King's College London

King's Research Portal

DOI:
10.1017/S0960129515000134

Document Version
Publisher's PDF, also known as Version of record

Link to publication record in King's Research Portal

Citation for published version (APA):
Athar, T., Barton, C., Bland, W., Gao, J., Iliopoulos, C. S., Liu, C., & Pissis, S. P. (2017). Fast circular dictionary-matching algorithm. *MATHEMATICAL STRUCTURES IN COMPUTER SCIENCE*, *27*(2), 143-156. https://doi.org/10.1017/S0960129515000134

Citing this paper
Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

General rights
Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

•Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
•You may not further distribute the material or use it for any profit-making activity or commercial gain
•You may freely distribute the URL identifying the publication in the Research Portal

Take down policy
If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 25. Oct. 2020

Math. Struct. in Comp. Science (2017), *vol.* 27, *pp.* 143–156. © Cambridge University Press 2015.
This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/3.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.
doi:10.1017/S0960129515000134 First published online 11 May 2015

Fast circular dictionary-matching algorithm

TANVER ATHAR†, CARL BARTON†, WIDMER BLAND‡, JIA GAO†, COSTAS S. ILIOPOULOS†, CHANG LIU† and SOLON P. PISSIS†

†*Department of Informatics, King's College London, London, UK*
‡*Department of Computing and Software, McMaster University, Hamilton, Canada*

Received 9 February 2014; revised 5 March 2015

Circular string matching is a problem which naturally arises in many contexts. It consists in finding all occurrences of the rotations of a pattern of length m in a text of length n. There exist optimal worst- and average-case algorithms for circular string matching. Here, we present a suboptimal average-case algorithm for circular string matching requiring time $\mathcal{O}(n)$ and space $\mathcal{O}(m)$. The importance of our contribution is underlined by the fact that the proposed algorithm can be easily adapted to deal with circular dictionary matching. In particular, we show how the circular dictionary-matching problem can be solved in average-case time $\mathcal{O}(n + M)$ and space $\mathcal{O}(M)$, where M is the total length of the dictionary patterns, assuming that the shortest pattern is sufficiently long. Moreover, the presented average-case algorithms and other worst-case approaches were also implemented. Experimental results, using real and synthetic data, demonstrate that the implementation of the presented algorithms can accelerate the computations by more than a factor of two compared to the corresponding implementation of other approaches.

1. Introduction

In order to provide an overview of our results and algorithms, we begin with a few definitions generally following (Smyth 2003). We think of a string x of length n as an array $x[0 \,..\, n-1]$, where every $x[i]$, $0 \leqslant i < n$, is a letter drawn from some fixed alphabet Σ of size $\sigma = |\Sigma|$. The empty string of length 0 is denoted by ε. A string x is a factor of a string y if there exist two strings u and v, such that $y = uxv$. Let the strings x, y, 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. Let x be a non-empty string of length n and y be a string. We say that there exists an occurrence of x in y, or, more simply, that x occurs in y, when x is a factor of y. Every occurrence of x can be characterized by a position in y. Thus we say that x occurs at the starting position i in y when $y[i \,..\, i+n-1] = x$. We define the ith prefix to be the prefix ending at position i i.e. $x[0 \,..\, i]$, $0 \leqslant i < n$. On the other hand, the ith suffix is the suffix starting at position i i.e. $x[i \,..\, n-1]$, $0 \leqslant i < n$.

A circular string of length m can be viewed as a traditional linear string which has the left- and right-most symbols wrapped around and stuck together in some way. Under this notion, the same circular string can be seen as m different linear strings, which would all be considered equivalent. Given a string x of length m, we denote by

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

$x^i = x[i\,..\,m-1]x[0\,..\,i-1]$, $0 < i < m$, the ith rotation of x and $x^0 = x$. Consider, for instance, the string $x = x^0 =$ `abababbc`; this string has the following rotations: $x^1 =$ `bababbca`, $x^2 =$ `ababbcab`, $x^3 =$ `babbcaba`, $x^4 =$ `abbcabab`, $x^5 =$ `bbcababa`, $x^6 =$ `bcababab`, $x^7 =$ `cabababb`.

Here, we consider the problem of finding occurrences of a pattern x of length m with circular structure in a text t of length n with linear structure. This is the problem of *circular string matching*; it has been considered in Lothaire (2005), where an $\mathcal{O}(n)$-time algorithm was presented. A naive solution with quadratic complexity consists in applying a classical algorithm for searching a finite set of strings after having built the trie of rotations of x. The approach presented in Lothaire (2005) consists in preprocessing x by constructing a suffix automaton of the string xx, by noting that every rotation of x is a factor of xx. Then, by feeding t into the automaton, the lengths of the longest factors of xx occurring in t can be found by the links followed in the automaton in time $\mathcal{O}(n)$. In Fredriksson and Grabowski (2009), the authors presented an optimal average-case algorithm for circular string matching, by also showing that the average-case lower bound for single string matching of $O(n \log_\sigma m/m)$ also holds for circular string matching. Very recently, in Chen *et al.* (2013), 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 optimal average-case time of $\mathcal{O}(n \log_\sigma m/m)$ is achieved.

Given a set $\mathcal{D}$ of d pattern strings, the dictionary-matching problem is to index $\mathcal{D}$ such that for any online query 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 (Aho and Corasick 1975; Chan *et al.* 2007), and an index taking optimal space and simultaneously supporting time-optimal queries is achieved (Belazzougui 2010; Hon *et al.* 2010). In some applications in computational molecular biology, such as, for instance, pattern matching of a collection of viral sequences, 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*.

A variant of this problem for an offline query text was first discussed in Iliopoulos and Rahman (2008). The authors proposed two index data structures, namely CPI-I and CPI-II. CPI-I can be constructed in time and space $\mathcal{O}(n \log^{1+\varepsilon} n)$, and an online pattern query can be answered in time $\mathcal{O}(m \log\log n + Occ)$, where Occ is the number of occurrences. However, CPI-I involves constructing two suffix trees (Weiner 1973) as well as a complex range-search data structure. Hence, it is suspected that, despite a good theoretical time bound, the practical performance of CPI-I would not be very good both in terms of time and space. CPI-II on the other hand uses a suffix array (Manber and Myers 1993), which is much more space-efficient than a suffix tree and does not require the range-search data structure. CPI-II is conceptually much simpler and can be built in time $\mathcal{O}(n)$ and requires $\mathcal{O}(n \log n)$ bits of space; an online pattern query can then be answered in time $\mathcal{O}(m \log n + Occ)$.

The problem for any online query text was studied in Hon *et al.* (2011). The authors proposed a variant of suffix tree, called circular suffix tree and showed that it can be

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

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. Very recently, in Hon *et al.* (2013) the authors proposed the first algorithm for the efficient construction of the circular suffix tree, which requires time $\mathcal{O}(M \log M)$ and $O(M \log \sigma + d \log M)$ bits of working space, where M is the total length of the dictionary patterns.

In this article, we revisit the following two problems in the *average-case* setting.

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

CIRCULARDICTIONARYMATCHING
Input: a set $\mathcal{D} = \{x_0, x_1 \ldots, x_{d-1}\}$ of patterns of total length M and a text t of length n, such that $n > |x_j|$, $0 \leqslant j < d$
Output: all factors u of t such that $u = x_j^i$, $0 \leqslant j < d$, $0 \leqslant i < |x_j|$

1.1. *Our contribution*

We present a new suboptimal average-case algorithm for circular string matching requiring time $\mathcal{O}(n)$ and space $\mathcal{O}(m)$. We show how it can be extended to solve the circular dictionary-matching problem in average-case time $\mathcal{O}(n + M)$ and space $\mathcal{O}(M)$, assuming that the shortest pattern is sufficiently long. Furthermore, we implement the presented average-case algorithms and the index data structure CPI-II presented in Iliopoulos and Rahman (2008) for an offline query text. Experimental results, using real and synthetic data, demonstrate that the implementation of the algorithms proposed here can accelerate the computations by more than a factor of two compared to the corresponding implementation of CPI-II. A preliminary version of this work appeared in Barton *et al.* (2013).

2. Properties of the partitioning technique

In this section, we give 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. The partitioning technique, introduced in Wu and Manber (1992), and in some sense earlier in Rivest (1976), 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 idea behind the partitioning technique was initially proposed for approximate string matching, but here we show that this can also be used for exact 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

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

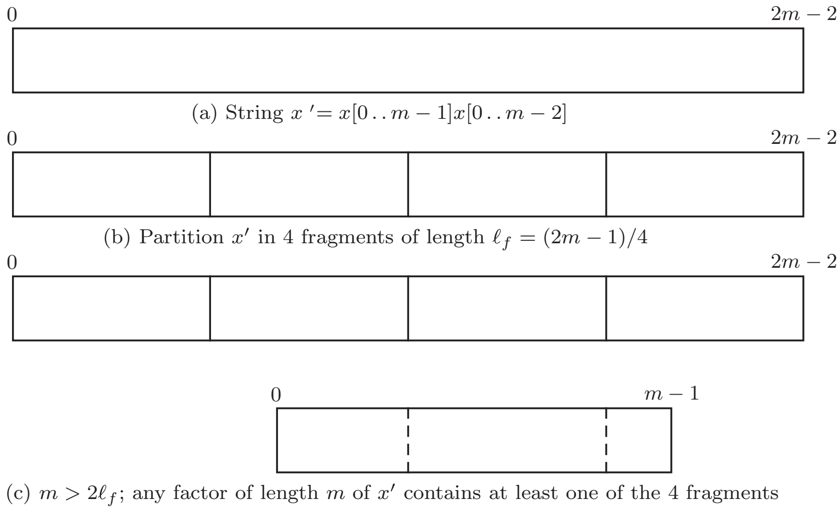

Fig. 1. Illustration of Lemma 2.1. (a) String $x' = x[0\,..\,m-1]x[0\,..\,m-2]$. (b) Partition x' in four fragments of length $\ell_f = (2m-1)/4$. (c) $m > 2\ell_f$; any factor of length m of x' contains at least one of the four fragments.

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 (Frousios *et al.* 2010), even if it is not theoretically optimal.

For exact circular string matching, for an efficient solution, we cannot simply apply well-known exact 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 exact occurrence of a rotation. Lemma 2.1 together with the following fact provide this number.

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

Lemma 2.1. Let x be a string of length m. If we partition $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ in 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'.

Proof. Let ℓ_f denote the length of the fragment. If we partition x' in at least four fragments of length $\lfloor(2m-1)/4\rfloor$ and $\lceil(2m-1)/4\rceil$, we have that

$$\ell_f \leqslant (2m-1)/4,$$

which gives $2m > 4\ell_f$ and $m > 2\ell_f$. Therefore, any factor of length m of x', and, by Fact 1, any rotation of x, must contain at least one of the fragments. For a graphical illustration of this proof inspect Figure 1. □

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

3. Circular string matching via filtering

In this section, we present CSMF, a new suboptimal average-case algorithm for exact circular string matching via filtering. It is based on the partitioning technique and a series of practical and well-established data structures.

3.1. *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. lce queries are an important part of the algorithms presented later on.

Let SA denote the array of positions of the sorted suffixes of string x of length n, i.e. for all $1 \leqslant r < n$, we have $x[\mathsf{SA}[r-1]\,..\,n-1] < x[\mathsf{SA}[r]\,..\,n-1]$. The inverse iSA of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leqslant r < n$. Let $\mathsf{lcp}(r,s)$ denote the length of the longest common prefix of the strings $x[\mathsf{SA}[r]\,..\,n-1]$ and $x[\mathsf{SA}[s]\,..\,n-1]$, for all $0 \leqslant r,s < n$, and 0 otherwise. Let LCP denote the array defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1,r)$, for all $1 < r < n$, and $\mathsf{LCP}[0] = 0$. We perform the following linear-time and linear-space preprocessing:

— Compute arrays SA and iSA of x (Nong *et al.* 2009).
— Compute array LCP of x (Fischer 2011).
— Preprocess array LCP for range minimum queries, we denote this by $\mathsf{RMQ}_{\mathsf{LCP}}$ (Fischer and Heun 2011).

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 (Ilie *et al.* 2010):

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

Example 3.1. Let the string $x = \texttt{abbababba}$. 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 $\mathsf{LCE}(x,1,2) = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(\mathsf{iSA}[2]+1, \mathsf{iSA}[1])] = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(6,8)] = 1$, implying that the lce of `bbababba` and `bababba` is 1.

3.2. *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 CircularStringMatching problem is as follows.

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

1. Construct the string $x' = x[0\,..\,m-1]x[0\,..\,m-2]$ 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 2.1, 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 (Dori and Landau 2006). Let $\mathcal{L}$ be a list of size Occ of tuples, where $< p_{x'}, \ell, p_t > \in \mathcal{L}$ is a three-tuple such that $0 \leqslant p_{x'} < 2m-1$ is the position where the fragment occurs in x', ℓ is the length of the corresponding fragment, and $0 \leqslant p_t < n$ 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 $< p_{x'}, \ell, p_t > \in \mathcal{L}$, we try to extend to the right via computing

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

   in other words, we compute the length $\mathcal{E}_r$ of the longest common prefix of $x'[p_{x'} + \ell\,..\,2m-1]$ and $t[p_t+\ell\,..\,n-1]$, both being suffixes of T. Similarly, we try to extend to the left via computing $\mathcal{E}_l$ using lce queries on the suffixes of T_r.
6. For each $\mathcal{E}_l, \mathcal{E}_r$ computed for tuple $< p_{x'}, \ell, p_t > \in \mathcal{L}$, we report all the valid starting positions in t by first checking if the total length $\mathcal{E}_l + \ell + \mathcal{E}_r \geqslant m$; 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

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

Example 3.2. Let the pattern x = `GGGTCTA` of length $m = 7$, and the text t = `GATACGATACCTAGGGTGATAGAATAG`. Then x' = `GGGTCTAGGGTCT` (Step 1). x' is partitioned in `GGGT`, `CTA`, `GGG`, and `TCT` (Step 2). Consider $< 4, 3, 10 > \in \mathcal{L}$, that is, fragment $x'[4\,..\,6]$ = `CTA`, of length $\ell = 3$, occurs at starting position $p_t = 10$ in t (Step 3). Then T = `GGGTCTAGGGTCTGATACGATACCTAGGGTGATAGAATAG` and T_r = `TCTGGGATCTGGGGATAAGATAGTGGGATCCATAGCATAG` (Step 4). Extending to the left gives $\mathcal{E}_l = 0$, since $T_r[9] \neq T_r[30]$; and extending to the right gives $\mathcal{E}_r = 4$, since $T[7\,..\,10] = T[26\,..\,29]$ and $T[11] \neq T[30]$ (Step 5). We check that $\mathcal{E}_l + \ell + \mathcal{E}_r = 7 = m$, and therefore we report position 10 (Step 6):

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

that is, x^4 = `CTAGGGT` occurs at starting position 10 in t.

Theorem 3.1. Given a pattern x of length m drawn from alphabet Σ, $\sigma = |\Sigma|$, and a text t of length $n > m$ drawn from Σ, algorithm CSMF requires average-case time $\mathcal{O}(n)$ to solve the CircularStringMatching problem.

Proof. Constructing and partitioning the string x' from x can trivially be done in time $\mathcal{O}(m)$ (Steps 1–2). Building the Aho–Corasick automaton of the four fragments requires time $\mathcal{O}(m)$; and the search time is $\mathcal{O}(n + Occ)$ (Step 3) (Dori and Landau 2006). The preprocessing step for the lce queries on the suffixes of T and T_r can be done in

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

time $\mathcal{O}(n)$ (Step 4). Computing $\mathcal{E}_l$ and $\mathcal{E}_r$ for each occurrence of a fragment requires time $\mathcal{O}(Occ)$ (Step 5). For each extended occurrence of a fragment, we report $\mathcal{O}(m)$ valid starting positions, thus $\mathcal{O}(mOcc)$ in total (Step 6). Since, the expected number Occ of occurrences of the four fragments in t is $4n/\sigma^{(2m-1)/4} = \mathcal{O}(\frac{n}{\sigma^{\frac{2m-1}{4}}})$, algorithm CSMF requires average-case time $\mathcal{O}((1 + \frac{m}{\sigma^{\frac{2m-1}{4}}})n)$. It achieves average-case time $\mathcal{O}(n)$ *iff*

$$f = \frac{4m}{\sigma^{\frac{2m-1}{4}}} n \leqslant cn$$

for some fixed constant c. For $\sigma = 2$, the maximum value of f is attained at

$$m = 2/\ln 2 \approx 2.8853$$

and so for $\sigma > 1$ we get

$$\frac{4m}{\sigma^{\frac{2m-1}{4}}} n \leqslant 5.05n.$$

□

3.3. *Algorithm CSMF-Simple*

In this section, we present algorithm CSMF-Simple, 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 $\langle p_{x'}, \ell, p_t \rangle \in \mathcal{L}$, Step 5 of algorithm CSMF is performed without the use of the pre-computed indexes. In other words, we compute $\mathcal{E}_r$ and $\mathcal{E}_\ell$ by simply performing letter comparisons and counting the number of mismatches occurred. The extension stops right before the first mismatch.

Fact 2. The expected number of letter comparisons required for each extension in algorithm CSMF-Simple is less than three.

Proof. Recall that on an alphabet of size σ, the probability that two random strings of length ℓ are equal is $(1/\sigma)^\ell$. Thus, given two long strings, and setting $r = 1/\sigma$, such that $r < 1$, there is probability r that the initial letters are equal, r^2 that the prefixes of length two are equal, and so on. Thus, the expected number of positions to be matched before inequality occurs can be described by the summation of infinite terms

$$S = r + 2r^2 + \cdots = \sum_{k=1}^{\infty} kr^k$$

which is bounded by $r/(1-r)^2 < 2$ for $r < 1$.

Thus S, the expected number of matching positions, is less than two, and hence, the expected number of letter comparisons required for each extension in algorithm CSMF-Simple is less than three. □

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

Theorem 3.2. Given a pattern x of length m drawn from alphabet Σ, $\sigma = |\Sigma|$, and a text t of length $n > m$ drawn from Σ, algorithm CSMF-Simple requires average-case time $\mathcal{O}(n)$ and space $\mathcal{O}(m)$ to solve the CircularStringMatching problem.

Proof. By Fact 2, computing $\mathcal{E}_\ell$ and $\mathcal{E}_r$ for each occurrence of a fragment requires expected time $\mathcal{O}(Occ)$. Therefore, algorithm CSMF-Simple requires average-case time $\mathcal{O}(n)$. The required space is reduced to $\mathcal{O}(m)$ since Step 4 of algorithm CSMF is not performed. □

4. Circular dictionary matching via filtering

In this section, we give a generalization of our algorithm for circular string matching and show that it can easily be modified to solve the problem of circular dictionary matching. We denote this new 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 $\mathcal{D} = \{x_0, x_1, \ldots, x_{d-1}\}$ of patterns of total length M and we must find all occurrences of the patterns in $\mathcal{D}$ or any of their rotations. To modify algorithm CSMF to solve this problem we perform Steps 1 and 2 for every pattern in $\mathcal{D}$, 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 2.1. 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 $\mathcal{L}$ such that it now consists of tuples of the form $\langle p_{x'_j}, \ell, j, p_t \rangle$, where j identifies the pattern the fragment was extracted from; $p_{x'_j}$ 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.

Theorem 4.1. 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 $0 \leqslant j < d$, drawn from Σ, algorithm CDMF requires average-case time $\mathcal{O}((1 + \frac{d|x_{\max}|}{\sigma^{\frac{2|x_{\min}|-1}{4}}})n + M)$ to solve the CircularDictionaryMatching problem, where $x_{\min}$ and $x_{\max}$ are the minimum- and maximum-length patterns in $\mathcal{D}$, respectively.

Proof. Constructing and partitioning the strings $x'_0, x'_1, \ldots, x'_{d-1}$ from $\mathcal{D}$ can trivially be done in time $\mathcal{O}(M)$ (Steps 1–2). Building the Aho–Corasick automaton of the $4d$ fragments requires time $\mathcal{O}(M)$; and the search time is $\mathcal{O}(n+Occ)$ (Step 3). The preprocessing step for the lce queries on the suffixes of T and T_r can be done in time $\mathcal{O}(n)$ (Step 4). Computing $\mathcal{E}_l$ and $\mathcal{E}_r$ for each occurrence of a fragment requires time $\mathcal{O}(Occ)$ (Step 5). For each extended occurrence of some fragment $\langle p_{x'_j}, \ell, j, p_t \rangle$, we may report $\mathcal{O}(|x_j|)$ valid starting positions. The expected number of occurrences for any fragment of some pattern x_j is $n/\sigma^{(2|x_j|-1)/4}$ thus $4n/\sigma^{(2|x_j|-1)/4}$ for all four fragments. So the total expected number Occ of occurrences is $\sum_{j=0}^{d-1} \frac{4n}{\sigma^{\frac{2|x_j|-1}{4}}} = \mathcal{O}(\frac{dn}{\sigma^{\frac{2|x_{\min}|-1}{4}}})$, where $x_{\min}$ is the minimum-length string in $\mathcal{D}$ (Step 6). Since the expected number Occ of occurrences of the fragments in t is

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

$\mathcal{O}(\frac{dn}{\sigma^{\frac{2|x_{\min}|-1}{4}}})$, algorithm CDMF requires average-case time $\mathcal{O}((1+\frac{d|x_{\max}|}{\sigma^{\frac{2|x_{\min}|-1}{4}}})n+M)$, where $x_{\max}$ is the maximum-length string in $\mathcal{D}$. □

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 4.2. 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 $0\leqslant j<d$, drawn from Σ, algorithm CDMF-Simple requires average-case time $\mathcal{O}((1+\frac{d|x_{\max}|}{\sigma^{\frac{2|x_{\min}|-1}{4}}})n+M)$ and space $\mathcal{O}(M)$ to solve the CircularDictionaryMatching problem, where $x_{\min}$ and $x_{\max}$ are the minimum- and maximum-length patterns in $\mathcal{D}$, respectively.

Algorithm CDMF achieves average-case time $\mathcal{O}(n+M)$ *iff*

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

for some fixed constant c. So we have

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

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

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

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

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

Corollary 4.1. 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 $0\leqslant j<d$, drawn from Σ, algorithm CDMF-Simple solves the CircularDictionaryMatching problem in average-case time $\mathcal{O}(n+M)$ *iff* $|x_{\min}|\geqslant 2(\log_\sigma d+\log_\sigma|x_{\max}|)$, where $x_{\min}$ and $x_{\max}$ are the minimum- and maximum-length patterns in $\mathcal{D}$, respectively.

5. Detailed description of CPI-II

The index data structure CPI-II essentially consists of the suffix array SA and the inverse suffix array iSA of t. The result of a query for a pattern x of length m on the suffix array SA of t of length n can be represented in the form of an interval $[s,e]$, such that the starting positions of x in t are given by the set $\{\mathsf{SA}[s],\mathsf{SA}[s+1],\ldots,\mathsf{SA}[e]\}$ of cardinality *Occ*. The query algorithm on CPI-II relies on the following two lemmas. We denote this interval by Int^t_x.

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

Lemma 5.1 (Gusfield (1997)). Given a text t of length n, its suffix array SA, and an interval Int^t_x of a query for a pattern x, the interval $\mathsf{Int}^t_{x\mathtt{c}}$ for any character $\mathtt{c}$ can be computed in time $\mathcal{O}(\log n)$.

Lemma 5.2 (Huynh *et al.* (2006)). Given a text t of length n, its suffix array SA, its inverse suffix array iSA, and two intervals $\mathsf{Int}^t_{x_1}$ and $\mathsf{Int}^t_{x_2}$ of queries for patterns x_1 and x_2, respectively, the interval $\mathsf{Int}^t_{x_1x_2}$ can be computed in time $\mathcal{O}(\log n)$.

In order to find all the rotations of a string x, we can first find the intervals for all prefixes and suffixes of x. Let $\mathsf{Pref}[i] = \mathsf{Int}^t_{x[0\,..\,i]}$ and $\mathsf{Suff}[i] = \mathsf{Int}^t_{x[i\,..\,m-1]}$.

Consider the prefixes in ascending order by length. The interval $\mathsf{Pref}[0]$ for prefix $x[0]$ can be found in time $\mathcal{O}(\log n)$ by a binary search on SA. Each subsequent interval $\mathsf{Pref}[i]$ for prefix $x[0\,..\,i]$ can be found in time $\mathcal{O}(\log n)$ by Lemma 5.1. Hence, the intervals for all prefixes can be found in time $\mathcal{O}(m \log n)$.

The intervals for the suffixes can be found in time $\mathcal{O}(m \log n)$ in a similar way. The interval $\mathsf{Suff}[m-1]$ for suffix $x[m-1]$ can be found in time $\mathcal{O}(\log n)$. Each subsequent interval $\mathsf{Suff}[i]$ can also be found in time $\mathcal{O}(\log n)$ by first finding $\mathsf{Int}^t_{\mathtt{c}}$, where $\mathtt{c} = x[i]$, in time $\mathcal{O}(\log n)$, then applying Lemma 5.2 with $\mathsf{Int}^t_{x_1} = \mathsf{Int}^t_{\mathtt{c}}$ and $\mathsf{Int}^t_{x_2} = \mathsf{Suff}[i+1]$.

Having found the intervals $\mathsf{Pref}[i]$ and $\mathsf{Suff}[i]$ for all $i \in [0, \ldots, m-1]$, we next find the set of intervals for all the rotations of x. The interval for the first rotation of x is $\mathsf{Suff}[m-1]$. For $j \in [0, \ldots, m-1]$, the interval for the j-th rotation of x can be found in time $\mathcal{O}(\log n)$ using Lemma 5.2 with $\mathsf{Int}^t_{x_1} = \mathsf{Suff}[j]$ and $\mathsf{Int}^t_{x_2} = \mathsf{Pref}[(j+m-2) \bmod (m-1)]$. Thus all intervals for all the rotations of x can be found in time $\mathcal{O}(m \log n)$. Finally, we use SA to convert this set of intervals into a set of occurrences in t of cardinality Occ. In all, this query algorithm requires time $\mathcal{O}(n + m \log n + Occ)$.

6. Experimental results

We implemented algorithms CDMF-Simple and the index data structure CPI-II as library functions to perform circular string matching for a set of patterns. The functions were implemented in the C programming language and developed under GNU/Linux operating system. They take as input arguments a set $\mathcal{D} = \{x_0, x_1, \ldots, x_{d-1}\}$ of patterns and a text t; and then return the list of starting positions of the occurrences of the rotations of $x_0, x_1, \ldots, x_{d-1}$ in t as output. The library implementation of CDMF-Simple is distributed under the GNU General Public License (GPL), and it is available at `http://www.inf.kcl.ac.uk/research/projects/asmf/`, which is set up for maintaining the source code and the man-page documentation. The library implementation of CPI-II is distributed under the GPL, and it is available at `http://github.com/blandw/cpm`. The experiments were conducted on a Desktop PC using one core of Intel i7 2600 CPU at 3.4 GHz under GNU/Linux. The programmes were compiled with `gcc` version 4.6.3 at optimization level 3 (-O3). Time and memory measurements were taken using the GNU/Linux `time` command.

To evaluate the efficiency of CDMF-Simple, we compared its performance against CPI-II using real data. As input datasets we used: for the set of patterns, a set of $d = 22{,}918$ *Sugarcane white streak virus* short reads, produced by Illumina platform, of total length

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

Table 1. *Elapsed-time and speed-up comparisons of algorithms CPI-II and CDMF-Simple using synthetic DNA data ($\sigma = 4$) for $n = 100$ MB.*

Number of patterns	Length of each pattern	CPI-II (s)	CDMF-Simple (s)	Speed-up of CDMF-Simple
10	25	8.5	4.9	1.73
100	25	8.6	5.1	1.68
1000	25	8.6	5.6	1.53
10,000	25	9.0	6.2	1.45
10	50	8.4	4.3	1.95
100	50	8.6	5.1	1.68
1000	50	8.6	5.7	1.50
10,000	50	9.4	6.2	1.51
10	100	8.6	4.3	2
100	100	8.6	4.9	1.75
1000	100	8.7	5.8	1.50
10,000	100	9.6	7.2	1.33

$M = 516,076$ base pairs; and, for the text, the *Homo sapiens* chromosome 1 sequence of length $n = 248,956,422$ base pairs. CPI-II finished the assignment in 25.1 seconds; CDMF-Simple finished in 17.7 seconds. The maximum allocated memory was 2436 MB for CPI-II and 691 MB for CDMF-Simple. As input datasets we also used: for the set of patterns, a set of $d = 14,880$ *Human immunodeficiency virus* (HIV-1) short reads, produced by 454 Life Sciences platform, of total length $M = 3,040,354$ base pairs; and, for the text, the *Homo sapiens* chromosome 1 sequence of length $n = 248,956,422$ base pairs. CPI-II finished the assignment in 30.1 seconds; CDMF-Simple finished in 25.5 seconds. The maximum allocated memory was 2436 MB for CPI-II and 675 MB for CDMF-Simple.

To further evaluate the efficiency of CDMF-Simple, we compared its performance against CPI-II using synthetic data. The data were generated using a randomized script; and the parameters for this script were chosen based on the properties (length) of real data. Tables 1 and 2 illustrate elapsed-time and speed-up comparisons for different combinations of input parameters. As it is demonstrated by the experimental results, algorithm CDMF-Simple is in all cases the fastest with a speed-up improvement between 1.3 and 2.3 over CPI-II. The maximum allocated memory (per task) for the experiments in Table 1 was 998 MB for CPI-II and 337 MB for CDMF-Simple. The maximum allocated memory (per task) for the experiments in Table 2 was 1940 MB for CPI-II and 525 MB for CDMF-Simple. Notice that, this occurs for the largest number of patterns, and CDMF-Simple has a lower memory footprint for less patterns, whilst CPI-II remains static. This confirms our theoretical findings in terms of space complexity.

Here, it becomes evident what is generally expected in practical terms: there is a certain point, as the value of M grows and n remains static, that CDMF-Simple will become slower than CPI-II; however, for smaller M, CDMF-Simple is expected to retain a significant speed-up, in particular when one has *multiple* online query texts.

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

Table 2. *Elapsed-time and speed-up comparisons of algorithms* CPI-II *and* CDMF-Simple *using synthetic DNA data ($\sigma = 4$) for $n = 200$ MB.*

Number of patterns	Length of each pattern	CPI-II (s)	CDMF-Simple (s)	Speed-up of CDMF-Simple
10	25	21.1	10.4	2.02
100	25	21.3	10.8	1.97
1000	25	21.4	11.9	1.79
10,000	25	21.5	13.3	1.61
10	50	21.2	9.2	2.3
100	50	21.5	10.8	1.99
1000	50	21.5	12.3	1.74
10,000	50	22.3	13.4	1.66
10	100	21.3	9.1	2.34
100	100	21.6	10.4	2.07
1000	100	22	12.4	1.77
10,000	100	22.3	15.3	1.45

7. Final remarks

In this article, we presented a suboptimal average-case algorithm for circular string matching requiring time $\mathcal{O}(n)$ and space $\mathcal{O}(m)$. We showed how it can be extended to solve the circular dictionary-matching problem in average-case time $\mathcal{O}(n + M)$ and space $\mathcal{O}(M)$, assuming that the shortest pattern is sufficiently long. Experimental results, using real and synthetic data, demonstrate that the implementation of the presented algorithms can accelerate the computations by more than a factor of two compared to the corresponding implementation of other approaches. For future work, we will try to improve our algorithms in order to achieve average-case optimality; and to extend our approaches to the edit distance model (Barton *et al.* 2014, 2015).

The publication costs for this article were funded by the Open Access funding scheme of King's College London. Tanver Athar is supported by an EPSRC grant (Doctoral Training Grant #EP/L504798/1). Solon P. Pissis is supported by a Research Grant (#RG130720) awarded by the Royal Society.

References

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

Barton, C., Iliopoulos, C. S. and Pissis, S. P. (2013). Circular string matching revisited. In: *Proceedings of the 4th Italian Conference on Theoretical Computer Science (ICTCS 2013)* 200–205.

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

Barton, C., Iliopoulos, C. S. and Pissis, S. P. (2014). Fast algorithms for approximate circular string matching. *Algorithms for Molecular Biology* **9** (9). Available at `http://www.almob.org/content/9/1/9`.

Barton, C., Iliopoulos, C. S. and Pissis, S. P. (2015). Average-case optimal approximate circular string matching. In: Dediu, A.-H., Formenti, E., Martin-Vide, C. and Truthe, B. (eds.) *Language and Automata Theory and Applications*, Lecture Notes in Computer Science, volume 8977 Springer, Berlin 85–96.

Belazzougui D. (2010). Succinct dictionary matching with no slowdown. In: Amir, A. and Parida, L. (eds.) *Combinatorial Pattern Matching*, Lecture Notes in Computer Science, volume 6129 Springer, Berlin 88–100.

Chan, H., Hon, W., Lam, T. and Sadakane, K. (2007). Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms* **3** (2). Available at `http://dl.acm.org/citation.cfm?doid=1240233.1240244`.

Chen, K., Huang, G. and Lee, R. C. (2013). Bit-parallel algorithms for exact circular string matching. *Computer Journal* **57** (5) 731–743.

Dori, S. and Landau, G. M. (2006). Construction of Aho Corasick automaton in linear time for integer alphabets. *Information Processing Letters* **98** (2) 66–72.

Fischer, J. (2011). Inducing the LCP-array. In: Dehne, F., Iacono, J. and Sack, J.-R. (eds.) *Algorithms and Data Structures*, Lecture Notes in Computer Science, volume 6844, Springer, Berlin 374–385.

Fischer, J. and Heun, V. (2011). Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing* **40** (2) 465–492.

Fredriksson, K. and Grabowski, S. (2009). Average-optimal string matching. *Journal of Discrete Algorithms* **7** (4) 579–594.

Frousios, K., Iliopoulos, C. S., Mouchard, L., Pissis, S. P. and Tischler, G. (2010). 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*, USA, ACM 154–159.

Gusfield D. (1997). *Algorithms on Strings, Trees and Sequences*, Cambridge University Press.

Hon, W., Ku, T., Shah, R. and Thankachan, S. V. (2013). Space-efficient construction algorithm for the circular suffix tree. In Fischer, J. and Sanders, P. (eds.) *Combinatorial Pattern Matching*, Lecture Notes in Computer Science, volume 7922, Springer, Berlin 142–152.

Hon, W., Ku, T., Shah, R., Thankachan, S. V. and Vitter, J. S. (2010). Faster compressed dictionary matching. In: Chavez, E. and Lonardi, S. (eds.) *String Processing and Information Retrieval*, Lecture Notes in Computer Science, volume 6393, Springer, Berlin 191–200.

Hon, W., Lu, C., Shah, R. and Thankachan, S. V. (2011). Succinct indexes for circular patterns. In Asano, T., Nakano, S.-I., Okamoto, Y. and Watanabe, O (eds.) *Algorithms and Computation*, Lecture Notes in Computer Science, volume 7074, Springer, Berlin 673–682.

Huynh, T. N. D., Hon, W., Lam, T. and Sung, W. (2006). Approximate string matching using compressed suffix arrays. *Theoretical Computer Science* **352** (1) 240–249.

Ilie, L., Navarro, G. and Tinta, L. (2010). The longest common extension problem revisited and applications to approximate string searching. *Journal of Discrete Algorithms* **8** (4) 418–428.

Iliopoulos, C. S. and Rahman, M. S. (2008). Indexing circular patterns. In: Nakano, S.-I. and Rahman, Md. S. (eds.) *WALCOM: Algorithms and Computation*, Lecture Notes in Computer Science, volume 4921, Springer, Berlin 46–57.

Lothaire, M. (ed.) (2005). *Applied Combinatorics on Words,* Cambridge University Press.

Manber, U. and Myers, E. W. (1993). Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* **22** (5) 935–948.

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134

Nong, G., Zhang, S. and Chan, W. H. (2009). Linear suffix array construction by almost pure induced-sorting. In: Storer, J. A. and Marcellin, M. W. (eds.) *Proceedings of the 2009 Data Compression Conference, DCC 09*, Washington, DC, USA, IEEE Computer Society 193–202.

Rivest, R. (1976). Partial-match retrieval algorithms. *SIAM Journal on Computing* **5** (1) 19–50.

Smyth, B. (2003). *Computing Patterns in Strings*. Pearson, Addison-Wesley.

Weiner, P. (1973). Linear pattern matching algorithms. In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, IEEE Computer Society 1–11.

Wu, S. and Manber, U. (1992). Fast text searching: Allowing errors. *Communications of the ACM* **35** (10) 83–91.

Downloaded from https:/www.cambridge.org/core. King's College London, on 20 Jan 2017 at 13:34:56, subject to the Cambridge Core terms of use, available at https:/www.cambridge.org/core/terms. https://doi.org/10.1017/S0960129515000134