Improved Circular Dictionary Matching

Nicola Cotumaccio
University of Helsinki, Finland

Abstract

The circular dictionary matching problem is an extension of the classical dictionary matching problem where every string in the dictionary is interpreted as a circular string: after reading the last character of a string, we can move back to its first character. The circular dictionary matching problem is motivated by applications in bioinformatics and computational geometry.

In 2011, Hon et al. [ISAAC 2011] showed how to efficiently solve circular dictionary matching queries within compressed space by building on Mantaci et al.'s eBWT and Sadakane's compressed suffix tree. The proposed solution is based on the assumption that the strings in the dictionary are all distinct and non-periodic, no string is a circular rotation of some other string, and the strings in the dictionary have similar lengths.

In this paper, we consider *arbitrary* dictionaries, and we show how to solve circular dictionary matching queries in $O((m+occ)\log n)$ time within compressed space using $n\log\sigma(1+o(1))+O(n)+O(d\log n)$ bits, where n is the total length of the dictionary, m is the length of the pattern, occ is the number of occurrences, d is the number of strings in the dictionary and σ is the size of the alphabet. Our solution is based on an extension of the suffix array to arbitrary dictionaries and a sampling mechanism for the LCP array of a dictionary inspired by recent results in graph indexing and compression.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases Circular pattern matching, dictionary matching, suffix tree, compressed suffix tree, suffix array, LCP array, Burrows-Wheeler Transform, FM-index

Digital Object Identifier 10.4230/LIPIcs.CPM.2025.18

Related Version *Full Version*: `https://arxiv.org/abs/2504.03394` [21]

Funding Funded by the Helsinki Institute for Information Technology (HIIT).

Acknowledgements I would like to thank Kunihiko Sadakane for introducing me to the topic.

1 Introduction

The Burrows-Wheeler transform (BWT) [12] and the FM-index [27] support pattern matching on the compressed representation of a *single* string. If we want to encode a *collection* $\mathcal{T}$ of strings, we may append a distinct end-of-string $\$_i$ to each string and store the Burrows-Wheeler Transform of the concatenation of the strings. This approach is only a naive extension of the BWT and can significantly increase the size of the alphabet. In 2007, Mantaci et al. introduced the eBWT [40], a more sophisticated and elegant extension of the BWT to a collection of strings. When sorting all suffixes of all strings in the collection, Mantaci et al. define the mutual order between two suffixes T_i and T_j to be the mutual order of T_i^ω and T_j^ω in the lexicographic order, where T_i^ω and T_j^ω are the infinite strings obtained by concatenating T_i with itself and T_j with itself infinitely many times (see [13] for a recent paper on different variants of the eBWT). From the definition of the eBWT we obtain that, if we extend the backward search mechanism of the FM-index to multisets of strings, we are not matching a pattern P against all suffixes T_i's, but against all strings T_i^ω's. Equivalently, we are interpreting each string in the collection as a *circular* string in which, after reaching the last character of the string, we can go back to its first character.

In 2011, Hon et al. applied the framework of the eBWT to solve *circular dictionary matching* queries [36], even if they explicitly spotted the relationship between their techniques (which extends Sadakane's compressed suffix tree [45] to a collection of strings) and the eBWT only in a subsequent paper [32]. The circular dictionary matching problem is an extension of the *dictionary matching problem*, which admits a classical solution based on Aho-Corasick automata [1], as well as more recent solutions within compressed space [8, 34]. Other variations of the same problem include dictionary matching with gaps [6, 35, 5] or mismatches [30], dictionary matching in the streaming model [16], dynamic dictionary matching [31] and internal dictionary matching [14]. The circular dictionary matching problem is motivated by some applications, see [38]. In bioinformatics, the genome of herpes simplex virus (HSV-1) and many other viruses exists as circular strings [47], and microbial samples collected from the environment directly without isolating and culturing the samples have circular chromosomes [26, 46]. In computational geometry, a polygon can be stored by listing its vertices in clockwise order. The circular dictionary matching problem has also been studied from other perspectives (average-case behavior [37], approximate variants [15]).

1.1 Our Contribution

Consider a dictionary $\mathcal{T} = (T_1, T_2, \ldots, T_d)$ of total length n on an alphabet of size σ. In [36], Hon et al. show that, by storing a data structure of $n \log \sigma(1 + o(1)) + O(n) + O(d \log n)$ bits, it is possible to solve circular dictionary matching queries in $O((m + occ) \log^{1+\epsilon} n)$ time, where m is the length of the pattern. This result holds under three assumptions: (i) the T_h's are distinct and not periodic, (ii) no T_h is a circular rotation of some other $T_{h'}$, and (iii) the T_h's have bounded aspect ratio, namely, $(\max_{h=1}^{d} T_h)/(\min_{h=1}^{d} T_h) = O(1)$. Assumption (ii) was made explicit only in the subsequent paper [32], where Hon et al. also mention that, in an extension of the paper, they will show how to remove assumptions (i-ii). Assumption (iii) is required to store a space-efficient data structure supporting longest common prefix (LCP) queries. In [33], Hon et al. sketched a new data structure for LCP queries that uses $O(n + d \log n)$ bits without requiring assumption (iii), but all details are again deferred to an extended version.

The main result of our paper is Theorem 1. In particular, we give two main contributions.

- We obtain results that are valid for an arbitrary dictionary $\mathcal{T}$, without imposing any restriction. To this end, in Section 3.2 we introduce a more general compressed suffix array for an arbitrary collection of strings. In particular, we support LCP functionality within only $O(n) + o(n \log \sigma)$ bits by using a sampling mechanism inspired by recent results [23, 17] on Wheeler automata.
- We provide a self-contained proof of our result and a full example of the data structure for solving circular dictionary matching queries. The original paper [36] contains the main ideas to extend Sadakane's compressed suffix tree to a multiset of strings, but it is very dense and proofs are often only sketched or missing, which may explain why additional properties to potentially remove assumptions (i - iii) were only observed in subsequent papers. We will provide an intuitive explanation of several steps by using graphs (and in particular cycles, see Figure 2), consistently with a research line that has shown how the suffix array, the Burrows-Wheeler transform and the FM-index admit a natural interpretation in the graph setting [29, 3, 24, 22, 18, 4, 2, 25, 20].

We will also show that the time bound of Theorem 1 can be improved when the number of occurrences is large.

$\mathcal{T}$	a	b	c	a	b	c	b	c	a	b	c	c	a	b
k	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$B_1[k]$	1	0	0	0	0	0	1	0	0	0	0	1	0	0

(a)

$\mathcal{S}_j$	D_j	j	LCP$[j]$	BWT$[j]$	BWT$^*[j]$	$B_2[j]$
$a\ b\ c\ a\ b\ c\ a\ b\ c\ \dots$	1					
$a\ b\ c\ a\ b\ c\ a\ b\ c\ \dots$	4	1		c	a	1
$a\ b\ c\ a\ b\ c\ a\ b\ c\ \dots$	13					
$a\ b\ c\ b\ c\ a\ b\ c\ b\ \dots$	9	2	3	c	a	0
$b\ c\ a\ b\ c\ a\ b\ c\ a\ \dots$	2					
$b\ c\ a\ b\ c\ a\ b\ c\ a\ \dots$	5	3	0	a	b	1
$b\ c\ a\ b\ c\ a\ b\ c\ a\ \dots$	14					
$b\ c\ a\ b\ c\ b\ c\ a\ b\ \dots$	7	4	5	c	b	0
$b\ c\ b\ c\ a\ b\ c\ b\ c\ \dots$	10	5	2	a	b	0
$c\ a\ b\ c\ a\ b\ c\ a\ b\ \dots$	3					
$c\ a\ b\ c\ a\ b\ c\ a\ b\ \dots$	6	6	0	b	c	1
$c\ a\ b\ c\ a\ b\ c\ a\ b\ \dots$	12					
$c\ a\ b\ c\ b\ c\ a\ b\ c\ \dots$	8	7	4	b	c	0
$c\ b\ c\ a\ b\ c\ b\ c\ a\ \dots$	11	8	1	b	c	0

(b)

Figure 1 Consider the dictionary $\mathcal{T} = (abcabc, bcabc, cab)$, our running example. In (b), the strings $\mathcal{S}_j$'s are sorted lexicographically, and every block identifies strings T_k's that correspond to the same $\mathcal{S}_j$. Each string T_k is the orange prefix of $\mathcal{S}_j$. For example, we have $4 \in D_1$, $\mathcal{T}_4 = abcabc$ and $\mathcal{S}_1 = (abc)^\omega$.

The paper is organized as follows. In Section 2 we introduce the circular dictionary matching problem. In Section 3 we define our compressed suffix tree. In Section 4 we present an algorithm for circular dictionary matching. Due to space constraints, most proofs and some auxiliary results can be found in the full version [21].

2 Preliminaries

Let Σ be a finite alphabet of size $\sigma = |\Sigma|$. We denote by Σ^* the set of all finite strings on Σ (including the empty string ϵ) and by Σ^ω the set of all countably infinite strings on Σ. For example, if $\Sigma = \{a, b\}$, then $abbba$ is a string in Σ^* and $ababab\dots$ is a string in Σ^ω. If $T \in \Sigma^*$, we denote by $|T|$ the length of T. If $1 \le i \le |T|$, we denote by $T[i]$ the i-th character of T, and if $1 \le i \le j \le |T|$, we define $T[i, j] = T[i]T[i+1]\dots T[j-1]T[j]$. If $i > j$, let $T[i, j] = \epsilon$. Analogously, if $T \in \Sigma^\omega$, then $T[i]$ is the i-th character of T, we have $T[i, j] = T[i]T[i+1]\dots T[j-1]T[j]$ for every $j \ge i \ge 1$, and if $i > j$ we have $T[i, j] = \epsilon$.

If $T \in \Sigma^*$, then for every integer $z \ge 1$ the string $T^z \in \Sigma^*$ is defined by $T^z = TT\dots T$, where T is repeated z times, and the string $T^\omega \in \Sigma^\omega$ is defined by $T^\omega = TTT\dots$, where T is repeated infinitely many times. Given $T_1, T_2 \in \Sigma^* \cup \Sigma^\omega$, let $\mathsf{lcp}(T_1, T_2)$ be the length of the longest common prefix between T_1 and T_2.

We say that a string $T \in \Sigma^*$ is *primitive* is for every $S \in \Sigma^*$ and for every integer $z \ge 1$, if $T = S^z$, then $z = 1$ and $T = S$. For every $T \in \Sigma^*$, there exists exactly one primitive string $R \in \Sigma^*$ and exactly one integer $z \ge 1$ such that $T = R^z$ (see [39, Prop. 1.3.1]; we say that R is the *root* of T and we write $R = \rho(T)$.

Let $T \in \Sigma^*$ a string of length n. We define the following queries on T: (i) $\mathsf{access}(T, i)$: given $1 \leq i \leq n$, return $T[i]$; (ii) $\mathsf{rank}_c(T, i)$: given $c \in \Sigma$ and $1 \leq i \leq n$, return $|\{1 \leq h \leq i \mid T[h] = c\}|$; (iii) $\mathsf{select}_c(T, i)$: given $c \in \Sigma$ and $1 \leq i \leq \mathsf{rank}_c(T, n)$, return the unique $1 \leq j \leq n$ such that $T[j] = c$ and $|\{1 \leq h \leq j \mid T[h] = c\}| = i$. To handle limit cases easily, it is expedient to define $\mathsf{select}_c(T, \mathsf{rank}_c(T, n) + 1) = n + 1$. A bit array of length n can be stored using a data structure (called *bitvector*) of $n + o(n)$ bits that supports access, rank and select queries in $O(1)$ time [42].

We consider a fixed total order $\preceq$ on Σ and we extend it to $\Sigma^* \cup \Sigma^\omega$ *lexicographically*. For every $T_1, T_2 \in \Sigma^* \cup \Sigma^\omega$, we write $T_1 \prec T_2$ if $(T_1 \preceq T_2) \wedge (T_1 \neq T_2)$. In our examples (see e.g. Figure 1), Σ is a subset of the English alphabet and $\preceq$ is the usual order on letters. In our data structures, $\Sigma = \{0, 1, \ldots, |\Sigma| - 1\}$ and $\preceq$ is the usual order such that $0 \prec 1 \prec \cdots \prec |\Sigma| - 1$.

2.1 Circular Dictionary Matching

Let $T, P \in \Sigma^*$ and P. For every $1 \leq i \leq |P| - |T| + 1$, we say that T *occurs* in P at position i if $P[i, i + |T| - 1] = T$.

Let $T \in \Sigma^*$ and let $1 \leq i \leq |T|$. The *circular suffix* T_i of T is the string $T[i, |T|]T[1, i - 1]$. For example, if $T = cab$, then $T_1 = cab$, $T_2 = abc$ and $T_3 = bca$.

Let $d \geq 1$, and let $T_1, T_2, \ldots, T_d \in \Sigma^*$ be nonempty strings. We say that $\mathcal{T} = (T_1, T_2, \ldots, T_d)$ is a *dictionary*. We assume that the alphabet Σ is *effective*, that is, every character in Σ occurs in some T_j (our results could be extended to larger alphabets by using standard tools such as dictionaries [42, 22]). Define the *total length* n of $\mathcal{T}$ to be $n = \sum_{k=1}^{d} |T_k|$. In the example of Figure 1a, we have $d = 3$ and $n = 14$. For every $1 \leq k \leq n$, the circular suffix $\mathcal{T}_k$ of $\mathcal{T}$ is the string $T_f(g)$, where f is the largest integer in $\{1, 2, \ldots, d\}$ such that $\sum_{h=1}^{f-1} |T_h| < k$ and $1 \leq g \leq |T_f|$ is such that $k = (\sum_{h=1}^{f-1} |T_h|) + g$. In other words, the circular suffix $\mathcal{T}_k$ is the circular suffix starting at position k in the concatenation $T_1 T_2 \ldots T_d$. In Figure 1, if $k = 13$, we have $f = 3$, $g = 2$ and $\mathcal{T}_{13} = abc$. Given a pattern $P \in \Sigma^*$, where $|P| = m$, define:

$$\mathsf{Cdm}(\mathcal{T}, P) = \{(i, k) \in \{1, 2, \ldots, m\} \times \{1, 2, \ldots, n\} \mid \mathcal{T}_k \text{ occurs in } P \text{ at position } i\}$$

and let $occ = |\mathsf{Cdm}(\mathcal{T}, P)|$. For example, if $P = abcbca$, in the example of Figure 1 we have $\mathsf{Cdm}(\mathcal{T}, P) = \{(1, 9), (1, 13), (2, 10), (4, 14)\}$ and $occ = 4$.

The main result of this paper is the following.

▶ **Theorem 1.** *Let $\mathcal{T} = (T_1, T_2, \ldots, T_d)$ be a dictionary of total length n. Then, $\mathcal{T}$ can be encoded using a data structure of $n \log \sigma(1 + o(1)) + O(n) + O(d \log n)$ bits such that, given a string P of length m, we can compute $\mathsf{Cdm}(\mathcal{T}, P)$ in $O((m + occ) \log n)$ time.*

We can also improve the time bound in Theorem 1 to $O(m \log n + \min\{occ \log n, n \log n + occ\})$, without needing to know the value occ in advance. This is useful if occ is large.

Following Hon et al. [36], we will build a data structure that extends Sadakane's suffix tree to $\mathcal{T}$. We will achieve the space bound in Theorem 1 as follows. (i) We will store the FM-index of $\mathcal{T}$ using $n \log \sigma(1 + o(1)) + O(n)$ bits, see Theorem 9. (ii) We will introduce a notion of suffix array (Section 3.2) and a notion of longest comment prefix (LCP) array (Section 3.3). Storing the suffix array and LCP array explicitly would require $O(n \log n)$ bits so, in both cases, we will only store a sample (Theorem 10 and Theorem 13), leading to a compressed representation of both arrays. (iii) We will store some auxiliary data structures ($O(n)$ bits in total): 8 bitvectors (called B_1-B_8), a data structure supporting range minimum

queries on the LCP array (see Section 3.3), a data structure storing the topology of the suffix tree (see Section 3.4), a data structure storing the topology of the auxiliary tree $\mathsf{Suff}^*(\mathcal{T})$ (see Section 4), and a data structure supporting range minimum queries on the auxiliary array Len (see Section 4).

3 The Compressed Suffix Tree of $\mathcal{T}$

In this section, we extend Sadakane's compressed suffix tree to the dictionary $\mathcal{T}$. To this end, we will introduce a BWT-based encoding of $\mathcal{T}$ (Section 3.1), a compressed suffix array (Section 3.2), an LCP array (Section 3.3), and the topology of the suffix tree (Section 3.4).

3.1 The Burrows-Wheeler Transform and the FM-index of $\mathcal{T}$

Let us consider our dictionary $\mathcal{T}$ (recall that $n = \sum_{k=1}^{d} |T_k|$). We can naturally define a bijective function ϕ from the set $\{1, 2, \dots, n\}$ to the set $\{(f, g) \mid 1 \leq f \leq d, 1 \leq g \leq |T_f|\}$ such that, if $\phi(k) = (f, g)$, then the k-th character of the concatenation $T_1 T_2 \dots T_d$ is the g-th character of T_f. For example, in Figure 1 we have $\phi(13) = (3, 2)$. Let us define a bitvector B_1 to compute ϕ and ϕ^{-1} in $O(1)$ time. B_1 is the bitvector of length n that contains exactly d ones such that, for every $1 \leq h \leq d$, the number of consecutive zeros after the h-th one is $|T_h| - 1$ (see Figure 1a). Assume that $\phi(k) = (f, g)$. Given k, we have $f = \mathsf{rank}_1(B_1, k)$ and $g = k - \mathsf{select}_1(B_1, f) + 1$. Conversely, given f and g, we have $k = \mathsf{select}_1(B_1, f) + g - 1$. Note that in $O(1)$ time we can also compute $|T_h|$ for every $1 \leq h \leq d$, because $|T_h| = \mathsf{select}_1(B_1, h + 1) - \mathsf{select}_1(B_1, h)$.

To exploit the circular nature of the dictionary, we will not sort the finite strings $\mathcal{T}_k$'s, but we will sort the infinite strings $\mathcal{T}_k^\omega$'s, and the suffix tree of $\mathcal{T}$ will be the trie of the $\mathcal{T}_k$'s. Notice that we may have $\mathcal{T}_k = \mathcal{T}_{k'}$ for distinct $1 \leq k, k' \leq n$ (in Figure 1, we have $\mathcal{T}_1 = \mathcal{T}_4 = \mathcal{T}_{13} = (abc)^\omega$). The next lemma shows that this happens exactly when $\mathcal{T}_k$ and $\mathcal{T}_{k'}$ have the same root.

▶ **Lemma 2.** *Let $\mathcal{T}$ be a dictionary, and let $1 \leq k, k' \leq n$. Then, $\mathcal{T}_k^\omega = \mathcal{T}_{k'}^\omega$ if and only if $\rho(\mathcal{T}_k) = \rho(\mathcal{T}_{k'})$.*

The next step is to sort the "suffixes" $\mathcal{T}_k$'s. Since in general the $\mathcal{T}_k$'s are not pairwise distinct, we will use an ordered partition (see Figure 1b).

▶ **Definition 3.** *Let $\mathcal{T}$ be a dictionary.*

- *Let $\mathcal{D} = (D_1, D_2, \dots, D_{n'})$ be the ordered partition of $\{1, 2, \dots, n\}$ such that, for every $k, k' \in \{1, 2, \dots, n\}$, (i) if k and k' are in the same D_j, then $\mathcal{T}_k^\omega = \mathcal{T}_k^\omega$ and (ii) if $k \in D_j$, $k' \in D_{j'}$ and $j < j'$, then $\mathcal{T}_k^\omega \prec \mathcal{T}_{k'}^\omega$.*
- *For every $1 \leq j \leq n'$, let $\mathcal{S}_j = \mathcal{T}_k^\omega$, where $1 \leq k \leq n$ is such that $k \in D_j$.*

Note that $\mathcal{S}_j$ is well defined because it does not depend on the choice of $k \in D_j$ by the definition of $\mathcal{D}$. Note also that $\mathcal{S}_1 \prec \mathcal{S}_2 \prec \cdots \prec \mathcal{S}_{n'}$.

Let $1 \leq k \leq n$, and assume that $\phi(k) = (f, g)$. Define $\mathsf{pred}(k)$ as follows: (i) if $g \geq 2$, then $\mathsf{pred}(k) = k - 1$, and (ii) if $g = 1$, then $\mathsf{pred}(k) = \phi^{-1}(f, |T_f|)$. In other words, $\mathsf{pred}(k)$ equals $k - 1$, modulo staying within the same string of $\mathcal{T}$. In Figure 1a, we have $\mathsf{pred}(9) = 8$, $\mathsf{pred}(8) = 7$ but $\mathsf{pred}(7) = 11$. For every $1 \leq k \leq n$, we can compute $\mathsf{pred}(k)$ in $O(1)$ time by using the bitvector B_1: if $B_1[k] = 0$, then $\mathsf{pred}(k) = k - 1$, and if $B_1[k] = 1$, then $\mathsf{pred}(k) = \mathsf{select}_1(\mathsf{rank}_1(B_1, k) + 1) - 1$.

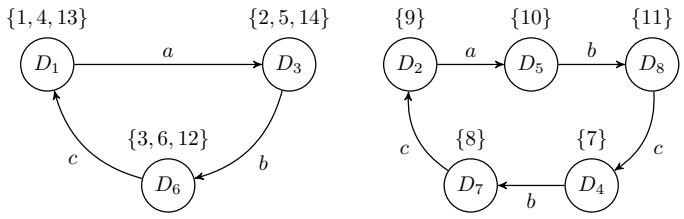


Figure 2 The graph $G_{\mathcal{T}}$ for the dictionary $\mathcal{T} = (abcabc, bcabc, cab)$ of Figure 1.

If $U \subseteq \{1, 2, \ldots, n\}$, define $\mathsf{pred}(U) = \bigcup_{k \in U} \mathsf{pred}(k)$. Since the function pred is a permutation of the set $\{1, 2, \ldots, n\}$, we always have $|\mathsf{pred}(U)| = |U|$ for every $U \subseteq \{1, 2, \ldots, n\}$. Let us prove that pred yields a permutation of the D_j's.

▶ **Lemma 4.** *Let $\mathcal{T}$ be a dictionary, and let $1 \leq j \leq n'$. Then, there exists $1 \leq j' \leq n$ such that $\mathsf{pred}(D_j) = D_{j'}$. Moreover, if $c = \mathcal{S}_{j'}[1]$, we have $\mathcal{S}_{j'} = c\mathcal{S}_j$.*

For every $1 \leq j \leq n'$, let $\psi(j) = j'$, where $\mathsf{pred}(D_j) = D_{j'}$ as in Lemma 4. Notice that ψ is permutation of $\{1, 2, \ldots, n'\}$ because it is subjective: indeed, for every $1 \leq j' \leq n'$, if we pick any $k' \in D_{j'}$, and consider $1 \leq k \leq n'$ such that $k' = \mathsf{pred}(k)$ and $1 \leq j \leq n'$ such that $k \in D_j$, then $\mathsf{pred}(D_j) = D_{j'}$. Moreover, for every $1 \leq j \leq n'$ define $\mu(j) = \mathcal{S}_{\psi(j)}[1]$. By Lemma 4, we know that $\mathcal{S}_{\psi(j)} = \mu(j)\mathcal{S}_j$ for every $1 \leq j \leq n'$.

We can visualize ψ by drawing a (directed edge-labeled) graph $G_{\mathcal{T}} = (V, E)$, where $V = \{D_j \mid 1 \leq j \leq n'\}$ and $E = \{(D_{\psi(j)}, D_j, \mu(j) \mid 1 \leq j \leq n'\}$, see Figure 2. Since ψ is a permutation of $\{1, 2, \ldots, n'\}$, then every node of G has exactly one incoming edge and exactly one ongoing edge, so G is the disjoint union of some cycles. Moreover, for every $1 \leq j \leq n'$ the infinite string that we can read starting from node D_j and following the edges of the corresponding cycle is $\mathcal{S}_j$, because we know that $\mathcal{S}_{\psi(j)} = \mu(j)\mathcal{S}_j$ for every $1 \leq j' \leq n'$. For example, in Figure 2 the infinite string that we can read starting from D_3 is $\mathcal{S}_3 = (bca)^\omega$.

We will not explicitly build the graph $G_{\mathcal{T}}$ to solve circular dictionary matching queries, but $G_{\mathcal{T}}$ naturally captures the cyclic nature of $\mathcal{T}$ and will help us detect some properties of the D_j's. For example, since we know that the infinite string starting from node D_j is $\mathcal{S}_j$, and $\mathcal{S}_1 \prec \mathcal{S}_2 \prec \cdots \prec \mathcal{S}_{n'}$, we can easily infer the following properties (see Figure 2): if we consider two edges $(D_{j'_1}, D_{j_1}, c)$ and $(D_{j'_2}, D_{j_2}, d)$, then (i) if $j'_1 < j'_2$, then $c \preceq d$, and (ii) if $c = d$ and $j'_1 < j'_2$, then $j_1 < j_2$. We can formalize these properties in our graph-free setting as follows.

▶ **Lemma 5.** *Let $\mathcal{T}$ be a dictionary. let $1 \leq j_1, j'_1, j_2, j'_2 \leq n'$ and $c, d \in \Sigma$ such that $\mathcal{S}_{j'_1} = c\mathcal{S}_{j_1}$ and $\mathcal{S}_{j'_2} = d\mathcal{S}_{j_2}$.*

1. *(Property 1) If $j'_1 < j'_2$, then $c \preceq d$.*
2. *(Property 2) If $c = d$ and $j'_1 < j'_2$, then $j_1 < j_2$.*

We now want to extend the *backward search* [27] to our dictionary $\mathcal{T}$. We will again use $G_{\mathcal{T}}$ to guide our intuition. Given $U \subseteq \{1, 2, \ldots, n'\}$ and given $c \in \Sigma$, let $\mathsf{back}(U, c)$ be the set of all $1 \leq j' \leq n'$ such that there exists an edge $(D_{j'}, D_j, c)$ for some $j \in U$. For example, in Figure 2 we have $\mathsf{back}(\{6, 7, 8\}, b) = \{3, 4, 5\}$. Notice that $\{6, 7, 8\}$ is convex, and $\{3, 4, 5\}$ is also convex. This is true in general: $\mathsf{back}(U, c)$ is always convex if U is convex. Indeed, if $j'_1 < j' < j'_2$ and $j'_1, j'_2 \in \mathsf{back}(U, c)$, then from the properties of $G_{\mathcal{T}}$ mentioned before Lemma 5 we first conclude that the edge leaving node $D_{j'}$ is labeled with c, and then we infer that the node D_j reached by this edge must be such that $j \in U$, which implies $j' \in \mathsf{back}(U, c)$. We can now formally define $\mathsf{back}(U, c)$ in our graph-free setting and prove that $\mathsf{back}(U, c)$ is convex if U is convex.

▶ **Definition 6.** *Let $\mathcal{T}$ be a dictionary, let $U \subseteq \{1, 2, \ldots, n'\}$ and let $c \in \Sigma$. Define* $\mathsf{back}(U, c) = \{1 \leq j' \leq n \mid$ *there exists $j \in U$ such that $\mathcal{S}_{j'} = c\mathcal{S}_j$*$\}$.

Note that, if $U_1 \subseteq U_2 \subseteq \{1, 2, \ldots, n'\}$ and $c \in \Sigma$, then $\mathsf{back}(U_1, c) \subseteq \mathsf{back}(U_2, c)$.

▶ **Lemma 7.** *Let $\mathcal{T}$ be a dictionary, let $U \subseteq \{1, 2, \ldots, n'\}$ and let $c \in \Sigma$. If U is convex, then* $\mathsf{back}(U, c)$ *is convex.*

Let us define the Burrows-Wheeler transform (BWT) of $\mathcal{T}$.

▶ **Definition 8.** *Let $\mathcal{T}$ be a dictionary. Define the string* $\mathsf{BWT} \in \Sigma^*$ *of length n' such that* $\mathsf{BWT}[j] = \mu(j)$ *for every $1 \leq j \leq n'$.*

In other words, $\mathsf{BWT}[j]$ is the label of the edge *reaching* node D_j for every $1 \leq j \leq n'$ (see Figure 1b and Figure 2). The data structure that we will define in Theorem 9 is based on two sequences, BWT^* and B_2, that are related to BWT (see Figure 1b). The sequence BWT^* is obtained from BWT by sorting its elements. We know that for every pair of edges $(D_{j_1'}, D_{j_1}, c)$ and $(D_{j_2'}, D_{j_2}, d)$, if $j_1' < j_2'$, then $c \preceq d$, so $\mathsf{BWT}^*[j]$ is the label of the edge *leaving* node D_j for every $1 \leq j \leq n'$, which implies $\mathsf{BWT}^*[j] = \mathcal{S}_j[1]$ for every $1 \leq j \leq n'$. The bitvector B_2 has length n' and is obtained by marking with 1 the beginning of each equal-letter run in BWT^*. Formally, for every $1 \leq j \leq n'$, we have $B_2[j] = 1$ if and only if $j = 1$ or $(j > 2) \wedge (\mathsf{BWT}^*[j-1] \neq \mathsf{BWT}^*[j])$. Since the alphabet is effective, the set $\{\mathsf{select}_1(B_2, c), \mathsf{select}_1(B_2, c) + 1, \mathsf{select}_1(B_2, c) + 2, \ldots, \mathsf{select}_1(B_2, c+1) - 2, \mathsf{select}_1(B_2, c+1) - 1\}$ consists of all $1 \leq j' \leq n'$ such that the edge leaving node $D_{j'}$ is labeled with c.

We can now extend the properties of the Burrows-Wheeler Transform and the FM-index to $\mathcal{T}$. In particular, we will show that BWT if an encoding of $G_{\mathcal{T}}$. This result shows that BWT is related to the eBWT of Mantaci et al. [40], but there are two main differences: (1) we do not need assumptions (i-ii) in Section 1 to define BWT (in particular, the T_h's need not be primitive), and (2) BWT is an encoding of $G_{\mathcal{T}}$ but not of $\mathcal{T}$ (namely, BWT encodes the cyclic nature of the T_h's and not a specific circular suffix of each T_h). To extend the FM-index to $\mathcal{T}$, we will once again base our intuition on $G_{\mathcal{T}}$.

Recall that two (directed edge-labeled) graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there exists a bijection f from V_1 to V_2 such that for every $u, v \in V_1$ and for every $c \in \Sigma$ we have $(u, v, c) \in E_1$ if and only if $(f(u), f(v), c) \in E_2$. In other words, two graphs are isomorphic if they are the same graph up to renaming the nodes.

▶ **Theorem 9.** *Let $\mathcal{T}$ be a dictionary.*

1. BWT *is an encoding of $G_{\mathcal{T}}$, that is, given* BWT*, we can retrieve $G_{\mathcal{T}}$ up to isomorphism.*
2. *There exists a data structure encoding $G_{\mathcal{T}}$ of $n' \log \sigma(1 + o(1)) + O(n')$ bits that supports the following queries in $O(\log \log \sigma)$ time: (i)* access*,* rank*, and* select *queries on* BWT*, (ii)* $\mathsf{bws}(\ell, r, c)$*: given $1 \leq \ell \leq r \leq n'$ and $c \in \Sigma$, decide if* $\mathsf{back}(\{\ell, \ell+1, \ldots, r\}, c)$ *is empty and, if it is nonempty, return ℓ' and r' such that* $\mathsf{back}(\{\ell, \ell+1, \ldots, r\}, c) = \{\ell', \ell'+1, \ldots, r'\}$*, (iii)* $\mathsf{prev}(j)$*: given $1 \leq j \leq n'$, return $1 \leq j' \leq n'$ such that* $\mathsf{pred}(D_j) = D_{j'}$*, and (iv)* $\mathsf{follow}(j')$*: given $1 \leq j' \leq n'$, return $1 \leq j \leq n'$ such that* $\mathsf{pred}(D_j) = D_{j'}$*.*

Note that, even if BWT is an encoding of $G_{\mathcal{T}}$, it is not an encoding of $\mathcal{T}$ (in particular, from $G_{\mathcal{T}}$ we cannot recover $\mathcal{T}$). This is true even when all D_j's are singletons because $G_{\mathcal{T}}$ only stores circular strings, so we cannot retrieve the bitvector B_1 the marks the beginning of each string (we cannot even retrieve the specific enumeration $T_1, T_2, \ldots, T_d$ of the strings in $\mathcal{T}$).

t	1	2	3	4	5	6	7	8	9	10	11
$B_3[t]$	1	0	0	1	0	0	0	0	1	0	0
$\mathsf{SA}[t]$	1	13	9	2	14	7	10	3	12	8	11
$B_4[t]$	1	0	1	1	0	1	1	1	0	1	1
$\mathsf{SA}^*[t]$	1	9	14	7	3	12	11				
$B_5[t]$	1	0	1	0	1	1	0	1	1	0	1
$\mathsf{Len}[t]$	6	3	5	6	3	5	5	6	3	5	5

Figure 3 The compressed suffix array of the dictionary $\mathcal{T} = (abcabc, bcabc, cab)$ in Figure 1. We use the sampling factor $s = 2$.

3.2 The Compressed Suffix Array of $\mathcal{T}$

We know that $\mathcal{D} = (D_1, D_2, \ldots, D_{n'})$ is an ordered partition of $\{1, 2, \ldots, n\}$, and we know that $\mathcal{S}_1 \prec \mathcal{S}_2 \prec \cdots \prec \mathcal{S}_{n'}$. The suffix array of $\mathcal{T}$ should be defined in such a way that we can answer the following query: given $1 \leq j \leq n'$, return the set D_j.

In the following, we say that a position $1 \leq k \leq n$ *refers* to the string T_h if $h = \mathsf{rank}_1(B_1, k)$. Every position refers to exactly one string. Notice that a set D_j may contain elements that refer to the same string T_h. In Figure 1, we have $2, 5 \in D_3$, and both 2 and 5 refer to T_1. This happens because T_1 is not a primitive string. Let us show that, if we know *any* element of D_j referring to T_h, we can retrieve *all* elements of D_j referring to T_h.

For every $1 \leq h \leq d$, let ρ_h be the root of T_h. Then, $|T_h|$ is divisible by $|\rho_h|$, and for every $1 \leq k \leq n$ the root $\rho(\mathcal{T}_k)$ of $\mathcal{T}_k$ is $\mathcal{T}_k[1, \rho_h]$, where k refers to string T_h. For every $1 \leq h \leq d$, let $k_h = \mathsf{select}_1(B_1, h)$ be the index in $\{1, 2, \ldots, n\}$ corresponding to the first character of T_h.

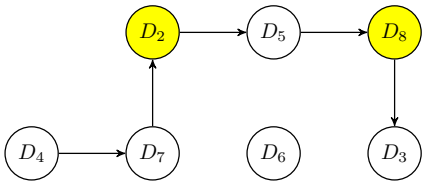
Fix $1 \leq j \leq n'$ and $1 \leq h \leq d$, and let $k \in D_j$ *any* position referring to T_h. For every k' referring to T_h, we have $k' \in D_j$ if and only if $\rho(\mathcal{T}_k) = \rho(\mathcal{T}_{k'})$ (by Lemma 2), if and only if $\mathcal{T}_k[1, \rho_h] = \mathcal{T}_{k'}[1, \rho_h]$, if and only if $|k' - k|$ is a multiple of $|\rho_h|$. Then, if G is the set of *all* elements of D_j referring to T_h, we have $G = \{k_h + (k - k_h + w|\rho_h| \bmod |T_h|) \mid 0 \leq w \leq (|T_h|/|\rho_h|) - 1\}$. For example, if Figure 1b, if we consider $j = 3$ and $h = 1$ and we pick $k = 5$, then $5 \in D_3$, 5 refers to T_1, $\rho_1 = abc$, $|\rho_1| = 3$, $|T_1| = 6$, $k_1 = 1$ and $G = \{2, 5\}$.

To compute G we need to compute $|T_h|$ and $|\rho_h|$. We have already seen that we can compute $|T_h|$ in $O(1)$ time using B_1 (see Section 3.1), and we now store a bitvector B_3 to compute $|\rho_h|$ in $O(1)$ time (see Figure 3). The bitvector B_3 contains exactly d ones, we have $B_3[1] = 1$, and for every $1 \leq h \leq d$ the number of consecutive zeros after the h-th one is $|\rho_h| - 1$. Then, $|\rho_h| = \mathsf{select}_1(B_3, h + 1) - \mathsf{select}_1(B_3, h)$ for every $1 \leq h \leq d$.

Let us define a suffix array for $\mathcal{T}$. Let $\sim$ be the equivalence relation on $\{1, 2, \ldots, n\}$ such that for every $1 \leq k, k^* \leq n$ we have $k \sim k^*$ if and only if k and k^* belong to the same D_j and refer to the same string T_h. Fix $1 \leq j \leq n'$. Notice that D_j is the union of some $\sim$-equivalence classes. Let D'_j be the subset of D_j obtained by picking the smallest element of each $\sim$-equivalence class contained in D_j. In Figure 1b, for $j = 3$, the partition of D_3 induced by $\sim$ is $\{\{2, 5\}, \{14\}\}$, so $D'_3 = \{2, 14\}$. Then SA is the array obtained by concatenating the elements in D'_1, the elements in D'_2, ..., the elements in $D'_{n'}$ (see Figure 3), where the elements in D'_j are sorted from smallest to largest. Equivalently, the elements in each D'_j are sorted according to the indexes of the strings to which they refer (by definition, two distinct elements of D'_j cannot refer to the same string). We also define a bitvector B_4 to mark the beginning of each D'_j (see again Figure 3). More precisely, B_4 contains exactly n' ones and for every $1 \leq j \leq n'$ the number of consecutive zeros after the j-th one is $|D'_j| - 1$.

j	2	3	4	5	6	7	8
$\mathsf{LCP}[j]$	3	0	5	2	0	4	1
$B_6[j]$	1	0	0	0	0	0	1
$\mathsf{LCP}^*[j]$	3	1					

(a)



(b)

Figure 4 The LCP array of the dictionary $\mathcal{T} = (abcabc, bcabc, cab)$ in Figure 1. We use the sampling factor $s = 2$. The graph in (b) is the graph Q used to determine which values will be sampled. Yellow nodes are the sampled nodes.

Fix $1 \leq k \leq n$, where $k \in D_j$ refers to string T_h. Note that there exists t such that $\mathsf{SA}[t] = k$ if and only if $k \in D'_j$, if and only if $k_h \leq k \leq k_h + |\rho_h| - 1$. Moreover, for every $k \in D_{j'}$ we have that $[k]_\sim = \{k + w|\rho_h| \mid 0 \leq w \leq (|T_h|/|\rho_h|) - 1\}$ is the set of all elements of D_j referring to T_h. In particular, SA is an array of length $n^* = \sum_{j=1}^{n'} |D'_j| = \sum_{h=1}^{d} |\rho_h|$ (in Figure 3, we have $n^* = 11$).

The suffix array SA has the desired property: given $1 \leq j \leq n'$, we can compute the set D_j in $O(|D_j|)$ time as follows. We first retrieve D'_j by observing that its elements are stored in $\mathsf{SA}[t]$, $\mathsf{SA}[t+1]$, $\mathsf{SA}[t+2]$, $\ldots$, $\mathsf{SA}[t']$, where $t = \mathsf{select}_1(B_4, j)$ and $t' = \mathsf{select}_1(B_4, j+1) - 1$. Then, for every $k \in D'_j$, we know that k refers to string T_h, where $h = \mathsf{rank}_1(B_1, k)$, and we can compute the set $[k]_\sim$ of all elements of D_j referring to T_h as shown above. Then, we have $D_j = \bigcup_{k \in D'_j} [k]_\sim$ and the union is disjoint.

Storing SA explicitly can require up to $n \log n$ bits, so to achieve the space bound in Theorem 1 we need a sampling mechanism similar to the compressed suffix array of a string: we will sample some entries and we will reach a sampled entry by using the backward search to navigate the string (see [42]). More precisely, in our setting we will use the query $\mathsf{prev}(j)$ of Theorem 9 to navigate $G_\mathcal{T}$ in a backward fashion. Note that in the conventional case of a string, if we start from the end of the string and we apply the backward search, we can reach each position of the string, but in our setting the graph $G_\mathcal{T}$ is the disjoint union of some cycles (see Figure 2), and the backward search does not allow navigating from a cycle to some other cycle. This means that we will need to sample at least one value per cycle. In the worst case, we have d cycles (one for each T_h), so in the worst case we need to sample at least d values.

After choosing an appropriate sampling factor s, we store the sampled values in an array SA^* (respecting the order in SA), and we use a bitvector B_5 to mark the entries of $\mathsf{SA}[k]$ that have been sampled (see Figure 3). We finally obtain the following theorem.

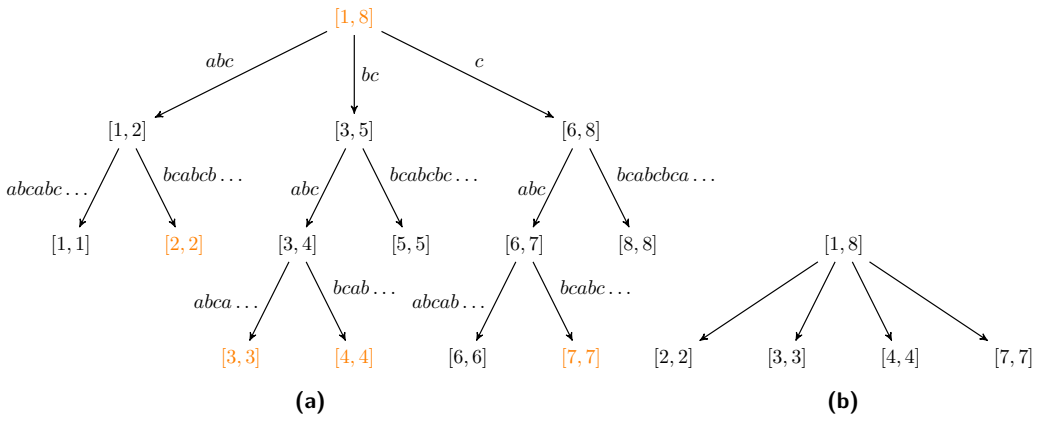
▶ **Theorem 10.** *Let $\mathcal{T}$ be a dictionary. By storing $o(n \log \sigma) + O(n) + O(d \log n)$ bits, we can compute each entry $\mathsf{SA}[t]$ in $O(\log n)$ time.*

We conclude that, by storing the data structure of Theorem 10, for every $1 \leq j \leq n'$ we can compute D_j in $O(|D_j| \log n)$ time.

3.3 The LCP Array of $\mathcal{T}$

Let us define the longest common prefix (LCP) array of $\mathcal{T}$. We know that $\mathcal{S}_1 \prec \mathcal{S}_2 \prec \cdots \prec \mathcal{S}_{n'}$, so it is natural to give the following definition (see Figure 1b and Figure 4a).

▶ **Definition 11.** *Let $\mathcal{T}$ be a dictionary. Let $\mathsf{LCP} = \mathsf{LCP}[2, n']$ be the array such that $\mathsf{LCP}[j] = \mathsf{lcp}(\mathcal{S}_{j-1}, \mathcal{S}_j)$ for every $2 \leq j \leq n'$.*



t	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8
$Z_{\mathsf{Suff}(\mathcal{T})}[t]$	1	1	1	0	1	0	0	1	1	1	0	1	0	0	1	0	0	1	1	1	0	1	0	0	1	0	0	0
$B_7[t]$	0	0	1	0	1	0	0	0	0	1	0	1	0	0	1	0	0	0	0	1	0	1	0	0	1	0	0	0
$B_8[t]$	1	0	0	0	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
$Z_{\mathsf{Suff}^*(\mathcal{T})}[t]$	1	1	0	1	0	1	0	1	0	0																		

(c)

Figure 5 (a) The suffix tree $\mathsf{Suff}(\mathcal{T})$ of the dictionary $\mathcal{T} = (abcabc, bcabc, cab)$ in Figure 1. Marked nodes are orange. (b) The tree $\mathsf{Suff}^*(\mathcal{T})$. (c) The bit arrays to navigate $\mathsf{Suff}(\mathcal{T})$ and $\mathsf{Suff}^*(\mathcal{T})$. The first row compactly includes all indexes t from 1 to 28.

Note that each $\mathsf{LCP}[j]$ is finite because the $\mathcal{S}_j$'s are pairwise distinct. Let us prove a stronger result: $\mathsf{LCP}[j] \leq n' - 2$ for every $2 \leq j \leq n'$ (Lemma 12). This implies that we can store an entry $\mathsf{LCP}[j]$ using at most $\log n$ bits.

▶ **Lemma 12.** *Let $\mathcal{T}$ be a dictionary. Then, $\mathsf{LCP}[j] \leq n' - 2$ for every $2 \leq j \leq n'$.*

Storing the LCP array explicitly can require $n \log n$ bits, so to achieve the space bound of Theorem 1 we need a sampling mechanism similar to the one for suffix arrays in Section 3.2.

Recall that, given an array $A[1, n]$, we can define *range minimum queries* as follows: for every $1 \leq i \leq j \leq n$, $\mathsf{rmq}_A(i, j)$ returns an index $i \leq k \leq j$ such that $A[k] = \min_{i \leq h \leq j} A[h]$. There exists a data structure of $2n + o(n)$ bits that can solve a query $\mathsf{rmq}_A(i, j)$ in $O(1)$ time *without accessing* A [28]; moreover the data structure always returns the *smallest* $i \leq k \leq j$ such that $A[k] = \min_{i \leq h \leq j} A[h]$.

Let us store a data structure for solving range minimum queries $\mathsf{rmq}_{\mathsf{LCP}}$ on LCP in $O(1)$ time. We will use this data structure to retrieve each entry of the LCP array from our sampled LCP array. The main idea is to use the sampling mechanism in [23]: an auxiliary graph Q allows determining which values will be sampled based on a sampling parameter s (see Figure 4b). Then, we define an array LCP^* that stores the sampled values (respecting the order in LCP, see Figure 4a), and a bitvector B_6 to mark the sampled entries of LCP (see Figure 4a). By choosing an appropriate s, we obtain the following result.

▶ **Theorem 13.** *Let $\mathcal{T}$ be a dictionary. By storing $o(n \log \sigma) + O(n)$ bits, we can compute each entry $\mathsf{LCP}[j]$ in $O(\log n)$ time.*

3.4 The Topology of the Suffix Tree of $\mathcal{T}$

We can now introduce the suffix tree $\mathsf{Suff}(\mathcal{T})$ of our dictionary $\mathcal{T}$ (the details are discussed in the full version). Refer to Figure 5a for an example. The (infinite) set of all finite strings that we can read starting from the root is $\mathcal{P} = \{\mathcal{S}_j[1,t] \mid 1 \leq j \leq n', t \geq 0\}$. Every node is an interval that describes the strings that can be read starting from the root and reading a nonempty prefix of the last edge. For example, the set of all strings P for which P is a prefix of $\mathcal{S}_j$ if and only if $3 \leq j \leq 4$ is $\{bca, bcab, bcabc\}$ and, in fact, the set of all strings that reach node $[3,4]$ is $\{bca, bcab, bcabc\}$. The suffix tree contains exactly n' leaves (namely, $[j,j]$ for every $1 \leq j \leq n'$), and the set $\mathcal{N}$ of all nodes has size at most $2n'-1$. The set of all strings reaching a node $[\ell,r]$ is finite if and only if $[\ell,r]$ is an internal node, and in this case the longest string reaching $[\ell,r]$ has length $\lambda_{\ell,r} = \mathsf{lcp}(\mathcal{S}_\ell, \mathcal{S}_r)$, which can be computed by using the LCP array (for example, $\lambda_{3,4} = \mathsf{lcp}(\mathcal{S}_3, \mathcal{S}_4) = 5$). The suffix tree $\mathcal{T}$ is an ordinal tree, and any ordinal tree T with t nodes can be encoded using its *balanced paranthesis* representation $Z_T[1,2t]$, a bit array that we can build incrementally as follows. Visit all nodes of the tree in depth-first search order (respecting the order of the children of each node) starting from the root. Every time we encounter a new node u we append an 1, and as soon as we leave the subtree rooted at u we add a 0. Hence $Z_T[1,2t]$ is a bit array containing exactly t values equal to 1 and t values equal to 0 (see Figure 5c for the balanced parenthesis representation $Z_{\mathsf{Suff}(\mathcal{T})}$ of $\mathsf{Suff}(\mathcal{T})$). Navarro and Sadakane showed how to support navigational queries on the balanced representation of an ordinal tree in compact space and constant time (see the full version), and we store such a data structure for $\mathsf{Suff}(\mathcal{T})$ (which requires $O(n)$ bits). To correctly switch to the balanced parenthesis representation of $\mathsf{Suff}(\mathcal{T})$, we also store a bitvector B_7 of length $2|\mathcal{N}|$ such that for every $1 \leq t \leq 2|\mathcal{N}|$ we have $B_7[t] = 1$ if and only if (i) $Z_{\mathsf{Suff}(\mathcal{T})}[t] = 1$ and (ii) the index t corresponds to a leaf of $\mathsf{Suff}(\mathcal{T})$ (see Figure 5c).

4 Solving Circular Dictionary Matching Queries

In the following, we consider a pattern $P = P[1,m]$ of length m, and we want to compute $\mathsf{Cdm}(\mathcal{T}, P)$.

Fix $1 \leq i \leq m$. We need to compute all $1 \leq k \leq n$ such that $\mathcal{T}_k$ occurs in P at position i. In general, if $k \in D_j$, then $\mathcal{S}_j = \mathcal{T}_k^\omega$, hence $\mathcal{T}_k = \mathcal{S}_j[1, |\mathcal{T}_k|] \in \mathcal{P}$, which means that $\mathcal{T}_k$ identifies a node of $\mathsf{Suff}(\mathcal{T})$. This means that, if k occurs in P at position i, then there exists $i' \geq i$ such that $P[i,i'] \in \mathcal{P}$, and then every prefix of $P[i,i']$ is also in $\mathcal{P}$ (because every prefix of a string in $\mathcal{P}$ is also in $\mathcal{P}$). Consequently, by considering the *largest* i^* such that $P[i,i^*] \in \mathcal{P}$, we know that we only need to consider $P[i,i^*]$ to compute all $1 \leq k \leq n$ such that $\mathcal{T}_k$ occurs in P at position i. We can then give the following definition.

▶ **Definition 14.** *Let $\mathcal{T}$ be a dictionary, and let $P \in \Sigma^*$. For every $1 \leq i \leq m$, let t_i be the largest integer such that $i-1 \leq t_i \leq m$ and $P[i,t_i] \in \mathcal{P}$.*

Note that t_i is well-defined for every $1 \leq i \leq m$ because $P[i,i-1] = \epsilon \in \mathcal{P}$.

Fix $1 \leq i \leq m$, $1 \leq j \leq n'$ and $k \in D_j$. If $j \in [\ell_{P[i,t_i]}, r_{P[i,t_i]}]$, then $P[i,t_i]$ is a prefix of $\mathcal{S}_j = \mathcal{T}_k^\omega$ and so $\mathcal{T}_k$ occurs in P at position i if and only $|\mathcal{T}_k| \leq |P[i,t_i]|$. If $j \notin [\ell_{P[i,t_i]}, r_{P[i,t_i]}]$, we know that $P[i,t_i]$ is not a prefix of $\mathcal{T}_k^\omega$, but it might still be true that a prefix of $P[i,t_i]$ is equal to $\mathcal{T}_k$. Let $P[i,i^*]$ be the *longest* prefix of $P[i,t_i]$ that is a prefix of $\mathcal{T}_k^\omega$ (or equivalently, let i^* be the largest integer such that $j \in [\ell_{P[i,i^*]}, r_{P[i,i^*]}]$). Then, $\mathcal{T}_k$ occurs in P at position i if and only $|\mathcal{T}_k| \leq |P[i,i^*]|$. To compute $|P[i,i^*]|$, first notice that every prefix of $P[i,t_i]$ reaches an ancestor of $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$, and every string reaching

a strict ancestor of $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ is a strict prefix of $P[i,t_i]$. As a consequence, we can first compute the nearest ancestor $[\ell', r']$ of $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ for which $j \in [\ell, r]$, and then $|P[i,i^*]|$ is the length $\lambda_{\ell',r'}$ of the largest string reaching node $[\ell', r']$.

The following lemma captures our intuition.

▶ **Lemma 15.** *Let $\mathcal{T}$ be a dictionary, let $P \in \Sigma^*$, let $1 \le i \le m$ and let $1 \le j \le n'$.*

1. *Assume that $j \in [\ell_{P[i,t_i]}, r_{P[i,t_i]}]$. Then, for every $k \in D_j$ we have that $\mathcal{T}_k$ occurs in P at position i if and only if $|\mathcal{T}_k| \le t_i - i + 1$.*
2. *Assume that $j \notin [\ell_{P[i,t_i]}, r_{P[i,t_i]}]$. Let $[\ell, r]$ be the nearest ancestor of $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ in $\mathsf{Suff}(\mathcal{T})$ for which $j \notin [\ell, r]$. Then, $[\ell, r]$ is not the root of $\mathsf{Suff}(\mathcal{T})$. Moreover, let $[\ell', r']$ be the parent of $[\ell, r]$ in $\mathsf{Suff}(\mathcal{T})$. Then, $j \in [\ell', \ell - 1] \cup [r + 1, r']$, and for every $k \in D_j$ we have that $\mathcal{T}_k$ occurs in P at position i if and only if $|\mathcal{T}_k| \le \lambda_{\ell',r'}$.*

Fix $1 \le i \le m$. To implement Lemma 15, we should navigate $\mathsf{Suff}(\mathcal{T})$ starting from node $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$, so we need to know $\ell_{P[i,t_i]}$ and $r_{P[i,t_i]}$. Moreover, if $j \in [\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ and $k \in D_j$, we know that $\mathcal{T}_k$ occurs in P at position i if and only if $|\mathcal{T}_k| \le t_i - i + 1$, so we also need to compute t_i. In the full version, we present an $O(m \log n)$ algorithm to compute all the t_i's, all the $\ell_{P[i,t_i]}$'s and all the $r_{P[i,t_i]}$'s. The algorithm shares many similarities with Ohlebusch et al.'s algorithm for computing matching statistics using the FM-index and the LCP array of a string [44].

For every $1 \le i \le m$, let $\mathsf{Cdm}_i(\mathcal{T}, P)$ be the set of all $1 \le k \le n$ such that $\mathcal{T}_k$ occurs in P at position i, and let $occ_i = |\mathsf{Cdm}_i(\mathcal{T}, P)|$. Computing $\mathsf{Cdm}(\mathcal{T}, P)$ is equivalent to computing $\mathsf{Cdm}_i(\mathcal{T}, P)$ for every $1 \le i \le m$, and $occ = \sum_{i=1}^{m} occ_i$. In view of Theorem 1, it will be sufficient to show that we can compute $\mathsf{Cdm}_i(\mathcal{T}, P)$ in $O((1 + occ_i) \log n)$ time, because then we can compute each $\mathsf{Cdm}(\mathcal{T}, P)$ in $\sum_{i=1}^{m} O((1 + occ_i) \log n) = O((m + occ) \log n)$ time.

Fix $1 \le i \le m$. Lemma 15 suggests that, to compute $\mathsf{Cdm}_i(\mathcal{T}, P)$, we can proceed as follows. We start from node $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ is $\mathsf{Suff}(\mathcal{T})$, we consider every $j \in [\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ and every $k \in D_j$, and we decide whether $\mathcal{T}_k$ occurs in P at position i (we will see later how to do this efficiently). Next, we navigate from node $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ up to the root. Every time we move from a node $[\ell, r]$ to its parent $[\ell', r']$, we consider every $j \in [\ell', \ell - 1] \cup [r + 1, r']$ and every $k \in D_j$, and we decide whether $\mathcal{T}_k$ occurs in P at position i (again, we will see later how to do this efficiently). To navigate $\mathsf{Suff}(\mathcal{T})$ from $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ to the root, in the worst case we visit many nodes – say $\Omega(\log n)$ nodes. If each of these steps leads to discovering at least one element in $\mathsf{Cdm}_i(\mathcal{T}, P)$ we can hope to achieve the bound $O((1 + occ_i) \log n)$, but in the worst case many steps are not successful, occ_i is small and we cannot achieve the bound $O((1 + occ_i) \log n)$.

Notice that i only determines the initial node $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$, but once we are navigating $\mathsf{Suff}(\mathcal{T})$ up to the root, we do not need i to assess whether we have found an element of $\mathsf{Cdm}_i(\mathcal{T}, P)$, because $\mathcal{T}_k$ occurs in P at position i if and only if $|\mathcal{T}_k| \le \lambda_{\ell',r'}$, and $\lambda_{\ell',r'}$ does not depend on i. This means that we can determine which nodes will allow discovering an element of $\mathsf{Cdm}_i(\mathcal{T}, P)$ *before* knowing the pattern P (that is, at indexing time). We can then give the following definition which, crucially, does not depend on i or P (see Figure 5a).

▶ **Definition 16.** *Let $\mathcal{T}$ be a dictionary, and let $[\ell, r] \in \mathcal{N}$. We say that $[\ell, r]$ is* marked *if one of the following is true: (i) $[\ell, r] = [1, n']$ (i.e., $[\ell, r]$ is the root of $\mathsf{Suff}(\mathcal{T})$), or (ii) $[\ell, r] \neq [1, n']$ (i.e., $[\ell, r]$ is not the root of $\mathsf{Suff}(\mathcal{T})$) and, letting $[\ell', r']$ be the parent of $[\ell, r]$ in $\mathsf{Suff}(\mathcal{T})$, there exists $j \in [\ell', \ell - 1] \cup [r + 1, r']$ and there exists $k \in D_j$ such that $|\mathcal{T}_k| \le \lambda_{\ell',r'}$.*

When we navigate $\mathsf{Suff}(\mathcal{T})$ from $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ to the root, we should skip all non-marked nodes. Notice the set of all marked nodes induces a new tree structure. More precisely, consider the ordinal tree $\mathsf{Suff}^*(\mathcal{T})$ with root $[1, n']$ defined as follows. The set $\mathcal{N}^*$ of nodes is

the set of all marked nodes in $\mathcal{N}$; in particular, the root $[1, n']$ of $\mathsf{Suff}(\mathcal{T})$ belongs to $\mathcal{N}^*$. We can now build $\mathsf{Suff}^*(\mathcal{T})$ incrementally. We traverse all nodes of $\mathsf{Suff}(\mathcal{T})$ in depth-first search order, respecting the order of the children of each node (this is the same order used for the balanced parenthesis representation of $\mathsf{Suff}(\mathcal{T})$). The first marked node that we encounter is $[1, n']$, which will be root of $\mathsf{Suff}^*(\mathcal{T})$. Then, every time we encounter a marked node $[\ell, r]$, let $[\ell', r']$ be the nearest strict ancestor of $[\ell, r]$ in $\mathsf{Suff}(\mathcal{T})$ that is marked, namely, $[\ell', r']$ is the first marked node that we encounter after $[\ell, r]$ in the (backward) path from $[\ell, r]$ to $[1, n']$ in $\mathsf{Suff}(\mathcal{T})$. Then, $[\ell', r']$ will be the parent of $[\ell, r]$ in $\mathsf{Suff}^*(\mathcal{T})$, and if $[\ell', r']$ has already been assigned $q \geq 0$ children, then $[\ell, r]$ will be its $(q+1)$-th smallest child. See Figure 5b for an example, and see Figure 5c for the balanced parenthesis representation of $\mathsf{Suff}^*(\mathcal{T})$. In addition to Navarro and Sadakane's data structure for the tree $\mathsf{Suff}(\mathcal{T})$, we also store the same data structure for the tree $\mathsf{Suff}^*(\mathcal{T})$, which also requires $O(n)$ bits.

We also remember which nodes of $\mathsf{Suff}(\mathcal{T})$ are marked by using a bitvector B_8 of length $2|\mathcal{N}|$ such that for every $1 \leq t \leq 2|\mathcal{N}|$ we have $B_8[t] = 1$ if and only if $Z_{\mathsf{Suff}(\mathcal{T})}[t]$ is one of the two values corresponding to a marked node of $\mathsf{Suff}(\mathcal{T})$. More precisely, we build B_8 as follows. We visit all nodes of $\mathsf{Suff}(\mathcal{T})$ in depth-first search order, respecting the order of the children of each node. Every time we encounter a new node u we append an 1 if u is marked and a 0 if u is not marked, and as soon as we leave the subtree rooted at u we add a 1 if u is marked and a 0 if u is not marked (see Figure 5c). By using B_8, in $O(1)$ time (i) we can move from $\mathsf{Suff}(\mathcal{T})$ to $\mathsf{Suff}^*(\mathcal{T})$ and from $\mathsf{Suff}^*(\mathcal{T})$ to $\mathsf{Suff}(\mathcal{T})$ and (ii) we can determine the nearest marked ancestor of a node (see the full version).

Define the array Len of length $n^* = \sum_{h=1}^{d} |\rho_h|$ (the length of SA) such that $\mathsf{Len}[t] = |\mathcal{T}_{\mathsf{SA}[t]}|$ for every t (see Figure 3). We will not store Len, but only a data structure supporting range minimum queries on Len in $O(1)$ time, which requires $O(n)$ bits (see Section 3.3).

We now have all the ingredients to prove Theorem 1, our main claim. As we have seen, it will suffice to compute each $\mathsf{Cdm}_i(\mathcal{T}, P)$ in $O((1 + occ_i) \log n)$. We start from node $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$, and for every $k \in \bigcup_{\ell_{P[i,t_i]} \leq j \leq r_{P[i,t_i]}} D_j$ we determine whether $\mathcal{T}_k$ occurs in P at position i. By Lemma 15, we need to check whether $|\mathcal{T}_k| \leq t_i - i + 1$, so we repeatedly solve range minimum queries on Len starting from the interval $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$ to find all k's for which $\mathcal{T}_k$ occurs in P at position i in time proportional to the number of such occurrences. Next, we compute the lowest marked ancestor of $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$, and we use $\mathsf{Suff}^*(\mathcal{T})$ to determine all marked ancestors of $[\ell_{P[i,t_i]}, r_{P[i,t_i]}]$. Let $[\ell, r]$ be one of these ancestors, and let $[\ell', r']$ be its parent in $\mathsf{Suff}(\mathcal{T})$. For every $k \in (\bigcup_{\ell' \leq j \leq \ell - 1} D_j) \cup (\bigcup_{r+1 \leq j \leq r'} D_j)$, we determine whether $\mathcal{T}_k$ occurs in P at position i. By Lemma 15, we need to check whether $|\mathcal{T}_k| \leq \lambda_{\ell', r'}$, so we repeatedly solve range minimum queries on Len starting from the intervals $[\ell', \ell - 1]$ and $[r + 1, r']$ to find all k's for which $\mathcal{T}_k$ occurs in P at position i in time proportional to the number of such occurrences.

The details of the proof of Theorem 1 are in the full version. Note that we cannot directly infer that our data structure is an encoding of $\mathcal{T}$ from Theorem 9 because the graph $G_{\mathcal{T}}$ is not sufficient to retrieve $\mathcal{T}$.

5 Conclusions and Future Work

We have shown how to improve and extend previous results on circular dictionary matching. In the literature, much effort has been devoted to designing construction algorithms for the data structure of Hon et al. [36] and, implicitly, the eBWT of Mantaci et al. [40]. All available approaches first build the eBWT and the suffix array of circular strings, and then use the suffix array of circular strings to build the remaining auxiliary data structures. In

[32], Hon et al. showed that the eBWT and the suffix array can be built in $O(n \log n)$ time using $O(n \log \sigma + d \log n)$ bits of working space. Bannai et al. [7] improved the time bound to $O(n)$ by showing that the bijective BWT can be built in linear time (Bonomo et al. [10, Section 6] showed how to reduce in linear time the problem of computing the eBWT to the problem of computing the bijective BWT). A more direct algorithm was proposed by Boucher et al. [11]. However, all these algorithms are still based on assumptions (i-ii) in Section 1 and cannot immediately applying to our setting in which we consider an arbitrary dictionary. After building the eBWT and the suffix array, it is possible to build the remaining auxiliary data structure in $O(n \log n)$ time using $O(n \log \sigma + d \log n)$ bits of working space [33]. Some proofs in [33] are only sketched, but it is not too difficult to show that, if we do not insist on achieving $O(n \log \sigma + d \log n)$ bits of working space, it is possible to build the remaining auxiliary data structure from the eBWT and the suffix array in linear time.

In a companion paper, we will show that the data structure of Theorem 1 can be built in $O(n)$ time. The main technical issue is understanding how to remove assumptions (i-ii) in Section 1 when building the suffix array. The algorithm by Boucher et al.'s [11] is a recursive algorithm based on the SAIS algorithm [43] for building the suffix array. SAIS divides all suffixes into two categories (S-type and L-type). We will show that, to remove assumptions (i-ii), we will use three (and not two) categories, building on a recent recursive algorithm [19] for constructing the "suffix array" of a deterministic automaton.

The natural question is whether the data structure of Theorem 1 (or a similar data structure with the same functionality) can be built in $O(n)$ time within $O(n \log \sigma + d \log n)$ bits of working space. We conjecture that this is possible but, thinking of the case $d = 1$, this problem should be at least as difficult as building the compressed suffix tree of a string in $O(n)$ time and $O(n \log \sigma)$ working bits, which requires advanced techniques [41, 9].

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975. doi:10.1145/360825.360855.

2 Jarno Alanko, Nicola Cotumaccio, and Nicola Prezza. Linear-time minimization of Wheeler DFAs. In *2022 Data Compression Conference (DCC)*, pages 53–62. IEEE, 2022. doi:10.1109/DCC52660.2022.00013.

3 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular languages meet prefix sorting. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 911–930. SIAM, 2020. doi:10.1137/1.9781611975994.55.

4 Jarno N Alanko, Davide Cenzato, Nicola Cotumaccio, Sung-Hwan Kim, Giovanni Manzini, and Nicola Prezza. Computing the LCP array of a labeled graph. In *35th Annual Symposium on Combinatorial Pattern Matching (CPM 2024)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

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*, 81:2123–2157, 2019. doi:10.1007/S00453-018-0526-2.

6 Amihood Amir, Avivit Levy, Ely Porat, and B Riva Shalom. Dictionary matching with a few gaps. *Theoretical Computer Science*, 589:34–46, 2015. doi:10.1016/J.TCS.2015.04.011.

7 Hideo Bannai, Juha Kärkkäinen, Dominik Köppl, and Marcin Piątkowski. Constructing the bijective and the extended Burrows-Wheeler transform in linear time. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

8 Djamal Belazzougui. Succinct dictionary matching with no slowdown. In Amihood Amir and Laxmi Parida, editors, *Combinatorial Pattern Matching*, pages 88–100, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-13509-5_9.

9 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transactions on Algorithms (TALG)*, 16(2):1–54, 2020. doi:10.1145/3381417.

10 Silvia Bonomo, Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. Sorting conjugates and suffixes of words in a multiset. *International Journal of Foundations of Computer Science*, 25(08):1161–1175, 2014.

11 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*, pages 129–142, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-86692-1_11.

12 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Systems Research Center, 1994.

13 Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone. Computing the optimal BWT of very large string collections. In *2023 Data Compression Conference (DCC)*, pages 71–80, 2023. doi:10.1109/DCC55655.2023.00015.

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

15 Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Solon P Pissis, Wojciech Rytter, Tomasz Waleń, and Wiktor Zuba. Approximate circular pattern matching. In *ESA 2022-30th Annual European Symposium on Algorithms*, 2022.

16 Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya. Dictionary matching in a stream. In *Algorithms-ESA 2015: 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, pages 361–372. Springer, 2015. doi:10.1007/978-3-662-48350-3_31.

17 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on Wheeler DFAs. In *2023 Data Compression Conference (DCC)*, pages 150–159, 2023. doi:10.1109/DCC55655.2023.00023.

18 Nicola Cotumaccio. Graphs can be succinctly indexed for pattern matching in $O(|E|^2 + |V|^{5/2})$ time. In *2022 Data Compression Conference (DCC)*, pages 272–281, 2022. doi:10.1109/DCC52660.2022.00035.

19 Nicola Cotumaccio. Prefix sorting DFAs: A recursive algorithm. In *34th International Symposium on Algorithms and Computation (ISAAC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

20 Nicola Cotumaccio. A Myhill-Nerode theorem for generalized automata, with applications to pattern matching and compression. In *41st International Symposium on Theoretical Aspects of Computer Science (STACS 2024)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

21 Nicola Cotumaccio. Improved circular dictionary matching, 2025. arXiv:2504.03394.

22 Nicola Cotumaccio, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Colexicographically ordering automata and regular languages - part I. *J. ACM*, 70(4), August 2023. doi:10.1145/3607471.

23 Nicola Cotumaccio, Travis Gagie, Dominik Köppl, and Nicola Prezza. Space-time trade-offs for the LCP array of Wheeler DFAs. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval*, pages 143–156, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-43980-3_12.

24 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2585–2599. SIAM, 2021. doi:10.1137/1.9781611976465.153.

25 Nicola Cotumaccio and Catia Trubiani. Convex Petri nets. In Michael Köhler-Bussmeier, Daniel Moldt, and Heiko Rölke, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering 2024 co-located with the 45th International Conference on Application and Theory of Petri Nets and Concurrency (PETRI NETS 2024), June 24 - 25,*

2024, Geneva, Switzerland, volume 3730 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2024. URL: https://ceur-ws.org/Vol-3730/poster1.pdf.

26 Jonathan A Eisen. Environmental shotgun sequencing: its potential and challenges for studying the hidden world of microbes. *PLoS biology*, 5(3):e82, 2007.

27 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005. doi:10.1145/1082036.1082039.

28 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.

29 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical computer science*, 698:67–78, 2017. doi:10.1016/J.TCS.2017.06.016.

30 Paweł Gawrychowski and Tatiana Starikovskaya. Streaming dictionary matching with mismatches. *Algorithmica*, pages 1–21, 2019.

31 Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Dynamic dictionary matching in the online model. In *Algorithms and Data Structures: 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5–7, 2019, Proceedings 16*, pages 409–422. Springer, 2019. doi:10.1007/978-3-030-24766-9_30.

32 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*, pages 257–268, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-31265-6_21.

33 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, and Sharma V. Thankachan. Space-efficient construction algorithm for the circular suffix tree. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching*, pages 142–152, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-38905-4_15.

34 Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Faster compressed dictionary matching. *Theoretical Computer Science*, 475:113–119, 2013. doi:10.1016/j.tcs.2012.10.050.

35 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*, 80:698–713, 2018. doi:10.1007/S00453-017-0288-2.

36 Wing-Kai Hon, Chen-Hua Lu, Rahul Shah, and Sharma V Thankachan. Succinct indexes for circular patterns. In *Algorithms and Computation: 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings 22*, pages 673–682. Springer, 2011. doi:10.1007/978-3-642-25591-5_69.

37 Costas S Iliopoulos, Solon P Pissis, and M Sohel Rahman. Searching and indexing circular patterns. *Algorithms for Next-Generation Sequencing Data: Techniques, Approaches, and Applications*, pages 77–90, 2017. doi:10.1007/978-3-319-59826-0_3.

38 Costas S. Iliopoulos and M. Sohel Rahman. Indexing circular patterns. In Shin-ichi Nakano and Md. Saidur Rahman, editors, *WALCOM: Algorithms and Computation*, pages 46–57, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-77891-2_5.

39 M. Lothaire. *Combinatorics on words*, volume 17. Cambridge university press, 1997.

40 Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the Burrows–Wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007. doi:10.1016/J.TCS.2007.07.014.

41 J Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 408–424. SIAM, 2017. doi:10.1137/1.9781611974782.26.

42 Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

43 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011. doi:10.1109/TC.2010.188.

44 Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In Edgar Chavez and Stefano Lonardi, editors, *String Processing and Information Retrieval*, pages 347–358, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-16321-0_36.

45 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theor. Comp. Sys.*, 41(4):589–607, December 2007. doi:10.1007/s00224-006-1198-x.

46 Carola Simon and Rolf Daniel. Metagenomic analyses: past and future trends. *Applied and environmental microbiology*, 77(4):1153–1161, 2011.

47 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.