# Circular Dictionary Matching Using Extended BWT

**Wing-Kai Hon**
National Tsing Hua University, Hsinchu, Taiwan

**Rahul Shah**
Louisiana State University, Baton Rouge, LA, USA

**Sharma V. Thankachan**
North Carolina State University, Raleigh, NC, USA

## Abstract

The *dictionary matching problem* involves preprocessing a set of strings (patterns) into a data structure that efficiently identifies all occurrences of these patterns within a query string (text). In this work, we investigate a variation of this problem, termed *circular dictionary matching*, where the patterns are circular, meaning their cyclic shifts are also considered valid patterns. Such patterns naturally occur in areas such as bioinformatics and computational geometry. Based on the *extended* Burrows-Wheeler Transformation (eBWT), we design a space-efficient solution for this problem. Specifically, we show that a dictionary of $d$ circular patterns of total length $n$ can be indexed in $n \log \sigma + O(n + d \log n + \sigma \log n)$ bits of space and support circular dictionary matching on a query text $T$ in $O((|T| + \mathsf{occ}) \log n)$ time, where $\sigma$ represents the size of the underlying alphabet and $\mathsf{occ}$ represents the output size.

**2012 ACM Subject Classification** Theory of computation → Pattern matching

**Keywords and phrases** String algorithms, Burrows-Wheeler transformation, suffix trees, succinct data structures

**Digital Object Identifier** 10.4230/OASIcs.Manzini.2025.11

**Funding** Supported by the US National Science Foundation (NSF) under Grant Numbers 2137057, 2434261 (R Shah) and 2315822 (S Thankachan).

**Acknowledgements** We thank all the anonymous reviewers (and Mano Prakash Parthasarathi) for their valuable feedback, which helped improve this paper. We also thank Travis Gagie for pointing out the independent work by Cotumaccio [25].

## 1 Introduction

Text indexing is a fundamental problem in computer science, where we are given a long string (the text) for preprocessing into a data structure (the index) that supports efficient substring matching. Specifically, when a shorter string (the pattern) is provided as a query, the goal is to find all occurrences of the pattern as a substring within the text. Data structures such as suffix trees and suffix arrays are commonly used for this task [67, 53]. However, a major drawback of suffix trees and suffix arrays is their significant space requirement. To address this issue, compact and space-efficient solutions have been developed [28, 41, 62]. Among these, FM-index [28], Compressed Suffix Arrays [41], and Compressed Suffix Trees [62] are particularly notable due to their historical significance. We refer to [57] for a comprehensive survey on this topic.

Dictionary matching is an orthogonal problem to text indexing. Here, we are given a set of patterns (the dictionary) and our aim is to design a data structure capable of finding all occurrences of these patterns as substrings within a query text. Solutions to this problem have applications in areas such as intrusion detection and bioinformatics, where they are used to identify known DNA or protein sequences in genomic data. The classic

© Wing-Kai Hon, Rahul Shah, and Sharma V. Thankachan;
licensed under Creative Commons License CC-BY 4.0
The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday.
Editors: Paolo Ferragina, Travis Gagie, and Gonzalo Navarro; Article No. 11; pp. 11:1–11:14
OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

solution to this problem is the Aho-Corasick automaton [1], which efficiently matches multiple patterns simultaneously. However, as before, this data structure also suffers from a high space requirement. To address this issue, compact and space-efficient solutions have been proposed. The current best succinct-space result is due to Belazzougui [10], where an xBWT-like technique [27] is applied for the succinct encoding of the Aho-Corasick automaton. We refer the reader to [55, 59, 16] for follow-up work in this direction, including an entropy-compressed solution [45]. For an an alternative solution based on sparse suffix trees, see [42]. A wide range of variations on this problem have been studied, including dynamic dictionary matching [3, 18, 46, 26, 65], online dictionary matching [35, 51, 6], dictionary matching in streaming model [36, 34, 37], dictionary matching with errors or gaps [5, 4, 24, 44, 47], internal dictionary matching [19], dictionary matching under parameterized or order-preserving models [52, 32, 33], 2D dictionary matching [2, 58], etc.

In this work, we investigate another variant of dictionary matching, termed *circular dictionary matching* [48]. Here the patterns in the given dictionary are circular, meaning their cyclic shifts are also considered valid patterns. For example, the set of cyclic shifts of `abcd` is $\{\texttt{abcd}, \texttt{bcda}, \texttt{cdab}, \texttt{dabc}\}$, whereas that of `abab` is $\{\texttt{abab}, \texttt{baba}\}$. Note that circular patterns arise naturally in applications in bioinformatics and computational geometry. For instance, the genomes of many viruses, such as the herpes simplex virus (HSV-1), exist as circular strings [64]. In computational geometry, polygons are often represented by listing the coordinates of their vertices in clockwise order. The problem of matching a circular pattern, or a collection of circular patterns, in a given text has been extensively studied from an algorithmic perspective [23, 22, 20, 7, 49, 9, 40, 21]. Our objective is to design a solution for the data-structural version of this problem.

▶ **Problem 1** (Circular Dictionary Matching [48]). *Given a set of $d$ circular patterns $\mathcal{D} = \{P_1, P_2, \ldots, P_d\}$ of total length $n$ on an alphabet $\Sigma = [\sigma]$, design a data structure (called an index) that supports the following query efficiently:*

- *Input: A text $T$ over $\Sigma$*
- *Output: All occurrences of all circular patterns in $T$. Specifically, every substring of $T$ that corresponds to a cyclic shift of any pattern in $\mathcal{D}$; a substring can be denoted by the starting and ending position in the text. We use* $\mathsf{occ}$ *to denote the output size.*

Problem 1 is equivalent to standard dictionary matching on an expanded dictionary $\mathcal{D}'$, which includes all circular patterns in $\mathcal{D}$ along with their cyclic shifts. Therefore, one approach is to index $\mathcal{D}'$, which is clearly inefficient, as the sum of the sizes of all patterns in $\mathcal{D}'$ can be quadratic to that of $\mathcal{D}$ in the worst case. To that end, we presented a succinct space index of space $n \log \sigma + O(n + d \log n + \sigma \log n)$ bits and query complexity $O(|T| \log^2 n + \mathsf{occ} \log n)$ in our earlier work [48], which we improve upon in this paper.

▶ **Theorem 2.** *For the circular dictionary matching problem, there exists an index requiring space $n \log \sigma + O(n + d \log n + \sigma \log n)$ bits and query time $O((|T| + \mathsf{occ}) \log n)$. Here $n$ denotes the total length of circular patterns in the dictionary $\mathcal{D}$, $d = |\mathcal{D}|$, $\sigma$ denotes the size of the alphabet set, $T$ denotes the text (and $|T|$ denotes its length) that comes as a query, and* $\mathsf{occ}$ *denotes the output size.*

Our new solution is based on a structure similar to the FM-index [28], which is built from the *extended Burrows-Wheeler Transformation* (eBWT) [54], an extension of the traditional Burrows-Wheeler Transformation (BWT) [15] that works over multiple strings [8, 13, 14, 17, 60]. We remark that the application of the eBWT to the space-efficient indexing of circular patterns is not new. For example, in recent work, Boucher et al. [14] showed how to construct an eBWT-based index for a collection of strings with full cyclic pattern matching functionality in compressed space. This problem is distinct from the one we address in this work; nonetheless, the techniques employed in this paper are closely similar to those in their work.

▶ Remark. In work parallel to ours, Cotumaccio [25] independently achieved an index with similar space-time complexity; specifically $n \log \sigma(1 + o(1)) + O(n + d \log n)$ bits of space and $O((|T| + \mathsf{occ}) \log n)$ query time.

## 2 Preliminaries

Let $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$ denote the underlying alphabet. For a string $S[1 \,..\, s]$, we use $|S|$ to denote its length, $S[i \,..\, j]$ to denote its substring $S[i] \circ S[i+1] \circ \cdots \circ S[j]$ when $i \leq j$ and an empty string when $i > j$, where $\circ$ denotes concatenation. In addition, $S[i \,..\, j) = S[i \,..\, j-1]$ and $S(i \,..\, j] = S[i+1 \,..\, j]$. For an integer $k \geq 0$, $S^k$ denotes an empty string if $k = 0$, and $S \circ S^{k-1}$ otherwise, while $S^\infty$ denotes the concatenation of infinite copies of $S$. The *root* of a string $S$, $root(S)$ is defined as the shortest string $R$, such that $S = R^k$ for some integer $k$; we call $S$ *primitive* if $S = root(S)$. For example, $\mathtt{a}, \mathtt{ab}, \mathtt{ababa}$ are primitive, whereas $\mathtt{abab} = (\mathtt{ab})^2$ is not. The result below follows from Fine and Wilf's theorem [29] (also see Theorem 1 in [12]).

▶ **Lemma 3.** *Let $X$ and $Y$ be two* distinct *primitive strings. Then, the length of the longest common prefix of their infinite repetitions, $X^\infty$ and $Y^\infty$, is at most $|X| + |Y| - \gcd(|X|, |Y|)$, where $\gcd(\cdot, \cdot)$ denotes the greatest common divisor. This implies that although $X^\infty$ and $Y^\infty$ are infinite in length, their lexicographic order can be established by comparing a bounded number of characters.*

**Rank and Select Queries.** For a character $\alpha \in \Sigma$, $\mathsf{rank}_S(i, \alpha)$ denotes the number of occurrences of $\alpha$ in $S[1 \,..\, i]$, $\mathsf{select}_S(j, \alpha)$ denotes the location of $j$-th occurrence of $\alpha$ in $S$, and $\mathsf{partialRank}_S(i) = \mathsf{rank}_S(i, S[i])$. There exist different space-time trade-offs for supporting these operations. For example, a wavelet tree structure of space $s \log \sigma + o(s) + O(\sigma \log s)$ bits can support all three operations in $O(\log \sigma)$ time [39]. See [11, 38] for faster alternatives.

For our purpose, we use an $s \log \sigma + O(s + \sigma \log s)$-bit structure, which requires $O(\log \log s)$ time for rank and only $O(1)$ time for others. The idea is to use the following result (indexible dictionaries) by Raman et al. [61]: a binary string $B[1 \,..\, s]$ with $f$ number of 1s can be represented in $f \log(s/f) + O(f)$ bits and support $\mathsf{rank}_B(\cdot, \cdot)$ in $O(\log \log s)$ time, $\mathsf{select}_B(j, 1)$ and $\mathsf{partialRank}_B(i)$ in $O(1)$ time, where $\mathsf{partialRank}_B(i) = \mathsf{rank}_B(i, B[i])$ if $B[i] = 1$ and is an arbitrary number otherwise. Therefore, for each $\alpha \in \Sigma$, we maintain the following bit vectors as indexible dictionaries: $B_\alpha[1 \,..\, s]$, where $B_\alpha[i] = 1$ iff $S[i] = \alpha$. Let $f_\alpha$ be the number of occurrences of $\alpha$ in $S$. Then the total space is $\sum_{\alpha \in \Sigma} f_\alpha \log(s/f_\alpha) + O(f_\alpha) = s \log \sigma + O(s)$ bits. To enable the operations on $S$, we employ the following connections: $\mathsf{rank}_S(i, \alpha) = \mathsf{rank}_{B_\alpha}(i, 1)$, $\mathsf{select}_S(j, \alpha) = \mathsf{select}_{B_\alpha}(j, 1)$, and $\mathsf{partialRank}_S(i) = \mathsf{partialRank}_{B_{S[i]}}(i)$.

**Range Minimum Queries (RMQ).** Let $A$ be an array of numbers, we can design an $2|A| + o(|A|)$ bit structure that supports the following query [30] in constant time: input is a range $[a, b]$ and output is position $t \in [a, b]$, such that $A[t]$ is the smallest element in $A[a, b]$. For answering RMQ, we do not need to explicitly store $A$.

**Tree Topology.** The topology of an ordinal tree can be encoded in linear number of bits and support various tree operations in $O(1)$ time [63, 56]. The ones relevant to us are finding the parent of a node, the range of leaves (i.e., the first and last leaves) in the subtree of a node, and the lowest common ancestor (LCA) of two nodes. Here a node is referred to by its pre-order rank.

## 3 Extended Burrows-Wheeler Transformation and Related Structures

In this section, we formally define the *extended BWT*, along with data structures analogous to suffix trees and suffix arrays for circular strings, which we refer to as the *extended suffix tree* and *extended suffix array*. The definition of extended BWT is first proposed by Mantaci et al. [54]. For related concepts, different terminologies have been used in prior work; for example, the extended suffix array was referred to as the circular suffix array in [48, 43], and as the generalized conjugate array in recent work by Boucher et al. [14]. However, it is straightforward to observe that these structures are equivalent. As we will show, the succinct encoding of the extended suffix array follows naturally from the original ideas of the FM-index. However, the definition of the extended suffix tree, or the use of it to solve pattern matching problems, seem to be new. In the following, we will follow the terminologies in [48, 43] to define the extended BWT, as they align more naturally in defining (more importantly *compressing*) the extended suffix tree than those in the original paper [54]. Also, we remark that encoding certain components of the extended suffix tree requires non-trivial modifications of known results.

Let $\mathcal{Q} = \{Q_1, Q_2, Q_3, \ldots, Q_r\}$ be a given set of $r$ primitive strings, where $Q_i = Q_i[1\mathinner{..}q_i]$ and $m = \sum_{i=1}^{r} q_i$. We call $Q_{i,k} = Q_i[k\mathinner{..}q_i] \circ Q_i[1\mathinner{..}k)$ the $k$-th *cyclic shift* of $Q_i$, where $k \in [1, q_i]$. Without loss of generality, we make an important assumption that no string in $\mathcal{Q}$ is a cyclic shift of another.[1] Next, we define the following sets:

$$\text{eSUFFIXES}(Q_i) = \{Q_{i,k}^{\infty} \mid k \in [1, q_i]\} \text{ and } \text{eSUFFIXES}(\mathcal{Q}) = \bigcup_{i=1}^{r} \text{eSUFFIXES}(Q_i)$$

Note that eSUFFIXES($\mathcal{Q}$) is a collection of $m$ strings, which are infinite in length, but pairwise distinct; therefore, their lexicographic order is well-defined. Moreover, by Fine and Wilf's theorem (see Lemma 3), the length of the longest common prefix between any two strings in eSUFFIXES($\mathcal{Q}$) is at most $2m$.

The **Extended Suffix Tree** (denoted by eST) is a compacted trie of all strings in eSUFFIXES($\mathcal{Q}$). It consists of $m$ leaves, say $\ell_1, \ell_2, \ldots, \ell_m$ in the left-to-right order, which are enumerated according to the lexicographic order of the infinite strings in eSUFFIXES($\mathcal{Q}$). The number of internal nodes is at most $m - 1$. For any node $u$, $\mathsf{str}(u)$ denotes the concatenation of edge labels on the path from the root to $u$ and $\mathsf{strlen}(u)$ denotes the length of $\mathsf{str}(u)$. Note that $\mathsf{strlen}(\cdot) < 2m$ for all internal nodes from Lemma 3. However, each edge connecting to a leaf has a label with an infinite length. This is unlike the standard suffix tree. Yet, the tree topology is well-defined, because the lexicographic order of strings in eSUFFIXES($\mathcal{Q}$) is well defined[2]. We remark that the idea of having leaf edges with

[1] As we will see, our goal is to efficiently represent all cyclic shifts of strings in the collection. If two strings are cyclic shifts of each other, they share the same set of shifts. Therefore, it's sufficient to store only one representative from each group of cyclically equivalent strings.

[2] Note that this remains true even when each infinite string in eSUFFIXES($\mathcal{Q}$) is treated as a finite string by considering only its first $2m$ characters.

unspecified lengths is not new; e.g., see Ukkonen's suffix tree construction algorithm [66]. The *suffix range* of a string $S$ is the maximal range $[sp, ep]$ where $S$ is a prefix of $\mathsf{str}(\ell_j)$ for all $j \in [sp, ep]$, and is NIL if no such $j$ exists. The *range of leaves* of a node $u$ is the suffix range of the string $\mathsf{str}(u)$.

The **Extended Suffix Array** $\mathrm{eSA}[1..m]$ is an array of pairs such that $\mathrm{eSA}[j] = (i, k)$, where $\mathsf{str}(\ell_j) = Q^{\infty}_{i,k}$, and the **Extended BWT** $\mathrm{eBWT}[1..m]$ is a string, where $\mathrm{eBWT}[j]$ is the last character of $Q_{i,k}$. We remark that the terminology used in the original definition of eBWT in [54] is slightly different (i.e., omega order), but there is no technical difference. Note that eST and eSA take $O(m \log m)$ bits, whereas eBWT takes only $m \log \sigma$ bits.

As an example, consider the set $\mathcal{Q} = \{Q_1, Q_2, Q_3, Q_4\}$, where $Q_1 = aab, Q_2 = ab, Q_3 = abb$ and $Q_4 = b$. Then eST consists of 9 leaves, such that

1. $\mathsf{str}(\ell_1) = Q^{\infty}_{1,1} = aabaab\dots$
2. $\mathsf{str}(\ell_2) = Q^{\infty}_{1,2} = abaaba\dots$
3. $\mathsf{str}(\ell_3) = Q^{\infty}_{2,1} = ababab\dots$
4. $\mathsf{str}(\ell_4) = Q^{\infty}_{3,1} = abbabb\dots$
5. $\mathsf{str}(\ell_5) = Q^{\infty}_{1,3} = baabaa\dots$
6. $\mathsf{str}(\ell_6) = Q^{\infty}_{2,2} = bababa\dots$
7. $\mathsf{str}(\ell_7) = Q^{\infty}_{3,3} = babbab\dots$
8. $\mathsf{str}(\ell_8) = Q^{\infty}_{3,2} = bbabba\dots$
9. $\mathsf{str}(\ell_9) = Q^{\infty}_{4,1} = bbbbbb\dots$

The corresponding $\mathrm{eSA}[1..9]$ is $(1,1),(1,2),(2,1),(3,1),(1,3),(2,2),(3,3),(3,2),(4,1)$ and $\mathrm{eBWT}[1..9]$ is $babbaabab$.

## 3.1 Succinct Representation of eSA

We maintain eBWT in $m \log \sigma + O(m + \sigma \log m)$ bits to support $\mathsf{rank}, \mathsf{select}$ and $\mathsf{partialRank}$ operations efficiently as described in Section 2. Similar to the FM-index, we now implement LF mapping and backward search [28].

**Last-to-Front (LF) Mapping.** Define $\mathrm{eLF}(j) = j'$, where deleting the first character of $\mathsf{str}(\ell_{j'})$ gives $\mathsf{str}(\ell_j)$. The function $\mathrm{eLF}(j)$ can also be described in terms of eBWT as $\mathrm{eLF}(j) = \mathsf{count}(\mathrm{eBWT}[j]) + \mathsf{partialRank}_{\mathrm{eBWT}}(j)$. For any character $\alpha \in \Sigma$, $\mathsf{count}(\alpha) = |\{i \in [1, m] \mid \mathrm{eBWT}[i] < \alpha\}|$. For all $\alpha \in \Sigma$, this information can be stored explicitly in $O(\sigma \log m)$ bits, and the $\mathsf{partialRank}$ operation takes only $O(1)$ time. Therefore, the overall time for eLF operation is constant.

**Backward Search.** Given the suffix range $[sp, ep]$ of a string $S$, we can compute the suffix range $[sp', ep']$ of $\alpha \circ S$ for any $\alpha \in \Sigma$ in $O(\log \log m)$ time as follows. First compute the following

$$sp' = \mathsf{count}(\alpha) + \mathsf{rank}_{\mathrm{eBWT}}(sp - 1, \alpha) + 1$$

$$ep' = \mathsf{count}(\alpha) + \mathsf{rank}_{\mathrm{eBWT}}(ep, \alpha)$$

If $sp' \leq ep'$, then return $[sp', ep']$ as the suffix range of $\alpha \circ S$. Otherwise, conclude that the suffix range of $\alpha \circ S$ does not exist. The time complexity $O(\log \log m)$ comes from the time for rank operations.

To encode eSA, we replace it with a sparse eSA, where we store eSA values only at some sampled positions. Specifically, we store an entry $(i, k)$ iff $k \bmod \Delta = 1$, where $\Delta$ be a parameter. This reduces the number of values stored to $O(d + m/\Delta)$, where $d$ is the number of indexed patterns; space is $O(\log m)$ bits per value. Suppose $\mathrm{eSA}[j] = (i, k)$, then $\mathrm{eSA}[\mathrm{eLF}[j]] = (i, k-1)$, $\mathrm{eSA}[\mathrm{eLF}[\mathrm{eLF}[j]]] = (i, k-2)$, etc., which means we can decode $\mathrm{eSA}[j]$ (if not explicitly stored) by iteratively applying eLF (say $h$ times) until we arrive at a positions where $\mathrm{eSA}[\cdot] = (i, k')$ is stored. We then obtain $\mathrm{eSA}[j] = (i, k' + h)$. The time complexity is $O(h)$ and $h < \Delta$. By fixing $\Delta = \lceil \log m \rceil$, we obtain the following result.

▶ **Lemma 4.** *There exists an $m \log \sigma + O(m + d \log m + \sigma \log m)$-bit data structure that returns $\mathrm{eSA}[j]$ for any given $j$ in time $O(\log m)$.*

## 3.2 Succinct Representation of eST

We encode and maintain the topology of eST in $O(m)$ bits for supporting the relevant tree operations in $O(1)$ time. To encode the values of $\mathsf{strlen}(\cdot)$, we modify the techniques introduced by Sadakane [62].

For all $i \in [1, r]$, define $\mathrm{ePLCP}^i[1\,..\,q_i]$, where $\mathrm{ePLCP}^i[k]$ is the length of the longest common prefix of $Q^\infty_{i,k}$ and the string in the set $\mathrm{eSUFFIXES}(\mathcal{Q})$ that comes next in the lexicographic order (say $Q^\infty_{i',k'}$). Note that longest common prefix of $Q^\infty_{i,(k+1) \bmod q_i}$ and $Q^\infty_{i',(k'+1) \bmod q_{i'}}$ shares at least $\mathrm{ePLCP}^i[k] - 1$ characters. So, $\mathrm{ePLCP}^i[(k+1) \bmod q_i] \geq \mathrm{ePLCP}^i[k] - 1$. By adding $k$ on both sides and rearranging, we have

$$\mathrm{ePLCP}^i[k \bmod q_i] + k \leq \mathrm{ePLCP}^i[(k+1) \bmod q_i] + (k+1)$$

Letting $f_i(k) = \mathrm{ePLCP}^i[k \bmod q_i] + k$, we have

$$f_i(1) \leq f_i(2) \leq \cdots \leq f_i(|Q_i|) \leq f_i(|Q_i| + 1) = f_i(1) + |Q_i|$$

This means $f_i(k) : k \in [1, q_i]$ is non-decreasing and its range is $[f_i(1), f_i(1) + |Q_i|]$. We store $f_i(1)$ explicitly in $O(\log m)$ bits and use unary encoding for the rest. Specifically, for each $i$ we store a binary string $B_i = \mathtt{1}\mathtt{0}^{f_i(2)-f_i(1)}\mathtt{1}\mathtt{0}^{f_i(3)-f_i(2)}\mathtt{1}\mathtt{0}^{f_i(4)-f_i(3)}\ldots$ with $O(1)$ time rank/select supported [61]. To find $f_i(k)$, we simply count the number of $\mathtt{0}$s before $k$-th $\mathtt{1}$ and add $f_i(1)$. Subtracting $k$ from this value gives $\mathrm{ePLCP}^i[k]$. Space required for a fixed $i$ is $O(|B_i| + \log m)$, where $|B_i| = O(q_i)$. Therefore, space over all $i$'s is $O(m + d \log m)$ bits.

We now present the last component for encoding $\mathsf{strlen}(\cdot)$ values. Define array $\mathrm{LCP_e}[1\,..\,m)$ as follows. Suppose $\mathrm{eSA}[j] = (i, k)$, then $\mathrm{LCP_e}[j] = \mathrm{ePLCP}^i[k]$. We do not store $\mathrm{LCP_e}$ explicitly, but an RMQ structure over it in $2m + o(m)$ bits. To compute $\mathsf{strlen}(u)$, find $[a, b]$, the range of in the subtree of $u$ and the position $t \in [a, b)$ corresponding to the minimum in $\mathrm{LCP_e}[a, b)$ using an RMQ. Then obtain $\mathrm{eSA}[t] = (i', k')$ and $\mathsf{strlen}(u) = \mathrm{ePLCP}^{i'}[k']$. The time complexity (asymptotically) is equal to that of an eSA query.

▶ **Lemma 5.** *By associating an $O(m + d \log m)$ bit structure with eSA, we can find $\mathsf{strlen}(\cdot)$ of any given node in time $O(\log m)$.*

This concludes our discussion of the main data structure. In the following section, we demonstrate how it can be used to solve the circular dictionary matching problem.

# 4 A Succinct Index for Circular Dictionary Matching

Traditional "non-circular" dictionary matching can be solved efficiently with a generalized suffix tree of the patterns, by successively finding the *locus* of each suffix of the query text $T$ within the suffix tree, and then reporting the patterns that appear as the prefix of the corresponding suffix. For circular patterns, we show that this can be done analogously with the extended suffix tree. Yet, there are some technical challenges. First, as patterns of different lengths can be represented by the same leaf, we need some care to verify if a pattern actually appears in a certain text position. Another, more difficult, challenge is to obtain the loci of the suffixes; a direct adaptation of the "non-circular" dictionary matching approach cannot bound the running time as desired. In the following, we will explain how such challenges can be solved.

Given a set of $d$ circular patterns to index. We first make a critical step of replacing each pattern with its lexicographically smallest cyclic shift. Denote the resulting set by $\mathcal{D} = \{P_1, P_2, \ldots, P_d\}$. We then collect the roots of all $P_i$'s and call it $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_r\}$. The first step ensures that $\mathcal{Q}$ is invariant of the cyclic shifts of $P_i$'s. Let $n = \sum_i |P_i|$ and $m = \sum_i |Q_i|$. Note that $r \leq d$ and $m \leq n$.

## 4.1 The Data Structure

We construct and maintain the structures in Lemma 4 and Lemma 5 over the strings in $\mathcal{Q}$. Additional components are below.

1. We encode each pattern $P_i \in \mathcal{D}$ as a pair $(i', |P_i|)$, where $Q_{i'}$ is the root of $P_i$, equivalently $P_i = Q_{i'}^{|P_i|/|Q_{i'}|}$. For each $Q_{i'} \in \mathcal{Q}$, we maintain a list $\text{LIST}_{i'}$ of patterns (in encoded form) with $Q_{i'}$ being their root. Each list is sorted in the ascending order of the pattern lengths. Total space is $O(d \log n)$ bits. Therefore, given any $(i', \tau)$, we can list all patterns with root $Q_{i'}$ and length $\leq \tau$ in optimal time.
2. We want to support the query $\text{REPORT}([a, b], \tau)$, which reports all patterns $P_i = (i', |P_i|)$ (i.e., in encoded form), where $|P_i| \leq \tau$ and $\text{eSA}[j] = (i', \cdot)$ for some $j \in [a, b]$. Define an array $\mathsf{Length}[1, m]$, where $\mathsf{Length}[j]$ is the length of the *shortest* pattern in $\text{LIST}_{i'}$, where $\text{eSA}[j] = (i', \cdot)$. We do not store this array, but an RMQ structure over it in $2m + o(m)$ bits. With that, we support the query using the following standard procedure.
   - When $a = b$, we decode $i'$, where $\text{eSA}[a] = (i', \cdot)$ and obtain the output from $\text{LIST}_{i'}$.
   - When $a \neq b$, we find the position $t \in [a, b]$ of the minimum element in $\mathsf{Length}[a, b]$ using RMQ, and then make the query $\text{REPORT}([t, t], \tau)$. If it returns NIL (i.e., output is empty), we conclude that $\text{REPORT}([a, b], \tau)$ is also NIL. Otherwise, we continue our search for more answers in the remaining parts of the array, specifically in $[a, t)$ and $(t, b]$ using queries $\text{REPORT}([a, t), \tau)$ and $\text{REPORT}((t, b], \tau)$, recursively.

   Let $g$ be the output size. Then it can be observed that the original query is split into $O(1 + g)$ subqueries. Therefore, the time complexity is $O((1 + g) \log m)$.
3. We mark a node $u$ in eST if it is the highest node (i.e., closest to root) for some $i \in [1, d]$, such that $P_i$ or a cyclic shift of it is a prefix of $\mathsf{str}(u)$. With each node, we associate a bit, indicating if it is marked or not. Next, we want to support the following query: given an arbitrary node $u$, list its marked ancestors. One could easily accomplish this by first finding all of its ancestors (via finding parent nodes iteratively, starting $u$) and then extracting those that are marked. But the time taken could be $\Theta(m)$ in the worst case. To bound the time in terms of the number of marked ancestors, we store the lowest marked ancestor (LMA) of the following sampled nodes explicitly: $\text{LCA}(\ell_{t \log m}, \ell_{(t+1) \log m})$ for all $t \in [1, m/\log m)$. This scheme requires $O(m)$ extra bits and guarantees that any path towards the root with $\log m$ nodes contains a sampled node. Therefore, to list all the marked ancestors of $u$, we traverse the path from $u$ to root as before with the difference that when we are at a sampled node, we jump to its LMA (i.e., skip all nodes in between) and continue. All marked ancestors will be visited and the total number of nodes visited (and total time) is bounded by $O(\log m)$ times the number of marked ancestors.

Total space is $m \log \sigma + O(m + d \log n + \sigma \log m) \subseteq n \log \sigma + O(n + d \log n + \sigma \log n)$ bits.

## 4.2 Query Algorithm

We report all $(x, i)$, where $x \in [1, |T|]$ and (*a cyclic shift of*) $P_i \in \mathcal{D}$ occurs at position $x$ in $T$, using the following steps.

1. Find the maximum $L_x$, such that there exists a node $u_x$ where $T[x \mathinner{.\,.} x + L_x)$ is a prefix of $\mathsf{str}(u_x)$ and $u_x$ is the highest such node. Also, let $[sp_x, ep_x]$ be the range of leaves of $u_x$.
2. Report $(x, i)$, if *a cyclic shift of* $P_i \in \mathcal{D}$ is a prefix of $\mathsf{str}(u_x)$ and its length is $\leq L_x$.

Note that the first step corresponds to finding the loci of $T[x..]$ in the extended suffix tree, for each $x$. For traditional suffix tree approach, we find the loci in *ascending* order of $x$. Here, we do so in the opposite manner, in *descending* order of $x$. Intuitively, this change allows us to replace the "downward" traversal of tree edges in the extended suffix tree to "upward" traversals, where the latter can be implemented more efficiently with our auxiliary data structures. We now show how to implement these two steps efficiently.

### 4.2.1 Details of Step 1

Initialize $x = |T|$, $L_{x+1} = 0$, and fix $[sp_{x+1}, ep_{x+1}] = [1, m]$. Then we compute $(L_x, u_x, sp_x, ep_x)$ in the descending order of $x$ using backward search as follows. We have two cases.

1. If $T[x]$ has an occurrence in $\text{eBWT}[sp_{x+1}, ep_{x+1}]$, then find the smallest number $a$ and the largest number $b$ in $[sp_{x+1}, ep_{x+1}]$, such that $\text{eBWT}[a] = \text{eBWT}[b] = T[x]$. Then obtain $sp_x = \text{eLF}[a], ep_x = \text{eLF}[b], u_x = \text{LCA}(\ell_{sp_x}, \ell_{ep_x})$ and $L_x = 1 + L_{x+1}$. The time complexity in this case is $O(\log\log m)$.
2. If $T[x]$ has no occurrence in $\text{eBWT}[sp_{x+1}, ep_{x+1}]$, then we find the lowest ancestor of $u_{x+1}$, say $w$, such that $T[x]$ has an occurrence in $\text{eBWT}[z, z']$, where $[z, z']$ is the range of leaves below $w$. Then find the smallest number $a$ and the largest number $b$ in $[z, z']$, such that $\text{eBWT}[a] = \text{eBWT}[b] = T[x]$. Then obtain $sp_x = \text{eLF}[a], ep_x = \text{eLF}[b], u_x = \text{LCA}(\ell_{sp_x}, \ell_{ep_x})$ and $L_x = \mathsf{strlen}(u_x)$. To find $w$, we perform a linear search on the path from $u_{x+1}$ to root until we find a node satisfying the required condition. This requires $O(h)$ rank queries, where $h = O(1 + L_{x+1} - L_x)$ is the number of nodes on the path from $u_{x+1}$ to $w$. Therefore, the time for a particular value of $x$ is $O(h \log\log m + \log m)$. When considering all values of $x$, we obtain the following total time complexity.[3]

$$\sum_{x=1}^{|T|}((L_{x+1} - L_x)\log\log m + \log m) = (L_{|T|+1} - L_1)\log\log m + |T|\log m = O(|T|\log m).$$

In summary, the time for step 1 for all values of $x$ combined is $O(|T|\log m)$.

### 4.2.2 Details of Step 2

Find $u_x^0 = u_x, u_x^1, u_x^2, \ldots, u_x^f$, where $u_x^h$ is the lowest marked ancestor of $u_x^{h-1}$ for all $h \in [1, f]$ and $f$ is the number of marked ancestors of $u_x$. Let $[sp^h, ep^h]$ be the range of leaves in the subtree of $u_x^h$. We make the following queries and collect their answers.

- $\text{REPORT}([sp^0, ep^0], L_x)$, which returns the $P_i$'s corresponding to $(x, i)$ as the final output, where the leaves corresponding to them are below $u_x^0$.

---

3 An alternative way to find $w$ is to perform a binary search along the path from $u_{x+1}$ to the root. This method requires a logarithmic number of rank queries and takes $O(\log m \log\log m)$ time for a particular value of $x$. However, it results in a higher overall time complexity compared to what we described above.

- For $h \in [1, f]$, we perform the following two queries: REPORT($[sp^h, sp^{h-1}), \mathsf{strlen}(u_x^h)$) and REPORT($(ep^{h-1}, ep^h], \mathsf{strlen}(u_x^h)$), which return the $P_i$'s corresponding to $(x, i)$ as the final output, where the leaves corresponding to them are below $u_x^h$, but not below $u_x^{h-1}$ (this avoids reporting the same answer multiple times). For a fixed $x$, the output size is at least $f$ from the definition of marked nodes.

The total time of Step 2 is $O((|T| + \mathsf{occ}) \log m)$, which thereby establishes the overall time complexity. This completes the proof of Theorem 2.

## 5 Discussion and Open Problems

We present a succinct index for the circular dictionary matching problem that supports queries efficiently in $O((|T| + \mathsf{occ}) \log n)$ time. In our earlier work [48], we achieved a similar result under the assumption that all patterns are approximately the same length (i.e., $\Theta(n/d)$). There, this assumption was necessary for the efficient encoding of $\mathsf{strlen}(\cdot)$ values of nodes in a suffix tree-like structure. Although we later managed to remove this restriction, it came at the cost of a higher query time of $O((|T| \log n + \mathsf{occ}) \log n)$. The improved result in this paper builds on a novel use of the eBWT, combined with a careful adaptation of Sadakane's technique for encoding $\mathsf{strlen}(\cdot)$ values of nodes in the eST, as described in section 3.2.

We conclude with a list of problems that remain open for future research.

1. In contrast to the best known succinct solution for the standard dictionary matching problem [10], which achieves a query time of $O(|T| + \mathsf{occ})$, our solution has a query complexity of $O((|T| + \mathsf{occ}) \log n)$ in the circular setting. Even the recent alternative solution by Cotumaccio [25] has the same query time. This raises an important and natural question: Can the query time be further improved while maintaining the same space bound? Although this does not seem immediate, we remark that a trade-off allowing faster query time might be possible by adapting techniques from [41]; for example, using $O(n \log \sigma)$ bits of space and achieving $O((|T| + \mathsf{occ})(\log_\sigma^\epsilon n + \log \log n))$ query time, where $\epsilon > 0$ is an arbitrarily small constant.
2. Another important question is the efficient construction of our data structure. While the construction of the extended Burrows-Wheeler Transform is already a well-studied problem [8, 13] (also see [43]), the remaining challenge lies in designing the construction of the additional structures. While linear-space construction algorithms that require near-linear time appear achievable, achieving space efficiency (i.e., using small working space) and optimizing the polylogarithmic factors in time can be challenging. Additionally, introducing engineering solutions, possibly through heuristics, could lead to a practical approach, which would benefit from experimental analysis.
3. A *repetition-aware* index for circular dictionary matching would be desirable. While it is relatively straightforward to reduce the space of the extended Burrows-Wheeler Transform by applying run-length compression [14], encoding the remaining components (specifically, the RMQ structures in Section 4.1) in a repetition-aware manner remains challenging. Addressing this may require a careful adaptation of the techniques from [31] or from very recent work in [50].

## References


1 Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

2 Amihood Amir and Martin Farach. Two-Dimensional Dictionary Matching. *Inf. Process. Lett.*, 44(5):233–239, 1992. doi:10.1016/0020-0190(92)90206-B.

3 Amihood Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. Dynamic Dictionary Matching. *J. Comput. Syst. Sci.*, 49(2):208–222, 1994. doi:10.1016/S0022-0000(05)80047-9.

4 Amihood Amir, Dmitry Keselman, Gad M. Landau, Moshe Lewenstein, Noa Lewenstein, and Michael Rodeh. Text Indexing and Dictionary Matching with One Error. *J. Algorithms*, 37(2):309–325, 2000. doi:10.1006/JAGM.2000.1104.

5 Amihood Amir, Tsvi Kopelowitz, Avivit Levy, Seth Pettie, Ely Porat, and B. Riva Shalom. Mind the Gap! – Online Dictionary Matching with One Gap. *Algorithmica*, 81(6):2123–2157, 2019. doi:10.1007/S00453-018-0526-2.

6 Amihood Amir, Avivit Levy, Ely Porat, and B. Riva Shalom. Online Recognition of Dictionary with One Gap. *Inf. Comput.*, 275:104633, 2020. doi:10.1016/J.IC.2020.104633.

7 Tanver Athar, Carl Barton, Widmer Bland, Jia Gao, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Fast Circular Dictionary-Matching Algorithm. *Math. Struct. Comput. Sci.*, 27(2):143–156, 2017. doi:10.1017/S0960129515000134.

8 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piatkowski. Constructing and Indexing the Bijective and Extended Burrows-Wheeler Transform. *Inf. Comput.*, 297:105153, 2024. doi:10.1016/J.IC.2024.105153.

9 Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Fast Algorithms for Approximate Circular String Matching. *Algorithms Mol. Biol.*, 9:9, 2014. doi:10.1186/1748-7188-9-9.

10 Djamal Belazzougui. Succinct Dictionary Matching with No Slowdown. In Amihood Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21-23, 2010. Proceedings*, volume 6129 of *Lecture Notes in Computer Science*, pages 88–100. Springer, 2010. doi:10.1007/978-3-642-13509-5_9.

11 Djamal Belazzougui and Gonzalo Navarro. Optimal Lower and Upper Bounds for Representing Sequences. *ACM Trans. Algorithms*, 11(4):31:1–31:21, 2015. doi:10.1145/2629339.

12 Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *Int. J. Found. Comput. Sci.*, 25(8):1161, 2014. doi:10.1142/S0129054114400309.

13 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. Computing the Original eBWT Faster, Simpler, and with Less Memory. In Thierry Lecroq and Hélène Touzet, editors, *String Processing and Information Retrieval - 28th International Symposium, SPIRE 2021, Lille, France, October 4-6, 2021, Proceedings*, volume 12944 of *Lecture Notes in Computer Science*, pages 129–142. Springer, 2021. doi:10.1007/978-3-030-86692-1_11.

14 Christina Boucher, Davide Cenzato, Zsuzsanna Lipták, Massimiliano Rossi, and Marinella Sciortino. r-Indexing the eBWT. *Inf. Comput.*, 298:105155, 2024. doi:10.1016/J.IC.2024.105155.

15 Michael Burrows and David J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. *SRC Research Report, 124*, 1994.

16 Bastien Cazaux and Eric Rivals. Linking BWT and XBW via Aho-Corasick Automaton: Applications to Run-Length Encoding. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy*, volume 128 of *LIPIcs*, pages 24:1–24:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CPM.2019.24.

17 Davide Cenzato and Zsuzsanna Lipták. A Theoretical and Experimental Analysis of BWT Variants for String Collections. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CPM.2022.25.

18 Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Dynamic Dictionary Matching and Compressed Suffix Trees. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada,*

*January 23-25, 2005*, pages 13–22. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070436.

19 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal Dictionary Matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/S00453-021-00821-Y.

20 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Circular Pattern Matching with $k$ Mismatches. In Leszek Antoni Gasieniec, Jesper Jansson, and Christos Levcopoulos, editors, *Fundamentals of Computation Theory - 22nd International Symposium, FCT 2019, Copenhagen, Denmark, August 12-14, 2019, Proceedings*, volume 11651 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2019. doi:10.1007/978-3-030-25027-0_15.

21 Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Circular Pattern Matching with $k$ Mismatches. *J. Comput. Syst. Sci.*, 115:73–85, 2021. doi:10.1016/J.JCSS.2020.07.003.

22 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P. Pissis, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate Circular Pattern Matching. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPIcs*, pages 35:1–35:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ESA.2022.35.

23 Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen, and Wiktor Zuba. Approximate Circular Pattern Matching Under Edit Distance. In Olaf Beyersdorff, Mamadou Moustapha Kanté, Orna Kupferman, and Daniel Lokshtanov, editors, *41st International Symposium on Theoretical Aspects of Computer Science, STACS 2024, March 12-14, 2024, Clermont-Ferrand, France*, volume 289 of *LIPIcs*, pages 24:1–24:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.STACS.2024.24.

24 Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary Matching and Indexing with Errors and Don't Cares. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004. doi:10.1145/1007352.1007374.

25 Nicola Cotumaccio. Improved circular dictionary matching. In Paola Bonizzoni and Veli Mäkinen, editors, *36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025, June 17-19, 2025, Milan, Italy*, volume 331 of *LIPIcs*, pages 18:1–18:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.CPM.2025.18.

26 Guy Feigenblat, Ely Porat, and Ariel Shiftan. An Improved Query Time for Succinct Dynamic Dictionary Matching. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, volume 8486 of *Lecture Notes in Computer Science*, pages 120–129. Springer, 2014. doi:10.1007/978-3-319-07566-2_13.

27 Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Structuring Labeled Trees for Optimal Succinctness, and Beyond. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 184–196. IEEE Computer Society, 2005. doi:10.1109/SFCS.2005.69.

28 Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *J. ACM*, 52(4):552–581, 2005. doi:10.1145/1082036.1082039.

29 Nathan J Fine and Herbert S Wilf. Uniqueness Theorems for Periodic Functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

30 Johannes Fischer and Volker Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.*, 40(2):465–492, 2011. doi:10.1137/090779759.

31 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *J. ACM*, 67(1):2:1–2:54, 2020. doi:10.1145/3375890.

32 Arnab Ganguly, Wing-Kai Hon, Kunihiko Sadakane, Rahul Shah, Sharma V. Thankachan, and Yilin Yang. A Framework for Designing Space-Efficient Dictionaries for Parameterized and Order-Preserving Matching. *Theor. Comput. Sci.*, 854:52–62, 2021. doi:10.1016/J.TCS.2020.11.036.

33 Arnab Ganguly, Rahul Shah, and Sharma V. Thankachan. pBWT: Achieving Succinct Data Structures for Parameterized Pattern Matching and Related Problems. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 397–407. SIAM, 2017. doi:10.1137/1.9781611974782.25.

34 Pawel Gawrychowski and Tatiana Starikovskaya. Streaming Dictionary Matching with Mismatches. *Algorithmica*, 84(4):896–916, 2022. doi:10.1007/S00453-021-00876-X.

35 Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Dynamic Dictionary Matching in the Online Model. In Zachary Friggstad, Jörg-Rüdiger Sack, and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *Lecture Notes in Computer Science*, pages 409–422. Springer, 2019. doi:10.1007/978-3-030-24766-9_30.

36 Shay Golan, Tsvi Kopelowitz, and Ely Porat. Towards Optimal Approximate Streaming Pattern Matching by Matching Multiple Patterns in Multiple Streams. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 65:1–65:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.ICALP.2018.65.

37 Shay Golan and Ely Porat. Real-Time Streaming Multi-Pattern Search for Constant Alphabet. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 41:1–41:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.ESA.2017.41.

38 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/Select Operations on Large Alphabets: A Tool for Text Indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373. ACM Press, 2006. URL: http://dl.acm.org/citation.cfm?id=1109557.1109599.

39 Roberto Grossi, Ankur Gupta, and Jeffrey Scott 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. ACM/SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644250.

40 Roberto Grossi, Costas S. Iliopoulos, Robert Mercas, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, and Fatima Vayani. Circular Sequence Comparison: Algorithms and Applications. *Algorithms Mol. Biol.*, 11:12, 2016. doi:10.1186/S13015-016-0076-6.

41 Roberto Grossi and Jeffrey Scott Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.*, 35(2):378–407, 2005. doi:10.1137/S0097539702402354.

42 Wing-Kai Hon, Tsung-Han Ku, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressing Dictionary Matching Index via Sparsification Technique. *Algorithmica*, 72(2):515–538, 2015. doi:10.1007/S00453-013-9863-3.

43 Wing-Kai Hon, Tsung-Han Ku, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Efficient Algorithm for Circular Burrows-Wheeler Transform. In Juha Kärkkäinen and Jens Stoye, editors, *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings*, volume 7354 of *Lecture Notes in Computer Science*, pages 257–268. Springer, 2012. doi:10.1007/978-3-642-31265-6_21.

**44** Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressed Dictionary Matching with One Error. In James A. Storer and Michael W. Marcellin, editors, *2011 Data Compression Conference (DCC 2011), 29-31 March 2011, Snowbird, UT, USA*, pages 113–122. IEEE Computer Society, 2011. doi:10.1109/DCC.2011.18.

**45** Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster Compressed Dictionary Matching. *Theor. Comput. Sci.*, 475:113–119, 2013. doi:10.1016/J.TCS.2012.10.050.

**46** Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Siu-Lung Tam, and Jeffrey Scott Vitter. Succinct Index for Dynamic Dictionary Matching. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 1034–1043. Springer, 2009. doi:10.1007/978-3-642-10631-6_104.

**47** Wing-Kai Hon, Tak Wah Lam, Rahul Shah, Sharma V. Thankachan, Hing-Fung Ting, and Yilin Yang. Dictionary Matching with a Bounded Gap in Pattern or in Text. *Algorithmica*, 80(2):698–713, 2018. doi:10.1007/S00453-017-0288-2.

**48** Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V. Thankachan. Succinct Indexes for Circular Patterns. In Takao Asano, Shin-Ichi Nakano, Yoshio Okamoto, and Osamu Watanabe, editors, *Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings*, volume 7074 of *Lecture Notes in Computer Science*, pages 673–682. Springer, 2011. doi:10.1007/978-3-642-25591-5_69.

**49** Costas S. Iliopoulos, Solon P. Pissis, and M. Sohel Rahman. Searching and Indexing Circular Patterns. In Mourad Elloumi, editor, *Algorithms for Next-Generation Sequencing Data, Techniques, Approaches, and Applications*, pages 77–90. Springer, 2017. doi:10.1007/978-3-319-59826-0_3.

**50** Dominik Kempa and Tomasz Kociumaka. Collapsing the Hierarchy of Compressed Data Structures: Suffix Arrays in Optimal Compressed Space. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1877–1886. IEEE, 2023. doi:10.1109/FOCS57990.2023.00114.

**51** Tsvi Kopelowitz, Ely Porat, and Yaron Rozen. Succinct Online Dictionary Matching with Improved Worst-Case Guarantees. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 6:1–6:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.CPM.2016.6.

**52** Avivit Levy and B. Riva Shalom. Online Parameterized Dictionary Matching with One Gap. *Theor. Comput. Sci.*, 845:208–229, 2020. doi:10.1016/J.TCS.2020.09.016.

**53** Udi Manber and Eugene W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

**54** Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An Extension of the Burrows-Wheeler Transform. *Theor. Comput. Sci.*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.

**55** Giovanni Manzini. XBWT Tricks. In Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai, editors, *String Processing and Information Retrieval - 23rd International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, volume 9954 of *Lecture Notes in Computer Science*, pages 80–92, 2016. doi:10.1007/978-3-319-46049-9_8.

**56** J. Ian Munro and Venkatesh Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM J. Comput.*, 31(3):762–776, 2001. doi:10.1137/S0097539799364092.

**57** Gonzalo Navarro and Veli Mäkinen. Compressed Full-Text Indexes. *ACM Comput. Surv.*, 39(1):2, 2007. doi:10.1145/1216370.1216372.

**58** Shoshana Neuburger and Dina Sokol. Succinct 2D Dictionary Matching. *Algorithmica*, 65(3):662–684, 2013. doi:10.1007/S00453-012-9615-9.

**59** Enno Ohlebusch, Stefan Stauß, and Uwe Baier. Trickier XBWT Tricks. In Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, editors, *String Processing and*

*Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings*, volume 11147 of *Lecture Notes in Computer Science*, pages 325–333. Springer, 2018. doi:10.1007/978-3-030-00479-8_26.

60 Eric M. Osterkamp and Dominik Köppl. Extending the Parameterized Burrows-Wheeler Transform. In Ali Bilgin, James E. Fowler, Joan Serra-Sagristà, Yan Ye, and James A. Storer, editors, *Data Compression Conference, DCC 2024, Snowbird, UT, USA, March 19-22, 2024*, pages 143–152. IEEE, 2024. doi:10.1109/DCC58796.2024.00022.

61 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct Indexable Dictionaries with Applications to Encoding $k$-ary Trees, Prefix Sums and Multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

62 Kunihiko Sadakane. Compressed Suffix Trees with Full Functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

63 Kunihiko Sadakane and Gonzalo Navarro. Fully-Functional Succinct Trees. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 134–149. SIAM, 2010. doi:10.1137/1.9781611973075.13.

64 Blair L Strang and Nigel D Stow. Circularization of the Herpes Simplex Virus Type 1 Genome upon Lytic Infection. *Journal of Virology*, 79(19):12487–12494, 2005.

65 Kazuya Tsuruta, Dominik Köppl, Shunsuke Kanda, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. c-trie++: A Dynamic Trie tailored for Fast Prefix Searches. *Inf. Comput.*, 285(Part):104794, 2022. doi:10.1016/J.IC.2021.104794.

66 Esko Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

67 Peter 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. doi:10.1109/SWAT.1973.13.