Longest Property-Preserved Common Factor

Lorraine A. K. Ayad[1], Giulia Bernardini[2], Roberto Grossi[3], Costas S. Iliopoulos[1], Nadia Pisanti[3,4(✉)], Solon P. Pissis[1], and Giovanna Rosone[3]

[1] Department of Informatics, King's College London, London, UK
{lorraine.ayad,c.iliopoulos,solon.pissis}@kcl.ac.uk
[2] Department of Informatics, Systems and Communication, University of Milan-Bicocca, Milan, Italy
giulia.bernardini@unimib.it
[3] Department of Computer Science, University of Pisa, Pisa, Italy
{grossi,pisanti}@di.unipi.it, giovanna.rosone@unipi.it
[4] ERABLE Team, INRIA, Lyon, France

Abstract. In this paper we introduce a new family of string processing problems. We are given two or more strings and we are asked to compute a factor common to all strings that preserves a specific property and has maximal length. Here we consider two fundamental string properties: square-free factors and periodic factors under two different settings, one per property. In the first setting, we are given a string x and we are asked to construct a data structure over x answering the following type of on-line queries: given string y, find a longest square-free factor common to x and y. In the second setting, we are given k strings and an integer $1 < k' \le k$ and we are asked to find a longest periodic factor common to at least k' strings. We present linear-time solutions for both settings. We anticipate that our paradigm can be extended to other string properties.

Keywords: Longest common factor · Periodicity · Squares Algorithms

1 Introduction

In the longest common factor problem, also known as longest common substring problem, we are given two strings x and y, each of length at most n, and we are asked to find a maximal-length string occurring in both x and y. This is a classical and well-studied problem in computer science arising out of different practical scenarios. It can be solved in $\mathcal{O}(n)$ time and space [8,15] (see also [18,23]). Recently, the same problem has been extensively studied under distance metrics; that is, the sought factors (one from x and one from y) must be at distance at most k and have maximal length [1,7,21,22,24,25] (and references therein).

In this paper we initiate a new related line of research. We are given two or more strings and our goal is to compute a *factor* common to all strings that preserves a specific *property* and has maximal length. An analogous line of research was introduced in [9]. It focuses on computing a *subsequence* (rather than a factor) common to all strings that preserves a specific property and has maximal length. Specifically, in [2,9,16], the authors considered computing a longest common palindromic subsequence and in [17] computing a longest common square subsequence.

We consider two fundamental string properties: *square-free* factors and *periodic* factors [20] under two different settings, one per property. In the first setting, we are given a string x and we are asked to construct a data structure over x answering the following type of on-line queries: given string y, find a longest square-free factor common to x and y. In the second setting, we are given k strings and an integer $1 < k' \leq k$ and we are asked to find a longest periodic factor common to at least k' strings. We present linear-time solutions for both settings. We anticipate that our paradigm can be extended to other string properties.

1.1 Definitions and Notation

An *alphabet* Σ is a non-empty finite ordered set of letters of size $\sigma = |\Sigma|$. In this work we consider that $\sigma = \mathcal{O}(1)$ or that Σ is a linearly-sortable integer alphabet. A *string* x on an alphabet Σ is a sequence of elements of Σ. The set of all strings on an alphabet Σ, including the *empty string* ε of length 0, is denoted by Σ^*. For any string x, we denote by $x[i..j]$ the *substring* (sometimes called *factor*) of x that starts at position i and ends at position j. In particular, $x[0..j]$ is the *prefix* of x that ends at position j, and $x[i..|x|-1]$ is the *suffix* of x that starts at position i, where $|x|$ denotes the *length* of x. A string uu, $u \in \Sigma^*$, is called a *square*. A *square-free* string is a string that does not contain a square as a factor.

A *period* of $x[0..|x|-1]$ is a positive integer p such that $x[i] = x[i+p]$ holds for all $0 \leq i < |x| - p$. The smallest period of x is denoted by $\mathsf{per}(x)$. String u is called *periodic* if and only if $\mathsf{per}(u) \leq |u|/2$. A *run* of string x is an interval $[i, j]$ such that for the smallest period $p = \mathsf{per}(x[i..j])$ it holds that $2p \leq j - i + 1$ and the periodicity cannot be extended to the left or right, *i.e.,* $i = 0$ or $x[i-1] \neq x[i+p-1]$, and, $j = |x| - 1$ or $x[j-p+1] \neq x[j+1]$.

1.2 Algorithmic Toolbox

The maximum number of runs in a string of length n is less than n [3], and, moreover, all runs can be computed in $\mathcal{O}(n)$ time [3,19].

The *suffix tree* $\mathsf{ST}(x)$ of a non-empty string x of length n is a compact trie representing all suffixes of x. $\mathsf{ST}(x)$ can be constructed in $\mathcal{O}(n)$ time [12]. We can analogously define and construct the *generalised suffix tree* $\mathsf{GST}(x_0, x_1, \ldots, x_{k-1})$ for a set of k strings. We assume the reader is familiar with these data structures.

The matching statistics capture all matches between two strings x and y [6]. More formally, the *matching statistics* of a string $y[0..|y|-1]$ with respect to a string x is an array $\mathsf{MS}_y[0..|y|-1]$, where $\mathsf{MS}_y[i]$ is a pair (ℓ_i, p_i) such that (i) $y[i..i+\ell_i-1]$ is the longest prefix of $y[i..|y|-1]$ that is a factor of x; and (ii)

$x[p_i..p_i + \ell_i - 1] = y[i..i + \ell_i - 1]$. Matching statistics can be computed in $\mathcal{O}(|y|)$ time for $\sigma = \mathcal{O}(1)$ by using $\mathsf{ST}(x)$ [5,14,15].

Given a rooted tree T with n leaves coloured from 0 to $k-1$, $1 < k \leq n$, the *colour set size* problem is finding, for each internal node u of T, the number of different leaf colours in the subtree rooted at u. In [8], the authors present an $\mathcal{O}(n)$-time solution to this problem.

In the *weighted ancestor* problem, introduced in [13], we consider a rooted tree T with an integer weight function μ defined on the nodes. We require that the weight of the root is zero and the weight of any other node is strictly larger than the weight of its parent. A weighted ancestor query, given a node v and an integer value $\ell \leq \mu(v)$, asks for the highest ancestor u of v such that $\mu(u) \geq \ell$, *i.e.*, such an ancestor u that $\mu(u) \geq \ell$ and $\mu(u)$ is the smallest possible. When T is the suffix tree of a string x of length n, we can locate the locus of any factor of $x[i..j]$ using a weighted ancestor query. We define the weight of a node of the suffix tree as the length of the string it represents. Thus a weighted ancestor query can be used for the terminal node corresponding to $x[i..n-1]$ to create (if necessary) and mark the node that corresponds to $x[i..j]$. Given a collection Q of weighted ancestor queries on a weighted tree T on n nodes with integer weights up to $n^{\mathcal{O}(1)}$, all the queries in Q can be answered *off-line* in $\mathcal{O}(n + |Q|)$ time [4].

2 Square-Free-Preserved Matching Statistics

In this section, we introduce the square-free-preserved matching statistics problem and provide a linear-time solution. In the *square-free-preserved matching statistics* problem we are given a string x of length n and we are asked to construct a data structure over x answering the following type of on-line queries: given string y, find the longest square-free prefix of $y[i..|y|-1]$ that is a factor of x, for all $0 \leq i < |y| - 1$. (For related work see [10].) We represent the answer using an integer array $\mathsf{SQMS}_y[0..|y|-1]$ of lengths, but we can trivially modify our algorithm to report the actual factors. It should be clear that a maximum element in SQMS gives the length of some longest square-free factor common to x and y.

Construction. Our data structure over string x consists of the following:

- An integer array $L_x[0..n-1]$, where $L_x[i]$ stores the length of the longest square-free factor starting at position i of string x.
- The suffix tree $\mathsf{ST}(x)$ of string x.

The idea for constructing array L_x efficiently is based on the following crucial observation.

Observation 1. *If $x[i..n-1]$ contains a square then $L_x[i]+1$, for all $0 \leq i < n$, is the length of the* shortest prefix *of $x[i..n-1]$ (factor f) containing a square. In fact, the square is a suffix of f, otherwise f would not have been the shortest. If $x[i..n-1]$ does not contain a square then $L_x[i] = n - i$.*

We thus shift our focus to computing the shortest such prefixes. We start by considering the runs of x. Specifically, we consider squares in x observing that a run $[\ell, r]$ with period p contains $r - \ell - 2p + 2$ squares of length $2p$ with the leftmost one starting at position ℓ. Let $r' = \ell + 2p - 1$ denote the ending position of the leftmost such square of the run. In order to find, for all i's, the shortest prefix of $x[i..n-1]$ containing a square s, and thus compute $L_x[i]$, we have two cases:

1. s is part of a run $[\ell, r]$ in x that starts *after* i. In particular, $s = x[\ell..r']$ such that $r' \leq r$, $\ell > i$, and r' is minimal. In this case the shortest factor has length $\ell + 2p - i$; we store this value in an integer array $C[0..n-1]$. If no run starts after position i we set $C[i] = \infty$. To compute C, after computing in $\mathcal{O}(n)$ time all the runs of x with their p and r' [3,19], we sort them by r'. A right-to-left scan after this sorting associates to i the closest r' with $\ell > i$.
2. s is part of a run $[\ell, r]$ in x and $i \in [\ell, r]$. This implies that if $i \leq r - 2p + 1$ then a square *starts at* i and we store the length of the shortest such square in an integer array $S[0..n-1]$. If no square starts at position i we set $S[i] = \infty$. Array S can be constructed in $\mathcal{O}(n)$ time by applying the algorithm of [11].

Since we do not know which of the two cases holds, we compute both C and S. By Observation 1, if $C[i] = S[i] = \infty$ ($x[i..n-1]$ does not contain a square) we set $L_x[i] = n - i$; otherwise ($x[i..n-1]$ contains a square) we set $L_x[i] = \min\{C[i], S[i]\} - 1$.

Finally, we build the suffix tree $\mathsf{ST}(x)$ of string x in $\mathcal{O}(n)$ time [12]. This completes our construction.

Querying. We rely on the following fact for answering the queries efficiently.

Fact 2. *Every factor of a square-free string is square-free.*

Let string y be an on-line query. Using $\mathsf{ST}(x)$, we compute the matching statistics MS_y of y with respect to x. For each $j \in [0, |y| - 1]$, $\mathsf{MS}_y[j] = (\ell_i, i)$ indicates that $x[i..i + \ell_i - 1] = y[j..j + \ell_i - 1]$. This computation can be done in $\mathcal{O}(|y|)$ time [5,15]. By applying Fact 2, we can answer any query y in $\mathcal{O}(|y|)$ time for $\sigma = \mathcal{O}(1)$ by setting $\mathsf{SQMS}_y[j] = \min\{\ell_i, L_x[i]\}$, for all $0 \leq j \leq |y| - 1$.

We arrive at the following result.

Theorem 3. *Given a string x of length n over an alphabet of size $\sigma = \mathcal{O}(1)$, we can construct a data structure of size $\mathcal{O}(n)$ in time $\mathcal{O}(n)$, answering SQMS_y on-line queries in $\mathcal{O}(|y|)$ time.*

Proof. The time complexity of our algorithm follows from the above discussion.

We next show the correctness of our algorithm. Let us first show the correctness of computing array L_x. The square contained in the shortest prefix of $x[i..n-1]$ (containing a square) starts by definition either at i or after i. If it starts at i this is correctly computed by the algorithm of [11] which assigns the length of the shortest such square in $S[i]$. If it starts after i it must be the leftmost square of another run by the runs definition. $C[i]$ stores the length of the shortest prefix containing such a square. Then by Observation 1, $L_x[i]$ is computed correctly.

It suffices to show that, if w is the longest square-free substring common to x and y occurring at position i_x in x and at position i_y in y, then (i) $\mathsf{MS}_y[i_y] = (\ell, i_x)$ with $\ell \geq |w|$ and $x[i_x..i_x+\ell-1] = y[i_y..i_y+\ell-1]$; (ii) w is a prefix of $x[i_x..i_x+L_x[i_x]-1]$; and (iii) $\mathsf{SQMS}_y[i_y] = |w|$. Case (i) directly follows from the correctness of the matching statistics algorithm. For Case (ii), since w occurs at i_x and w is square-free, $L_x[i_x] \geq |w|$. For Case (iii), since w is square-free we have to show that $|w| = \min\{\ell_i, L_x[i]\}$. We know from (i) that $\ell \geq |w|$ and from (ii) that $L_x[i_x] \geq |w|$. If $\min\{\ell_i, L_x[i]\} = \ell$, then w cannot be extended because the possibly longer than $|w|$ square-free string occurring at i_x does not occur in y, and in this case $|w| = \ell$. Otherwise, if $\min\{\ell_i, L_x[i]\} = L_x[i_x]$ then w cannot be extended because it is no longer square-free, and in this case $|w| = L_x[i_x]$. Hence we conclude that $\mathsf{SQMS}_y[i_y] = |w|$. The statement follows. □

The following example provides a complete overview of the workings of our algorithm.

Example 4. Let x = `aababaababb` and y = `bababab baaab`. The length of a longest common square-free factor is 3, and the factors are `bab` and `aba`.

i	0	1	2	3	4	5	6	7	8	9	10	
$x[i]$	a	a	b	a	b	a	a	b	a	b	b	
$C[i]$	5	6	5	4	3	5	5	4	3	∞	∞	
$S[i]$	2	4	4	6	∞	2	4	∞	∞	2	∞	
$L_x[i]$	1	3	3	3	2	1	3	3	2	1	1	
j	0	1	2	3	4	5	6	7	8	9	10	11
$y[j]$	b	a	b	a	b	a	b	b	a	a	a	b
$\mathsf{MS}_y[j]$	(4,2)	(5,1)	(4,2)	(5,6)	(4,7)	(3,8)	(2,9)	(3,4)	(2,0)	(3,0)	(2,1)	(1,2)
$\mathsf{SQMS}_y[j]$	3	3	3	3	3	2	1	2	1	1	2	1

3 Longest Periodic-Preserved Common Factor

In this section, we introduce the longest periodic-preserved common factor problem and provide a linear-time solution. In the *longest periodic-preserved common factor* problem, we are given $k \geq 2$ strings $x_0, x_1, \ldots, x_{k-1}$ of total length N and an integer $1 < k' \leq k$, and we are asked to find a longest periodic factor common to at least k' strings. We represent the answer $\mathsf{LPCF}_{k'}$ by the length of a longest factor, but we can trivially modify our algorithm to report an actual factor. Our algorithm, denoted by LPCF, works as follows.

1. Compute the runs of string x_j, for all $0 \leq j < k$.
2. Construct the generalised suffix tree $\mathsf{GST}(x_0, x_1, \ldots, x_{k-1})$ of $x_0, x_1, \ldots, x_{k-1}$.

3. For each string x_j and for each run $[\ell, r]$ with period p_ℓ of x_j, augment GST with the explicit node spelling $x[\ell..r]$, decorate it with p_ℓ, and mark it as a *candidate* node. This can be done as follows: for each run $[\ell, r]$ of x_j, for all $0 \leq j < k$, find the leaf corresponding to $x_j[\ell..|x_j|-1]$ and answer the weighted ancestor query in GST with weight $r-\ell+1$. Let aGST be this tree.
4. Mark as *good* the nodes of aGST having at least k' different colours on the leaves of the subtree rooted there.
5. Return as $\mathsf{LPCF}_{k'}$ the string depth of a candidate node in aGST which is also a good node, and that has maximal string depth (if any, otherwise return 0).

Theorem 5. *Given k strings of total length N on alphabet $\Sigma = \{1, \dots, N^{\mathcal{O}(1)}\}$, and an integer $1 < k' \leq k$, algorithm* LPCF *returns $\mathsf{LPCF}_{k'}$ in time $\mathcal{O}(N)$.*

Proof. Let us assume wlog that $k' = k$, and let w with period p be the longest periodic factor common to all strings. By the construction of aGST (Steps 1-4 of LPCF), the path spelling w leads to a good node n_w as w occurs in all the strings. We make the following observation.

Observation 6. *Each periodic factor with period p of string x is a factor of $x[i..j]$, where $[i, j]$ is a run with period p.*

By Observation 6, in all strings, w is included in a run having the same period. Observe that for at least one of the strings, there is a run ending with w, otherwise we could extend w obtaining a longer periodic common factor. Therefore n_w is *both* a good and a candidate node. By definition, n_w is at string depth at least $2p$ and, by construction, $\mathsf{LPCF}_{k'}$ is the string depth of a deepest such node; thus $|w|$ will be returned by Step 5.

As for the time complexity, Step 1 [3,19] and Step 2 [12] can be done in $\mathcal{O}(N)$ time. Since the total number of runs is less than N [3], Step 3 can be done in $\mathcal{O}(N)$ time using off-line weighted ancestor queries [4], and the size of the aGST is still in $\mathcal{O}(N)$. Step 4 can be done in $\mathcal{O}(N)$ time [8]. Step 5 can be done in $\mathcal{O}(N)$ by a post-order traversal of aGST. □

The following example provides a complete overview of the workings of our algorithm.

Example 7. Consider $x =$ `ababbabba`, $y =$ `ababaab`, and $k = k' = 2$. The runs of x are: $r_0 = [0, 3]$, per(abab) $= 2$, $r_1 = [1, 8]$, per(babbabba) $= 3$, $r_2 = [3, 4]$, per(bb) $= 1$, and $r_3 = [6, 7]$, per(bb) $= 1$; those of y are $r_4 = [0, 4]$, per(ababa) $= 2$ and $r_5 = [4, 5]$, per(aa) $= 1$. Figure 1 shows aGST for x, y, and $k = k' = 2$. Algorithm LPCF outputs $4 = |$abab$|$, with per(abab) $= 2$, as the node spelling `abab` is the deepest good one that is also a candidate.

4 Final Remarks

We introduced a new family of string processing problems. The goal is to compute factors common to a set of strings preserving a specific property and having

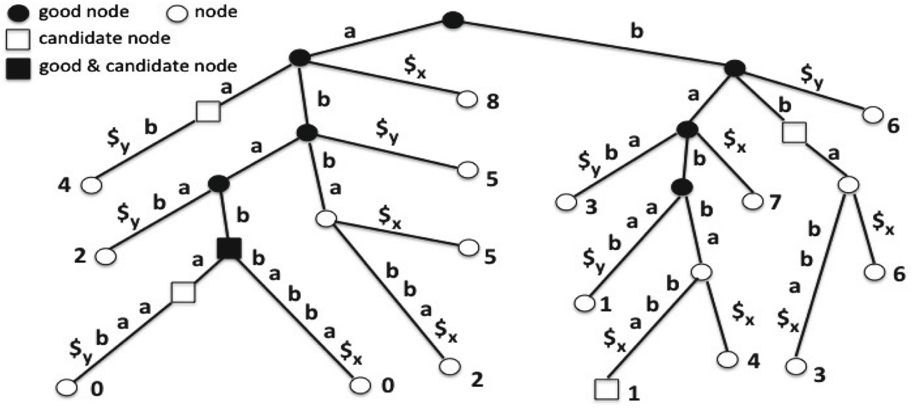

Fig. 1. aGST for $x = \texttt{ababbabba}$, $y = \texttt{abab aab}$, and $k = k' = 2$.

maximal length. We showed linear-time algorithms for square-free and periodic factors. We anticipate that our paradigm can be extended to other string properties.

Acknowledgements. Solon P. Pissis and Giovanna Rosone are partially supported by the Royal Society project IE 161274 "Processing uncertain sequences: combinatorics and applications". Giovanna Rosone and Nadia Pisanti are partially supported by the project Italian MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

References

1. Ayad, L.A.K., Barton, C., Charalampopoulos, P., Iliopoulos, C.S., Pissis, S.P.: Longest common prefixes with k-errors and applications. In: Gagie, T., et al. (eds.) SPIRE 2018. LNCS, vol. 11147, pp. 27–41. Springer, Heidelberg (2018)
2. Bae, S.W., Lee, I.: On finding a longest common palindromic subsequence. Theor Comput Sci **710**, 29–34 (2018). Advances in Algorithms and Combinatorics on Strings (Honoring 60th birthday for Prof. Costas S, Iliopoulos)
3. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. SIAM J. Comput. **46**(5), 1501–1514 (2017)
4. Barton, C., Kociumaka, T., Liu, C., Pissis, S.P., Radoszewski, J.: Indexing weighted sequences: neat and efficient. CoRR, arXiv:abs/1704.07625 (2017)
5. Belazzougui, D., Cunial, F.: Indexed matching statistics and shortest unique substrings. In: Moura, E., Crochemore, M. (eds.) SPIRE 2014. LNCS, vol. 8799, pp. 179–190. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11918-2_18
6. Chang, W.I., Lawler, E.L.: Sublinear approximate string matching and biological applications. Algorithmica **12**(4), 327–344 (1994)
7. Charalampopoulos, P., et al.: Linear-time algorithm for long LCF with K mismatches. In: CPM. LIPIcs, vol. 105, pp. 23:1–23:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)

8. Chi, L., Hui, K.: Color set size problem with applications to string matching. In: Apostolico, A., Crochemore, M., Galil, Z., Manber, U. (eds.) CPM 1992. LNCS, vol. 644, pp. 230–243. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-56024-6_19
9. Chowdhury, S.R., Hasan, M.M., Iqbal, S., Rahman, M.S.: Computing a longest common palindromic subsequence. Fundam. Inf. **129**(4), 329–340 (2014)
10. Dumitran, M., Manea, F., Nowotka, D.: On prefix/suffix-square free words. In: Iliopoulos, C., Puglisi, S., Yilmaz, E. (eds.) SPIRE 2015. LNCS, vol. 9309, pp. 54–66. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23826-5_6
11. Duval, J.-P., Kolpakov, R., Kucherov, G., Lecroq, T., Lefebvre, A.: Linear-time computation of local periods. Theor. Comput. Sci. **326**(1), 229–240 (2004)
12. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science (FOCS), pp. 137–143 (1997)
13. Farach, M., Muthukrishnan, S.: Perfect hashing for strings: formalization and algorithms. In: Hirschberg, D., Myers, G. (eds.) CPM 1996. LNCS, vol. 1075, pp. 130–140. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61258-0_11
14. Federico, M., Pisanti, N.: Suffix tree characterization of maximal motifs in biological sequences. Theor. Comput. Sci. **410**(43), 4391–4401 (2009)
15. Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
16. Inenaga, S., Hyyrö, H.: A hardness result and new algorithm for the longest common palindromic subsequence problem. Inf. Process. Lett. **129**, 11–15 (2018)
17. Inoue, T., Inenaga, S., Hyyrö, H., Bannai, H., Takeda, M.: Computing longest common square subsequences. In: 29th Symposium on Combinatorial Pattern Matching (CPM), LIPIcs, vol. 105, pp. 15:1–15:13 (2018)
18. Kociumaka, T., Starikovskaya, T., Vildhøj, H.W.: Sublinear space algorithms for the longest common substring problem. In: Schulz, A.S., Wagner, D. (eds.) ESA 2014. LNCS, vol. 8737, pp. 605–617. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44777-2_50
19. Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: 40th Symposium on Foundations of Comp Science, pp. 596–604 (1999)
20. Lothaire, M.: Applied Combinatorics on Words. Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge (2005)
21. Peterlongo, P., Pisanti, N., Boyer, F., do Lago, A.P., Sagot, M.: Lossless filter for multiple repetitions with hamming distance. J. Discr. Alg. **6**(3), 497–509 (2008)
22. Peterlongo, P., Pisanti, N., Boyer, F., Sagot, M.-F.: Lossless filter for finding long multiple approximate repetitions using a new data structure, the Bi-factor array. In: Consens, M., Navarro, G. (eds.) SPIRE 2005. LNCS, vol. 3772, pp. 179–190. Springer, Heidelberg (2005). https://doi.org/10.1007/11575832_20
23. Starikovskaya, T., Vildhøj, H.W.: Time-space trade-offs for the longest common substring problem. In: Fischer, J., Sanders, P. (eds.) CPM 2013. LNCS, vol. 7922, pp. 223–234. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38905-4_22
24. Thankachan, S.V., Aluru, C., Chockalingam, S.P., Aluru, S.: Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In: Raphael, B.J. (ed.) RECOMB 2018. LNCS, vol. 10812, pp. 211–224. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89929-9_14
25. Thankachan, S.V., Apostolico, A., Aluru, S.: A provably efficient algorithm for the k-mismatch average common substring problem. J. Comput. Biol. **23**(6), 472–482 (2016)