Approximate Online Matching of Circular Strings*

Tommi Hirvola and Jorma Tarhio

Department of Computer Science and Engineering
Aalto University, P.O. Box 15400, FI-00076 Aalto, Finland
firstname.surname@aalto.fi

Abstract. Recently a fast algorithm based on the BNDM algorithm has been presented for exact matching of circular strings. From this algorithm, we derive several sublinear methods for approximate online matching of circular strings. The applicability of the new algorithms is demonstrated with practical experiments. In many cases, the new algorithms are faster than an earlier solution.

1 Introduction

The task of string matching is to find all occurrences of a *pattern* $P = p_1 \dots p_m$ in a *text* $T = t_1 \dots t_n$ where all characters in P and T are drawn from a finite alphabet Σ of size σ. In *approximate string matching* the occurrences are allowed to contain errors. We consider two variations of approximate string matching. In the k *differences problem*, the total number of mismatches, deletions and insertions between the pattern and its occurrence should be at most k. In the k *mismatches problem*, only mismatches are allowed. Many algorithms [21] for both variations of approximate string matching have been developed.

The *circular string* is a set of strings $C(S) = \{S^{(1)}, \dots, S^{(m)}\}$ corresponding to a given string $S = s_1 \dots s_m$, where $S^{(i)} = s_i \dots s_m s_1 \dots s_{i-1}$ is a *rotation* (or *conjugate*) of S. Given a text T and a circular pattern $C(P)$ for $P = p_1 \dots p_m$, the *circular pattern matching problem* (CPM) is to find all occurrences of $C(P)$ in T.

In the *exact CPM problem* (ECPM), the Z algorithm [12] finds all the exact occurrences in time $O(n + m)$. Lin and Adjeroh [16] present an $O(n \log \sigma)$ algorithm based on suffix trees. Chen et al. [6] give a solution that is sublinear on the average for ECPM. Susik et al. [26] describe a filtering method that solves the ECPM problem in average sublinear time. By *sublinearity* we mean that a part of text characters can be skipped. There are several solutions [11,16,18,23] for the *approximate CPM problem* (ACPM), but none of them are sublinear. For example, Lin and Adjeroh [16] give an $O(km^2n)$ algorithm.

In this paper, we develop sublinear online algorithms for the ACPM problem. Our point of view is the practical efficiency of algorithms. There are several

* Work supported by the Academy of Finland (grant 134287).

sublinear algorithms [21] for a single noncircular pattern, but a straightforward application of them does not guarantee sublinearity in ACPM, because a run is needed for each of the m rotations and the total time complexity is even in the best case at least $m \cdot O(n/m)$. However, Fredriksson and Navarro [9] give a sublinear multi-pattern algorithm (FN for short), which can be applied to the ACPM problem with k differences in $O((k + \log_\sigma m)\ n/m)$ average time at moderate error levels. As far as we know, the FN algorithm is the fastest one among earlier online algorithms for the ACPM problem. In this paper, we introduce three new algorithms ASB, ACB, and ACBq for the ACPM problem. The experimental results presented in Section 4 show that ASB and ACB are mostly faster than FN for English and random data. For DNA data, ACBq is mostly faster than FN.

Matching of circular strings has several applications. In computational geometry, one needs to find a polygon within a set of polygons. If the initial vertex is not provided, a polygon of k vertices has k representations and the problem can be reduced to the search of circular strings [4,13]. Music retrieval is another application area [16]. Circular strings are also found in bioinformatics. There are hundreds of protein pairs having sequences which are rotations of each other [17]. Most of the rotations retain their original three-dimensional protein structure and biological function [2], which makes their identification important. Traditional sequence alignment methods have been developed for linear sequences, and they are not well suited for circular strings, hence there is a need for new methods.

In the algorithms, we use C-like notations: '|', '&', '$\ll$', and '$\gg$' represent bitwise operations OR, AND, left shift, and right shift, respectively. The size of the computer word is denoted by w.

2 CBNDM

Our method for approximate matching of circular strings is based on the CBNDM algorithm introduced by Chen et al. [6]. Therefore we start by explaining it.

After the advent of the Shift-Or [1] algorithm, bit-parallel string matching methods have gained more and more interest. The BNDM (Backward Nondeterministic DAWG Matching) algorithm [20] is a nice example of an elegant, compact, and efficient piece of code for exact string matching. Superficially BNDM appears to be a cross of the Shift-Or and Boyer–Moore algorithms [3]. However, the Boyer–Moore algorithm searches for suffixes of the pattern while BNDM searches for factors of the pattern. BNDM simulates the nondeterministic finite factor automaton of the reverse pattern. The precomputed table B associates each character with an *instance vector* expressing its locations in the pattern. The inner loop of BNDM checks an alignment of the pattern (i.e. an alignment window in the text) in the right-to-left order. At the same time the loop recognizes prefixes of the pattern. The leftmost one of the found prefixes determines

the next alignment window of the algorithm. BNDM can be modified for approximate matching [14].

SBNDM [22,24] (short for Simple BNDM) is a simplified version of BNDM. SBNDM does not explicitly take care of prefixes, but shifts the pattern simply over the text character which caused the *state vector* D to become zero or over the first character of the alignment window in case of a match. In practice, SBNDM is slightly faster than BNDM especially for short patterns even though it examines more text characters [24]. The pseudocode of SBNDM is given as Algorithm 1. This version works for patterns of at most w characters.

Algorithm 1 (SBNDM)
1: `for` $a \in \Sigma$ `do` $B[a] \leftarrow 0$
2: `for` $i \leftarrow 1$ `to` m `do` $B[p_i] \leftarrow B[p_i] \mid (1 \ll (m-i))$
3: $i \leftarrow 1$
4: `while` $i \leq n - m + 1$ `do`
5: $j \leftarrow 1$; $D \leftarrow B[t_{i+m-j}]$
6: `while` $D \neq 0^m$ `and` $j < m$ `do`
7: $j \leftarrow j + 1$
8: $D \leftarrow (D \ll 1)$ & $B[t_{i+m-j}]$
9: `if` $j = m$ `then output` i
10: $i \leftarrow i + (m - j + 1)$

The Circular BNDM algorithm (CBNDM) is based on SBNDM. In CBNDM the left shift is replaced by the rotating left shift of m bits. The state vector on line 8 is updated as follows:

$$D \leftarrow (D \overleftrightarrow{\ll} 1) \ \& \ B[t_{i+m-j}]$$

For example, $10100 \overleftrightarrow{\ll} 1$ results in 01001 for $m = 5$.

Let $D[1], \ldots, D[m]$ be the last m bits of D. Now $D[r] = 1$ means that the substring $t_{i+m-j} \ldots t_{i+m-1}$ is the same as the prefix u of $P^{(r)}$, $|u| = j$. Especially, if $D[r] = 1$ holds when $j = m$, the alignment window contains an occurrence of $P^{(r)}$.

The only difference between the CBNDM and SBNDM algorithms is the updating of the state vector—other parts of the algorithms are identical. Actually the condition $j < m$ on line 6 is necessary only in CBNDM, not in SBNDM [7].

SBNDMq [7] is a variation of SBNDM applying q-grams. In each alignment window, SBNDMq first processes q text characters $t_i, \ldots, t_{i+q-1}$ before testing the state vector D. In practice, SBNDMq is considerably faster than SBNDM or BNDM. The running time of CBNDM can be improved in a similar way.

3 New algorithms

A filtering method for approximate matching finds match candidates, which are then checked by another method. First, we introduce a filtering algorithm based on SBNDM for *noncircular* patterns with k allowed mismatches. Later this algorithm is modified for circular patterns and for k differences. We call

this algorithm ASB for **A**pproximate **SB**NDM. Its idea is related to the Chang–Lawler algorithm (CL) [5] which applies the so-called *matching statistics*:

$$M(r, P) = \max\{j \mid j = -1 \text{ or } t_{r-j} \dots t_r \text{ is a factor of } P\} + 1.$$

ASB searches up to $k+1$ break points in M within an alignment window of m text characters. The search starts at the last character of the window: $M(i+m-1, P) = v$. Now $t_{i+m-1-v}$ is the first break point, i.e., text character that causes state vector D to become zero. See an example in Table 1.

Table 1. *Example of recognition of break points for $P =$ abbab.*

text	c	a	b	a	b	b	a	a
M	0	1	2	2	3	3	4	1
breaks	$*$		$*$				$*$	
D	00000	01001	00000	00001	00010	10110	00000	01001

Each break point corresponds to either a character not appearing in the pattern or a starting character of a new factor in a factor switch. When $k+1$ break points have been found, the window is shifted forward. If k or less break points are found, the window contains a match candidate. This is because an approximate occurrence can contain at most $k+1$ non-overlapping maximal factors of P [5].

In the CL algorithm, the distance of subsequent alignment windows is fixed, whereas ASB uses dynamic shifting. The matching statistics of CL is based on a suffix tree, which is considerably slower than the bit-parallel technique of ASB. There has been recent improvements in computation of matching statistics (e.g., [15,25]), but these methods require construction of heavy data structures which makes them slower than ASB for short patterns.

Algorithm 2 (ASB, k mismatches)
1: `for` $a \in \Sigma$ `do` $B[a] \leftarrow 0$
2: `for` $i \leftarrow 1$ `to` m `do` $B[p_i] \leftarrow B[p_i] \mid (1 \ll (i-1))$
3: $E \leftarrow 1^{m+1}$; $i \leftarrow 1$
4: `while` $i \leq n - m + 1$ `do`
5: $e \leftarrow 0$; $D \leftarrow E$; $j \leftarrow 0$
6: `while` $e \leq k$ `and` $j < m$ `do`
7: $j \leftarrow j + 1$; $D \leftarrow (D \gg 1)$ & $B[t_{i+m-j}]$
8: `if` $D = 0^m$ `then` $e \leftarrow e + 1$; $D \leftarrow E$
9: `if` $j = m$ `and` $e \leq k$ `then check candidate` $t_i \dots t_{i+m-1}$
10: $i \leftarrow i + (m - j + 1)$

The pseudocode of ASB is given as Algorithm 2. The algorithm itself does not use the matching statistics, but the break points are recognized as the state vector D becoming zero, see the example in Table 1. This is because instance vectors B form a factor automaton of the pattern. The state vector D stays

nonzero while a factor is processed from right to left. If the factor ends at t_r, then D becomes zero at t_{r-1}. After that $B[t_{r-2}]$ is assigned to D, and the computation is resumed. Note that ASB applies the right shift instead of the left shift in SBNDM. Therefore the instance vectors B are reversed and we need $m+1$ bits for the constant vector E. For this reason, it is assumed that $m \leq w-1$ holds. Alternatively, m bits could be used for E in order to be able to handle the case $m = w$ but then the code would be more complicated.

ASB can be applied directly to circular patterns by considering the discontinuation point of a circular occurrence as an extra error. So we search for $k+2$ break points, and if less break points are found in the alignment window, we have found a match candidate. ASB can also be modified to handle circular patterns by switching the bit shift to the bit rotation as done for SBNDM in Section 2. We call this algorithm ACB (short for **A**pproximate **CB**NDM). In ACB, the width of the constant vector E can be reduced to m to handle the case $m = w$.

As with most bit-parallel algorithms, ASB and ACB can be extended for patterns longer than w by using arrays for the state vector D and for each instance vector $B[a]$. ASB can also support long patterns by searching the prefix of length w of the pattern and checking for the full pattern on line 9 of Algorithm 2. In the case of ACB, one may consider both the prefix and the suffix of length w.

It is possible to speed up ASB and ACB by using q-grams as with SBNDMq. However, this variation needs that also the number of seen break points e is precomputed for each q-gram because q-grams can contain break points that would otherwise go undetected. Moreover, it is beneficial to fetch the precomputed values at each break point in addition to the right end of the alignment. We call this algorithm ACBq.

The k mismatches variation of ASB is easily extended to handle k differences. In the k differences problem, the width of the alignment window is $m-k$ to ensure that if an occurrence starts at the window position then any suffix of the window is a factor of the pattern with at most k differences so that the window is not abandoned. Moreover, in the verification step, we consider a text substring of length $m+k$ as the match candidate. The pseudocode of ASB with k differences is given as Algorithm 3. The pseudocode assumes that $m < w$ holds.

Algorithm 3 (ASB, k differences)
1: `for` $a \in \Sigma$ `do` $B[a] \leftarrow 0$
2: `for` $i \leftarrow 1$ `to` m `do` $B[p_i] \leftarrow B[p_i] \mid (1 \ll (i-1))$
3: $E \leftarrow 1^{m+1}$; $i \leftarrow 1$
4: `while` $i \leq n - (m-k) + 1$ `do`
5: $e \leftarrow 0$; $D \leftarrow E$; $j \leftarrow 0$
6: `while` $e \leq k$ `and` $j < m - k$ `do`
7: $j \leftarrow j + 1$; $D \leftarrow (D \gg 1)$ & $B[t_{i+(m-k)-j}]$
8: `if` $D = 0^m$ `then` $e \leftarrow e + 1$; $D \leftarrow E$
9: `if` $j = m - k$ `and` $e \leq k$ `then check candidate` $t_i \ldots t_{i+(m+k)-1}$
10: $i \leftarrow i + ((m-k) - j + 1)$

As done previously, circular patterns can be handled by switching the bit shift to the bit rotation or allowing one extra break point.

In the ACPM problem with k mismatches, the reported candidates can be verified by searching the match candidate S with k allowed mismatches in a text string PP. This can be done because the string PP contains all rotations of P. In ACPM with k differences, the lengths of the occurrences are not known in advance and the same method cannot be used. Instead, we search each rotation of P in S using an approximate matching algorithm that allows k differences, e.g., Myers' algorithm [19].

4 Analysis

In this section, we prove that the k differences variation of ACB is sublinear on the average. Sublinearity of our other algorithms can be proven with the same method by adjusting the window size and constants which do not change the resulting average time complexity. The proof is similar to that of the CL algorithm [5].

The goal is to first determine the expected number of character comparisons done in each alignment window. This number gives us the average shift length, which has to exceed half of the window width in order to skip characters in the text. Moreover, we need to consider the probability of a match candidate occurring in the window to show that the verification step does not worsen the average performance with reasonable values of m and k.

Assume that the characters of P and T are chosen independently and uniformly from Σ of size $\sigma \geq 2$. The inner loop of ACB finds up to $k+1$ factors and break points of P in the alignment window. Let X_i be a random variable representing the length of the ith factor. Since the circular pattern P contains at most m distinct substrings of length $\log_\sigma m + d$, and there are $m\sigma^d$ different strings of that length, we get:

$$\forall \text{ integer } d \geq 0, \Pr[X_i = \log_\sigma m + d] < \sigma^{-d}$$

This gives us the following upper bound for the expected factor length:

$$E[X_i] < \log_\sigma m + \sum_{d=0}^{\infty} d\sigma^{-d} \leq \log_\sigma m + 2$$

Thus, ACB reads $(k+1)(\log_\sigma m + 3)$ characters per window on the average. This has to be less than $(m-k)/2$ to achieve sublinearity. Equating $(k+1)(\log_\sigma m + 3) < (m-k)/2$ yields us the following threshold for k:

$$k < \frac{m - 2\log_\sigma m - 6}{2\log_\sigma m + 7}$$

Now, we get the average time complexity of ACB by multiplying the expected number of examined windows with the average work done in each window:

$$O\left(\frac{n-m+1}{m-k-(k+1)(\log_\sigma m+3)}\right) \cdot O((k+1)(\log_\sigma m+3))$$

This is equal to $O((n/m)k \log_\sigma m)$ given that $n \gg m$ and $k < m/(\log_\sigma m + O(1))$.

In conclusion, ACB is sublinear on the average when k is bounded appropriately and the match verification time is excluded. It is evident that match candidates are rare and do not affect the average time complexity if m is large and k is small. This can be formally proven by computing an upper bound for $\Pr[k + X_1 + X_2 + \cdots + X_{k+1} \geq m - k]$ with the help of Chernoff bounds. In [5], the probability is shown to be less than $1/m^3$ when k is upper bounded by the threshold $m/(\log_\sigma m + O(1))$.

5 Experiments

The test data consisted of three different types of text: DNA (4.5 MB), English (4.0 MB) and Rand256 (5.0 MB). The DNA text was the genome of E.coli. The English text was the King James Bible. The Rand256 text contained randomly generated data in the alphabet of 256 symbols. The texts were taken from the corpus of the SMART tool [8].

The patterns used in the tests were selected from the texts, and the resulting strings were randomly rotated and substituted in 0–5 characters positions to simulate mismatches. The pattern sets for the k differences algorithms allowed also insertions and deletions. In each test run, 1000 patterns were searched in the 2 MB long prefix of each text. Test runs were repeated 3 times for each algorithm and for each test set. Verification of match candidates was done as explained in Section 3.

As a reference method, we used the FN algorithm [9]. As far as we know, FN has been the best online algorithm for approximate matching of multiple patterns. We applied FN to the ACPM problem by forming all possible rotations of a pattern and searching them simultaneously. The implementation was obtained from the authors and ran with command-line options `-D -t6 -B -Sb` for the DNA text, and `-A -t2 -B -Sb` for the English and Rand256 texts. The option `-s` was used for searches allowing only mismatches. FN is also a filtering method and its implementation uses similar verification algorithms as our methods.

We also ran preliminary tests with an approximate multiple pattern matching algorithm proposed by Fulwider and Mukherjee [10], but that algorithm was not competitive.

The test computer has the Intel® Core™ i7 860 2.80 GHz processor with 16 GB DDR3 main memory. The operating system is Linux (kernel 3.2.0-48-generic). The test processes were run on a single core. The algorithms were implemented in C and compiled with gcc 4.6.3 using O2 or O3 optimization.

Table 2 shows test results in the case of k mismatches. ACB is faster than ASB on all the inputs and the values of k. This is because ASB gives more false positives than ACB. ACB is also significantly faster than FN for English and Rand256 data. When the preprocessing times of the patterns are included, our algorithms are much faster than FN in most tested cases because FN has a heavy preprocessing phase whereas ASB and ACB have negligible preprocessing

Table 2. *Running times (in seconds) of algorithms for approximate circular matching with k mismatches. Preprocessing times of FN are in parentheses. Preprocessing times of ASB and ACB have been included in the running times.*

		$k = 1$			$k = 2$			$k = 5$		
Σ	m	FN	ASB	ACB	FN	ASB	ACB	FN	ASB	ACB
DNA	20	2.16 (6.21)	8.64	4.41	4.75 (6.16)	16.01	8.23	442.68 (6.29)	1741.85	684.77
	40	1.05 (15.11)	3.55	2.13	1.97 (15.04)	5.14	3.49	4.14 (15.18)	14.47	10.49
	60	0.77 (28.70)	2.26	1.43	1.31 (28.61)	3.13	2.28	2.25 (28.73)	6.93	5.53
English	20	2.70 (56.81)	4.75	2.55	4.04 (56.95)	6.92	3.97	63.91 (78.91)	28.84	14.33
	40	1.68 (263.71)	2.44	1.41	2.38 (217.48)	3.40	2.09	5.62 (203.50)	7.05	4.83
	60	1.33 (443.65)	1.69	1.02	1.69 (397.06)	2.29	1.49	3.19 (382.84)	4.38	3.19
Rand256	20	1.69 (57.42)	1.12	0.82	2.23 (57.56)	1.53	1.13	5.99 (79.51)	3.04	2.32
	40	1.19 (266.26)	0.80	0.52	1.34 (219.86)	1.04	0.71	2.51 (205.70)	1.75	1.35
	60	0.94 (450.82)	0.67	0.43	0.94 (404.10)	0.85	0.57	1.74 (390.07)	1.37	1.04

times (at most a few milliseconds). However, excluding the preprocessing times, FN beats the other algorithms on DNA data and ASB on English data. The significance of preprocessing times depends much on user needs. The shorter texts are processed, the larger is the proportion of preprocessing in the total times. In addition, it must be noted that FN has a specific option for DNA data allowing tuned computation, whereas ASB and ACB make no assumptions about the input data. Also, decreasing m reduces the preprocessing times of FN accordingly, which makes FN more competitive in terms of total execution time for short patterns.

Table 3. *Running times (in seconds) of algorithms for approximate circular matching with k differences. The preprocessing times of FN are shown in parentheses. Preprocessing times of ASB and ACB are included in the running times.*

		$k = 1$			$k = 2$			$k = 5$		
Σ	m	FN	ASB	ACB	FN	ASB	ACB	FN	ASB	ACB
DNA	20	2.56 (5.60)	7.21	3.67	7.19 (5.56)	34.88	8.79	1932.83 (5.49)	6066.03	5540.40
	40	1.18 (8.92)	2.84	1.71	2.06 (8.81)	4.32	3.09	14.17 (8.80)	31.03	12.81
	60	0.98 (12.29)	1.83	1.16	1.19 (12.41)	2.71	2.01	2.91 (12.46)	7.26	5.84
English	20	2.99 (44.24)	3.65	2.39	5.16 (44.76)	6.83	4.35	661.07 (66.48)	840.25	254.65
	40	1.86 (169.99)	1.85	1.26	2.69 (123.98)	2.81	2.02	11.29 (109.40)	7.69	6.14
	60	1.59 (212.83)	1.33	0.91	1.86 (166.77)	1.96	1.43	4.01 (152.77)	4.33	3.81
Rand256	20	1.75 (44.27)	0.95	0.69	2.60 (44.25)	1.40	1.09	8.82 (66.65)	4.54	3.78
	40	1.18 (170.25)	0.61	0.44	1.45 (123.77)	0.84	0.66	3.00 (109.56)	1.99	1.80
	60	0.90 (213.54)	0.49	0.36	1.01 (167.20)	0.67	0.54	2.12 (152.99)	1.55	1.41

The results for approximate matching with k differences are presented in Table 3. Again ACB is the fastest algorithm in nearly all the test runs when the preprocessing times of FN are included. The only exception is DNA for $m = 20$ and $k = 5$ where FN beats both ASB and ACB. In this test run, the number of checked match candidates is high and FN handles the case better than our algorithms. In terms of scanning time only, ACB is the fastest algorithm on English and Rand256, but slightly slower than FN on DNA.

Note that in many cases ASB and ACB are faster when allowing differences instead of only mismatches. This is due to the faster occurrence verification algorithm applied in the k differences implementations of the algorithms. Also, the preprocessing of FN is faster for the k differences case than k mismatches.

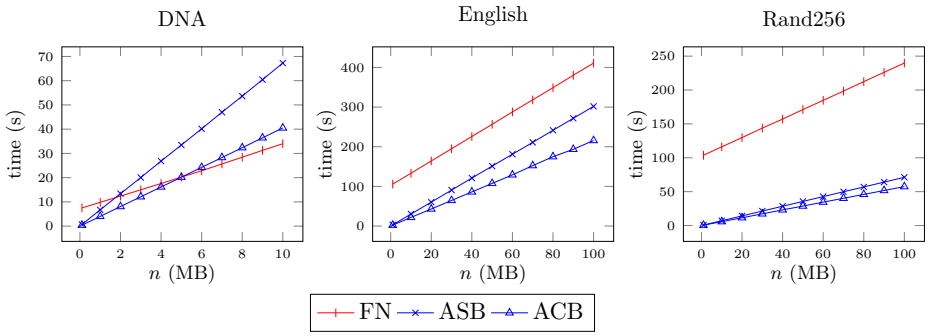

Fig. 1. *Total running times of the k difference algorithms as a function of the input text size for $m = 30$ and $k = 3$. The range is 0–10 MB for DNA and 0–100 MB for English and Rand256.*

Figure 1 shows that the running times of the algorithms grow linearly when the text length is increased. The long input texts were produced by concatenating the texts with themselves. We used 10 MB for DNA in order to visualize the intersection points clearly, while 100 MB was used for English and Rand256. On the DNA text, FN is slower than ASB and ACB for short texts due to the heavy preprocessing. However, FN becomes faster than the other algorithms when n increases. ASB and ACB are slower than FN on DNA when n exceeds 2 MB and 5 MB, respectively. On English and Rand256, our algorithms are faster than FN for all values of n.

Finally, we tuned our best algorithm, ACB, with q-grams for DNA data. The precomputed D and e values were stored into a table of σ^q ($\sigma = 4$) elements. The results are shown in Table 4.

Table 4. *Running times (in seconds) of ACBq for the k differences problem. Preprocessing times are excluded. The times are comparable to those in Table 3.*

		$k = 1$			$k = 2$			$k = 5$		
Σ	m	ACB4	ACB6	ACB8	ACB4	ACB6	ACB8	ACB4	ACB6	ACB8
DNA	20	2.62	1.97	1.36	6.19	7.10	6.63	5522.47	5524.56	5545.28
	40	1.28	1.10	0.75	2.07	2.40	2.59	9.51	9.42	11.42
	60	0.91	0.84	0.66	1.46	1.60	1.81	4.55	4.34	5.36

The q-gram variations are faster than the original ACB algorithm in all the test cases except for $q = 8$, $k = 5$ and $m = 20$. Interestingly, ACB8 was the fastest algorithm for $k = 1$, ACB4 for $k = 2$ and ACB6 for $k = 5$. ACBq also beats the running times of FN in several cases, especially for low k values.

6 Concluding remarks

We have developed several sublinear algorithms for approximate online matching of circular strings with k errors. Our experiments show that the new algorithms work well in practice at reasonable error levels. According to our tests, our algorithms are faster than the current top algorithm FN [9] in many cases. The main advantages over FN are a compact implementation and significantly lower preprocessing times, which makes the new algorithms faster than FN by orders of magnitude for short texts and pattern lengths close to w. Our algorithms are also better suited for large alphabets than FN.

References

1. Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Communications of the ACM 35(10), 74–82 (1992).
2. Bliven, S., Prlic, A.: Circular permutation in proteins. PLoS Computational Biology 8(3) (2012).
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Comm. ACM 20(10), 762–772 (1977).
4. Bunke, H., Bühler, U.: Applications of approximate string matching to 2D shape recognition. Pattern Recognition 26(12), 1797–1812 (1993).
5. Chang, W.I., Lawler, E.L.: Sublinear approximate string matching and biological applications. Algorithmica 12(4/5), 327–344 (1994).
6. Chen, K.H., Huang, G.S., Lee, R.C.T.: Exact circular pattern matching using the bit-parallelism and q-gram technique. In: Proc. The 29th Workshop on Combinatorial Mathematics and Computation Theory. pp. 18–27. National Taipei College of Business (2012).
7. Ďurian, B., Holub, J., Peltola, H., Tarhio, J.: Improving practical exact string matching. Information Processing Letters 110(4), 148–152 (2010).
8. Faro, S., Lecroq, T.: Smart: a string matching algorithm research tool. University of Catania and Univeristy of Rouen (2011). `http://www.dmi.unict.it/~faro/smart/`
9. Fredriksson, K., Navarro, N.: Average-optimal single and multiple approximate string matching. ACM Journal of Experimental Algorithmics 9, article 1.4 (2004).
10. Fulwider, S., Mukherjee, A.: Multiple Pattern Matching. In: PATTERNS 2010, The Second International Conferences on Pervasive Patterns and Applications, 78–83 (2010).
11. Gregor, J., Thomason, M.G.: Dynamic programming alignment of sequences representing cyclic patterns. IEEE Trans. Pattern Anal. Mach. Intell. 15, 129–135 (1993).
12. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997).

13. Huh, Y., Yu, K., Heo, J.: Detecting conjugate-point pairs for map alignment between two polygon datasets. Computers, Environment and Urban Systems 35(3), 250–262 (2011).
14. Hyyrö, H., Navarro, G.: Bit-parallel witnesses and their applications to approximate string matching. Algorithmica 41(3), 203–231 (2005).
15. Kärkkäinen, J., Kempa, D., Puglisi, S.: Lightweight Lempel-Ziv parsing. In: Experimental Algorithms, pp. 139–150. Springer Berlin Heidelberg (2013).
16. Lin, J., Adjeroh, D.: All-against-all circular pattern matching. Computer Journal 55(7), 897–906 (2012).
17. Lo, W.C., Lee, C.C., Lee, C.Y., Lyu, P.C.: CPDB: A database of circular permutation in proteins. Nucleic acids research 37(Suppl. 1), D328–D332 (2009).
18. Marzal, A., Barrachina, S.: Speeding up the computation of the edit distance for cyclic strings. In: Int'l Conference on Pattern Recognition, pp. 891–894. IEEE Computer Society (2000).
19. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM 46(3), 395–415 (1999).
20. Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. Journal of Experimental Algorithmics 5 (2000).
21. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. 33(1), 31–88 (2001).
22. Navarro, G.: NR-grep: A fast and flexible pattern-matching tool. Softw. Pract. Exp. 31(13), 1265–1312 (2001).
23. Oncina, J.: The Cocke-Younger-Kasami algorithm for cyclic strings. In: ICPR '96: Proc. 13th Int. Conf. Pattern Recognition, Vienna, Austria, pp. 413–416. IEEE Computer Society (1996).
24. Peltola, H., Tarhio, J.: Alternative algorithms for bit-parallel string matching. In: Proc. String Processing and Information Retrieval, 10th International Symposium (SPIRE '03). Lecture Notes in Computer Science, vol. 2857, pp. 80–94. Springer (2003).
25. Schnattinger, T., Ohlebusch, E., Gog, S.: Bidirectional search in a string with wavelet trees and bidirectional matching statistics. Information and Computation 213, 13–22 (2012).
26. Susik, R., Grabowski, S., Deorowicz, S.: Fast and simple circular pattern matching. In: Proc. 3rd International Conference on Man-Machine Interactions, Advances in Intelligent Systems and Computing, vol. 242, pp. 537–544. Springer (2014).