Computing Abelian Periods in Words

Gabriele Fici¹, Thierry Lecroq², Arnaud Lefebvre², and ´Elise Prieur-Gaston²

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

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 \( \sigma \) 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 \( 1 \leq i \leq n - p \). 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 \( w = aaba \) is \((3, 1, 0)\). The reader can refer to [1] 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 \( u_0u_1 \cdots u_{k-1}u_k \) where all the \( u_i \)’s are of length \( p \) and have the same Parikh vector \( P \) for \( 0 < i < k \) and the Parikh vectors of \( u_0 \) and \( u_k \) are contained in \( P \) [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 [11, 2, 7, 4, 12, 1, 5, 17]. Efficient algorithms for Abelian pattern matching have been designed [8, 6, 14, 15]. However, apart from 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 4 we give some basic definitions and notation. Section 5 presents off-line algorithms while Section 6 presents on-line algorithms. In Section 7 we give some experimental results on execution times. Finally, Section 8 contains conclusions and perspectives.
2 Definitions and notation

Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be a finite ordered alphabet of cardinality $\sigma$ and $\Sigma^*$ the set of words on alphabet $\Sigma$. We set $\text{ind}(a_i) = i$ 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 $P_w$, counts the occurrences of each letter of $\Sigma$ in $w$, that is $P_w = (|w|_{a_1}, \ldots, |w|_{a_\sigma})$. Notice that two words have the same Parikh vector if and only if one word is a permutation of the other.

We denote by $P_w(i, m)$ the Parikh vector of the factor of length $m$ beginning at position $i$ in the word $w$.

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

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

**Definition 1 (\cite{9}).** A word $w$ has an Abelian period $(h, p)$ if $w = u_0u_1 \cdots u_{k-1}u_k$ such that:
- $P_{u_0} \subset P_{u_1} = \cdots = P_{u_{k-1}} \supset P_{u_k}$
- $|P_{u_0}| = h$, $|P_{u_1}| = p$.

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 $t = (|w| - h) \mod p$.

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 $\Sigma$ is $\Theta(n^2)$.

**Proof.** The word $(a_1a_2 \cdots a_\sigma)^{n/\sigma}$ has Abelian period $(h, p)$ for any $p \equiv 0 \mod \sigma$ and $h < p$. \hfill $\Box$

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 to $\lfloor (n - 1)/2 \rfloor$ the algorithm tests all the possible values of $p$ such that $p > h$ and $h + p \leq n$. This is a reformulation of the algorithm given in \cite{10}. The algorithm easily adapts to give only the smallest Abelian period or the weak repetitions.
ABELIANPERIOD-BruteForce($w, n$)
1 for $h ← 0$ to $\lfloor(n - 1)/2\rfloor$ do
2 \hspace{1em} $p ← h + 1$
3 \hspace{1em} while $h + p ≤ n$ do
4 \hspace{2em} if $(h, p)$ is an Abelian period of $w$ then
5 \hspace{3em} OUTPUT($h, p$)
6 \hspace{2em} $p ← p + 1$

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

Example 4. For $w = \text{abaababa}$ 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 3 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 $\sum_{h=0}^{(n-1)/2} \sum_{p=h+1}^{n} n/p = O(\sum_{h=1}^{n} \sum_{p=h}^{n} n/p) = O(n^2)$ 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 $O(n^2 \times \sigma)$. \hfill $\Box$

3.2 Select-based algorithm

Let us introduce the select function defined as follows.

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

- $\text{select}_a(w, 0) = 0$;
- $\forall 1 ≤ i ≤ |w|_a$, $\text{select}_a(w, i) = j$ iff $j$ is the position of the $i$-th occurrence of letter $a$ in $w$;
- $\forall i > |w|_a$, $\text{select}_a(w, i)$ 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 $a_\sigma$. In addition to $S_w$, we also consider an array $C_w$ of $\sigma + 1$ elements such that $C_w[i] = \sharp \{w[k] = a_j \mid j < i \text{ and } 1 ≤ k ≤ n\} + 1$ for $1 ≤ i ≤ \sigma$ and $C_w[\sigma + 1] = n + 1$. Then, for $i > 0$,

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

$C_w[i] - 1$ 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 \( w = \text{abaababa} \) the select function uses the following three arrays:

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{w} & \text{a} & \text{b} & \text{a} & \text{a} & \text{b} & \text{a} & \text{b} \\
\text{ind} & 1 & 2 & & & & & \\
\text{C}_w & 1 & 6 & 9 & & & & \\
\text{S}_w & 1 & 3 & 4 & 6 & 8 & 2 & 5 \ \\
\end{array}
\]

Then \( \text{select}_b(w, 2) = \text{S}_w[\text{ind}(b)] + 2 - 1 = 5 \), which means that the second \( b \) in \( w \) appears in position 5.

The \text{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.

\[\text{ComputeSelect}(w, n)\]
\[
1 \quad C_w[1] \leftarrow 1 \\
2 \quad \text{for } i \leftarrow 2 \text{ to } \sigma + 1 \text{ do} \\
3 \quad C_w[i] \leftarrow C_w[i - 1] + \mathcal{P}_w[i - 1] \\
4 \quad \text{for } i \leftarrow 1 \text{ to } \sigma \text{ do} \\
5 \quad P[i] \leftarrow 0 \\
6 \quad \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
7 \quad S_w[C_w[\text{ind}(w[i])] + P[\text{ind}(w[i])]] \leftarrow i \\
8 \quad P[\text{ind}(w[i])] \leftarrow P[\text{ind}(w[i])] + 1 \\
9 \quad \text{return } (C_w, S_w)
\]

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 \( w = \text{abaaaaabaa} \) 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 \( \Sigma \). Then \( \forall 0 \leq h \leq \lfloor (n-1)/2 \rfloor \), \( \mathcal{M}_w[h] \) is defined by

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

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

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

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

Proposition 9. Let \( w \) be a word of length \( n \) on alphabet \( \Sigma \) and \( 0 \leq h \leq \lfloor (n-1)/2 \rfloor \). If \( \mathcal{M}_w[h] = -1 \), then \( \mathcal{M}_w[h'] = -1 \ \forall h' \geq h \) 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 \( |w[1..h]|_a \leq |w[(h + 1)\ldots(h + p)]|_a \). It is clear that this is also true for all value \( h' > h \). \( \square \)
**Definition 10.** Let \( w \) be a word of length \( n \) on alphabet \( \Sigma \). Then \( \forall 0 \leq h \leq \lfloor (n-1)/2 \rfloor \), \( G_w[h] \) is defined by
\[
G_w[h] = \max \{ \text{select}_a(w, i + 1) - \text{select}_a(w, i) \mid a \in \Sigma, h < \text{select}_a(w, i) < \text{select}_a(w, i + 1) \leq n \}.
\]
Actually, \( G_w[h] \) is the maximal value \( j - i \) such that \( h < i < j \) and \( w[i] = w[j] \).

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

**Formula 4.** Algorithm computing the \( G_w \) array.

**Proposition 11.** Let \( w \) be a word of length \( n \) on alphabet \( \Sigma \). Let \( 0 \leq h \leq \lfloor (n-1)/2 \rfloor \). If \( h < p < \max\{M_w[h], \lfloor (G_w[h] + 1)/2 \rfloor \} \) then \((h, p)\) is not an Abelian period of \( w \).

**Proof.** From the definition of \( M_w[h] \), it directly follows that if \( p < M_w[h] \), then \((h, p)\) cannot be an Abelian period of \( w \).

Given \( h \), let \( a \in \Sigma \) be such that there exists \( 1 \leq i < n \) and \( \text{select}_a(w, i + 1) - \text{select}_a(w, i) = G_w[h] \). Let \( j = \text{select}_a(w, i) \) and \( j' = \text{select}_a(w, i + 1) \). If \( p < \lfloor (G_w[h] + 1)/2 \rfloor \), \( k = \min\{k' \mid h + kp \geq j\} \) 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).
Lemma 12. Let $w$ be a word of length $n$ on alphabet $\Sigma$. Let $H = \mathcal{P}_w(1, h)$ and $\mathcal{P} = \mathcal{P}_w(h + 1, p)$. Let $i = h + kp$ such that $0 < k, p \leq n - i$ 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 $a \in \Sigma$

$$select_a(w, H[\text{ind}(a)]) + \left(1 + \left\lceil \frac{i}{p} \right\rceil \right) \times \mathcal{P}[\text{ind}(a)] \leq i + p.$$ 

Proof. Since $(h, p)$ is an Abelian period of $w[1..i]$ with $i = h + kp$ for some $k > 0$ then $|w[1..i]|_a = H[\text{ind}(a)] + k \times \mathcal{P}[\text{ind}(a)]$ for each letter $a \in \Sigma$. Notice that since $h < p$ then $k = [i/p]$.

$(1 \Rightarrow 2)$. The fact that $(h, p)$ is an Abelian period of $w[1..i + p]$ implies that, for all $a \in \Sigma$, $|w[1..i + p]|_a = H[\text{ind}(a)] + (k + 1) \times \mathcal{P}[\text{ind}(a)]$. Thus, by definition of select, $select_a(w, H[\text{ind}(a)]) + (1 + \lceil i/p \rceil) \times \mathcal{P}[\text{ind}(a)] \leq i + p$.

$(2 \Rightarrow 1)$. The fact that $select_a(w, H[\text{ind}(a)]) + (1 + \lceil i/p \rceil) \times \mathcal{P}[\text{ind}(a)] \leq i + p$ implies that $|w[1..i + p]|_a = H[\text{ind}(a)] + (k + 1) \times \mathcal{P}[\text{ind}(a)]$. We know that $|w[1..i]|_a = H[\text{ind}(a)] + k \times \mathcal{P}[\text{ind}(a)]$. By difference, $|w[i + 1..i + p]|_a = \mathcal{P}[\text{ind}(a)]$. 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 5 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 $n - ((n - h) \mod p)$ of $w$ in time $O\left(\frac{n}{p} \times \sigma\right)$ 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 $\sigma$ times, thus the time complexity is $O\left(\frac{n}{p} \times \sigma\right)$. 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. $(H, P) \leftarrow (P_w(1, h), P_w(h+1, p))$
2. $i \leftarrow h + p$
3. while $i + p \leq n$
4. for $a \in \Sigma$
5. $s \leftarrow \text{select}_w(w, H[ind(a)] + (1 + \lfloor i/p \rfloor) \times P[ind(a)])$
6. if $s$ is undefined or $s > i + p$ then
7. return false
8. $i \leftarrow i + p$
9. return true

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

AbelianPeriod-Select($w, n$)
1. $(C_w, S_w) \leftarrow \text{ComputeSelect}(w, n)$
2. $M_w \leftarrow \text{ComputeM}(w, n, C_w, S_w)$
3. $G_w \leftarrow \text{ComputeG}(w, n)$
4. $h \leftarrow 0$
5. while $h \leq \lfloor (n - 1)/2 \rfloor$ and $M_w[h] \neq -1$
6. $p \leftarrow \max(M_w[h], \lfloor (G_w[h] + 1)/2 \rfloor)$
7. while $h + p \leq n$
8. if AbelianPeriod-Shift($h, p, w, n, C_w, S_w$) then
9. $t \leftarrow (n - h) \mod p$
10. if $P_w(n - t + 1, t) \subset P_w(h + 1, p)$ then
11. Output($h, p$)
12. $p \leftarrow p + 1$
13. $h \leftarrow h + 1$

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 6, 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 $M_w$ and $G_w$ 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 $O(n^2 \times \sigma)$. 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 AbelianPeriod-Select 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 \(h + p \leq n = |w|\), 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 \(h + p \leq n\). 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 \(h \leq i, i' \leq |w| − p + 1\) 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. \(t = (|w| − h) \mod p\) 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 \(\mathcal{P}_{wa}(|w| − t + 1, t + 1) \not\subset \mathcal{P}_{wa}(|w| − p − t + 1, p)\) and \((h, p)\) is not an Abelian period of \(wa\).

### 4.1 Two dimensional array

We now 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 \(T:\)

<table>
<thead>
<tr>
<th>(h\backslash p)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cells \(T[h, p] = |w|\) correspond to pairs \((h, p)\) output by algorithm AbelianPeriod-BruteForce 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 $\sigma$ 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 $(|w| - h_i) \mod p_i = t > 0$ for $1 \leq i < s$. 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](image)

**Proof.** Since $(h_1, p_1) < (h_2, p_2) < \cdots < (h_s, p_s)$ are Abelian periods of $w$, $w = u_0u_1 \cdots u_{i-1}u_{i}, u_{i,k} = z$ for $1 \leq i < s$. If $P_{za} \subseteq P_{u_1,k_i-1}$ then $P_{za} \subseteq P_{u_1,k_i-1}$ for every $2 \leq i \leq s$.

The algorithm given in Figure 8 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 EXTRACTUPTILOK 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 13).

The function $\text{ROOT}(H)$ returns the smallest element of the heap $H$, the function $\text{INSERT}(H, e)$ inserts element $e$ in the heap $H$, while the function $\text{REMOVE}(H)$ removes the smallest element of the heap $H$. 
\textbf{ABELIANPERIOD-heap}(w, n)
1 \hspace{1em} L \gets \text{list with one heap containing (0, 1)}
2 \hspace{1em} \text{for } i \gets 2 \text{ to } n \text{ do}
3 \hspace{2em} \text{NewHeap} \gets \emptyset
4 \hspace{2em} \text{for all } H \in L \text{ do}
5 \hspace{3em} (h, p) \gets \text{Root}(H)
6 \hspace{3em} t \gets p - ((i - h) \mod p)
7 \hspace{3em} \text{if } \mathcal{P}_w(i - t + 1, t) \nsubseteq \mathcal{P}_w(i - t - p + 1, p) \text{ then}
8 \hspace{4em} \text{ExtractUntilOK}(H)
9 \hspace{2em} \text{else if } t = p \text{ then}
10 \hspace{3em} \text{Remove}(H)
11 \hspace{3em} \text{Insert}(\text{NewHeap}, (h, p))
12 \hspace{3em} h \gets 0
13 \hspace{2em} \text{while } h < \lfloor (i + 1) / 2 \rfloor \text{ and } \mathcal{P}_w(1, h) \subseteq \mathcal{P}_w(h + 1, i - h) \text{ do}
14 \hspace{3em} \text{Insert}(\text{NewHeap}, (h, i - h))
15 \hspace{3em} h \gets h + 1
16 \hspace{1em} L \gets L \cup \text{NewHeap}
17 \hspace{1em} \text{return } L

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

\textbf{Theorem 18.} The algorithm \textbf{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 $O(n^2)$.

\textit{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 \texttt{for} loops of lines 2 and 3 and the different calls to \texttt{ExtractUntilOK} in line 8 and \texttt{Insert} and \texttt{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 \texttt{ExtractUntilOK}, \texttt{Insert} and \texttt{Remove} in a single run of the \texttt{for} loop of line 3 is $O(n \log n)$. \hfill \Box

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 \textbf{ABELIANPERIOD-BRUTEFORCE}, \textbf{ABELIANPERIOD-SELECT} and \textbf{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.

<table>
<thead>
<tr>
<th>algo|w</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>100</th>
<th>250</th>
<th>500</th>
<th>750</th>
<th>1000</th>
<th>1250</th>
<th>1500</th>
<th>1750</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute-Force</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>43</td>
<td>286</td>
<td>836</td>
<td>1984</td>
<td>3952</td>
<td>6527</td>
<td>10636</td>
<td>15560</td>
<td></td>
</tr>
<tr>
<td>Select</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>25</td>
<td>221</td>
<td>399</td>
<td>1319</td>
<td>3287</td>
<td>5270</td>
<td>9009</td>
<td>13334</td>
<td></td>
</tr>
<tr>
<td>Heaps</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>32</td>
<td>260</td>
<td>752</td>
<td>1746</td>
<td>3592</td>
<td>5901</td>
<td>9643</td>
<td>14372</td>
<td></td>
</tr>
</tbody>
</table>

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. $h + 2p \leq |w|$ (see Figure 12).

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

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

An Abelian period $(h, p)$ of $w$ is non-deducible if there does not exist another Abelian period $(h', p')$ of $w$ such that $\text{Cut}_w(h, p) \subset \text{Cut}_w(h', p')$. 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.
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 \( h + 2p \ll |w| \). See the difference with Figure [10].

References