Locating All Maximal Approximate Runs in a String

Mika Amit, Maxime Crochemore, Gad Landau

Locating All Maximal Approximate Runs in a String

Mika Amit[1] ⋆ , Maxime Crochemore[3,4] , Gad M. Landau[1,2] ⋆⋆

[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-Poly, 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

Abstract. An exact run in a string, T, is a non-empty substring of T that can be divided into adjacent non-overlapping identical substrings. Finding exact runs in strings is an important problem and therefore a well studied one in the strings community. 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 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 every non-empty substring T' of T such that changing at most k letters in T' transforms it into a maximal exact run in T. We present an $O(nk^2 \log k \log \frac{n}{k})$ algorithm.
Keywords: algorithms on strings, pattern matching, repetitions, tandem repeats, runs.

1 Introduction

The domain of Algorithms on strings is fond of combinatorial properties on words. They are used to analyze the behavior of algorithms in conjunction with statistical results, and often lead to improving their design up to optimal characteristics. Conversely, some combinatorial properties on words, obtained without any algorithmic objectives, yield algorithms that are surprisingly efficient according to various aspects (time, space, design, etc.).

The most central properties relate to periodicities in words and pop up in many examples. The notion is doubtlessly at the core of many stringology questions. It constitutes a fundamental area of string combinatorics due to important applications to text algorithms, data compression, biological sequences analysis, or music analysis. Indeed, periods are ubiquitous in string algorithms because stuttering is likely to slow down any of the algorithms for pattern matching, text compression, and genome assembly, for example.

A maximal exact run in a string is informally an occurrence of a non-extensible segment having a small period. The concept captures all the power of local periodicities and repetitions and therefore has attracted a lot of studies.

Several methods are available to detect all the occurrences of exact repetitions in strings with some small variations on the elements they target (see [Cro81,AP83,ML84]). For a given string T, of length n, these algorithms run in $O(n \log n)$ time, which is optimal because some strings contain this number of elements. Selecting some of their occurrences or just distinct repetitions regardless of their number of occurrences (see [Kos94,GS04]) paved the path to faster algorithms.

Runs have been introduced by Iliopoulos, Moore, and Smyth [IMS97] who showed that Fibonacci words contain only a linear number of them according to their length. Kolpakov and Kucherov [KK99] (see also

⋆ partially supported by the Israel Science Foundation grant 347/09.
⋆⋆ partially supported by the National Science Foundation Award 0904246, Israel Science Foundation grant 347/09, Yahoo and Grant No. 2008217 from the United States-Israel Binational Science Foundation (BSF).

[CHL07], Chapter 8) proved that the property holds for any string. Meanwhile they designed an algorithm to compute all runs in a string of length n over an alphabet Σ in $\mathcal{O}(n \log(|\Sigma|))$ time. Their algorithm extends Main's algorithm [Mai89], which itself extends the method in [Cro83] (see also [CHL07]).

The design of a linear-time algorithm for building the Suffix Array of a string on an integer alphabet (see [CHL07]) and the introduction of another related data structure (the Longest Previous Factor table in [CI08]) have eventually led to a linear-time solution (independent of the alphabet size) for computing all runs in a string.

Finding approximate runs is more sensible than finding exact runs in some applications. 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 ([LSS01,SIPS99,KK99,AEL10]). The k-approximate run problem is defined as follows: given a string x and a number k, divide x into non-empty substrings, $x = u_1 u_2 \cdots u_t$, such that the distance between every two adjacent substrings, u_i and u_{i+1}, is not greater than k, or the distance between *every* two substrings u_i and u_j in x cannot exceed k. Another definition of approximate run is as follows: a string x is a k-approximate run if $x = u_1 u_2 \cdots u_t$ and the removal of the same k positions from all u_i generates an exact run. A different approach for the problem is defined as: given a string x and a number k, find a consensus substring u such that x can be divided into a number of adjacent non overlapping occurrences of substrings $x = u_1 u_2 \cdots u_t$ and the distance between u and every u_i is not greater than k.

Our definition of approximate run is more global: find all non-empty substrings that by the modification of at most k letters form an exact run. In other words, for each such substring there exists a consensus string, u, such that the sum over all Hamming distances between u and u_i is not greater than k. We present an $O(nk^2 \log k \log \frac{n}{k})$-time algorithm that solves this problem.

Roadmap: we start in section 2 with definitions and notations that will be used throughout the paper. In section 3, we present a simple $O(n^2)$ algorithm for finding all approximate runs in the string. Then, in section 4 we continue to an improved $O(nk^3 \log \frac{n}{k})$ algorithm. Finally, on section 5, we present the $O(nk^2 \log k \log \frac{n}{k})$ algorithm.

2 Definitions and Notations

Let $T = T[1]T[2] \cdots T[n]$ be a string of size n defined over the constant alphabet Σ. In this abstract the size of the alphabet is constant. We denote the i'th letter in T as $T[i]$, and the substring of T that starts at position i and ends at position j as $T[i \,..\, j] = T[i]T[i+1] \cdots T[j]$. Let T^R be the reversed substring of T.

An *exact run* is a non empty string, T, that can be divided into a number of identical adjacent non overlapping substrings $T = u_1 u^t u_2$, where the first substring u_1 can be a suffix of u, and the last substring u_2 can be a prefix of u. u is called a *period* and its length is denoted as the *period length*, $p = |u|$. The exponent of the run is of size $\frac{|T|}{p}$ and it is greater or equal to 2. For instance, *'abababababa'*, has exact runs with period length 2 and 4. Their exponent are 4.5 and 2.25, respectively.

A *maximal exact run* is an exact run that cannot be extended to either right or left. For instance, *'dabababac'*, has a maximal exact run starting at position 2 with period length 2 and exponent 3.5. If a substring 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. Similarily, if the maximal exact run ends at position j, then either $j = n$ or $T[j+1] \neq T[j+1-p]$.

A *k-maximal approximate run (k-MAR)* is a non empty string, T, such that the modification of at most k letters in T generates a maximal exact run. For instance, *'abaabcaba'*, is a 1-maximal approximate run with period length 3 and exponent 3. In this example, the letter in position 6 is a *modified* letter, since modifying it from $'c'$ to $'a'$ generates an exact run.

For the rest of this paper we distinguish between two notations: a *mismatch* and a *modified letter*, that have a very strong relation between each other, but, as we will explain shortly, are not always identical in their meanings. If 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. We mark position $i+p$ as the position in which the mismatch occurs. A *modified letter* corresponds to a modification of a letter in an approximate run in order to convert it to an exact run. For instance, for period length $p=3$ and the substring $'abcabcabd'$, there is a *mismatch* at position 9 (between $'c'$ and $'d'$) and there is a modified letter in the same position, 9.

Observe that while a mismatch can imply that a modified letter needs to be used in the k-MAR and vice versa, it is not always straightforward:

A1. Two mismatches can imply only one modified letter. For instance, for a substring $'abaabcaba'$ with period length 3, there are two mismatches in positions (6) and (9), but only one modified letter is needed in position 6 in order to convert the original substring into an exact run (by modifying the letter 'c' to 'a').

A2. One mismatch can imply at most $\frac{n}{2p}$ modified letters. For instance, in the 2-MAR with period length 3 of the string $'abcabcabdabdabd'$, there is only one mismatch in position 9 and two modified letters on positions 3 and 6.

We continue with a definition of Parikh matrix (see also [Par66]) that will be used throughout the paper: A *Parikh matrix*, $P[1..|\Sigma|, 1..p]$, is a two-dimensional array defined over a substring $T[i..j]$ and a period length p. An entry $P[a,c]$ contains the number of occurrences of $a \in \Sigma$ in the column c of the period. In addition, for each column c, we keep an additional variable, $win(c)$, that contains the *winner* letter - the letter that occurs more than any other letter in this column (breaking ties arbitrarily).

We use the Parikh matrix in order to count the number of modified letters used in a k-MAR of period length p that starts at position i and ends at position j. For simplicity, we set the first column of the period to be position 1 in the text. This means that an approximate run can start at any column of the period, and for period length p, the column of index i is $Column(i) = i \ mod \ p$ (with the exception of the case where $i \ mod \ p = 0$, in which $Column(i) = p$). The number of modified letters associated with a column is the sum over all letters that are not the winner letter. i.e. $\Sigma_{a \neq win(c)} P[c,a]$. If the sum of modified letters in a column is greater than 0, the column is denoted as a *problematic column*. The number of modified letters associated with the k-MAR is the sum of modified letters over all its problematic columns.

Figure 1 shows examples of Parikh matrices for three period lengths computed for the same prefix of a string.

2.1 Problem Definition

In this paper we present three algorithms that solve the following problem:

Definition 1 (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-MAR in the string T of all period lengths p, $1 \leq p \leq \frac{n}{2}$.*

3 A Simple $O(n^2)$ Algorithm Using Parikh Matrices

We start with a simple algorithm for computing all k-MAR in a string T. The algorithm iterates over all period lengths p, $1 \leq p \leq \frac{n}{2}$, and for each period length computes all k-MAR that exist in the string. We describe the procedure for finding all k-MAR with period length p:

i		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$T[i]$		a	b	b	a	c	a	b	a	a	c	a	b	b	b	c	a	a	b	b	c	b	a	.
p=3	a	1			2		**1**	2	**1**	2	2	2												
	b		1	1		1	1	**1**	1	**1**	**1**	**1**												
	c					**1**			**1**		**1**	**1**												
	win	a	b	b	a	b	b	a	b	**a**	a	**a**												
$T[i]$		a	b	b	a	c	a	b	a	a	c	a	b	b	b	c	a	a	b	b	c	b	a	.
p=5	a	1			1		2		**1**	2		3		**1**	2		4	**1**	**1**	2		4		
	b		1	1				2	1				3	2	**1**			3	3	**2**		**1**		
	c					1					2					3					4			
	win	a	b	b	a	c	a	b	b	a	c	a	b	b	a	c	a	b	b	a	c	a		
$T[i]$		a	b	b	a	c	a	b	a	a	c	a	b	b	b	c	a	a	b	b	c	b	a	.
p=7	a	1			1		1		2	**1**		2		1		2	2							
	b		1	1				1		1	1		**1**	**1**	2		**1**							
	c					1					**1**		1			**1**								
	win	a	b	b	a	c	a	b	a	b	b	a	c	a	b	a	a							

Fig. 1: Examples of maximal approximate runs with the maximum of 5 modified letters. The longest run has length 11 for period length 3, 21 for period length 5, and 16 for period length 7. The three Parikh matrices are computed from left to right.

FINDKMAR(T, p)

```
1  Initialize Parikh matrix P
2  T[n+1] ← $
3  ℓ ← 1, r ← 1, count ← 0, newKmar ← true
4  Move r :
5  while r ≤ n+1 do
6      if Winner(r) = true or count < k then
7          if Winner(r) = false then
8              count ← count + 1
9          update Parikh matrix according to T[r]
10         r ← r + 1
11         newKmar ← true
12     else \* r cannot be increased *\
13         if newKmar then
14             announce k-MAR T[ℓ .. r − 1]
15             newKmar ← false
16         goto Move ℓ
17
18 Move ℓ :
19 if r = n + 1 then
20     return
21 update Parikh matrix according to T[ℓ] delete
22 if Winner(ℓ) = false then
23     count ← count − 1
24 ℓ ← ℓ + 1
25 goto Move r
26
27 return
```

In the $FindKmar$ procedure described above we keep two pointers, ℓ and r, on the string T, such that they define possible leftmost and rightmost positions of a k-MAR, respectively. The computation uses Parikh

matrix, P, of size $|\Sigma| \times p$. The initialization of the Parikh matrix (line 1) consists of setting the winners of all columns of p according to the letters in $T[1\,..\,p]$.

The procedure uses an auxiliary function, $Winner()$, that gets a position i in the text as a parameter and returns true if the number of occurrences of the letter $T[i]$ in its column is equal to the number of occurrences of the winner letter in it (lines 6, 7, 22). Recall that the first column of the period is set to position 1 in the text.

Each iteration call checks whether r position can be increased: either r is one of the winners in its column (thus no modified letters are used when r is increased) or the number of used modified letters is less than the maximum allowed. When r position cannot be increased, ℓ position is increased. Note that as long as $r \leq n$ each update of ℓ position calls $Move\ r$ sub procedure. The reason for that is that the deletion of $T[\ell]$ can either release a modified letter (line 22) or change the current winners list of ℓ's column. In the case where these two cases have not occurred, $Move\ r$ procedure does not increase r and the iteration returns to $Move\ \ell$ sub procedure (line 16).

A new k-MAR is announced whenever r position cannot be increased and its position has moved since the last k-MAR announcement. In this case $T[\ell\,..\,r-1]$ is announced as a k-MAR (line 14). Observe that in every string T there is always a k-MAR that consists of n position. This case is handled as follows: an extra letter, \$, (that is not included in the alphabet) is added to the text in position $n+1$ (line 2). This way, all k-MARS, including the rightmost one, will be announced by the procedure.

An example of the procedure run is demonstrated in figure 2.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$T[i]$	a	b	b	a	c	a	b	a	a	c	a	b	b	a	b	b	a	a	b	c	b	a	.
($\ell=1$) a	1			2		**1**	2	**1**	2	2	2												
($r=12$) b		1	1		1	1	**1**	1	**1**	**1**	**1**												
c					**1**			**1**		**1**	**1**												
win	a	b	b	a	b	b	a	b	**a**	a	**a**												
$T[i]$	a	b	b	a	c	a	b	a	a	c	a	b	b	a	b	b	a	a	b	c	b	a	.
($\ell=3$) a	.	.		1		**1**	1	**1**	2	1	2												
($r=12$) b	.	.	1			1	**1**		**1**	**1**													
c	.	.			1			1		**1**	**1**												
win	.	.	b	a	c	b	a	c	**a**	a	**a**												
$T[i]$	a	b	b	a	c	a	b	a	a	c	a	b	b	a	b	b	a	a	b	c	b	a	.
($\ell=3$) a	.	.		1		**1**	1	**1**	2	1	2	2	**1**	3	**2**	**1**	4						
($r=18$) b	.	.	1			1	**1**		**1**	**1**		**2**	2		3	3							
c	.	.			1			1		**1**	**1**		**1**	**1**		**1**	**1**						
win	.	.	b	a	c	b	a	c	**a**	a	**a**	a	**b**	a	**b**	b	a						

Fig. 2: One iteration of finding 5-MAR with period length 3. The first Parikh matrix is computed until 5 modified letters are in use. Then, ℓ is moved to the right until one modified letter is released. Then, r is moved to the right until one modified letter is used. At the end of this iteration the 5-MAR $T[3\,..\,17]$ is announced. In the next iteration, when ℓ is moved to the right ($\ell = 4$), there is a tie between the letters a and b in column 3, therefore, r can be moved to the right without using modified letters until position 20.

Time Complexity: each Parikh matrix entry computation takes $O(1)$, either when adding a letter (moving r index to the right) or when subtracting a letter (moving ℓ to the right). Therefore, the time complexity depends on the number of such updates. For a single period length, p, each position i, $1 \leq i \leq n$, is updated in the Parikh matrix at most twice: when $\ell = i$ and when $r = i$. This gives a total of $O(n)$ time per period. p goes from 1 to $\frac{n}{2}$, which gives a total time complexity of $O(n^2)$ for the entire algorithm.

4 An $O(nk^3 \log \frac{n}{k})$ Algorithm

The general strategy of the improved algorithm is a divide-and-conquer recursive scheme similar to the one used by Main and Lorentz [ML84] for computing squares and to that of [LSS01] for computing approximate repetitions. It works as follows: during the first step, we locate all k-MAR that contain the middle position of T, i.e. $\frac{n}{2}$. This is computed for all possible period lengths p such that $1 \leq p \leq \frac{n}{2}$. Then, the same procedure for finding all k-MAR with period lengths $1 \leq p \leq \frac{n}{4}$ is applied independently on the first half ($T[1\,..\,\frac{n}{2}-1]$) and on the second half of T ($T[\frac{n}{2}+1\,..\,n]$).

High level description of the algorithm is described in algorithm ApproxRep.

APPROXREP$(T, start, end)$

```
1  n ← (end − start)
2  if n > 2k then
3        mid ← start + ⌊n/2⌋
4        for p ← 1 to ⌊n/2⌋ do
5              Find all k-MAR with period length p that contain mid
6        APPROXREP(T, start, mid − 1)
7        APPROXREP(T, mid + 1, end)
```

In the rest of this section we describe the algorithm for finding all k-MAR with period length p that exist on a text T of size n and contains position $\frac{n}{2}$ (line 5 in the above algorithm).

4.1 Initialization Step: Defining the Substring Boundaries

We start with a simple observation regarding the boundaries of the substring that should be handled on each iteration. Let $T[i\,..\,j]$ be a k-MAR that contains position $\frac{n}{2}$. According to observation A1 in section 2, since one modified letter can imply on at most 2 mismatches, there are no more than $2k$ mismatches such that $T[i] \neq T[i+p]$ in the substring $T[i\,..\,j]$. Thus, all k-MAR substrings that contain position $\frac{n}{2}$ must start at a position that is right to the position of the $2k+2$ mismatch going from $\frac{n}{2}$ to the left on the reversed substring of T. More precisely, if x is the position of the $2k+1$ mismatch going from $\frac{n}{2}$ to the left, then i must be right to position $x-p$ (this is due to the fact that by definition, a mismatch between two positions, i and $i+p$, is marked in $i+p$ position and not in i). Let ℓ_1 be the position $x-p$. In addition, j must be left to the position of the $2k+1$ mismatch going from $\frac{n}{2}$ to the right. Let r_1 be the position of this mismatch. Any extension to these positions cannot contain the position $\frac{n}{2}$ in the k-MAR.

As an initialization step, the algorithm uses the technique described in [LSS01], using [LV89] and [GG86], in order to find $2k+1$ mismatches between two copies of T shifted by p positions starting at index $\frac{n}{2}$ going right, and $2k+1$ mismatches starting at index $\frac{n}{2}$ going left. This is done by constructing two suffix trees (or suffix arrays), one for the string T and one for its reversed string T^R, and using the "kangaroo" jumps of [GG86] (i.e., using suffix trees and LCA algorithm for a constant time "jump" over equal substrings of the aligned copies of T) in order to find the positions of $2k+1$ mismatches between the suffixes $T^R[\frac{n}{2}\,..\,n-p]$ and $T^R[\frac{n}{2}+p\,..\,n]$, and the positions of $2k+1$ mismatches between the substrings $T[\frac{n}{2}\,..\,n-p]$ and $T[\frac{n}{2}+p\,..\,n]$.

Observe that the mismatches define the *problematic columns* that should be handled in the current iteration over p. A modified letter can only be used in one of these columns. Hence, there are at most $4k+2 = O(k)$ such columns.

We use two sorted lists: $ColumnList$ that contains the problematic columns in the substring $T[\ell_1\,..\,r_1]$, and $MismatchList$ that contains the mismatch positions in the substring. The mismatch positions are in-

serted in a sorted manner to the $MismatchList$, such that m_1 is the leftmost mismatch. In addition, $ColumnList$ is updated with all problematic columns as follows: for each mismatch in position i, we add its column to $ColumnList$.

Time Complexity: in order to find all $4k + 2$ mismatches, two suffix trees are constructed. This is done once for the entire algorithm in $O(n)$ time. The algorithm for finding $O(k)$ mismatch positions between the substrings is done in $O(k)$ time, including the update of both $MismatchList$ and $ColumnList$.

4.2 Main Procedure: Finding All k-MAR in the Substring

The main procedure of the algorithm uses both $MismatchList$ and $ColumnList$ in order to find all k-MAR in between the boundaries ℓ_1 and r_1. It is similar to the simple algorithm described in section 3 which uses the Parikh matrix in order to find k-MAR, and that it keeps two pointers on the string, ℓ and r, that are increased in turns. The difference between the two algorithms is that in this algorithm we take advantage of the fact that not all positions in the substring need to be visited, the only relevant positions are the ones of the *problematic columns* in all periods.

Observe that the mismatch positions in $MismatchList$ divide the text $T[\ell_1 \,..\, r_1]$ into $4k + 1$ adjacent non overlapping substrings. We denote each such substring as a *zone*. The leftmost zone is the substring $T[\ell_1 \,..\, m_1 - 1]$, the second zone is the substring $T[m_1 \,..\, m_2 - 1]$, and so on. Each *zone* presents either an exact run (since there are no mismatches in it), or a prefix of an exact run (if the zone's length is smaller than $2p$). Note that because the mismatch position of $(i, i + p)$ is marked in position $i + p$ by definition, these exact runs are not maximally extended - they can be extended to the left by $p - 1$ letters.

Consider a possible k-MAR $T[\ell + 1 \,..\, r - 1]$, such that $\ell_1 < \ell < \frac{n}{2}$ and $\frac{n}{2} < r < r_1$: from the definition of k-MAR we get that both substrings $T[\ell \,..\, r - 1]$ and $T[\ell + 1 \,..\, r]$ contain $k + 1$ modified letters (for otherwise, the approximate run is not maximally extended). We denote the *zone* in which ℓ is contained as L and the *zone* in which r is contained as R. Let m_i be the mismatch position in L zone, and let m_j be the mismatch position in R zone (see figure 3).

We continue with an observation regarding the possible positions of ℓ and r:

Observation 1 *ℓ position can only be on the $k + 1$ rightmost periods in L zone. r position can only be on the $k + 1$ leftmost periods in R zone. Denote these periods as the $(k + 1)$-periods.*

Proof. We prove this property by contradiction: assume that L zone contains $z > k + 1$ periods and that ℓ position is left to the rightmost $k+1$ periods in L. We consider two cases regarding the problematic columns in L:
Case 1: there exists at least one letter in L that is a non-winner letter of its column - this means that the number of modified letters in $T[\ell + 1 \,..\, r - 1]$ exceeds $k + 1$, a contradiction.
Case 2: all letters in L are winners of their columns - note that the substring $T[m_i - p + 1 \,..\, m_{i+1} - 1]$ is an exact run, therefore, in order for $T[\ell + 1 \,..\, r - 1]$ to be maximally extended, ℓ has to be equal to $m_i - p$. This position is not in L zone (and it was visited on an earlier iteration), a contradiction. In a similar way, it is easy to show that r position can only be on the leftmost $k + 1$ periods in R.

Recall that a *problematic column* is a column that contains at least two different letters in the substring $T[\ell_1 \,..\, r_1]$. We continue with a definition of a *problematic position*: a problematic position is a position in the substring T such that its column is a problematic column and it is in the $(k + 1)$-periods of L or R zones. There are a total of $O(k)$ zones and each one contains $O(k)$ periods that needs to be visited. Each period contains $O(k)$ problematic columns. Hence, the total number of problematic positions in the substring is $O(k^3)$. Note that a problematic position is a position in which a modified letter might be used. These positions are the positions that ℓ and r are assigned to in the improved algorithm.

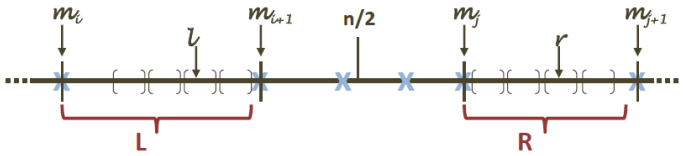

Fig. 3: L, R zones, and the positions ℓ and r in them. The mismatch positions are marked with 'X' and the boundaries of the substrings are marked with vertical lines. For $k = 3$, the relevant periods in each of the zones are marked with brackets.

The algorithm works in a similar way to the simple $O(n^2)$ algorithm with the two following differences: first, the Parikh matrix contains only the problematic columns of $ColumnList$. The second difference is that on each iteration over ℓ and r, their position is increased to the next *problematic position* (and not necessarily increased by 1).

This complicates the Parikh matrix update procedure, and we distinguish between two cases of it:
Case 1: moving a position inside a zone: moving ℓ in L (or r in R) updates only one column in the Parikh matrix, which is decreased (or increased) by at most 1.
Case 2: moving a position in between zones: assume that ℓ is moved from L zone to L' zone on its right. Suppose that L' contains $q > k+1$ periods, then ℓ position is increased to the position of the first problematic column in L' such that ℓ is in the rightmost $k+1$ periods of L'. All problematic columns in the Parikh matrix need to be updated according to the number of times the column occurs in the leftmost $q-(k+1)$ periods of L'. The situation is similar when moving r from R zone to R' zone: if R contains more than $k+1$ periods, then the increase of r position should update the entire Parikh matrix and compute the total number of modified letters used in the new substring.

The procedure stops when either $\ell > \frac{n}{2}$ or $r = r_1$.

Time Complexity: each position move of either ℓ or r updates the Parikh matrix. If the position move is in the same zone, the update takes $O(1)$ time. If a position is increased to a different zone, the entire Parikh matrix is computed, which takes $O(k)$ time. On each zone there are at most $O(k^2)$ positions, and there are a total of $O(k)$ zones. This gives a total of $O(k^3)$ updates in the same zone and a total of $O(k)$ updates of zone changing. Therefore, the total time complexity of the main procedure is $O(k^3)$.

4.3 Total Time Complexity Analysis

Finding the initial boundaries of the substring takes $O(k)$ time. The time complexity of the main procedure is $O(k^3)$. The total time complexity for finding all k-MAR with period length p that exist in a substring T is $O(k^3)$. There are at most $O(n \log \frac{n}{k})$ iterations in the ApproxRep algorithm, and each iteration is done in $O(k^3)$ time, which gives a total time complexity of $O(nk^3 \log \frac{n}{k})$ for the entire algorithm.

In the next section, we describe an algorithm that improves the time complexity of the main procedure: instead of going over all $O(k^2)$ possible problematic positions for ℓ and r in each zone, the algorithm only visits $O(k \log k)$ such positions.

5 An improved $O(nk^2 \log k \log \frac{n}{k})$ Algorithm

In this algorithm we improve the main procedure of the previous algorithm. The main idea behind the improvement is that there are still redundant positions that were visited in the previous algorithm: in this algorithm we only visit positions in L and in R that are the positions of modified letters. In addition, we visit each one of the $(k+1)$-periods in L. The iteration over ℓ and r positions and the update of the Parikh

matrix according to them is done in a similar way to the previous algorithm. The only difference is in the initialization and the handling of the relevant positions that need to be visited.

Let $T[\ell + 1 \,..\, r - 1]$ be a k-MAR, and let m_i, m_j be the mismatch positions in L and R, respectively. We denote the substring $T[m_{i+1} \,..\, m_j - 1]$ as M (see figure 4).

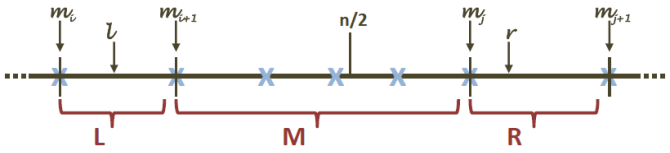

Fig. 4: L, M and R zones, and the positions ℓ and r in them. The mismatch positions are marked with 'X' and the boundaries of the substrings are marked with vertical lines.

Note that the k modified letters in the k-MAR are spread among L, M and R.

In order to handle only the relevant problematic positions in the substring, the following sub procedures are added to the previous algorithm: at first, when either ℓ or r reaches the leftmost position of its zone, an initialization is done, in which initial positions that need to be visited on L and R are found. Second, when iterating over ℓ position - on each one of the $(k + 1)$-periods in L, additional columns are added to the list of problematic columns that are relevant for the rest of the iteration. The third sub procedure is added to the iteration over r position - problematic columns are added or removed from the lists of problematic columns in L and R.

5.1 Sub Procedure 1: Updating the Problematic Positions

This sub procedure is called whenever ℓ or r moves between zones. Then, the substrings L, M and R are redefined according to the substring $T[\ell \,..\, r]$, and the procedure begins.

Note that there are two different options for positions ℓ and r: either ℓ is on the $(k + 1)$'th period of L and r is a position in between the leftmost $k + 1$ periods in R, or ℓ is a position in between the $k + 1$ rightmost periods in L and r is the mismatch position of R zone. In both options, the Parikh matrix is updated according to the substring $T[\ell \,..\, r]$.

The algorithm uses two lists, $LeftList$ and $RightList$, that keep the problematic columns that need to be visited in L and in R, respectively. In addition, for each period in the $(k + 1)$-periods of L, we keep a list, $newColumnList$, that contains added problematic columns that need to be visited in L starting this period.

Let c be a problematic column, and let x be the letter of column c in L, having $|x|$ occurrences in $T[\ell \,..\, r]$. Let y be the letter of c in R having $|y|$ occurrences in $T[\ell \,..\, r]$, and z be the letter with the maximum number of occurrences, $|z|$, in c column in M.

We first define the positions in L that need to be visited by the algorithm. The following cases regarding the majority of x in the initial substring $T[\ell \,..\, r]$ are considered:

- Case 1: x is the loser of its column - in this case, the column c is added to $LeftList$, since it implies on modified letters. Note that in the case where $x = y$, a losing letter might gain majority and become a winner of its column.
- Case 2: x is the winner of its column, but might lose its majority. This case can be either one of the following two sub cases (or both):
  - x might lose its majority to y - this case can happen when $|y| < |x| \leq |y| + 2k$. The column c is added to $newColumnList$ of the $|y|$'th period of L.
  - x might lose its majority to z - this case can happen when $|z| < |x| \leq |z| + k$. The column c is added to $newColumnList$ of the $|z|$'th period of L.

Note that in the case that x can lose to both y and z, the column c is added to $newColumnList$ of the $max\{|y|, |z|\}$ period of L only.
- Case 3: x is the winner of its column and will win throughout the entire iteration - in this case, the column is not added to any list.

The number of columns in both $LeftList$ and $newColumnList$ cannot exceed $O(k)$, since there are at most $O(k)$ problematic columns in $T[\ell \,..\, r]$. Furthermore, in the initialization step, each column in $LeftList$ implies on a column that will be visited throughout the entire iteration over ℓ. The number of such visited positions cannot exceed $O(k)$, since they are the positions of the modified letters in L. Also note that each column in $newColumnList$ of a period i of L implies on i visited positions in L. These are the positions that x might lose its majority to either z or y. The number of such visited positions cannot exceed $O(k)$ either, since these positions imply on modified letters in either M or R. Thus, the total number of the initial visited positions in L is bounded by $O(k)$.

We continue with the visited positions in R. We again consider the following cases regarding the majority of y in the initial substring $T[\ell \,..\, r]$:

- Case 1: y is the loser of its column - in this case, the column c is added to $RightList$, since it may imply on modified letters.
- Case 2: y is the winner of its column, but might lose its majority to z - this case can only happen when $y = x$ and $|z| < |y| \leq |z| + k$. The column c is added to $RightList$, with a special flag that says that this column needs to be visited until r reaches the $|z|$'th period of R.
- Case 3: y is the winner of its column and will win throughout the entire iteration - in this case, the column is not added to any list.

The number of columns in $RightList$ is bounded by $O(k)$.

Time Complexity: each column is added to a list in $O(1)$ time and there are $O(k)$ such columns, which gives a total time complexity of $O(k)$ for the update phase.

5.2 Sub Procedure 2: Period Handling in L Zone

This step is called whenever ℓ position moves from one period to the one on its right (in the same zone). The procedure is very simple: it goes over all columns in $newColumnList$, and adds them to $LeftList$. The total number of columns that can be added throughout the entire iteration over ℓ in L is $O(k)$.

Time Complexity: each column is added to $LeftList$ in a sorted manner in $O(\log k)$ time and there are $O(k)$ such columns, which gives a total time complexity of $O(k \log k)$ for all period handling steps.

5.3 Sub Procedure 3: Iterating Over r in R and ℓ in L

In this algorithm, in a similar way to the algorithm described in section 4, positions ℓ and r are increased to the next problematic positions. The difference is that in this algorithm, the problematic positions are taken from $LeftList$ and $RightList$. On each position visit, the Parikh matrix is updated in the same manner as before.

This procedure is called whenever r position is increased. Let c be the problematic column of position r, and let y be the letter in this column. Since the number of occurrences of y was increased by one, the two following cases need to be checked:

Adding a column to $LeftList$: if y is not the winner of its column and it loses to the letter x in L, then a new wakeup call needs to be added to the respected period in L: column c is added to $newColumnList$ of the $|y|$'th period of L, and is removed from the previous $newColumnList$ (on period $|y|-1$ in L), if such exists. Each insertion is done in $O(1)$ time. 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 $O(\log k)$ time.

Removing a column from $RightList$: if y is the winner of its column, we consider the two cases: $x = y$ and $x \neq y$. If $x \neq y$, it means that y will continue to win throughout the entire iteration, and therefore its column is removed from $RightList$. If $x = y$, the situation is a bit more complicated since y can again lose its majority as ℓ position is increased. Therefore, before the column is removed from $RightList$, a check to see whether r position is on a period greater than $|z|$. If yes, the column is removed from the list. Otherwise, it is not removed, and will be visited on the next period, as well. Each deletion from $RightList$ is done in a sorted manner in $O(\log k)$ time.

Time Complexity: adding a column to $newColumnList$ is done in $O(1)$ time, since the list does not need to be sorted. adding or removing a column from $LeftList$ and $RightList$ in a sorted manner is done in $O(\log k)$ time. But, each problematic column is added or removed from the lists at most once, and there are $O(k)$ such columns. This gives a total time complexity of $O(k \log k)$ for the entire iteration over r in R and ℓ in L. In addition, each Parikh matrix update is done in $O(1)$ time (when ℓ and r stay in the same zone), and as proved in Lemma 1 below, there are at most $O(k \log k)$ such updates for both ℓ and r.

Recall that the iteration over ℓ and r visits the following positions: for a letter x in L, as long as x is the winner of its column, the algorithm visits only the periods in L, in which it might lose its majority to z (or y). In a similar way, for a letter y in R, as long as y is the winner of its column, the algorithm visits only the periods in R, in which it might lose its majority to z (this can only happen when $x = y$). If x (or y) is the loser of its column, then all the positions in which it loses (there are at most $k+1$ such positions) are visited.

Note that although new columns are never added to $RightList$, additional columns are added to $LeftList$ from $newColumnList$. This means that the number of positions visited in L (and as a result in R) can be greater than $O(k)$. Lemma 1 proves that this number is bounded by $O(k \log k)$:

Lemma 1. *The total number of visited positions in L and in R is bounded by $O(k \log k)$.*

Proof. We start by counting the number of visited positions in R, and distinguish between the two cases when $x \neq y$ and when $x = y$:

- Case 1: $x \neq y$
  Assume that on the initial substring $T[\ell \,..\, r]$, there are no modified letters in R, and as the iteration over ℓ and r continues, all k modified letters are moved into R. In this situation, the number of visited positions in R is k. Additional visited positions are added to R when a loser letter y becomes a winner, this way r can be extended to the right without using more than k modified letters.
  Suppose that when r is increased to the next problematic position, y becomes a winner of its column. Let a be the number of problematic columns in R ($a \leq k$), then k/a additional modified letters can be visited in R. Now, there are at most $a-1$ problematic columns in R. The next time it happens, at most $k/(a-1)$ modified letters that can be visited are added to R, and so on. Thus, the maximal number of problematic positions in R is equal to $\Sigma_{i=1}^{a} k/i$, which gives a total of $O(k \log k)$.
- Case 2: $x = y$
  If y becomes a winner in its column, it does not necessarily mean that it will continue to win throughout the iteration, as ℓ position is increased. This situation is handled in the initialization step (see case 2), and the column is checked until the number of y occurrences is greater than $|z|$. As mentioned above, the

total number of such visited positions cannot exceed $O(k)$, as it is bounded by the number of modified letters used in M.

Thus, the total number of visited positions in R cannot exceed $O(k \log k)$. In a similar way, it is easy to show that the total number of visited positions in L cannot exceed $O(k \log k)$.

5.4 Total Time Complexity Analysis

The difference between this algorithm and the previous $O(nk^3 \log \frac{n}{k})$ algorithm is in the number of visited positions of the main procedure. For each L and R possible substring definition, the algorithm visits at most $O(k \log k)$ positions, and updates the column lists of L and R in $O(k \log k)$ time. There are at most $O(k)$ such combinations of L and R, which gives a $O(k^2 \log k)$ time complexity for this step. In addition, moving between zones takes $O(k)$ time, which does not add to the total time. Thus, the total time complexity of the main procedure of the algorithm is $O(k^2 \log k)$. This gives a total time complexity $O(nk^2 \log k \log \frac{n}{k})$ for the entire algorithm.

References

[AEL10] A. Amir, E. Eisenberg, and A. Levy. Approximate periodicity. In *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I*, volume 6506, pages 25–36. Springer, 2010.

[AP83] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theor. Comput. Sci.*, 22:297–315, 1983.

[CHL07] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. 392 pages.

[CI08] 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.

[Cro81] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[Cro83] M. Crochemore. Recherche linéaire d'un carré dans un mot. *C. R. Acad. Sc. Paris Sér. I Math.*, 296(18):781–784, 1983.

[GG86] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17(4):52–54, 1986.

[GS04] 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.

[IMS97] 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.

[KK99] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Foundations of Computer Science*, pages 596–604, 1999.

[Kos94] S. Rao Kosaraju. Computation of squares in a string (preliminary version). In *CPM*, pages 146–150, 1994.

[LSS01] G. M. Landau, J. P. Schmidt, and D. Sokol. An algorithm for approximate tandem repeats. *Journal of Computational Biology*, 8(1):1–18, 2001.

[LV89] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10:157–169, 1989.

[Mai89] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.

[ML84] 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.

[Par66] R. J. Parikh. On context-free languages. *Journal of the ACM*, 13:570–581, 1966.

[SIPS99] 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.