pBWT: Achieving Succinct Data Structures for Parameterized Pattern Matching and Related Problems*

Arnab Ganguly[†] Rahul Shah[‡] Sharma V. Thankachan[§]

Abstract

The fields of succinct data structures and compressed text indexing have seen quite a bit of progress over the last two decades. An important achievement, primarily using techniques based on the Burrows-Wheeler Transform (BWT), was obtaining the full functionality of the suffix tree in the optimal number of bits. A crucial property that allows the use of BWT for designing compressed indexes is *order-preserving suffix links*. Specifically, the relative order between two suffixes in the subtree of an internal node is same as that of the suffixes obtained by truncating the first character of the two suffixes. Unfortunately, in many variants of the text-indexing problem, for e.g., parameterized pattern matching, 2D pattern matching, and order-isomorphic pattern matching, this property does not hold. Consequently, the compressed indexes based on BWT do not directly apply. Furthermore, a compressed index for any of these variants has been elusive throughout the advancement of the field of succinct data structures. We achieve a positive breakthrough on one such problem, namely the *Parameterized Pattern Matching* problem.

Let T be a text that contains n characters from an alphabet Σ, which is the union of two disjoint sets: Σ_s containing static characters (s-characters) and Σ_p containing parameterized characters (p-characters). A pattern P (also over Σ) matches an equal-length substring S of T iff the s-characters match exactly, and there exists a one-to-one function that renames the p-characters in S to that in P. The task is to find the starting positions (occurrences) of all such substrings S. Previous index [Baker, STOC 1993], known as *Parameterized Suffix Tree*, requires $\Theta(n \log n)$ bits of space, and can find all *occ* occurrences in time $O(|P| \log \sigma + occ)$, where $\sigma = |\Sigma|$. We introduce an $n \log \sigma + O(n)$-bit index with $O(|P| \log \sigma + occ \cdot \log n \log \sigma)$ query time. At the core, lies a new BWT-like transform, which we call the *Parameterized Burrows-Wheeler Transform* (pBWT). The techniques are extended to obtain a succinct index for the *Parameterized Dictionary Matching* problem of Idury and Schäffer [CPM, 1994].

1 Introduction

Pattern matching is a fundamental problem in Computer Science with applications in web-data, texts and biological sequences. In the data structural sense, the text T (of n characters) is pre-processed and an index is built to answer pattern matching queries for a pattern P. Both text and pattern come from alphabet set Σ of size σ. In the basic pattern matching query, all *occ* occurrences of P in T, identified by their location in T, are reported. Suffix trees [47] are the most powerful and ubiquitous data structures for this purpose. According to Gusfield's book [24], they find myriad applications in sequence analysis for many different applications. Broadly speaking, there are two kinds of applications: (1) where we use augmenting data or arrays on top of the suffix tree [37] and (2) where a variant of suffix tree is required [5, 11, 20, 31, 17, 46].

In the era of budget, one of the negative aspects of suffix tree was seen to be its space utilization – about 50 times the text for DNA sequences. In theoretical sense, although considered linear in terms of words, the suffix trees take $\Theta(n \log n)$ space in terms of bit. However, the optimal is $n \log \sigma$ bits, leading to a complexity gap. The advent of succinct data structures and compressed text indexing, where the goal is to have data structure in the space equal to the information theoretical minimum, presented us with new indexes like Compressed Suffix Array (CSA) [23] and FM-Index [16], and eventually leading to a wonderful data structure called fully-functional compressed suffix tree (CST) [43, 45]. In practical sense, these achieved remarkable breakthroughs by saving orders of magnitude of space. After the introduction of CST, it could be used as a black box to replace suffix tree. In more advanced applications, one research line was to compress the augmenting data and achieve succinct results. This found considerable

*Arnab Ganguly was partially supported by National Science Foundation (NSF) Grants CCF–1218904 and CCF–1527435, and a Louisiana State University (LSU) Dissertation Fellowship.

[†]School of EECS at LSU. Email: ju.arnab@gmail.com

[‡]School of EECS at LSU, and NSF. Email: rahul@csc.lsu.edu, rahul@nsf.gov

[§]Department of CS at the University of Central Florida (UCF). Email: sharma.thankachan@ucf.edu

success [37, 39]. However, the applications where variants of suffix trees are required (especially, one which do not follow some crucial structural properties of suffix trees) there has not been any significant (if any) progress in achieving succinct/compressed indexes.

An important ingredient of suffix trees, crucial to compressed text indexing, is *suffix links*. In suffix trees, the leaves are arranged in the lexicographic order of the suffix they represent. Suffix links have the following *order-preserving* property. Consider two non-root internal nodes u and v. The leaves obtained by following suffix links from the leaves in u's subtree appear in the same relative order in the subtree of v. Thus, the permutation of the suffixes in v's subtree can be encoded in terms of the permutation in u's subtree. In applications like p-suffix tree [5], 2D suffix tree [20, 31], structural suffix tree [46] etc., this property does not necessarily hold. This brings in new challenges in how to encode such permutations, and even 15 years after the introduction of the CSA and the FM-Index, developing a succinct index for these classes of problem has been elusive. Also, it has been largely unknown whether succinct data structures are even possible. We achieve a positive breakthrough on one such problem, popularly known as *Parameterized Pattern Matching*.

1.1 Contribution

Introduced by Baker [5], we now formally define the parameterized pattern matching problem. The alphabet Σ is the union of two disjoint sets: Σ_s having σ_s static characters (s-characters) and Σ_p having σ_p parameterized characters (p-characters). Two strings are a *parameterized match* (p-match) if one can be transformed to the other by applying a one-to-one function that renames the p-characters.

PROBLEM 1. ([5]) *Let* T *be a text having* n *characters from* Σ. *Assume that* T *terminates in an s-character* \$ *that appears only once. The task is to index* T, *such that for a pattern* P *(over* Σ*), we can report the start positions (occurrences) of all the substrings of* T *that are a p-match with* P.

Example. Let $\Sigma_s = \{A, B, C, \$\}$ and $\Sigma_p = \{w, x, y, z\}$. Then, $P = AxByCx$ p-matches within $\mathsf{T}[1, 21] = AyBxCyAwBxCzxyAzBwCz\$$ at positions 1 and 15. At position 1, the mapping is $x \to y$ and $y \to x$; whereas at position 15, the mapping is $x \to z$ and $y \to w$. Note that P does not match at position 7 because x would have to match with both w and z.

Baker [5] presented an index known as *Parameterized Suffix Tree* (p-suffix tree) that uses $\Theta(n \log n)$ bits. It can count the number of occurrences in $O(|P| \log \sigma)$ time, and then report each occurrence in $O(1)$ time. Following is our main contribution.

THEOREM 1. *By using an* $n \log \sigma + O(n)$*-bit self-index, we can count the number of p-matches of a pattern* P *in* $\mathsf{T}[1, n]$ *in* $O(|P| \log \sigma)$ *time. Subsequently, each match can be reported in* $O(\log \sigma \log n)$ *time.*

At the core of our index, lies a *new* BWT-like transform for a parameterized text, called the *Parameterized* BWT. Using this, we handle the *order-inversion* in the case of p-suffixes when their first characters are truncated. To achieve this, we implement analogous versions of the last-to-first column mapping of Ferragina and Manzini [16] using newly introduced concepts coupled with existing succinct data structure toolkit.

The orthogonal problem to text indexing is the so-called *Dictionary Matching* problem [2, 9, 26, 27]. The task is to index multiple patterns and given a text, find the positions having at least one occurrence of a pattern. Idury and Schäffer [29] considered the following variant known as *Parameterized Dictionary Matching*.

PROBLEM 2. ([29]) *Let* $\mathcal{D}$ *be a collection of* d *patterns* $\{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_d\}$ *of total length* n *characters that are chosen from* Σ. *The task is to index* $\mathcal{D}$, *such that given a text* T *(also over* Σ*), we can report all pairs* $\langle j, \mathsf{P}_i \rangle$ *i.e., a position* j *and a pattern* $\mathsf{P}_i \in \mathcal{D}$ *which is a p-match with* $T[j - |\mathsf{P}_i| + 1, j]$.

Largely based on the Aho-Corasick (AC) automaton [2], Idury and Schäffer presented an $\Theta(m \log m)$ bit index, where $m \leq n+1$ is the number of states in the automaton, that can report all *occ* pairs in time $O(|T| \log \sigma + occ)$. Recently, Ganguly et al. [18] presented an $O(n \log \sigma + d \log n)$-bit index with $O(|T|(\log \sigma + \log_\sigma n) + occ)$ query time (see [19] for its dynamic version). By largely reusing the index for proving Theorem 1, coupled with a transform that closely resembles the XBWT of Ferragina et al. [15], we prove the following theorem which improves the existing results [18, 29].

THEOREM 2. *All occ pairs* $\langle j, \mathsf{P}_i \rangle$*, such that a pattern* $\mathsf{P}_i \in \mathcal{D}$ *is a p-match with* $T[j - |\mathsf{P}_i| + 1, j]$*, can be found in* $O(|T| \log \sigma + occ)$ *time using an* $m \log \sigma + O(m + d \log(m/d))$*-bit index.*

1.2 Applications and Related Work

The main motivation behind Problem 1 is software plagiarism/clone detection. Baker [6] presented the role of the problem and the efficiency of p-suffix trees using a program called *Dup*. Subsequently, the methodology became an integral part of various tools for software version management and clone detection, where identifiers and/or literals are renamed. Typically, these are referred to as Type 2 clones in the literature. (See [33, 41, 42] for well-cited surveys on this topic.) Although there are different methodologies available, use of p-suffix trees to detect Type-2 clones has proven useful [8, 34, 44]. These typically use a hybrid approach, such as a combination of (i) a parse tree, which converts literals into parameterized symbols, and (ii) a p-suffix tree on top of these symbols. Unfortunately, as with traditional suffix trees, the space occupied by p-suffix trees is too large for most practical purposes. In fact, one of the available tools (CLICS [1]) very clearly acknowledges that the major space consumption is due to the use of suffix tree over parameterized symbols. This inhibits the tool to be used for large software repositories. Some other tools [28] use more IR type methodology for indexing repositories based on variants of the inverted index. Although less space consuming, there are no theoretical guarantees possible on query-times in such indexes. Following are a few other works that have used p-suffix trees: finding relevant information based on regular expressions in sequence databases [13, 14], detecting cloned web pages [12], detecting similarities in JAVA sources from bytecodes [7], etc.

On the theoretical side, parameterized pattern matching has seen constant development since its inception by Baker [5] in 1993. In one direction, the focus was to design fast construction algorithms of p-suffix trees [10, 32]. Other works [4, 25, 30] include addressing variants such as p-matching in the streaming model and approximate p-matching. Further generalizations of p-matching have also played an important role in computational biology for finding similar sequences [3, 46]. We refer the reader to [35, 36] for recent surveys.

1.3 Map

In Section 2, we take a close look at the parameterized suffix tree of Baker [5] as it plays a crucial role in the proposed index. The details of the parameterized BWT, its accompanying last-to-first column mapping implementation, and the adaptation of the backward-search methodology are presented in Sections 3, 4, and 5 respectively. Section 6 presents a succinct index for Problem 2. We conclude the paper in Section 7.

2 Parameterized Suffix Tree

Throughout this paper, we use the following terminologies: for a string S, $|S|$ is its length, $S[i], 1 \leq i \leq |S|$, is its ith character and $S[i,j] = S[i] \circ S[i+1] \circ \cdots \circ S[j]$, where $\circ$ denotes *concatenation*. If $i > j$, $S[i,j]$ denotes an empty string. Also S_i denotes the circular suffix starting at position i. Specifically, S_i is S if $i = 1$ and is $S[i, |S|] \circ S[1, i-1]$ otherwise.

Baker [5] introduced the following encoding scheme for matching strings over $\Sigma = \Sigma_s \cup \Sigma_p$. A string S is encoded into a string $\mathsf{prev}(S)$ of length $|S|$ by replacing the first occurrence of every p-character in S by 0 and any other occurrence of a p-character by the difference in text position from its previous occurrence. Specifically, for any $i \in [1, |S|]$, $\mathsf{prev}(S)[i] = S[i]$ if $S[i]$ is an s-character; otherwise, $\mathsf{prev}(S)[i] = (i - j)$, where $j < i$ is the last occurrence of $S[i]$ before i. If j does not exist, then $j = i$. For example, $\mathsf{prev}(AxByBx) = A0B0B4$, where $A, B \in \Sigma_s$ and $x, y \in \Sigma_p$. Note that $\mathsf{prev}(S)$ is a string over $\Sigma' = \Sigma_s \cup \{0, 1, \ldots, |S|-1\}$, and can be computed in time $O(|S| \log \sigma)$.

Fact 1. ([5]) *Two strings S and S' are a p-match iff $\mathsf{prev}(S) = \mathsf{prev}(S')$. Also S and a prefix of S' are a p-match iff $\mathsf{prev}(S)$ is a prefix of $\mathsf{prev}(S')$.*

Moving forward, we follow the convention below.

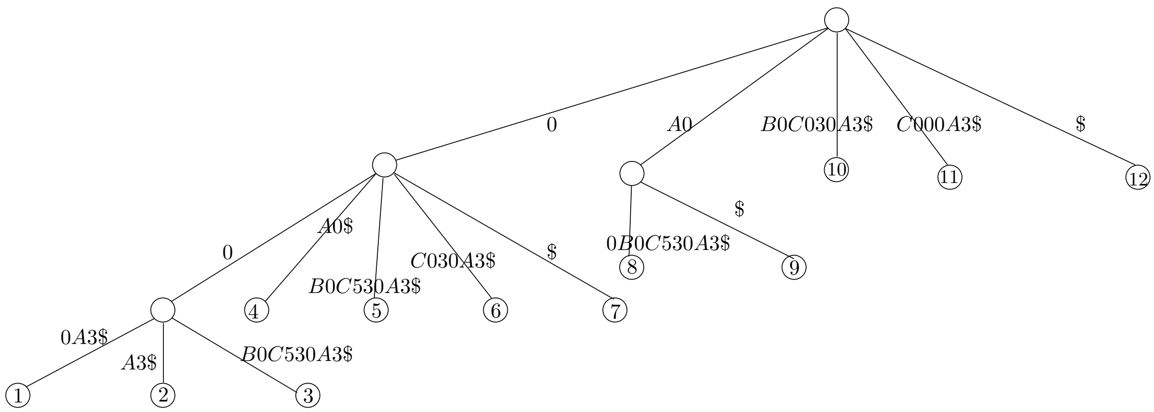
Convention 1. *In Σ', the integer characters (corresponding to p-characters) are lexicographically smaller than s-characters. An integer character i comes before another integer character j iff $i < j$. Also, $ is lexicographically larger than all other characters.*

Parameterized Suffix Tree (pST) is the compacted trie of all strings in $\mathcal{P} = \{\mathsf{prev}(\mathsf{T}[k, n]) \mid 1 \leq k \leq n\}$. Each edge is labeled with a string over Σ'. We use $\mathsf{path}(u)$ to denote the concatenation of edge labels on the path from root to node u. Clearly, pST consists of n leaves (one per each encoded suffix) and at most $n-1$ internal nodes. The space required is $\Theta(n \log n)$ bits. See Figure 1 for an illustration. The path of each leaf node corresponds to the encoding of a unique suffix of T, and leaves are ordered in the lexicographic order of the corresponding encoded suffix.

To find all the occurrences of P, traverse the pST from root by following the edge labels and find the highest node u (called *locus*) such that $\mathsf{path}(u)$ is prefixed by $\mathsf{prev}(P)$. Then find the range $[\mathsf{sp}, \mathsf{ep}]$ (called *suffix range* of $\mathsf{prev}(P)$) of leaves in the subtree of u and report $\{\mathsf{pSA}[i] \mid \mathsf{sp} \leq i \leq \mathsf{ep}\}$ as the output. Here, $\mathsf{pSA}[1, n]$ is the parameterized suffix array i.e., $\mathsf{pSA}[i] = j$ and $\mathsf{pSA}^{-1}[j] = i$ iff $\mathsf{prev}(\mathsf{T}[j, n])$ is the ith lexicographically smallest string in $\mathcal{P}$. (Note that $\mathsf{path}(\ell_i) = \mathsf{prev}(\mathsf{T}[\mathsf{pSA}[i], n])$, where ℓ_i is the ith leftmost leaf in pST.) The query time is $O(|P| \log \sigma + occ)$.

3 Parameterized Burrows-Wheeler Transform

We introduce a similar transform to that of the BWT, which we call the Parameterized Burrows-Wheeler Transform (pBWT). To obtain the pBWT of T, we first create a matrix M with each row corresponding to a unique circular suffix of T. Then, we sort all this rows



i	T_i	$\mathsf{prev}(\mathsf{T}_i)$	$\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]})$	$\mathsf{T}_{\mathsf{pSA}[i]}$	$\mathsf{pSA}[i]$	$L[i]$	f_i	$\mathsf{pBWT}[i]$	$\mathsf{pLF}(i)$
1	AxyBzCxzwAz$	A00B0C530A3$	000A3$A70B6C	xzwAz$AxyBzC	7	C		C	11
2	xyBzCxzwAz$A	00B0C530A3$A	00A3$A00B6C5	zwAz$AxyBzCx	8	x	7	3	1
3	yBzCxzwAz$Ax	0B0C030A3$A7	00B0C530A3$A	xyBzCxzwAz$A	2	A		A	8
4	BzCxzwAz$Axy	B0C030A3$A70	0A0$A00B6C53	wAz$AxyBzCxz	9	z	3	2	2
5	zCxzwAz$AxyB	0C030A3$A70B	0B0C030A3$A7	yBzCxzwAz$Ax	3	x	5	3	3
6	CxzwAz$AxyBz	C000A3$A70B6	0C030A3$A70B	zCxzwAz$AxyB	5	B		B	10
7	xzwAz$AxyBzC	000A3$A70B6C	0$A00B5C530A	z$AxyBzCxzwA	11	A		A	9
8	zwAz$AxyBzCx	00A3$A00B6C5	A00B0C530A3$	AxyBzCxzwAz$	1	$		$	12
9	wAz$AxyBzCxz	0A0$A00B6C53	A0$A00B5C530	Az$AxyBzCxzw	10	w	12	4	4
10	Az$AxyBzCxzw	A0$A00B5C530	B0C030A3$A70	BzCxzwAz$Axy	4	y	12	4	5
11	z$AxyBzCxzwA	0$A00B5C530A	C000A3$A70B6	CxzwAz$AxyBz	6	z	3	2	6
12	$AxyBzCxzwAz	$A00B0C530A3	$A00B0C530A3	$AxyBzCxzwAz	12	z	6	3	7

Figure 1: The text is $\mathsf{T}[1,12] = AxyBzCxzwAz\$$, where $\Sigma_s = \{A, B, C, \$\}$ and $\Sigma_p = \{w, x, y, z\}$

lexicographically according to the $\mathsf{prev}(\cdot)$ encoding of the corresponding unique circular suffix, and obtain the last column L of the sorted matrix M. Clearly, the ith row is equal to $\mathsf{T}_{\mathsf{pSA}[i]}$. Moving forward, denote by f_i, the first occurrence of $L[i]$ in $\mathsf{T}_{\mathsf{pSA}[i]}$, where $L[i] \in \Sigma_p$. The pBWT of T, denoted by $\mathsf{pBWT}[1, n]$, is defined as:

$$\mathsf{pBWT}[i] = \begin{cases} L[i], \text{if } L[i] \text{ is an s-character,} \\ \text{number of distinct p-characters in} \\ \qquad \mathsf{T}_{\mathsf{pSA}[i]}[1, f_i], \text{ otherwise.} \end{cases}$$

In other words, when $L[i] \in \Sigma_s$, $\mathsf{pBWT}[i] = \mathsf{T}[\mathsf{pSA}[i]-1]$ (define $\mathsf{T}[0] = \mathsf{T}[n] = \$$) and when $L[i] \in \Sigma_p$, $\mathsf{pBWT}[i]$ is the number of 0's in the f_i-long prefix of $\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]})$. Thus, pBWT is a sequence of n characters over the set $\Sigma'' = \Sigma_s \cup \{1, 2, \ldots, \sigma_p\}$ of size $\sigma_s + \sigma_p = \sigma$. See Figure 1 for an illustration.

In order to represent pBWT in succinct space, we map each s-character in Σ'' to a unique integer in $[\sigma_p + 1, \sigma]$. Specifically, the ith smallest s-character will be denoted by $(i + \sigma_p)$. Moving forward, $\mathsf{pBWT}[i] \in [1, \sigma_p]$ iff $L[i]$ is a p-character and $\mathsf{pBWT}[i] \in [\sigma_p + 1, \sigma]$ iff $L[i]$ is a s-character. We summarize the relation between $\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]})$ and $\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]-1})$ below.

Observation 1. *Let $1 \le i \le n$. If $\mathsf{pBWT}[i] \in \Sigma_s$,*

$$\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]-1}) = \mathsf{pBWT}[i] \circ \mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]})[1, n-1]$$

Otherwise, if $\mathsf{pBWT}[i] \notin \Sigma_s$,

$$\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]-1}) = 0 \circ \mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]})[1, f_i - 1] \circ \\ f_i \circ \mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]})[f_i + 1, n - 1]$$

3.1 Parameterized LF-Mapping

Based on the conceptual matrix M, the parameterized last-to-first column (pLF) mapping of i is the position at which the character at $L[i]$ lies in the first column of M. Specifically, $\mathsf{pLF}(i) = \mathsf{pSA}^{-1}[\mathsf{pSA}[i] - 1]$. The significance is summarized in Theorem 3.

Theorem 3. *Assume $\mathsf{pLF}(\cdot)$ can be computed in t_{pLF} time. Then, for any parameter Δ, by using an additional $O((n/\Delta)\log n)$-bit structure, we can compute $\mathsf{pSA}[\cdot]$ and $\mathsf{pSA}^{-1}[\cdot]$ in $O(\Delta \cdot t_{\mathsf{pLF}})$ time.*

Proof. Define, $\mathsf{pLF}^0(i) = i$ and $\mathsf{pLF}^k(i) = \mathsf{pLF}(\mathsf{pLF}^{k-1}(i)) = \mathsf{pSA}^{-1}[\mathsf{pSA}[i] - k]$ for any integer $k > 0$. We maintain two Δ-sampled arrays, one each for pSA and pSA^{-1}. More specifically, we explicitly maintain $\mathsf{pSA}[j]$ and $\mathsf{pSA}^{-1}[j]$ if the value belongs to $\{1, 1 + \Delta, 1 + 2\Delta, 1 + 3\Delta, \ldots, n\}$. The total space for each sampled array can be bounded by $O((n/\Delta)\log n)$ bits. To find $\mathsf{pSA}[i]$, repeatedly apply the $\mathsf{pLF}(\cdot)$ operation (starting from i) until you obtain a j such that $\mathsf{pSA}[j]$ has been explicitly stored. Suppose, the number of such operations is k. Then, $j = \mathsf{pLF}^k(i) = \mathsf{pSA}^{-1}[\mathsf{pSA}[i] - k]$, which gives $\mathsf{pSA}[i] = \mathsf{pSA}[j] + k$. Since $k \leq \Delta$, $\mathsf{pSA}[i]$ is computed in $O(\Delta \cdot t_{\mathsf{pLF}})$ time. To find $\mathsf{pSA}^{-1}[i]$, find the smallest $j \geq i$ whose $\mathsf{pSA}^{-1}[j]$ is explicitly stored. Then, $\mathsf{pSA}^{-1}[i] = \mathsf{pLF}^{j-i}(\mathsf{pSA}^{-1}[j])$. As $j - i \leq \Delta$, the time is bounded by $O(\Delta \cdot t_{\mathsf{pLF}})$. ∎

We remark that using Theorem 3, $\mathsf{prev}(\mathsf{T}[x, y])$ can be extracted in $O(\Delta \cdot t_{\mathsf{pLF}} + (y - x + 1)(t_{\mathsf{pLF}} + \log \sigma))$ time.

To aid the reader's intuition for computing pLF mapping, we present Lemma 1, which shows how to compare the lexicographic rank of two encoded suffixes when prepended by their respective previous characters. This key concept is then implemented in Section 4 to arrive at Theorem 4.

LEMMA 1. *Consider two suffixes i and j corresponding to the leaves ℓ_i and ℓ_j in pST. Then, $\mathsf{pLF}(i)$ and $\mathsf{pLF}(j)$ are related as follows:*

(a) *If $L[i] \in \Sigma_p$ and $L[j] \in \Sigma_s$, then $\mathsf{pLF}(i) < \mathsf{pLF}(j)$.*

(b) *If both $L[i], L[j] \in \Sigma_s$, then $\mathsf{pLF}(i) < \mathsf{pLF}(j)$ iff one of the following holds:*

- $\mathsf{pBWT}[i] < \mathsf{pBWT}[j]$
- $\mathsf{pBWT}[i] = \mathsf{pBWT}[j]$ *and* $i < j$.

(c) *Assume both $L[i], L[j] \in \Sigma_p$ and $i < j$. Let u be the lowest common ancestor of ℓ_i and ℓ_j in pST, and z be the number of 0's in the string $\mathsf{path}(u)$.*

  (1) *If $\mathsf{pBWT}[i], \mathsf{pBWT}[j] \leq z$, then $\mathsf{pLF}(i) < \mathsf{pLF}(j)$ iff $\mathsf{pBWT}[i] \geq \mathsf{pBWT}[j]$.*

  (2) *If $\mathsf{pBWT}[i] \leq z < \mathsf{pBWT}[j]$, then $\mathsf{pLF}(i) > \mathsf{pLF}(j)$.*

  (3) *If $\mathsf{pBWT}[i] > z \geq \mathsf{pBWT}[j]$, then $\mathsf{pLF}(i) < \mathsf{pLF}(j)$.*

  (4) *If $\mathsf{pBWT}[i], \mathsf{pBWT}[j] > z$, then $\mathsf{pLF}(i) > \mathsf{pLF}(j)$ iff*

  - $\mathsf{pBWT}[i] = z + 1$,
  - *the leading character on the u to ℓ_i path is 0, and*
  - *the leading character on the u to ℓ_j path is not an s-character.*

Proof. (a) and (b): Follows immediately from Convention 1 and Observation 1.
(c) Recall that f_i and f_j are the first occurrences of the characters $L[i]$ and $L[j]$ in the circular suffixes $\mathsf{T}_{\mathsf{pSA}[i]}$ and $\mathsf{T}_{\mathsf{pSA}[j]}$ respectively. Let $d = |\mathsf{path}(u)|$. Clearly, the conditions (1)–(4) can be written as: (1) Both $f_i, f_j \leq d$, (2) $f_i \leq d$ and $f_j > d$, (3) $f_i > d$ and $f_j \leq d$, and (4) Both $f_i, f_j > d$.

Then the claims (1)–(3) are immediate from Observation 1 and Convention 1. For proving (4), first observe that if $\mathsf{T}_{\mathsf{pSA}[j]}[d+1]$ is an s-character, then $\mathsf{T}_{\mathsf{pSA}[j]-1}[d+2] > \mathsf{T}_{\mathsf{pSA}[i]-1}[d+2]$, and $\mathsf{pLF}(i) < \mathsf{pLF}(j)$. So, assume otherwise. Let e_i and e_j be the $(d+1)$th characters of $\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]})$ and $\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[j]})$ respectively. Since the suffixes i and j separate after u, $f_i \neq f_j$. Also, $i < j$ implies $0 \leq e_i < e_j \leq d$. Note that if $\mathsf{pBWT}[i] = z + 1$ and $e_i = 0$, then $L[i] = \mathsf{T}_{\mathsf{pSA}[i]}[d+1]$ i.e., $f_i = d+1$, and $\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]-1})[d+2] = d+1 > e_j = \mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[j]-1})[d+2]$. Otherwise, $\mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[i]-1})[d+2] = e_i < e_j \leq \mathsf{prev}(\mathsf{T}_{\mathsf{pSA}[j]-1})[d+2]$. ∎

THEOREM 4. *We can compute $\mathsf{pLF}(i)$ in $O(\log \sigma)$ time using $n \log \sigma + O(n)$ bits.*

4 Implementing pLF Mapping

We prove Theorem 4 in this section.

4.1 Data Structure Toolkit

Following are the key components of the data structure.

4.1.1 Wavelet Tree over pBWT

Grossi, Gupta, and Vitter [22] introduced the wavelet tree (WT) data structure, which generalizes the well-known rank and select queries over bit-vectors.[1] Specifically, given an array A over an alphabet Σ, by using a data structure of size $|A| \log |\Sigma| + o(|A| \log |\Sigma|)$ bits, the following queries can be supported in $O(\log |\Sigma|)$ time:

(a) $A[i]$.

(b) $\mathsf{rank}_A(i, x)$ = number of occurrences of x in $A[1, i]$.

(c) $\mathsf{select}_A(i, x)$ = ith occurrence of x in A.

(d) $\mathsf{rangeCount}_A(i, j, x, y)$ = number of elements in $A[i, j]$ that are at least x and at most y.

We drop the subscript A when the context is clear. The pBWT is a string of length n over an alphabet set $\Sigma'' = \Sigma_s \cup \{1, 2, \ldots, \sigma_p\}$ of size $\sigma = \sigma_s + \sigma_p$. By maintaining a WT over pBWT in $n \log \sigma + o(n \log \sigma)$ bits, we can support the above operations over the pBWT. Using generalized WT [16], we can improve the query time to $t_{\mathsf{WT}} = O(1 + \log \sigma / \log \log n)$ for the above operations. As noted by Navarro [38], we can apply the technique of Golynski et al. [21] to reduce the redundancy of $o(n \log \sigma)$ bits to $o(n)$ bits. The time to answer the above queries remains unaffected.

[1]Given a bit-vector B and $c \in \{0, 1\}$, $\mathsf{rank}(i, c) = |\{j \mid j \leq i \text{ and } B[j] = c\}|$ and $\mathsf{select}(i, c) = \min\{j \mid \mathsf{rank}(j, c) = i\}$.

4.1.2 Succinct representation of pST

We rely on the following result of Navarro and Sadakane [40]. Any tree having m nodes can be represented in $2m + o(m)$ bits, such that if each node is labeled by its pre-order rank, the following operations can be supported in $O(1)$ time (note that $m < 2n$ in our case):

(a) $\mathsf{pre\text{-}order}(u)/\mathsf{post\text{-}order}(u)$ = pre-order/post-order rank of node u.

(b) $\mathsf{parent}(u)$ = parent of node u.

(c) $\mathsf{nodeDepth}(u)$ = number of edges on the path from root to u.

(d) $\mathsf{child}(u, q)$ = qth leftmost child of node u.

(e) $\mathsf{lca}(u, v)$ = lowest common ancestor (LCA) of two nodes u and v.

(f) $\mathsf{lmostLeaf}(u)/\mathsf{rmostLeaf}(u)$ = leftmost/rightmost leaf in the subtree rooted at u.

(g) $\mathsf{levelAncestor}(u, D)$ = ancestor of u such that $\mathsf{nodeDepth}(u) = D$.

Also, we can find the pre-order rank of the ith leftmost leaf in $O(1)$ time. Moving forward, we will use ℓ_i to denote the leaf corresponding to the ith lexicographically smallest prev-encoded suffix.

4.2 ZeroDepth and ZeroNode

For a node u, $\mathsf{zeroDepth}(u)$ is the number of 0's in $\mathsf{path}(u)$. For a leaf ℓ_i with $\mathsf{pBWT}[i] \in [1, \sigma_p]$, $\mathsf{zeroNode}(\ell_i)$ is the highest node z on the root to ℓ_i path such that $\mathsf{zeroDepth}(z) \geq \mathsf{pBWT}[i]$. Thus, z is the locus of $\mathsf{path}(\ell_i)[1, f_i]$. Note that z necessarily exists as $\mathsf{zeroDepth}(\ell_i) \geq \mathsf{pBWT}[i]$. Moving forward, whenever we refer to $\mathsf{zeroNode}(\ell_i)$, we assume $\mathsf{pBWT}[i] \in [1, \sigma_p]$. We present the following important lemma (proof deferred to Section 4.5).

LEMMA 2. *Using an additional $O(n)$-bit structure, we can find $\mathsf{zeroNode}(\ell_i)$ in $O(\log \sigma)$ time.*

We remark that the following additional functionalities: $\mathsf{leafLeadChar}(\cdot)$, $\mathsf{fSum}(\cdot)$ and $\mathsf{pCount}(\cdot)$ will be defined later. Each of these can be computed in $O(1)$ time using an $O(n)$-bit structure.

4.3 Computing $\mathsf{pLF}(i)$ when $\mathsf{pBWT}[i] \in [\sigma_p + 1, \sigma]$

In this case, $L[i] = \mathsf{pBWT}[i]$ is an s-character. Using Lemma 1, we conclude that $\mathsf{pLF}(i) > \mathsf{pLF}(j)$ iff either $j \in [1, n]$ and $\mathsf{pBWT}[j] < \mathsf{pBWT}[i]$, or $j \in [1, i-1]$ and $\mathsf{pBWT}[i] = \mathsf{pBWT}[j]$. Then, $\mathsf{pLF}(i) = 1 + \mathsf{rangeCount}(1, n, 1, c-1) + \mathsf{rangeCount}(1, i-1, c, c)$, where $c = \mathsf{pBWT}[i]$.

4.4 Computing $\mathsf{pLF}(i)$ when $\mathsf{pBWT}[i] \in [1, \sigma_p]$

In this case, $L[i]$ is a p-character. Let $z = \mathsf{zeroNode}(\ell_i)$ and $v = \mathsf{parent}(z)$. Then, $f_i = (|\mathsf{path}(v)| + 1)$ if the leading character on the edge from v to z is 0 and $\mathsf{pBWT}[i] = (\mathsf{zeroDepth}(v) + 1)$; otherwise, $f_i > (|\mathsf{path}(v)| + 1)$. For a leaf ℓ_j in pST, $\mathsf{leafLeadChar}(j)$ is a boolean variable, which is 0 iff $f_j = (|\mathsf{path}(\mathsf{parent}(\mathsf{zeroNode}(\ell_j)))| + 1)$. Using this information, in constant time, we can determine which of the following two cases the suffix corresponding to ℓ_i satisfies (see Figure 2).

4.4.1 Case 1 ($f_i = |\mathsf{path}(v)| + 1$)

In this case, z is the leftmost child of v. Let w be the parent of v. We partition the leaves into four sets:

(a) $\mathcal{S}_1$: leaves to the left of the subtree of v.

(b) $\mathcal{S}_2$: leaves in the subtree of z.

(c) $\mathcal{S}_3$: leaves to the right of the subtree of v.

(d) $\mathcal{S}_4$: leaves in the subtree of v but not of z.

In case, v is the root node r, we take $w = r$; consequently, $\mathcal{S}_1 = \mathcal{S}_3 = \varnothing$.

4.4.2 Case 2 ($f_i > |\mathsf{path}(v)| + 1$)

We partition the leaves into three sets:

(a) $\mathcal{S}_1$ (resp. $\mathcal{S}_3$): leaves to the left (resp. right) of the subtree of z.

(b) $\mathcal{S}_2$: leaves in the subtree of z.

We first compute $z = \mathsf{zeroNode}(\ell_i)$ (using Lemma 2), and then locate $v = \mathsf{parent}(z)$. Using $\mathsf{leafLeadChar}(i)$ and the $\mathsf{lmostLeaf}(\cdot)/\mathsf{rmostLeaf}(\cdot)$ tree operations, we find the desired ranges. Let $[L_x, R_x]$ denote the range of leaves in the subtree of any node x. In order to compute $\mathsf{pLF}(i)$, we first compute N_1, N_2, and N_3, which are respectively the number of leaves ℓ_j in the ranges $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ such that $\mathsf{pLF}(j) \leq \mathsf{pLF}(i)$. Likewise, we compute N_4 (w.r.t $\mathcal{S}_4$) if we are in the first case. Then, $\mathsf{pLF}(i) = N_1 + N_2 + N_3 + N_4$.

4.4.3 Computing N_1

For any leaf $\ell_j \in \mathcal{S}_1$, $\mathsf{pLF}(j) < \mathsf{pLF}(i)$ iff $f_j > 1 + |\mathsf{path}(\mathsf{lca}(z, \ell_j))|$ and $L[j] \in \Sigma_p$. Therefore, N_1 is the number of leaves ℓ_j, $L[j] \in \Sigma_p$, which comes before z in pre-order with $f_j > 1 + |\mathsf{path}(\mathsf{lca}(z, \ell_j))|$. Define, $\mathsf{fCount}(x)$ of a node x as the number of leaves ℓ_j in x's subtree such that $|\mathsf{path}(y)| + 2 \leq f_j \leq |\mathsf{path}(x)| + 1$, where $y = \mathsf{parent}(x)$. If x is the root node, then $\mathsf{fCount}(x) = 0$. Define $\mathsf{fSum}(x)$ of a node x as $\sum \mathsf{fCount}(y)$ of all nodes y which come before x in pre-order and are not ancestors of x. By this definition, $N_1 = \mathsf{fSum}(z)$ is computed as follows.

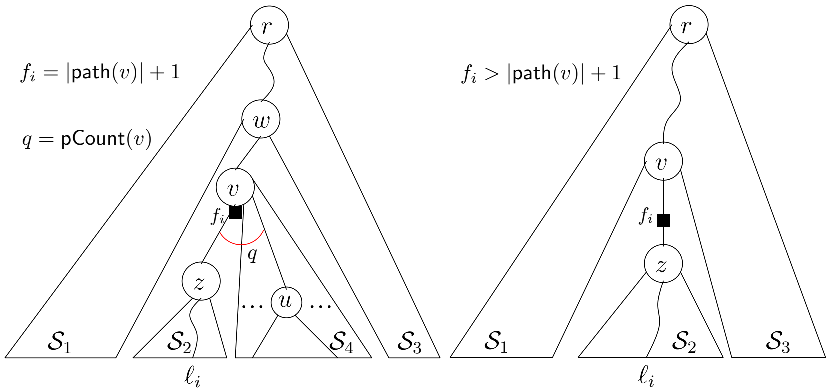


Figure 2: Illustration of various suffix ranges when the suffix $\mathsf{T}_{\mathsf{pSA}[i]}$ is preceded by a p-character

LEMMA 3. *By maintaining an $O(n)$-bit structure, we can compute $\mathsf{fSum}(x)$ in $O(1)$ time.*

Proof. Traverse the pST in DFS order, and write $\mathsf{fCount}(v)$ of a node v in unary when exiting the node in the traversal, i.e., $\mathsf{fCount}(v)$ is associated with $\mathsf{post\text{-}order}(v)$. Maintain a $\mathsf{rank\text{-}select}$ structure on this bit-string B. Since $\sum_v \mathsf{fCount}(v) \leq n$, $|B| \leq 3n$, and the space needed is $3n + o(n)$ bits. Note that $\mathsf{fSum}(x)$ is same as the number of 1s in B up to the position corresponding to y, where y is conceptually found as follows. Traverse from x to root until we get a node y' which has a child to the left of the path. Then y is the rightmost child of y' that lies to the left of the path. If $L_x = 1$, then y is not defined and $\mathsf{fSum}(x) = 0$. Otherwise, $y = \mathsf{levelAncestor}(L_x - 1, \mathsf{nodeDepth}(\mathsf{lca}(L_x, L_x - 1)) + 1)$ and $\mathsf{fSum}(x) = \mathsf{rank}_B(\mathsf{select}_B(\mathsf{post\text{-}order}(y), 0), 1)$. Clearly, the time required is $O(1)$. ■

4.4.4 Computing N_2

Note that for any leaf $\ell_j \in \mathcal{S}_2$, $\mathsf{pLF}(j) \leq \mathsf{pLF}(i)$ iff $L[j] \in \Sigma_p$ and either $f_j > f_i$ or $f_j = f_i$ and $j \leq i$. Therefore, N_2 is the number of leaves ℓ_j in $\mathcal{S}_2$ which satisfy one of the following conditions: (a) $\mathsf{pBWT}[i] < \mathsf{pBWT}[j] \leq \sigma_p$, or (b) $\mathsf{pBWT}[i] = \mathsf{pBWT}[j]$ and $j \leq i$. Then, $N_2 = \mathsf{rangeCount}(L_z, R_z, c+1, \sigma_p) + \mathsf{rangeCount}(L_z, i, c, c)$, where $c = \mathsf{pBWT}[i]$.

4.4.5 Computing N_3

For any leaf $\ell_j \in \mathcal{S}_3$, $\mathsf{pLF}(j) > \mathsf{pLF}(i)$. Thus, $N_3 = 0$.

4.4.6 Computing N_4

Note that $\mathsf{pBWT}[i]$ is same as $(\mathsf{zeroDepth}(v) + 1)$. Consider a leaf $\ell_j \in \mathcal{S}_4$ with $L[j] \in \Sigma_p$. Since the suffix j deviates from the suffix i at the node v, we have $f_j \neq f_i$. Therefore, $\mathsf{pLF}(j) < \mathsf{pLF}(i)$ iff $f_j > f_i$, and the leading character on the path from v to ℓ_j is not an s-character. For a node x, $\mathsf{pCount}(x)$ is the number of children y of x such that the leading character from x to y is not an s-character. Note that $\sum_x \mathsf{pCount}(x) = O(n)$. Therefore, we encode $\mathsf{pCount}(\cdot)$ of all nodes in $O(n)$ bits using unary encoding, such that $\mathsf{pCount}(x)$ can be retrieved in constant time. Let u be the $\mathsf{pCount}(v)$th child of v. Then, N_4 is the number of leaves ℓ_j in $\mathcal{S}_4$ such that $j \leq R_u$ and $\sigma_p \geq \mathsf{pBWT}[j] \geq \mathsf{pBWT}[i]$ i.e., $N_4 = \mathsf{rangeCount}(R_z + 1, R_u, \mathsf{pBWT}[i], \sigma_p)$.

We summarize the LF mapping procedure in Algorithm 1. Once $\mathsf{zeroDepth}(\ell_i)$ is known, N_1 is computed in $O(1)$ time, and both N_2 and N_4 are computed in $O(1 + \log\sigma/\log\log n)$ time. Combining these results with Lemma 2, we arrive at Theorem 4.

4.5 Proof of Lemma 2

For any node x on the root to ℓ_i path π, define $\alpha(x)$ = the number of leaves $\ell_j \in \mathsf{leaf}(x)$ such that $L[j] \in \Sigma_p$ and $f_j \leq |\mathsf{path}(x)|$, and $\beta(x) = \mathsf{rangeCount}(L_x, R_x, 1, \mathsf{pBWT}[i])$. Here, $\mathsf{leaf}(x)$ is the range of leaves in the subtree of x. Consider a node u_k on π. Observe that $\mathsf{zeroNode}(\ell_i)$ is below u_k iff $\beta(u_k) > \alpha(u_k)$. Therefore, $\mathsf{zeroNode}(\ell_i)$ is the shallowest node $u_{k'}$ on this path that satisfies $\beta(u_{k'}) \leq \alpha(u_{k'})$. Equipped with this knowledge, now we can binary search on π (using $\mathsf{nodeDepth}$ and $\mathsf{levelAncestor}$ operations) to find the exact location. The first question is to compute $\alpha(x)$, which is handled by Lemma 4. A normal binary search will have to consider n nodes on the path in the worst case. Lemma 5 shows how to reduce this to $\lceil \log\sigma \rceil$. Thus, the binary search has at most $\lceil \log\log\sigma \rceil$ steps, and the total time is $\log\log\sigma \times \lceil \frac{\log\sigma}{\log\log n} \rceil = O(\log\sigma)$, as required.

LEMMA 4. *By maintaining an $O(n)$-bit structure, we can find $\alpha(x)$ in $O(1)$ time.*

Proof. Let $A[1,n]$ be a bit-array such that $A[i] = 1$ iff $L[i] \in \Sigma_p$. Maintain a $\mathsf{rank\text{-}select}$ structure over A. Define $\gamma(v)$ as the number of leaves $\ell_j \in \mathsf{leaf}(v)$ that satisfy $L[j] \in \Sigma_p$ and $|\mathsf{path}(\mathsf{parent}(v))| < f_j \leq |\mathsf{path}(v)|\}|$. Traverse pST in DFS order, and write $\gamma(v)$

Algorithm 1 computes $\mathsf{pLF}(i)$

$c \leftarrow \mathsf{pBWT}[i]$
if $(c > \sigma_p)$ **then** $\mathsf{pLF}(i) \leftarrow 1 + \mathsf{rangeCount}(1, n, 1, c-1) + \mathsf{rangeCount}(1, i-1, c, c)$
else $z \leftarrow \mathsf{zeroNode}(\ell_i)$, $v \leftarrow \mathsf{parent}(z)$, $L_z \leftarrow \mathsf{lmostLeaf}(z)$, $R_z \leftarrow \mathsf{rmostLeaf}(z)$
    $N_1 \leftarrow \mathsf{fSum}(z)$, $N_2 \leftarrow \mathsf{rangeCount}(L_z, R_z, c+1, \sigma_p) + \mathsf{rangeCount}(L_z, i, c, c)$
    if ($\mathsf{leafLeadChar}(i)$ is 0) **then**
        $u \leftarrow \mathsf{child}(v, \mathsf{pCount}(v))$, $N_4 \leftarrow \mathsf{rangeCount}(R_z + 1, \mathsf{rmostLeaf}(u), c, \sigma_p)$
    $\mathsf{pLF}(i) \leftarrow N_1 + N_2 + N_4$

in unary when entering v's subtree. Maintain a rank-select structure on this bit-vector B. Since $\sum_v \gamma(v) \leq n$, $|B| \leq 3n$. The total space needed is $4n + o(n)$ bits. Let $\alpha'(x)$ be the number of leaves $\ell_j \in \mathsf{leaf}(x)$ such that $L[j] \in \Sigma_p$ and $f_j > |\mathsf{path}(x)|$. Then,

$$\alpha'(x) = \mathsf{rank}_B(\mathsf{select}_B(\text{pre-order}(\ell_{R_x}), 0), 1) - \mathsf{rank}_B(\mathsf{select}_B(\text{pre-order}(x), 0), 1)$$

$$\alpha(x) = \mathsf{rank}_A(R_x, 1) - \mathsf{rank}_A(L_x - 1, 1) - \alpha'(x)$$

Clearly, the space-and-time bounds are met. ∎

Lemma 5. *Using an additional $O(n)$-bit structure, in $O(\log \sigma)$ time, we can find an ancestor w_i of ℓ_i such that $\mathsf{zeroDepth}(w_i) < \mathsf{pBWT}[i]$ and w_i is at most $\lceil \log \sigma \rceil$ nodes above $\mathsf{zeroNode}(\ell_i)$.*

Proof. Let $g = \lceil \log \sigma \rceil$ be a sampling factor. We first mark all those nodes v in the pST such that $\mathsf{nodeDepth}(v)$ is a multiple of g and the subtree of v has at least g nodes. Also, mark the root node. It is easy to see that (i) between any two closest marked nodes (or a lowest marked node and a leaf in its subtree) there are at most g nodes, and (ii) the number of marked nodes is $O(n/g)$. Maintain a bit-array B such that $B[k] = 1$ iff the node with pre-order rank k is a marked node. Also, maintain a rank-select structure on B. The space needed is $O(n)$ bits. We also maintain an array D, such that $D[k]$ equals the zeroDepth of the marked node corresponding to the kth 1-bit in B. Given a marked node with pre-order rank k', its corresponding position in D is given by $\mathsf{rank}_B(k', 1)$. We do not maintain D explicitly; instead, we maintain a wavelet tree over it. The space needed is $O(\frac{n}{g} \log \sigma) = O(n)$ bits.

Given a leaf node ℓ_i, in $O(\log \sigma)$ time, first locate its lowest marked ancestor u by traversing the tree upwards. Then, find the position j corresponding to u in the array D. If $\mathsf{zeroDepth}(u) < \mathsf{pBWT}[i]$, then $w_i = u$, and we are done. Otherwise, locate the rightmost position $j' < j$ in D such that $D[j'] < \mathsf{pBWT}[i]$. Using the wavelet tree over D, this predecessor search takes $O(\log \sigma)$ time. (Since the root node is marked, and its zeroDepth equals 0, the position j' necessarily exists.) Obtain the marked node v corresponding to the j'th 1-bit in B via a $\mathsf{select}_B(j', 1)$ operation. Then, $w_i = \mathsf{lca}(u, v)$. The time required is $O(\log \sigma)$. To see the correctness, observe that $\mathsf{lca}(u, v)$ is an ancestor of ℓ_i. For a node x, $\mathsf{zeroDepth}(x) \geq \mathsf{zeroDepth}(\mathsf{parent}(x))$. Thus, $\mathsf{zeroDepth}(\mathsf{lca}(u, v)) \leq \mathsf{zeroDepth}(u) < \mathsf{pBWT}[i]$. If $\mathsf{lca}(u, v)$ is not the desired node, then it has a marked descendant $u' \neq u$ on the path to u such that $\mathsf{zeroDepth}(u') < \mathsf{pBWT}[i]$. But u' appears after v and before u in pre-order, a contradiction. ∎

5 Pattern Matching via Backward Search

We modify the backward search algorithm in the FM-index [16]. In particular, given a proper suffix Q of P, assume that we know the suffix range $[\mathsf{sp}_1, \mathsf{ep}_1]$ of $\mathsf{prev}(Q)$. Our task is to find the suffix range $[\mathsf{sp}_2, \mathsf{ep}_2]$ of $\mathsf{prev}(c \circ Q)$, where c is the character previous to Q in P.

If c is static, then $\mathsf{prev}(c \circ Q) = c \circ \mathsf{prev}(Q)$. The backward search in this case is similar to that in FM-index. Specifically,

$$\mathsf{sp}_2 = 1 + \mathsf{rangeCount}(1, n, 1, c-1) + \mathsf{rangeCount}(1, \mathsf{sp}_1 - 1, c, c)$$

$$\mathsf{ep}_2 = \mathsf{rangeCount}(1, n, 1, c-1) + \mathsf{rangeCount}(1, \mathsf{ep}_1, c, c)$$

Now, we consider the scenario when c is parameterized. By maintaining a bit-vector $B[1, \sigma_p]$, in $O(|P|)$ time, we first identify all positions j, where $P[j] \in \Sigma_p$ is not in $P[j+1, |P|]$. We have the following two cases.

5.1 Case 1 (c does not appear in Q)

Note that $\mathsf{pLF}(i) \in [\mathsf{sp}_2, \mathsf{ep}_2]$ iff $i \in [\mathsf{sp}_1, \mathsf{ep}_1]$, $L[i]$ is a p-character and $f_i > |Q|$. This holds iff $i \in [\mathsf{sp}_1, \mathsf{ep}_1]$ and $\mathsf{pBWT}[i] \in [d+1, \sigma_p]$. Here, d is the number of distinct p-characters in Q, which can be obtained in $O(1)$ time by initially pre-processing P in $O(|P|)$ time. Then, $(\mathsf{ep}_2 - \mathsf{sp}_2 + 1) = \mathsf{rangeCount}(\mathsf{sp}_1, \mathsf{ep}_1, d+1, \sigma_p)$. Now, $\mathsf{pLF}(i) < \mathsf{sp}_2$ iff $i < \mathsf{sp}_1$, $L[i] \in \Sigma_p$, and $f_i > 1 + |\mathsf{path}(\mathsf{lca}(u, \ell_i))|$, where $u = \mathsf{lca}(\ell_{\mathsf{sp}_1}, \ell_{\mathsf{ep}_1})$. Finally, we compute $\mathsf{sp}_2 = 1 + \mathsf{fSum}(u)$ in constant time (refer to Lemma 3).

5.2 Case 2 (c appears in Q)

Note that $\mathsf{pLF}(i) \in [\mathsf{sp}_2, \mathsf{ep}_2]$ iff $i \in [\mathsf{sp}_1, \mathsf{ep}_1]$, $L[i]$ is a p-character, and f_i is the same as the first occurrence of c in Q. This holds iff $i \in [\mathsf{sp}_1, \mathsf{ep}_1]$ and $\mathsf{pBWT}[i] = d$. Here, d is the number of distinct p-characters in Q until (and including) the first occurrence of c. We can compute d in constant time by initially pre-processing P in $O(|P| \log \sigma)$ time[2].

Consider $i, j \in [\mathsf{sp}_1, \mathsf{ep}_1]$ such that $i < j$ and $\mathsf{pLF}(i), \mathsf{pLF}(j) \in [\mathsf{sp}_2, \mathsf{ep}_2]$. Now, both f_i and f_j equals the first occurrence of c in Q. Based on Observation 1, we conclude that $\mathsf{pLF}(i) < \mathsf{pLF}(j)$. Therefore, $\mathsf{sp}_2 = \mathsf{pLF}(i_{min})$ and $\mathsf{ep}_2 = \mathsf{pLF}(i_{max})$, where

$$\begin{aligned} i_{min} &= \min\{j \mid j \in [\mathsf{sp}_1, \mathsf{ep}_1] \text{ and } \mathsf{pBWT}[j] = d\} \\ &= \mathsf{select}(\mathsf{rank}(\mathsf{sp}_1 - 1, d) + 1, d) \\ i_{max} &= \max\{j \mid j \in [\mathsf{sp}_1, \mathsf{ep}_1] \text{ and } \mathsf{pBWT}[j] = d\} \\ &= \mathsf{select}(\mathsf{rank}(\mathsf{ep}, d), d) \end{aligned}$$

We have $t_{\mathsf{pLF}} = O(\log \sigma)$ and $t_{\mathsf{WT}} = O(1 + \log \sigma / \log \log n)$. Therefore, we find the suffix range in $O(|P| \log \sigma)$ time. Theorem 1 follows from Theorem 4 and by choosing $\Delta = \lceil \log n \rceil$ in Theorem 3.

6 Dictionary Matching

Let us first look at the index of Idury and Schäffer [29]. For simplicity, we only consider p-characters, and defer the complete details to the full-version. We begin by obtaining $\mathsf{prev}(\mathsf{P}_i)$ for every P_i in $\mathcal{D}$, and then create a trie $\mathcal{T}$ for all the encoded patterns. The number of nodes in the trie is $m \le n + 1$. For each node u in the trie, denote by $\mathsf{path}(u)$ the string formed by concatenating the edge labels from root to u. Mark a node u in the trie as *final* iff $\mathsf{path}(u) = \mathsf{prev}(\mathsf{P}_i)$ for some P_i in $\mathcal{D}$. Clearly, the number of final nodes is d. For any prev-encoded string $\mathsf{prev}(S)$ of a string S, and an integer $j \in [1, |S|]$, we obtain a string $\zeta(S, j)$ as follows. Initialize $\zeta(S, j) = \mathsf{prev}(S)[j, |S|]$. For each $j' \in [1, |S| - j + 1]$, assign $\zeta(S, j)[j'] = 0$ iff $\zeta(S, j)[j'] \ge j'$. Conceptually, $\zeta(S, j) = \mathsf{prev}(S[j, |S|])$. Each node u is associated with the following 3 links:

(a) $\mathsf{next}(u, c) = v$ iff the label on the edge from the node u to v is labeled by the character c,

(b) $\mathsf{failure}(u) = v$ iff $\mathsf{path}(v) = \zeta(\mathsf{path}(u), j)$, where $j > 1$ is the smallest index for which such a node v exists, and

(c) $\mathsf{report}(u) = v$ iff v is a final node and $\mathsf{path}(v) = \zeta(\mathsf{path}(u), j)$, where $j > 1$ is the smallest index for which such a node v exists.

If no such j exists, then $\mathsf{failure}(u)/\mathsf{report}(u)$ points to the root. We first modify the label of each edge in the trie as follows. If any edge is labeled by 0 we assign it a new p-character. Otherwise, if it has value x, then it gets the character assigned to the edge that is x levels above it on the path to root. Each state u is *conceptually labeled* by the lexicographic rank of $\overleftarrow{\mathsf{prev}}(u)$ in the set $\{\overleftarrow{\mathsf{prev}}(v) \mid v \text{ is a node in the trie}\}$, where $\overleftarrow{\mathsf{prev}}(u)$ is the string obtained by prev-encoding the path from u to root. For any $e = (w, x)$, we define $\mathsf{Z}(x) =$ the number of 0's in $\overleftarrow{\mathsf{prev}}(w)[1, f_x]$, where f_x is the first occurrence of the p-character labeling e in the string from w to root. (Note that $\mathsf{Z}(x) \le \sigma_p$.) If f_x is not defined, we let $\mathsf{Z}(x)$ equal the number of 0's in $\overleftarrow{\mathsf{prev}}(x)$. For any two distinct nodes u and v in $\mathcal{T}$, we denote $u \prec v$ iff $\overleftarrow{\mathsf{prev}}(u)$ is lexicographically smaller than $\overleftarrow{\mathsf{prev}}(v)$. Since $\overleftarrow{\mathsf{prev}}(u) \ne \overleftarrow{\mathsf{prev}}(v)$, the relation $u \prec v$ is well-defined.

We create a compressed $\overleftarrow{\mathcal{T}}$ as follows. Initially $\overleftarrow{\mathcal{T}}$ is empty. For each non-leaf node u in $\mathcal{T}$ and for each child u_i of u, we add the string $\overleftarrow{\mathsf{prev}}(u) \circ \$_{u,i}$ to $\overleftarrow{\mathcal{T}}$. Clearly, each string corresponds to a leaf, say $\ell_{u,i}$, in $\overleftarrow{\mathcal{T}}$. We order the leaves according to the (lexicographic) rank of the string they represent. For any two nodes $u, v \in \mathcal{T}$, the rank of a string $\overleftarrow{\mathsf{prev}}(u) \circ \$_{u,i}$ is smaller than $\overleftarrow{\mathsf{prev}}(v) \circ \$_{v,j}$ iff $u \prec v$. On the other hand, the rank of a string $\overleftarrow{\mathsf{prev}}(u) \circ \$_{u,i}$ is smaller than that of $\overleftarrow{\mathsf{prev}}(u) \circ \$_{u,j}$ iff $\mathsf{Z}(u_i) > \mathsf{Z}(u_j)$. (Note that $\mathsf{Z}(u_i) \ne \mathsf{Z}(u_j)$, as u_i and u_j share the same parent.) If two leaves have distinct parents, then their order is defined by the relation $\prec$ on their parents. In both the cases, the rank of the two strings corresponding to any two leaves is well defined.

The key idea to perform the next-transition is presented in the following lemma.

LEMMA 6. *Consider two non-leaf nodes u and v (not necessarily distinct) and its respective children u_i and v_j in $\mathcal{T}$. Let the respective characters (from Σ) on the edges be c_i and c_j. Assume either $u \prec v$ or $u = v$. Let $x = \mathsf{lca}(u, v)$, and z be the number of 0's in $\mathsf{prev}(x)$.*

(a) *If $\mathsf{Z}(u_i), \mathsf{Z}(v_j) \le z$, then $\mathsf{next}(u, c_i) \prec \mathsf{next}(v, c_j)$ iff $\mathsf{Z}(u_i) \ge \mathsf{Z}(v_j)$.*

(b) *$\mathsf{Z}(u_i) \le z < \mathsf{Z}(v_j)$, then $\mathsf{next}(v, c_j) \prec \mathsf{next}(u, c_i)$.*

(c) *$\mathsf{Z}(v_j) \le z < \mathsf{Z}(u_i)$, then $\mathsf{next}(u, c_i) \prec \mathsf{next}(v, c_j)$.*

(d) *If $\mathsf{Z}(u_i), \mathsf{Z}(v_j) > z$, then $\mathsf{next}(v, c_j) \prec \mathsf{next}(u, c_i)$ iff*

- *$\mathsf{Z}(u_i) = z + 1$, and*
- *the leading character on the path from x to $\ell_{u,i}$ is 0.*

[2] Maintain an array $F[1, \sigma_p]$, all values initialized to 0, and a balanced binary search tree $\mathcal{T}$ (initially empty). Scan the string from right to left, and when a p-character c_p is encountered at a position x, check $F[c_p]$. If $F[c_p] = 0$, insert c_p in $\mathcal{T}$ keyed by x. Otherwise, the count at x is the number of nodes in $\mathcal{T}$ with key at most $F[c_p]$. Update $F[c_p]$ and the key of c_p to x. Since the size of $\mathcal{T}$ is $O(\sigma_p)$, search, insertion, and update time is $O(\log \sigma)$.

The above lemma is closely reminiscent of Lemma 1, and by employing similar strategies as in Section 4, we can perform a next-operation in $O(\log \sigma)$ time using $m \log \sigma + O(m)$ bits. For any two nodes u and v, if $\mathsf{failure}(u) = v$, then it $\overleftarrow{\mathsf{prev}}(v)$ is the longest prefix of $\overleftarrow{\mathsf{prev}}(u)$ that appears in $\mathcal{T}$. Similar remarks hold for $\mathsf{report}(u) = v$, where v is a final node. Therefore, these behave exactly in the same manner as in the case of traditional pattern matching, and we can re-use the idea of Belazzougui [9] to perform these transitions in $O(1)$ time using $O(m + d \log(n/d))$ bits. Putting these together we obtain Theorem 2.

7 Discussion

We leave a few questions unanswered. The first one, concerning the space consumption, is "Can we convert $O(n)$ term to $o(n)$ in our space requirements?". The second one is related to construction of the index. Given the p-suffix tree, our index can be constructed in $O(n \log \sigma)$ time using $O(n \log n)$ bits. Therefore, by first creating pST using Kosaraju's algorithm [32], we have an $O(n \log \sigma)$ time and $O(n \log n)$ bit construction algorithm. An immediate question is "Does there exist a (possibly randomized) algorithm for constructing a compressed index for the parameterized matching problem that uses $O(n \log \sigma)$ bits of working space and attains (nearly) the same bounds of the best-known algorithms for constructing p-suffix trees [10, 32]?". An important direction in compressed text indexing is to achieve entropy bounds. In this regard, an obvious question is "Can we design an index, whose size is bounded by $nH_k(\mathsf{T})$, where $H_k(\mathsf{T})$ denotes the kth order entropy of the text T?". Even for $k = 0$, this seems challenging as the 0th order entropy of pBWT can either be smaller or greater than that of T.

References

[1] CLoning Analysis and Categorization System (CLICS). http://www.swag.uwaterloo.ca/clics. Accessed: 2016-06-08.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[3] A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and E. Porat. Function matching: Algorithms, applications, and a lower bound. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, pages 929–942, 2003.

[4] A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Inf. Process. Lett.*, 49(3):111–115, 1994.

[5] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 71–80, 1993.

[6] B. S. Baker. On finding duplication and near-duplication in large software systems. In *2nd Working Conference on Reverse Engineering, WCRE '95, Toronto, Canada, July 14-16, 1995*, pages 86–95, 1995.

[7] B. S. Baker and U. Manber. Deducing similarities in java sources from bytecodes. In *USENIX Annual Technical Conference*, pages 179–190, 1998.

[8] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pages 368–377, 1998.

[9] D. Belazzougui. Succinct dictionary matching with no slowdown. In *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, pages 88–100, 2010.

[10] R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. *SIAM J. Comput.*, 33(1):26–42, 2003.

[11] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Order-preserving incomplete suffix trees and order-preserving indexes. In *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, pages 84–95, 2013.

[12] G. A. Di Lucca, M. Di Penta, A. R. Fasolino, and P. Granato. Clone analysis in the web era: An approach to identify cloned web pages. In *Seventh IEEE Workshop on Empirical Studies of Software Maintenance*, pages 107–113, 2001.

[13] C. du Mouza, P. Rigaux, and M. Scholl. Efficient evaluation of parameterized pattern queries. In *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pages 728–735, 2005.

[14] C. du Mouza, P. Rigaux, and M. Scholl. Parameterized pattern queries. *Data Knowl. Eng.*, 63(2):433–456, 2007.

[15] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[16] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[17] T. Gagie and G. Manzini. Toward a succinct index for order-preserving pattern matching. 2016.

[18] A. Ganguly, W. Hon, K. Sadakane, R. Shah, S. V. Thankachan, and Y. Yang. Space-efficient dictionaries for parameterized and order-preserving pattern matching. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, pages 2:1–2:12, 2016.

[19] A. Ganguly, W. Hon, and R. Shah. A framework for dynamic parameterized dictionary matching. In *15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22-24, 2016, Reykjavik, Iceland*, pages 10:1–10:14, 2016.

[20] R. Giancarlo. The suffix of a square matrix, with applications. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas.*, pages 402–411, 1993.

[21] A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In *Algorithms - ESA 2007, 15th Annual European Symposium, Eilat, Israel, October 8-10, 2007, Proceedings*, pages 371–382, 2007.

[22] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850, 2003.

[23] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[24] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

[25] C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. In *Algorithms - ESA 2004, 12th Annual European Symposium, Bergen, Norway, September 14-17, 2004, Proceedings*, pages 414–425, 2004.

[26] W. Hon, T. Ku, T. W. Lam, R. Shah, S. Tam, S. V. Thankachan, and J. S. Vitter. Compressing dictionary matching index via sparsification technique. *Algorithmica*, 72(2):515–538, 2015.

[27] W. Hon, T. Ku, R. Shah, S. V. Thankachan, and J. S. Vitter. Faster compressed dictionary matching. *Theor. Comput. Sci.*, 475:113–119, 2013.

[28] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–9, Sept 2010.

[29] R. M. Idury and A. A. Schäffer. Multiple matching of parameterized patterns. In *Combinatorial Pattern Matching, 5th Annual Symposium, CPM 94, Asilomar, California, USA, June 5-8, 1994, Proceedings*, pages 226–239, 1994.

[30] M. Jalsenius, B. Porat, and B. Sach. Parameterized matching in the streaming model. In *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, pages 400–411, 2013.

[31] D. K. Kim, Y. A. Kim, and K. Park. Generalizations of suffix arrays to multi-dimensional matrices. *Theor. Comput. Sci.*, 302(1-3):223–238, 2003.

[32] S. R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees (preliminary version). In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 631–637, 1995.

[33] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software, 23.07. - 26.07.2006*, 2006.

[34] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 253–262, 2006.

[35] M. Lewenstein. Parameterized pattern matching. In *Encyclopedia of Algorithms.* 2015.

[36] J. Mendivelso and Y. J. Pinzón. Parameterized matching: Solutions and extensions. In *Proceedings of the Prague Stringology Conference 2015, Prague, Czech Republic, August 24-26, 2015*, pages 118–131, 2015.

[37] G. Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4):52:1–52:47, 2013.

[38] G. Navarro. Wavelet trees for all. *J. Discrete Algorithms*, 25:2–20, 2014.

[39] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[40] G. Navarro and K. Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014.

[41] D. Rattan, R. K. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information & Software Technology*, 55(7):1165–1199, 2013.

[42] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENS UNIVERSITY*, 115, 2007.

[43] L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53, 2011.

[44] F. V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pages 336–339, 2004.

[45] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 225–232, 2002.

[46] T. Shibuya. Generalization of a suffix tree for RNA structural pattern matching. In *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, pages 393–406, 2000.

[47] P. Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11, 1973.