches. Idury and Schäffer [22] studied the multiple p-match problem using the traditional Aho–Corasick [23] automata. The parameterized suffix array (p-suffix array) and the parameterized longest common prefix (*pLCP*) array combination is analogous to the suffix array and *LCP* array for traditional strings [14,18–20], which is both time and space efficient for pattern matching. Direct p-suffix array and *pLCP* construction was first introduced by Deguchi et al. [11] for binary strings with $|\Pi| = 2$, which required $O(n)$ work. Deguchi et al. [10] later proposed the first approach to p-suffix sorting and *pLCP* construction with an arbitrary alphabet size requiring $O(n^2)$ time in the worst case. We introduce new algorithms in [12,13] to p-suffix sort in linear time on average using coding methods from information theory.

In a novel application of the suffix array and the corresponding *LCP* array, Crochemore and Ilie [2] introduced the longest previous factor (LPF) problem for traditional strings. Table 1 shows an example LPF calculation for a short sequence $W = AAABABAB\$$. For any suffix u beginning at index i in string W, the LPF problem is to identify the exact matching longest factor between u and another suffix v starting prior to index i in W. We note that this definition is similar to (though not the same as) the *Prior* array used in [18]. Crochemore and Ilie [2] exploited the notion that the nearby elements within a suffix array are closely related en route to proposing a linear time solution to the LPF problem. They also proposed another linear time algorithm to compute the *LPF* array by using the *LCP* structure. The significance of an efficient solution to the LPF is that the resulting data structure simplifies computations in various string analysis procedures. Typical examples include computing the Lempel–Ziv factorization [3,24], which is fundamental in string compression algorithms such as the UNIX

Table 1
LPF calculation for string $W = AAABABAB\$$.

i	$SA[i]$	$W[SA[i] \dots n]$	$LCP[i]$	$W[i \dots n]$	$LPF[i]$
1	9	\$	0	*AAABABAB*\$	0
2	1	*AAABABAB*\$	0	*AABABAB*\$	2
3	2	*AABABAB*\$	2	*ABABAB*\$	1
4	7	*AB*\$	1	*BABAB*\$	0
5	5	*ABAB*\$	2	*ABAB*\$	4
6	3	*ABABAB*\$	4	*BAB*\$	3
7	8	*B*\$	0	*AB*\$	2
8	6	*BAB*\$	1	*B*\$	1
9	4	*BABAB*\$	3	\$	0

gzip utility [18,19] and in algorithms for detecting repeats in a string [4]. Other variants of the LPF problem were studied in [25–27]. Alternative solutions to the LPF problem are proposed in [28]. Our motivation to study the LPF in terms of p-strings is the power of parameterization with relevance to various important applications.

3. Preliminaries

A string on an alphabet Σ is a production $T = T[1]T[2] \dots T[n]$ from Σ^n with $n = |T|$ the length of T. We will use the following string notations: $T[i]$ refers to the ith symbol of string T, $T[i \dots j]$ refers to the substring $T[i]T[i+1] \dots T[j]$, and $T[i \dots n]$ refers to the ith suffix of T: $T[i]T[i+1] \dots T[n]$. Parameterized pattern matching requires the finite alphabets Σ and Π. Alphabet Σ denotes the set of constant symbols while Π represents the set of parameter symbols. Alphabets are defined such that $\Sigma \cap \Pi = \emptyset$. Furthermore, we append the terminal symbol $\$ \notin \Sigma \cup \Pi$ to the end of all strings to clearly distinguish between suffixes. For practical purposes, we can assume that $|\Sigma| + |\Pi| \leq n$ since otherwise a single mapping can be used to enforce the condition.

Definition 1. Parameterized string (p-string): A p-string is a production T of length n from $(\Sigma \cup \Pi)^*\$$.

Consider the alphabet arrangements $\Sigma = \{A, B\}$ and $\Pi = \{w, x, y, z\}$. Example p-strings include $S = AxByABxy\$$, $T = AwBzABwz\$$, and $U = AyByAByy\$$.

Definition 2. ([5,11]) **Parameterized matching (p-match)**: A pair of p-strings S and T are p-matches with $n = |S|$ if and only if $|S| = |T|$ and each $1 \leq i \leq n$ corresponds to one of the following:

1. $S[i], T[i] \in (\Sigma \cup \{\$\}) \wedge S[i] = T[i]$
2. $S[i], T[i] \in \Pi \wedge ((a) \vee (b))$ /* parameter bijection */
   (a) $S[i] \neq S[j], T[i] \neq T[j]$ **for any** $1 \leq j < i$
   (b) $S[i] = S[i-q]$ **iff** $T[i] = T[i-q]$ **for any** $1 \leq q < i$.

In our example, we have a p-match between the p-strings S and T since every constant/terminal symbol matches and there exists a bijection of parameter symbols between S and T. U does not satisfy the parameter bijection to p-match with S or T. The process of p-matching leads to defining the `prev` encoding.

Definition 3. ([5,11]) **Previous (`prev`) encoding**: Given $\mathbb{Z}$ as the set of non-negative integers, the function `prev`: $(\Sigma \cup \Pi)^*\$ \rightarrow (\Sigma \cup \mathbb{Z})^*\$$ accepts a p-string T of length n and produces a string Q of length n that (1) encodes constant/terminal symbols with the same symbol and (2) encodes parameters to point to ***previous*** like-parameters. More formally, Q is constructed of individual $Q[i]$ with $1 \leq i \leq n$ where:

$$Q[i] = \begin{cases} T[i], \textbf{if } T[i] \in (\Sigma \cup \{\$\}) \\ 0, \textbf{if } T[i] \in \Pi \wedge T[i] \neq T[j] \textbf{ for any } 1 \leq j < i \\ i-k, \textbf{if } T[i] \in \Pi \wedge k = \max\{j \mid T[i] = T[j], 1 \leq j < i\}. \end{cases}$$

For a p-string T of length n, the above $O(n)$ space `prev` encoding requires $O(n \log(\min\{n, |\Pi|\}))$ time for construction, which follows from the discussions of Baker [5,21] and Amir et al. [6] on the dependency of alphabet Π in p-match applications. Given an indexed alphabet and an auxiliary $O(|\Pi|)$ mapping structure, we can construct `prev` in $O(n)$ time. Using Definition 3, our working examples evaluate to $\texttt{prev}(S) = A0B0AB54\$$, $\texttt{prev}(T) = A0B0AB54\$$, $\texttt{prev}(U) = A0B2AB31\$$. The relationship between p-strings and the lexicographical ordering of the `prev` encoding is fundamental to the p-match problem.

Definition 4. `prev` **Lexicographical ordering**: Given the p-strings S and T and two symbols s and t from the encodings $\texttt{prev}(S)$ and $\texttt{prev}(T)$ respectively, the relationships $=$, $\neq$, $<$, and $>$ refer to lexicographical ordering between s and t. We define the ordering of symbols from a `prev` encoding of the production $(\Sigma \cup \mathbb{Z})^*\$$ to be $\$ < \zeta \in \mathbb{Z} < \sigma \in \Sigma$, where each ζ and σ is lexicographically sorted in their respective alphabets. The relationships $=$, $\neq$, $\prec$, and $\succ$ refer to the lexicographical ordering between strings. In the case of $\texttt{prev}(S)$ and $\texttt{prev}(T)$, $\texttt{prev}(S) \prec \texttt{prev}(T)$ when $\texttt{prev}(S)[1] = \texttt{prev}(T)[1]$, $\texttt{prev}(S)[2] = \texttt{prev}(T)[2], \dots, \texttt{prev}(S)[j-1] = \texttt{prev}(T)[j-1]$, $\texttt{prev}(S)[j] < \texttt{prev}(T)[j]$ for some $j, j \geq 1$. Similarly, we can define $=_k$, $\neq_k$, $\prec_k$, $\preceq_k$, $\succ_k$, and $\succeq_k$ to refer to the lexicographical relationships between a pair of p-strings considering only the first $k \geq 0$ symbols.

It is shown in [12,13] how to map a symbol in prev to an integer based on the ordering of Definition 4 and subsequently, call the function $\texttt{in}(x, X)$ to answer alphabet membership questions of the form $x \in X$ in constant time. The following proposition essential to the p-matching problem is directly related to the established symbol ordering.

Proposition 5. ([5]) *Two p-strings S and T p-match when* $\texttt{prev}(S) = \texttt{prev}(T)$. *Also,* $S \prec T$ *when* $\texttt{prev}(S) \prec \texttt{prev}(T)$ *and* $S \succ T$ *when* $\texttt{prev}(S) \succ \texttt{prev}(T)$.

The example prev encodings show a p-match between S and T since $\texttt{prev}(S) = A0B0AB54\$$ and $\texttt{prev}(T) = A0B0AB54\$$. Also, $U \succ S$ and $U \succ T$ since $\texttt{prev}(U) = A0B2AB31\$ \succ \texttt{prev}(S) = \texttt{prev}(T) = A0B0AB54\$$. We use the ordering established in Definition 4 to define the parameterized suffix array and the parameterized longest common prefix array.

Definition 6. Parameterized suffix array (*pSA*): The *pSA* for a p-string T of length n maintains a lexicographical ordering of the indices i representing individual p-suffixes $\texttt{prev}(T[i \ldots n])$ with $1 \leq i \leq n$, such that $\texttt{prev}(T[pSA[q] \ldots n]) \prec \texttt{prev}(T[pSA[q+1] \ldots n]) \forall q$, $1 \leq q < n$.

The *pSA* is analogous to the suffix array *SA* defined for traditional strings. Let the rank array R rank each p-suffix index in the p-string T to its position in the corresponding *pSA* or *SA*. The following *pLCP* array is used with the *pSA* for efficient p-matching [10,11,13].

Definition 7. Parameterized longest common prefix (*pLCP*) array: The *pLCP* array for a p-string T of length n maintains the length of the longest common prefix between neighboring p-suffixes. We define $\texttt{plcp}(\alpha, \beta) = \max\{k \mid \texttt{prev}(\alpha) =_k \texttt{prev}(\beta)\}$. Then, $pLCP[1] = 0$ and $pLCP[i] = \texttt{plcp}(T[pSA[i] \ldots n], T[pSA[i-1] \ldots n])$, $2 \leq i \leq n$.

For the working example $T = AwBzABwz\$$ with $\texttt{prev}(T) = A0B0AB54\$$, we have $pSA = \{9, 8, 7, 4, 2, 1, 5, 6, 3\}$ and $pLCP = \{0, 0, 1, 1, 1, 0, 1, 0, 2\}$. The encoding prev is supplemented by the encoding forw.

Definition 8. ([12,13]) **Forward (forw) encoding**: Let the function $\texttt{rev}(T)$ reverse the p-string T and let $\texttt{repl}(T, x, y)$ replace all occurrences in T of the symbol x with y. We define the function forw for the p-string T of length n as $\texttt{forw}(T) = \texttt{rev}(\texttt{repl}(\texttt{prev}(\texttt{rev}(T)), 0, n))$.

For a p-string T of length n, the encoding forw (1) encodes constant/terminal symbols with the same symbol and (2) encodes each parameter p with the ***forward*** distance to the next occurrence of p or an unreachable forward distance n. Our definition of forw generates output mirroring the fw encoding used by Deguchi et al. [10,11]. The forw encodings in our example with $n = 9$ are $\texttt{forw}(S) = A5B4AB99\$$, $\texttt{forw}(T) = A5B4AB99\$$, $\texttt{forw}(U) = A2B3AB19\$$.

Definition 9. ([2]) **Longest previous factor (*LPF*)**: For an n-length traditional string W, the LPF is defined for each index $1 \leq i \leq n$ such that $LPF[i] = \max(\{0\} \cup \{k \mid W[i \ldots n] =_k W[h \ldots n], 1 \leq h < i\})$.

The traditional string $W = AAABABAB\$$ yields $LPF = \{0, 2, 1, 0, 4, 3, 2, 1, 0\}$.

4. Parameterized LPF

We define the parameterized longest previous factor (pLPF) problem as follows to observe the traditional LPF problem in terms of p-strings.

Definition 10. Parameterized longest previous factor (*pLPF*): For a p-string T of length n, the pLPF array is defined for each index $1 \leq i \leq n$ to maintain the length of the *longest factor* between a p-suffix and the longest factor previously occurring in T. More formally, $pLPF[i] = \max(\{0\} \cup \{k \mid \texttt{prev}(T[i \ldots n]) =_k \texttt{prev}(T[h \ldots n]), 1 \leq h < i\})$.

The pLPF problem requires that we deal with p-suffixes, which are suffixes encoded with prev. This task is more demanding than the LPF for traditional strings because Lemma 11 indicates that we cannot guarantee the individual suffixes of a single prev encoding to be p-suffixes. Thus, the changing nature of the prev encoding poses a major challenge to efficient and correct construction of the *pLPF* array using current algorithms that construct the *LPF* array for traditional strings.

Lemma 11. *Given a p-string T of length n, the suffixes of* $\texttt{prev}(T)$ *are not necessarily the p-suffixes of T. More formally, if* $\pi \in \Pi$ *occurs more than once in T, then* $\exists i$, *s.t.* $\texttt{prev}(T[i \ldots n]) \neq \texttt{prev}(T)[i \ldots n]$, $1 \leq i \leq n$.

Proof. Suppose the *only* parameter symbol to occur in the p-string T is $\pi \in \Pi$, which exists *only* at positions α and β with $\alpha < \beta$. Suppose that indeed $\texttt{prev}(T[\alpha \ldots n]) = \texttt{prev}(T)[\alpha \ldots n]$ and $\texttt{prev}(T[\beta \ldots n]) = \texttt{prev}(T)[\beta \ldots n]$. By Definition 3, the first occurrence of π at position α will be prev encoded by 0 and the π at position β will be prev encoded by $\beta - \alpha$. So, in the case of suffix α, $\texttt{prev}(T[\alpha \ldots n]) = \texttt{prev}(T)[\alpha \ldots n]$. At suffix β, the encoding of π at position β in T will *change* to 0 in $\texttt{prev}(T[\beta \ldots n])$ by Definition 3 whereas $\texttt{prev}(T)[\beta \ldots n]$ will retain the *old* encoding of $\beta - \alpha$ since π *still* occurs in $\texttt{prev}(T)$ at position α. The π at position β forces $\texttt{prev}(T[\beta \ldots n]) \neq \texttt{prev}(T)[\beta \ldots n]$, a contradiction. □

Table 2
pLPF calculation for p-string $T = AAAwBxyyAAAzwwB\$$.

i	$pSA[i]$	$pLCP[i]$	$\texttt{prev}(T[pSA[i]\ldots n])$	$before_<[pSA[i]]$	$before_>[pSA[i]]$	$pLPF[i]$
1	16	0	\$	−1	6	0
2	6	0	001*AAA*001*B*\$	−1	4	2
3	12	3	001*B*\$	6	7	1
4	7	1	01*AAA*001*B*\$	6	4	0
5	13	2	01*B*\$	7	8	0
6	8	1	0*AAA*001*B*\$	7	4	1
7	14	1	0*B*\$	8	4	1
8	4	2	0*B*001*AAA*091*B*\$	−1	3	1
9	11	0	*A*001*B*\$	4	3	4
10	3	2	*A*0*B*001*AAA*091*B*\$	−1	2	3
11	10	1	*AA*001*B*\$	3	2	2
12	2	3	*AA*0*B*001*AAA*091*B*\$	−1	1	3
13	9	2	*AAA*001*B*\$	2	1	2
14	1	4	*AAA*0*B*001*AAA*091*B*\$	−1	−1	2
15	15	0	*B*\$	1	5	1
16	5	1	*B*001*AAA*001*B*\$	1	−1	0

Algorithm 1. pLPF computation.

```
1  int[] compute_pLPF(int before<[],int before>[],int R[]){
2     int  pLPF[n], pLPF<[n]={0,...,0}, pLPF>[n]={0,...,0}, i
3     for i = 1 to n {
4        (j,k) = Ω(i,pLPF<,pLPF>,before<,before>,R)
5        pLPF<[i] = Λ(i,before<[i],j)
6        if(before> ≠ null)
7           pLPF>[i] = Λ(i,before>[i],k)
8        pLPF[i] = max{pLPF<[i],pLPF>[i]}
9     }return pLPF
10 }
```

Consider the p-string $T = AAAwBxyyAAAzwwB\$$ using the previously defined alphabets. Table 2 shows the pLPF computation for the defined p-string T. We note the intricacies of Lemma 11 since simply using the traditional LPF algorithm (1) with T yields $LPF = \{0, 2, 1, 0, 0, 0, 0, 1, 3, 2, 1, 0, 1, 2, 1, 0\}$, (2) with $\texttt{prev}(T)$ produces $LPF = \{0, 2, 1, 0, 0, 0, 1, 1, 0, 4, 3, 2, 1, 0, 1, 1, 0\}$, and (3) with $\texttt{forw}(T)$ generates the array $LPF = \{0, 2, 1, 0, 0, 0, 0, 1, 3, 2, 1, 3, 2, 1, 1, 0\}$, neither of which is the correct $pLPF$ array.

Crochemore and Ilie [2] efficiently solve the LPF problem for a traditional string W by exploiting the properties of the suffix array SA. They construct the arrays $prev_<[1\ldots n]$ and $prev_>[1\ldots n]$, which for each i in W maintain the suffix $h < i$ positioned respectively before and after suffix i in SA; when no such suffix exists, the element is denoted by -1. The conceptual idea to compute the $prev_<$ and $prev_>$ arrays in linear time via deletions in a doubly linked list of the SA was suggested in [2]. The algorithm is given in [13]. Furthermore, we will refer to $prev_<$ and $prev_>$ as $before_<$ and $before_>$ respectively, in order to avoid confusion with the $\texttt{prev}$ encoding for p-strings. Then, $LPF[i]$ is the maximum q between $W[i\ldots n] =_q W[before_<[i]\ldots n]$ and $W[i\ldots n] =_q W[before_>[i]\ldots n]$. The magic of a linear time solution to constructing the LPF array is achieved through the computation of an element by *extending* the previous element, more formally $LPF[i] \geq LPF[i-1] - 1$, which is a variant of the extension property used in LCP construction proven by Kasai et al. [29]. We prove that this same property holds for the pLPF problem defined on p-strings.

Lemma 12. *The pLPF for a p-string T of length n is such that $pLPF[i] \geq pLPF[i-1] - 1$ with $1 < i \leq n$.*

Proof. Consider $pLPF[i]$ at $i = 1$ by which Definition 10 requires that we find a previous factor at $1 \leq h < 1$ that does not exist; i.e., $pLPF[1] = 0$. At $i = 2$, indeed $pLPF[2] \geq pLPF[1] - 1 = -1$ is clearly true for all succeeding elements in which a previous factor does not exist. For arbitrary $i = j$ with $1 < j < n$, suppose that the maximum length factor is at $g < j$ and without loss of generality, consider that the first $q \geq 2$ symbols match so that $\texttt{prev}(T[j\ldots n]) =_q \texttt{prev}(T[g\ldots n])$. Thus, $pLPF[j] = q$. Shifting the computation to $i = j + 1$, we lose the symbols $\texttt{prev}(T[j])$ and $\texttt{prev}(T[g])$ in the p-suffixes at j and g respectively. By Proposition 5, $\texttt{prev}(T[j\ldots j+q-1]) = \texttt{prev}(T[g\ldots g+q-1]) \Rightarrow \texttt{prev}(T[j]) = \texttt{prev}(T[g])$ and as a consequence of the $\texttt{prev}$ encoding in Definition 3 we have $\texttt{prev}(T[i\ldots n]) =_{q-1} \texttt{prev}(T[g+1\ldots n])$. Since we can guarantee that $\exists$ a factor with $(q-1)$ symbols for $pLPF[i]$ or possibly find another factor at h with $1 \leq h < i$ matching q or more symbols, the lemma holds. □

For the traditional LPF problem, the property of $LPF[i] \geq LPF[i-1] - 1$ assists in extending each match between the suffix at i and the suffixes at $before_<[i]$ and $before_>[i]$ by observing the respective matches at $i - 1$. In other words, traditional strings have the property that *always* $T[before_<[i-1]+1\ldots n] \preceq T[before_<[i]\ldots n] \prec T[i\ldots n] \prec T[before_>[i]\ldots n] \preceq T[before_>[i-1]+1\ldots n]$ with $i > 1$ as long as the $before_<$ and $before_>$ elements exist. This lexicographical ordering allows us to separately and individually extend the matches between the suffixes i and $before_<[i]$ and also, between the suffixes i

Algorithm 2a. p-matcher function Λ.

```
1  int Λ(int a, int b, int q) {
2     boolean c = true
3     int x, y
4     if(b = −1) return 0
5     while(c ∧ (a+q) ≤ n ∧ (b+q) ≤ n) {
6        x = prevT[a+q], y = prevT[b+q]
7        if(in(x,Σ) ∧ in(y,Σ)){
8           if(x = y) q++
9           else c = false
10       }else if(in(x,ℤ) ∧ in(y,ℤ)){
11          if(q < x) x = 0
12          if(q < y) y = 0
13          if(x = y) q++
14          else c = false
15       }else c = false
16    }return q
17 }
```

Algorithm 2b. p-match manager function Ω.

```
1  (int,int) Ω(int i,int pLPF<[],int pLPF>[],int before<[],int before>[],int R[]) {
2    int j = 0, k = 0, a = 0, b = 0, c = 0, d = 0
3    if(i > 1 ∧ before<≠null){
4      a = before<[i−1]+1, c = pLPF<[i−1]−1
5      if(before>≠null) { b = before>[i−1]+1, d = pLPF>[i−1]−1 }
6      if(before>≠null ∧ before<[i]≠−1 ∧ before>[i]≠−1){
7        if(a=0 ∧ b=0){ j = pLPF>[before<[i]], k = pLPF<[before>[i]] }
8        else if(a=0 ∧ R[b]<R[i]){ j = d, k = pLPF<[before>[i]] }
9        else if(a=0 ∧ R[b]>R[i]){ j = pLPF>[before<[i]], k = d }
10       else if(b=0 ∧ R[a]<R[i]){ j = c, k = pLPF<[before>[i]] }
11       else if(b=0 ∧ R[a]>R[i]){ j = pLPF>[before<[i]], k = c }
12       /* Fig. 1(a) */ else if(R[a]<R[i]<R[b]){  j = c, k = d  }
13       /* Fig. 1(b) */ else if(R[b]<R[i]<R[a]){  j = d, k = c  }
14       /* Fig. 1(c) */ else if(R[a]<R[b]<R[i]){  j = d, k = pLPF<[before>[i]] }
15       /* Fig. 1(d) */ else if(R[b]<R[a]<R[i]){  j = c, k = pLPF<[before>[i]] }
16       /* Fig. 1(e) */ else if(R[i]<R[a]<R[b]){  j = pLPF>[before<[i]], k = c }
17       /* Fig. 1(f) */ else if(R[i]<R[b]<R[a]){  j = pLPF>[before<[i]], k = d }
18     }else if(a>0 ∧ b>0 ∧ (before<[i]=−1 ∨ before>[i]=−1)) {
19       if(R[a]<R[b]<R[i]) j = d
20       else if(R[b]<R[a]<R[i]) j = c
21       else if(R[i]<R[a]<R[b]) k = c
22       else if(R[i]<R[b]<R[a]) k = d
23     }else if(a>0 ∧ (before> = null ∨ before<[i]=−1 ∨ before>[i]=−1)) {
24       if(R[a]<R[i]) j = c
25       else k = c
26     }else if(b>0 ∧ (before<[i]=−1 ∨ before>[i]=−1)) {
27       if(R[b]<R[i]) j = d
28       else k = d
29     }j = max{0,j}, k = max{0,k}
30   }return (j,k)
31 }
```

and $before_>[i]$ for the traditional LPF problem. Even though we similarly prove that $pLPF[i] \geq pLPF[i-1]-1$ in Lemma 12, the traditional lexicographical ordering of suffixes does not hold for p-suffixes.

Lemma 13. *Given an n-length p-string T, let* $x = before_<[i]$ *exist* $(1 \leq x < n)$ *and* $y = before_>[i]$ *exist* $(1 \leq y < n)$ *with* $1 \leq i < n$. *Even though* $R[x] < R[i] < R[y]$, *it is not guaranteed that* $R[x+1] < R[i+1] < R[y+1]$.

Proof. By the definition of the $before_<$ and $before_>$ arrays, it is the case that $x = before_<[i]$ is chosen such that $R[x] < R[i]$ and $y = before_>[i]$ is chosen such that $R[y] > R[i]$. Thus, $R[x] < R[i] < R[y]$ or more formally, $\texttt{prev}(T[x \ldots n]) \prec \texttt{prev}(T[i \ldots n]) \prec \texttt{prev}(T[y \ldots n])$. Consider, for instance, that $\texttt{prev}(T[x \ldots n]) =_q \texttt{prev}(T[i \ldots n]) =_q \texttt{prev}(T[y \ldots n])$ for some $q > 3$ with $(\max\{i, x, y\} + q) < n$. Further consider that $T[x+q]$, $T[i+q]$, and $T[y+q]$ are parameters and $\texttt{prev}(T[x \ldots n])[q+1] = 1$, $\texttt{prev}(T[i \ldots n])[q+1] = 2$, and $\texttt{prev}(T[y \ldots n])[q+1] = q$. It follows that $\texttt{prev}(T[x+1 \ldots n])[q] = 1$, $\texttt{prev}(T[i+1 \ldots n])[q] = 2$, and $\texttt{prev}(T[y+1 \ldots n])[q] = 0$. Since it is still the case that $\texttt{prev}(T[x+1 \ldots n]) =_{q-1} \texttt{prev}(T[i+1 \ldots n]) =_{q-1} \texttt{prev}(T[y+1 \ldots n])$, the fact that $(\texttt{prev}(T[y+1 \ldots n])[q] = 0) < (\texttt{prev}(T[x+1 \ldots n])[q] = 1) < (\texttt{prev}(T[i+1 \ldots n])[q] = 2)$ yields the lexicographical relationship $\texttt{prev}(T[y+1 \ldots n]) \prec \texttt{prev}(T[x+1 \ldots n]) \prec \texttt{prev}(T[i+1 \ldots n])$ and $R[y+1] < R[x+1] < R[i+1]$, proves the lemma. □

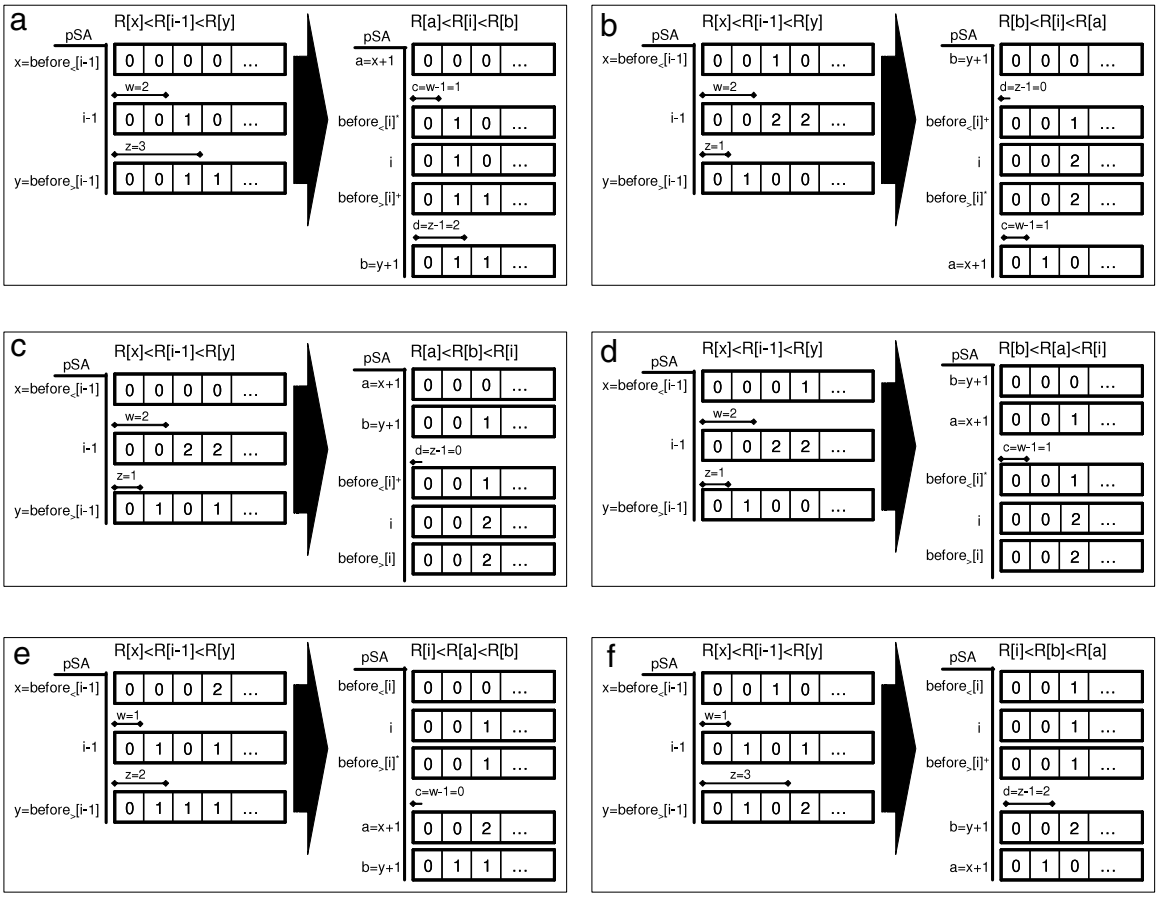

Fig. 1. Examples that correspond to core cases of the Ω function (note: * and $^+$ denote that the p-suffix may possibly be the p-suffixes at a and b respectively).

Lemmas 11 and 13 formally identify the significant differences between the LPF and pLPF problems. In this research, we show that it is possible to solve the pLPF problem with an algorithm similar to the `compute_LPF` algorithm in [2] by addressing the lemmas with two different functions. We introduce `compute_pLPF` in Algorithm 1 to construct the *pLPF* array. (For future reasons in this work, the `compute_pLPF` algorithm permits $before_> =$ **null**.) This algorithm utilizes (1) the special p-matcher function Λ in Algorithm 2a to address Lemma 11 by properly handling the matching of p-suffixes and (2) the p-match manager function Ω in Algorithm 2b to address Lemma 13 by identifying the lexicographical similarities between p-suffixes and returning the appropriate match length, which in turn, is used to extend future matches. More specifically, the role of Λ is to *extend* the matches between the p-suffixes at a and b beyond the initial q symbols by directly comparing constant/terminal symbols and comparing the dynamically adjusted parameter encodings for each p-suffix. Exactly *where* the Λ function begins p-matching is determined by the Ω function.

In more detail, the Ω function uses the previously matched p-suffixes to identify *how* the lexicographical ordering of those p-suffixes can assist in future matching between p-suffixes. The lexicographical orderings are displayed in Fig. 1. Using what is known in previous matches and the identified lexicographical ordering, the Ω function returns a pair (j, k), which indicates where to begin matching between the p-suffixes at i and $before_<[i]$ and between the p-suffixes at i and $before_>[i]$ respectively. More formally, the (j, k) pair proves that the following matches *already* exist: $\texttt{prev}(T[before_<[i] \ldots n]) =_j \texttt{prev}(T[i \ldots n])$ and $\texttt{prev}(T[before_>[i] \ldots n]) =_k \texttt{prev}(T[i \ldots n])$. The matches may be *extended* beyond this (j, k) starting point via the Λ function. The correctness of Ω is proven in the following lemma.

Lemma 14. *For any i chosen sequentially $i = 2, 3, \ldots, n$ with n as the length of the p-string T, the function Ω correctly returns the number of matching symbols j and k with respect to p-suffix i based on symbols previously matched such that $\texttt{prev}(T[before_<[i] \ldots n]) =_j \texttt{prev}(T[i \ldots n])$ and $\texttt{prev}(T[before_>[i] \ldots n]) =_k \texttt{prev}(T[i \ldots n])$.*

Proof. Let $x = before_<[i-1]$ and $y = before_>[i-1]$. Without loss of generality, assume that x and y exist, namely $1 \le x < n$ and $1 \le y < n$. Further, let $1 \le m \le n$, $pLPF_<[m] = \max(\{0\} \cup \{q \mid \texttt{prev}(T[before_<[m] \ldots n]) =_q \texttt{prev}(T[m \ldots n])\})$ and $pLPF_>[m] = \max(\{0\} \cup \{q \mid \texttt{prev}(T[before_>[m] \ldots n]) =_q \texttt{prev}(T[m \ldots n])\})$. Consider $2 \le i < n$ and assume that the following are completed: $\texttt{prev}(T[x \ldots n]) =_w \texttt{prev}(T[i-1 \ldots n])$ and $\texttt{prev}(T[y \ldots n]) =_z \texttt{prev}(T[i-1 \ldots n])$ or alternatively, $w = pLPF_<[i-1]$ and $z = pLPF_>[i-1]$. Let $c = w - 1$ and $d = z - 1$. We now prove that Ω uses previous

match lengths of w and z at p-suffix $(i-1)$ to correctly return what is known about the current matches with the p-suffix at i. Primarily, we need to identify the lexicographical ordering between the previously matched p-suffixes less the first symbol, i.e. the p-suffixes at $a = x+1$, i, and $b = y+1$ or $\texttt{prev}(T[a \ldots n])$, $\texttt{prev}(T[i \ldots n])$, and $\texttt{prev}(T[b \ldots n])$. As a basis of the proof, it is known from the definition of the $before_<$ and $before_>$ arrays that $R[x] < R[i-1] < R[y]$ and for $g = before_<[i]$ and $h = before_>[i]$, also $R[g] < R[i] < R[h]$. As a consequence of Lemma 13, the following non-trivial orderings (displayed in Fig. 1) are possible: (a) $R[a] < R[i] < R[b]$, (b) $R[b] < R[i] < R[a]$, (c) $R[a] < R[b] < R[i]$, (d) $R[b] < R[a] < R[i]$, (e) $R[i] < R[a] < R[b]$, and (f) $R[i] < R[b] < R[a]$.

- For case (a) $R[a] < R[i] < R[b]$, since already $\texttt{prev}(T[x \ldots n]) =_w \texttt{prev}(T[i-1 \ldots n])$ and now $R[a] \leq R[g] < R[i] \Rightarrow \texttt{prev}(T[g \ldots n]) =_{\max\{0,c\}} \texttt{prev}(T[i \ldots n])$. Also, since already $\texttt{prev}(T[y \ldots n]) =_z \texttt{prev}(T[i-1 \ldots n])$ and now $R[i] < R[h] \leq R[b] \Rightarrow \texttt{prev}(T[h \ldots n]) =_{\max\{0,d\}} \texttt{prev}(T[i \ldots n])$. Function Ω correctly returns $(j = \max\{0, c\}, k = \max\{0, d\})$.
- For case (b) $R[b] < R[i] < R[a]$, since already $\texttt{prev}(T[y \ldots n]) =_z \texttt{prev}(T[i-1 \ldots n])$ and now $R[b] \leq R[g] < R[i] \Rightarrow \texttt{prev}(T[g \ldots n]) =_{\max\{0,d\}} \texttt{prev}(T[i \ldots n])$. Also, since already $\texttt{prev}(T[x \ldots n]) =_w \texttt{prev}(T[i-1 \ldots n])$ and now $R[i] < R[h] \leq R[a] \Rightarrow \texttt{prev}(T[h \ldots n]) =_{\max\{0,c\}} \texttt{prev}(T[i \ldots n])$. Function Ω correctly returns $(j = \max\{0, d\}, k = \max\{0, c\})$.
- For case (c) $R[a] < R[b] < R[i]$, since already $\texttt{prev}(T[y \ldots n]) =_z \texttt{prev}(T[i-1 \ldots n])$ and now $R[a] < R[b] \leq R[g] < R[i] \Rightarrow \texttt{prev}(T[g \ldots n]) =_{\max\{0,d\}} \texttt{prev}(T[i \ldots n])$. Due to the fact that the previously matched p-suffixes are *both* lexicographically less than p-suffix i, the only matches conducted previously that can be used to connect the match between $\texttt{prev}(T[h \ldots n])$ and $\texttt{prev}(T[i \ldots n])$ are from the matches between $\texttt{prev}(T[h \ldots n])$ and $\texttt{prev}(T[before_<[h] \ldots n])$. For the sake of discussion, assume that $before_<[h]$ exists. Then, we already know that $\texttt{prev}(T[before_<[h] \ldots n]) =_v \texttt{prev}(T[h \ldots n])$ and now $R[before_<[h]] < R[i] < R[h] \Rightarrow \texttt{prev}(T[before_<[h] \ldots n]) =_v \texttt{prev}(T[i \ldots n]) =_v \texttt{prev}(T[h \ldots n])$ where $v = pLPF_<[h] \geq 0$. Function Ω correctly returns $(j = \max\{0, d\}, k = v)$.
- For case (d) $R[b] < R[a] < R[i]$, the argument is similar to that of case (c) except that now $R[b] < R[a] \leq R[g] < R[i]$ and because already $\texttt{prev}(T[x \ldots n]) =_w \texttt{prev}(T[i-1 \ldots n]) \Rightarrow \texttt{prev}(T[g \ldots n]) =_{\max\{0,c\}} \texttt{prev}(T[i \ldots n])$. Function Ω correctly returns $(j = \max\{0, c\}, k = pLPF_<[h])$.
- For case (e) $R[i] < R[a] < R[b]$, since already $\texttt{prev}(T[x \ldots n]) =_w \texttt{prev}(T[i-1 \ldots n])$ and now $R[i] < R[h] \leq R[a] < R[b] \Rightarrow \texttt{prev}(T[h \ldots n]) =_{\max\{0,c\}} \texttt{prev}(T[i \ldots n])$. Due to the fact that the previously matched p-suffixes are *both* lexicographically greater than p-suffix i, the only matches conducted previously that can be used to connect the match between $\texttt{prev}(T[g \ldots n])$ and $\texttt{prev}(T[i \ldots n])$ are from the matches between $\texttt{prev}(T[g \ldots n])$ and $\texttt{prev}(T[before_>[g] \ldots n])$. For the sake of discussion, assume that $before_>[g]$ exists. Then, we already know that $\texttt{prev}(T[before_>[g] \ldots n]) =_v \texttt{prev}(T[g \ldots n])$ and now $R[g] < R[i] < R[before_>[g]] \Rightarrow \texttt{prev}(T[before_>[g] \ldots n]) =_v \texttt{prev}(T[i \ldots n]) =_v \texttt{prev}(T[g \ldots n])$ where $v = pLPF_>[g] \geq 0$. Function Ω correctly returns $(j = v, k = \max\{0, c\})$.
- For case (f) $R[i] < R[b] < R[a]$, the argument is similar to that of case (e) except that now $R[i] < R[h] \leq R[b] < R[a]$ and because already $\texttt{prev}(T[y \ldots n]) =_z \texttt{prev}(T[i-1 \ldots n]) \Rightarrow \texttt{prev}(T[h \ldots n]) =_{\max\{0,d\}} \texttt{prev}(T[i \ldots n])$. Function Ω correctly returns $(j = pLPF_>[g], k = \max\{0, d\})$.

The other trivial cases of function Ω are situations when one or both of the entries in $before_<$ or $before_>$ do not exist and are handled using the same techniques as the previous cases. Thus, Ω correctly uses previous matches to return the pair (j, k) of known lengths of the current p-matches between the p-suffixes at $before_<[i]$, i, and $before_>[i]$ where $\texttt{prev}(T[before_<[i] \ldots n]) =_j \texttt{prev}(T[i \ldots n])$ and $\texttt{prev}(T[before_>[i] \ldots n]) =_k \texttt{prev}(T[i \ldots n])$. □

Even though there are several cases and significant detail within the Ω function, a single call to Ω requires $O(1)$ time, which is formalized in the following lemma.

Lemma 15. *Each call to the function Ω executes in $O(1)$ time.*

Proof. By an analysis of the Ω function in Algorithm 2b, we can trivially conclude that each call to Ω only executes a series of selection and assignment statements and therefore, the lemma holds. □

Now, the discussion of pLPF moves toward analyzing the time complexity of the complete `compute_pLPF` algorithm. The traditional LPF problem is solved in [2] by `compute_LPF` in $O(n)$ time. The pLPF problem includes the added intricacies of Lemmas 11 and 13, which are addressed by functions Λ and Ω respectively. These intricacies require a more involved time complexity analysis, which is formalized in the following theorem.

Theorem 16. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the `prev` encoding of T, and pSA, the parameterized suffix array for T, the `compute_pLPF` algorithm constructs the pLPF array in $O(\max\{n, m\gamma\})$ time, where m is the length of the longest p-match between a p-suffix at i and two defined p-suffixes in T and γ is dependent on the lexicographical orderings of specified p-suffixes in T.*

Proof. Since Algorithm `compute_pLPF` utilizes Ω in order to extend p-matches by knowledge of previous p-matches, it follows from Lemma 14 that `compute_pLPF` correctly exploits the properties of p-suffixes and pLPF to correctly compute factors with the p-matching function Λ. We now analyze the running time of `compute_pLPF`. Primarily, the time to compute the arrays $before_<$ and $before_>$ require $O(n)$ processing as detailed in [13]. What remains now is to show that,

between Algorithms 1, 2a, and 2b, the total number of matches performed, or the number of times that the body of the **while** loop (lines 6–15 in Algorithm 2a) will be executed, is in $O(\max\{n, m\gamma\})$. Consider the first iteration of the **for** loop in Algorithm 1, i.e. $i = 1$. The initial call to Ω returns $(0, 0)$ in $O(1)$ time by Lemma 15. Since $before_<[1] = before_>[1] = -1$ in this case, no matching is performed by Λ, yielding $pLPF_<[1] = 0$, $pLPF_>[1] = 0$, and the result $pLPF[1] = 0$. Next, consider $i = 2$. Here, Ω returns $(0, 0)$ in $O(1)$ time by Lemma 15. Assume that $\texttt{prev}(T[1 \ldots n]) \prec \texttt{prev}(T[2 \ldots n])$ and so $before_<[2] = 1$ and $before_>[2]$ does not exist, namely $before_>[2] = -1$. Assume that, in the worst case, $\texttt{prev}(T[1 \ldots n]) =_{n-2} \texttt{prev}(T[2 \ldots n])$, which requires $O(n)$ work from Λ to compute $pLPF_<[2] = n - 2$, $pLPF_>[2] = 0$, and the result $pLPF[2] = n - 2$. At this point, the entries $pLPF[1]$ and $pLPF[2]$ are computed in $O(n)$ time. Now, we must consider how the lexicographical ordering between the p-suffix $\texttt{prev}(T[before_<[2] + 1 \ldots n])$ and $\texttt{prev}(T[3 \ldots n])$ helps us to extend the next match at $i = 3$. More specifically, considering that currently $pLPF_<[i - 1] = n - 2$ and $pLPF_>[i - 1] = 0$, is $\texttt{prev}(T[before_<[i] \ldots n]) =_{\max\{0, pLPF_<[i-1]-1\}} \texttt{prev}(T[i \ldots n])$ or $\texttt{prev}(T[before_>[i] \ldots n]) =_{\max\{0, pLPF_<[i-1]-1\}} \texttt{prev}(T[i \ldots n])$? So, we must consider the impact of the individual cases of function Ω as they relate to the matching performed by Λ. These cases were proven for correctness in Lemma 14 and the non-trivial cases are illustrated in Fig. 1. For brevity and without loss of generality, we further consider these non-trivial cases by assuming that all $before_<[i] \geq 1$ and $before_>[i] \geq 1$. The proof is divided into the following two parts.

- First, assume that every call to Ω will execute either case (a) or case (b) of Fig. 1. In these situations, we know that the p-suffix at i is lexicographically between the p-suffixes at $before_<[i - 1] + 1$ and $before_>[i - 1] + 1$, i.e., $\texttt{prev}(T[before_<[i - 1] + 1 \ldots n]) \prec \texttt{prev}(T[i \ldots n]) \prec \texttt{prev}(T[before_>[i - 1] + 1 \ldots n])$ or $\texttt{prev}(T[before_>[i - 1] + 1 \ldots n]) \prec \texttt{prev}(T[i \ldots n]) \prec \texttt{prev}(T[before_<[i - 1] + 1 \ldots n])$. Since we already know that the p-suffix at i is lexicographically between the p-suffixes at $before_<[i]$ and $before_>[i]$ by definition of the $before_<$ and $before_>$ arrays, we can guarantee that the p-suffixes at $before_<[i]$ and $before_>[i]$ are *at least as lexicographically similar* to the p-suffix at i as the previously matched p-suffixes at $before_<[i - 1] + 1$ and $before_>[i - 1] + 1$. Let $u = \max\{0, pLPF_<[i - 1] - 1\}$ and $v = \max\{0, pLPF_>[i - 1] - 1\}$. It follows that $\texttt{prev}(T[before_<[i - 1] + 1 \ldots n]) \preceq_u \texttt{prev}(T[before_<[i] \ldots n]) \prec_u \texttt{prev}(T[i \ldots n]) \prec_v \texttt{prev}(T[before_>[i] \ldots n]) \preceq_v \texttt{prev}(T[before_>[i - 1] + 1 \ldots n])$ or $\texttt{prev}(T[before_>[i - 1] + 1 \ldots n]) \preceq_v \texttt{prev}(T[before_<[i] \ldots n]) \prec_v \texttt{prev}(T[i \ldots n]) \prec_u \texttt{prev}(T[before_>[i] \ldots n]) \preceq_u \texttt{prev}(T[before_<[i - 1] + 1 \ldots n])$. So, we can always use previous matches to compute $pLPF_<[i]$, $pLPF_>[i]$, and hence, $pLPF[i]$. Recall that earlier in the proof, already $pLPF[2] = n - 2$ is processed in $O(n)$ time. From Lemma 12 and the fact that the decreasing lengths of the succeeding p-suffixes at $i = 3, 4, \ldots, n$ cannot *extend* the matches any further, successive calls to Λ and Ω are performed in $O(1)$ time and the previous work of $O(n)$ is amortized across *all* of the n iterations of the **for** loop in `compute_pLPF`. Thus, the total work is $O(n)$.
- Second, assume *initially* that every call to Ω will execute either case (c), (d), (e), or (f) of Fig. 1. Let $u = \max\{0, pLPF_<[i - 1] - 1\}$ and $v = \max\{0, pLPF_>[i - 1] - 1\}$. In general from these cases, we know that $\texttt{prev}(T[before_<[i] \ldots n]) =_u \texttt{prev}(T[i \ldots n]) \vee \texttt{prev}(T[before_<[i] \ldots n]) =_v \texttt{prev}(T[i \ldots n]) \vee \texttt{prev}(T[before_>[i] \ldots n]) =_u \texttt{prev}(T[i \ldots n]) \vee \texttt{prev}(T[before_>[i] \ldots n]) =_v \texttt{prev}(T[i \ldots n])$ by a call to Ω that requires $O(1)$ time by Lemma 15. In other words, we only know how to extend the match between the p-suffix at i and *either* the p-suffix at $before_<[i]$ or $before_>[i]$. In order to determine how to extend the other match, the Ω function oracles in $O(1)$ time either $pLPF_<[before_>[i]]$ or $pLPF_>[before_<[i]]$ – the values of these entries are dependent on the individual p-string. In order to get a bound on the work required for Λ to extend matches in this case, assume that $pLPF_<[before_>[i]] = pLPF_>[before_<[i]] = 0$. In other words, these set values will force additional work by Λ since nonzero entries provide symbols for future matches and in turn, require less work from Λ. Let m be the length of the longest p-match between any p-suffix and its respective $before_<$ or $before_>$ element or more formally, $m = \max\{0, t \mid t = \max\{r, s \mid \texttt{prev}(T[before_<[e] \ldots n]) =_r \texttt{prev}(T[e \ldots n]), \texttt{prev}(T[before_>[e] \ldots n]) =_s \texttt{prev}(T[e \ldots n])\ \forall\ 1 \leq e \leq n \text{ with } before_<[e] \geq 1, before_>[e] \geq 1\}\}$. Under the previous conditions, the Λ function will need to perform $O(m)$ work for a total of γ times in the worst case requiring $O(m\gamma)$ time, where γ is the number of times that the p-string T forces either case (c), (d), (e), or (f) of Fig. 1, i.e. γ is a count of the number of times $(R[before_<[b - 1] + 1] < R[b] \wedge R[before_>[b - 1] + 1] < R[b]) \vee (R[b] < R[before_<[b-1]+1] \wedge R[b] < R[before_>[b-1]+1])\ \forall\ 2 \leq b \leq n$. Recall that earlier in the proof, already $pLPF[2] = n-2$ is processed in $O(n)$ time. Thus, $O(n + m\gamma)$ time is required overall. Moreover, the time required by Λ to interleave p-matching with other possible cases (a) and (b) of Fig. 1 is clearly absorbed in this time complexity.

By combining the cases, it follows that Algorithm `compute_pLPF` executes in $O(n + m\gamma)$ time. Depending on the p-string, the actual value of $m\gamma$ may be the same as n (i.e. $m\gamma = n$), less than n (i.e. $m\gamma < n$ or $m\gamma << n$), or greater than n (i.e. $m\gamma > n$ or $m\gamma >> n$). Therefore, `compute_pLPF` executes in $O(\max\{n, m\gamma\})$ time. □

The difference in the time complexities between `compute_LPF` and `compute_pLPF` is due to the structure of p-strings. Even though the worst case time of the `compute_pLPF` algorithm is bounded by $O(\max\{n, m\gamma\})$, the expected running time will be in $O(n)$. This is because the actual values of m and γ are not independent and rather, they share a special relationship in terms of tradeoffs. Note that we do not need to actually compute m and γ in order to execute the algorithm. Instead, these values are simply used to indicate the bound on the time required to execute the algorithm. Consider the n-length p-string T. If $T[i] \in \Sigma\ \forall\ 1 \leq i < n$ and $T[n] = \$$, then T is a traditional string and thus, $\gamma = 0$. In this case, the worst case time complexity is $O(n)$. In addition, let p be the number of $T[i] \in \Pi$. If p is small (i.e. $p << n$), then also, the time complexity is $O(n)$. In the absolute worst case, suppose that it is possible that the values

Algorithm 3. pLCP computation.

```
1  int[] compute_pLCP(int before_<[],int after_<[],int R[]){
2     int pLCP[n], X[n], Y[n], i
3     X = compute_pLPF(before_<,null,R)
4     Y = compute_pLPF(after_<,null,R)
5     for i = 1 to n
6        pLCP[R[i]] = max{X[i],Y[i]}
7     return pLCP
8  }
```

of p, m, and γ are all large, i.e. $p \approx m \approx \gamma = O(n)$. What if this is the case? Consider two p-suffixes at c and d where $c < d$, $\texttt{prev}(T[c \ldots n]) \prec \texttt{prev}(T[d \ldots n])$, and $\texttt{prev}(T[c \ldots n]) =_{\frac{n}{c}} \texttt{prev}(T[d \ldots n])$ with $\frac{n}{c}$ as an integer and c as a constant. In this case, let $T[c + \frac{n}{c} - 1] = T[c + \frac{n}{c}] = \pi_1 \in \Pi \Rightarrow \texttt{prev}(T[c \ldots n])[\frac{n}{c} + 1] = 1$ and let $T[d] = T[d + \frac{n}{c}] = \pi_2 \in \Pi$ where $T[q] \neq \pi_2 \, \forall \, d < q < d + \frac{n}{c} \Rightarrow \texttt{prev}(T[d \ldots n])[\frac{n}{c} + 1] = \frac{n}{c}$. Now, when we consider the ordering between the p-suffixes $\texttt{prev}(T[c+1 \ldots n])$ and $\texttt{prev}(T[d+1 \ldots n])$, the lexicographical ordering changes to $\texttt{prev}(T[d+1 \ldots n]) \prec \texttt{prev}(T[c + 1 \ldots n])$ because $\texttt{prev}(T[d + 1 \ldots n]) =_{\frac{n}{c}-1} \texttt{prev}(T[c + 1 \ldots n])$ and $(\texttt{prev}(T[d + 1 \ldots n])[\frac{n}{c}] = 0) < (\texttt{prev}(T[c + 1 \ldots n])[\frac{n}{c}] = 1)$. These p-suffixes still share a significant number of symbols and their respective lexicographical orderings have been altered. However, the p-suffixes will not lexicographically change again until we process the p-suffix starting at $c + \frac{n}{c}$, which requires handling p-suffixes starting beyond the several symbols already matched. Should there be lexicographically closer p-suffixes than $c + 1$ and $d + 1$, the additional matches will yield *even fewer* changes in the lexicographical orderings. In this situation, it is shown that *significant* matches (large m) yield an *insignificant* number of lexicographical changes (small γ). Hence, there is a tradeoff between m and γ that allows the $O(n)$ time bound to absorb any instances where additional matching is required. The following corollary formalizes this expectation.

Corollary 17. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* prev *encoding of T, and pSA, the parameterized suffix array for T, the pLPF array can be constructed in $O(n)$ expected time.*

5. From pLPF to pLCP

Deguchi et al. [10,11] studied the problem of constructing the *pLCP* array given the *pSA*. They showed that constructing the *pLCP* array requires a non-trivial modification of the traditional *LCP* construction by Kasai et al. [29]. In [2], the *LCP* array was used as the basis for constructing the *LPF* array for traditional strings. Here, we present a simpler algorithm for constructing the *pLCP* array. In particular, we show that, unlike in [2], it is possible to go the other way around: that is, given the pLPF solution, we now construct the *pLCP* array. Later, we show that the same pLPF algorithm can be used to construct the *LCP* array and the *LPF* array for traditional strings.

Recall that Lemma 11 identifies that the suffixes of $\texttt{prev}(T)$ are not necessarily the p-suffixes of T. Also recall that Lemma 13 indicates that a tuple of p-suffixes can change their respective lexicographical ordering (i.e. their respective positions in the p-suffix array) when considering successive p-suffixes. These lemmas formalize the differences between traditional strings and p-strings. From the previous section, our compute_pLPF algorithm correctly addresses these added challenges. As a novel use of our compute_pLPF algorithm, we introduce a way to construct the *pLCP* array by modifying the parameters. The key observation is that we can exploit the fact that the *pLCP* occurs between neighboring p-suffixes by first preprocessing the $before_<$ array, which for each i in the p-string T maintains the p-suffix at $h < i$ positioned prior to the p-suffix at i in *pSA*. We can also construct the array $after_<$ to maintain the p-suffix at $j > i$ also positioned prior to the p-suffix at i in *pSA*. Since the p-suffixes at h and j are both positioned prior to the p-suffix at i in *pSA*, we can guarantee that either h or j must be the nearest neighbor to i. So, the maximum factor determines the nearest neighbor and thus, $pLCP[R[i]]$, where the rank array R is the inverse of *pSA* (see Algorithm 3). The following theorem formalizes this computation.

Theorem 18. *Given an n-length p-string T, $prevT = \texttt{prev}(T)$, the* prev *encoding of T, and pSA, the parameterized suffix array for T, the* compute_pLPF *algorithm can be used to construct the pLCP array in $O(\max\{n, m\phi\})$ time, where m is the length of the longest p-match between a p-suffix at i and two defined p-suffixes in T and ϕ is dependent on the lexicographical orderings of specified p-suffixes in T.*

Proof. We can clearly relax the p-suffix selection restrictions enforced by the problem pLPF to exploit the idea of *extending* factors as in Lemma 12. Subsequently, only the parameters of Algorithms 1, 2a, and 2b impose such restrictions. Let $R[1 \ldots n]$ be the rank array, the inverse of *pSA*. Let $before_<[1 \ldots n]$ and $after_<[1 \ldots n]$ maintain, for all the i in T, the p-suffixes at $h < i$ at position $R[h]$ in *pSA* and $j > i$ at position $R[j]$ in *pSA*, respectively, that are positioned prior to the p-suffix i at position $R[i]$ in *pSA*; when no such p-suffix exists, the element is denoted by -1. Let $X = \texttt{compute\_pLPF}(before_<, \textbf{null}, R)$ and $Y = \texttt{compute\_pLPF}(after_<, \textbf{null}, R)$. We prove that the *pLCP* is constructed by the statement $pLCP[R[i]] = \max\{X[i], Y[i]\}$. Without loss of generality, suppose that both h and j exist and $2 < i \leq n$, so we have either $R[h] = R[i] - 1$ or $R[j] = R[i] - 1$ as the neighboring p-suffix. When $x = \texttt{plcp}(\texttt{prev}(T[h \ldots n]), \texttt{prev}(T[i \ldots n]))$ and $y = \texttt{plcp}(\texttt{prev}(T[j \ldots n]), \texttt{prev}(T[i \ldots n]))$ then $\max\{x, y\}$ distinguishes which p-suffix at h or j is closer to i, identifying the nearest neighbor and in turn, $pLCP[R[i]]$. Since it is the case that $X[i] = x$ and $Y[i] = y$, then it follows that $pLCP[R[i]] = \max\{X[i], Y[i]\}$. We

Algorithm 4. Improved pLCP computation.

```
1  int[] compute_pLCP(int R[]) {
2     int M[n]={−1, ...,−1}, Q[n], i
3     for i = 1 to n
4        Q[R[i]] = i
5     for i = 2 to n
6        M[Q[i]] = Q[i−1]
7     M = compute_pLPF(M, null, R)
8     for i = 1 to n
9        Q[R[i]] = M[i]
10    return Q
11 }
```

have yet to prove the time complexity. The parameter $after_<$ can be computed in $O(n)$ time by deletions and indexing into a doubly linked list similar to $before_<$ from [13]. Further, the array X is computed in $O(\max\{n, m_1\gamma_1\})$ time and the array Y is computed in $O(\max\{n, m_2\gamma_2\})$ time by Theorem 16. Clearly, the rearranging of the results within X and Y into the *pLCP* array is performed in $O(n)$ time. So, $O(\max\{n, m_1\gamma_1, m_2\gamma_2\})$ time is required. Let $m = \max\{m_1, m_2\}$ and $\phi = \max\{\gamma_1, \gamma_2\}$. Overall, the time required is in $O(\max\{n, m\phi\})$. □

It readily follows that since the $m\phi$ term is a result of the `compute_pLPF` algorithm, the logical discussion leading to Corollary 17 also applies here with respect to the `compute_pLCP` algorithm. Hence, it is expected that `compute_pLCP` executes in $O(n)$ time.

Corollary 19. *Given an n-length p-string T, prevT* $=$ `prev`(T)*, the* `prev` *encoding of T, and pSA, the parameterized suffix array for T, the pLCP array can be constructed in O(n) expected time.*

For purposes of discussion, Algorithm 3 uses the preprocessed arrays $before_<$ and $after_<$ in calls to `compute_pLPF` to infer the number of symbols matching with the neighboring p-suffix by observing the temporary results. This process of finding the neighboring p-suffix can be found trivially with a p-suffix array, and thus, the novel use of *both* the $before_<$ and $after_<$ arrays may be omitted for practical space. This observation is the focus of the improved solution in Algorithm 4. It is possible to use *pSA* to identify the neighboring p-suffixes to yield one parameter: M. The call to `compute_pLPF`$(M,$ **null**$, R)$ is possible because Algorithms 1, 2a, and 2b support the case when the parameter $before_> =$ **null**. In the algorithm, the array Q is used as both an intermediate *pSA* (derived from R) and the resulting *pLCP* data structure. In passing, we identify that upon the completion of line 7 in Algorithm 4, the M array is the permuted longest common prefix (*PLCP*) data structure observed in [30] for traditional strings.

6. From pLPF to LPF and LCP

The power of defining the pLPF problem in terms of p-strings is the generalization of a p-string production. A useful property of p-strings is that a special case of the alphabet definitions or composition of symbols will yield a traditional string. If $|\Sigma| > 0 \wedge |\Pi| = 0$, then only traditional strings are valid p-string productions. Similarly, when all of the individual symbols σ of a p-string are such that $\sigma \in \Sigma$, this also yields a traditional string. Such generalization by the p-string allows us to offer solutions to multiple problems with a single algorithm based on p-strings. We show in Theorems 20 and 21 that our `compute_pLPF` algorithm also computes the traditional *LPF* and *LCP* arrays.

Theorem 20. *Given an n-length traditional string W, the* `compute_pLPF` *algorithm constructs the LPF array in O(n) time.*

Proof. Since $W[i] \in \Sigma\ \forall\, i,\ 1 \leq i < n$ and $W[n] \in \{\$\}$, then by Definition 1 we have $W \in (\Sigma \cup \Pi)^*\$$, which classifies W as a valid p-string. Since W consists of no such symbol $\pi \in \Pi$, then Theorem 16 proves that the construction of *pLPF* for the n-length p-string W requires $O(\max\{n, m\gamma\}) \in O(n)$ time with $\gamma = 0$ because Lemma 13 does not apply. Lemma 11 identifies that `prev`$(W[i \ldots n])$ $=$ `prev`$(W)[i \ldots n]$ and further $W =$ `prev`(W) by Definition 3, so $W[i \ldots n] =$ `prev`$(W)[i \ldots n]$, which constrains the pLPF in Definition 10 to the LPF problem in Definition 9. Thus, `compute_pLPF` computes the LPF of W in $O(n)$ time. □

Theorem 21. *Given an n-length traditional string W, the* `compute_pLCP` *algorithm constructs the LCP array in O(n) time.*

Proof. In the same manner as Theorem 20, we may classify W as a valid p-string. Given this, Theorem 18 proves that the construction of *pLCP* for the n-length p-string W requires $O(\max\{n, m\phi\}) \in O(n)$ time with $\phi = 0$ because Lemma 13 does not apply. Mirroring the proof of Theorem 20, we have $W[i \ldots n] =$ `prev`$(W)[i \ldots n]$, which constrains the pLCP in Definition 7 to the traditional LCP problem. Thus, `compute_pLCP` computes the LCP of W in $O(n)$ time. □

7. Analysis

The algorithms presented in this work operate on an n-length p-string and theoretically demand $O(n)$ time and $O(n)$ space. Since our algorithms are developed for the p-string and improve the theoretical complexity for the construction

Table 3
Comparison of algorithms for a text T of length n where v is chosen and δ is determined based on application of text.

Structure	Algorithm	Time [*1]	Space [*2]	Note
		traditional strings		
LCP	Lcp6 by Manzini [31]	$O(n)$	$(6+\delta)n+o(n)$	overwrites *SA*
LCP	space–time tradeoff by Puglisi et al. [32]	$O(nv)$	$5n + O(n/\sqrt{v})$	–
LCP	Lcp9 by Manzini [31]	$O(n)$	$9n + o(n)$	saves *SA*
LCP	`GetHeight` by Kasai et al. [29]	$O(n)$	$13n + o(n)$	–
LCP	`P-Kasai` by Tomohiro I et al. [10]	$O(n^2)$	$14n + o(n)$ [*3]	–
LCP	`compute_pLCP` (Algorithm 4)	$O(n)$	$17n + o(n)$ [*3]	–
LPF	`compute_LPF` by Crochemore et al. [2]	$O(n)$	$21n + o(n)$	first *SA* solution
LPF	`compute_LPF` by Crochemore et al. [24]	$O(n)$	$12n + o(n)$	–
LPF	`LPF-OPTIMAL` by Crochemore et al. [28]	$O(n)$	$12n + o(1)$	–
LPF	`compute_pLPF` (Algorithm 1)	$O(n)$	$21n + o(n)$ [*3]	also constructs *pLPF*, *pLCP*, and *LCP*
		p-strings		
pLCP	`P-Kasai` by Tomohiro I et al. [10]	$O(n^2)$	$14n + o(n)$ [*3] $20n + o(n)$ [*4]	first solution
pLCP	`compute_pLCP` (Algorithm 4)	$O(n)$ [*5]	$17n + o(n)$ [*3] $20n + o(n)$ [*4]	–
pLPF	`compute_pLPF` (Algorithm 1)	$O(n)$ [*6]	$21n + o(n)$ [*3] $24n + o(n)$ [*4]	only solution

[*1] for purposes of comparison, assume that $|\Sigma|$ and $|\Pi|$ are constant
[*2] claimed space usage in bytes or analysis of the most bytes needed by algorithm during execution, assuming `sizeof`(**int**) $= 4$ bytes and `sizeof`(**char**) $= 1$ byte; preprocessing excluded
[*3] since (1) in the worst case for traditional strings or (2) on average for p-strings, a `prev` encoding occupies a **char** array
[*4] since in the worst case for p-strings, a `prev` encoding occupies an **int** array
[*5] the expected running time (Corollary 19) or, more specifically, $O(\max\{n, m\phi\})$ based on the text
[*6] the expected running time (Corollary 17) or, more specifically, $O(\max\{n, m\gamma\})$ based on the text

time of p-string data structures, we choose to analyze the practical space used by our solutions. We aim to show that even though we are addressing p-string problems by exploiting the utility of a single algorithm `compute_pLPF`, our algorithms are competitive, and thus represent practical solutions for constructing data structures for both p-strings *and* traditional strings.

Algorithms that work with p-strings, to solve the p-match problem, use the `prev` encoding to absorb the original text and naturally avoid the housekeeping of parameters. The `prev` encoding, as defined in Definition 3, encodes constant symbols with the same symbol and encodes parameters with numbers that identify the distance to the previous occurrence of that parameter in the p-string. The question becomes, is it sufficient to encode an n-length `prev` array with 1-byte **char** variables or 4-byte **int** variables? In reality, this is dependent on the application or more specifically, the maximum distance reported in the `prev` encoding. Suppose that we are working with the ASCII character set and every $\sigma \in \Sigma$ and $\pi \in \Pi$ is a printable character, including the $\mathbb{V} = 95$ characters in the decimal range $[32 - 126]$ or $[1 - 95]$ after applying an offset $\mathbb{F} = -31$. Following the ordering scheme of Definition 4, we obtain the following scheme given that each element of `prev` uses an **unsigned char**: $0 \rightarrow \$, 1 \rightarrow SPACE, 2 \rightarrow !, \ldots, 34 \rightarrow A, 35 \rightarrow B, \ldots, 95 \rightarrow \tilde{}, 96 \rightarrow d_0, 97 \rightarrow d_1, \ldots, 255 \rightarrow d_{159}$, where d_x denotes the parameter encoded distance x. In the average case, if each of the $\mathbb{V}$ printable characters occurs uniformly, then the maximum numeric distance encoded by `prev` is $\mathbb{V}$, which is clearly represented by the d_{159} of an **unsigned char** since $159 > \mathbb{V}$. Thus, an **unsigned char** array can represent the `prev` data structure on average for p-strings. When operating with traditional strings, `prev` can be represented with a **char** in the worst case since only the terminal symbol and the $\mathbb{V}$ constant symbols need to be encoded. In the case that the symbols are not uniformly distributed and the parameter π only occurs in the p-string T at positions $T[1]$ and $T[2^{32} - \mathbb{V} - 1]$, then an **unsigned int** array will suffice to represent the `prev` encoding. Any other assumptions about the encoded numeric distances in `prev` will vary the amount of space allocated.

We compile a comparative list of solutions in Table 3 that construct the *LCP*, *LPF*, *pLCP* and *pLPF* data structures. In the table, the time complexities are displayed as theoretically identified in the corresponding papers. Unless the actual space analysis was provided in the respective papers, we analyzed the provided algorithms, isolated from any preprocessing, in terms of the most space required during an instance of execution of the algorithm. It is evident from Table 3 that our algorithms are superior for problems with limited solutions such as *pLCP* processing and also, practically competitive for constructing the *LCP* and *LPF* arrays. More specifically, the `compute_pLPF` algorithm for the *pLPF* and *LPF* arrays requires at most $2n$ **int** elements of storage for the parameters $before_<$ and $before_>$, $2n$ **int** elements for saving previous matches in

$pLPF_<$ and $pLPF_>$, n **int** elements for the rank array R, n **int** elements for the resulting *pLPF* data structure, and n elements for the `prev` encoding that absorbs the original text. While Algorithm 1 uses various arrays for readability and ease of future discussion, we acknowledge that it is possible to save an array in the algorithm by populating $pLPF[i]$ after the main loop, which will allow us to reuse $pLPF_<$ for the resulting *pLPF* array. (On a similar note, when $before_> =$**null** no memory allocation is needed for $pLPF_>$.) Thus, `compute_pLPF` requires either $6n$ **int** elements or $5n$ **int** and n **char** elements on average. This algorithm is the only solution for constructing the *pLPF* data structure and the complexities are similar to that of the original `compute_LPF` in [2].

We also show how to exploit our `compute_pLPF` algorithm to construct the *LCP* and *pLCP* arrays. Since we provide construction algorithms for several data structures using the *pLPF* construction as the groundwork, we are faced with the practical limitation that our algorithms are only as efficient as the `compute_pLPF` solution. The `compute_pLCP` (Algorithm 4) computes the *pLCP* and *LCP* data structures in expected linear time. Its space requirements are best analyzed by comparing with the previous space analysis of `compute_pLPF`. The only differences are that (1) we use an additional **int** array Q as an intermediate *pSA* and resulting *pLCP* array and (2) we use only one **int** parameter M, which saves the need to allocate the two **int** arrays $before_>$ and $pLPF_>$. This yields a savings of n **int** elements over the space required by `compute_pLPF`. Thus, in the worst case for p-strings, $5n$ **int** elements are required whereas on average for p-strings and in the worst case for traditional strings, $4n$ **int** elements are required in addition to n **char** elements for the `prev` encoding. For the *pLCP*, our `compute_pLCP` algorithm is an improvement in terms of time complexity when compared to the claimed $O(n^2)$ result of the `P-Kasai` [10] algorithm and in terms of practical space, our solution is competitive. In general, we handle the dynamically changing p-suffixes differently than `P-Kasai` to save space and we use extra space to retain previous matches in order to more quickly extend future matches.

In the context of this work, we chose to maximize the properties of pLPF and the utility of `compute_pLPF` to construct many data structures with one algorithm, instead of refining different algorithms for each data structure. We acknowledge the possibility of reworking the `compute_pLPF` algorithm to incorporate the *LCP* indexing contributions of [31] to further improve the memory footprint of the algorithm. In passing, we note the possibility to improve the space consumption of the *LCP* array by utilizing compression and other variants of the data structure as proposed in [30,32,33], since *pLCP* is also an array of integers analogous to the traditional *LCP*. Also, we acknowledge the future research problem of integrating our *LPF*, *LCP*, *pLPF* and *pLCP* solutions with the contributions in [28].

8. Conclusion

We introduce the parameterized longest previous factor (pLPF) problem for p-strings, which is analogous to the longest previous factor (LPF) problem defined for traditional strings. An expected linear time algorithm is provided to construct the *pLPF* array for a given p-string. The advantage of implementing our solution `compute_pLPF` is that the algorithm may be used to compute the p-string related arrays *pLPF* and *pLCP* in an expected time linear to the length of the given p-string and the traditional arrays *LPF*, *LCP*, and even the permuted *LCP* [30] in linear time. These are fundamental data structures preprocessed for the efficiency of countless pattern matching applications. The significance of working though the *LPF* as an intermediate data structure is the straightforward and space efficient algorithm to construct the Lempel–Ziv (LZ) factorization based on the *LPF* structure [2,24]. Similarly, the *pLPF* array can easily derive the *LZ* data structure and allow us to study such applications as maximal runs in p-strings extended to source code plagiarism or redundancies in biological sequences.

Acknowledgements

We express our sincere gratitude to the reviewers for their critical comments and suggestions that have led to improvements within the paper.

References

[1] R. Beal, D. Adjeroh, Parameterized longest previous factor, in: IWOCA'11, Springer, Heidelberg, 2011, pp. 31–43.
[2] M. Crochemore, L. Ilie, Computing longest previous factor in linear time and applications, Inf. Process. Lett. 106 (2) (2008) 75–80.
[3] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory. 23 (3) (1977) 337–343.
[4] M. Main, Detecting leftmost maximal periodicities, Discrete Appl. Math. 25 (1–2) (1989) 145–153.
[5] B. Baker, A theory of parameterized pattern matching: Algorithms and applications, in: STOC'93, ACM, New York, 1993, pp. 71–80.
[6] A. Amir, M. Farach, S. Muthukrishnan, Alphabet dependence in parameterized matching, Inf. Process. Lett. 49 (1994) 111–115.
[7] T. Shibuya, Generalization of a suffix tree for RNA structural pattern matching, Algorithmica. 39 (1) (2004) 1–19.
[8] B. Baker, Finding clones with dup: Analysis of an experiment, IEEE Trans. Software Eng. 33 (9) (2007) 608–621.
[9] B. Zeidman, Software v. Software, IEEE Spectr. 47 (2010) 32–53.
[10] T. I, S. Deguchi, H. Bannai, S. Inenaga, M. Takeda, Lightweight parameterized suffix array construction, in: IWOCA'09, in: LNCS, vol. 5874, Springer, Heidelberg, 2009, pp. 312–323.
[11] S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, M. Takeda, Parameterized suffix arrays for binary strings, in: PSC'08, Czech Republic, 2008, pp. 84–94.
[12] R. Beal, D. Adjeroh, p-Suffix sorting as arithmetic coding, in: IWOCA'11, Springer, Heidelberg, 2011, pp. 44–56.
[13] R. Beal, Parameterized Strings: Algorithms and Data Structures, MS Thesis, West Virginia University, 2011.
[14] U. Manber, G. Myers, Suffix arrays: A new method for on-line string searches, SIAM J. Comput. 22 (1993) 935–948.
[15] S. Kosaraju, Faster algorithms for the construction of parameterized suffix trees, in: FOCS'95, ACM, Washington, DC, 1995, pp. 631–637.

[16] R. Cole, R. Hariharan, Faster suffix tree construction with missing suffix links, SIAM J. Comput. 33 (1) (2003) 26–42.
[17] T. Lee, J. Na, K. Park, On-line construction of parameterized suffix trees for large alphabets, Inf. Process. Lett. 111 (5) (2011) 201–207.
[18] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, UK, 1997.
[19] W. Smyth, Computing Patterns in Strings, Pearson, New York, 2003.
[20] D. Adjeroh, T. Bell, A. Mukherjee, The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching, Springer, New York, 2008.
[21] B. Baker, Parameterized pattern matching by Boyer-Moore-type algorithms, in: SODA'95, ACM, Philadelphia, PA, 1995, pp. 541–550.
[22] R. Idury, A. Schäffer, Multiple matching of parameterized patterns, Theor. Comput. Sci. 154 (1996) 203–224.
[23] A.V. Aho, M.J. Corasick, Efficient string matching: An aid to bibliographic search, Commun. ACM 18 (1975) 333–340.
[24] M. Crochemore, L. Ilie, W. Smyth, A simple algorithm for computing the Lempel Ziv factorization, in: DCC'08, 2008, pp. 482–488.
[25] M. Crochemore, G. Tischler, Computing longest previous non-overlapping factors, Inf. Process. Lett. 111 (6) (2011) 291–295.
[26] S. Chairungsee, M. Crochemore, Efficient computing of longest previous reverse factors, in: CSIT'09, Yerevan, Armenia, 2009, pp. 27–30.
[27] M. Crochemore, C. Iliopoulos, M. Kubica, W. Rytter, T. Waleń, Efficient algorithms for two extensions of LPF table: The power of suffix arrays, in: SOFSEM'10, Springer, Heidelberg, 2009, pp. 296–307.
[28] M. Crochemore, L. Ilie, C.S. Iliopoulos, M. Kubica, W. Rytter, T. Walen, LPF computation revisited, in: IWOCA'09, Springer, Heidelberg, 2009, pp. 158–169.
[29] T. Kasai, G. Lee, et al., Linear-time longest-common-prefix computation in suffix arrays and its applications, in: CPM'01, in: LNCS, vol. 2089, 2001, pp. 181–192.
[30] J. Kärkkäinen, G. Manzini, S. Puglisi, Permuted longest-common-prefix array, in: CPM'09, Springer, Heidelberg, 2009, pp. 181–192.
[31] G. Manzini, Two space saving tricks for linear time LCP array computation, in: SWAT'04, in: LNCS, vol. 3111, 2004, pp. 372–383.
[32] S. Puglisi, A. Turpin, Space–time tradeoffs for longest-common-prefix array computation, in: ISAAC'08, Springer, Heidelberg, 2008, pp. 124–135.
[33] J. Fischer, Wee LCP. Inf. Process. Lett. 110 (8-9) (2010) 317–320.