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, 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 $\Sigma$, which is the union of two disjoint sets: $\Sigma_s$ containing static characters (s-characters) and $\Sigma_p$ containing parameterized characters (p-characters). A pattern $P$ (also over $\Sigma$) 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 $\Sigma$ of size $\sigma$. 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 bits. 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@eecs.lsu.edu
Email: rahul@nsf.gov
§Department of CS at the University of Central Florida (UCF). Email: sharma.thankachan@ucf.edu
success [37, 38]. 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 [3], 2D suffix tree [20, 31], structural suffix tree [40] 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 \( \Sigma \) is the union of two disjoint sets: \( \Sigma_s \) having \( \sigma_s \) static characters (s-characters) and \( \Sigma_p \) having \( \sigma_p \) 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 \( \Sigma \). 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 \( \Sigma \)), 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, \( P = A x B y C z x \) p-matches within \( T[1, 2] = A y B x C y A w B x C z x y A z B w C z \$ \) at positions 1 and 15. At position 1, the mapping is \( x \rightarrow y \) and \( y \rightarrow x \); whereas at position 15, the mapping is \( x \rightarrow z \) and \( y \rightarrow w \). 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 \( T[1, n] \) 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 [10] 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] [29] [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 \( D \) be a collection of \( d \) patterns \( \{ P_1, P_2, \ldots, P_d \} \) of total length \( n \) characters that are chosen from \( \Sigma \). The task is to index \( D \), such that given a text \( T \) (also over \( \Sigma \)), we can report all pairs \( \langle j, P_i \rangle \) i.e., a position \( j \) and a pattern \( P_i \in D \) which is a p-match with \( T[j − |P_i| + 1, j] \).

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 \( O(|T| \log \sigma + \text{occ}) \). Recently, Ganguly et al. [18] presented an \( O(n \log \sigma + d \log n) \)-bit index with \( O(|T|(\log \sigma + \log_{\log n} n) + \text{occ}) \) 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, 20].

**Theorem 2.** All \( occ \) pairs \( \langle j, P_i \rangle \), such that a pattern \( P_i \in D \) is a p-match with \( T[j − |P_i| + 1, j] \), can be found in \( O(|T| \log \sigma + \text{occ}) \) 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 [5] 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][1][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][31][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 [1][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 [14][40]. 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 $i$th 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 $\text{prev}(S)$ of length $|S|$ by replacing the first occurrence of every $p$-character by 0 and any other occurrence of a $p$-character by the difference in text position from its previous occurrence. Also, $i$ starting at position $S$ where $i > 0$ and any other occurrence of a $p$-character by the difference in text position from its previous occurrence. Specifically, for any $i \in [1,|S|]$, $\text{prev}(S)[i] = S[i]$ if $S[i]$ is an $s$-character; otherwise, $\text{prev}(S)[i] = (i-j)$, where $j < i$ is the last occurrence of $S[i]$ before $i$. If $j$ does not exist, then $j = i$. For example, $\text{prev}(ABxyBx) = A0B0B4$, where $A,B \in \Sigma_s$ and $x,y \in \Sigma_p$. Note that $\text{prev}(S)$ 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 $\text{prev}(S) = \text{prev}(S')$. Also $S$ and a prefix of $S'$ are a p-match iff $\text{prev}(S)$ is a prefix of $\text{prev}(S')$.

Moving forward, we follow the convention below.

**Convention 1.** In $\Sigma'$, 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, $\Sigma$ is lexicographically larger than all other characters.

Parameterized Suffix Tree (pST) is the compact trie of all strings in $P = \{\text{prev}(T[k,n]) \mid 1 \leq k \leq n\}$. Each edge is labeled with a string over $\Sigma'$. We use $\text{path}(u)$ 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 $\text{path}(u)$ is prefixed by $\text{prev}(P)$. Then find the range $[s_p, e_p]$ (called suffix range of $\text{prev}(P)$) of leaves in the subtree of $u$ and report $\{pSA[i] \mid sp \leq i \leq ep\}$ as the output. Here, $pSA[i,n]$ is the parameterized suffix array i.e., $pSA[i] = j$ and $pSA^{-1}[j] = i$ if $\text{prev}(T[j,n])$ is the $i$th lexicographically smallest string in $P$. (Note that $\text{path}(\ell_i) = \text{prev}(T[pSA[i],n])$, where $\ell_i$ is the $i$th leftmost leaf in $pST$.) The query time is $O(|P| \log \sigma + \text{acc})$.

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
Let \( \map{\sigma}{\Sigma} \) be the \( \map{\sigma}{\Sigma} \) encoding of the corresponding unique circular suffix, and obtain the last column \( L \) of the sorted matrix \( M \). Clearly, the \( i \)-th row is equal to \( \map{p\SigmaA}{i} \). Moving forward, denote by \( f_i \), the first occurrence of \( L[i] \) in \( \map{p\SigmaA}{i} \), where \( L[i] \in \Sigma_p \). The pBWT of \( T \), denoted by \( \map{pBWT}{1, n} \), is defined as:

\[
\map{pBWT}{i} = \begin{cases} 
L[i], & \text{if } L[i] \text{ is an s-character,} \\
\text{number of distinct p-characters in } \map{p\SigmaA}{i}[1, f_i], & \text{otherwise.}
\end{cases}
\]

In other words, when \( L[i] \in \Sigma_s \), \( \map{pBWT}{i} = \map{p\SigmaA}{i} - 1 \) (define \( \map{T}{0} = \map{T}{n} = \$ \)) and when \( L[i] \in \Sigma_p \), \( \map{pBWT}{i} \) is the number of 0’s in the \( f_i \)-long prefix of \( \map{p\SigmaA}{i} \). Thus, \( \map{pBWT}{i} \) 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 \( \map{pBWT}{i} \) in succinct space, we map each s-character in \( \Sigma'' \) to a unique integer in \( [\sigma_p + 1, \sigma] \). Specifically, the \( i \)-th smallest s-character will be denoted by \( (i + \sigma_p) \). Moving forward, \( \map{pBWT}{i} \in [1, \sigma_p] \) iff \( L[i] \) is a p-character and \( \map{pBWT}{i} \in [\sigma_p + 1, \sigma] \) iff \( L[i] \) is a s-character. We summarize the relation between \( \map{prev}{\map{p\SigmaA}{i}} \) and \( \map{prev}{\map{p\SigmaA}{i} - 1} \) below.

**Observation 1.** Let \( 1 \leq i \leq n \). If \( \map{pBWT}{i} \in \Sigma_s \),

\[
\map{prev}{\map{p\SigmaA}{i} - 1} = \map{pBWT}{i} \circ \map{prev}{\map{p\SigmaA}{i}}[1, n - 1]
\]

Otherwise, if \( \map{pBWT}{i} \notin \Sigma_s \),

\[
\map{prev}{\map{p\SigmaA}{i} - 1} = 0 \circ \map{prev}{\map{p\SigmaA}{i}}[1, f_i - 1] \circ \map{f_i}{\circ \map{prev}{\map{p\SigmaA}{i}}}[f_i + 1, n - 1]
\]

### 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, \( \map{pLF}{i} = \map{pSA}{i} - 1 \). The significance is summarized in Theorem 3.

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

**Proof.** Define \( \map{pLF^0}{i} = i \) and \( \map{pLF^k}{i} = \map{pLF}{\map{pLF^{k-1}}{i}} \) for any integer \( k > 0 \). We maintain two \( \Delta \)-sampled arrays, one each for \( \map{pSA}{i} \) and \( \map{pSA}{i}^{-1} \). More specifically, we explicitly maintain \( \map{pSA}{j} \) and \( \map{pSA}{i}^{-1} \) 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 \( \map{pSA}{i} \), repeatedly apply the \( \map{pLF}{i} \) operation (starting from \( i \)) until you obtain a \( j \) such that \( \map{pSA}{j} \) has been explicitly stored. Suppose, the number of such operations is \( k \). Then, \( j = \map{pLF^k}{i} = \map{pSA}{i}^{-1} \map{pSA}{i} - k \), which
gives \( pSA[i] = pSA[j] + k \). Since \( k \leq \Delta \), \( pSA[i] \) is computed in \( O(\Delta \cdot t_{pLF}) \) time. To find \( pSA^{-1}[i] \), find the smallest \( j \geq i \) whose \( pSA^{-1}[j] \) is explicitly stored. Then, \( pSA^{-1}[i] = pLF^{-1}(pSA^{-1}[j]) \). As \( j - i \leq \Delta \), the time is bounded by \( O(\Delta \cdot t_{pLF}) \). ■

We remark that using Theorem 3, \( \text{prev}(T[x, y]) \) can be extracted in \( O(\Delta \cdot t_{pLF} + (y - x + 1)(t_{pLF} + \log \sigma)) \) 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 \( \ell_i \) and \( \ell_j \) in \( pST \). Then, \( pLF(i) \) and \( pLF(j) \) are related as follows:

(a) If \( L[i] \in \Sigma_p \) and \( L[j] \in \Sigma_s \), then \( pLF(i) < pLF(j) \).

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

- \( pBWT[i] < pBWT[j] \)
- \( pBWT[i] = pBWT[j] \) and \( i < j \).

(c) Assume both \( L[i], L[j] \in \Sigma_p \) and \( i < j \). Let \( u \) be the lowest common ancestor of \( \ell_i \) and \( \ell_j \) in \( pST \), and \( z \) be the number of \( 0 \)’s in the string \( \text{path}(u) \).

1. If \( pBWT[i], pBWT[j] \leq z \), then \( pLF(i) < pLF(j) \) iff \( pBWT[i] \geq pBWT[j] \).
2. If \( pBWT[i] \leq z < pBWT[j] \), then \( pLF(i) > pLF(j) \).
3. If \( pBWT[i] > z \geq pBWT[j] \), then \( pLF(i) < pLF(j) \).
4. If \( pBWT[i], pBWT[j] > z \), then \( pLF(i) > pLF(j) \) iff
   - \( pBWT[i] = z + 1 \),
   - the leading character on the \( u \) to \( \ell_i \) path is 0, and
   - the leading character on the \( u \) to \( \ell_j \) 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 \( T_{pSA[i]} \) and \( T_{pSA[j]} \) respectively. Let \( d = \lvert \text{path}(u) \rvert \). 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 \( T_{pSA[j]}[d + 1] \) is an \( s \)-character, then \( T_{pSA[j-1]}[d + 2] > T_{pSA[j]}[d + 2] \), and \( pLF(i) < pLF(j) \). So, assume otherwise. Let \( e_i \) and \( e_j \) be the \((d + 1)\)th characters of \( \text{prev}(T_{pSA[i]}) \) and \( \text{prev}(T_{pSA[j]}) \) 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 \( pBWT[i] = z + 1 \) and \( e_i = 0 \), then \( L[i] = T_{pSA[i]}[d + 1] \) i.e., \( f_i = d + 1 \), and \( \text{prev}(T_{pSA[j-1]}[d + 2]) = d + 1 > e_j = \text{prev}(T_{pSA[j-1]}[d + 2]) \). Otherwise, \( \text{prev}(T_{pSA[j-1]}[d + 2]) = e_i < e_j < \text{prev}(T_{pSA[j-1]}[d + 2]) \). ■

**Theorem 4.** We can compute \( pLF(i) \) 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. Specifically, given an array \( A \) over an alphabet \( \Sigma \), 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) \( \text{rank}_A(i, x) = \text{number of occurrences of } x \text{ in } A[i, i] \).

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

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

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 \( t_{WT} = O(1 + \log \sigma / \log \log n) \) for the above operations. As noted by Navarro [55], we can apply the technique of Golyński 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.

[4] Given a bit-vector \( B \) and \( c \in \{0, 1\} \), \( \text{rank}(i, c) = \{j \mid j \leq i \text{ and } B[j] = c\} \) and \( \text{select}(i, c) = \min \{j \mid \text{rank}(j, c) = i\} \).
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) \( \text{pre-order}(u)/\text{post-order}(u) = \text{pre-order/post-order rank of node } u \).

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

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

(d) \( \text{child}(u, q) = q\text{th leftmost child of node } u \).

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

(f) \( \text{leftmostLeaf}(u)/\text{rightmostLeaf}(u) = \text{leftmost/rightmost leaf in the subtree rooted at } u \).

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

Also, we can find the pre-order rank of the \( i \)th leftmost leaf in \( O(1) \) time. Moving forward, we will use \( \ell_i \) to denote the leaf corresponding to the \( i \)th lexicographically smallest pre-processed suffix.

4.2 ZeroDepth and ZeroNode

For a node \( u \), zeroDepth(\( u \)) is the number of 0’s in path(\( u \)). For a leaf \( \ell_i \) with \( \text{pBWT}[\ell_i] \in [1, \sigma_p] \), zeroNode(\( \ell_i \)) is the highest node \( z \) on the root to \( \ell_i \) path such that \( \text{zeroDepth}(z) \geq \text{pBWT}[\ell_i] \). Thus, \( z \) is the locus of path(\( \ell_i \))[1, \( f_i \)]. Note that \( z \) necessarily exists as \( \text{zeroDepth}(\ell_i) \geq \text{pBWT}(\ell_i) \). Moving forward, whenever we refer to zeroNode(\( \ell_i \)), we assume \( \text{pBWT}[\ell_i] \in [1, \sigma_p] \).

We present the following important lemma (proof deferred to Section 4.4.5).

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

We remark that the following additional functionalities: leafLeadChar(\( \cdot \)), fSum(\( \cdot \)) and 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 pLF(\( i \)) when pBWT[\( i \)] is \( [\sigma_p + 1, \sigma] \)

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

4.4 Computing pLF(\( i \)) when pBWT[\( i \)] is \( [1, \sigma_p] \)

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

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

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) \( S_1 \): leaves to the left of the subtree of \( v \).

(b) \( S_2 \): leaves in the subtree of \( v \).

(c) \( S_3 \): leaves to the right of the subtree of \( v \).

(d) \( S_4 \): leaves in the subtree of \( v \) but not of \( z \).

In case, \( v \) is the root node \( r \), we take \( w = r \); consequently, \( S_1 = S_3 = \emptyset \).

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

We partition the leaves into three sets:

(a) \( S_1 \) (resp. \( S_3 \)): leaves to the left (resp. right) of the subtree of \( z \).

(b) \( S_2 \): leaves in the subtree of \( z \).

We first compute \( z = \text{zeroNode}(\ell_i) \) (using Lemma 2), and then locate \( v = \text{parent}(z) \). Using leafLeadChar(\( i \)) and the \( \text{leftmostLeaf}(\cdot)/\text{rightmostLeaf}(\cdot) \) 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 \( \text{pLF}(i) \), we first compute \( N_1, N_2, \) and \( N_3 \), which are respectively the number of leaves \( \ell_j \) in the ranges \( S_1, S_2, \) and \( S_3 \) such that \( \text{pLF}(j) \leq \text{pLF}(i) \). Likewise, we compute \( N_4 \) (w.r.t. \( S_4 \)) if we are in the first case. Then, \( \text{pLF}(i) = N_1 + N_2 + N_3 + N_4 \).

4.4.3 Computing \( N_1 \)

For any leaf \( \ell_j \in S_1 \), \( \text{pLF}(j) < \text{pLF}(i) \) iff \( f_j > 1 + |\text{path}(\text{lca}(z, \ell_j))| \) and \( L[j] \in \sigma_p \). Therefore, \( N_1 \) is the number of leaves \( \ell_j \) in the ranges \( S_1, S_2, \) and \( S_3 \) such that \( \text{pLF}(j) \leq \text{pLF}(i) \). Define \( fCount(x) \) of a node \( x \) as the number of leaves \( \ell_j \) in \( x \)'s subtree such that \( |\text{path}(y)| + 2 \leq f_j \leq |\text{path}(x)| + 1 \), where \( y = \text{parent}(x) \). If \( x \) is the root node, then \( fCount(x) = 0 \). Define \( fSum(x) \) of a node \( x \) as \( \sum fCount(y) \) of all nodes \( y \) which come before \( x \) in pre-order and are not ancestors of \( x \). By this definition, \( N_1 = fSum(z) \) is computed as follows.
LEMMA 3. By maintaining an \( O(n) \)-bit structure, we can compute \( \text{fSum}(x) \) in \( O(1) \) time.

Proof. Traverse the \( \text{pST} \) in DFS order, and write \( \text{fCount}(v) \) of a node \( v \) in unary when exiting the node in the traversal, i.e., \( \text{fCount}(v) \) is associated with post-order \( (v) \). Maintain a rank-select structure on this bit-string \( B \). Since \( \sum \text{fCount}(v) \leq n \), \( |B| \leq 3n \), and the space needed is \( 3n + o(n) \) bits. Note that \( \text{fSum}(x) \) 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 \( \text{fSum}(x) = 0 \). Otherwise, \( y = \text{levelAncestor}(L_x - 1, \text{nodeDepth}(\text{Ica}(L_x, L_x - 1))) \) and \( \text{fSum}(x) = \text{rankB}(\text{selectB}(\text{post-order}(y), 0), 1) \). Clearly, the time required is \( O(1) \).

4.4.4 Computing \( N_2 \)
Note that for any leaf \( \ell_j \in S_2 \), \( \text{pLF}(j) \leq \text{pLF}(i) \) if \( L[j] \in \Sigma_p \) and either \( f_j > f_i \) or \( f_j = f_i \) and \( j \leq i \). Therefore, \( N_2 \) is the number of leaves \( \ell_j \in S_2 \) which satisfy one of the following conditions: (a) \( \text{pBWT}[i] < \text{pBWT}[j] \leq \sigma_p \), or (b) \( \text{pBWT}[i] = \text{pBWT}[j] \) and \( j \leq i \). Then, \( N_2 = \text{rangeCount}(L_x, R_x, c + 1, \sigma_p) + \text{rangeCount}(L_x, i, c, c) \), where \( c = \text{pBWT}[i] \).

4.4.5 Computing \( N_3 \)
For any leaf \( \ell_j \in S_3 \), \( \text{pLF}(j) > \text{pLF}(i) \). Thus, \( N_3 = 0 \).

4.4.6 Computing \( N_4 \)
Note that \( \text{pBWT}[i] \) is same as \( (\text{zeroDepth}(v) + 1) \). Consider a leaf \( \ell_j \in S_4 \) 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, \( \text{pLF}(j) < \text{pLF}(i) \) if \( f_j > f_i \), and the leading character on the path from \( v \) to \( \ell_j \) is not an s-character. For a node \( x \), \( \text{pCount}(x) \) is the number of children \( y \) of \( x \) such that the leading character from \( x \) to \( y \) is not an s-character. Note that \( \sum \text{pCount}(x) = O(n) \). Therefore, we encode \( \text{pCount}(x) \) of all nodes in \( O(n) \) bits using unary encoding, such that \( \text{pCount}(x) \) can be retrieved in constant time. Let \( u \) be the \( \text{pCount}(v) \)th child of \( v \). Then, \( N_4 \) is the number of leaves \( \ell_j \in S_4 \) such that \( j \leq R_u \) and \( \sigma_p \geq \text{pBWT}[j] \geq \text{pBWT}[i] \) i.e., \( N_4 = \text{rangeCount}(R_x + 1, R_u, \text{pBWT}[i], \sigma_p) \).

We summarize the LF mapping procedure in Algorithm 1. Once \( \text{zeroDepth}(\ell_i) \) 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 1.

4.5 Proof of Lemma 2
For any node \( x \) on the root to \( \ell_i \) path, define \( \alpha(x) = \) the number of leaves \( \ell_j \in \text{leaf}(x) \) such that \( L[j] \in \Sigma_p \) and \( f_j \leq |\text{path}(x)| \), and \( \beta(x) = \text{rangeCount}(L_x, R_x, 1, \text{pBWT}[i]) \). Here, \( \text{leaf}(x) \) is the range of leaves in the subtree of \( x \). Consider a node \( u_k \) on \( \pi \). Observe that \( \text{zeroNode}(\ell_i) \) is below \( u_k \) iff \( \beta(u_k) > \alpha(u_k) \). Therefore, \( \text{zeroNode}(\ell_i) \) 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 \( \pi \) (using \( \text{nodeDepth} \) and \( \text{levelAncestor} \) operations) to find the exact location. The first question is to compute \( \alpha(x) \), which is handled by Lemma 3. 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 \( |\log \sigma| \). Thus, the binary search has at most \( |\log \log \sigma| \) steps, and the total time is \( \log \sigma \times \left( \frac{|\log \sigma|}{\log \log \sigma} \right) = O(\log \sigma) \), 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 rank-select structure over \( A \). Define \( \gamma(v) \) as the number of leaves \( \ell_j \in \text{leaf}(v) \) that satisfy \( L[j] \in \Sigma_p \) and \( |\text{path}(\text{parent}(v))| < f_j \leq |\text{path}(v)| \). Traverse \( \text{pST} \) in DFS order, and write \( \gamma(v) \)
Algorithm 1 computes pLF(i)

1: \( c \leftarrow \text{pBWT}\{i\} \)
2: if \( (c > \sigma_p) \) then \( \text{pLF}(i) \leftarrow 1 + \text{rangeCount}(1, n, 1, c - 1) + \text{rangeCount}(1, i - 1, c, c) \)
3: else \( z \leftarrow \text{zeroNode}(\ell_j), v \leftarrow \text{parent}(z), L_z \leftarrow \text{lmostLeaf}(z), R_z \leftarrow \text{rmostLeaf}(z) \)
4: \( N_1 \leftarrow \text{fSum}(z), N_2 \leftarrow \text{rangeCount}(L_z, R_z, c + 1, \sigma_p) + \text{rangeCount}(L_z, i, c, c) \)
5: if \( \langle \text{leafLeadChar}(i) \rangle = 0 \) then
6: \( u \leftarrow \text{child}(v, \text{pCount}(v)), N_4 \leftarrow \text{rangeCount}(R_z + 1, \text{rmostLeaf}(u), c, \sigma_p) \)
7: \( \text{pLF}(i) \leftarrow N_1 + N_2 + N_4 \)

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

\[
\alpha'(x) = \text{rank}_B(\text{select}_B(\text{pre-order}(\ell_j), 0), 1) - \text{rank}_B(\text{select}_B(\text{pre-order}(x), 0), 1) \\
\alpha(x) = \text{rank}_A(R, 1) - \text{rank}_A(L - 1, 1) - \alpha'(x)
\]

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

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

**Proof.** Let \( g = \lceil \log \sigma \rceil \) be a sampling factor. We first mark all those nodes \( v \) in the \( \text{pST} \) such that \( \text{nodeDepth}(v) \) is a multiple of \( g \) and \( \text{subtree} \) 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 \( \text{zeroDepth} \) of the marked node corresponding to the \( k \)-th 1-bit in \( B \). Given a marked node with pre-order rank \( k' \), its corresponding position in \( D \) is given by \( \text{rank}_B(k', 1) \). We do not maintain \( D \) explicitly; instead, we maintain a wavelet tree over it. The space needed is \( O(n) \) bits.

Given a leaf node \( \ell_i \), 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 \( \text{zeroDepth}(u) < \text{pBWT}(i) \), then \( w_i = u \), and we are done. Otherwise, locate the rightmost position \( j' < j \) in \( D \) such that \( D[j'] < \text{pBWT}(i) \). Using the wavelet tree over \( D \), this predecessor search takes \( O(\log \sigma) \) time. (Since the root node is marked, and its \( \text{zeroDepth} \) equals 0, the position \( j' \) necessarily exists.) Obtain the marked node \( v \) corresponding to the \( j' \)-th 1-bit in \( B \) via a \( \text{select}(j', 1) \) operation. Then, \( w_i = \text{lca}(u, v) \). The time required is \( O(\log \sigma) \). To see the correctness, observe that \( \text{lca}(u, v) \) is an ancestor of \( \ell_i \). For a node \( x \), \( \text{zeroDepth}(x) \geq \text{zeroDepth}(\text{parent}(x)) \). Thus, \( \text{zeroDepth}(\text{lca}(u, v)) \leq \text{zeroDepth}(u) < \text{pBWT}(i) \).

| 5 Pattern Matching via Backward Search |

We modify the backward search algorithm in the FM-index [10]. In particular, given a proper suffix \( Q \) of \( P \), assume that we know the suffix range \([sp_1, ep_1]\) of \( \text{prev}(Q) \). Our task is to find the suffix range \([sp_2, ep_2]\) of \( \text{prev}(c \circ Q) \), where \( c \) is the character previous to \( Q \) in \( P \).

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

\[
sp_2 = 1 + \text{rangeCount}(1, n, 1, c - 1) + \text{rangeCount}(1, sp_1 - 1, c, c) \\
ep_2 = \text{rangeCount}(1, n, 1, c - 1) + \text{rangeCount}(1, ep_1, c, c)
\]

Now, we consider the scenario when \( c \) is parameterized. By maintaining a bit-vector \( B[1, \sigma_p] \), in \( O(\lceil |P| \rceil) \) 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 \( \text{pLF}(i) \in [sp_2, ep_2] \) iff \( i \in [sp_1, ep_1] \), \( L[i] \) is a \( p \)-character and \( f_i > |Q| \). This holds iff \( i \in [sp_1, ep_1] \) and \( \text{pBWT}(i) \in [d + 1, \sigma_p] \). 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 \( O(\lceil |P| \rceil) \) time. Then, \( (ep_2 - sp_2 + 1) = \text{rangeCount}(sp_1, \text{ep}_1, d + 1, \sigma_p) \).

Now, \( \text{pLF}(i) < sp_2 \) iff \( i < sp_1 \), \( L[i] \in \Sigma_p \), and \( f_i > 1 + \lceil \text{path}(\text{lca}(u, \ell_i)) \rceil \), where \( u = \text{lca}(\ell_{sp_1}, \ell_{ep_1}) \). Finally, we compute \( sp_2 = 1 + \text{fSum}(u) \) in constant time (refer to Lemma 3).
5.2 Case 2 (c appears in $Q$)

Note that $pLF(i) \in \{sp_2, ep_2\}$ iff $i \in \{sp_1, ep_1\}$. $L[i]$ is a p-character, and $f_j$ is the same as the first occurrence of $c$ in $Q$. This holds iff $i \in \{sp_1, ep_1\}$ and $PBWT[i] = d$. 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.

Consider $i, j \in \{sp_1, ep_1\}$ such that $i < j$ and $pLF(i), pLF(j) \in \{sp_2, ep_2\}$. Now, both $f_i$ and $f_j$ equals the first occurrence of $c$ in $Q$. Based on Observation 1, we conclude that $pLF(i) < pLF(j)$. Therefore, $sp_2 = pLF(i_{\text{min}})$ and $ep_2 = pLF(i_{\text{max}})$, where

$$i_{\text{min}} = \min\{j \mid j \in \{sp_1, ep_1\} \text{ and } PBWT[j] = d\}$$

$$= \text{select}(\text{rank}(sp_1, 1, d) + 1, d)$$

$$i_{\text{max}} = \max\{j \mid j \in \{sp_1, ep_1\} \text{ and } PBWT[j] = d\}$$

$$= \text{select}(\text{rank}(ep_1, d), d)$$

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

6 Dictionary Matching

Let us first look at the index of Idury and Schäffer [20]. For simplicity, we only consider p-characters, and defer the complete details to the full-version. We begin by obtaining $\text{prev}(P_i)$ for every $P_i \in D$, and then create a trie $T$ for all the encoded patterns. The number of nodes in the trie is $m \leq n + 1$. For each node $u$ in the trie, denote by $\text{path}(u)$ the string formed by concatenating the edge labels from root to $u$. Mark a node $u$ in the trie as final iff $\text{path}(u) = \text{prev}(P_i)$ for some $P_i \in D$. Clearly, the number of final nodes is $d$. For any prev-encoded string $\text{prev}(S)$ of a string $S$, and an integer $j \in [1, |S|]$, we obtain a string $\zeta(S, j)$ as follows. Initialize $\zeta(S, j) = \text{prev}(S)[j, |S|]$. For each $j' \in [1, |S| - j + 1]$, assign $\zeta(S, j)[j'] = 0$ iff $\zeta(S, j)[j'] \geq j'$. Conceptually, $\zeta(S, j) = \text{prev}(S[j, |S|])$. Each node $u$ is associated with the following 3 links:

(a) $\text{next}(u, c) = v$ iff the label on the edge from the node $u$ to $v$ is labeled by the character $c$,
(b) $\text{failure}(u) = v$ iff $\text{path}(v) = \zeta(\text{path}(u), j)$, where $j > 1$ is the smallest index for which such a node $v$ exists, and
(c) $\text{report}(u) = v$ iff $v$ is a final node and $\text{path}(v) = \zeta(\text{path}(u), j)$, where $j > 1$ is the smallest index for which such a node $v$ exists.

If no such $j$ exists, then $\text{failure}(u)/\text{report}(u)$ 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 $\text{prev}(u)$ in the set $\{\text{prev}(v) \mid v \text{ is a node in the trie}\}$, where $\text{prev}(u)$ is the string obtained by prev-encoding the path from $u$ to root. For any $e = (u, x)$, we define $Z(x) = \text{number of 0’s in } \text{prev}(w)[1, x]$, where $x$ is the first occurrence of the p-character labeling $e$ in the string from $w$ to root. (Note that $Z(x) \leq \sigma_p$.) If $x$ is not defined, we let $Z(x) = 0$.

We create a compressed $\overline{T}$ as follows. Initially $\overline{T}$ is empty. For each non-leaf node $u$ in $\overline{T}$ and for each child $u_i$ of $u$, we add the string $\overline{\text{prev}(u)} \circ u_i$ to $\overline{T}$. Clearly, each string corresponds to a leaf, say $\ell_{u,i}$, in $\overline{T}$. We order the leaves according to the (lexicographic) rank of the string they represent. For any two nodes $u, v \in \overline{T}$, the rank of a string $\overline{\text{prev}(u)} \circ u$ is smaller than that of $\overline{\text{prev}(v)} \circ v$ iff $u < v$. On the other hand, the rank of a string $\overline{\text{prev}(u)} \circ u$ is smaller than that of $\overline{\text{prev}(v)} \circ v$ iff $Z(u_i) > Z(u_j)$. (Note that $Z(u_i) \neq Z(u_j)$, as $u_i$ and $u_j$ share the same parent.) If two leaves have distinct parents, then their order is defined by the relation $<$. 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 $\overline{T}$. Let the respective characters (from $\Sigma$) on the edges be $c_i$ and $c_j$. Assume either $u < v$ or $u = v$. Let $x = \text{lca}(u, v)$, and $z$ be the number of 0’s in $\text{prev}(x)$.

(a) If $Z(u_i) \leq z$, then $\text{next}(u, c_i) < \text{next}(v, c_j)$ iff $Z(u_i) < Z(v_j)$.
(b) If $Z(u_i) < Z(v_j)$, then $\text{next}(v, c_j) < \text{next}(u, c_i)$.
(c) $Z(u_i) \leq z < Z(v_j)$, then $\text{next}(u, c_i) < \text{next}(v, c_j)$.
(d) If $Z(u_i) > Z(v_j)$, then $\text{next}(v, c_j) < \text{next}(u, c_i)$ iff
   - $Z(u_i) = z + 1$, and
   - the leading character on the path from $x$ to $\ell_{u,i}$ is 0.
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 $\text{failure}(u) = v$, then it $\text{prev}(v)$ is the longest prefix of $\text{prev}(u)$ that appears in $T$. Similar remarks hold for the report $(u) = v$, where $v$ is a final node. Therefore, these behave exactly in the same manner as in the case of the 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 $n H_k(T)$, where $H_k(T)$ denotes the $k$th 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


