Locating maximal approximate runs in a string

Mika Amit, Maxime Crochemore, Gad Landau, Dina Sokol

Locating Maximal Approximate Runs in a String *

Mika Amit[1], Maxime Crochemore[3,4], Gad M. Landau[1,2], and Dina Sokol[5]

[1]Department of Computer Science, University of Haifa, Mount Carmel, Haifa, Israel, mika.amit2@gmail.com, landau@cs.haifa.ac.il
[2]Department of Computer Science and Engineering, NYU Polytechnic School of Engineering, New York University, Brooklyn, NY, USA
[3]King's College London, Strand, London WC2R 2LS, UK, maxime.crochemore@kcl.ac.uk
[4]Université Paris-Est, Institut Gaspard-Monge, 77454 Marne-la-Vallée Cedex 2, France
[5]Department of Computer and Information Science, Brooklyn College of the City University of New York, Brooklyn NY, USA, sokol@sci.brooklyn.cuny.edu

May 20, 2017

Abstract

An exact run in a string T is a non-empty substring of T that is a repetition of a smaller substring. Finding maximal exact runs in strings is an important problem and therefore a well-studied one in the area of stringology. For a given string T of length n, finding all maximal exact runs in the string can be done in $O(n \log n)$ time on general ordered alphabets or $O(n)$ time on integer alphabets. In this paper, we investigate the maximal approximate runs problem: for a given string T and a number k, find non-empty substrings T' of T such that changing at most k letters in T' transforms them into a maximal exact run. We present an $O(nk^2 \log^2 k + occ)$ algorithm to solve this problem, where occ is the number of substrings found.

Keywords: algorithms on strings, pattern matching, repetitions, tandem repeats, runs.

1 Introduction

Periodicities and repetitions are ubiquitous in nature, and they play a central role in the field of stringology. They are used to obtain efficient algorithms for pattern matching problems, to conserve space via text compression, and to better analyze data, e.g. in biological sequences. The research of repetitions and their characteristics has been thoroughly investigated for both exact and approximate ones.

Exact Repetitions: Several methods are available to detect all the occurrences of exact squares in strings, where a *square* is defined as exactly two consecutive copies of a pattern (see [7, 2, 26]). For a given string T, of length n, these algorithms run in $O(n \log n)$ time, which is optimal since

[*] A preliminary version appeared in the proceeding of CPM 2013.

it is possible for a string to contain $\Omega(n \log n)$ squares. Selecting some of their occurrences, or just *distinct* squares, regardless of their number of occurrences, paved the path to faster algorithms [22, 15] (for constant alphabets) and [4] (for integer alphabets).

Runs have been introduced by Iliopoulos, Moore, and Smyth [16], and are defined as repetitions with two or more consecutive copies of a pattern. They showed that Fibonacci words contain only a linear number of maximal runs. Kolpakov and Kucherov [20] (see also [9], Chapter 8) proved that this property holds for any string. In [3], the authors provided a simple and elegant proof that the number of maximal runs in a string of length n is at most $n - 3$. Recently, in [12], it was shown that for binary strings this number is bounded by $0.957n$.

In [20], the authors designed an algorithm to compute all maximal runs in a string of length n over an alphabet Σ. The time complexity of this algorithm is $\mathcal{O}(n \log |\Sigma|)$. Their algorithm extends Main's algorithm [25], which itself extends the method in [8] (see also [9]).

The design of a linear-time algorithm for building the Suffix Array of a string on an integer alphabet (see [17, 18, 19]) and the introduction of another related data structure (the Longest Previous Factor table in [10]) have eventually led to a linear-time solution (for integer alphabet) for computing all maximal runs in a string. This is a consequence of the linear-time computation of the Ziv-Lempel factorization on integer alphabets (see [1] and [6]), which removed the $O(n \log |\Sigma|)$ time bottleneck in the Kolpakov-Kucherov algorithms [20]. A recent algorithm by Bannai et al. [3] uses similar tools and also runs in linear time. On an ordered alphabet, namely where letters can be compared w.r.t. a linear order, the optimal computing time is $O(n \log n)$ [26, 11].

Approximate Repetitions: In many applications, finding *approximate* runs is more sensible than finding exact runs. A typical example is genetic sequence analysis. This problem was widely researched and many different measurements have been used in order to find such runs. In [28], a k-approximate run is defined as follows: a string x is an approximate run if there exists a consensus string u such that x can be divided into a number of adjacent occurrences of substrings $x = u_1 u_2 \cdots u_t$ where the distance between u and every u_i is not greater than k. In this version of the problem, the difference between two periods u_i and u_j can be as big as $2k$, for example: the string $x = bacd\ abdc\ cbad$ is a 2-approximate run, since the difference between the substring $u = abcd$ and each u_i, $1 \leq i \leq 3$ is exactly 2. [28] provide an $O(n^3)$-time algorithm for finding all such maximal repetitions in an input string of size n.

A different approach to the problem is defined as follows [14]: given a string x and an integer k, x is an approximate run if it can be divided into a number of adjacent substrings $x = u_1 u_2 \cdots u_t$, such that the sum over all distances between adjacent substrings, u_i and u_{i+1}, is not greater than k. In this version, the first period and the last period can be completely different from each other. For instance: the string $x = abcd\ dbcd\ dccd\ dcbd\ dcba$ is a 4-approximate run. [14] provide an $O(n^2)$-time algorithm for finding such maximal approximate runs in a string. This version of the problem can be extended to the problem where the sum over all distances between *every* two substrings u_i and u_j in x (for $1 \leq i < j \leq t$) cannot exceed k. In this case, all substrings u_i, $1 \leq i \leq t$ must be similar to each other, as one error between two substrings u_i and u_j may imply $O(t)$ errors . For example, the string $x = (abcd)^{t-1}abed$ is a $(t-1)$-approximate run according to this definition.

In [23] another definition of approximate run is given: a substring x is a k-approximate run if $x = u_1 u_2 \cdots u_t$ and the removal of the same k positions from each u_i will generate an exact run. According to this definition, any number of mismatches in the same column of the period is counted as 1 mismatch. For example, the string $x = abcd\ abdd\ abbd\ abad$ is a 1-approximate run. The algorithm for finding such repetitions [23] has time complexity $O(nka\log(n/k))$, where n is the length of the input string, k is the number of allowed error columns, and a is the maximum number of periods in any found repeat.

In this paper we introduce a novel, more global definition of an approximate run. Informally, in our problem we count the total number of letters that need to be replaced in order to generate an exact run. A k-approximate run can be transformed into an exact run through the modification of at most k letters. This definition is similar to the one presented in [28], as it finds a consensus string u that is similar to all substrings u_i of x. But, as opposed to the former version, this version sums the *total* number of differences between all u_i and u, which requires the substrings to be more similar to each other. The formal definition of the problem is given in Section 2. For example, the substring $x = bacd\ abdc\ cbad$, that contains periods that are very different from each other, is a 2-approximate run according to the former definition, and in our problem definition, it is a 6-approximate run. Note that the substring $x = abcd\ aadd\ abcd\ abcd$ is a 2-approximate run according to both definitions. In this paper we present an $O(nk^2\log^2 k + occ)$-time algorithm to find such maximal approximate runs in a given input string of length n, where occ is the number of maximal approximate runs that are found.

Roadmap: We start in Section 2 with definitions and notations that will be used throughout the paper. In Section 3, we present the main procedure of our algorithm. Initially, in subsection 3.1, we describe a simple $O(n)$ algorithm for the main procedure, and then in subsection 3.4 we present an improved $O(k^3)$ algorithm for it. In Section 4, we describe the efficient $O(k^2\log k)$-time algorithm for the main procedure. Finally, in Section 5, we present the entire algorithm for searching a given input string of length n for maximal approximate runs with k modifications.

2 Definitions and Notation

Let $T = T[1]T[2]\cdots T[n]$ be a string of size n defined over the constant size alphabet Σ. We denote the substring of T that starts at position i and ends at position j as $T_{i,j} = T[i]T[i+1]\cdots T[j]$. A position h is *contained* in the substring $T_{i,j}$ if $i \leq h \leq j$. The following definitions are needed in order to formally define the problem we solve in the paper.

Exact Run. An *exact run* is a non-empty string, x, that can be written as $x = u_1u^tu_2$, where $t \geq 2$, the first substring u_1 is a (possibly empty) suffix of u, and the last substring u_2 is a (possibly empty) prefix of u. u is called a *period* and its length is denoted as the *period length*, $p = |u|$. Each position i in u ($1 \leq i \leq p$) is called a *column* of u. The exponent of the run is of size $\frac{|x|}{p} \geq 2$. For instance, *abababababa*, has exact runs with period length 2 ($u = ab$) and 4 ($u = abab$). Their exponent are 4.5 and 2.25, respectively. For both exact runs we have u_1 is the empty string, and u_2 is the letter a.

For simplicity, we set the first column of the period to be position 1 in the text. This means that an exact run, $x = u_1u^tu_2$, can start at any column of the period, and for period length p, the column of index i, denoted by `Column(i)`, is equal to $i\ mod\ p$ (with the exception of the case where $i\ mod\ p \equiv 0$, in which `Column(i)` is p).

Maximal Exact Run. A *maximal exact run* is an exact run, x, that is a substring of a longer string, T, such that x cannot be extended in T either to the right or left. For instance, *dababababac*, has a maximal exact run starting at position 2 with period length 2 and exponent 3.5. If a string T contains a maximal exact run starting at index i, it means that either $i = 1$ or $T[i-1] \neq T[i+p-1]$, for otherwise, the exact run is not maximally extended. Similarly, if the maximal exact run ends at position j, then either $j = n$ or $T[j+1] \neq T[j+1-p]$.

k-Maximal Approximate Run (k-MAR). A *k-MAR* is a non-empty substring S of T, such that the modification of at most k letters in S generates a maximal exact run. We denote the positions of these letters as *modified positions*.

For instance, let $T = haabaabcabaabcd$. For $p = 3$ and $k = 1$, the string has four 1-maximal approximate runs. One starts at location 2 and ends at location 13, with exponent 4. In this example, the letter in position 8 is a *modified* position, since modifying it from c to a generates a maximal exact run. The maximal approximate run is $T_{2,13} = u_1u^3u_2$, where $u = baa$, $u_1 = aa$ and $u_2 = b$. The second 1-MAR is $T_{6,14} = u_1u^2u_2$, with $u = bca$, $u_1 = a$, $u_2 = bc$ and exponent 3. Here, the letter in position 11 is a *modified* position, since modifying it from a to c generates a maximal exact run. The third 1-MAR is the substring $T_{1,7} = u^2u_2$, with approximate period $u = baa$ and $u_2 = b$. Here, position 1 is a *modified* position. The fourth 1-MAR is the substring $T_{3,10} = u_1u^2u_2$, with approximate period $u = bca$, $u_1 = a$ and $u_2 = b$. Here, position 5 is a *modified* position. Observe that $T_{3,10}$ is contained entirely in the 1-MAR $T_{2,13}$.

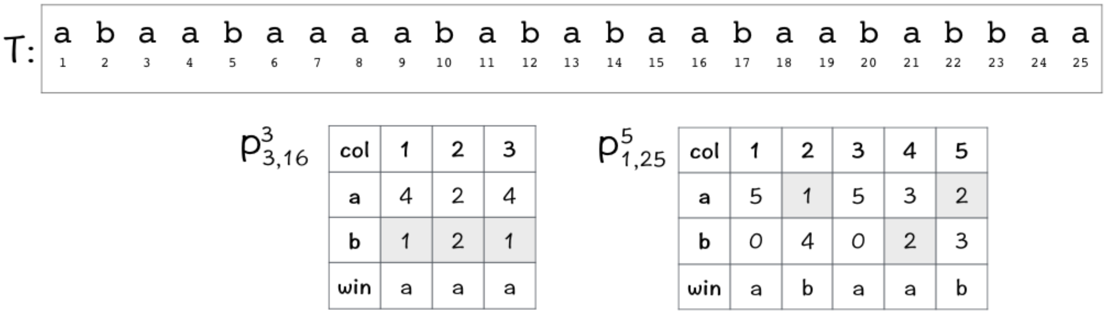
Contained k-MAR. Let $T_{i,j}$ and $T_{i',j'}$ be two k-MARs with period length p. $T_{i,j}$ is *contained* in $T_{i',j'}$ if and only if $i' \leq i \leq j \leq j'$. For example, let $T = habcbcbababade$. For $p = 2$ the string contains two 2-maximal approximate runs from location 2 to 9 (where modifying the a to c at positions 2 and 8, generates a maximal exact run of cb with exponent 4) and from location 2 to 10 (where modifying the c to a at positions 4 and 6, generates a maximal exact run of ab with exponent 4.5). The 2-MAR $T_{2,9}$ is contained in the 2-MAR $T_{2,10}$. For a given period length p, our algorithm reports only k-MARs that are not contained in another k-MAR.

We are now ready to present the formal definition of our problem:

The k-Maximal Approximate Runs Problem:
Given a string T of size n defined over the alphabet Σ, and a number k, find all k-MARs in the string T of all period lengths p, $1 \leq p \leq \frac{n}{2}$. In the case where a k-MAR is contained in another k-MAR, only the containing k-MAR is reported.

We continue with a definition of Parikh matrix (see also [27]) that will be used throughout the paper to count the number of modified positions in a substring with respect to a period length. A *Parikh matrix* , $P^p_{i,j} = P[1..|\Sigma|, 1..p]$, is a two-dimensional array defined over a substring $T_{i,j}$ and a period length p. An entry $P^p_{i,j}[let, col]$ contains the number of occurrences of $let \in \Sigma$ in column col of p in $T_{i,j}$. In addition, for each column, col, we keep an additional variable $win(col)$. The variable $win(col)$ contains the *winner* letter - the letter that occurs more than any other letter in this column (in case of a tie, the variable win is arbitrarily set to one of the winners).

We use the Parikh matrix, $P^p_{i,j}$, in order to count the number of modified positions in the substring $T_{i,j}$ with regards to a period length p. We say that *col contains* modified positions if the sum over all letters that are not the winner letter in *col* is greater than 0. i.e. the number of modified



T:	a	b	a	a	b	a	a	a	a	b	a	b	a	b	a	a	b	a	a	b	a	b	b	a	a
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

$P^3_{3,16}$

col	1	2	3
a	4	2	4
b	1	2	1
win	a	a	a

$P^5_{1,25}$

col	1	2	3	4	5
a	5	1	5	3	2
b	0	4	0	2	3
win	a	b	a	a	b

Figure 1: Examples of Parikh matrices. $P^3_{3,16}$ contains 4 modified positions with approximate period *aaa*. Note that in column 2 the winner letter is chosen arbitrarily (we can also have that $u = aba$). $P^5_{1,25}$ contains 5 modified positions, with approximate period *abaab*.

positions that *col* contains is equal to $\Sigma_{let \neq win(col)} P^p_{i,j}[let, col]$. Similarly, a substring, $T_{i,j}$, *contains* modified positions in p if the sum of modified positions over all its columns is greater than 0.

In the following, when it is clear from the context, we omit the period length p, when referring to modified letters with respect to the period length.

Figure 1 shows two examples of Parikh matrices computed for two substrings of T and two period lengths.

3 The Main Procedure

The main framework of our algorithm uses the ideas of Kolpakov and Kucherov [21] for finding maximal approximate runs. Similar methods were used in [26, 20, 23]. As in [21], our algorithm generates a set of calls to the main procedure. Each call contains a substring T, an anchor position, *anc*, and a period length, p. The procedure reports the k-MARs in T that contain position *anc* and have period length p. In Section 5 the outline of the entire algorithm is described.

The procedure can be formally described as follows.

Input: (a) string T of length n. (b) anchor position, *anc*, $1 \leq anc < n$. (c) period length, p.

Output: All k-MARs, $T' = T_{i,j}$, such that T' is a k-MAR with period length p, and $i \leq anc < j$.

We start with a high level description of the main procedure. Let $T_{i,j}$ be a k-MAR. The Parikh matrix $P^p_{i,j}$ contains exactly k modified positions. In addition, from the maximality of k-MARs, we have that if we extend the substring to the right or to the left by one letter, the number of modified positions in the substring exceeds k.

In the main procedure we keep two pointers to the string, ℓ and r, which represent possible leftmost and rightmost boundary of a k-MAR, respectively. For convenience reasons, that will become clearer in the following sections, we always set ℓ to one position before a possible k-MAR and r to one position after it. Thus, the pointers ℓ and r are initially set to $i - 1$ and $j + 1$, respectively.

The procedure works in iterations, each iteration starts with a k-MAR, $T_{\ell+1,r-1}$, and ends with a k-MAR. At the beginning of the iteration, since r cannot be increased without exceeding the number of allowed modified positions, we start with increasing ℓ and stop only when r can be increased. Then, we increase r for as long as $T_{\ell+1,r-1}$ contains at most k modified positions. When r is stopped, the substring $T_{\ell+1,r-1}$ is a new k-MAR, and it is reported ($T_{\ell+1,r-1}$ is reported only if its length is greater than or equal to $2p$).

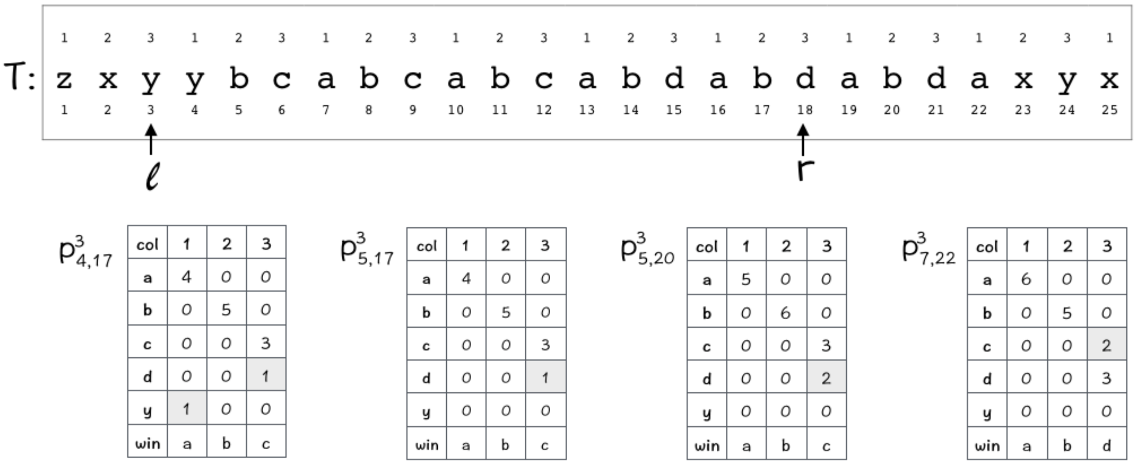
Assume that the leftmost k-MAR in T was found, T^{first}, and that its Parikh matrix is given (this initial step is described in Subsection 4.5). We set ℓ (r) to one position before (after) T^{first}. The main procedure continues until either ℓ is equal to *anc* or r is greater than n. For simplicity of presentation, assume that T is not a k-MAR (the case where the entire string is a k-MAR is easily computed in $O(n)$ time).

We now proceed with a detailed description of the pointers move. We start with the move of ℓ. At first, $T_{\ell+1,r-1}$ contains exactly k modified positions, and since moving r to the right requires another modified position, we start with moving ℓ. This move contains two steps: first, we move the pointer, and then we update the Parikh matrix and decide whether the next move will be of ℓ or r. Let $x = T[\ell]$, and let *col* be its column. Clearly, the number of occurrences of x in *col* was decreased by 1 in the Parikh matrix. We describe three cases (see Figure 2):

- **continue with ℓ:** The letter x was the only winner in *col* before the contraction, and stays the only winner after it. This contraction does not change the number of modified positions, and therefore we continue with moving ℓ to the right.
- **move to r:** The letter x was either a losing letter in *col* or it was one of the winners before the contraction. In both cases, x is a loser in the contracted substring. Clearly, this contraction releases a modified position. Therefore, ℓ is stopped and r pointer is increased.
- **special case:** The letter x was the only winner before the contraction, and after the contraction there is a tie between x and (at least one) other letter in the contracted substring. Although this contraction does not reduce the number of modified positions, there might be a case where r can be moved without using additional modified positions: this will happen if after the contraction $y = T[r]$ is one of the winners in *col*. If this is the case, ℓ is stopped and r pointer is increased.

After ℓ is stopped, we move to r. Now, $T_{\ell+1,r-1}$ contains either $k-1$ or k (if ℓ was stopped on the special case) modified positions. As described above, r will be moved as long as the substring $T_{\ell+1,r-1}$ contains at most k modified positions. The move of r contains two steps: we first check whether $T[r]$ can be added to the substring without exceeding the maximum number of allowed modified positions. If so, the substring is *extended* to include $T[r]$ and r is further increased. Else, we report the found k-MAR, r is stopped and we continue to the next iteration to move ℓ. Let $y = T[r]$, and let *col* be its column. Clearly, if r is increased, the number of occurrences of y in *col* is increased by one in the Parikh matrix. We describe the following cases (see Figure 2):

- **free move of r:** The letter y is the only winner or one of the winners in *col* before the extension. This means that in the extended substring, y is the only winner in its column. This extension does not use a modified position, and therefore we can continue with moving r to the right.
- **r move uses a modified position:** The letter y is a losing letter before the extension. After the extension, y either stays a losing letter, or is one of more than one winners in *col*.



$P^3_{4,17}$

col	1	2	3
a	4	0	0
b	0	5	0
c	0	0	3
d	0	0	1
y	1	0	0
win	a	b	c

$P^3_{5,17}$

col	1	2	3
a	4	0	0
b	0	5	0
c	0	0	3
d	0	0	1
y	0	0	0
win	a	b	c

$P^3_{5,20}$

col	1	2	3
a	5	0	0
b	0	6	0
c	0	0	3
d	0	0	2
y	0	0	0
win	a	b	c

$P^3_{7,22}$

col	1	2	3
a	6	0	0
b	0	5	0
c	0	0	2
d	0	0	3
y	0	0	0
win	a	b	d

Figure 2: Examples of ℓ and r moves. Let $p = 3$ and $k = 2$.
$T_{4,17}$ is a 2-MAR with period *abc*. We start with moving ℓ from position 3 to 4. Since $y = T[4]$ is a losing letter in column 1, this move releases a modified position and ℓ is stopped. We continue with a move of r. We first check whether $d = T[18]$ can be added to the substring. The letter d is a losing letter in column 3, and since there is only 1 modified position in $T_{5,17}$, we can increase r from 18 to 19. We proceed with moving r pointer. Both a and b are winners in columns 1 and 2, respectively, therefore r is increased to position 21. Now, the 2-MAR $T_{5,20}$ is reported, and r is stopped.
The letter $b = T[\ell + 1] = T[5]$ is the only winner in column 2, therefore, the move of ℓ from 4 to 5 does not reduce the number of modified positions. We continue with ℓ, that moves to position 6. Now, c becomes one of the winners in column 3, and since $d = T[r] = T[21]$ becomes one of the winners in column 3, ℓ is stopped and we continue with moving r until position 23. The 2-MAR $T_{7,22}$ is reported.

Clearly, this extension uses a modified position, and hence the move of r depends on the number of modified positions in the substring before the extension: the number is either (a) $k - 1$ or (b) k. In case (a), the substring can be extended to contain $T[r]$, and we continue with r moves. In case (b), if r is further increased, the number of modified positions exceeds the maximum allowed, and therefore r is stopped and the new k-MAR, $T_{\ell+1,r-1}$, is reported. Now, we continue to the next iteration with ℓ move.

- **special case:** As in the previous case, the move of r uses a modified position. In the special case, the number of modified positions in the substring before the extension is exactly $k - 1$. Therefore, we continue with moving r to the right.

In the following we present three algorithms for the main procedure.

3.1 A Simple $O(n)$ Main Procedure

In the simple implementation, we keep a Parikh matrix, P, of size $(|\Sigma| + 1) \times p$. The pointers ℓ and r are always increased by 1. Each increase of the pointers is followed by an update of the Parikh matrix. A run example of the simple procedure is demonstrated in Figure 2.

Time Complexity: The algorithm visits every position in $[1..n]$ at most once. Assuming that $|\Sigma|$ is constant, each such visit consists of a constant time update to the Parikh matrix (a discussion on larger alphabet size is presented in Subsection 4.9). This gives a total time of $O(n)$.

3.2 Definitions and Observations

In the following subsections, we present improved algorithms for the main procedure. For that, we need some definitions and observations.

Let T be the input string, and let $T_{i,j}$ be a k-MAR containing *anc*. In the improved implementation of the main procedure we take advantage of the fact that not all the positions in T need to be visited. Assume that there exists a column *col* ($1 \leq col \leq p$) that contains the same letter in all its positions in T. This means that modified positions will never be used in this column in all k-MARs $T_{i,j}$, and therefore its positions need not be visited by the algorithm. We proceed with defining the actual positions that need to be visited.

Mismatch. When two letters $T[i]$ and $T[i+p]$ are not identical, we say that there is a *mismatch* between positions i and $i+p$ in T.

In a k-MAR, $T_{i,j}$, a *mismatch* and a *modified position* (defined in Section 2) have a strong relation, but are not always identical. Given a string T, the mismatch positions with respect to a period length p are permanent, whereas the modified positions depend only on $T_{i,j}$ that is now being inspected.

In the following example, the number of mismatches and modified positions in a specific substring is the same. This relation between the terms is not always straightforward, as we will describe shortly.

Example 1 (Same number of mismatches and modified positions in a k-MAR)**.** *Let* $k = 2$ *and* $T =$ *abc abd abd abc abc. There are two mismatches between positions* 3 *and* 6 *and between positions* 9 *and* 12 *(between c and d). The positions* 6 *and* 9 *are modified positions in* T*, where the letters are modified to c in order to generate a maximal exact run of* $u =$ *abc.*

Problematic Column in T. Let i and $i+p$ be positions of a mismatch (i.e. $T[i] \neq T[i+p]$). Then, the column of i in p is denoted as a *problematic column* in T. For example, in Figure 2, column 1 contains a mismatch between positions 4 and 7. Therefore, column 1 is a *problematic column* in T.

We continue with some observations regarding the relation between the terms. Each observation is followed by an example. In the examples assume that $p = 3$.

Observation 1. *For all* $T_{i,j}$ *in* T*, the modified positions in* $T_{i,j}$ *can be only in positions of problematic columns. However, since the problematic columns are defined in* T*, for a specific k-MAR,* $T_{i,j}$*, it is possible that there will be no modified positions in the positions of a problematic column (this will happen when there are no mismatches in* $T_{i,j}$*).*

Example 2. *In Figure 2, in the* 2*-MAR* $T_{5,20}$ *there are no modified positions in column* 1*, although it is a problematic column in* T*.*

Observation 2 (One mismatch implies more than one modified position in a k-MAR)**.** *In a* k*-MAR* $T_{i,j}$*, one mismatch can imply at most* $\frac{|T_{i,j}|}{2p}$ *modified positions.*

Example 3. *In the* 3*-MAR of* $T =$ *abc abc abc abd abd abd abd, there is only one mismatch between positions* 9 *and* 12 *and there are three modified positions at positions* $3, 6$ *and* 9*.*

Observation 3 (Two mismatches might imply one modified position in a k-MAR)**.** *In a* k*-MAR,* $T_{i,j}$*, it is possible for two mismatches to imply one modified position.*

Example 4. *Let* $k = 1$ *and* $T =$ *aba aba abc aba aba. There are two mismatches between positions* 6 *and* 9 *and between positions* 9 *and* 12*. Here, there is one modified position in* 9*, since modifying the c to a generates a maximal exact run of* $u =$ *aba.*

Observation 4 (At most $2k$ mismatches in a k-MAR)**.** *In a* k*-MAR,* $T_{i,j}$*, there are at most* $2k$ *mismatches such that* $T[h] \neq T[h + p]$*, for* $i \leq h \leq j - p$*.*

Following Observation 4, if $T_{i,j}$ is the leftmost k-MAR that contains *anc*, then the substring $T_{i,anc}$ includes at most $2k$ mismatches. Similarly, if $T_{i,j}$ is the rightmost k-MAR that contains *anc*, then the substring $T_{anc,j}$ includes at most $2k$ mismatches.

Observation 5 ($O(k)$ problematic columns in T)**.** *For all* $T_{i,j}$ *in* T *containing anc, there are at most* $O(k)$ *columns that can contain modified positions.*

Marking Mismatches. A mismatch occurs between two positions i and $i + p$, and we want to mark one of the two positions as the mismatch position. We divide the mismatches into three groups according to the positions i, $i+p$, and *anc*. When i is to the right of *anc*, we mark position $i + p$ as the mismatch position, and denote the mismatch by m^r. When $i + p$ is to the left of *anc*, we mark position i as the mismatch position, and denote the mismatch by m^ℓ. In the special case where $i \leq anc \leq i + p$, we do not mark either of its positions. Note that even though we don't mark such mismatches, their corresponding column is a problematic column (see Subsection 3.3).

In a substring $T_{i,j}$, we enumerate the mismatches starting from *anc*. i.e., the mismatches to the left of *anc* are enumerated from right to left, and the mismatches to the right of *anc* are enumerated from left to right (see Figure 4).

Example 5. *Let* $T =$ *abc abc abd abd abe abc abd abd abc,* $p = 3$*, and* $anc = 14$*. The i position of the mismatches in* T *are* $\{6, 12, 15, 18, 24\}$*. We mark the mismatches at* $\{m^\ell_1 = 6, m^r_1 = 18, m^r_2 = 21, m^r_3 = 27\}$*.*

Zone. We divide the string T into adjacent periodic substrings according to the mismatch positions in T and *anc*. We call each such substring a *zone*. Following the symmetric way we mark mismatches, we define zones (see Figure 3). The mismatches to the left of *anc* define the ℓ-zones: an ℓ-zone starts one position to the right of a mismatch and ends at a mismatch (i.e., for two mismatches, m^ℓ_j and m^ℓ_{j+1}, the substring of T starting at position $m^\ell_{j+1} + 1$ and ending at position m^ℓ_j is an ℓ-zone). The mismatches to the right of *anc* define the r-zones: an r-zone starts at a mismatch position and ends one position to the left of a mismatch (i.e., for two mismatches, m_j^r and m_{j+1}^r, the substring of T starting at position m_j^r and ending at position $m_{j+1}^r - 1$ is an r-zone). Finally, we define two additional zones that overlap *anc*. An ℓ-zone $T[m_1^\ell + 1..anc]$ and an r-zone $T[anc..m_1^r - 1]$.

For a *zone*, T_{ℓ_z,r_z}, although there are no mismatches in it, it is not necessarily a maximal exact run: on the one hand, it can be smaller than $2p$, and on the other hand, following the definitions of zones and mismatches, if it is an ℓ-zone, it can be extended to the right by at most $p-1$ positions, or, if it is an r-zone, it can be extended to the left by at most $p-1$ positions.

In Example 5, the ℓ-zones are: $T_{1,6}$ and $T_{7,14}$, and the r-zones are: $T_{14,17}$, $T_{18,20}$, $T_{21,26}$, and $T[27]$.

Our algorithm will look for the k-MARs that start in a specific ℓ-zone and end in a specific r-zone in T, considering each viable pair of such zones. The following lemma defines the relation between zones and the possible leftmost and rightmost positions of a k-MAR in them.

Lemma 1. *Let $z = T_{\ell_z,r_z}$ be a zone, and let $T_{i,j}$ be a k-MAR. If $T_{i,j}$ starts in z then* ***either*** *the entire zone* ***or*** *less than $k+1$ of its rightmost periods are contained in $T_{i,j}$. Symmetrically, if $T_{i,j}$ ends in z then* ***either*** *the entire zone* ***or*** *less than $k+1$ of its leftmost periods are contained in $T_{i,j}$.*

Proof. We prove the case where $T_{i,j}$ starts with z, i.e., z is the leftmost ℓ-zone in the k-MAR. The proof for the case where $T_{i,j}$ ends in z is symmetric. Cleary, unless the entire string T is periodic, a k-MAR cannot start and end in the same zone. Also, note that if $|z|$ is smaller than $(k+1)p$, the lemma is trivially true.

Assume z contains more than $k+1$ periods. We prove the lemma by contradiction.

Assume that the leftmost position of the k-MAR is a position i in z, and that this position is not its first position, ℓ_z, and is to the left of its rightmost $k+1$ periods, namely $\ell_z < i < r_z - (k+1)p$. We consider two cases regarding the problematic columns in this zone.

Case 1: There exists at least one letter in z that is a non-winner letter in its column - since all periods in a zone are equal, this means that the number of modified positions in $T_{i,j}$ exceeds k, a contradiction.

Case 2: All letters in z are winners of their columns - since all periods in a zone are equal, and since all letters in z are winners, the k-MAR must start at position ℓ_z. Contradicting the fact that the k-MAR is maximally extended. □

(k+1)-periods. Denote the $k+1$ rightmost (leftmost) periods of an ℓ-zone (r-zone) as the $(k+1)$-*periods* of the zone.

We start with the procedure that finds the mismatch positions in T. The mismatches are used to define the zones and the problematic columns in T in both the $O(k^3)$-time and the $O(k^2 \log k)$-time improved algorithms.

3.3 Finding the Mismatches.

Recall that there are three groups of mismatches that we want to find: mismatches that are entirely to the right of *anc*, mismatches that are entirely to the left of *anc* and mismatches that overlap position *anc*. We use the technique described in [23], using [24] and [13], in order to find these mismatches. the output of the procedure is two lists of mismatch positions ($RightMismatches$ and $LeftMismatches$) and a set of problematic columns ($ProblematicColumns$).

First, for the mismatches to the right of anc, we want to find all mismatches $T[i] \neq T[i+p]$ such that $i \geq anc$. For that, we construct a suffix tree of T, and use the "kangaroo" jumps of [13] to find the $2k+1$ mismatch positions in T starting at anc going to the right (i.e., using suffix trees and LCA algorithm for a constant time "jump" over equal substrings of the aligned copies of T). These mismatches are marked by their right position, $i+p$, and are saved in increasing order to the auxiliary list, $RightMismatches = \{m_1^r, m_2^r, ..., m_{2k+1}^r\}$. In addition, for each mismatch, its column is added to the set $ProblematicColumns$. Second, for the mismatches to the left of anc, we construct a suffix tree of the reversed string of T, T^R, and again use the "kangaroo" jumps of [13] to find the $2k+1$ mismatch positions starting from anc going to the left. These mismatches are marked by their left position, i, and are saved in decreasing order to the auxiliary list $LeftMismatches = \{m_1^\ell, m_2^\ell, ..., m_{2k+1}^\ell\}$. In addition, for each such mismatch, its column is added to the set $ProblematicColumns$.

Finally, for the mismatches $T[i] \neq T[i+p]$, such that $i \leq anc \leq i+p$, we use the suffix tree of T and the "kangaroo" jumps in order to find $2k+1$ mismatch positions in the substring $T_{anc-p,anc+p}$. These mismatches are not marked, but their columns are added to the set $ProblematicColumns$.

Clearly, if the substring contains less than $2k+1$ mismatches to the left (right) of anc, the corresponding mismatch list will contain less than $2k+1$ positions. Furthermore, if there are more than $2k+1$ mismatches $T[i] \neq T[i+p]$ such that $i \leq anc \leq i+p$, only $2k+1$ positions are found and are saved to the $ProblematicColumns$ set (this actually means that the rightmost k-MAR that contains anc must end at the leftmost r-zone).

Time Complexity: The suffix trees of T and T^R are built and preprocessed for answering LCA queries once for the entire algorithm in linear time. Therefore, we add $O(n)$ to the total time complexity of the algorithm. Finding the $O(k)$ mismatches with respect to anc is done in $O(k)$ time.

We are now ready to present the $O(k^3)$-time algorithm. We use the definitions from above and Lemma 1 to define the positions that both ℓ and r pointers visit.

3.4 An Improved $O(k^3)$ Main Procedure

Following Observation 1, since modified positions are only used on problematic columns, the Parikh matrix can contain only the $O(k)$ problematic columns of T. Recall that in our implementation, we always set the pointer ℓ (r) to a position that is one position to the left (right) of a possible k-MAR. Therefore, following Lemma 1, ℓ (r) should only point to positions within the $(k+1)$-periods of a zone.

In this implementation, we define two different kinds of a pointer move: a move inside the $(k+1)$-periods of the zone, and a move between zones. Let L denote the ℓ-zone in which ℓ is currently contained, and R be the r-zone of r (see Figure 3).

A move inside the $(k+1)$-periods of the zone. When a pointer moves inside the $(k+1)$-periods of a zone, it only visits positions of problematic columns: the pointer is moved from one problematic column to the next problematic column to its right.

In a similar way to the simple algorithm, on each such move the procedure updates the Parikh matrix in the respected problematic column in constant time.

A move between zones. We distinguish between the move of r and of ℓ between zones:

A move of r. The r pointer is moved from the last problematic column in the $(k+1)$-periods of an r-zone, R', to the first position of the r-zone to its right, R. This move requires an update of the Parikh matrix. If the size of R' is smaller than $(k+1)p$, we only need to update the value in one column. Otherwise, all letters in R' are winners in their column, and this move can result in an update of $O(k)$ columns. Each update is done in $O(1)$ time since a zone is periodic, the alphabet size is constant, and the size of R' is known in advanced.

A move of ℓ. The move of ℓ between zones starts when ℓ is moved from the last position of an ℓ-zone to a position of the ℓ-zone to its right, L. The previous position of ℓ is of a problematic column, by definition. Observe that the case when the entire L zone is contained in a k-MAR was already reported by the algorithm.

Assume that the size of L is greater than $(k+1)p$ (otherwise this move is treated as a move inside the $(k+1)$-periods of a zone). We want to find a new position for ℓ in L: we first set ℓ to the leftmost position of a problematic column in the $(k+1)$-periods of L. Then, we update the Parikh matrix. This move might increase ℓ by more than one problematic column, and therefore the Parikh matrix is updated in all its columns. Again, since L is periodic, the alphabet size is constant, the size of L is known in advance, and the Parikh matrix of $T_{\ell,r}$ for the previous ℓ is already computed at this point, this update can be done in $O(k)$ time.

Now, we keep moving ℓ from one problematic column to the one on its right, until the number of modified positions in $T_{\ell+1,r-1}$ is exactly k.

Total Time Complexity: For each zone, moving to the zone takes $O(k)$ time to update the Parikh matrix. Then, the pointers visit $O(k^2)$ positions of problematic columns in the $(k+1)$-periods of the zone. Each such visit updates the Parikh matrix in one column, therefore takes $O(1)$ time, and therefore, takes $O(k^2)$ time per zone. There are $O(k)$ zones in T. Therefore, the algorithm that finds the k-MARs in T takes $O(k^3)$ time.

4 The Efficient $O(k^2 \log k)$ Main Procedure

In the efficient algorithm we further decrease the number of visited positions in each zone. Observe that for a specific pair of ℓ-zone and r-zone, there might be *problematic columns* that we do not need to visit. If, for instance, r-zone contains a letter, y, that is a winner in $T_{\ell+1,r-1}$, then no matter where the r pointer is positioned in this r-zone, y stays the winner. Therefore, there is no need to visit its positions in this r-zone. In general, only the positions of letters that are losers or might lose their majority are of interest. We continue with defining these positions. First, we introduce the following notations.

Consider the substring $T_{\ell+1,r-1}$ ($\ell < anc < r$). Denote the *ℓ-zone* in which ℓ is contained as L and the *r-zone* in which r is contained as R. Let m_i^ℓ be the mismatch position of L zone (m_i^ℓ is the rightmost position in L), and let m_j^r be the mismatch position in the R zone (m_j^r is the leftmost

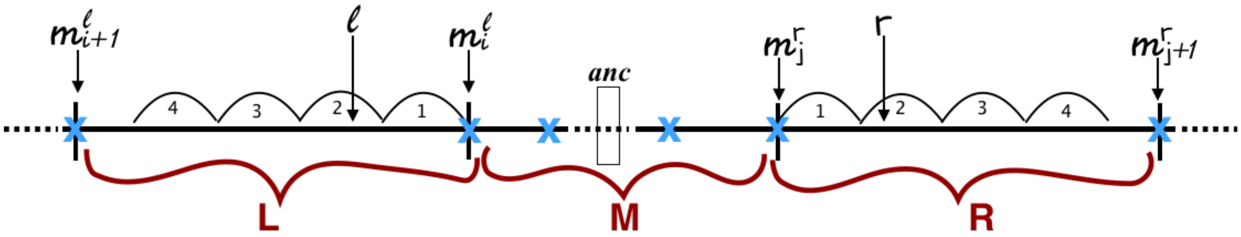


Figure 3: L, M and R substrings, and the positions ℓ and r in them. The mismatch positions are marked with X. For $k = 3$, the relevant periods in L and R zones are marked with brackets and are numbered from 1 to 4.

position in R). We also denote the middle substring, $T_{m^{\ell}_i+1,m^{r}_j-1}$, between L and R as M (see Figures 3 and 4).

For a specific pair of L and R zones, the positions of a problematic column, *col*, are visited by the algorithm if one of the following cases apply:

- The letter in L and the letter in R are different. This means that at least one of the letters is a losing letter.
- The letter in L equals the letter in R, and it is the losing letter in $T_{\ell+1,r-1}$.
- The letter in L, x, equals the letter in R, and it is the winner in $T_{\ell+1,r-1}$. Let w ($w \neq x$) be the letter with the maximal number of occurrences in M. Denote this number by $|w|$. If $|w| > |x|$ in the substring $T_{m^{\ell}_i+1,r-1}$, then x might lose its majority to w, as ℓ increases.

Furthermore, a problematic column that is visited in L zone might not be visited in R zone, and vice versa. This depends on the majority of the letter in L (or in R) in its column. The decision whether to visit a column in L or in R is thoroughly described in subsection 4.2. The columns that should be visited in L and R are saved in auxiliary lists (*LeftList* and *RightList*, respectively, see definitions in subsection 4.1), and are updated as the pointers increase.

In addition, consider the case where a letter in L is initially a winner in its column, but as ℓ increases, the number of its occurrences decreases, and it becomes a loser. In the efficient algorithm we visit its positions only when the letter becomes a loser. Therefore, additional methodological stops are added whenever ℓ moves between periods in the same zone. On each such stop positions that need to be visited starting this current period are added to *LeftList*.

The improved main procedure can be roughly divided into the following three sub-procedures, described in detail in the ensuing subsections:

Moving between zones. This sub-procedure is called whenever ℓ or r is moves from one zone to the next zone to its right. It is responsible for computing the first position of ℓ in the new zone, for updating the Parikh matrix according to the new position, and for updating auxiliary lists with the relevant problematic columns. It is explained in subsection 4.2.

Moving between periods. This sub-procedure is called whenever ℓ is moved from one period to the next period to its right. It is responsible for computing the first position of ℓ in the new period, for updating the Parikh matrix according to the new position, and for adding new relevant problematic columns that need to be visited in the L zone, starting the new period. It is described in 4.3.

Moving between positions (within a period). This is the basic step that updates the Parikh matrix according to the new position of ℓ or r. It is described in 4.4.

The rest of this section is organized as follows. We start in 4.1 with a description of the data structures used in the algorithm. We continue with a description of each one of the sub-procedures for ℓ and r moves. In Subsection 4.5 we describe the procedure that finds the leftmost k-MAR that contains *anc*. In Subsection 4.6 a running example of the algorithm is presented and in Subsection 4.7 we provide the pseudocode for the efficient algorithm. Then, in Subsection 4.8 we prove Lemma 2, that bounds the number of visited positions in every zone. Finally, in Subsection 4.9, we analyze the total time complexity of the main procedure.

4.1 Data Structures

We start with describing the data structures used in the main procedure. As mentioned above, we use the Parikh matrix in order to count the number of modified letters in the substring $T_{\ell+1,r-1}$. The Parikh matrix contains only $O(k)$ columns, which are the problematic columns in T.

In addition, the following auxiliary lists are used throughout the algorithm. *LeftList* and *RightList* are sorted lists of problematic columns that need to be visited in L and R, respectively. Both lists contain a subset of the problematic columns in T. Each list is implemented as a balanced search tree, where the keys to the records are column positions. These lists are both initialized whenever ℓ or r is moved to a new zone, and in addition, are updated as ℓ and r increase.

Observe that for a problematic column in L, there might be a situation in which the letter in L starts as the only winner of its column, and as ℓ and r increase, the letter becomes a loser in its column. We want to visit this column in L only in the positions where the letter becomes a loser (or one of more than one winners in the column). Therefore, we need to add this problematic column to *LeftList* only when it is relevant.

In order to support this kind of update, an additional data structure is used. For *each* period i in the $(k+1)$-periods of L, we keep a list, $newColumnList_i$, that contains problematic columns that need to be added to *LeftList* starting period i. Each $newColumnList_i$ is a doubly linked list. We keep an array of $O(k)$ pointers, such that for every problematic column in T, we have a pointer to the specific list that it is contained in, or $Null$ if it is not contained in any *newColumnList*. Then, we can find the list a column is contained in, and update the lists (insert and delete columns) in constant time. This data structure is further discussed in Subsections 4.2 and 4.3, and in the example in Subsection 4.6.

4.2 Moving Between Zones

This sub-procedure is called whenever ℓ or r moves between zones. It is responsible for finding the initial position for the pointer within its new zone, for updating the Parikh matrix according to the new position and for finding the problematic columns that need to be visited by the algorithm in this current pair of L and R zones.

When r moves from R' zone to R zone to its right, its new position is the leftmost position in R zone, which is a position of a mismatch (and, of a problematic column in T). The computation

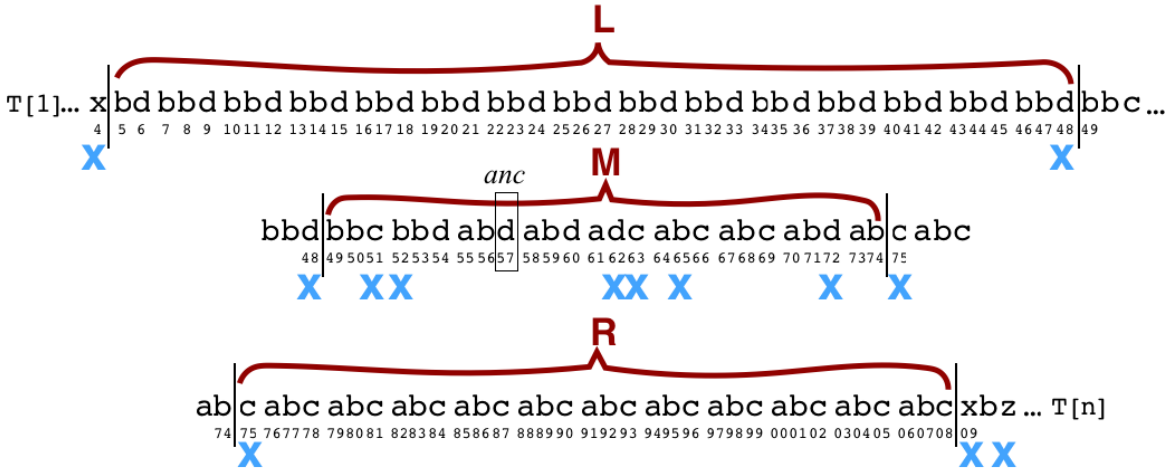


Figure 4: Given a text T, a period size $p = 3$, and mismatch positions (marked with x), a possible partition for L, M and R zones is given. Note that for simplicity of presentation, the text is divided into 3 overlapping substrings, as presented in the figure. L zone contains 15 (not full) periods (i.e., $bd(bbd)^{14}$) and R zone contains 12 (not full) periods (i.e., $c(abc)^{11}$). M contains 9 different zones. The *anc* position is 57.

of the new position of ℓ in a new L zone is more complicated: following Lemma 1, ℓ is within the $(k+1)$-periods of L. The procedure finds the leftmost position in L such that the number of modified positions in the substring $T_{\ell+1,r-1}$ does not exceed k. Since all periods in the L zone are equal, and since the Parikh matrix of the substring before the contraction is already computed, it is easy to find this position in $O(k)$ time. Observe that there might be a situation in which there is no possible position for ℓ in L zone. This can happen when all the letters in L zone are winners in their column. This means that no modified position can be released on this zone, and ℓ has to be moved to the next ℓ*-zone* to its right.

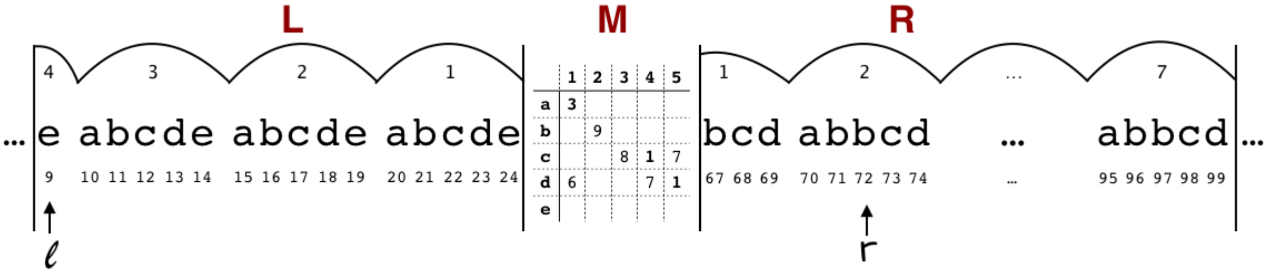
After the new position of the pointer is computed, the Parikh matrix is updated and the auxiliary lists, *LeftList*, *RightList*, and *newColumnList* are initialized according to the majority of the letters in the problematic columns in both L and R.

We introduce the following notations that will be used throughout this subsection. Denote by $first_\ell$ ($first_r$) the first position of ℓ (r) in L (R) zone after the new position for either ℓ or r is computed (if ℓ moved between zones, r has not changed, and vice versa). Denote by p_ℓ (p_r) the period in which $first_\ell$ ($first_r$) is positioned. Clearly, we have $p_\ell, p_r \leq k+1$.

For every problematic column, *col*, we decide whether the positions of the column in either L or R should be visited. Let x and y be the letters in *col* in L and R zones, respectively. The columns that need to be visited in R zone are added to *RightList*, whereas the columns that need to be visited in L are added to *LeftList*. In addition, as described in subsection 4.1, columns that should be visited starting a specific period in L are added to *newColumnList*.

For *RightList* we consider the following three cases:

- **R1:** y is the winner in its column, and will stay the winner as ℓ and r increase in the specific R and L zones. In this case *col* is not added to *RightList*.



	1	2	3	4	5
a	3				
b		9			
c			8	1	7
d	6			7	1
e					

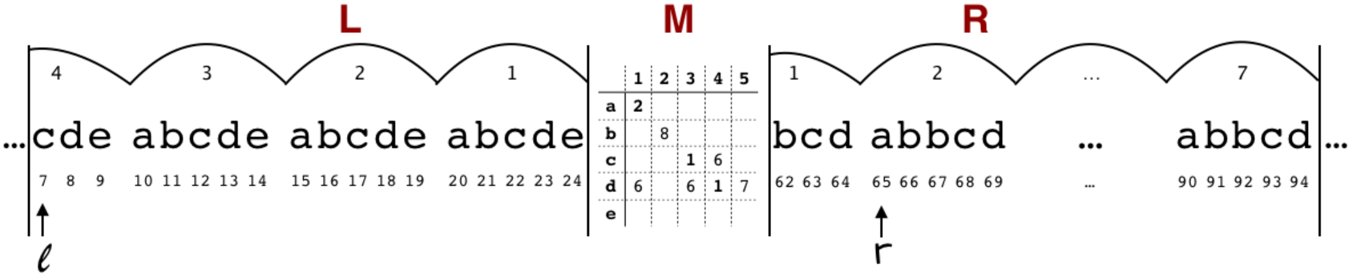
Figure 5: An example for case R3. Let $p = 5$ and $k = 14$. Assume that ℓ moved to a new ℓ-zone, $L = T_{9,24}$. We have $first_\ell = 9$ and $first_r = 72$. *LeftList* contains only column 5. *RightList* contains columns $1, 3, 4$ and 5. Observe that although a is the winner in column 1 in $T_{10,71}$, it is a losing letter in $T_{20,73}$. Therefore, it is added to *RightList*. When r reaches position 94, the letter a can no longer lose its majority to the letter d and it is removed from the *RightList*.

- **R2:** y is the loser in its column. In this case *col* is added to *RightList*. Note that there might be a situation in which the letter will gain majority as r is increased, and the column should be removed from the corresponding list. This is handled as part of the moving between positions sub-procedure.
- **R3:** y is the winner in its column, but as ℓ and r are increased, the letter might lose its majority to a letter in M. This can only happen when $x = y$, and as ℓ increases, the number of occurrences of y is decreased. In this case, *col* is added to *RightList*. The column will be removed from *RightList* when y cannot lose its majority to the letter in M (as in case **R1**). Observe that although y might be the winner of its column, its positions are visited in R. Since the number of such positions over all problematic columns is bounded by the number of modified positions in M, i.e, k, we allow these redundant visits (see Figure 5 for an example).

For *LeftList* we consider the following three cases:

- **L1:** x is the winner in its column, and will stay the winner as ℓ and r increase in the specific R and L zones. In this case *col* is not added to *LeftList*.
- **L2:** x is the loser in its column. In this case *col* is added to *LeftList*. Note that there might be a situation in which the letter will gain majority as r is increased. This can happen when $x = y$ and a letter z in M loses its majority to x. As in case **R3**, *col* will be visited by the algorithm although x might be the winner in *col*. Observe that since initially, when we move between zones, these positions are modified positions (since x was a losing letter to begin with), there cannot be more than k such positions over all problematic columns in L (see Figure 6 for an example).
- **L3:** x is the winner in its column, but as ℓ and r increase, the letter might lose its majority to a letter w in M or in R. In this case, we want ℓ to visit the positions of *col* only in the periods where the letter is a loser. Thus, *col* is not added to *LeftList*, but we want to mark the period in L where it might lose its majority.

  Assume that the difference between the number of x and w occurrences in $T_{first_\ell+1,first_r-1}$ is q, There are p_ℓ occurrences of x in L (or maybe $p_\ell - 1$ if $first_\ell$ points to a column greater



	1	2	3	4	5
a	2				
b		8			
c			1	6	
d	6		6	1	7
e					

Figure 6: An example for case L2. Let $p = 5$ and $k = 19$. Assume that ℓ moved to a new ℓ-zone, $L = T_{7,24}$. We have $first_\ell = 7$ and $first_r = 65$. *LeftList* contains the columns $1, 3, 4$ and 5. *RightList* contains columns 1 and 3. Observe that after ℓ is moved to position 8, r is further moved to position 67 and a becomes one of the winners of column 1. Although the letter in L is the winning letter in column 1, *LeftList* contains it. Initially (when ℓ moved to L zone), there were 3 modified positions in column 1 in L, these positions will be visited by the algorithm, although a might be the winner in them.

> than *col*). This means that starting period $i = p_\ell - q$ (or maybe $p_\ell - q - 1$), x might lose its majority to w. Therefore, *col* is added to the corresponding $newColumnList_i$. During the run of the main procedure, we might need to move *col* to a period greater than i (left to i), according to the move of r. This is handled in the sub-procedure for moving between positions, described in 4.4 (see running example in 4.6).

Observe that in a special scenario of case **R3** we add *col* to *RightList* even though the letter y in R (which is equal to the letter in L) is the winner in its column. Only when r is at a position where y cannot lose its majority to a letter in M (regardless of ℓ position), *col* is removed from *RightList*. This means that the number of such positions in R is bounded by the number of initial modified positions in M, which is $O(k)$. Furthermore, the number of such positions over all possible problematic columns in R is bounded by $O(k)$.

Observation 6. *There are at most $O(k)$ initial positions that are marked to be visited in R although the letter in R is a winner in its column.*

In a similar way, in a special scenario of case **L2**, we do not remove *col* from *LeftList* although the letter x in L (which is equal to the letter in R) is a winner in its column. Observe that initially, when *col* was added to *LeftList*, the letter x was a losing letter, meaning that the number of such positions cannot exceed the number of allowed modified positions, k. Furthermore, the number of such positions over all problematic columns in L can be at most k.

Observation 7. *There are at most $O(k)$ initial positions that are marked to be visited in L although the letter in L is a winner in its column.*

We proceed with the time complexity analysis of this sub-procedure.

Time Complexity: First, the sub-procedure finds the first position for the pointer in its new zone. For r, this is done in constant time, and for ℓ it is done in $O(k)$ time.

Second, the sub-procedure updates all problematic columns in the Parikh matrix and initializes *LeftList*, *RightList* and *newColumnList* lists. These updates are done in $O(k)$ time for each zone move. Note that even though *LeftList* and *RightList* are sorted lists, their initialization is done in linear time to their sizes since the columns are inserted to the lists in increasing order using the list of problematic columns in T. Thus, overall the time complexity is $O(k)$ for a move between zones.

4.3 Moving Between Periods

This sub-procedure is called whenever the ℓ position moves from a period to the period on its right in the same ℓ-zone. It checks whether new columns need to be added to *LeftList*. The procedure is very simple: it goes over all columns in $newColumnList_i$, and adds them to *LeftList* in a sorted manner. Then, the Parikh matrix is updated according to the new position of ℓ in L.

Time Complexity: On each period i in the $(k+1)$-periods of L zone, *LeftList* is updated as follows. Every problematic column in $newColumnList_i$ is added to *LeftList*. This is done in $O(\log k)$ time per column. The total number of such updates in one ℓ-zone is bounded by the number of problematic columns in T, which is $O(k)$. This gives a total of $O(k \log k)$ for an entire ℓ-zone handling.

4.4 Moving Between Positions Within a Period

This sub-procedure is called when ℓ or r are moved inside a period in their zone. ℓ (r) is moved to the next problematic column according to the *LeftList* (*RightList*) list, and the Parikh matrix is updated accordingly. Recall that *LeftList* and *RightList* don't necessarily contain the same problematic columns. This means that when a pointer is moved to a new position in a problematic column, *col*, the Parikh matrix should be updated according to the number of occurrences of both the letter x in L zone and the letter y in R zone. Actually, if *col* is not in *LeftList* (*RightList*), we don't need to update the entry $P[x, col]$ $(P[y, col])$ on the move of r (ℓ) since it is a winning letter in the column, but for easier handling we update them. Since all periods in a zone are equal, and since the starting and ending positions of all zones in $T_{\ell+1,r-1}$ are known, computing these numbers is done in constant time.

When r pointer is moved, we perform two additional checks: first, if the letter y in R becomes the only winner in its column, and cannot lose its majority, its column is removed from *RightList*. Second, we check whether *newColumnList* needs to be updated.

Recall that *newColumnList* represents the first period q in L that a winner letter x in L may lose its majority to another letter (in M or in R). It is initialized when ℓ or r moves between zones. In the case when x is different than y, we check whether the position in which y might gain majority over x changed (it might need to be moved to the left by one period). If so, we update the respected *newColumnList*. See example in Subsection 4.6, when r pointer is increased to position 81.

Note the special case in which the column is added to a period that ℓ is currently in. In this case, the column is added straight to *LeftList* in a sorted manner.

Observe that when $x = y$ and x might lose its majority to a letter in M, we do not update *newColumnList*. This can add $O(k)$ redundant visited positions.

Time Complexity: When either r or ℓ positions are increased, both the Parikh matrix and the auxiliary lists, *LeftList*, *RightList* and *newColumnList*, are updated. The Parikh matrix is updated in at most 3 entries: the entry counting the number of occurrences of the letter in the column in L zone, the entry counting the number of occurrences of the letter in the column in R zone, and the winner entry of the column. The entries are updated in constant time per visited position. In addition, the update of *newColumnList* is also done in constant time. Therefore, each pointer move takes constant time for updating the Parikh matrix and *newColumnList*.

Last, adding or removing problematic columns from *LeftList* and *RightList* are done in $O(\log k)$ time. A problematic column can be added at most once to *LeftList* (since a letter that becomes a loser in L remains a loser at least as long as r does not move between zones). In a similar way, a problematic column is removed from *RightList* when it is the winner of the column, and it cannot lose its majority. Therefore, each column is inserted or deleted from a list at most once, which gives a total of $O(k \log k)$ time for all updates for a specific pair of L and R zones..

4.5 Finding the Leftmost k-MAR.

We continue with a description of the procedure that finds the leftmost k-MAR in T that contains *anc*. The process of finding the first k-MAR is different from the process of finding the rest of the k-MARs that contain *anc*. We keep two pointers to the text, ℓ are r, representing the possible leftmost and rightmost positions of a k-MAR, respectively. We start with setting ℓ to $anc - 1$ and r to $anc + 1$. The procedure moves the ℓ pointer to the left until the number of modified positions in the substring $T_{\ell+1,r-1}$ exceeds k. The ℓ pointer is moved from one ℓ-zone to the one on its left, setting it each time to the leftmost position of the ℓ-zone. There, the number of modified positions in $T_{\ell+1,r-1}$ is computed using the Parikh matrix (note that since modified positions can be used only in a problematic column, the Parikh matrix contains only these columns). We stop when the number of modified positions in $T_{\ell+1,r-1}$ exceeds k.

Let z be the ℓ-zone to which the ℓ points. We now find the leftmost position for ℓ in z such that the number of modified positions in $T_{\ell+1,r-1}$ is exactly k. Following Lemma 1, ℓ points to a position within the $(k+1)$-periods of z. Therefore, it is easy to find in $O(k)$-time the period, and the problematic column in this period, to which ℓ should point.

Finally, we try to extend the k-MAR to the right. The procedure finds the rightmost position for r such that the number of modified positions in $T_{\ell+1,r-1}$ is exactly k. Recall that by definition, the leftmost r-zone does not contain mismatches in it, therefore, either the r position is within the $(k+1)$-periods of the zone, or it should point to the first position of the R zone to its right. After r is found, the algorithm reports the k-MAR $T_{\ell+1,r-1}$.

Observe that if there are less than k modified positions to the left of *anc*, ℓ pointer is set to 0, and the procedure moves r pointer to the right in a similar way to the move of ℓ above, until the k-MAR $T_{1,r-1}$ is found.

Time Complexity: Each move of either ℓ or r updates the Parikh matrix in $O(k)$ columns, which is done in $O(k)$ time. There are $O(k)$ zones in the substring, which leads to $O(k^2)$-time. Finding the position of ℓ in its zone is done by moving the pointer from one period to the next period on its right, for at most $O(k)$ periods. Each move is done in $O(k)$ time. This gives a total of $O(k^2)$ time for finding the position ℓ. Finding the final position for r pointer, is done in $O(k^2)$

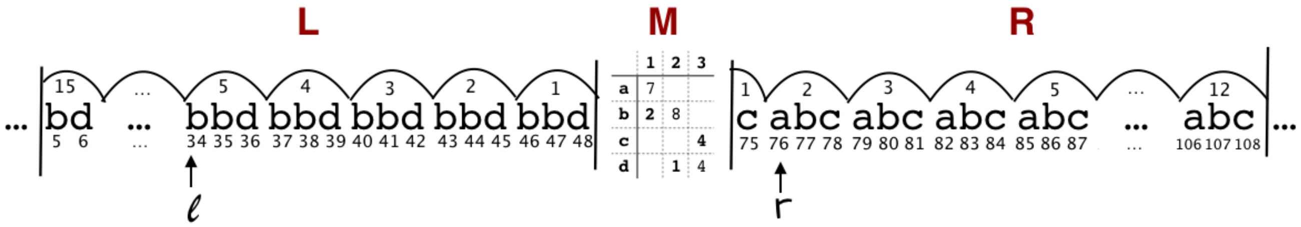


Figure 7: Running Example of the Efficient Main Procedure.

time.

Therefore, the time complexity of the initialization step is bounded by $O(k^2)$.

4.6 Running Example of the Efficient Main Procedure

We use Figure 7 to illustrate the algorithm of the efficient main procedure. The same substring of T is presented in Figure 4 where the current L, M and R zones are shown. For simplicity of presentation, in Figure 7, the M substring is presented by its Parikh matrix. In both examples, the period length, p, is 3, and the number of allowed modified positions, k, is 12. Assume that the sub-procedure starts when ℓ is moved to a new zone L. Although L zone contains 15 periods of *bbd*, ℓ position can be placed within the rightmost 13 periods of L (the periods are numbered above the text). Furthermore, since the first position of ℓ in the new zone should imply exactly 12 modified positions in $T_{\ell+1,r-1}$, the leftmost such position is computed and ℓ is set to 34.

The Parikh matrix of $T_{\ell+1,r-1} = T_{35,75}$ is computed: column 1 having 7 occurrences of a and 6 occurrences of b, column 2 has 13 occurrences of b and 1 occurrence of d, and column 3 having 5 occurrences of c and 9 occurrences of d. Note that column 2 should not be visited, since for both L and R the column contains the same letter, b, and even though it is a problematic column, b will stay the winner for the L and R zones: $L = T_{5,48}$ and $R = T_{75,108}$. Thus, the Parikh matrix of this L and R zones contains only two columns, 1 and 3.

Now, we proceed with updating the problematic column lists, *LeftList*, *RightList* and *newColumnList*. *LeftList* contains column 1, since b is a losing letter in the column (Case **L2**). *RightList* contains column 3 since c is currently a losing letter (Case **R2**).

Observe that as r and ℓ pointers are increased, d might lose its majority to c in column 3. The initial period in which it can happen is the first period of L zone (in position 48). Thus, we add column 3 to $newColumnList_1$ (Case **L3**).

We proceed with a move of r. The pointer is moved to column 3, to position 78. Since a is now the winning letter in its column, this move did not use a modified position. As r cannot be further moved to the right without using additional modified positions, the new 12-MAR, $T_{35,77}$ is reported.

The ℓ pointer is moved to column 1, at position 37. This move releases a modified position, since b is a losing letter in column 1. Now, we can continue with moving r. The r pointer is moved to position 81, and uses 1 modified position. Now the number of c occurrences in column 3 increases by 1, and *newColumnList* is updated: column 3 is removed from $newColumnList_1$ and is added to $newColumnList_2$ (since d can lose its majority to c in this period).

Observe that even though r points to position 81, the Parikh matrix contains $P[a, 1] = 8$, where actually the number of occurrences of a in $T_{\ell+1,r-1}$ is equal to 9. This column will be updated when ℓ position is moved to column 1.

Now, the number of modified positions in $T_{38,80}$ is equal to 12 and the new 12-MAR is reported.

We continue with a move of ℓ to position 40. The Parikh matrix is updated as follows. $P[b,1] = 4$ and $P[a,1] = 9$. This move released a modified position and we continue with moving r to position 84.

4.7 Pseudocode for the Efficient Main Procedure

We are now ready to present the following pseudocode of the efficient main procedure for finding k-MARs. Given a string T, an anchor position, *anc*, and a period length, p, the procedure finds the k-MARs with period p, containing *anc*, in the string.

As an initialization, the sub-procedure for finding the leftmost k-MAR that contains *anc* is called, and it is reported. After the initial positions for ℓ and r are set, the variable *count*, that counts the number of used modified positions, is set to k, and the auxiliary lists (*LeftList*, *RightList* and *newColumnList*) are updated according to $T_{\ell+1,r-1}$.

Then, we proceed to find the rest of the k-MARs that contain *anc*. The body of the main while loop (lines 5-28) is an iteration of the procedure. It starts with a k-MAR that was previously found, $T_{\ell+1,r-1}$. Then, the pointer ℓ is moved until r can be moved without exceeding the number of allowed modified letters (lines 6 - 15). When ℓ is stopped, the procedure continues with moving r to the right, until a new k-MAR is found (lines 16-26). Finally, the new k-MAR, $T_{\ell+1,r-1}$ is reported (line 27), and another iteration starts.

The move of ℓ starts in line 7, where ℓ is moved to the next position according to *LeftList*. If ℓ was moved between zones, a new position is found for it in the new zone, and the auxiliary lists are updated according to the new L and (the same) R zones (lines 8-10). Else, if ℓ was moved from one period, to the period on its right (lines 11-13), the respected *newColumnList* is checked and the columns it contains are added to *LeftList*. Observe that there might be a situation where new columns were added to *LeftList*, and ℓ is positioned to the right of these columns. Therefore, in line 13, the leftmost possible position for ℓ is found.

In line 14, the Parikh matrix and the count variable are updated according to the contracted substring. Then, in line 15, we check whether ℓ should be stopped. There are two cases where we want to continue with r: either the contraction released a modified position (happens when $T[\ell]$ is a losing letter) or the special case where $T[r]$ becomes the winner of its column, which means that r has a "free" move. In all other cases, ℓ is further moved to the right.

The while loop for the r move, in line 16, has two conditions: either the number of modified positions (count) is smaller than k, and then r can be moved no matter if $T[r]$ is a losing letter or not. Or, $T[r]$ is a winner letter, and therefore the move of r is a "free" move.

If any of the conditions apply, r can be moved. Then, the Parikh matrix and *count* variable are updated according to the extension of the substring to end at position r. Now, we proceed to check whether a move between zones happened as a result of the extension (line 18). In this case, the auxiliary lists are updated according to the (same) L and the new R zones.

In line 19, we check whether the column of r, denoted by *col*, should be removed from *RightList*. In lines 20-23, we update *newColumnList*, if needed.

Then, the new position for r is set according to *RightList*. If the position is to the right of the $(k+1)$-periods of R, in line 25, r is set to the first position of the next r-zone.

Finally, in line 27, when r cannot be further moved to the right, if the size of $T_{\ell+1,r-1}$ is greater than $2p$, the new found k-MAR is reported.

```
   input  : string $T$, position $anc$, period length $p$
   output: All $k$-MARs containing $anc$ in $T$
 1 compute leftmost $k$-MAR, $T_{i,j}$, and its Parikh matrix
 2 $(\ell, r, count) \leftarrow (i-1, j+1, k)$
 3 report new $k$-MAR, $T_{\ell+1,r-1}$
 4 update LeftList, RightList and newColumnList
 5 while $\ell < anc$ and $r < |T|$ do
 6     repeat /* move $\ell$ to the right until $r$ can be moved */
 7         $\ell \leftarrow$ next position according to LeftList
 8         if $\ell$ is in a new zone then
 9             $\ell \leftarrow$ leftmost possible position for $\ell$ in new zone
10             update LeftList, RightList and newColumnList
11         else if $\ell$ is in a new period $i$ then
12             update LeftList using $newColumnList_i$
13             $\ell \leftarrow$ leftmost possible position for $\ell$ in new period
14         update Parikh Matrix and count according to $T_{\ell+1,r-1}$
15     until $!Winner(\ell)$ or $Winner(r)$
16     while $count = k-1$ or $(count = k$ and $Winner(r))$ do /* move $r$ to the right */
17         update Parikh Matrix and count according to $T_{\ell+1,r}$
18         if $r$ is in a new zone then  update LeftList, RightList and newColumnList
19         if $T[r]$ cannot lose its majority in its column, col then  remove col from RightList
20         if $!Winner(r)$ and $T[r]$ wins starting period $i$ of $L$ then
21             remove col from $newColumnList_{i-1}$
22             add col to $newColumnList_i$
23         end
24         $r \leftarrow$ next position according to RightList
25         if $r$ is outside $(k+1)$-periods of its zone then  $r \leftarrow$ first position of next zone
26     end
27     if $\ell - r \geq 2p$ then  report new $k$-MAR, $T_{\ell+1,r-1}$
28 end
```

Algorithm 1: The Improved Find k-MARs Algorithm

4.8 The Number of Visited Positions in a Zone

In the efficient algorithm we visit the following positions: (a) positions of modified positions, (b) methodological stop at each period in the $(k+1)$-periods of an ℓ-zone, and (c) additional $O(k)$ special case positions described in observations 6 and 7. The total number of visited positions of cases (b) and (c) is bounded by $O(k)$. Therefore, we now proceed to bound the number of positions that can be modified positions in L and R zones.

Following Lemma 1, there are $O(k^2)$ possible positions for modified positions in a zone. Although only these positions can be modified, the efficient algorithm does not visit all of them. In Lemma 2 we prove that we actually visit only $O(k \log k)$ positions in a zone.

For clarity of presentation, we *conceptually* mark modified positions with *pebbles.* We put a *pebble* on each modified position in $T_{\ell+1,r-1}$. Clearly, if $T_{\ell+1,r-1}$ is a k-MAR, there are exactly k pebbles in $T_{\ell+1,r-1}$. A move of ℓ can decrease the number of pebbles in the substring (when a modified position is released) and a move of r can increase their number (when a modified position is used). Additionally, a move of the pointers can *move* pebbles from R to L (and maybe M). This happens when a letter in R becomes the only winner in its column, *col*. If the pebbles were placed in R (and M) before the extension of the substring, they are now conceptually *moved* to the other positions of *col* in L (and M).

The modified positions are the positions marked with pebbles in L and R. Computing the number of these positions is not trivial, since pebbles are moved back and forth from L to R. Every pebble that is released as a result of an ℓ move implies an additional position that can be visited in R (and be marked with a pebble). In addition, an r move may result a move of pebbles from R to L, which means that positions in L that were not marked with pebbles are now marked and should be visited. Lemma 2 proves a bound for the total number of visited positions.

Lemma 2. *For a given L and R zones, the efficient algorithm visits $O(k \log k)$ positions.*

Proof. We start with positions that are not marked with pebbles. First, we visit every period of the $(k+1)$-periods of L. Second, following observations 6 and 7, there are at most $O(k)$ redundant positions in L and R that are visited although they may not be marked with pebbles. Clearly, the number of these positions is bounded by $O(k)$.

We now proceed to count the maximum number of positions that can be marked with pebbles in both L and R.

We consider two cases: when ℓ moves between zones (and r points to some position in the $(k+1)$-periods of the R zone) and when r moves between zones (and ℓ points to some position in the $(k+1)$-periods of the L zone). We start with the move of ℓ.

Assume that ℓ is moved from an ℓ-zone, L', to position $first_\ell$ in the ℓ-zone on its right, L (see Subsection 4.2). Clearly, k (or $k-1$) pebbles are spread in L, M and R, according to the modified positions in $T_{first_\ell+1,r-1}$. During the computation of the algorithm pebbles move from L to R and from R to L. The move of pebbles from L to R occurs when ℓ moves to the right and a pebble is released. Then, as r increases, the pebble is used. Observe that an r move can use a pebble, only after a pebble was moved from L.

Clearly, the number of pebbles that are moved from L to R is bounded by the initial number of pebbles in L and the number of pebbles that can be moved from R to L. Thus, the challenge is computing the number of pebbles that can be moved from R to L. This situation occurs when a letter in R gains majority over a letter in L in the same column.

In the second case, when r moves between zones, from an r-zone, R', to the r-zone on its right, R. Recall that the first position of r in R is the leftmost position of R (see Subsection 4.2). This means that k (or $k-1$) pebbles are spread in L and M only, according to the modified positions in $T_{\ell+1,r-1}$. Now, there are no initial pebbles in R, but as ℓ increases pebbles move from L to R, and can again move from R to L as letters in R gain majority.

Since we want to bound the number of pebbles moved from R to L, we assume that initially there are $O(k)$ pebbles in R zone. We distinguish between the two cases when the letter in R equals the one in L in the same column, and when they are different.

Let x be the letter of a column *col* in L, y be the letter of *col* in R, and w be the letter with the maximum number of occurrences in *col* in M.

- Case 1: $x \neq y$

  Assume that the number of problematic columns in *RightList* is a ($a \leq k$). Consider the first positions ℓ', r' where a losing letter y gains majority over a letter in L (in the extended substring $T_{\ell'+1,r'}$, y becomes the winner in *col*).

  After the extension, the pebbles in R that were marking positions in *col* are moved to the left, to L zone (and possibly M). If R contained b pebbles before the extension ($b \leq k$), then after the extension at most $\lceil k/a \rceil$ pebbles are moved from R to the left. Note that from this point on, y will never again lose its majority to x (at least as long as ℓ does not move between zones), since the number of its occurrences will only increase as r pointer increases. The number of problematic columns in *RightList* decreases to $a - 1$.

  As the algorithm continues, pebbles are released from L zone, and are moved to R zone. Allowing for at most $\lceil k/a \rceil$ additional positions that can be visited in R.

  When another letter y' in R gains majority in another problematic column, *col*$'$, at most $\lceil k/(a-1) \rceil$ pebbles are moved from R to L. This situation can continue until either r reaches the end position of the $(k+1)$-periods of R zone or all the letters in R are winning letters in their columns (i.e., when r moves between zones).

  Therefore, the maximal number of positions visited in R in this case is equal to $\Sigma_{i=1}^{k} \lceil k/i \rceil$, which gives a total of $O(k \log k)$.

- Case 2: $x = y$

  If y becomes a winner in its column (i.e. $|y| \geq |w|$), it does not necessarily mean that y will continue to be the winner in its column as ℓ and r pointers are increased (since the number of occurrences of y is decreased as ℓ position is increased). This situation is described in Subsection 4.2: all positions of this problematic column are visited in R zone until r reaches the $|w|$'th period (counting from left). Starting from this position, y cannot lose its majority to w anymore. Since the positions of w M are positions of modified positions in a k-MAR, their number is bounded by the number of allowed modified positions in a k-MAR, k. Thus, the number of visited positions of R for the letter y is also bounded by k. Moreover, the total number of all such cases, for all the problematic columns in which $x = y$, cannot exceed k, as it is bounded by the number of modified positions used in M.

Thus, the total number of positions marked with pebbles in R cannot exceed $O(k \log k)$, resulting at most $O(k \log k)$ visited positions in L zone. This gives a total of $O(k \log k)$ visited positions in both L and R zones. □

4.9 Time Complexity Analysis of the Main Procedure

The time complexity of the initialization step is $O(k)$, and it is done once.

Moving between zones is done in $O(k)$ time for each zone move (see Subsection 4.2).

Moving between periods is done in $O(k \log k)$ for an entire ℓ-zone handling (see Subsection 4.3).

Moving between positions is done in constant time per each visited position in a zone. Following Lemma 2, there are at most $O(k \log k)$ visited positions in each pair of L and R zone. Thus the time complexity of moving between positions sub-procedure is bounded by $O(k \log k)$.

There are $O(k)$ zones in the input string T. Each one of the zones is visited at most once as L or as R, which gives a total of $O(k^2 \log k)$ time complexity for the entire procedure.

Alphabet Size: Note that in this time complexity analysis the alphabet size is constant. For ordered alphabet, the time complexity of visiting a position is higher since updating the winner letter of a column cannot be done in constant time. Assume that the letter $T[r] = a$ was a loser letter in its column. As r position increases, the number of occurrences of a in the column increases and it might become a winner in the column. In order to support quick queries for finding the winner letter of a column, we can keep a priority queue for each column of the period. The priority queue stores the letters occurring in the column ordered by the number of their occurrences. On each update of the Parikh matrix, the queue is also updated in $O(\log |\Sigma|)$ time. After the queue is updated, finding a letter with maximal number of occurrences is done in constant time. This will raise the time complexity of the main procedure to $O(k^2 \log k \log |\Sigma|)$ time for ordered alphabet.

The next section describes the entire algorithm, that given a string T locates the k-MARs in the string. The algorithm uses this $O(k^2 \log k)$ version of the main procedure for finding the k-MARs in substrings of T.

5 The Algorithm

We are now ready to present the entire algorithm, following the framework of Kolpakov and Kucherov [21] for finding maximal approximate runs. As in [21], we decompose the text into *factors* and *blocks* using a variation of the LZ-factorization of the string, called LZ-factorization with overlap.

The LZ-factorization (with copy overlap) of a string T divides T into non-overlapping substrings, $f_1 \cdot f_2 \cdots f_r$, each f_i is called a *factor*, as follows. The first factor is the first letter of T, i.e. $T[1]$. For every $i > 1$, f_i is the shortest substring occurring in T immediately after $f_1 \cdot f_2 \cdots f_{i-1}$ that does not occur in $f_1 \cdot f_2 \cdots f_{i-1} \cdot f_i$ other than as a suffix. For example, the LZ-factorization of the string *aabbababababbbb* is $a|ab|ba|bababb|bb$. The computation of the LZ-factorization of a string can be done in time linear in its length. Then, the string T is divided into consecutive *blocks*, each of which contains $2k + 2$ consecutive LZ-factors[1]. Let $T = B_1 \cdot B_2 \cdots B_r$ be the partition of T into these *blocks*.

We now present our algorithm. Given a string T of length n and a number k, the algorithm outputs an array, *results*, of size n, such that an entry $results[left]$ contains a list of couples $(p, right)$ that corresponds to a k-MAR, $T_{left,right}$, with period length p.

[1] Observe that in [21], each block contains $k + 1$ factors. Here, since we want to capture modified positions (as opposed to mismatches in [21]), we compose each block from $2k + 2$ factors.

The algorithm is divided into four steps, following the steps of [21].

- Decompose the string T into factors and blocks. During the process of decomposing the string into factors, we keep a pointer, $orig$, from each factor to its previous occurrence in the string (without its rightmost letter). In addition, build the suffix trees of S and S^R, and preprocess them to answer constant time LCA queries.

- Find k-MARs crossing blocks: for each period length p ranging from k to $|B_iB_{i+1}|$, let anc be the rightmost position of the block B_i, and let lb (rb) be the position of the $2k+1$ mismatch to the left (right) of anc. Call the efficient main procedure with $T_{lb,rb}$, anc and p. Discard the k-MARs that contain the rightmost letter of block B_{i+1}. All found k-MARs, $T_{left,right}$, are saved in the $results$ array.

- Find k-MARs that are entirely contained inside blocks: for each block, $B = f_i \dots f_{i+m}$, a recursive binary division of the block according to its factors is done, such that $B = B'B''$, $B' = f_i \dots f_{i+(m/2)}$ and $B'' = f_{i+(m/2)+1} \dots f_{i+m}$. Let anc be the rightmost position of factor $f_{i+(m/2)}$, and let lb (rb) be the position of the $2k+1$ mismatch to the left (right) of anc. For period length p ranging from k to $|B|/2$, call the efficient main procedure with $T_{lb,rb}$, anc and p. Discard the k-MARs that contain the rightmost letter of f_{i+m} or the leftmost letter of f_i. All found k-MARs are saved in the $results$ array. Finally this step is called recursively with sub-blocks B' and B'', independently.

- Find k-MARs that are entirely contained inside factors: in this step, we go over the factors from left to right, and for each factor starting at position i we follow its pointer to its generating factor, $orig$: if $results[orig]$ contains a k-MAR and its length is not longer than the current factor's length, its data is copied to $results[i]$ (if the k-MAR length is longer than the factor's length, the k-MAR was found in an earlier stage).

Total Time Complexity: The time complexity analysis is similar to the analysis in [21]. The only difference is that the main procedure in our algorithm takes $O(k^2 \log k)$ time and not $O(k)$ as in their algorithm.

The first step is done in linear time. In the second step, for each block B_i, the k-MARs that touch the rightmost letter of B_i are found, discarding the k-MARs that touch the rightmost letter of B_{i+1}. The period size of these k-MARs is bounded by the sizes of B_i and B_{i+1} blocks, i.e. $p < 2|B_iB_{i+1}|$. For each block B_i and for each period length p ranging from k to $|B_iB_{i+1}|$ we first find the boundaries of the substring $T_{lb,rb}$ using the sub-procedure described in Subsection 3.3. This is done in $O(k)$ per period size. Then, we call the efficient main procedure to find the k-MARs that contain anc. The efficient main procedure runs in $O(nk^2 \log k)$ per period size. This gives a total of $O(nk^2 \log k)$ time complexity for the second step.

In the third step, for each block B_i and each period length p ranging from k to $|B_i|/2$, a binary division of the block according to its factors is done. Similarly to the previous step, we start with finding the boundaries lb and rb in $O(k)$ time. Then, the efficient main procedure is called and runs in $O(nk^2 \log k)$ per period length. For each block, the number of recursion levels is derived from the number of its factors, i.e. $O(\log k)$. Thus, for all blocks, the time complexity is bounded by $O(nk^2 \log^2 k)$ time.

The final step takes $O(n + occ)$ time, where occ is the number of k-MARs found, as we perform a linear scan on the results array.

This gives a total time complexity of $O(nk^2 \log^2 k + occ)$ for the entire algorithm.

6 Acknowledgments

The first and the third authors are partially supported by the Israel Science Foundation grant 347/09, the National Science Foundation Award 0904246, DFG and Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF).

References

[1] M. I. Abouelhoda, S. K., and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

[2] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theor. Comput. Sci.*, 22:297–315, 1983.

[3] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The ”runs” theorem. *CoRR*, abs/1406.0263v4, 2014.

[4] H. Bannai, S. Inenaga, and D. Köppl. Computing all distinct squares in linear time for integer alphabets. *CoRR*, abs/1610.03421, 2016.

[5] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics*, pages 88–94. Springer, 2000.

[6] G. Chen, S. J. Puglisi, and W. F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, 2008.

[7] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[8] M. Crochemore. Recherche linéaire d'un carré dans un mot. *C. R. Acad. Sc. Paris Sér. I Math.*, 296(18):781–784, 1983.

[9] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007. 392 pages.

[10] M. Crochemore and L. Ilie. Computing longest previous factors in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008. DOI: 10.1016/j.ipl.2007.10.006.

[11] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń. New simple efficient algorithms computing powers and runs in strings. *Discrete Applied Mathematics*, 163(3):258–267, 2014.

[12] Johannes Fischer, Štěpán Holub, I Tomohiro, and Moshe Lewenstein. Beyond the runs theorem. In *International Symposium on String Processing and Information Retrieval*, pages 277–286. Springer, 2015.

[13] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

[14] R. Groult, M. Leonard, and L. Mouchard. Speeding up the detection of evolutive tandem repeats. *Theor. Comput. Sci.*, 310(1-3):309–328, 2004.

[15] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

[16] C. S. Iliopoulos, D. Moore, and W. F. Smyth. A characterization of the squares in a Fibonacci string. *Theor. Comput. Sci.*, 172(1-2):281–291, 1997.

[17] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

[18] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Combinatorial pattern matching*, pages 186–199. Springer, 2003.

[19] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005.

[20] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Foundations of Computer Science*, pages 596–604, 1999.

[21] R. M. Kolpakov and G. Kucherov. Finding approximate repetitions under Hamming distance. *Theor. Comput. Sci*, 1(303):135–156, 2003.

[22] S. Rao Kosaraju. Computation of squares in a string (preliminary version). In *CPM*, pages 146–150, 1994.

[23] G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.

[24] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.

[25] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.

[26] M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.

[27] R. J. Parikh. On context-free languages. *Journal of the ACM*, 13:570–581, 1966.

[28] J. S. Sim, C. S. Iliopoulos, K. Park, and W. F. Smyth. Approximate periods of strings. *Lecture Notes in Computer Science*, 1645:123–133, 1999.