Online Computation of Abelian Runs

Gabriele Fici[1], Thierry Lecroq[2], Arnaud Lefebvre[2], and Élise Prieur-Gaston[2]

[1] Dipartimento di Matematica e Informatica, Università di Palermo, Italy
Gabriele.Fici@unipa.it
[2] Normandie Université, LITIS EA4108, NormaStic CNRS FR 3638, IRIB, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France
{Thierry.Lecroq,Arnaud.Lefebvre,Elise.Prieur}@univ-rouen.fr

Abstract. Given a word w and a Parikh vector $\mathcal{P}$, an abelian run of period $\mathcal{P}$ in w is a maximal occurrence of a substring of w having abelian period $\mathcal{P}$. We give an algorithm that finds all the abelian runs of period $\mathcal{P}$ in a word of length n in time $O(n \times |\mathcal{P}|)$ and space $O(\sigma + |\mathcal{P}|)$.

Keywords: Combinatorics on Words, Text Algorithms, Abelian Period, Abelian Run

1 Introduction

Computing maximal (non-extendable) repetitions in a string is a classical topic in the area of string algorithms (see for example [7] and references therein). Detecting maximal repetitions of substrings, also called *runs*, gives information on the repetitive regions of a string, and is used in many applications, for example in the analysis of genomic sequences.

Kolpakov and Kucherov [5] gave a linear time algorithm for computing all the runs in a word and conjectured that any word of length n contains less than n runs. Bannai et al. [1] recently proved this conjecture using the notion of Lyndon root of a run.

Here we deal with a generalization of this problem to the commutative setting. Recall that an abelian power is a concatenation of two or more words that have the same Parikh vector, i.e., that have the same number of occurrences of each letter of the alphabet. For example, *aababa* is an abelian square, since *aab* and *aba* both have 2 a's and 1 b. When an abelian power occurs within a string, one can search for its "maximal" occurrence by extending it to the left and to the right character by character without violating the condition on the number of occurrences of each letter. Following the approach of Constantinescu and Ilie [2], we say that a Parikh vector $\mathcal{P}$ is an abelian period for a word w over a finite ordered alphabet $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ if w can be written as $w = u_0 u_1 \cdots u_{k-1} u_k$ for some $k > 2$ where for $0 < i < k$ all the u_i's have the same Parikh vector $\mathcal{P}$ and the Parikh vectors of u_0 and u_k are contained in $\mathcal{P}$. Note that the factorization above is not necessarily unique. For example, $a \cdot bba \cdot bba \cdot \varepsilon$ and $\varepsilon \cdot abb \cdot abb \cdot a$ (ε denotes the empty word) are two factorizations

of the word *abbabba* both corresponding to the abelian period $(1, 2)$. Moreover, the same word can have different abelian periods.

In this paper we define an *abelian run* of period $\mathcal{P}$ in a word w as an occurrence of a substring v of w such that v has abelian period $\mathcal{P}$ and this occurrence cannot be extended to the left nor to the right by one letter into a substring having the same abelian period $\mathcal{P}$.

For example, let $w = ababaaa$. Then the prefix $ab \cdot ab \cdot a = w[1..5]$ has abelian period $(1, 1)$ but it is not an abelian run since the prefix $a \cdot ba \cdot ba \cdot a = w[1..6]$ has also abelian period $(1, 1)$. This latter, instead, is an abelian run of period $(1, 1)$ in w.

Looking for abelian runs in a string can be useful to detect those regions in a string in which there is some kind of non-exact repetitiveness, for example regions in which there are several consecutive occurrences of a substring or its reverse.

Matsuda et al. [6] recently presented an offline algorithm for computing all abelian runs of a word of length n in $O(n^2)$ time. Notice that, however, the definition of abelian run in [6] is slightly different from the one we consider here. We will comment on this in Section 3.

We present an online algorithm that, given a word w of length n over an alphabet of cardinality σ, and a Parikh vector $\mathcal{P}$, returns all the abelian runs of period $\mathcal{P}$ in w in time $O(n \times |\mathcal{P}|)$ and space $O(\sigma + |\mathcal{P}|)$.

2 Definitions and Notation

Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be a finite ordered alphabet of cardinality σ and let Σ^* be the set of finite words over Σ. We let $|w|$ denote the length of the word w. Given a word $w = w[0..n-1]$ of length $n > 0$, we write $w[i]$ for the $(i+1)$-th symbol of w and, for $0 \leqslant i \leqslant j < n$, we write $w[i..j]$ for the substring of w from the $(i+1)$-th symbol to the $(j+1)$-th symbol, both included. We let $|w|_a$ denote the number of occurrences of the symbol $a \in \Sigma$ in the word w.

The *Parikh vector* of w, denoted by $\mathcal{P}_w$, counts the occurrences of each letter of Σ in w, that is, $\mathcal{P}_w = (|w|_{a_1}, \ldots, |w|_{a_\sigma})$. Notice that two words have the same Parikh vector if and only if one word is a permutation (i.e., an anagram) of the other.

Given the Parikh vector $\mathcal{P}_w$ of a word w, we let $\mathcal{P}_w[i]$ denote its i-th component and $|\mathcal{P}_w|$ its norm, defined as the sum of its components. Thus, for $w \in \Sigma^*$ and $1 \leqslant i \leqslant \sigma$, we have $\mathcal{P}_w[i] = |w|_{a_i}$ and $|\mathcal{P}_w| = \sum_{i=1}^{\sigma} \mathcal{P}_w[i] = |w|$.

Finally, given two Parikh vectors $\mathcal{P}, \mathcal{Q}$, we write $\mathcal{P} \subset \mathcal{Q}$ if $\mathcal{P}[i] \leqslant \mathcal{Q}[i]$ for every $1 \leqslant i \leqslant \sigma$ and $|\mathcal{P}| < |\mathcal{Q}|$.

Definition 1 (Abelian period [2]). *A Parikh vector $\mathcal{P}$ is an abelian period for a word w if $w = u_0 u_1 \cdots u_{k-1} u_k$, for some $k > 2$, where $\mathcal{P}_{u_0} \subset \mathcal{P}_{u_1} = \cdots = \mathcal{P}_{u_{k-1}} \supset \mathcal{P}_{u_k}$, and $\mathcal{P}_{u_1} = \mathcal{P}$.*

Note that since the Parikh vector of u_0 and u_k cannot be included in $\mathcal{P}$ it implies that $|u_0|, |u_k| < |\mathcal{P}|$. We call u_0 and u_k respectively the *head* and the

tail of the abelian period. Note that in [2] the abelian period is characterized by $|u_0|$ and $|\mathcal{P}|$ thus we will sometimes use the notation (h, p) for an abelian period of norm p and head length h of a word w. Notice that the length t of the tail is uniquely determined by h, p and $n = |w|$, namely $t = (n - h) \bmod p$.

Definition 2 (Abelian repetition). *A substring $w[i..j]$ is an abelian repetition with period length p if $i - j + 1$ is a multiple of p, $i - j + 1 \geq 2p$ and there exists a Parikh vector $\mathcal{P}$ of norm p such that $\mathcal{P}_{w[i+kp..i+(k+1)p-1]} = \mathcal{P}$ for every $0 \leq k \leq p/(i - j + 1)$.*

An abelian repetition $w[i..j]$ with period length p such that $i - j + 1 = 2p$ is called an abelian square. An abelian repetition $w[i..j]$ of period length p of a string w is maximal if:

1. $\mathcal{P}_{w[ip..i1]} \neq \mathcal{P}_{w[i..i+p1]}$ or $ip < 0$;
2. $\mathcal{P}_{w[jp+1..j]} \neq \mathcal{P}_{w[j+1..j+p]}$ or $j + p \geq n$.

We now give the definition of an abelian run. Let $v = w[b..e]$, $0 \leq b \leq e \leq |w| - 1$, be an occurrence of a substring in w and suppose that v has an abelian period $\mathcal{P}$, with head length h and tail length t. Then we denote this occurrence by the tuple (b, h, t, e).

Definition 3. *Let w be a word. An occurrence (b, h, t, e) of a substring of w starting at position b, ending at position e, and having abelian period $\mathcal{P}$ with head length h and tail length t is called **left-maximal** (resp. **right maximal**) if there does not exist an occurrence of a substring $(b-1, h', t', e)$ (resp. $(b, h', t', e+1)$) with the same abelian period $\mathcal{P}$. An occurrence (b, h, t, e) is called **maximal** if it is both left-maximal and right-maximal.*

This definition leads to the one of abelian run.

Definition 4. *An **abelian run** is a maximal occurrence (b, h, t, e) of a substring with abelian period $\mathcal{P}$ of norm p such that $(e - b - h - t + 1) \geq 2p$ (see Fig. 1).*

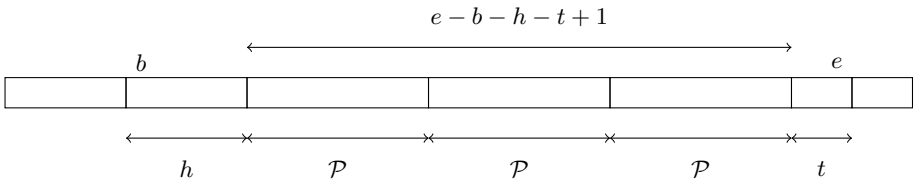

Fig. 1. The tuple (b, h, t, e) denotes an occurrence of a substring starting at position b, ending at position e, and having abelian period $\mathcal{P}$ with head length h and tail length t.

The next result limits the number of abelian runs starting at each position in a word.

Lemma 5. *Let w be a word. Given a Parikh vector $\mathcal{P}$, there is at most one abelian run with abelian period $\mathcal{P}$ starting at each position of w.*

Proof. If two abelian runs start at the same position, the one with the shortest head cannot be maximal. □

Corollary 6. *Let w be a word. Given a Parikh vector $\mathcal{P}$, for every position i in w there are at most $|\mathcal{P}|$ abelian runs with period $\mathcal{P}$ overlapping at i.*

The next lemma shows that a left-maximal abelian substring at the right of another left-maximal abelian substring starting at position i in a word w cannot begin at a position smaller than i.

Lemma 7. *If $(b_1, h_1, 0, e_1)$ and $(b_2, h_2, 0, e_2)$ are two left-maximal occurrences of substrings with the same abelian period $\mathcal{P}$ of a word v such that $e_1 < e_2$ and $b_1 > e_1 - 2 \times |\mathcal{P}| + 1$ and $b_2 > e_2 - 2 \times |\mathcal{P}| + 1$, then $b_1 \leq b_2$.*

Proof. If $b_2 < b_1$ then since $e_2 > e_1$, $w[b_1..b_1 + h_1 - 1]$ is a substring of $w[b_2..b_2 + h_2 - 1]$. Thus $\mathcal{P}_{w[b_1..b_1+h_1-1]} \subset \mathcal{P}_{w[b_2..b_2+h_2-1]} \subset \mathcal{P}$ which implies that $\mathcal{P}_{w[b_1-1..b_1+h_1-1]} \subset \mathcal{P}$ meaning that $(b_1, h_1, 0, e_1)$ is not left-maximal: a contradiction. □

We recall the following proposition, which shows that if we can extend the abelian period with the longest tail of a word w when adding a symbol a, then we can extend all the other abelian periods with shorter tail.

Proposition 8 ([4]). *Suppose that a word w has s abelian periods $(h_1, p_1) < (h_2, p_2) < \cdots < (h_s, p_s)$ such that $(|w| - h_i) \mod p_i = t > 0$ for every $1 \leq i \leq s$. If for a letter $a \in \Sigma$, (h_1, p_1) is an abelian period of wa, then $(h_2, p_2), \ldots, (h_s, p_s)$ are also abelian periods of wa.*

We want to give an algorithm that, given a string w and a Parikh vector $\mathcal{P}$, returns all the abelian runs of w having abelian period $\mathcal{P}$.

3 Previous Work

In [6], the authors presented an algorithm that computes all the abelian runs of a string w of length n in $O(n^2)$ time and space complexity. They consider that a substring $w[i-h..j+t]$ is an abelian run if $w[i..j]$ is a maximal abelian repetition with period length p and $h, t \geq 0$ are the largest integers satisfying $\mathcal{P}_{w[i-h..i-1]} \subset \mathcal{P}_{w[i..i+p-1]}$ and $\mathcal{P}_{w[j+1..j+t]} \subset \mathcal{P}_{w[i..i+p-1]}$. Their algorithm works as follows. First, it computes all the abelian squares using the algorithm of [3]. For each $0 \leq i \leq n-1$, it computes a set L_i of integers such that

$$L_i = \{j \mid \mathcal{P}_{w[i-j..i]} = \mathcal{P}_{w[i+1..i+j+1]}, 0 \leq j \leq \min\{i+1, n-i\}\}.$$

The L_i's are stored in a two-dimensional boolean array L of size $\lfloor n/2 \rfloor \times (n-1)$: $L[j, i] = 1$ if $j \in L_i$ and $L[j, i] = 0$ otherwise. An example of array L is given in

Figure 2. All entries in L are initially unmarked. Then, for each $1 \leq j \leq \lfloor n/2 \rfloor \times$ all maximal abelian repetitions of period length j are computed in $O(n)$. The j-th row of L is scanned in increasing order of the column index. When an unmarked entry $L[j,i] = 1$ is found then the largest non-negative integer k such that $L[j, i + pj + 1] = 1$, for $1 \leq p \leq k$, is computed. This gives a maximal abelian repetition with period length j starting at position $i - j + 1$ and ending at position $i + (k + 1)j$. Meanwhile all entries $L[j, i + pj + 1]$, for $-1 \leq p \leq k$, are marked. Thus all abelian repetitions are computed in $O(n^2)$ time. It remains to compute the length of their heads and tails. This cannot be done naively otherwise it would lead to a $O(n^3)$ time complexity overall. Instead, for each $0 \leq i \leq n-1$, let T_i be the set of positive integers such that for each $j \in T_i$ there exists a maximal abelian repetition of period j and starting at position $i-j+1$. Elements of T_i are processed in increasing order. Let j_k denote the k-th smallest element of T_i. Let h_k denote the length of the head of the abelian run computed from the abelian repetition of period j_k. Then h_k can be computed from h_{k-1}, j_{k-1} and j_k as follows. Two cases can arise:

1. If $k = 0$ or $j_{k-1} + h_{k-1} \leq j_k$, then h_k can be computed by comparing the Parikh vector $\mathcal{P}_{w[i-j_k-p..i-j_k]}$ for increasing values of $\mathcal{P}$ from 0 up to $h_k + 1$, with the Parikh vector $\mathcal{P}_{w[i-j_k+1..i]}$.
2. If $j_{k-1} + h_{k-1} > j_k$, then
   $\mathcal{P}_{w[i-j_{k-1}-h_k..i-j_k]}$ can be computed from $\mathcal{P}_{w[i-j_{k-1}-h_{k-1}+1..i-j_{k-1}]}$. Then, h_k is computed by comparing the Parikh vector $\mathcal{P}_{w[i-j_{k-1}-h_{k-1}+1-p..i-j_k]}$ for increasing values of p from 0 up to $h_k + j_k - h_{k-1} - j_{k-1} + 1$.

This can be done in $O(n)$ time. The lengths of the tails can be computed similarly. Overall, all the runs can be computed in time and space $O(n^2)$.

	a	b	a	a	b	a	b	a	a	b	b	b
	0	1	2	3	4	5	6	7	8	9	10	11
1	0	0	1	0	0	0	0	1	0	1	1	0
2	0	0	0	0	1	1	0	1	0	0	0	0
3	0	0	1	0	1	1	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 2. An example of array L for $w = \texttt{abaababaabbb}$. $L_{4,6} = 1$ which means that $\mathcal{P}_{w[3..6]} = \mathcal{P}_{w[7..10]}$.

This previous method works offline: it needs to know the whole string before reporting any abelian run. We will now give what we call an online method meaning that we will be able to report the abelian runs ending at position $i-1$ of a string w when processing position i. However, this method is restricted to a given Parikh vector.

4 A Method for Computing Abelian Runs of a Word with a Given Parikh Vector

4.1 Algorithm

Positions of w are processed in increasing order. Assume that when processing position i we know all the, at most $|\mathcal{P}|$, abelian substrings ending at position $i-1$. At each position i we checked if $\mathcal{P}_{w[i-|\mathcal{P}|+1..i]} = \mathcal{P}$ then all abelian substrings ending at position $i-1$ can be extended and thus become abelian substrings ending at position i. Otherwise, if $\mathcal{P}_{w[i-|\mathcal{P}|+1..i]} \neq \mathcal{P}$ then abelian substrings ending at positions $i-1$ are processed in decreasing order of tail length. When an abelian substring cannot be extended it is considered as an abelian run candidate. As soon as an abelian substring ending at position $i-1$ can be extended then all the others (with smaller tail length) can be extended: they all become abelian substrings ending at position i. At most one candidate (with the smallest starting position) can be output at each position.

4.2 Implementation

The algorithm $\textsc{Runs}(\mathcal{P}, w)$ given below computes all the abelian runs with Parikh vector $\mathcal{P}$ in the word w. It uses:

- function $\textsc{Find}(\mathcal{P}, w)$, which returns the ending position of the first occurrence of Parikh vector $\mathcal{P}$ in w or $|w|+1$ if such an occurrence does not exist;
- function $\textsc{FindHead}(w, i, \mathcal{P})$, which returns the leftmost position $j < i$ such that $\mathcal{P}_{w[j..i-1]} \subset \mathcal{P}$ or i is such a substring does not exist;
- function $\textsc{Min}(B)$ that returns the smallest element of the integer array B.

Positions of w are processed in increasing order (Lines 4–21). We will now describe the situation when processing position i of w:

- array B stores the starting positions of abelian substrings ending at position $i-1$ for the different $|\mathcal{P}|$ tail lengths (B is considered as a circular array);
- t_0 is the index in B of the possible abelian substring with a tail of length 0 ending at position i.

All the values of the array B are initially set to $|w|$. Then, when processing position i of w, for $0 \leq k < |\mathcal{P}|$ and $k \neq t_0$, if $B[k] = b < |w|$ then $w[b..i-1]$ is an abelian substring with Parikh vector $\mathcal{P}$ with tail length $((t_0 - k + |\mathcal{P}|) \bmod |\mathcal{P}|) - 1$. Otherwise, if $B[k] = |w|$ then it means that there is no abelian substring in w ending at position $i-1$ with tail length $((t_0 - k + |\mathcal{P}|) \bmod |\mathcal{P}|) - 1$.

The algorithm $\textsc{Runs}(\mathcal{P}, w)$ uses two other functions:

- function $\textsc{GetTail}(tail, t_0, p)$, which returns $(t_0 - tail + p) \bmod p$ which is the length of the tail for the abelian substring ending at position $i-1$ and starting in $B[tail]$;

– function GetRun$(B, tail, t_0, e, p)$, which returns the abelian substring $(B[tail], h, t, e)$.

If $\mathcal{P}_{w[i-|\mathcal{P}|+1..i]} = \mathcal{P}$ (Line 6) then all abelian substrings ending at position $i-1$ can be extended (see Fig. 3). Either this occurrence does not extend a previous occurrence at position $i-|\mathcal{P}|$ (Line 7): the starting position has to be stored in B (Line 8) or this occurrence extends a previous occurrence at position $i-|\mathcal{P}|$ and the starting position is already stored in the array B.

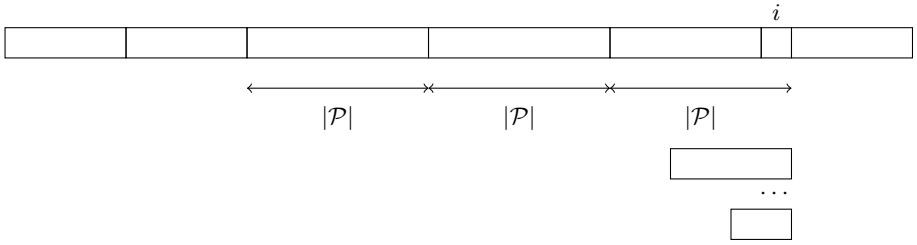

Fig. 3. If $\mathcal{P}_{w[i-|\mathcal{P}|+1..i]} = \mathcal{P}$ then $\mathcal{P}_{w[j..i]} \subset \mathcal{P}$ for $i-|\mathcal{P}|+1 < j < i$.

If $\mathcal{P}_{w[i-|\mathcal{P}|+1..i]} \neq \mathcal{P}$ (Lines 9-21) then abelian substrings ending at position $i-1$ are processed in decreasing order of tail length. To do that, the circular array B is processed in increasing order of index starting from t_0 (Lines 11-19).

Let $tail$ be the current index in array B. At first, $tail$ is set to t_0 (Line 10). In this case there is no need to check if there is an abelian substring with tail length 0 ending at position i (since it has been detected in Line 6) and thus $(B[t_0], h, |\mathcal{P}|-1, i-1)$ is considered as an abelian substring candidate (Line 15) and array B is updated (Line 16) since $(B[t_0], h, 0, i)$ is not an abelian substring.

When $tail \neq t_0$, let $t =$ getTail$(tail, t_0, |\mathcal{P}|)$. If $\mathcal{P}_{w[i-t+1..i]} \not\subset \mathcal{P}$ and thus $(B[tail], h, t, i-1)$ is considered as an abelian substring candidate (Line 15) and array B is updated (Line 16) since $(B[tail], h, t+1, i)$ is not an abelian substring. If $\mathcal{P}_{w[i-t+1..i]} \subset \mathcal{P}$ then, for $tail \leq k \leq (t_0 - 1 + |\mathcal{P}|) \bmod |\mathcal{P}|$, $\exists\, h'_k, t'_k$ such that $(B[k], h'_k, t'_k, i)$ is an abelian substring. It comes directly from Prop. 8.

At each iteration of the loop in Lines 11-19 b is either equal to $|w|$ or to the position of the leftmost abelian run ending at position $i-1$. Thus a new candidate is found if its starting position is smaller than b (Lines 14-15). It comes directly from Lemma 7.

Algorithm 1: getTail$(tail, t_0, p)$

1 **return** $(t_0 - tail + p) \bmod p$

Algorithm 2: GetRun$(B, tail, t_0, e, p)$

```
1  b ← B[tail]
2  if tail = t_0 then
3      t ← p − 1
4  t ← GetTail(tail, t_0, p) − 1
5  h ← (e − t − b + 1) mod p
6  return (b, h, t, e)
```

Algorithm 3: Runs$(\mathcal{P}, w)$

```
1  j ← Find(P, w)
2  (B, t_0) ← (|w|^{|P|}, 0)
3  B[t_0] ← FindHead(w, j − |P| + 1, P)
4  for i ← j + 1 to |w| do
5      t_0 ← (t_0 + 1) mod |P|
6      if i < |w| and P_{w[i−|P|+1..i]} = P then
7          if B[t_0] = |w| then
8              B[t_0] ← FindHead(w, i − |P| + 1, P)
9      else
10         (b, tail) ← (|w|, t_0)
11         repeat
12             if B[tail] ≠ |w| then
13                 if tail = t_0 or i = |w| or P_{w[i−GetTail(tail,t_0,|P|)+1..i]} ⊄ P then
14                     if B[tail] ⩽ b then
15                         (b, h, t, e) ← GetRun(B, tail, t_0, i − 1, |P|)
16                     B[tail] ← |w|
17                 else break
18             tail ← (tail + 1) mod |P|
19         until tail = t_0
20         if min(B) > b and e − t − h − b + 1 > |P| then
21             Output(b, h, t, e)
```

Example

Let us see the behaviour of the algorithm on $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $w = \mathtt{abaababaabbb}$ and $\mathcal{P} = (2, 2)$:

$j = 4, B = (12, 12, 12, 12), t_0 = 0$
$B[0] = 0, B = (0, 12, 12, 12)$
$i = 5$
$t_0 = 1$
$\mathcal{P}_{w[2..5]} \neq \mathcal{P}$
$(b, tail) = (12, 1)$
$tail = 3, 2, 1, 0$
$i = 6$
$t_0 = 2$
$\mathcal{P}_{w[3..6]} = \mathcal{P}$
$B[2] = 0, B = (0, 12, 0, 12)$
$i = 7$
$t_0 = 3$
$\mathcal{P}_{w[4..7]} = \mathcal{P}$
$B[3] = 1, B = (0, 12, 0, 1)$
$i = 8$
$t_0 = 0$
$\mathcal{P}_{w[5..8]} \neq \mathcal{P}$
$(b, tail) = (12, 0)$
$(b, h, t, e) = (0, 1, 3, 7)$
$B[0] = 12, B = (12, 12, 0, 1)$
$tail = 1$
$tail = 2$
$i = 9$
$t_0 = 1$
$\mathcal{P}_{w[6..9]} = \mathcal{P}$
$B[1] = 3, B = (12, 3, 0, 1)$
$i = 10$
$t_0 = 2$
$\mathcal{P}_{w[7..10]} = \mathcal{P}$
$B[2] \neq 12$
$i = 11$
$t_0 = 3$
$\mathcal{P}_{w[8..11]} \neq \mathcal{P}$
$(b, tail) = (12, 3)$
$(b, h, t, e) = (1, 3, 3, 10)$
$B[3] = 12, B = (12, 3, 0, 12)$
$tail = 0$
$tail = 1$
$i = 12$
$t_0 = 0$
$i \geq 12$

$(b, tail) = (12, 0)$
$tail = 1$
$(b, h, t, e) = (3, 3, 2, 11)$
$B[1] = 12, B = (12, 12, 0, 12)$
$tail = 2$
$(b, h, t, e) = (0, 3, 1, 11)$
$B[1] = 12, B = (12, 12, 12, 12)$
$tail = 3$
$tail = 0$
OUTPUT$((0, 3, 1, 11)$

4.3 Correctness and Complexity

Theorem 9. *The algorithm* RUN$(\mathcal{P}, w)$ *computes all the abelian runs with Parikh vector* $\mathcal{P}$ *in a string* w *of length* n *in time* $O(n \times |\mathcal{P}|)$ *and additional space* $O(\sigma + |\mathcal{P}|)$.

Proof. The correctness of the algorithm comes from Corollary 6, Lemma 7 and Prop. 8. The loop in lines 4-21 iterates at most n times. The loop in lines 11-19 iterates at most $|\mathcal{P}|$ times. The instructions in lines 6, 8 and 13 regarding the comparison of Parikh vectors can be performed in $O(n)$ time overall, independently from the alphabet size, by maintaining the Parikh vector of a sliding window of length $|\mathcal{P}|$ on w and a counter r of the number of differences between this Parikh vector and $\mathcal{P}$. At each sliding step, from $w[i - |\mathcal{P}|..i - 1]$ to $w[i - |\mathcal{P}| + 1..i]$ the counters of the characters $w[i - |\mathcal{P}|]$ and $w[i]$ are updated, compared to their counterpart in $\mathcal{P}$ and r is updated accordingly. The additional space comes from the Parikh vector and from the array B, which has $|\mathcal{P}|$ elements. □

5 Conclusions

We gave an algorithm that, given a word w of length n and a Parikh vector $\mathcal{P}$, returns all the abelian runs of period $\mathcal{P}$ in w in time $O(n \times |\mathcal{P}|)$ and space $O(\sigma + |\mathcal{P}|)$. The algorithm works in an online manner. To the best of our knowledge, this is the first algorithm solving the problem of searching for all the abelian runs having a given period.

We believe that further combinatorial results on the structure of the abelian runs in a word could lead to new algorithms.

One of the reviewers of this submission pointed out that our algorithm can be modified in order to achieve time complexity $O(n)$. Due to the limited time we had for preparing the final version of this paper, we did not include such improvement here. We will provide the details in a forthcoming full version of the paper. By the way, we warmly thank the reviewer for his comments.

References

1. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: A new characterization of maximal repetitions by Lyndon trees. CoRR abs/1406.0263 (2014), http://arxiv.org/abs/1406.0263
2. Constantinescu, S., Ilie, L.: Fine and Wilf's theorem for abelian periods. Bulletin of the European Association for Theoretical Computer Science 89, 167–170 (2006)
3. Cummings, L.J., Smyth, W.F.: Weak repetitions in strings. Journal of Combinatorial Mathematics and Combinatorial Computing 24, 33–48 (1997)
4. Fici, G., Lecroq, T., Lefebvre, A., Prieur-Gaston, É.: Algorithms for computing abelian periods of words. Discrete Applied Mathematics 163, 287–297 (2014)
5. Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Proceedings of the 1999 Symposium on Foundations of Computer Science (FOCS'99), New York (USA). pp. 596–604. IEEE Computer Society, New-York (October 17-19 1999)
6. Matsuda, S., Inenaga, S., Bannai, H., Takeda, M.: Computing abelian covers and abelian runs. In: Prague Stringology Conference 2014. p. 43 (2014)
7. Smyth, W.F.: Computing regularities in strings: a survey. European J. Combinatorics 34(1), 3–14 (2013)