REGULAR-SS7: Subquadratic-Time Algorithms for Abelian Stringology Problems

Tomasz Kociumaka[1,⋆], Jakub Radoszewski[1,⋆⋆,⋆⋆⋆], and Bartłomiej Wiśniewski[1]

Faculty of Mathematics, Informatics and Mechanics,
University of Warsaw, Warsaw, Poland
[kociumaka,jrad]@mimuw.edu.pl, b.wisniewski@students.mimuw.edu.pl

Abstract. We propose the first subquadratic-time algorithms to a number of natural problems in abelian pattern matching (also called jumbled pattern matching) for strings over a constant-sized alphabet. Two strings are considered equivalent in this model if the numbers of occurrences of respective symbols in both of them, specified by their so-called Parikh vectors, are the same. We propose the following algorithms for a string of length n:

- Counting and finding longest/shortest abelian squares in $O(n^2/\log^2 n)$ time. Abelian squares were first considered by Erdös (1961); Cummings and Smyth (1997) proposed an $O(n^2)$-time algorithm for computing them.
- Computing all shortest (general) abelian periods in $O(n^2/\sqrt{\log n})$ time. Abelian periods were introduced by Constantinescu and Ilie (2006) and the previous, quadratic-time algorithms for their computation were given by Fici et al. (2011) for a constant-sized alphabet and by Crochemore et al. (2012) for a general alphabet.
- Finding all abelian covers in $O(n^2/\log n)$ time. Abelian covers were defined by Matsuda et al. (2014).
- Computing abelian border array in $O(n^2/\log^2 n)$ time.

This work can be viewed as a continuation of a recent very active line of research on subquadratic space and time jumbled indexing for binary and constant-sized alphabets (e.g., Moosa & Rahman, 2012). All our algorithms work in linear space.

Keywords: jumbled pattern matching, jumbled indexing, abelian period, abelian square

1 Introduction

Algorithmic abelian stringology has been extensively studied in the recent years. Abelian pattern matching (also called jumbled pattern matching) can be viewed

⋆ Supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.

⋆⋆ Supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' program in 2015-2016 grant no 0392/IP3/2015/73.

⋆⋆⋆ The author is a Newton International Fellow at King's College London.

as an approximate variant of regular pattern matching. The problem that has received most attention in this area is jumbled indexing; see [2,3,4,11,12,15,16]. In the binary case, initially it was known that there exists a jumbled index of linear size that can answer in constant time queries asking if there is a substring of the text that is commutatively equivalent to a pattern specified by the number of zeroes and ones (see [4]). However, the straightforward construction time of such an index was quadratic. Due to a number of works [2,15,16] $O(n^2/\log^2 n)$-time construction algorithm of such an index was obtained. This line of research eventually lead to a very recent breakthrough $O(n^{1.859})$-time algorithm by Chan and Lewenstein [3].

In this work we present the first subquadratic-time algorithms computing several basic notions of abelian stringology in the case of binary and constant-sized alphabet. This includes:

- abelian squares, that were first considered from a combinatorial perspective by Erdös [8] and from the algorithmic perspective by Cummings and Smyth [7] (in the latter case, under the name of weak repetitions),
- abelian periods, defined by Constantinescu and Ilie [5],
- abelian covers, introduced by Matsuda et al. [14],
- natural notions of abelian borders and abelian border array.

Cummings and Smyth [7] presented an $O(n^2)$-time algorithm for computing all abelian squares in a string of length n. Fici et al. [9] showed an $O(n^2)$-time algorithm computing abelian periods in a string over a constant-sized alphabet and Crochemore et al. [6] solved this problem in $O(n^2)$ time for any alphabet. Other types of abelian periods are known, including regular and full abelian periods (see [10]), for which linear-time or almost linear-time algorithms are known [13].

Matsuda et al. [14] used a slightly different definition of abelian covers than the one that we use here (that is, they considered two abelian covers different if and only if the set of starting positions of the occurrences of the covers are different) and obtained an $O(n^2)$-time algorithm for computing a representation of all such (possibly exponentially many) abelian covers.

Our results:

- computing the longest, the shortest and the number of all abelian squares in $O(n^2/\log^2 n)$ time and computing the distribution of all abelian squares over their center positions in $O(n^2/\log n)$ time;
- computing the shortest (general) abelian period and all its occurrences, and an $O(n^2/\log n)$-representation of all abelian periods in $O(n^2/\sqrt{\log n})$ time;
- computing the shortest abelian cover in $O(n^2/\log n)$ time;
- computing the abelian border array in $O(n^2/\log^2 n)$ time and a representation of all abelian borders of prefixes of the string in $O(n^2/\log n)$ time (showing a similarity between computing abelian borders and abelian squares).

We assume constant-sized alphabet and the word-RAM model of computation. In all algorithms we apply in a fancy way the technique called *four-Russian trick*,

which was first involved in abelian pattern matching by Moosa and Rahman [16] in the problem of jumbled indexing. All our algorithms work in linear space, excluding the size of the result.

Structure of the paper: The following section includes definitions of the terms used in abelian stringology, as well as the basic notation. In Sections 3-5 we show our results for a binary alphabet. In Section 6 we show that all presented algorithms can be applied to the case of any constant-sized alphabet, preserving both time and space complexity. The Conclusions Section contains a brief discussion of possible future work.

2 Preliminaries

Let s be a string of length $n = |s|$ over an alphabet Σ of size $\sigma = |\Sigma|$. By $s[i]$, where $1 \leq i \leq n$, we denote the i-th symbol of s, and by $s[i, j]$ we denote a substring of s consisting of all symbols of s on positions from i to j inclusive. Substrings of the form $s[1, i]$ are called prefixes of s and substrings of the form $s[i, n]$ are called suffixes of s.

A *Parikh vector* of a string s, $\mathcal{P}(s)$, is a vector of size σ, where the element $\mathcal{P}(s)[l]$ stores the number of occurrences of symbol l in string s, i.e. $\mathcal{P}(s)[l] = x$ if and only if $|\{i : s[i] = l\}| = x$. The *norm* r of a Parikh vector R is the sum of its elements, $r = \Sigma_{l \in \Sigma} R[l]$. One can add or subtract Parikh vectors componentwise.

We say that two strings s and t are *abelian equivalent* (or commutatively equivalent), if s is a permutation (an anagram) of t, or equivalently $\mathcal{P}(s) = \mathcal{P}(t)$, and write $s \approx t$. We say that t is an *abelian factor* of s if and only if $\mathcal{P}(t)[l] \leq \mathcal{P}(s)[l]$ for every symbol l of the alphabet Σ. We say that a position i in a string t is an *abelian occurrence* of a string s if $s \approx t[i, i + |s| - 1]$.

We now proceed with the definitions of the main notions of abelian periodicity that we consider in the paper.

Definition 1. *An* abelian square *is a string t of length $2k$ such that $t[1, k] \approx t[k + 1, 2k]$.*

Definition 2. *A pair (i, k) is a* (general) abelian period *of s if and only if there exists an index j such that $s[i, i + k - 1] \approx s[i + k, i + 2k - 1] \approx \ldots \approx s[i + (j - 1)k, i + jk - 1]$ and $s[1, i - 1]$ and $s[i + jk, n]$ are abelian factors of $s[i, i + k - 1]$. A* regular abelian period *k of s is a general abelian period of s that starts at the first position of s, i.e. the general abelian period $(1, k)$ of s is a regular abelian period k of s.*

Definition 3. *An* abelian border *t of s is a prefix of s that is abelian equivalent to some suffix of s. A proper abelian border t of s is an abelian border t such that $|t| < n$. An* abelian border array *of s is a table π of length n such that $\pi[i]$ is the length of the longest proper abelian border of $s[1, i]$.*

Definition 4. *An abelian border t is an* abelian cover *of s if and only if the abelian occurrences of t in s* cover *the whole string s. That is, if I is a sorted sequence of positions i in s such that $s[i, i+|t|-1] \approx t$, then the first element of I is 1, the last element of I is $n-|t|+1$ and the differences between every two consecutive elements of I are no greater than $|t|$.*

In Sections 3-5 we consider strings over a binary alphabet $\Sigma = \{0, 1\}$. In this case, to check if two strings s and t of the same length are abelian equivalent it suffices to know only the number of 1s in each of them. We denote $ones(s) = |\{i : s[i] = 1\}|$. Let us note that $ones(s[i, j])$ can be computed in constant time using an array pre-computed in linear time that stores values of $ones(s[1, i])$ for every $1 \leq i \leq n$. We assume that $ones(s[i, j]) = 0$ for $i > j$.

In the binary case we assume that $s[i, i+m-1]$ for $m = O(\log n)$ can be stored using a constant number of integers (bit masks). An array that stores the representations of all such substrings for a given m can be computed in linear time.

3 Abelian squares

In this section we focus on finding the longest abelian squares which have their centers in every possible index i of string s. That is, for each $1 \leq i \leq n$ we want to find the largest k such that $s[i-k, i-1] \approx s[i, i+k-1]$.

This problem was solved in $O(n^2)$ time by Cummings and Smyth [7]. Let us briefly restate their solution in the case of a binary alphabet. For every i we take $k = \min(i-1, n-i+1)$ and store in a variable r the difference $ones(s[i-k, i-1]) - ones(s[i, i+k-1])$. If $r = 0$ then $s[i-k, i+k-1]$ is an abelian square. If not, then we decrease k by one, update r, and check again. We continue this way, until we find such k, for which the condition holds. One can see that this solution works in $O(n^2)$ time and $O(n)$ space. Our algorithm reduces the time complexity of this scheme by employing two optimizations.

3.1 First optimization

For the first optimization we will use a pre-computed 3-dimensional auxiliary array A of size $(2m+1) \times 2^m \times 2^m$; the particular value of m will be chosen later. If there exists a string w of length $2k \geq 2m$ with a m-letter prefix p, m-letter suffix q and $ones(w[1, k]) - ones(w[k+1, 2k]) = \Delta$, and the longest abelian square centered in the middle of w has length $k-l$ for $0 \leq l \leq m$, then $A[\Delta][p][q] = l$. Otherwise (if such an abelian square does not exist or it is shorter) $A[\Delta][p][q] = -1$.

One can see that when $|\Delta| > m$, then $A[\Delta][p][q] = -1$ for every p and q, so there is no need to store this information explicitly. Each field of array A can be computed in $O(m)$ time, so the whole array A can be constructed in $O(m^2 4^m)$ time. To compute each field of the array A we obviously do not need to generate the string w.

When analyzing a specific index i of string s and a specific length k we set a m-letter prefix of $s[i-k, i+k-1]$ as p, and a m-letter suffix of $s[i-k, i+k-1]$ as q, i.e. $p = s[i-k, i-k+m-1]$ and $q = s[i+k-m, i+k-1]$. Additionally let Δ be the difference in the number of 1s between the first and the second half of $s[i-k, i+k-1]$, i.e. $\Delta = ones(s[i-k, i-1]) - ones(s[i, i+k-1])$; see Fig. 1.

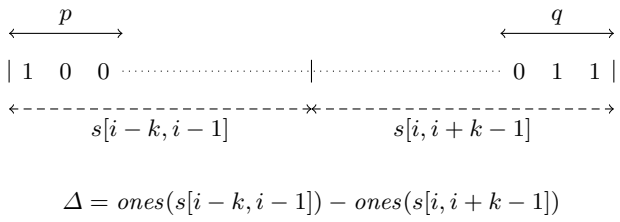

Fig. 1. How to choose p, q and Δ for given i, k and $m = 3$. If $\Delta = 1$, $A[\Delta][p][q] = 2$.

If $A[\Delta][p][q] \neq -1$ and k is the largest possible (that is, $k = \min(i-1, n-i+1)$), we have found the longest abelian square with a center in i. If $A[\Delta][p][q] = -1$, we know that the length of the longest abelian square is no greater than $k - m$. We can now decrement k by m and check the array A again. We finish either when the first abelian square is found or when k drops below m; in the latter case we find the longest abelian square in $O(m)$ time. This way we can compute the length of the longest abelian square with a center in i in $O(n/m+m)$ time and lengths of longest abelian squares for every center index i of s in $O(n^2/m + nm)$ time.

Lemma 5. *The longest abelian squares for every center index i of s can be computed in $O(n^2/\log n)$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \frac{\log n}{4} \right\rfloor$ the running time becomes:

$$\begin{aligned} O(n^2/m + nm + m^2 4^m) &= O(n^2/\log n + n\log n + 4^{\frac{\log n}{4}} \log^2 n) \\ &= O(n^2/\log n + \sqrt{n}\log^2 n) \\ &= O(n^2/\log n) \end{aligned}$$

The space complexity becomes:

$$\begin{aligned} O(n + m4^m) &= O(n + 4^{\frac{\log n}{4}} \log n) \\ &= O(n + \sqrt{n}\log n) \\ &= O(n) \end{aligned}$$

□

3.2 Second optimization

We will now use a second optimization, which will allow us to further decrease the time complexity. For this purpose we will pre-compute an auxiliary array B of size $(2m-1) \times 2^{2m-1} \times 2^{2m-1} \times 2^m \times 2^m$.

If there is a string w of length $2k+m-1$ ($k \geq m$) that satisfies all the conditions:

- $ones(w[1,k]) - ones(w[k+1,2k]) = \Delta$,
- its $(2m-1)$-letter prefix is p,
- its $(2m-1)$-letter suffix is q,
- $c = w[k+1, k+m-1]$,
- b is a bit mask of length m,

then $B[\Delta][p][q][c][b]$ stores all pairs (j,l) such that $j \leq m$, $l \leq m$, $b[j] = 0$ and $k-l$ is the length of the longest abelian square with a center in $k+j$.

Every field of array B can be computed in $O(m^2)$ time with a modified algorithm for finding abelian squares by Cummings and Smyth [7] that we have already recalled at the beginning of this section. We check m center indices and m lengths of potential abelian squares. The variable r can be initialized in constant time, using Δ.

In our algorithm, instead of analyzing a specific index i, we will analyze m indices i, $i+1$, ..., $i+m-1$ at a time. When considering i we look at $k = \min(i-1, n-i-m+2)$, set the $(2m-1)$-letter prefix of $s[i-k, i+k+m-2]$ as p and the $(2m-1)$-letter suffix of the same as q. We also take $\Delta = ones(s[i-k, i-1]) - ones(s[i, i+k-1])$ and $c = s[i, i+m-1]$.

Additionally, we maintain a bit mask b telling us for which of these m indices we have already found the longest abelian square (we initialize all bits of b to 0). Now we check the array B with proper values for each dimension (see Fig. 2), save all pairs stored in the corresponding field of B, replace b with the updated mask and decrease k by m. We continue this way until all bits of b are 1 or until $k < m$, in which case we search for the abelian squares at positions $i, \ldots, i+m-1$ in $O(m^2)$ time. Then we start considering $i+m$.

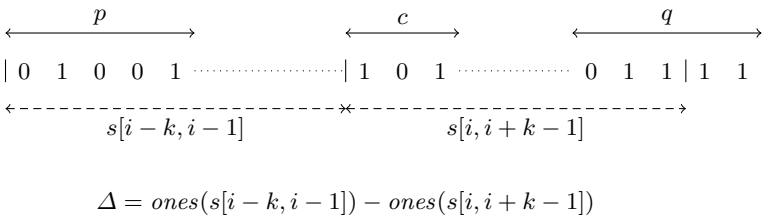

Fig. 2. How to choose p, q, c and Δ for given i, k and $m = 3$. If $\Delta = 1$ and $b = (0,0,1)$, then $B[\Delta][p][q][c][b] = ((1,1),(2,3))$.

This solution can be implemented in two nested loops, where both the inner and the outer loop execute $O(n/m)$ runs. This way the whole algorithm works in $O(n^2/m^2 + nm^2)$ time, after constructing array B in $O(m^3 64^m)$ time. Since the array can possibly store more than one value in each field, it uses $O(m^2 64^m)$ space.

This algorithm computes the lengths of the longest abelian squares with their centers in every index i. It can be easily modified to compute the lengths of the shortest abelian squares. Instead of starting with the largest possible k and decreasing it by m in every run of the inner loop, we may start with $k = 1$ and increase it by m in every run. The array B also needs to be modified to store information about the shortest abelian squares.

Theorem 6. *The longest (shortest) abelian squares for every center index i of s can be computed in $O(n^2/\log^2 n)$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \frac{\log n}{12} \right\rfloor$ the running time becomes:

$$\begin{aligned} O(n^2/m^2 + nm^2 + m^3 64^m) &= O(n^2/\log^2 n + n\log^2 n + 64^{\frac{\log n}{12}} \log^3 n) \\ &= O(n^2/\log^2 n + \sqrt{n}\log^3 n) \\ &= O(n^2/\log^2 n) \end{aligned}$$

The space complexity becomes:

$$\begin{aligned} O(n + m^2 64^m) &= O(n + 64^{\frac{\log n}{12}} \log^2 n) \\ &= O(n + \sqrt{n}\log^2 n) \\ &= O(n) \end{aligned}$$

□

3.3 Counting all abelian squares

We can modify the values stored in array B so that we can use them to count the number of all abelian squares in string s. In each field of B we may store the number of all abelian squares, when the values Δ, p, q and w are specified. This can be calculated in $O(m^2)$ time, which does not affect the time complexity (nor the space complexity).

If we are interested only in computing the number of all abelian squares, we may omit maintaining the bit mask b and cut one of the dimensions of array B. This has no effect on the time and space complexity.

One can also count the number of all abelian squares, which have their center in index i for every i. To make this possible, the array B can also store a list of length m, where the j-th element of the list tells us how many abelian squares we have found for the center $i+j-1$. Since we must process m operations, the time complexity increases to $O(n^2/\log n)$. We may also say that the time complexity is $O(n^2/\log^2 n + d)$, where d is the number of all abelian squares in string s, if we omit those indices, for which we have not found any abelian square.

Using the same approach we may also compute a list of all abelian squares in string s (i.e. pairs of centers and lengths for every abelian square). The time complexity remains $O(n^2/\log^2 n+d)$ and the space complexity becomes $O(n+d)$.

3.4 Abelian borders

It is known that a string s has an abelian border of length i if and only if it has an abelian border of length $n-i$ (see Lemma 4 in Matsuda et al. [14]). This way the longest and the shortest abelian borders are determined by each other. We will modify the algorithm for computing abelian squares of s so that it will compute the abelian border array of s.

We still will be analyzing m indices i, $i+1$, ..., $i+m-1$ at once, but this time they stand for m ends of consecutive prefixes $s[1,i]$, $s[1,i+1]$, ..., $s[1,i+m-1]$. We will focus on finding the shortest abelian borders for these prefixes. For given i we start with $k=1$. We set Δ to the difference of numbers of 1s in $s[1,k-1]$ and $s[i+m-k+1,i+m-1]$. We denote $(2m-1)$-letter prefix of $s[k,i+m-k]$, $p=s[k,k+2m-2]$ and $(2m-1)$-letter suffix of the same, $q=s[i-m-k+2,i+m-k]$. We fix $c=s[i,i+m-1]$ and maintain a bit mask b, where the j-th bit of b tells if we have already found the shortest abelian border of $s[1,i+j-1]$.

We will use a pre-computed auxiliary array C with the same number of dimensions and the same size of respective dimensions as array B. Given Δ, p, q, c and b the array C stores pairs (j,l) of shortest abelian borders that were found. Every field of this array can be computed in $O(m^2)$ time.

The algorithm executes two nested loops. The outer loop starts with $i=1$ and increases i by m after every run. The inner loop starts with $k=1$ and increases k by m after every run. This leads us to the following corollary.

Corollary 7. *The abelian border array of string s can be computed in $O(n^2/\log^2 n)$ time using $O(n)$ space.*

Similarly as in the case of computing a distribution of abelian squares by their centers, we can compute a representation of all abelian borders of prefixes of s in $O(n^2/\log n)$ time and $O(n)$ additional space.

4 Abelian periods

The problem of finding all abelian periods was solved in $O(n^2)$ time for any alphabet by Crochemore et al. [6]. Let us briefly recall their solution adapted to our binary alphabet case.

One can see that k is a regular abelian period of $s[i,n]$ if and only if all the conditions below hold:

- k is a regular abelian period of $s[i+k,n]$,
- if $n \geq i+2k-1$ then $s[i+k,i+2k-1] \approx s[i,i+k-1]$,
- if $n < i+2k-1$ then $s[i+k,n]$ is an abelian factor of $s[i,i+k-1]$.

Thereby (i, k) is a (general) abelian period of s if and only if k is a regular abelian period of $s[i, n]$ and $s[1, i-1]$ is an abelian factor of $s[i, i+k-1]$.

All general abelian periods of s are of the form (i, k), where $1 \leq i, k \leq n$, thus information about all of them can be represented as an array GP of size $n \times n$, where $GP[i][k] = 1$ if (i, k) is an abelian period of s and $GP[i][k] = 0$ otherwise. We can compute this array using dynamic programming. For this purpose we use an array RP of size $n \times n$, where $RP[i][k] = 1$ if k is a regular abelian period of $s[i, n]$ and $RP[i][k] = 0$ otherwise.

We will first show how to compute RP. We set $RP[i][k] = 1$ for every i and k such that $n \leq i + k - 1$. To compute $RP[i][k]$ we need to know $RP[i + k][k]$, so we will consider all remaining pairs (i, k) in a descending order of i. Checking if two strings are abelian equivalent or if one string is an abelian factor of the other can be done in constant time, so the whole array RP can be computed in $O(n^2)$ time and space.

To compute the array GP we look at each $RP[i][k] = 1$ such that $i \leq k$ and set $GP[i][k] = 1$ if and only if $s[1, i-1]$ is an abelian factor of $s[i, i+k-1]$.

The length of the shortest abelian period of s, the number of all of them and the number of all abelian periods of s can be easily extracted from the GP array.

4.1 Computing *RP* faster

For a better picture, let us assume that i numbers columns and k numbers rows of all arrays. Let us take $m < n$ and without loss of generality assume that m is a divisor of n (otherwise we increase n by at most $m-1$). We can code $O(\log n)$ fields of each of the arrays GP, RP into one machine word. In our case we will pack all fields on indices $\{i, i+1, \ldots, i+m-1\} \times \{k, k+1, \ldots, k+m-1\}$ (there are exactly m^2 of them), where $m|(i-1)$ and $m|(k-1)$, into unit square arrays of size $m \times m$. If m is small enough, we can code such a unit array into one machine word, row by row. This way both GP and RP are divided regularly into n^2/m^2 unit square arrays and stored in $O(n^2/m^2)$ space.

From now on we will still reference to a single field of GP and RP on indices (i, k) or a subarray of any of them, but we are assuming that the physical representation of arrays is as described. To improve time complexity of computing RP and GP we will fill out m fields of RP at a time and m fields of GP at a time.

To compute RP we will use three auxiliary arrays: A of size $(2m+1) \times 2^{m^2} \times 2^{m-1} \times 2^{2m-2}$, E of size $2^{m^2} \times 2^{m^2} \times m$ and F of size $2^{m^2} \times 2^m \times m$.

E is an array which for given two unit arrays X and Y and a shift i, stores a unit array Z such that the first $n-i+1$ columns of Z are the last $n-i+1$ columns of X and the last $i-1$ columns of Z are the first $i-1$ columns of Y.

F is an array which for given unit array X, a column (that is, a 1-d array) C and a shift i, stores a unit array Y which is X with the i-th column replaced by C.

Finally $A[\Delta][X][p][q] = C$ if and only if there exists a string w of length at least $2k + 2m - 2$ such that all of the conditions below hold:

- $p = w[k+1, k+m-1]$,
- $q = w[2k+1, 2k+2m-2]$,
- $\Delta = ones(w[1,k]) - ones(w[k+1, 2k])$,
- $k = lm + 1$ for some integer l,
- X is a unit array containing information about regular periods on suffixes of w starting on indices from $\{k+1, k+2, \ldots, k+m\}$ and having lengths from $\{k, k+1, \ldots, k+m-1\}$,
- C is a column containing information about regular periods of w with lengths from $\{k, k+1, \ldots, k+m-1\}$.

We will use this array, to instead of calculating one value $RP[i][k]$ at a time, calculate $RP[i][k, k+m-1]$ at once. For every Δ, X, p and q, there exists exactly one C. If $|\Delta| > m$, then $C = (0, 0, \ldots, 0)$ and we do not store this explicitly, otherwise we can simply pre-compute each field of A in $O(m)$ time.

To compute RP we will first set all $RP[i][k] = 1$, where $n \leq k + i - 1$, as previously. Noticing that there are only $O(n)$ unit arrays which we will fill with 1s only partially allows us to initialize RP in $O(n^2/m^2)$ time, instead of $O(n^2)$ time.

Now for the remaining pairs (i, k), we iterate first by k, starting with $k = n - m + 1$ and decreasing it by m. For a given k we iterate by i from n to 1, decreasing it by 1.

In each operation we take $w = s[i, n]$, $X = RP[i+k, i+k+m-1][k, k+m-1]$ and p, q and Δ respectively; see Fig. 3. X can be extracted from two particular unit arrays from the n^2/m^2 unit arrays that RP consist of, by using array E. After getting a new column C from array A, that corresponds to $RP[i][k, k+m-1]$, we can update array RP, using array F.

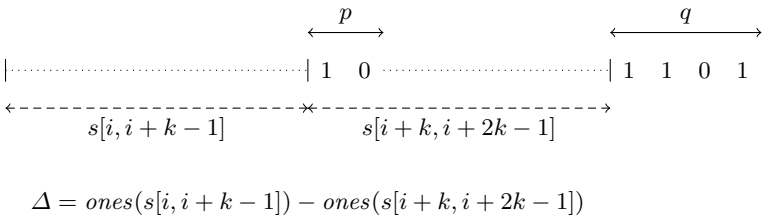

Fig. 3. How to choose p, q and Δ for given i, k and $m = 3$. If $\Delta = 0$ and $X = \begin{pmatrix} \mathbf{1} & 0 & 0 \\ 0 & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{1} \end{pmatrix}$, then $A[\Delta][X][p][q] = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$.

All the operations above are constant-time, thus the whole RP array can be computed in $O(n^2/m)$ time.

4.2 Computing GP faster

Notice that if $RP[i][k] = 0$, then $GP[i][k] = 0$. Thus the array GP is the array RP with some fields set to 0.

To compute the array GP we will first copy the array RP into it. Then we will update $GP[i][k] = 0$ for all $i > k$. As already mentioned, that can be done in $O(n^2/m^2)$ time.

The remaining fields that have to be set to 0 are determined by all i and k for which $s[1, i-1]$ is not an abelian factor of $s[i, i+k-1]$. To detect all such i and k, for each i we compute the minimal k_i, such that $s[1, i-1]$ is an abelian factor of $s[i, i+k_i-1]$. This can be done in linear time, since obviously $(i+1)+k_{i+1} \geq i+k_i$; see [6].

Having i and k_i we set $GP[i][k] = 0$ for all $k < k_i$. Since all such k form an interval, it can be done in $O(n/m)$ time for specific i and k_i using the array F and in $O(n^2/m)$ time for the whole array GP.

For now the arrays RP and GP use $O(n^2/m^2)$ space. If we are interested only in counting periods of every size and storing all shortest periods, it suffices to store only m consecutive rows of both arrays, i.e. rows $k, \ldots, k+m-1$ (and all k_i's). For this only $O(n/m)$ unit arrays and $O(n)$ additional space is necessary.

In conclusion, computing arrays A, E and F costs us $O(m^2 2^{m^2} 8^m + m^3 4^{m^2})$ time and $O(m 2^{m^2} 8^m + m 4^{m^2})$ space. Counting all abelian periods costs us additionally $O(n^2/m)$ time and $O(n)$ space.

Theorem 8. *All shortest abelian periods of string s can be computed in $O(n^2/\sqrt{\log n})$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \sqrt{\frac{\log n}{4}} \right\rfloor$ the running time becomes:

$$\begin{aligned} O(n^2/m + m^2 2^{m^2} 8^m + m^3 4^{m^2}) &= O(n^2/\sqrt{\log n} + 2^{\frac{\log n}{4}} 8^{\sqrt{(\log n)/4}} \log n \\ &\quad + 4^{\frac{\log n}{4}} \log^{\frac{3}{2}} n) \\ &= O(n^2/\sqrt{\log n} + n^{0.26} \log n + \sqrt{n} \log^{\frac{3}{2}} n) \\ &= O(n^2/\sqrt{\log n}) \end{aligned}$$

The space complexity becomes:

$$\begin{aligned} O(n + m 2^{m^2} 8^m + m 4^{m^2}) &= O(n + 2^{\frac{\log n}{4}} 8^{\sqrt{(\log n)/4}} \sqrt{\log n} + 4^{\frac{\log n}{4}} \sqrt{\log n}) \\ &= O(n + n^{0.26} \sqrt{\log n} + \sqrt{n \log n}) \\ &= O(n) \end{aligned}$$

□

5 Abelian covers

For given $i \leq n$ we can check if the prefix of length i of string s is an abelian cover of s. This is actually the abelian pattern matching problem, which for a constant-sized alphabet can obviously be solved in linear time.

Let us assume that we have found all occurrences of $s[1,i]$ in s and stored their (sorted) positions in a sequence I. Then $s[1,i]$ is an abelian cover of s if and only if the first element of I is 1, the last element of I is $n-i+1$ and the differences between all pairs of consecutive elements of I are no greater than i.

Checking all $1 \leq i \leq n$, the shortest abelian cover of string s can be found in $O(n^2)$ time.

5.1 Optimizing running time

Assume without loss of generality that m is a divisor of n. Instead of finding one occurrence of $s[1,i]$ in s at a time, we will only find the smallest and the largest positions from each of intervals $[1,m]$, $[m+1,2m]$, ..., $[n-m+1,n]$ such that $s[1,i]$ is an abelian match on s at these positions.

For this purpose we will use a pre-computed auxiliary array A of size $2^{m-1} \times (2m+1) \times 2^{m-1}$. $A[p][\Delta][q] = (l_1, l_2)$ if and only if there are strings w of length $i+m-1$ (text) and c of length i (potential cover) such that all of the conditions below hold:

- p is a $(m-1)$-letter prefix of w,
- q is a $(m-1)$-letter suffix of w,
- $\Delta = ones(w[1,i]) - ones(c)$,
- $1 \leq l_1 \leq l_2 \leq m$,
- $-ones(p[1,l_1-1]) + ones(q[1,l_1-1]) = \Delta$, i.e. $c \approx w[l_1, l_1+i-1]$,
- $-ones(p[1,l_2-1]) + ones(p[1,l_2-1]) = \Delta$, i.e. $c \approx w[l_2, l_2+i-1]$,
- l_1 is the smallest possible,
- l_2 is the largest possible.

If no such pair exists then $A[p][\Delta][q] = (-1,-1)$. Each element of array A can be pre-computed in $O(m)$ time. We consider only $-m \leq \Delta \leq m$, so the whole array can be then pre-computed in $O(m^2 4^m)$ time and uses $O(m4^m)$ space.

Our algorithm for finding the shortest abelian cover works as follows. First we check if any prefix of s of length less than $2m$ is a cover of s. This can be done by the straightforward solution in $O(nm)$ time. Now we assume that the shortest cover has length at least $2m$.

We iterate with i from $2m$ to n to check if $s[1,i]$ is an abelian cover of s. For every i we start with $I = ()$. Now we iterate with j from $j = 1$ to $n-i+1$, increasing j by m each time. We are checking if there is an abelian match on positions $[j, j+m-1]$, so we are implicitly considering substring $s[j, j+i+m-2]$.

For every i and j we denote three values:

- the $(m-1)$-letter prefix of $s[j, j+i+m-2]$, $p = s[j, j+m-2]$,
- the $(m-1)$-letter suffix of $s[j, j+i+m-2]$, $q = s[j+i, j+i+m-2]$ and
- the difference of 1s, $\Delta = ones(s[j, j+i-1]) - ones(s[1,i])$.

If $|\Delta| > m$ then $s[1,i]$ does not abelian match any substring in $s[j, j+i+m-2]$. Otherwise we can find the first and the last occurrence (in the considered interval) by referencing to the array A and, if they exist, add them to I.

After iterating with j, I contains $O(n/m)$ elements. We can then check if $s[1,i]$ is an abelian cover of s the same way, as we did it in the straightforward solution. This way the whole algorithm works in $O(nm + n^2/m)$ time.

Theorem 9. *The shortest abelian cover for string s can be computed in $O(n^2/\log n)$ time using $O(n)$ space.*

Proof. By choosing $m = \left\lfloor \frac{\log n}{4} \right\rfloor$ the running time becomes:

$$\begin{aligned} O(nm + n^2/m + m^2 4^m) &= O(n \log n + n^2/\log n + 4^{\frac{\log n}{4}} \log^2 n) \\ &= O(n^2/\log n + \sqrt{n} \log^2 n) \\ &= O(n^2/\log n) \end{aligned}$$

The space complexity becomes:

$$\begin{aligned} O(n + m4^m) &= O(n + 4^{\frac{\log n}{4}} \log n) \\ &= O(n + \sqrt{n} \log n) \\ &= O(n) \end{aligned}$$

□

It is clear that we can use the same algorithm to compute lengths of all abelian covers of s. If we are interested in detecting all occurrences of all abelian covers of s, then the array A needs to store not only the pair of the smallest l_1 and the largest l_2 that hold all mentioned conditions, but all such l. This has no effect on space complexity, but the time complexity becomes $O(n^2/\log n + d)$, where d is the number of all occurrences of all abelian covers in s.

The shortest abelian cover is probably of most interest. We can use our algorithm to find it and then we can find all its occurrences in s in $O(n)$ time, using abelian pattern matching in linear time.

6 The case of a larger alphabet

We have presented subquadratic-time algorithms for computing abelian squares, general abelian periods, abelian covers and abelian border array for a string over a binary alphabet. In the fundamental problem of abelian stringology—jumbled indexing—switching from binary to a larger constant-sized alphabet increases the hardness of the problem considerably [1,12]. However, all our algorithms can be applied to the case of any constant-sized alphabet, preserving both time and space complexity.

For this, in each of the presented auxiliary arrays, instead of considering Δ, that is, the difference in the number of 1s between two substrings, we may consider a difference of Parikh vectors of the corresponding substrings. The number of possible differences we should consider now is obviously no greater than $(2m+1)^\sigma$, where σ is the size of the alphabet. This increases the size of the

auxiliary arrays and the computation time by at most a factor of $O(\log^{\sigma} n)$. These sizes and construction times were always $o(n^{1-\epsilon})$ for some $\epsilon > 0$. Hence, they will remain sublinear for a constant σ.

Instead of using the function *ones*, we may use an analogous constant-time function that returns a Parikh vector for a substring. In all of the algorithms we assume that a substring of length m can be stored using one machine word. For this to be still true for $\sigma > 2$, we need to divide the proposed values of m in Theorems 6 and 9 by a factor of $\log \sigma$ and in Theorem 8 by a factor of $\sqrt{\log \sigma}$.

7 Conclusions

We have presented the first subquadratic-time algorithms for computing abelian squares, general abelian periods, abelian covers and abelian border array for a string over a constant-sized alphabet. An open question is if there exist $O(n^2/\log^c n)$-time algorithms for the problems considered in this paper with a larger constant c. A further question, for all of the problems, is if there exists an $O(n^{2-\epsilon})$-time algorithm or, possibly, if there is a lower bound on the time complexity of an algorithm solving this problem. In comparison, for the seminal problem in this area, the jumbled indexing, on one hand, $O(n^{2-\epsilon})$-time algorithms are known for any constant-sized alphabet [3], but on the other hand, for sufficiently large alphabets $\sigma > 2$ conditional lower bounds are known for the query vs construction time trade-off [1] based on hardness of the 3SUM problem.

References

1. A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein. On hardness of jumbled indexing. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 114–125. Springer, 2014.
2. P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták. On table arrangements, scrabble freaks, and jumbled pattern matching. In P. Boldi and L. Gargano, editors, *Fun with Algorithms, 5th International Conference, FUN 2010, Ischia, Italy, June 2-4, 2010. Proceedings*, volume 6099 of *Lecture Notes in Computer Science*, pages 89–101. Springer, 2010.
3. T. M. Chan and M. Lewenstein. Clustered integer 3SUM via additive combinatorics. In R. A. Servedio and R. Rubinfeld, editors, *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 31–40. ACM, 2015.
4. F. Cicalese, G. Fici, and Z. Lipták. Searching for jumbled patterns in strings. In J. Holub and J. Zdárek, editors, *Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009*, pages 105–117. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2009.
5. S. Constantinescu and L. Ilie. Fine and Wilf's theorem for Abelian periods. *Bulletin of the EATCS*, 89:167–170, 2006.

6. M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, J. Pachocki, J. Radoszewski, W. Rytter, W. Tyczyński, and T. Waleń. A note on efficient computation of all abelian periods in a string. *Inf. Process. Lett.*, 113(3):74–77, 2013.
7. L. J. Cummings and W. F. Smyth. Weak repetitions in strings. *J. Combinatorial Math. and Combinatorial Computing*, 24:33–48, 1997.
8. P. Erdös. Some unsolved problems. *Hungarian Academy of Sciences Mat. Kutató Intézet Közl.*, 6:221–254, 1961.
9. G. Fici, T. Lecroq, A. Lefebvre, and É. Prieur-Gaston. Computing Abelian periods in words. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2011*, pages 184–196, Czech Technical University in Prague, Czech Republic, 2011.
10. G. Fici, T. Lecroq, A. Lefebvre, E. Prieur-Gaston, and W. Smyth. Quasi-linear time computation of the Abelian periods of a word. In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2012*, pages 103–110, Czech Technical University in Prague, Czech Republic, 2012.
11. D. Hermelin, G. M. Landau, Y. Rabinovich, and O. Weimann. Binary jumbled pattern matching via all-pairs shortest paths. *CoRR*, abs/1401.2065, 2014.
12. T. Kociumaka, J. Radoszewski, and W. Rytter. Efficient indexes for jumbled pattern matching with constant-sized alphabet. In H. L. Bodlaender and G. F. Italiano, editors, *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*, pages 625–636. Springer, 2013.
13. T. Kociumaka, J. Radoszewski, and W. Rytter. Fast algorithms for abelian periods in words and greatest common divisor queries. In N. Portier and T. Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*, pages 245–256. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
14. S. Matsuda, S. Inenaga, H. Bannai, and M. Takeda. Computing abelian covers and abelian runs. In J. Holub and J. Zdárek, editors, *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*, pages 43–51. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014.
15. T. M. Moosa and M. S. Rahman. Indexing permutations for binary strings. *Inf. Process. Lett.*, 110(18-19):795–798, 2010.
16. T. M. Moosa and M. S. Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discrete Algorithms*, 10:5–9, 2012.