Chapter 3
Searching and Indexing Circular Patterns

Costas S. Iliopoulos, Solon P. Pissis, and M. Sohel Rahman

3.1 Introduction

Circular DNA sequences can be found in viruses, as plasmids in archaea and bacteria, and in the mitochondria and plastids of eukaryotic cells. Hence, circular sequence comparison finds applications in several biological contexts [4, 5, 32]. This motivates the design of efficient algorithms [6] and data structures [21] that are devoted to the specific comparison of circular sequences, as they can be relevant in the analysis of organisms with such structure [17, 18]. In this chapter, we describe algorithms and data structures for *searching* and *indexing* patterns with a circular structure.

In order to provide an overview of the described algorithms and data structures, we begin with a few definitions generally following [10]. A *string* x of length n is an array $x[0 \ldots n-1]$, where $x[i]$, $0 \leq i < n$, is a letter drawn from some fixed *alphabet* Σ of size $\sigma = |\Sigma| = \mathcal{O}(1)$. The string of length 0 is called *empty string* and is denoted by ε. A string x is a *factor* of a string y, or x *occurs* in y, if there exist two strings u and v, such that $y = uxv$. If $u = \varepsilon$ then x is a *prefix* of y. If $v = \varepsilon$ then x is a *suffix* of y. Every occurrence of a string x in a string y can be characterized by a position in y. Thus, we say that x of length n occurs at the *starting position* i in

This chapter was written when Rahman was on a sabbatical leave from BUET.

C.S. Iliopoulos • S.P. Pissis (✉)
Department of Informatics, King's College London, The Strand, London WC2R 2LS, UK
e-mail: c.iliopoulos@kcl.ac.uk; solon.pissis@kcl.ac.uk

M.S. Rahman
AℓEDA Group, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka 1205, Bangladesh
e-mail: msrahman@cse.buet.ac.bd

M. Elloumi (ed.), *Algorithms for Next-Generation Sequencing Data*,
DOI 10.1007/978-3-319-59826-0_3

y when $y[i \dots i+n-1] = x$. We define the *i-th prefix* of x as the prefix ending at position i, $0 \leq i < n$, i.e., the prefix $x[0 \dots i]$. On the other hand, the *i-th suffix* is the suffix starting at position i, $0 \leq i < n$, i.e., the suffix $x[i \dots n-1]$.

A circular string can be viewed as a linear string where the left-most character is concatenated to the right-most one. Hence, a circular string of length m can be seen as m different linear strings, all of which are considered as being equivalent. Given a string x of length m, we denote by $x^i = x[i \dots m-1]x[0 \dots i-1]$, $0 < i < m$, the *i-th rotation* of x and $x^0 = x$.

We first consider the problem of finding occurrences of a string *pattern* x of length m with a circular structure in a string *text* t of length n with a linear structure. This is the problem of *circular string matching*. A naïve solution with *quadratic* in m time complexity consists in applying a classical algorithm for searching a finite set of strings after having built the dictionary of the m rotations of x [10]. This problem has also been considered in [25], where an $\mathcal{O}(n)$-time algorithm was presented. The approach presented in [25] consists in preprocessing x by constructing the suffix automaton of the string $x' = xx$, noting that every rotation of x is a factor of x'. Then, by feeding t into the automaton, the lengths of the longest factors of x' occurring in t can be found by the links followed in the automaton in time $\mathcal{O}(n)$. In [15], the authors presented an *average-case* time-optimal algorithm for circular string matching, by showing that the average-case lower time bound for *single* string matching of $\mathcal{O}(n \log_\sigma m/m)$ also holds for circular string matching. Recently, in [9], the authors presented two fast average-case algorithms based on word-level parallelism. The first algorithm requires average-case time $\mathcal{O}(n \log_\sigma m/w)$, where w is the number of bits in the computer word. The second one is based on a mixture of word-level parallelism and q-grams. The authors showed that with the addition of q-grams and by setting $q = \mathcal{O}(\log_\sigma m)$, an average-case optimal time of $\mathcal{O}(n \log_\sigma m/m)$ is achieved.

Given a set $\mathcal{D}$ of d pattern strings, the problem of dictionary matching is to index $\mathcal{D}$ such that for any online query of a text t, one can quickly find the occurrences of any pattern of $\mathcal{D}$ in t. This problem has been well-studied in the literature [1, 8], and an index taking optimal space and simultaneously supporting time-optimal queries is achieved [7, 19]. In some applications in computational molecular biology [5] and also in pattern recognition [28], we are interested in searching for not only the original patterns in $\mathcal{D}$ but also *all* of their rotations. This is the problem of *circular dictionary matching* [2, 20, 21].

The aim of constructing an index to provide efficient procedures for answering *online* text queries related to the content of a fixed dictionary of circular patterns was studied in [20]. The authors proposed a variant of the suffix tree (for an introduction to suffix trees, see [33]), called *circular suffix tree* and showed that it can be compressed into succinct space. With a tree structure augmented to a circular pattern matching index called *circular suffix array*, the circular suffix tree can be used to solve the circular dictionary matching problem efficiently. Assuming that the lengths of any two patterns in $\mathcal{D}$ is bounded by a constant, an online text query can be answered in time $\mathcal{O}((n + Occ) \log^{1+\varepsilon} n)$, for any fixed $\varepsilon > 0$, where Occ is the number of occurrences in the output. In [21], the authors proposed the first

algorithm for space-efficient construction of the circular suffix tree, which requires time $\mathcal{O}(M \log M)$ and $\mathcal{O}(M \log \sigma + d \log M)$ bits of working space, where M is the total length of the dictionary patterns. In [2], the authors proposed an algorithm to solve the circular dictionary matching problem by constructing an indexing data structure in time and space $\mathcal{O}(M)$ and answering a subsequent online text query in average-case time $\mathcal{O}(n)$, assuming that the shortest pattern is sufficiently long.

Analogously, the aim of constructing an index to provide efficient procedures for answering *online* circular pattern queries related to the content of a fixed text was studied in [24]. The authors proposed two indexing data structures. The first one can be constructed in time and space $\mathcal{O}(n \log^{1+\varepsilon} n)$, and an online pattern query can be answered in time $\mathcal{O}(m \log \log n + Occ)$. However, it involves the construction of two suffix trees as well as a complex range-search data structure. Hence, it is suspected that, despite a good theoretical time bound, the practical performance of this construction would not be very good both in terms of time and space. The second indexing data structure, on the other hand, uses the suffix array (for an introduction to suffix arrays, see [26]), which is more space-efficient than the suffix tree and does not require the range-search data structure. It is conceptually much simpler and can be built in time and space $\mathcal{O}(n)$; an online pattern query can then be answered in time $\mathcal{O}(m \log n + Occ)$.

In this chapter, we revisit the aforementioned problems for searching and indexing circular patterns; we formally define them as follows:

CIRCULARDICTIONARYMATCHING (CDM)
Input: a fixed set $\mathscr{D} = \{x_0, x_1 \ldots, x_{d-1}\}$ of patterns of total length M and, subsequently, an online query of a text t of length n, such that $n > |x_j|$, $0 \leq j < d$
Output: all factors u of t such that $u = x_j^i$, $0 \leq j < d$, $0 \leq i < |x_j|$

CIRCULARPATTERNINDEXING (CPI)
Input: a fixed text t of length n and, subsequently, an online query of a pattern x of length m, such that $n > m$
Output: all factors u of t such that $u = x^i$, $0 \leq i < m$

Notice that in the CDM and CPI problem settings, once the relevant indexing data structures are constructed, an unlimited number of online queries can be handled. We present details of the following solutions to these problems:

- An algorithm to solve the CDM problem by constructing an indexing data structure in time and space $\mathcal{O}(M)$ and answering a subsequent online text query in average-case time $\mathcal{O}(n)$, assuming that the shortest pattern in $\mathscr{D}$ is sufficiently long [2].
- An algorithm to solve the CPI problem by constructing an indexing data structure in time and space $\mathcal{O}(n)$ and answering a subsequent online pattern query in time $\mathcal{O}(m \log n + Occ)$ [24].

The rest of the chapter is organized as follows: In Sect. 3.2, we present a solution to the CDM problem; in Sect. 3.3, we present a solution to the CPI problem; finally, in the last section, we summarize the presented results and present some future proposals.

3.2 Circular Dictionary Matching

In this section, we focus on the implementation of an index to provide efficient procedures for answering online text queries related to the content of a fixed dictionary of circular patterns. This is the circular dictionary matching (CDM) problem defined in Sect. 3.1. In this setting, we will be given a fixed dictionary of circular patterns beforehand for preprocessing, and, subsequently, one or more online text queries may follow, one at a time. All text queries would be against the dictionary supplied to us for preprocessing.

Here, we describe an algorithm to solve the CDM problem by constructing an indexing data structure in time and space $\mathcal{O}(M)$ and answering a subsequent online text query in average-case time $\mathcal{O}(n)$, assuming that the shortest pattern in $\mathcal{D}$ is sufficiently long. We start by giving a brief outline of the *partitioning* technique in general and then show some properties of the version of the technique we use for our algorithms. We then describe average-case suboptimal algorithms for circular string matching and show how they can be easily generalized to solve the CDM problem efficiently.

3.2.1 Properties of the Partitioning Technique

The partitioning technique, introduced in [34], and in some sense earlier in [29], is an algorithm based on *filtering out* candidate positions that could never give a solution to speed up string matching algorithms. An important point to note about this technique is that it reduces the search space but does not, by design, verify potential occurrences. To create a string matching algorithm, filtering must be combined with some verification technique. The technique was initially proposed for approximate string matching, but here we show that this can also be used for circular string matching.

The idea behind the partitioning technique is to partition the given pattern in such a way that at least one of the fragments must occur exactly in any valid approximate occurrence of the pattern. It is then possible to search for these fragments exactly to give a set of *candidate* occurrences of the pattern. It is then left to the verification portion of the algorithm to check if these are valid approximate occurrences of the pattern. It has been experimentally shown that this approach yields very good practical performance on large-scale datasets [16], even if it is not theoretically optimal.

For circular string matching, for an efficient solution, we cannot simply apply well-known string matching algorithms, as we must also take into account the rotations of the pattern. We can, however, make use of the partitioning technique and, by choosing an appropriate number of fragments, ensure that at least one fragment must occur in any valid occurrence of a rotation. Lemma 1, together with the following fact, provides this number.

Fact 1 ([2]) *Let x be a string of length m. Any rotation of x is a factor of $x' = x[0 \dots m-1]x[0 \dots m-2]$, and any factor of length m of x' is a rotation of x.*

Lemma 1 ([2]) *Let x be a string of length m. If we partition $x' = x[0 \dots m-1]x[0 \dots m-2]$ into four fragments of length $\lfloor (2m-1)/4 \rfloor$ and $\lceil (2m-1)/4 \rceil$, at least one of the four fragments is a factor of any factor of length m of x'.*

3.2.2 Circular String Matching via Filtering

In this section, we consider the circular string matching problem.

CIRCULARSTRINGMATCHING (CSM)
Input: a pattern x of length m and a text t of length $n > m$
Output: all factors u of t such that $u = x^i$, $0 \leq i < m$

We present average-case suboptimal algorithms for circular string matching via filtering [2, 3]. They are based on the partitioning technique and a series of practical and well-established data structures.

3.2.2.1 Longest Common Extension

First, we describe how to compute the longest common extension, denoted by lce, of two suffixes of a string in constant time. In general, the *longest common extension* problem considers a string x and computes, for each pair (i, j), the length of the longest factor of x that starts at both i and j [23]. lce queries are an important part of the algorithms presented later on.

Let SA denote the array of positions of the lexicographically sorted suffixes of string x of length n, i.e., for all $1 \leq r < n$, we have $x[\mathsf{SA}[r-1] \dots n-1] < x[\mathsf{SA}[r] \dots n-1]$. The inverse iSA of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n$. Let $\mathsf{lcp}(r, s)$ denote the length of the longest common prefix of the strings $x[\mathsf{SA}[r] \dots n-1]$ and $x[\mathsf{SA}[s] \dots n-1]$ for all $0 \leq r, s < n$ and 0 otherwise. Let LCP denote the array defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1, r)$, for all $1 < r < n$, and $\mathsf{LCP}[0] = 0$. Given an array A of n objects taken from a well-ordered set, the range minimum query $\mathsf{RMQ}_\mathsf{A}(l, r) = \operatorname{argmin} \mathsf{A}[k]$, $0 \leq l \leq k \leq r < n$ returns the position of the minimal element in the specified subarray $\mathsf{A}[l \dots r]$. We perform the following linear-time and linear-space preprocessing:

- Compute arrays SA and iSA of x [27];
- Compute array LCP of x [13];
- Preprocess array LCP for range minimum queries; we denote this by $\mathsf{RMQ}_{\mathsf{LCP}}$ [14].

With the preprocessing complete, the lce of two suffixes of x starting at positions p and q can be computed in constant time in the following way [23]:

$$\mathsf{LCE}(x,p,q) = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{iSA}[p]+1, \mathsf{iSA}[q])]. \tag{3.1}$$

Example 1 Let the string $x =$ `abbababba`. The following table illustrates the arrays SA, iSA, and LCP for x.

i	0	1	2	3	4	5	6	7	8
x[i]	a	b	b	a	b	a	b	b	a
SA[i]	8	3	5	0	7	2	4	6	1
iSA[i]	3	8	5	1	6	2	7	4	0
LCP[i]	0	1	2	4	0	2	3	1	3

We have $\mathsf{LCE}(x,2,1) = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{iSA}[2]+1, \mathsf{iSA}[1])] = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(6,8)] = 1$, implying that the lce of `bbababba` and `bababba` is 1.

3.2.2.2 Algorithm CSMF

Given a pattern x of length m and a text t of length $n > m$, an outline of algorithm CSMF for solving the CSM problem is as follows:

1. Construct the string $x' = x[0 \ldots m-1]x[0 \ldots m-2]$ of length $2m-1$. By Fact 1, any rotation of x is a factor of x'.
2. The pattern x' is partitioned in four fragments of length $\lfloor (2m-1)/4 \rfloor$ and $\lceil (2m-1)/4 \rceil$. By Lemma 1, at least one of the four fragments is a factor of any rotation of x.
3. Match the four fragments against the text t using an Aho-Corasick automaton [11]. Let $\mathscr{L}$ be a list of size Occ of tuples, where $< p_{x'}, \ell, p_t > \in \mathscr{L}$ is a 3-tuple such that $0 \le p_{x'} < 2m-1$ is the position where the fragment occurs in x', ℓ is the length of the corresponding fragment, and $0 \le p_t < n$ is the position where the fragment occurs in t.
4. Compute SA, iSA, LCP, and $\mathsf{RMQ}_{\mathsf{LCP}}$ of $T = x't$. Compute SA, iSA, LCP, and $\mathsf{RMQ}_{\mathsf{LCP}}$ of $T_r = \mathsf{rev}(tx')$, that is the reverse string of tx'.
5. For each tuple $< p_{x'}, \ell, p_t > \in \mathscr{L}$, we try to extend to the right via computing

$$\mathscr{E}_r \leftarrow \mathsf{LCE}(t, p_{x'}+\ell, 2m-1+p_t+\ell);$$

in other words, we compute the length $\mathscr{E}_r$ of the longest common prefix of $x'[p_{x'}+\ell \ldots 2m-1]$ and $t[p_t+\ell \ldots n-1]$, both being suffixes of T. Similarly,

we try to extend to the left via computing $\mathscr{E}_l$ using lce queries on the suffixes of T_r.

6. For each $\mathscr{E}_l$ and $\mathscr{E}_r$ computed for tuple $< p_{x'}, \ell, p_t > \in \mathscr{L}$, we report all the valid starting positions in t by first checking if the total length $\mathscr{E}_l + \ell + \mathscr{E}_r \geq m$; that is the length of the full extension of the fragment is greater than or equal to m, matching at least one rotation of x. If that is the case, then we report positions

$$\max\{p_t - \mathscr{E}_\ell, p_t + \ell - m\}, \ldots, \min\{p_t + \ell - m + \mathscr{E}_r, p_t\}.$$

Example 2 Let us consider the pattern $x = \texttt{GGGTCTA}$ of length $m = 7$ and the text

$$t = \texttt{GATACGATACCTAGGGTGATAGAATAG}.$$

Then, $x' = \texttt{GGGTCTAGGGTCT}$ (Step 1). x' is partitioned in GGGT, CTA, GGG, and TCT (Step 2). Consider $< 4, 3, 10 > \in \mathscr{L}$, that is, fragment $x'[4 \ldots 6] = \texttt{CTA}$, of length $\ell = 3$, occurs at starting position $p_t = 10$ in t (Step 3). Then,

$$T = \texttt{GGGTCTAGGGTCTGATACGATACCTAGGGTGATAGAATAG}$$

and

$$T_r = \texttt{TCTGGGATCTGGGGATAAGATAGTGGGATCCATAGCATAG}$$

(Step 4). Extending to the left gives $\mathscr{E}_l = 0$, since $T_r[9] \neq T_r[30]$, and extending to the right gives $\mathscr{E}_r = 4$, since $T[7 \ldots 10] = T[26 \ldots 29]$ and $T[11] \neq T[30]$ (Step 5). We check that $\mathscr{E}_l + \ell + \mathscr{E}_r = 7 = m$, and therefore, we report position 10 (Step 6):

$$p_t - \mathscr{E}_\ell = 10 - 0 = 10, \ldots, p_t + \ell - m + \mathscr{E}_r = 10 + 3 - 7 + 4 = 10;$$

that is, $x^4 = \texttt{CTAGGGT}$ occurs at starting position 10 in t.

Theorem 1 ([3]) *Given a pattern x of length m and a text t of length $n > m$, algorithm CSMF requires average-case time $\mathscr{O}(n)$ and space $\mathscr{O}(n)$ to solve the CSM problem.*

3.2.2.3 Algorithm CSMF-Simple

In this section, we present algorithm CSMF-Simple [2], a more space-efficient version of algorithm CSMF. Algorithm CSMF-Simple is very similar to algorithm CSMF. The only differences are:

- Algorithm CSMF-Simple does not perform Step 4 of algorithm CSMF.
- For each tuple $< p_{x'}, \ell, p_t > \in \mathscr{L}$, Step 5 of algorithm CSMF is performed without the use of the pre-computed indexes. In other words, we compute $\mathscr{E}_r$ and

$\mathscr{E}_\ell$ by simply performing letter-to-letter comparisons and counting the number of mismatches that occurred. The extension stops right before the first mismatch.

Fact 2 ([2]) *The expected number of letter comparisons required for each extension in algorithm CSMF-Simple is less than* 3.

Theorem 2 ([2]) *Given a pattern x of length m and a text t of length $n > m$, algorithm CSMF-Simple requires average-case time $\mathscr{O}(n)$ and space $\mathscr{O}(m)$ to solve the CSM problem.*

3.2.3 Circular Dictionary Matching via Filtering

In this section, we give a generalization of our algorithms for circular string matching to solve the problem of circular dictionary matching. We denote this algorithm by CDMF. Algorithm CDMF follows the same approach as before but with a few key differences. In circular dictionary matching, we are given a set $\mathscr{D} = \{x_0, x_1, \ldots, x_{d-1}\}$ of patterns of total length M and we must find all occurrences of the patterns in $\mathscr{D}$ or any of their rotations. To modify algorithm CSMF to solve this problem, we perform Steps 1 and 2 for every pattern in $\mathscr{D}$, constructing the strings $x'_0, x'_1, \ldots, x'_{d-1}$ and breaking them each into four fragments in the same way specified in Lemma 1. From this point, the algorithm remains largely the same (Steps 3–4); we build the automaton for the fragments from every pattern and then proceed in the same way as algorithm CSMF. The only extra consideration is that we must be able to identify, for every fragment, the pattern from which it was extracted. To do this, we alter the definition of $\mathscr{L}$ such that it now consists of tuples of the form $< p_{x'_j}, \ell, j, p_t >$, where j identifies the pattern from where the fragment was extracted, $p_{x'_j}$ and ℓ are defined identically with respect to the pattern x_j, and p_t remains the same. This then allows us to identify the pattern for which we must perform verification (Steps 5–6) if a fragment is matched. The verification steps are then the same as in algorithm CSMF with the respective pattern.

In a similar way as in algorithm CSMF-Simple, we can apply Fact 2 to obtain algorithm CDMF-Simple and achieve the following result:

Theorem 3 ([2]) *Given a set $\mathscr{D} = \{x_0, x_1, \ldots, x_{d-1}\}$ of patterns of total length M drawn from alphabet Σ, $\sigma = |\Sigma|$, and a text t of length $n > |x_j|$, where $0 \leq j < d$, drawn from Σ, algorithm CDMF-Simple requires average-case time $\mathscr{O}((1 + \frac{d|x_{\max}|}{\sigma^{\frac{2|x_{\min}|-1}{4}}})n + M)$ and space $\mathscr{O}(M)$ to solve the CDM problem, where $x_{\min}$ and $x_{\max}$ are the minimum- and maximum-length patterns in $\mathscr{D}$, respectively.*
Algorithm CDMF achieves average-case time $\mathscr{O}(n + M)$ *if and only if*

$$\frac{4d|x_{\max}|}{\sigma^{\frac{2|x_{\min}|-1}{4}}} n \leq cn$$

for some fixed constant c. So we have

$$\frac{4d|x_{\max}|}{\sigma^{\frac{2|x_{\min}|-1}{4}}} \leq c$$

$$\log_\sigma \left(\frac{4d|x_{\max}|}{c} \right) \leq \frac{2|x_{\min}|-1}{4}$$

$$4(\log_\sigma 4 + \log_\sigma d + \log_\sigma |x_{\max}| - \log_\sigma c) \leq 2|x_{\min}| - 1$$

Rearranging and setting c such that $\log_\sigma c \geq 1/4 + \log_\sigma 4$ gives a sufficient condition for our algorithm to achieve average-case time $\mathcal{O}(n+M)$:

$$|x_{\min}| \geq 2(\log_\sigma d + \log_\sigma |x_{\max}|).$$

Corollary 1 ([2]) *Given a set $\mathcal{D} = \{x_0, x_1, \ldots, x_{d-1}\}$ of patterns of total length M drawn from alphabet Σ, $\sigma = |\Sigma|$ and a text t of length $n > |x_j|$, where $0 \leq j < d$, drawn from Σ, algorithm CDMF-Simple solves the CDM problem in average-case time $\mathcal{O}(n+M)$ if and only if $|x_{\min}| \geq 2(\log_\sigma d + \log_\sigma |x_{\max}|)$, where $x_{\min}$ and $x_{\max}$ are the minimum- and maximum-length patterns in $\mathcal{D}$, respectively.*

3.2.4 Key Results

We described algorithm CDMF-Simple to solve the CDM problem by constructing an indexing data structure—the Aho-Corasick automaton of the patterns' fragments—in time and space $\mathcal{O}(M)$ and answering a subsequent online text query in average-case time $\mathcal{O}(n)$, assuming that the shortest pattern in $\mathcal{D}$ is sufficiently long.

3.3 Circular Pattern Indexing

In this section, we focus on the implementation of an index to provide efficient procedures for answering online circular patterns queries related to the content of a fixed text. This is the *Circular Pattern Indexing* (CPI) problem defined in Sect. 3.1. In this setting, we will be given a fixed text beforehand for preprocessing, and, subsequently, one or more online pattern queries may follow, one at a time. All pattern queries would be against the text supplied to us for preprocessing.

Here, we describe an algorithm to solve the CPI problem by constructing an indexing data structure in time and space $\mathcal{O}(n)$ and answering a subsequent online pattern query in time $\mathcal{O}(m \log n + Occ)$ [24]. For the rest of this section, we make use of the following notation. For rotation x^i of string x, we denote $x^i = x^{i,f}x^{i,\ell}$ where $x^{i,f} = x[i \ldots m-1]$ and $x^{i,\ell} = x[0 \ldots i-1]$.

3.3.1 The CPI-II Data Structure

In this section, we discuss a data structure called CPI-II. CPI-II and another data structure, called CPI-I, were presented in [24] to solve the CPI problem. CPI-II consists of the suffix array SA and the inverse suffix array iSA of text t. Both SA and iSA of t of length n can be constructed in time and space $\mathcal{O}(n)$ [27]. To handle the subsequent queries, CPI-II uses the following well-known results from [18] and [22]. The result of a query for a pattern x on the suffix array SA of t is given as a pair (s, e) representing an interval $[s \dots e]$, such that the output set is $\{\mathsf{SA}[s], \mathsf{SA}[s+1], \dots, \mathsf{SA}[e]\}$. In this case, the interval $[s \dots e]$ is denoted by Int^t_x.

Lemma 2 ([18]) *Given a text t of length n, the suffix array of t, and the interval $[s \dots e] = Int^t_x$ for a pattern x, for any letter α, the interval $[s' \dots e'] = Int^t_{x\alpha}$ can be computed in time $\mathcal{O}(\log n)$.*

Lemma 3 ([22]) *Given a text t of length n, the suffix array of t, the inverse suffix array of t, interval $[s' \dots e'] = Int^t_{x'}$ for a pattern x', and interval $[s'' \dots e''] = Int^t_{x''}$ for a pattern x'', the interval $[s \dots e] = Int^t_{x'x''}$ can be computed in time $\mathcal{O}(\log n)$.*

Remark 1 At this point, a brief discussion on how $\mathrm{Int}^t_{x'}$ is combined with $\mathrm{Int}^t_{x''}$ to get $\mathrm{Int}^t_{x'x''}$ is in order. We need to find the interval $[s \dots e]$. The idea here is to find the smallest s and the largest e such that both the suffixes $t[\mathsf{SA}[s] \dots n-1]$ and $t[\mathsf{SA}[e] \dots n-1]$ have $x'x''$ as their prefixes. Therefore, we must have that $[s \dots e]$ is a subinterval of $[s' \dots e']$. Now, suppose that $|x'| = m'$. By definition, the lexicographic order of the suffixes $t[\mathsf{SA}[s'] \dots n-1], t[\mathsf{SA}[s'+1] \dots n-1], \dots, t[\mathsf{SA}[e'] \dots n-1]$ is increasing. Then, since they share the same prefix, x', the lexicographic order of the suffixes $t[\mathsf{SA}[s'] + m' \dots n-1], t[\mathsf{SA}[s'+1] + m' \dots n-1], \dots, t[\mathsf{SA}[e'] + m' \dots n-1]$ is increasing as well. Therefore, we must have that $\mathsf{iSA}[\mathsf{SA}[s'] + m'] < \mathsf{iSA}[\mathsf{SA}[s'+1] + m'] < \cdots < \mathsf{iSA}[\mathsf{SA}[e'] + m']$. Finally, we just need to find the smallest s such that $s' \leq \mathsf{iSA}[\mathsf{SA}[s] + m'] \leq e'$ and the largest e such that $s'' \leq \mathsf{iSA}[\mathsf{SA}[e] + m'] \leq e''$.

Lemma 4 ([22]) *Given a text t of length n, the suffix array of t, the inverse suffix array of t, interval $[s \dots e] = Int^t_x$ for a pattern x, and an array* Count *such that* Count$[\alpha]$ *stores the total number of occurrences of all $\alpha' \leq \alpha$, where '$\leq$' implies lexicographically smaller or equal, for any letter α, the interval $[s' \dots e'] = Int^t_{\alpha x}$ can be computed in time $\mathcal{O}(\log n)$.*

CPI-II handles a query as follows. It first computes the intervals for all the prefixes and suffixes of x and stores them in the arrays $\mathsf{Pref}[0 \dots m-1]$ and $\mathsf{Suf}[0 \dots m-1]$, respectively. To compute the intervals for prefixes, Lemma 2 is used as follows. CPI-II first computes the interval for the letter $x[0]$ and stores the interval in $\mathsf{Pref}[0]$. This can be done in time $\mathcal{O}(\log n)$. Next, it computes the interval for $x[0 \dots 1]$ using $\mathsf{Pref}[0]$ in time $\mathcal{O}(\log n)$ by Lemma 2. Then, it computes the interval for $x[0 \dots 2]$ using $\mathsf{Pref}[1]$ and so on. So, in this way, the array $\mathsf{Pref}[0 \dots m-1]$ can be computed in time $\mathcal{O}(m \log n)$. Similarly, CPI-II computes the intervals for suffixes in $\mathsf{Suf}[0 \dots m-1]$. Using Lemma 4, this can be done in time $\mathcal{O}(m \log n)$

as well. Then CPI-II uses Lemma 3 to combine the intervals of $x^{j,f}$ and $x^{j,\ell}$ for $0 \leq j \leq m-1$ from the corresponding indexes of the Pref and Suf arrays. Thus, all the intervals for all the rotations of x are computed. Finally, CPI-II carefully reports all the occurrences in this set of intervals as follows. To avoid reporting a particular occurrence more than once, it sorts the m intervals according to the start of the interval requiring time $\mathcal{O}(m \log m)$, and then it does a linear traversal on the two ends of the sorted intervals to get an equivalent set of *disjoint* intervals. Clearly, CPI-II data structure can be constructed in time and space $\mathcal{O}(n)$ and can answer the relevant queries in time $\mathcal{O}(m \log n + Occ)$ per query.

Further improvement on the above running time seems possible. In fact, we can improve both from the time and space point of view, albeit not asymptotically, if we use the FM-index [12]. Using the *backward-search technique*, we can compute array Suf in time $\mathcal{O}(m)$. However, any analogous result for Lemma 2 still eludes us. We are also unable to improve the time requirement for Lemma 3. Notably, some results on bidirectional search [30] may turn out to be useful in this regard which seems to be a good candidate for future investigations.

3.3.2 A Folklore Indexing Data Structure

Before concluding this section, we briefly shed some light on another sort of folklore indexing data structure to solve the CPI problem. The data structure is based on the suffix tree and Weiner's algorithm for its construction [33]. Suppose we have computed the suffix tree ST_t of $t\$$, where $\$ \notin \Sigma$ using Weiner's algorithm. Then an online pattern query can be answered as follows:

1. Let $x' = xx\#$, where $\# \notin \Sigma$ and $\# \neq \$$. Set $\ell = |x'|$.
2. Let $\alpha = x'[\ell - 1]$. Extend ST_t such that it becomes the suffix tree of $t' = \alpha t\$$. While extending, keep track of whether the new suffix diverges from a node v with $depth(v) \geq m$ of the original tree ST_t. Let us call these nodes *output nodes*.
3. Set $\ell = \ell - 1$. If $\ell > 0$, then go to Step 2.
4. Output the starting positions of the suffixes corresponding to t from the output nodes.
5. Undo the steps so that we again get ST_t and become ready for any further query.

The correctness of the above approach heavily depends on how Weiner's algorithm constructs the suffix tree. We are not going to provide all the details here, but very briefly, Weiner's algorithm first handles the suffix $t[n-1 \ldots n-1]$, followed by $t[n-2 \ldots n-1]$, and so on. This is why feeding x' to ST_t (cf. the loop comprising Steps 2 and 3 above) works perfectly. At first glance, it seems that the above data structure would give us a time-optimal query of $\mathcal{O}(m + Occ)$ with a data structure construction time of $\mathcal{O}(n)$. However, there is a catch. The linear-time construction algorithm of Weiner (and in fact all other linear-time constructions, e.g., [31]) depends on an amortized analysis. Hence, while we can certainly say that

the construction of the suffix tree for the string $xx\#t\$$ can be done in time $\mathcal{O}(m+n)$, given the suffix tree for $t\$$, we cannot always claim that extending it for $xx\#t\$$ can be done in time $\mathcal{O}(m)$.

3.3.3 Key Results

In Sect. 3.3.1, we described an algorithm to solve the CPI problem by constructing an indexing data structure—the CPI-II data structure—in time and space $\mathcal{O}(n)$ and answering a subsequent online pattern query in time $\mathcal{O}(m \log n + Occ)$.

3.4 Final Remarks and Outlook

In this chapter, we described algorithms and data structures for searching and indexing patterns with a circular structure. We considered two different settings for this task: in the first one, we are given a fixed set of patterns, and we must construct a data structure to answer subsequent online text queries; in the second, we are given a fixed text, and we must construct a data structure to answer subsequent online pattern queries. This type of task finds applications in computational molecular biology and in pattern recognition.

For the first setting, we described an algorithm to solve the problem by constructing an indexing data structure in time and space $\mathcal{O}(M)$ and answering a subsequent online text query in average-case time $\mathcal{O}(n)$, assuming that the shortest pattern is sufficiently long. For the second setting, we described an algorithm to solve the problem by constructing an indexing data structure in time and space $\mathcal{O}(n)$ and answering a subsequent online pattern query in time $\mathcal{O}(m \log n + Occ)$.

Despite both theoretical and practical motivations for searching and indexing circular patterns, we noticed a lack of significant research effort on this particular topic, especially from the perspective of computational molecular biology. Sequence comparison algorithms require an *implicit assumption* on the input data: the left- and right-most positions for each sequence are relevant. However, this is *not* the case for circular structures. To this end, we conclude this chapter posing an exciting question that could alter the perspective of current knowledge and state of the art for sequence comparison:

Would dropping this implicit assumption yield more significant alignments and thereby new biological knowledge, for instance in phylogenetic analyses, for organisms with such structure?

We believe that the circular string matching paradigm could bring forth a complete new era in this context.

Acknowledgements Part of this research has been supported by an INSPIRE Strategic Partnership Award, administered by the British Council, Bangladesh, for the project titled "Advances in Algorithms for Next Generation Biological Sequences."

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Athar, T., Barton, C., Bland, W., Gao, J., Iliopoulos, C.S., Liu, C., Pissis, S.P.: Fast circular dictionary-matching algorithm. Math. Struct. Comput. Sci. **FirstView**, 1–14 (2015)
3. Barton, C., Iliopoulos, C.S., Pissis, S.P.: Circular string matching revisited. In: Proceedings of the Fourteenth Italian Conference on Theoretical Computer Science (ICTCS 2013), pp. 200–205 (2013)
4. Barton, C., Iliopoulos, C.S., Pissis, S.P.: Fast algorithms for approximate circular string matching. Algorithms Mol. Biol. **9**(9) (2014)
5. Barton, C., Iliopoulos, C.S., Kundu, R., Pissis, S.P., Retha, A., Vayani, F.: Accurate and efficient methods to improve multiple circular sequence alignment. In: Bampis, E. (ed.) Experimental Algorithms. Lecture Notes in Computer Science, vol. 9125, pp. 247–258. Springer International Publishing, Cham (2015)
6. Barton, C., Iliopoulos, C.S., Pissis, S.P.: Average-case optimal approximate circular string matching. In: Dediu, A.H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) Language and Automata Theory and Applications. Lecture Notes in Computer Science, vol. 8977, pp. 85–96. Springer, Berlin/Heidelberg (2015)
7. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: Proceedings of the 21st Annual Conference on Combinatorial Pattern Matching, CPM'10, pp. 88–100. Springer, Berlin/Heidelberg (2010)
8. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Trans. Algorithms **3**(2) (2007)
9. Chen, K.-H., Huang, G.-S., Lee, R.C.-T.: Bit-parallel algorithms for exact circular string matching. Comput. J **57**(5), 731–743 (2014)
10. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, New York (2007)
11. Dori, S., Landau, G.M.: Construction of Aho Corasick automaton in linear time for integer alphabets. Inf. Process. Lett. **98**(2), 66–72 (2006)
12. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings of the FOCS, pp. 390–398. IEEE Computer Society, Los Alamitos, CA (2000)
13. Fischer, J.: Inducing the LCP-array. In: Dehne, F., Iacono, J., Sack, J.R. (eds.) Algorithms and Data Structures. Lecture Notes in Computer Science, vol. 6844, pp. 374–385. Springer, Berlin/Heidelberg (2011)
14. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
15. Fredriksson, K., Grabowski, S.: Average-optimal string matching. J. Discrete Algorithms **7**(4), 579–594 (2009)
16. Frousios, K., Iliopoulos, C.S., Mouchard, L., Pissis, S.P., Tischler, G.: REAL: an efficient REad ALigner for next generation sequencing reads. In: Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology, BCB '10, pp. 154–159. ACM, New York (2010)
17. Grossi, R., Iliopoulos, C.S., Mercas, R., Pisanti, N., Pissis, S.P., Retha, A., Vayani, F.: Circular sequence comparison with q-grams. In: Pop, M., Touzet, H. (eds.) Proceedings of Algorithms in Bioinformatics - 15th International Workshop, WABI 2015, Atlanta, GA, Sept 10–12, 2015. Lecture Notes in Computer Science, vol. 9289, pp. 203–216. Springer, Berlin (2015)

18. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
19. Hon, W.K., Ku, T.H., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster compressed dictionary matching. In: Proceedings of the 17th International Conference on String Processing and Information Retrieval, SPIRE'10, pp. 191–200. Springer, Berlin/Heidelberg (2010)
20. Hon, W.K., Lu, C.H., Shah, R., Thankachan, S.V.: Succinct indexes for circular patterns. In: Proceedings of the 22nd International Conference on Algorithms and Computation, ISAAC'11, pp. 673–682. Springer, Berlin/Heidelberg (2011)
21. Hon, W.K., Ku, T.H., Shah, R., Thankachan, S.: Space-efficient construction algorithm for the circular suffix tree. In: Fischer, J., Sanders, P. (eds.) Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 7922, pp. 142–152. Springer, Berlin/Heidelberg (2013)
22. Huynh, T.N.D., Hon, W.K., Lam, T.W., Sung, W.K.: Approximate string matching using compressed suffix arrays. Theor. Comput. Sci. **352**(1–3), 240–249 (2006)
23. Ilie, L., Navarro, G., Tinta, L.: The longest common extension problem revisited and applications to approximate string searching. J. Discrete Algorithms **8**(4), 418–428 (2010)
24. Iliopoulos, C.S., Rahman, M.S.: Indexing circular patterns. In: Proceedings of the 2nd International Conference on Algorithms and Computation, WALCOM'08, pp. 46–57. Springer, Berlin/Heidelberg (2008)
25. Lothaire, M. (ed.): Applied Combinatorics on Words. Cambridge University Press, New York (2005)
26. Manber, U., Myers, E.W.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993)
27. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proceedings of the 2009 Data Compression Conference, DCC '09, pp. 193–202. IEEE Computer Society, Washington, DC (2009)
28. Palazón-González, V., Marzal, A.: Speeding up the cyclic edit distance using LAESA with early abandon. Pattern Recogn. Lett. (2015). http://dx.doi.org/10.1016/j.patrec.2015.04.013
29. Rivest, R.L.: Partial-match retrieval algorithms. SIAM J. Comput. **5**(1), 19–50 (1976). doi:10.1137/0205003
30. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional search in a string with wavelet trees and bidirectional matching statistics. Inf. Comput. **213**, 13–22 (2012)
31. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)
32. Uliel, S., Fliess, A., Unger, R.: Naturally occurring circular permutations in proteins. Protein Eng. **14**(8), 533–542 (2001)
33. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973), pp. 1–11. IEEE Computer Society, Washington, DC (1973)
34. Wu, S., Manber, U.: Fast text searching: allowing errors. Commun. ACM **35**(10), 83–91 (1992)