Computing Abelian Covers and Abelian Runs

Shohei Matsuda, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University, Japan
{shohei.matsuda,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp

Abstract. Two strings u and v are said to be *Abelian equivalent* if u is a permutation of the characters of v. We introduce two new regularities on strings w.r.t. Abelian equivalence, called *Abelian covers* and *Abelian runs*, which are generalizations of covers and runs of strings, respectively. We show how to determine in $O(n)$ time whether or not a given string w of length n has an Abelian cover. Also, we show how to compute an $O(n^2)$-size representation of (possibly exponentially many) Abelian covers of w in $O(n^2)$ time. Moreover, we present how to compute all Abelian runs in w in $O(n^2)$ time, and state that the maximum number of all Abelian runs in a string of length n is $\Omega(n^2)$.

Keywords: Abelian equivalence on strings, Parikh vectors, Abelian repetitions, covers of strings, string algorithms

1 Introduction

The study of *Abelian equivalence* of strings dates back to at least the early 60's, as seen in the paper by Erdös [6]. Two strings u, v are said to be *Abelian equivalent* if u is a permutation of the characters appearing in v. For instance, strings `aabba` and `baaba` are Abelian equivalent. Abelian equivalence of strings has attracted much attention and has been studied extensively in several contexts.

A variant of the pattern matching problem called the *jumbled pattern matching problem* is to determine whether there is a substring of an input text string w that is Abelian equivalent to a given pattern string p. There is a folklore algorithm to solve this problem in $O(n + m + \sigma)$ time using $O(\sigma)$ space, where n is the length of w, m is the length of p, and σ is the alphabet size. Assuming $m \leq n$ and all characters appear in w, the algorithm runs in $O(n)$ time and $O(\sigma)$ space. The indexed version of the jumbled pattern matching problem is more challenging, where the task is to preprocess an input text string w so that, given a query pattern string p, we can quickly determine whether or not there is a substring of w that is Abelian equivalent to p. For binary strings, there exists a data structure which occupies $O(n)$ space and answers the above query in $O(1)$ time. Burcsi et al. [2] and Moosa and Rahman [16] independently developed an $O(n^2/\log n)$-time algorithm to construct this data structure, and later Gagie et al. [9] showed an improved $O(n^2/\log^2 n)$-time algorithm. Very recently, Hermelin et al. [10] proposed an $n^2/2^{\Omega(\log n/\log\log n)^{\frac{1}{2}}}$-time solution to the problem for binary strings. For any constant-size alphabets, Kociumaka et al. [12] showed an algorithm that requires $O(n^2 \log^2 \log n/\log n)$ preprocessing time and $O((\log n/\log\log n)^{2\sigma-1})$ query time. Amir et al. [1] showed lower bounds on the indexing version of the jumbled pattern matching problem under a 3SUM-hardness assumption.

Abelian periodicity of strings has also been extensively studied in string algorithmics. A string w is said to have a full Abelian period if w is a concatenation $w_1 \cdots w_k$ of k Abelian equivalent strings $w_1, \ldots, w_k$ with $k \geq 2$, and the length of w_1 is called a full Abelian period of w. A string w is said to have an Abelian period if $w = yz$,

Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

where y has some full Abelian period d, and z is a non-empty string shorter than d such that the number of each character a contained in z is no more than that contained in the prefix $y[1..d]$ of length d of y. A string w is said to have a weak Abelian period d if $w = xy$, where y has some Abelian period d, and x is a non-empty string shorter than d such that the number of each character a contained in x is no more than that contained in the prefix $y[1..d]$ of length d of y. Fici et al. [7] proposed an $O(n \log\log n)$-time algorithm to compute all full Abelian periods and $O(n^2)$-time algorithm to compute all Abelian periods for a given string of length n. Recently, Kociumaka et al. [13] showed an optimal $O(n)$-time algorithm to compute all full Abelian periods, and an improved $O(n(\log\log n + \log\sigma))$-time algorithm to compute all Abelian periods, where σ is the alphabet size. Fici et al. [8] presented an $O(n^2\sigma)$-time algorithm to compute all weak Abelian periods, and later Crochemore et al. [3] gave an improved $O(n^2)$-time solution to the problem.

In the field of word combinatorics, Erdös [6] posed a question whether there exists an infinitely long string which contains no Abelian squares. A substring s of a string w is called an Abelian square if $s = s_1 s_2$ such that s_1 and s_2 are Abelian equivalent. Entringer et al. [5] proved that any infinite word over a binary alphabet contains arbitrary long Abelian squares. On the other hand, Pleasants [18] showed a construction of an infinitely long string which contains no Abelian squares over an alphabet of size 5, and later, Keränen [11] showed a construction over an alphabet of size 4. An interesting question in the field of string algorithmics is how efficiently we can compute the Abelian repetitions that occur in a given string of finite length n. Cummings and Smyth [4] presented an algorithm to compute all Abelian squares in $O(n^2)$ time. They also showed that there exist $\Omega(n^2)$ Abelian squares in a string of length n. Crochemore et al. [3] showed another $O(n^2)$-time algorithm to compute all squares in a string of length n.

In this paper, we introduce two new regularities on strings w.r.t. Abelian equivalence, called *Abelian covers* and *Abelian runs*, which are generalizations of covers [15] and runs [14] of strings, respectively, and we propose non-trivial algorithms to compute these new string regularities. A set C of intervals is called an Abelian cover of a string w if the substrings corresponding to the intervals in C are all Abelian equivalent, and every position in w is contained in at least one interval in C. We show that, given a string w of length n, we can determine whether or not w has an Abelian cover in optimal $O(n)$ time. Also, we present an $O(n^2)$-time algorithm to compute an $O(n^2)$-size representation of all (possibly exponentially many) Abelian covers of w. A substring s of w is said to be an Abelian run of w if s is a maximal substring which has a weak Abelian period. As a direct consequence from the result by Cummings and Smyth [4], it is shown that the maximum number of all Abelian runs in a string of length n is $\Omega(n^2)$. Then, we propose an $O(n^2)$-time algorithm to compute all Abelian runs in a given string of length n.

2 Preliminaries

Let $\Sigma = \{c_1, \ldots, c_\sigma\}$ be an ordered *alphabet*. We assume that for each $c_i \in \Sigma$, its rank i in Σ is already known and can be computed in constant time. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is the string of length 0, namely, $|\varepsilon| = 0$. For a string $w = xyz$, strings x, y, and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The i-th character of a string w of length n is denoted by $w[i]$ for $1 \le i \le n$. For $1 \le i \le j \le n$, let $w[i..j] = w[i] \cdots w[j]$,

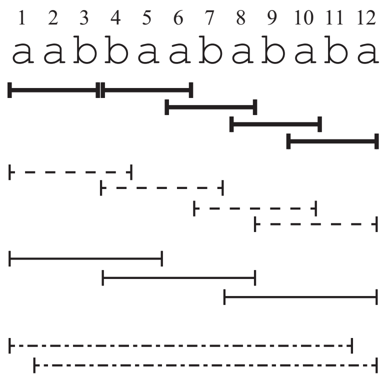

Figure 1. String aabbaababab a over a binary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ has an Abelian cover $\{[1,3],[4,6],[6,8],[8,10],[10,12]\}$ of length 3 with Parikh vector $\langle 2,1\rangle$, an Abelian cover $\{[1,4],[4,7],[7,10],[9,12]\}$ of length 4 with Parikh vector $\langle 2,2\rangle$, an Abelian cover $\{[1,5],[4,8],[8,12]\}$ of length 5 with Parikh vector $\langle 3,2\rangle$, and an Abelian cover $\{[1,11],[2,12]\}$ of length 11 with Parikh vector $\langle 6,5\rangle$. We remark that this string has other Abelian covers than the above ones.

i.e., $w[i..j]$ is the substring of w starting at position i and ending at position j in w. For convenience, let $w[i..j] = \varepsilon$ if $j < i$. For any $0 \le i \le n$, strings $w[1..i]$ and $w[i..n]$ are called prefixes and suffixes of w, respectively.

For any string w of length $n \ge 2$, a set $I = \{[b_1, e_1], \ldots, [b_{|I|}, e_{|I|}]\}$ of intervals is called a *cover* of w if $\bigcup_{1 \le k \le |I|}[b_k, e_k] = [1, n]$ and $[b_k, e_k] \ne [1, n]$ for every $1 \le k \le |I|$. Whenever we write $C = \{[b_1, e_1], \ldots, [b_{|C|}, e_{|C|}]\}$ for a cover C of a string w, then we assume that $b_j < b_{j+1}$ for all $1 \le k < |C|$.

Two strings $v, w \in \Sigma^*$ are said to be *Abelian equivalent* if v is a permutation of the characters in w. A Parikh vector [17] of a string $w \in \Sigma^*$, denoted P_w, is an array of length σ such that for any $1 \le i \le \sigma$, $P_w[i]$ stores the number of occurrences of character c_i in w. Let $\preceq$ be a partial order of Parikh vectors P_v and P_w for any strings $v, w \in \Sigma^*$ such that

$$\begin{aligned} &P_v = P_w \quad \text{if } P_v[i] = P_w[i] \text{ for all } 1 \le i \le \sigma, \text{ and} \\ &P_v \prec P_w \quad \text{if } P_v \ne P_w \text{ and } P_v[i] \le P_w[i] \text{ for all } 1 \le i \le \sigma. \end{aligned}$$

For instance, for strings $v = \mathtt{aababc}$ and $w = \mathtt{baba}$ over an ordered alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $P_v = [3, 2, 1]$ and $P_w = [2, 2, 0]$, and therefore $P_w \prec P_v$. Clearly, strings v, w are Abelian equivalent iff $P_v = P_w$. For any two strings $v, w \in \Sigma^*$, let $P_v \oplus P_w = P_{vw}$, namely, $(P_v \oplus P_w)[i] = P_v[i] + P_w[i]$ for each $1 \le i \le \sigma$.

A cover $C = \{[b_1, e_1], \ldots, [b_{|C|}, e_{|C|}]\}$ of a string w is called an *Abelian cover* of w if $P_{w[b_1..e_1]} = P_{w[b_j..e_j]}$ for all $1 < j \le |C|$. Clearly $e_j - b_j = e_1 - b_1$ holds for all $1 < j \le |C|$. The *length* and *size* of an Abelian cover $C = \{[b_1, e_1], \ldots, [b_{|C|}, e_{|C|}]\}$ of a string are $e_1 - b_1 + 1$ and $|C|$, respectively. See Figure 1 for examples of Abelian covers of a string.

A non-empty substring u of a string w is called an *Abelian repetition* with period d if $|u|$ is a multiple of an integer d ($1 \le d \le \frac{|u|}{2}$) and $P_{u[(k-1)d+1..kd]} = P_{u[kd+1..(k+1)d]}$ for all $1 \le k < \frac{|u|}{d}$. If $d = \frac{|u|}{2}$, then u is called an *Abelian square* of w. A substring $w[i..j]$ of a string w is called a *maximal Abelian repetition* of w if $w[i..j]$ is a non-extensible Abelian repetition with period d in w, namely, if $w[i..j]$ is an Abelian repetition satisfying (1) $P_{w[i-d..i-1]} \ne P_{w[i..i+d-1]}$ or $i - d < 0$ and (2) $P_{w[j-d+1..j]} \ne P_{w[j+1..j+d]}$ or $j + d > n$. A substring $w[i - h..j + h']$ of a string w is called an *Abelian run*

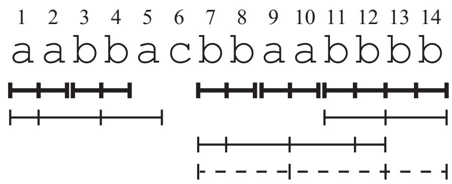

Figure 2. String aabbacbbaabbbb over a ternary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ has Abelian runs $(0, 1, 2, 1, 0)$, $(0, 3, 4, 1, 0)$, $(0, 7, 8, 1, 0)$, $(0, 9, 10, 1, 0)$, and $(0, 11, 14, 1, 0)$ of period 1, Abelian runs $(1, 2, 5, 2, 0)$, $(1, 8, 11, 2, 1)$, and $(0, 11, 14, 2, 0)$ of period 2, and an Abelian run $(0, 7, 12, 3, 2)$ of period 3.

of w if $w[i..j]$ is a maximal Abelian repetition with period d of w and $h, h' \geq 0$ are the largest integers satisfying $P_{w[i-h..i-1]} \prec P_{w[i..i+d]}$ and $P_{w[j+1..j+h']} \prec P_{w[j-d..j]}$, respectively. Each Abelian run $w[i-h..j+h']$ of period d in w is represented by a 5-tuple (h, i, j, d, h'), where h and h' are called the left hand and the right hand of the Abelian run, respectively. See Figure 2 for examples of Abelian runs in a string.

In this paper, we consider the following problems.

Problem 1 (Abelian cover existence). Given a string w, determine whether or not w has an Abelian cover.

Problem 2 (All Abelian covers). Given a string w, compute all Abelian covers of w.

Problem 3 (All Abelian runs). Given a string w, compute all Abelian runs in w.

3 Algorithms

In this section, we present our algorithms to solve the problems stated in the previous section. For simplicity, we assume that all characters in Σ appear in a given string w of length n, which implies $\sigma \leq n$.

3.1 Abelian cover existence

In this subsection, we consider Problem 1 of determining whether there exists an Abelian cover of a given string w of length n. Note that there exists an infinite sequence of strings over a binary alphabet which have no Abelian covers (e.g., $\mathtt{a}^{n-1}\mathtt{b}$ has no Abelian covers), and therefore Problem 1 is of interest. The following lemma is a key to our solution to the problem.

Lemma 4. *If there exists an Abelian cover $C = \{[b_1, e_1], \ldots, [b_{|C|}, e_{|C|}]\}$ of arbitrary size for a string w, then there exists an Abelian cover C' of size exactly 2 for w.*

Proof. It is clear when $|C| = 2$. Consider the case where $|C| \geq 3$. Since C is an Abelian cover of w, $P_{w[b_1, e_1]} = P_{w[b_{|C|}..e_{|C|}]}$. This implies that $P_{w[b_1..e_1]} \oplus P_{w[e_1+1..b_{|C|}-1]} = P_{w[e_1+1..b_{|C|}-1]} \oplus P_{w[b_{|C|}..e_{|C|}]}$. Therefore, $C' = \{[b_1..b_{|C|}-1], [e_1+1, e_{|C|}]\} = \{[1..b_{|C|}-1], [e_1+1, |w|]\}$ is an Abelian cover of size 2 of w. (See also Figure 3.) □

Using the above lemma, we obtain the following.

Theorem 5. *Given a string w of length n, we can determine whether w has an Abelian cover or not in $O(n)$ time with $O(\sigma)$ working space.*

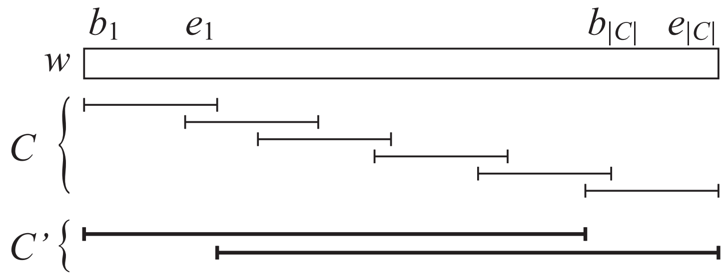

Figure 3. Illustration for Lemma 4. If a string w has a cover C, then w always has a cover C' of size 2.

Proof. By Lemma 4, Problem 1 of deciding whether there exists an Abelian cover of a given string w reduces to finding an Abelian cover of size 2 of w. Therefore, it suffices to find a prefix and a suffix of the same length ℓ such that $P_{w[1..\ell]} = P_{w[n-\ell+1..n]}$. To find such a prefix and a suffix, for each $1 \leq j \leq \lfloor \frac{n}{2} \rfloor$ in increasing order, we maintain an invariant d_j which represents the number of entries of $P_{w[1..\ell]}$ and $P_{w[n-\ell+1..n]}$ whose values differ, i.e.,

$$d_j = \{k \mid P_{w[1..j]}[k] \neq P_{w[n-j+1..n]}[k], 1 \leq k \leq \sigma\}.$$

Clearly w has an Abelian cover of size 2 iff $d_j = 0$ for some j. For any $1 \leq j \leq \lfloor \frac{n}{2} \rfloor$, let $w[j] = c_s$ and $w[n-j+1] = c_t$. We can update $P_{w[1..j-1]}$ (resp. $P_{w[n-j..n]}$) to $P_{w[1..j]}$ (resp. $P_{w[n-j+1..n]}$) in $O(1)$ time, increasing the value stored in the sth entry (resp. the tth entry) by 1. Also, d_j can be computed in $O(1)$ time from d_{j-1}, $P_{w[1..j-1]}[s]$, $P_{w[1..j]}[s]$, $P_{w[n-j..n]}[t]$, and $P_{w[n-j+1..n]}[t]$. Hence, the algorithm runs in a total of $O(n)$ time. The extra working space of the algorithm is $O(\sigma)$, due to the two Parikh vectors we maintain. $\square$

The following corollary is immediate from Lemma 4 and Theorem 5:

Corollary 6. *We can compute the longest Abelian cover of a given string of length n in $O(n)$ time with $O(\sigma)$ working space, if it exists.*

3.2 All Abelian covers

In this subsection, we consider Problem 2 of computing all Abelian covers of a given string w of length n. Note that the number of all Abelian covers of a string can be exponentially large w.r.t. n. For instance, string a^n has $\sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} 2^{n-k-1}$ Abelian covers of length at least $\lceil \frac{n}{2} \rceil$. This is because, for any $k \geq \lceil \frac{n}{2} \rceil$, the union of $\{[1,k],[n-k+1,n]\}$ and any subset of $\{[2,k+1],[3,k+2],\ldots,[n-k,n-1]\}$ is an Abelian cover of length k for a^n. Therefore, we consider to compute a "compact" representation of all Abelian covers of a given string.

Theorem 7. *Given a string w of length n, we can compute an $O(n^2)$-size representation of all Abelian covers of w in $O(n^2)$ time and $O(n)$ working space. Given a set I of s intervals sorted by the beginning positions of the intervals, the representation allows us to check if I is an Abelian cover of w in $O(s)$ time.*

Proof. For each $1 \leq \ell \leq n-1$, we compute a subset S_ℓ of positions in w such that $S_\ell = \{i \mid P_{w[i..i+\ell-1]} = P_{w[1..\ell]}, 1 \leq i \leq n-\ell+1\}$. Then, there exists an Abelian cover of length ℓ for w iff the distance between any two adjacent positions in S_ℓ is at most ℓ. If S_ℓ satisfies the above condition, then we represent S_ℓ as a bit vector B_ℓ of length n such that $B_\ell[i] = 1$ if $i \in S_\ell$, and $B_\ell[i] = 0$ otherwise. If S_ℓ does not satisfy the above condition, then we discard it. Now, given a set I of s intervals sorted by the beginning positions of the intervals, we first check if I is a cover of w and if each interval is of equal length ℓ in a total of $O(s)$ time. If I satisfies both conditions, then we can check if I is a subset of S_ℓ in $O(s)$ time, using the bit vector B_ℓ. Using a similar method to Theorem 5, for each $1 \leq \ell \leq n-1$, S_ℓ and its corresponding bit vector B_ℓ can be computed in $O(n)$ time. Hence, the overall time complexity of the algorithm is $O(n^2)$. The working space (excluding the output) is $O(n)$, since $|S_\ell| = O(n)$ for any ℓ and $\sigma = O(n)$. ☐

Given a set I of s intervals, a naïve algorithm to check whether I is an Abelian cover of length ℓ requires $O(s\ell)$ time. Therefore, the solution of Theorem 7 with $O(s)$ query time is more efficient than the naïve method.

3.3 All Abelian runs

In this subsection, we consider Problem 3 of computing all Abelian runs in a given string w of length n. We follow and extend the results by Cummings and Smyth [4] on the maximum number of all maximal Abelian repetitions in a string, and an algorithm to compute them. We firstly consider a lower bound on the maximum number of Abelian runs in a string.

Lemma 8 ([4]). *String* $(\mathtt{aababbaba})^n$ *of length* $8n$ *has* $\Theta(n^2)$ *maximal Abelian repetitions (in fact maximal Abelian squares).*

Since the number of Abelian runs in a string is equal to that of maximal Abelian repetitions in that string, the following theorem is immediate:

Theorem 9. *The maximum number of Abelian runs in a string* w *of length* n *is* $\Omega(n^2)$.

Next, we show how to compute all Abelian runs in a given string.

Theorem 10. *Given a string* w *of length* n, *we can compute all Abelian runs in* w *in* $O(n^2)$ *time and space.*

Proof. We firstly compute all Abelian squares in w using the algorithm proposed by Cummings and Smyth [4]. For each $1 \leq i \leq n$, we compute a set L_i of integers such that

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

Note that substring $w[i-\ell..i+\ell+1]$ is an Abelian square centered at position i iff $\ell \in L_i$. After computing all L_i's, we store them in a two dimensional array L of size $\lfloor \frac{n}{2} \rfloor \times n-1$ such that $L[\ell, i] = 1$ if $\ell \in L_i$ and $L[\ell, i] = 0$ otherwise. All entries of L are initialized *unmarked*. Then, for each $1 \leq \ell \leq n-1$, all maximal Abelian repetitions of period ℓ can be computed by in $O(n)$ time, as follows. We scan the ℓth row of L from left to right for increasing $i = 1, \ldots, n-1$, and if we encounter an unmarked entry (ℓ, i) such that $L[\ell, i] = 1$, then we compute the largest non-negative integer k such that $L[\ell, i + p\ell + 1] = 1$ for all $1 \leq p \leq k$ in $O(k)$ time, by skipping every $\ell - 1$ entries in between. This gives us a maximal Abelian repetitions with period ℓ

L

$\ell \backslash i$	1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	0	0	0	0	0	0
2	0	0	0	0	1	1	1	1	0	0
3	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0

Figure 4. The two dimensional array L for string `caaabababac`. The maximal Abelian repetitions `aaa` of period 1 starting at position 3 is found by concatenating two Abelian squares represented by $L[1,2]$ and $L[1,3]$. The maximal Abelian repetition `ababab` of period 2 starting at position 4 is found by concatenating two Abelian squares represented by $L[2,5]$ and $L[2,7]$. The maximal Abelian repetition `bababa` of period 2 starting at position 5 is found by concatenating two Abelian squares represented by $L[2,6]$ and $L[2,8]$. Finally, the maximal Abelian repetition `aababa` of period 3 starting at position 3 is found from $L[3,5]$ (this is not extensible to the right). Every concatenation procedure (represented by an arrow) starts from an unmarked entry, and once an entry is involved in computation of a maximal Abelian repetition, it gets marked. This way the algorithm runs in time linear in the size of L, which is $O(n^2)$.

starting at position $i - \ell + 1$ and ending at position $i + (k+1)\ell$. After computing the largest integer k, we mark the entries $L[\ell, i + p\ell + 1]$ for all $-1 \leq p \leq k$ in $O(k)$ time. Since each unmarked entry of the ℓth row is marked at most once and is accessed by a constant number of times, and since the above procedure starts only from unmarked entries, it takes a total of $O(n)$ time for each ℓ. Therefore, this takes a total of $O(n^2)$ time for all $1 \leq \ell \leq \lfloor \frac{n}{2} \rfloor$. (See also Figure 4 for a concrete example of the two dimensional array L and how to compute all maximal Abelian repetitions from L).

What remains is how to compute the left and right hands of each maximal Abelian runs. If we compute the left and right hands naïvely for all the maximal Abelian repetitions, then it takes a total of $O(n^3)$ time due to Theorem 9. To compute the left and right hands in a total of $O(n^2)$ time, we use the following property on Abelian repetitions: For each $1 \leq i \leq n$, let P_i be the set of positive integers such that for each $\ell \in P_i$ there exists a maximal Abelian repetition whose period is ℓ and beginning position is $i - \ell + 1$. For any $1 \leq j \leq |P_i|$, let ℓ_j denote the jth smallest element of P_i. We process ℓ_j in increasing order of $j = 1, \ldots, |P_i|$. Let h_j denote the left hand of the Abelian run that is computed from the maximal Abelian repetition whose period is ℓ_j and beginning position is $i - \ell_j + 1$. For any $1 \leq j < |P_i|$, assume that we have computed the length of the left hand h_{j-1} of the maximal Abelian repetition beginning at position $i - \ell_{j-1} + 1$. We are now computing the left hand h_j of the next Abelian run. There are two cases to consider:

1. If $j = 1$ or $\ell_{j-1} + h_{j-1} \leq \ell_j$, then we compute the left hand h_j of the maximal Abelian repetition beginning at position $i - \ell_j + 1$, by comparing the Parikh vector $P_{w[i-\ell_j-k..i-\ell_j]}$ for increasing k from 0 up to $h_j + 1$, with the Parikh vector $P_{w[i-\ell_j+1..i]}$. This takes $O(h_j)$ time. (See also Figure 5).
2. If $\ell_{j-1} + h_{j-1} > \ell_j$, then

$$P_{w[i-\ell_{j-1}-h_{j-1}+1..i-\ell_j]} \prec P_{w[i-\ell_{j-1}-h_{j-1}+1..i-\ell_{j-1}]} \prec P_{w[i-\ell_{j-1}+1..i]} \prec P_{w[i-\ell_j+1..i]}.$$

This implies that $h_j \geq \ell_{j-1} + h_{j-1} - \ell_j$. We can compute $P_{w[i-\ell_{j-1}-h_j..i-\ell_j]}$ from $P_{w[i-\ell_{j-1}-h_{j-1}+1..i-\ell_{j-1}]}$ in $O(\ell_j - \ell_{j-1})$ time. Then, we compute the left hand h_j

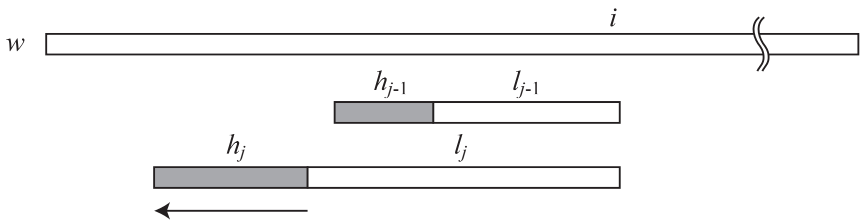

Figure 5. Illustration for Case 1 where $j = 1$ or $\ell_{j-1} + h_{j-1} \leq \ell_j$ of Theorem 10. We can compute the left hand h_j in $O(h_j)$ time by extending the substring to the left from position $i - \ell_j$.

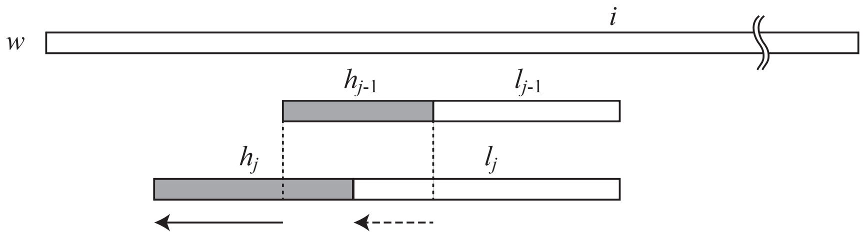

Figure 6. Illustration for Case 2 where $\ell_{j-1} + h_{j-1} > \ell_j$ of Theorem 10. In this case, we know that h_j is at least $\ell_{j-1} + h_{j-1} - \ell_j$. The Parikh vector of $w[i - \ell_{j-1} - h_j..i - \ell_j]$ can be computed in $O(\ell_j - \ell_{j-1})$ time by a scan of substring $w[i - \ell_j + 1..i - \ell_{j-1}]$ (dashed arrow). Then, we can compute the left hand h_j in a total of $O(h_j + \ell_j - h_{j-1} - \ell_{j-1})$ time by extending the substring to the left from position $i - \ell_{j-1} - h_{j-1}$ (solid arrow).

by comparing the Parikh vector $P_{w[i-\ell_{j-1}-h_{j-1}+1-k..i-\ell_j]}$ for increasing k from 0 up to $h_j + \ell_j - h_{j-1} - \ell_{j-1} + 1$. This takes $O(h_j + \ell_j - h_{j-1} - \ell_{j-1})$ time. (See also Figure 6).

Let J_i^1 and J_i^2 be the disjoint subsets of $[1, |P_i|]$ such that $j \in J_i^1$ if $\ell_j \in P_i$ corresponds to Case 1, and $j \in J_i^2$ if $\ell_j \in P_i$ corresponds to Case 2. Then, the summations $\sum_{j \in J_i^1}(h_j)$, $\sum_{j \in J_i^2}(\ell_j - \ell_{j-1})$, and $\sum_{j \in J_i^2}(h_j + \ell_j - \ell_{j-1} - h_{j-1})$ corresponding to the time costs for Cases 1 and 2 are all bounded by $O(n)$. Therefore, it takes a total of $O(n)$ time to compute the left hands of all Abelian runs that correspond to P_i, and the right hands can be computed similarly. Hence, it takes a total of $O(n^2)$ time to compute all Abelian runs in w. The working space of the algorithm is dominated by the two dimensional array L, which takes $O(n^2)$ space. □

4 Conclusions and future work

Abelian regularities on strings were initiated by Erdös [6] in the early 60's, and since then they have been extensively studied in Stringology. In this paper, we introduced new regularities on strings with respect to Abelian equivalence on strings, which we call *Abelian covers* and *Abelian runs*. Firstly, we showed an optimal $O(n)$-time $O(\sigma)$-space algorithm to determine whether or not a given string w of length n over an alphabet of size σ has an Abelian cover. As a consequence of this, we can compute the longest Abelian cover of w in $O(n)$-time. Secondly, we showed an $O(n^2)$-time algorithm to compute an $O(n^2)$-space representation of all (possibly exponentially many) Abelian covers of a string of length n. Thirdly, we presented an $O(n^2)$-time algorithm to compute all Abelian runs in a string of length n. We also remarked that the maximum number of Abelian runs in a string of length n is $\Omega(n^2)$.

Our future work includes the following:

– The algorithm of Theorem 7 allows us to compute a shortest Abelian cover of a given string of length n in $O(n^2)$ time. Can we compute a *shortest* Abelian cover in $o(n^2)$ time?
– The algorithm of Theorem 10 requires $\Theta(n^2)$ time to compute all Abelian runs of a given string w of length n. This is due to the two dimensional array L of $\Theta(n^2)$ space. Can we compute all Abelian runs in w in optimal $O(n+r)$ time, where r is the number of Abelian runs in w?

References

1. A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein: *On hardness of jumbled indexing*, in Proc. ICALP 2014 (to appear), 2014, Preprint is available at `http://arxiv.org/abs/1405.0189`.
2. P. Burcsi, F. Cicalese, G. Fici, and Zs. Lipták: *On table arrangements, scrabble freaks, and jumbled pattern matching*, in Proc. FUN 2010, 2010, pp. 89–101.
3. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, J. Pachocki, J. Radoszewski, W. Rytter, W. Tyczynski, and T. Walen: *A note on efficient computation of all Abelian periods in a string.* Inf. Process. Lett., 113(3) 2013, pp. 74–77.
4. L. J. Cummings and W. F. Smyth: *Weak repetitions in strings.* J. Combinatorial Mathematics and Combinatorial Computing, 24 1997, pp. 33–48.
5. R. C. Entringer and D. E. Jackson: *On nonrepetitive sequences.* J. Comb. Theory, Ser. A, 16(2) 1974, pp. 159–164.
6. P. Erdös: *Some unsolved problems.* Hungarian Academy of Sciences Mat. Kutató Intézet Közl, 6 1961, pp. 221–254.
7. G. Fici, T. Lecroq, A. Lefebvre, Élise Prieur-Gaston, and W. F. Smyth: *Quasi-linear time computation of the Abelian periods of a word*, in Proc. PSC 2012, 2012, pp. 103–110.
8. G. Fici, T. Lecroq, A. Lefebvre, and E. Prieur-Gaston: *Computing Abelian periods in words*, in Proc. PSC 2011, 2011, pp. 184–196.
9. T. Gagie, D. Hermelin, G. M. Landau, and O. Weimann: *Binary jumbled pattern matching on trees and tree-like structures*, in Proc. ESA 2013, 2013, pp. 517–528.
10. D. Hermelin, G. M. Landau, Y. Rabinovich, and O. Weimann: *Binary jumbled pattern matching via all-pairs shortest paths.* CoRR, abs/1401.2065 2014.
11. V. Keränen: *Abelian squares are avoidable on 4 letters*, in Proc. ICALP 1992, 1992, pp. 41–52.
12. T. Kociumaka, J. Radoszewski, and W. Rytter: *Efficient indexes for jumbled pattern matching with constant-sized alphabet*, in Proc. ESA 2013, 2013, pp. 625–636.
13. T. Kociumaka, J. Radoszewski, and W. Rytter: *Fast algorithms for Abelian periods in words and greatest common divisor queries*, in Proc. STACS 2013, 2013, pp. 245–256.
14. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. FOCS 1999, 1999, pp. 596–604.
15. Y. Li and W. F. Smyth: *Computing the cover array in linear time.* Algorithmica, 32(1) 2002, pp. 95–106.
16. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings.* Inf. Process. Lett., 110(18–19) 2010, pp. 795–798.
17. R. Parikh: *On context-free languages.* J. ACM, 13(4) 1966, pp. 570–581.
18. P. A. B. Pleasants: *Non-repetitive sequences.* Mathematical Proceedings of the Cambridge Philosophical Society, 68 9 1970, pp. 267–274.