Computing Abelian Periods in Words

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

[1] I3S, CNRS & Université de Nice-Sophia Antipolis, France
Gabriele.Fici@unice.fr
[2] Université de Rouen, LITIS EA4108, 76821 Mont-Saint-Aignan Cedex, France
{Thierry.Lecroq,Arnaud.Lefebvre,Elise.Prieur}@univ-rouen.fr

Abstract. In the last couple of years many works have been devoted to Abelian complexity of words. Recently, Constantinescu and Ilie (Bulletin EATCS 89, 167–170, 2006) introduced the notion of *Abelian period*. We show that a word w of length n over an alphabet of size σ can have $\Theta(n^2)$ distinct Abelian periods. However, to the best of our knowledge, no efficient algorithm is known for computing these periods. The Brute-Force algorithm computes all the Abelian periods either in time $O(n^3 \times \sigma)$ using $O(\sigma)$ space or in time $O(n^2 \times \sigma)$ using $O(n \times \sigma)$ space. We present an off-line algorithm running in time $O(n^2 \times \sigma)$ using $O(n+\sigma)$ space, thus improving the space complexity. This algorithm is based on a *select* function. We then present on-line algorithms that also enable to compute all the Abelian periods of all the prefixes of w. Experimental results show that the new off-line algorithm is faster than the Brute-Force one. Moreover, in most cases, one on-line algorithm, though having a worst case time complexity, is also faster than the Brute-Force one.

1 Introduction

An integer $p > 0$ is a (classical) period of a word w of length n if $w[i] = w[i + p]$ for any $1 \leqslant i \leqslant n-p$. Classical periods have been extensively studied in combinatorics on words [13] due to their direct applications in data compression and pattern matching.

The Parikh vector of a word w enumerates the cardinality of each letter of the alphabet in w. For example, given the alphabet $\Sigma = \{a, b, c\}$, the Parikh vector of the word $w = \texttt{aaba}$ is $(3, 1, 0)$. The reader can refer to [6] for a list of applications of Parikh vectors.

An integer p is an Abelian period of a word w if w can be written as $u_0 u_1 \cdots u_{k-1} u_k$ where all the u_i's are of length p and have the same Parikh vector $\mathcal{P}$ for $0 < i < k$ and the Parikh vectors of u_0 and u_k are contained in $\mathcal{P}$ [9]. This definition matches the one of *weak repetition* (also called *Abelian power*) when u_0 and u_k are the empty word and $k > 2$ [10].

In the last couple of years many works have been devoted to Abelian complexity [1,2,4,5,7,11,12,17]. Efficient algorithms for Abelian pattern matching have been designed [6,8,14,15]. However, apart of the greedy off-line algorithm given in [10], neither efficient nor on-line algorithms are known for computing all the Abelian periods of a given word.

In this article we present several efficient off-line and on-line algorithms for computing all the Abelian periods of a given word. In Section 2 we give some basic definitions and notation. Section 3 presents off-line algorithms while Section 4 presents on-line algorithms. In Section 5 we give some experimental results on execution times. Finally, Section 6 contains conclusions and perspectives.

2 Definitions and notation

Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be a finite ordered alphabet of cardinality σ and Σ^* the set of words on alphabet Σ. We set $ind(a_i) = i$ for $1 \leqslant i \leqslant \sigma$. We denote by $|w|$ the length of w. We write $w[i]$ the i-th symbol of w and $w[i..j]$ the factor of w from the i-th symbol to the j-th symbol, with $1 \leqslant i \leqslant j \leqslant |w|$. We denote by $|w|_a$ the number of occurrences of the symbol $a \in \Sigma$ in the word w.

The *Parikh vector* of a word 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 of the other.

We denote by $\mathcal{P}_w(i, m)$ the Parikh vector of the factor of length m beginning at position i in the word w.

Given the Parikh vector $\mathcal{P}_w$ of a word w, we denote by $\mathcal{P}_w[i]$ its i-th component and by $|\mathcal{P}_w|$ 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 ([9]). *A word w has an Abelian period (h, p) if $w = u_0 u_1 \cdots u_{k-1} u_k$ such that:*

- $\mathcal{P}_{u_0} \subset \mathcal{P}_{u_1} = \ldots = \mathcal{P}_{u_{k-1}} \supset \mathcal{P}_{u_k}$,
- $|\mathcal{P}_{u_0}| = h$, $|\mathcal{P}_{u_1}| = p$.

We call u_0 and u_k resp. the *head* and the *tail* of the Abelian period. Notice that the length $t = |u_k|$ of the tail is uniquely determined by h, p and $|w|$, namely $t = (|w| - h) \bmod p$.

The following lemma gives a bound on the maximum number of Abelian periods of a word.

Lemma 2. *The maximum number of Abelian periods for a word of length n over the alphabet Σ is $\Theta(n^2)$.*

Proof. The word $(a_1 a_2 \cdots a_\sigma)^{n/\sigma}$ has Abelian period (h, p) for any $p \equiv 0 \bmod \sigma$ and $h < p$. □

A natural order can be defined on the Abelian periods.

Definition 3. *Two distinct Abelian periods (h, p) and (h', p') of a word w are ordered as follows: $(h, p) < (h', p')$ if $p < p'$ or $(p = p'$ and $h < h')$.*

We are interested in computing all the Abelian periods of a word. The algorithms we present in this paper can be easily adapted to give only the smallest Abelian period.

3 Off-line algorithms

3.1 Brute-Force algorithm

In Figure 1 we present a Brute-Force algorithm which computes all the Abelian periods of an input word w of length n. For each possible head of length h from 1

```
AbelianPeriod-BruteForce(w, n)
 1  for h ← 0 to ⌊(n − 1)/2⌋ do
 2    p ← h + 1
 3    while h + p ≤ n do
 4      if (h, p) is an Abelian period of w then
 5        Output(h, p)
 6      p ← p + 1
```

Figure 1. Brute-Force algorithm for computing all the Abelian periods of a word w of length n.

to $\lfloor (n-1)/2 \rfloor$ the algorithm tests all the possible values of p such that $p > h$ and $h+p \leqslant n$. This is a reformulation of the algorithm given in [10]. The algorithm easily adapts to give only the smallest Abelian period or the weak repetitions.

Example 4. For $w = \texttt{abaababa}$ the algorithm outputs $(1,2)$, $(0,3)$, $(2,3)$, $(1,4)$, $(2,4)$, $(3,4)$, $(0,5)$, $(1,5)$, $(2,5)$, $(3,5)$, $(0,6)$, $(1,6)$, $(2,6)$, $(0,7)$, $(1,7)$ and $(0,8)$. Among these periods $(1,2)$ is the smallest.

Theorem 5. *The algorithm* AbelianPeriod-BruteForce *computes all the Abelian periods of a given word of length n in time $O(n^3 \times \sigma)$ with an $O(\sigma)$ space or in time $O(n^2 \times \sigma)$ with a space in $O(n \times \sigma)$.*

Proof. The correctness of the algorithm comes directly from Definition 1. Each test in line 4 consists in comparing n/p Parikh vectors. Comparing two Parikh vectors can be done in $\Theta(\sigma)$ time. The test in line 4 is performed $\sum_{h=0}^{\lfloor (n-1)/2 \rfloor} \sum_{p=h+1}^{n-h} n/p = O(\sum_{h=1}^{n} \sum_{p=h}^{n} n/p) = O(n^2)$ times. With no preprocessing, this gives an overall time of $O(n^3 \times \sigma)$. If the Parikh vectors of all the prefixes of the word have been already computed, this can be done by computing the difference between two Parikh vectors (see [3]). This requires space and time in $O(n \times \sigma)$ and gives an overall time of $O(n^2 \times \sigma)$. □

3.2 Select-based algorithm

Let us introduce the *select* function [16] defined as follows.

Definition 6. *Let w be a word of length n over alphabet Σ, then $\forall\, a \in \Sigma$:*

- $select_a(w, 0) = 0$*;*
- $\forall\, 1 \leqslant i \leqslant |w|_a$*, $select_a(w, i) = j$ iff j is the position of the i-th occurrence of letter a in w;*
- $\forall\, i > |w|_a$*, $select_a(w, i)$ is undefined.*

In order to compute the *select* function, we consider an array S_w of n elements that stores the increasing ordered positions of a_1, then the increasing ordered positions of a_2 and so on up to the increasing ordered positions of a_σ. In addition to S_w, we also consider an array C_w of $\sigma + 1$ elements such that $C_w[i] = \sharp\{w[k] = a_j \mid j < i \text{ and } 1 \leqslant k \leqslant n\} + 1$ for $1 \leqslant i \leqslant \sigma$ and $C_w[\sigma + 1] = n + 1$. Then, for $i > 0$,

$$select_a(w, i) = \begin{cases} S_w[C_w[ind(a)] + i - 1], & \text{if } i \leqslant C_w[ind(a) + 1] - C_w[ind(a)] \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$C_w[i] - 1$ is the number of letters in w strictly smaller than a_i. Array C_w serves as an index to access S_w.

Example 7. For $w = \texttt{abaababa}$ the *select* function uses the following three arrays:

	1	2	3	4	5	6	7	8
w	a	b	a	a	b	a	b	a

	a	b
ind	1	2

	1	2	3
C_w	1	6	9

	1	2	3	4	5	6	7	8
S_w	1	3	4	6	8	2	5	7

Then $select_{\texttt{b}}(w, 2) = S_w[C_w[ind(\texttt{b})] + 2 - 1] = 5$, which means that the second `b` in w appears in position 5.

The ComputeSelect function (see Figure 2) computes the two arrays C_w and S_w used by the *select* function. This can be done in $O(n + \sigma)$ time and space. Once these two arrays have been computed, each call to the *select* function is answered in constant time.

```
ComputeSelect(w, n)
 1  C_w[1] ← 1
 2  for i ← 2 to σ + 1 do
 3     C_w[i] ← C_w[i − 1] + P_w[i − 1]
 4  for i ← 1 to σ do
 5     P[i] ← 0
 6  for i ← 1 to n do
 7     S_w[C_w[ind(w[i])] + P[ind(w[i])] ← i
 8     P[ind(w[i])] ← P[ind(w[i])] + 1
 9  return (C_w, S_w)
```

Figure 2. Algorithm computing C_w and S_w arrays.

The Brute-Force algorithm tests all possible pairs (h, p) but it is clear that, given h, some pairs cannot be Abelian periods. For example, let $w = \texttt{abaaaaabaa}$ and $h = 2$. Since $\mathcal{P}_w(1, h)$ has to be included in $\mathcal{P}_w(h + 1, p)$, the pairs $(2, 3)$, $(2, 4)$ and $(2, 5)$ cannot be Abelian periods of w: the minimal p value such that $(2, p)$ can be an Abelian period is in fact 6, in order to include the second `b` of w. This remark leads us to give the following definitions and propositions.

Definition 8. *Let w be a word of length n on alphabet Σ. Then $\forall\, 0 \leqslant h \leqslant \lfloor (n-1)/2 \rfloor$, $\mathcal{M}_w[h]$ is defined by*

$$\mathcal{M}_w[h] = \begin{cases} \min\{p \mid \mathcal{P}_w(1, h) \subset \mathcal{P}_w(h + 1, p)\} & \text{if } \forall\, a \in \Sigma,\ 2 \times |w[1..h]|_a \leqslant |w|_a \\ -1 & \text{otherwise.} \end{cases}$$

In other words, if $\forall\, a \in \Sigma$, $select_a(w, 2 \times |w[1..h]|_a)$ is defined then

$$\mathcal{M}_w[h] = \max\{h + 1, \max\{select_a(w, 2 \times |w[1..h]|_a) \mid a \in \Sigma\} - h\},$$

otherwise $\mathcal{M}_w[h] = -1$.

Proposition 9. *Let w be a word of length n on alphabet Σ and $0 \leqslant h \leqslant \lfloor (n-1)/2 \rfloor$. If $\mathcal{M}_w[h] = -1$, then $\mathcal{M}_w[h'] = -1\ \forall\, h' \geqslant h$ and h' cannot be a head of an Abelian period of w.*

Proof. If $\mathcal{M}_w[h] = -1$, then by definition $\exists\, a \in \Sigma$ such that $2 \times |w[1..h]|_a > |w|_a$. Then, we cannot find a value p such that $|w[1..h]|_a \leqslant |w[(h + 1)..(h + p)]|_a$. It is clear that this is also true for all value $h' > h$. □

```
ComputeM(w, n, C_w, S_w)
 1  M_w[0] ← 0
 2  for a ∈ Σ do
 3     H[a] ← 0
 4  for h ← 1 to ⌊(n−1)/2⌋ do
 5     H[w[h]] ← H[w[h]] + 1
 6     s ← select_{w[h]}(w, 2 × H[w[h]])
 7     if s is defined then
 8        M_w[h] ← max{M_w[h − 1] − 1, s − h}
 9     else M_w[h] ← −1
10  for h ← 1 to ⌊(n−1)/2⌋ do
11     if M_w[h] = h then
12        M_w[h] ← h + 1
13  return M_w
```

Figure 3. Algorithm computing the $\mathcal{M}_w$ array.

The array $\mathcal{M}_w$ can be computed in time and space $O(n+\sigma)$, processing positions of w from left to right (see Figure 3).

Consider now the following definition.

Definition 10. *Let w be a word of length n on alphabet Σ. Then $\forall\, 0 \leqslant h \leqslant \lfloor (n-1)/2 \rfloor$, $\mathcal{G}_w[h]$ is defined by*

$$\mathcal{G}_w[h] = \max\{\ \mathit{select}_a(w, i+1) - \mathit{select}_a(w, i) \mid a \in \Sigma, \\ h < \mathit{select}_a(w, i) < \mathit{select}_a(w, i+1) \leqslant n\}.$$

Actually, $\mathcal{G}_w[h]$ is the maximal value $j - i$ such that $h < i < j$ and $w[i] = w[j]$.

The array $\mathcal{G}_w$ can be computed in time and space $O(n+\sigma)$, processing positions of w from right to left (see Figure 4).

```
ComputeG(w, n)
 1  G_w[n] ← 0
 2  for a ∈ Σ do
 3     T[a] ← 0
 4  for h ← n to 1 do
 5     if T[w[h]] = 0 then
 6        T[w[h]] ← h
 7        G_w[h − 1] ← G_w[h]
 8     else d ← T[w[h]] − h
 9        T[w[h]] ← h
10        G_w[h − 1] ← max{G_w[h], d}
11  return G_w
```

Figure 4. Algorithm computing the $\mathcal{G}_w$ array.

Proposition 11. *Let w be a word of length n on alphabet Σ. Let $0 \leqslant h \leqslant \lfloor (n-1)/2 \rfloor$. If $h < p < \max\{\mathcal{M}_w[h], \lfloor (\mathcal{G}_w[h]+1)/2 \rfloor\}$ then (h, p) is not an Abelian period of w.*

Proof. From the definition of $\mathcal{M}_w[h]$, it directly follows that if $p < \mathcal{M}_w[h]$, then (h, p) cannot be an Abelian period of w.

Given h, let $a \in \Sigma$ be such that there exists $1 \leqslant i < n$ and $select_a(w, i+1) - select_a(w, i) = \mathcal{G}_w[h]$. Let $j = select_a(w, i)$ and $j' = select_a(w, i+1)$. If $p < \lfloor(\mathcal{G}_w[h] + 1)/2\rfloor$, $k = \min\{k' \mid h + k'p \geqslant j\}$ then $h + (k+1)p < j'$ and $|w[k + kp + 1\,..\,h + (k+1)p]|_a = 0$. Thus (h, p) cannot be an Abelian period of w (see Figure 5). □

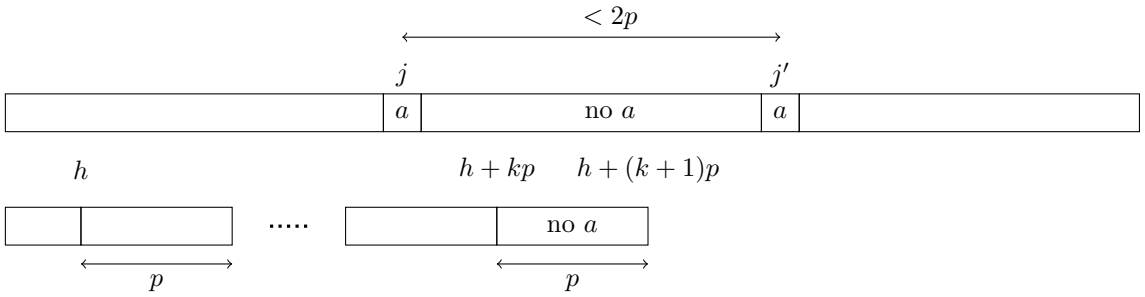

Figure 5. If the distance between two consecutive a's in w is greater than $2p$ then (h, p) cannot be an Abelian period of w for any $h < p$.

Arrays $\mathcal{M}_w$ and $\mathcal{G}_w$ give us, for every head length h, a minimal value for a possible p such that (h, p) can be an Abelian period of w. This allows us to skip a number of values for p that cannot give an Abelian period.

The following lemma shows how to check if (h, p) is indeed an Abelian period of w (except for the tail).

Lemma 12. *Let w be a word of length n on alphabet Σ. Let $\mathcal{H} = \mathcal{P}_w(1, h)$ and $\mathcal{P} = \mathcal{P}_w(h+1, p)$. Let $i = h + kp$ such that $0 < k$, $p \leqslant n - i$ and (h, p) is an Abelian period of $w[1\,..\,i]$ (with an empty tail). Then the following two points are equivalent:*

1. *(h, p) is an Abelian period of $w[1\,..\,i + p]$.*
2. *for all $a \in \Sigma$*

$$select_a(w, \mathcal{H}[ind(a)] + \left(1 + \left\lfloor \frac{i}{p} \right\rfloor\right) \times \mathcal{P}[ind(a)]) \leqslant i + p.$$

Proof. Since (h, p) is an Abelian period of $w[1\,..\,i]$ with $i = h + kp$ for some $k > 0$ then $|w[1\,..\,i]|_a = \mathcal{H}[ind(a)] + k \times \mathcal{P}[ind(a)]$ for each letter $a \in \Sigma$. Notice that since $h < p$ then $k = \lfloor i/p \rfloor$.
$(1 \Rightarrow 2)$. The fact that (h, p) is an Abelian period of $w[1\,..\,i + p]$ implies that, for all $a \in \Sigma$, $|w[1\,..\,i + p]|_a = \mathcal{H}[ind(a)] + (k+1) \times \mathcal{P}[ind(a)]$. Thus, by definition of *select*, $select_a(w, \mathcal{H}[ind(a)] + (1 + \lfloor i/p \rfloor) \times \mathcal{P}[ind(a)]) \leqslant i + p$.
$(2 \Rightarrow 1)$. The fact that $select_a(w, \mathcal{H}[ind(a)] + (1 + \lfloor i/p \rfloor) \times \mathcal{P}[ind(a)]) \leqslant i + p$ implies that $|w[1\,..\,i + p]|_a = \mathcal{H}[ind(a)] + (k+1) \times \mathcal{P}[ind(a)]$. We know that $|w[1\,..\,i]|_a = \mathcal{H}[ind(a)] + k \times \mathcal{P}[ind(a)]$. By difference, $|w[i+1\,..\,i+p]|_a = \mathcal{P}[ind(a)]$. Since it is true for all $a \in \Sigma$, $\mathcal{P}_w(i+1, p) = \mathcal{P}$ and then (h, p) is an Abelian period of $w[1\,..\,i+p]$. □

Figure 6 presents the algorithm AbelianPeriod-Shift based on the previous lemma.

Proposition 13. *Algorithm* AbelianPeriod-Shift(h, p, w, n, C_w, S_w) *returns* TRUE *iff (h, p) is an Abelian period of the prefix of length $n - ((n-h) \bmod p)$ of w in time $O(\frac{n}{p} \times \sigma)$ and space $O(\sigma)$.*

Proof. The correctness comes directly from Lemma 12. The **while** loop in line 3 is executed n/p times and the **for** loop in line 4 is executed σ times, thus the time complexity is $O(\frac{n}{p} \times \sigma)$. This algorithm only requires the storage of the two Parikh vectors $\mathcal{P}_w(1,h)$ and $\mathcal{P}_w(h+1,p)$. These vectors can be stored in space $O(\sigma)$ under the standard assumption that $\log n$ fits in a computer word. □

```
AbelianPeriod-Shift(h, p, w, n, C_w, S_w)
 1  (H, P) ← (P_w(1, h), P_w(h + 1, p))
 2  i ← h + p
 3  while i + p ⩽ n do
 4     for a ∈ Σ do
 5        s ← select_a(w, H[ind(a)] + (1 + ⌊i/p⌋) × P[ind(a)])
 6        if s is undefined or s > i + p then
 7           return FALSE
 8     i ← i + p
 9  return TRUE
```

Figure 6. Algorithm checking whether (h,p) is an Abelian period of the prefix of length $n - ((n - h) \bmod p)$ of w.

```
AbelianPeriod-Select(w, n)
 1  (C_w, S_w) ← ComputeSelect(w, n)
 2  M_w ← ComputeM(w, n, C_w, S_w)
 3  G_w ← ComputeG(w, n)
 4  h ← 0
 5  while h ⩽ ⌊(n − 1)/2⌋ and M_w[h] ≠ −1 do
 6     p ← max(M_w[h], ⌊(G_w[h] + 1)/2⌋)
 7     while h + p ⩽ n do
 8        if AbelianPeriod-Shift(h, p, w, n, C_w, S_w) then
 9           t ← (n − h) mod p
10           if P_w(n − t + 1, t) ⊂ P_w(h + 1, p) then
11              Output(h, p)
12        p ← p + 1
13     h ← h + 1
```

Figure 7. Algorithm computing all the Abelian periods of word w of length n, based on the *select* function.

Using Proposition 11 and Proposition 13, algorithm AbelianPeriod-Select, given in Figure 7, computes all the Abelian periods of a word w of length n.

Theorem 14. *Algorithm* AbelianPeriod-Select *computes all the Abelian periods of word w of length n in time $O(n^2 \times \sigma)$ and space $O(n + \sigma)$.*

Proof. The correctness of the algorithm comes from Proposition 11 and Proposition 13.

The *select* function and the arrays $\mathcal{M}_w$ and $\mathcal{G}_w$ can be computed in $O(n + \sigma)$ time and space. According to Proposition 11, the value of p computed in line 6 is the minimal value such that (h,p) can be an Abelian period of w. The AbelianPeriod-Shift function, called in line 8, simply verifies that (h,p) is an Abelian period of w in

time $O(\frac{n}{p} \times \sigma)$. The test in line 10 is done in $O(p)$ time. The complexity of the **while** loop in line 7 is $O(\sum_{p=h+1}^{n} \frac{n}{p}) = O(n)$. Consequently, algorithm AbelianPeriodSelect computes all the Abelian periods of w in time $O(n^2 \times \sigma)$ and space $O(n+\sigma)$ (output periods are not stored). □

4 On-line algorithms

We now propose two on-line algorithms to compute all the Abelian periods of a word w using dynamic programming. When processing $w[i]$, in the first algorithm, using a two dimensional array, we inspect all the possible values (h, p); in the second one, using heaps, we inspect the Abelian periods of $w[1..i-1]$ by groups built depending upon the tail length of the periods.

The following proposition states that if (h, p) is not an Abelian period of a word w, with $h + p \leqslant n = |w|$, then it cannot be an Abelian period of any word having w as prefix.

Proposition 15. *Let w be a word of length n and let h, p such that $h + p \leqslant n$. If (h, p) is not an Abelian period of w, then (h, p) is not an Abelian period of wa for any symbol $a \in \Sigma$.*

Proof. If (h, p) is not an Abelian period of w, at least one of the following three cases holds:

1. $\mathcal{P}_w(1, h) \not\subset \mathcal{P}_w(h + 1, p)$;
2. there exist two distinct indices $h \leqslant i, i' \leqslant |w| - p + 1$ such that $i = kp + h + 1$ and $i' = k'p + h + 1$ with k and k' two integers and $\mathcal{P}_w(i, p) \neq \mathcal{P}_w(i', p)$;
3. $t = (|w| - h) \bmod p$ and $\mathcal{P}_w(|w| - t + 1, t) \not\subset \mathcal{P}_w(|w| - p - t + 1, p)$.

If case 1 holds then $\mathcal{P}_{wa}(1, h) \not\subset \mathcal{P}_{wa}(h + 1, p)$ and (h, p) is not an Abelian period of wa. If case 2 holds then $\mathcal{P}_{wa}(i, p) \neq \mathcal{P}_{wa}(i', p)$ and (h, p) is not an Abelian period of wa. If case 3 holds then $\mathcal{P}_{wa}(|w| - t + 1, t + 1) \not\subseteq \mathcal{P}_{wa}(|w| - p - t + 1, p)$ and (h, p) is not an Abelian period of wa. □

4.1 Two dimensional array

We noxw propose an algorithm that uses a two dimensional array and Proposition 15 to compute all the Abelian periods of an input word w in an on-line manner. It processes the positions of w in increasing order. When processing position i, $T[h, p] = j$ iff $w[1..j]$ is the longest prefix of $w[1..i]$ having Abelian period (h, p). Thus if $j = i - 1$ the algorithm checks whether $w[1..i]$ has Abelian period (h, p) and updates $T[h, p]$ accordingly.

When $T[h, p] = i$ it means that $w[1..i]$ is the longest prefix of w that has (h, p) as an Abelian period. Thus when $T[h, p] = n$ it means that (h, p) is an Abelian period of w.

Example 16. For $w =$ `abaababa` the algorithm computes the following array T:

$h\backslash p$	1	2	3	4	5	6	7	8
0	1	3	8	6	8	8	8	8
1		8	6	8	8	8	8	
2			8	8	8	8		
3				8	8			

Cells $T[h,p] = |w|$ correspond to pairs (h,p) output by algorithm AbelianPeriodBruteForce of example 4. Empty cells on the left part of the array correspond to cases where $h \geq p$ and empty cells on the right part correspond to cases where $h + p > |w|$.

In order to improve the space complexity of this algorithm, the Abelian periods can be stored in a list instead of an array: When processing position i one only stores pairs (h,p) such that $w[1..i]$ has Abelian period (h,p); these pairs correspond to all the cells of array T, computed by the previous algorithm, such that $T[h,p] = i$. At the end of this process, when $i = n$, this list contains all the Abelian periods of w, and only them.

The above algorithms computes all the Abelian periods of a word of length n on an alphabet of size σ in $O(n^3 \times \sigma)$ time using $O(n^2)$ space.

4.2 Heaps

The following proposition shows that the set of Abelian periods of a prefix of a word can be partitioned into subsets depending of the length of the tail. In some cases all the periods of a subset can be processed at once by inspecting only the smallest period of the subset.

Proposition 17. *Let w have s Abelian periods $(h_1,p_1) < (h_2,p_2) < \cdots < (h_s,p_s)$ such that $(|w| - h_i) \bmod p_i = t > 0$ for $1 \leqslant i \leqslant s$. If (h_1,p_1) is an Abelian period of wa for any symbol $a \in \Sigma$ then $(h_2,p_2),\ldots,(h_s,p_s)$ are also Abelian periods of wa.*

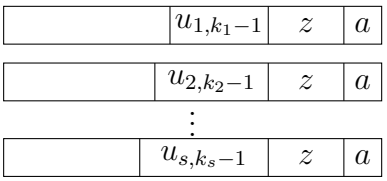

Figure 8. $w = u_{i,0}u_{i,1}\cdots u_{i,k_i-1}u_{i,k_i}$, $u_{i,k_i} = z$ for $1 \leqslant i \leqslant s$. If $\mathcal{P}_{za} \subseteq \mathcal{P}_{u_{1,k_1-1}}$ then $\mathcal{P}_{za} \subseteq \mathcal{P}_{u_{i,k_i-1}}$ for every $2 \leqslant i \leqslant s$.

Proof. Since $(h_1,p_1) < (h_2,p_2) < \cdots < (h_s,p_s)$ are Abelian periods of w, $w = u_{i,0}u_{i,1}\cdots u_{i,k_i-1}u_{i,k_i}$ with $|u_{i,0}| = h_i$, $|u_{i,j}| = p_i$ and $|u_{i,k_i}| = t$ for $1 \leqslant i \leqslant s$ and $1 \leqslant j \leqslant k_i$. If (h_1,p_1) is an Abelian period of wa, $\mathcal{P}_{u_{1,k_1}a} \subseteq \mathcal{P}_{u_{1,k_1-1}}$. Since $|u_{1,k_1}| = |u_{i,k_i}|$ and $|u_{1,k_1-1}| \leqslant |u_{i,k_i-1}|$ we have that $\mathcal{P}_{u_{i,k_i}a} \subseteq \mathcal{P}_{u_{i,k_i-1}}$ for $2 \leqslant i \leqslant s$. Thus $(h_2,p_2),\ldots,(h_s,p_s)$ are Abelian periods of wa (see Figure 8). □

The algorithm given in Figure 9 uses Proposition 17 for computing all the Abelian periods by gathering all the ongoing periods (h,p) with the same tail length together in a heap where the element at the root of the heap is the smallest period.

When processing $w[i]$, the algorithm processes every heap H for the different tail lengths:

- if the period (h, p) at the root of H is a period of $w[1..i]$ then by Proposition 17 all the elements of H are Abelian periods of $w[1..i]$. If the tail length becomes equal to p then (h, p) is removed from the current heap and is moved into a new heap corresponding to the empty tail.
- if the period (h, p) at the root of H is not a period of $w[1..i]$ then it is removed from H and the same process is applied until a pair (h', p') is an Abelian period of $w[1..i]$ or the heap becomes empty.
  In the former case, by Proposition 17, all the remaining elements of H are Abelian periods of $w[1..i]$. This is realized by function ExtractUntilOK in line 8.

Then all the degenerate cases (h, p) such that $h < p$ and $h + p = i$ have to be inserted in the heap corresponding to the empty tail (lines 12 to 15).

The function Root(H) returns the smallest element of the heap H, the function Insert(H, e) inserts element e in the heap H, while the function Remove(H) removes the smallest element of the heap H.

```
AbelianPeriod-heap(w, n)
 1  L ← list with one heap containing (0, 1)
 2  for i ← 2 to n do
 3     NewHeap ← ∅
 4     for all H ∈ L do
 5        (h, p) ← Root(H)
 6        t ← p − ((i − h) mod p)
 7        if P_w(i − t + 1, t) ⊈ P_w(i − t − p + 1, p) then
 8           ExtractUntilOK(H)
 9        else if t = p then
10              Remove(H)
11              Insert(NewHeap, (h, p))
12     h ← 0
13     while h < ⌊(i + 1)/2⌋ and P_w(1, h) ⊂ P_w(h + 1, i − h) do
14        Insert(NewHeap, (h, i − h))
15        h ← h + 1
16     L ← L ∪ NewHeap
17  return L
```

Figure 9. On-line algorithm for computing all the Abelian periods of a word w of length n using heaps.

Theorem 18. *The algorithm* AbelianPeriod-heap *computes all the Abelian periods of a given word of length n in time $O(n^2 \times (n \log n) \times \sigma)$ and space $O(n^2)$.*

Proof. The correctness of the algorithm comes from Proposition 17. The maximum number of heaps is $n/2$ and the total number of elements of all the heaps is $O(n^2)$ (Lemma 2). The space complexity for the list L is $O(n^2)$. The time complexity of the algorithm is due to the two **for** loops of lines 2 and 4 and the different calls to ExtractUntilOK in line 8 and Insert and Remove. The maximum number of heaps is $n/2$, and the maximum number of elements in a single heap is n. Thus, the total complexity for the calls to ExtractUntilOK, Insert and Remove in a single run of the **for** loop of line 4 is $O(n \log n)$. □

5 Experimental results

To compare practical performances of the different algorithms, they have been implemented in C in a homogeneous way and run on test sets of random words (1000 words each) of different lengths (from 10 to 2000) on different alphabet sizes (2, 3, 4, 8 and 16).

Tests were performed on a computer running Mac OS X with a 2.2 GHz processor and 2 GB RAM.

Figure 10 presents average running times over 1000 random words on alphabet size 16 of the algorithms AbelianPeriod-BruteForce, AbelianPeriod-Select and AbelianPeriod-Heaps. Corresponding values are given Figure 11. The results show that, as expected, the off-line algorithm using *select* function is indeed faster than the other ones. Moreover, our tests show that, for long words, the on-line algorithm using heaps becomes faster than the Brute-Force one. One can notice that the difference of running times between the three algorithms increases as the word length grows. Results for other alphabet sizes, natural languages texts or genomic sequences are not shown since they are similar to these ones.

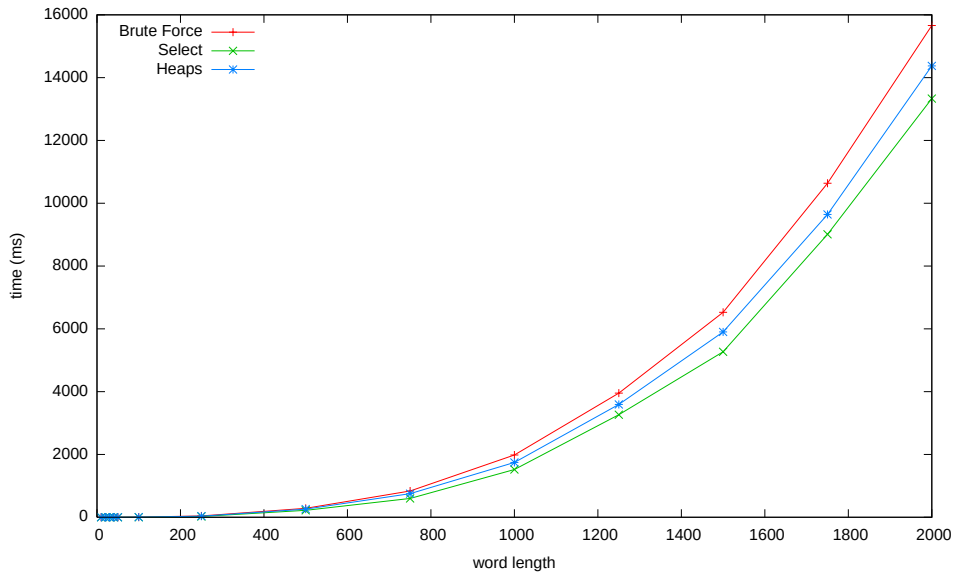

Figure 10. Average running times (in ms) over 1000 random words, of the Brute-Force, *select*-based and heaps-based algorithms on alphabet size 16.

algo|w|	10	20	30	40	50	100	250	500	750	1000	1250	1500	1750	2000
Brute-Force	0	0	0	0	0	4	43	286	836	1984	3952	6527	10636	15560
Select	0	0	0	0	0	2	25	221	599	1519	3267	5270	9009	13334
Heaps	0	0	0	0	0	3	32	260	752	1746	3592	5901	9643	14372

Figure 11. Values of average running times (in ms) of the Brute-Force, *select*-based and heaps-based algorithms on alphabet size 16.

6 Conclusion and perspectives

In this paper we presented different algorithms to compute all the Abelian periods of a word. This is the first attempt to give algorithms for computing all the Abelian periods of a word. In particular, we give a $O(n^2 \times \sigma)$ time off-line algorithm requiring $O(n + \sigma)$ space, thus reducing the space complexity compared to the Brute-Force algorithm. Moreover, in practice, this algorithm appears to be faster. It is even faster when one wants to compute Abelian periods (h, p) of a word w with at least two consecutive factors of length p having the same Parikh vector, *i.e.* $h + 2p \leqslant |w|$ (see Figure 12).

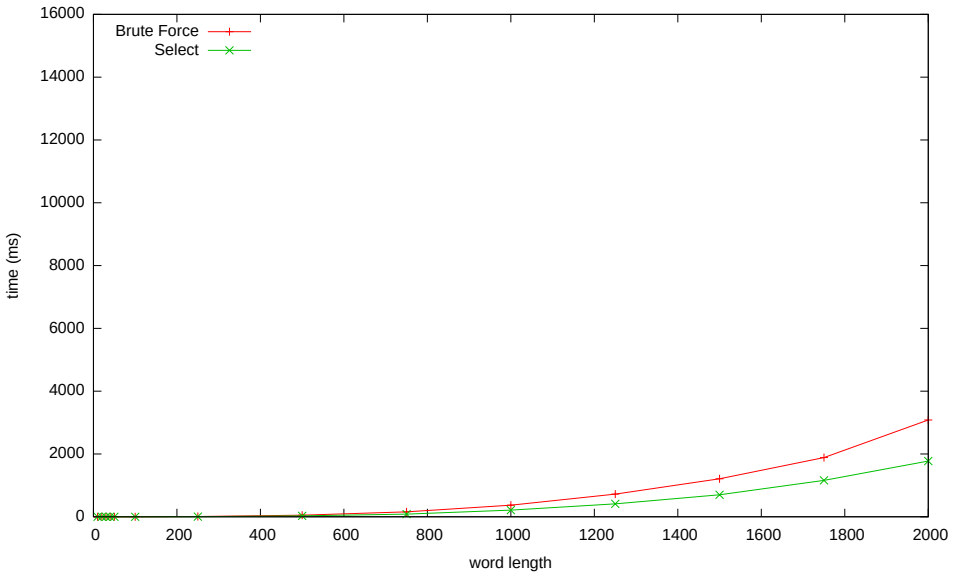

Figure 12. Average running times (in ms) over 1000 random words, of the Brute-Force and *select*-based algorithms on alphabet size 16, in the case where $h + 2p \leqslant |w|$. See the difference with Figure 10.

Cutting positions of an Abelian period (h, p) of a word w can be defined as follows:

$$Cut_w(h, p) = \{k = h + jp \mid 1 \leqslant k \leqslant |w| \text{ and } 0 \leqslant j\}.$$

An Abelian period (h, p) of w is non-deducible if there does not exist another Abelian period (h', p') of w such that $Cut_w(h, p) \subset Cut_w(h', p')$. In order to improve algorithm complexities, one way would consist in reporting only non-deducible Abelian periods.

It remains to obtain a bound on the minimal Abelian period given a word length and an alphabet size. Simple modifications of the presented algorithms would allow one to compute the minimal Abelian period of each factor of a word. This could have practical applications in areas such as bioinformatics and more precisely in the detection of DNA regions of homogeneous compositions.

References

1. S. Avgustinovich, A. Glen, B. Halldórsson, and S. Kitaev: *On shortest crucial words avoiding abelian powers.* Discrete Applied Mathematics, 158(6) 2010, pp. 605–607.

2. S. Avgustinovich, J. Karhumäki, and S. Puzynina: *On Abelian versions of Critical Factorization Theorem*, in Proceedings of the 13th Mons Theoretical Computer Science Days, 2010.

3. G. Benson: *Composition alignment*, in Algorithms in Bioinformatics, Third International Workshop, WABI 2003, Budapest, Hungary, September 15-20, 2003, Proceedings, G. Benson and R. Page, eds., vol. 2812 of Lecture Notes in Computer Science, Springer, 2003, pp. 447–461.

4. F. Blanchet-Sadri, J. I. Kim, R. Mercas, W. Severa, and S. Simmons: *Abelian square-free partial words*, in Proceedings of the 4th International Conference Language and Automata Theory and Applications, A.-H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer, 2010, pp. 94–105.

5. F. Blanchet-Sadri, A. Tebbe, and A. Veprauskas: *Fine and Wilf's theorem for abelian periods in partial words*, in Proceedings of the 13th Mons Theoretical Computer Science Days, 2010.

6. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *Algorithms for jumbled pattern matching in strings.* International Journal of Foundations of Computer Science, to appear.

7. J. Cassaigne, G. Richomme, K. Saari, and L. Zamboni: *Avoiding abelian powers in binary words with bounded abelian complexity.* International Journal of Foundations of Computer Science, 22(4) 2011.

8. F. Cicalese, G. Fici, and Zs. Lipták: *Searching for jumbled patterns in strings*, in Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, J. Holub and J. Zdárek, eds., Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009, pp. 105–117.

9. S. Constantinescu and L. Ilie: *Fine and Wilf's theorem for abelian periods.* Bulletin of the European Association for Theoretical Computer Science, 89 2006, pp. 167–170.

10. L. J. Cummings and W. F. Smyth: *Weak repetitions in strings.* Journal of Combinatorial Mathematics and Combinatorial Computing, 24 1997, pp. 33–48.

11. J. D. Currie and A. Aberkane: *A cyclic binary morphism avoiding abelian fourth powers.* Theoretical Computer Science, 410(1) 2009, pp. 44–52.

12. M. Domaratzki and N. Rampersad: *Abelian primitive words*, in Proceedings of the 15th Conference on Developments in Language Theory, G. Mauri and A. Leporati, eds., vol. 6795 of Lecture Notes in Computer Science, Springer, 2011.

13. M. Lothaire: *Algebraic Combinatorics on Words*, Cambridge University Press, 2002.

14. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings.* Information Processing Letters, 110(18-19) 2010, pp. 795–798.

15. T. M. Moosa and M. S. Rahman: *Sub-quadratic time and linear size data structures for permutation matching in binary strings.* Journal of Discrete Algorithms, to appear.

16. G. Navarro and V. Mäkinen: *Compressed full-text indexes.* ACM Computing Surveys, 39(1) 2007.

17. A. Samsonov and A. Shur: *On abelian repetition threshold*, in Proceedings of the 13th Mons Theoretical Computer Science Days, 2010.