k-Abelian Pattern Matching: Revisited, Corrected, and Extended

Golnaz Badkobeh[1], Hideo Bannai[2⋆], Maxime Crochemore[3], Tomohiro I[4⋆⋆], Shunsuke Inenaga[2⋆⋆⋆], and Shiho Sugimoto[5]

[1] Department of Computing, Goldsmiths University of London, London, UK
g.badkobeh@gold.ac.uk
[2] Department of Informatics, Kyushu University, Fukuoka, Japan
{bannai,inenaga}@inf.kyushu-u.ac.jp
[3] Department of Informatics, King's College London, London, UK
maxime.crochemore@kcl.ac.uk
[4] Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka, Japan
tomohiro@ai.kyutech.ac.jp
[5] Security Research Laboratories, NEC, Kawasaki, Japan
s-sugimoto@ik.jp.nec.com

Abstract. Two strings of equal length are called k-Abelian equivalent, if they share the same multi-set of factors of length at most k. Ehlers et al. [JDA, 2015] considered the *k-Abelian pattern matching problem*, where the task is to find all factors in a text T that are k-Abelian equivalent to a pattern P. They claimed a number of algorithmic results for the off-line and on-line versions of the k-Abelian pattern matching problem. In this paper, we first argue that some of the claimed results by Ehlers et al. [JDA, 2015] contain major errors, and then we present a new algorithm that correctly solves the offline version of the problem within the same bounds claimed by Ehlers et al., in $O(n + m)$ time and $O(m)$ space, where $n = |T|$ and $m = |P|$. We also show how to correct errors in their online algorithm, and errors in their real-time algorithms for a slightly different problem called the *extended* k-Abelian pattern matching problem.

Keywords: Abelian equivalence, pattern matching, suffix trees, suffix arrays

1 Introduction

Two strings X and Y of equal length are said to be *Abelian equivalent* if the numbers of occurrences of each letter are equal in X and Y. For instance, strings `ababaac` and `caaabba` are Abelian equivalent. Since the seminal paper by Erdős [14] published in 1961, the study of Abelian equivalence on strings has attracted much attention, both in word combinatorics and string algorithmics. One good example is the Abelian version of pattern matching problem, where the task is to locate all factors of a given text T that are Abelian equivalent to a given pattern P (reporting version), or to test whether there is such a factor in T (existence version). This problem is called the *jumbled pattern matching problem*, and a number of algorithms have been proposed for this problem; see the subsection for related work below.

k-Abelian equivalence is a natural generalization of Abelian equivalence: For a positive integer k, two strings X and Y of equal length are said to be k-Abelian equivalent if the numbers of occurrences of each string of length *at most* k are equal in X and Y. The notion of k-Abelian equivalence of strings was first introduced

⋆ Supported by JSPS KAKENHI Grant Number JP16H02783
⋆⋆ Supported by JSPS KAKENHI Grant Number JP19K20213
⋆⋆⋆ Supported by JSPS KAKENHI Grant Number JP17H01697

Golnaz Badkobeh, Hideo Bannai, Maxime Crochemore, Tomohiro I, Shunsuke Inenaga, Shiho Sugimoto: k-Abelian Pattern Matching: Revisited, Corrected, and Extended, pp. 29–40.
Proceedings of PSC 2019, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06618-8

by Huova et al. [20], and then has extensively been studied in the context of word combinatorics such as k-Abelian repetitions [20,19,21], k-Abelian periodicities [23], k-Abelian equivalence classes [22,10,25], just to mention a few.

The first (and only, to our knowledge) algorithmic results concerning k-Abelian equivalence of strings were given by Ehlers et al. [13] for the *k-Abelian pattern matching problem.* Here the task is, given a text T and pattern P, to locate all factors of T that are k-Abelian equivalent to P. Ehlers et al. [13] considered the offline and online versions of the k-Abelian pattern matching problem, and claimed a number of results with different bounds.

In this paper, we first argue that some of those claimed results by Ehlers et al. [13] contain major errors. These errors are due to the abuse of the van Emde Boas data structure [7] that uses space linear in the size of the *universe* of the integers, no matter how many elements are stored in the data structure. There are also other major issues such as carelessness on the size of the integer alphabet from which the text is drawn, unknown construction time of the weighted ancestor data structure on suffix trees [17], and so on. We then present a new algorithm that actually solves the offline version of the problem within the same bounds claimed by Ehlers et al., namely in $O(n + m)$ time and $O(m)$ space, where $n = |T|$ and $m = |P|$.

Ehlers et al. [13] also considered a slightly different variant of the k-Abelian pattern matching problem called the *extended k-Abelian pattern matching problem.* Here, two strings X and Y of equal length are said to be extended k-Abelian equivalent if the numbers of occurrences of each string of length *exactly* k are equal in X and Y. We point out the major errors in their solutions to extended k-Abelian pattern matching, and show how to obtain alternative solutions.

Related work

For jumbled pattern matching (i.e. 1-Abelian pattern matching), there is a simple algorithm that compares the number of occurrences of all letters $a \in \Sigma$ of the pattern P and a sliding window of length m over the text T. In case P is over an integer alphabet of size linear in m, shifting the window takes $O(1)$ time per text letter and hence this algorithm runs in $O(n + m)$ time and $O(m)$ working space, where n and m are the lengths of T and P, respectively. Butman et al. [9] considered how to solve this problem on run-length encoded strings. When n' and m' are respectively the sizes of the run-length encoded text and pattern, then their algorithm runs in $O(n' + m')$ time with $O(m')$ working space, given that the pattern is over an integer alphabet of size linear in m'. The essentially same algorithm was later rediscovered by Sugimoto et al. [35] and was used as a sub-routine in their algorithms to compute Abelian regularities from run-length encoded strings.

The *indexing* version of the jumbled pattern matching is more challenging and has attracted much attention. Amir et al. [1] proposed an indexing structure of $O(n^{1+\epsilon})$ space that can be constructed in $O(n^{1+\epsilon} \log \sigma)$ time and can decide whether there is an occurrence of the pattern in $O(m^{\frac{1}{\epsilon}} + \log \sigma)$ time, where σ is the alphabet size and $0 < \epsilon < 1$ is any constant. Their algorithm works for any alphabets. For any constant-size alphabets, Kociumaka et al. [29] proposed an $O(n^2/L)$-space data structure which can be constructed in $O(n^2(\log \log n)^2)/\log n)$ time and can report the left-most occurrence in $O(L^{2\sigma-1})$ time, where n is the length of a given text t and L is a trade-off parameter ranging from 1 to n. For alphabets of size $\sigma = \omega(1)$, Amir et al. [2] showed that jumbled indexing requires $\Omega(n^{2-\lambda})$ preprocessing time or $\Omega(n^{1-\delta})$ query time for every $\lambda, \delta > 0$, under the famous 3SUM-hardness assumption.

There have been several indexing structures for *binary* jumbled pattern matching (BJPM). Cicalese et al. [12] gave an index for BJPM that uses $O(n)$ space, can be constructed in $O(n^2)$ time and can decide the existence of occurrences in $O(1)$ time. Improved $O(n^2/\log n)$-time construction of the BJPM indexing was independently proposed by Burcsi et al. [8] and by Moosa and Rahman [31]. Later, construction time was further improved to $O(n^2/(\log n)^2)$ by Moosa and Rahman [32]. Hermelin et al. [18] showed how to reduce the BJPM problem to the all-pairs shortest paths problem and presented an $O(n^2/2^{\Omega(\log n/\log\log n)^{0.5}})$ preprocessing scheme for the BJPM problem. Chan and Lewenstein [11] presented a breakthrough solution for the BJPM problem that takes $O(n^{1.864})$ time for preprocessing and requires $O(1)$ time for queries. Their solution can also be extended to larger constant-size alphabets, with strongly sub-quadratic preprocessing time and strongly sub-linear query time.

2 Preliminaries

Let Σ be an ordered alphabet of size σ. An element of Σ^* is called a *string*. Let ε denote the empty string of length 0. For a non-negative integer k, let Σ^k denote the set of strings of length k. For a string $u = xyz$, x, y, and z are called a *prefix*, *factor*, and *suffix* of u, respectively. For a string u of length n, let $u[i]$ denote the ith letter in u for $1 \le i \le n$, and $u[i..j]$ denote the factor of u that begins at position i and ends at position j for $1 \le i \le j \le n$. For a non-negative integer k, a factor of length k in a string u is called a *k-gram* in u. For a positive integer n, let $[1..n]$ denote the set of n positive integers from 1 to n.

For any string $u \in \Sigma^*$ and letter $a \in \Sigma$, $|u|_a$ denotes the number of occurrences of a in u. Two strings u and v are said to be *Abelian equivalent* if $|u|_a = |v|_a$ for all letters $a \in \Sigma$. To simplify the argument, let us identify each letter $a \in \Sigma$ with its lexicographical rank in Σ.

Now, we extend the aforementioned notion from occurrences of letters to those of strings. Namely, for a string t, let $|u|_t$ denote the number of occurrences of t in u.

Definition 1 (k-Abelian equivalence). *For a positive integer k, two strings u and v of equal length n are said to be* k-Abelian equivalent *if either*

(1) $u = v$ or
(2) all the following conditions hold:
- *(a) $|u|, |v| \ge k$;*
- *(b) $|u|_t = |v|_t$ for all strings $t \in \Sigma^k$;*
- *(c) $u[1..k-1] = v[1..k-1]$;*
- *(d) $u[n-k+2..n] = v[n-k+2..n]$.*

According to [24], the last condition (2)-(d) for having the same suffix of length $k-1$ can actually be dropped.

We denote $u \equiv_k v$ when u and v are k-Abelian equivalent. It is known that $u \equiv_k v$ iff $|u|_s = |v|_s$ for every string s of length at most k.

Example 1. Let $x = abaababbaab$ and $y = abbaabaabab$. For $k = 3$, x and y are k-Abelian equivalent, since they satisfy $|x| = |y| = 11 \ge 3$, $|x|_t = |y|_t$ for all strings $t \in \Sigma^3$ i.e. $|x|_{aaa} = |y|_{aaa} = 0, |x|_{aab} = |y|_{aab} = 2, |x|_{aba} = |y|_{aba} = 2, |x|_{abb} = |y|_{abb} = 1, |x|_{baa} = |y|_{baa} = 2, |x|_{bab} = |y|_{bab} = 1, |x|_{bba} = |y|_{bba} = 1, |x|_{bbb} = |y|_{bbb} = 0$, and their prefixes of length $k-1 = 2$ are equal i.e. $x[1..2] = y[1..2] = ab$.

In this paper, we consider the following problem.

Problem 1. Given a text T and a pattern P over an alphabet Σ and a positive integer k, locate all factors of T that are k-Abelian equivalent to P.

For simplicity, suppose that a string u terminates with a special letter that does not appear elsewhere in u. The *suffix tree* of string u of length n is a rooted edge-labeled tree such that (1) each internal node is branching, (2) each edge is labeled with a non-empty substring of u, (3) the labels of out-going edges of each node are mutually distinct, and (4) there is a one-to-one correspondence between suffixes of u and the leaves of the tree. The *locus* of a substring x of u in the suffix tree of u is the ending position of the path that spells out x from the root. When there is a node such that the path from the root to this node spells out x, then the locus of x is on that node. When there is no such node, then the locus of x is on an edge.

The *suffix array* of string u of length n, denoted SA_u, is an array of length n such that, for $1 \leq i \leq n$, $\mathsf{SA}_u[i] = j$ iff $u[j..n]$ is the ith lexicographically smallest suffix of u. SA_u can be seen as an array of the leaves of the suffix tree for u where the out-going edges are sorted in lexicographical order. The *LCP* array of string u, denoted LCP_u, is an array of length n such that $\mathsf{LCP}_u[1] = -1$ and, for $2 \leq r \leq n$, $\mathsf{LCP}_u[r]$ stores the length of the longest common prefix of the suffixes stored at positions $r-1$ and r in the suffix array i.e., $u[\mathsf{SA}_u[r-1]..n]$ and $u[\mathsf{SA}_u[r]..n]$.

3 Online and offline k-Abelian pattern matching

In this section, we point out some errors in the claims from the previous work of Ehlers et al. [13]. They considered Problem 1 in two settings, the *offline version* where the whole text and pattern are given together as input, and the *online version* where the pattern is given first to preprocess and the text letters are given in an online manner, one by one from left to right. In the online version, each time a new text letter arrives, a new k-Abelian equivalent occurrence of the pattern in the text must be reported (if it exists).

3.1 Offline k-Abelian pattern matching problem

In this subsection, we consider the offline version of Problem 1.

Errors in the previous work. Ehlers et al. [13] stated the following claim.

Claim (Remark 2 of [13] in conjunction with Theorem 2 of [13]). The offline version of Problem 1 can be solved in $O(n+m)$ time and $O(m)$ space for integer alphabet $\Sigma = [1..n]$[1].

Below, we show that Ehlers et al.'s approach [13] does not fulfill the above claim and uses more space than $O(m)$. To see why, let us briefly describe their approach from Theorem 2 of their paper [13]. For a string $u \in \Sigma^n$, consider the k-encoded string $\#(u,k)$ of length $n-k+1$ such that for each i ($1 \leq i \leq n-k+1$), $\#(u,k)[i]$ stores the lexicographical rank of the k-gram $u[i..i+k-1]$ in the set $\{u[i..i+k-1] \mid 1 \leq i \leq n-k+1\}$ of all k-grams in T. Given a text T and pattern P, they consider a concatenated string $w = T\$P$ where \$ is a special letter not appearing in T or P,

[1] This assumption of the integer alphabet is given in Section 2 (Preliminaries) in [13].

and compute $\#(w,k)$. Let $T' = w[1..n-k+1]$ and $P' = w[n+1..n+m-k+2] = P[1..m-k+1]$. Then, a key observation is that for each i $(1 \le i \le n-m+1)$, $T[i..i+m-1]$ and P are k-Abelian equivalent iff $T'[i..i+m-k]$ and P' are Abelian equivalent, and $T[i..i+k-2] = P[1..k-1]$. To compute $\#(w,k)$ and to test whether $T'[i..i+m-k]$ and P' are Abelian equivalent or not, they build the suffix array SA_w for the concatenated string w together with the LCP array LCP_w enhanced with a constant-time range minimum query data structure [5].

While the aforementioned approach works in $O(n+m)$ time for the integer alphabet $\Sigma = [1..n]$, it also requires $O(n+m)$ space. As an attempt to reduce the space requirement to $O(m)$, they chose the following approach: For ease of explanation, suppose that n is divisible by m. For each $0 \le t \le \frac{n}{m} - 2$, they pick the factor $T[tm{+}1..(t{+}2)m]$ of length $2m$, and built the suffix array of $w_t = T[tm{+}1..(t{+}2)m]\$P$ and apply the above method to w_t, namely construct the suffix array SA_{w_t} and LCP array LCP_{w_t} for each t.

However, this approach indeed takes $O(n+m)$ time and $O(n)$ space for *each* t. This is because, regardless of its length, any factor of the text T over the integer alphabet $[1..n]$ can contain letters (i.e. integers) up to n. In other words, any factor of such text T is still a string over the integer alphabet $[1..n]$. The above argument implies that the *universe* of the letters in $T[tm{+}1..(t{+}2)m]$ is $[1..n]$ in the worst case. Recall that *any* existing linear-time suffix array construction algorithms for the integer alphabet $[1..n]$ use bucket sort [28,27,26,33,3], and that any suffix array construction with comparison-based sorting must take $\Omega(n \log n)$ time for any ordered alphabet of size $O(n)$ [15]. In general, bucket sort for a set of s integers over the integer universe $[1..u]$ requires $O(s+u)$ time and $O(u)$ space, since it uses an integer array of length u. Therefore, Ehlers et al.'s method (Remark 2 of [13]) must use $O(n+m)$ time and $O(n)$ space for *each* t. Moreover, this leads to $O(n(n+m)/m)$ total time for all t's, which is super-linear in reasonably common cases where $m = o(n)$.

New offline algorithm. Now we present a new algorithm for the offline version of Problem 1 that indeed uses only $O(m)$ space. To achieve this goal, we introduce a reasonable assumption that the pattern P of length m is over an integer alphabet of size $[1..cm]$ with any positive constant c such that cm is a positive integer. Then we show the following:

Theorem 1. *Let P be a pattern of length m over an integer alphabet $[1..cm]$ with any positive constant c, and T be a text of length n over an arbitrary integer alphabet. Then, for a given integer $k > 0$, we can solve Problem 1 in $O(n+m)$ time using $O(m)$ space.*

Proof. Our proposed algorithm uses suffix trees. Namely, for each t $(0 \le t \le \frac{n}{m} - 2)$, we construct the suffix tree of $w_t = T[tm+1..(t+2)m]\$P$. For each occurrence of a k-gram in P, we construct a bucket that is associated to the locus of the k-gram in the suffix tree. The locus is an implicit or explicit node of string depth k. If a k-gram occurs z times in P, then there will be z buckets in its corresponding locus. Initially, all the buckets are empty.

Now, we check whether each factor of T of length m fulfill all the buckets. For each $x = 0, \ldots, t-1$ in increasing order, we map the factor $T[tm+1+x..tm+k+x]$ the (implicit or explicit) node of string depth k representing $T[tm{+}1{+}x..tm{+}k{+}x]$, and if there is a bucket there, we fulfill it with position $tm{+}1{+}x$. This can easily be done in $O(1)$ time per x after an $O(m)$-time preprocessing – for every leaf in the suffix tree, we can compute its ancestor of string depth k in $O(m)$ total time with a standard tree traversal. We keep track of a sliding window of length m over $T[tm+1..(t+2)m]$, and the positions in the buckets are removed as soon as they are out of the window. This can easily be done by implementing the set of buckets in each node by a queue. Each time all the buckets are fulfilled, then we additionally check if the $(k-1)$-gram of the text beginning with the current smallest position j in the buckets satisfies Condition (2)-(c) of Definition 1 with $P[1..k-1]$. This additional step can easily be done in $O(1)$ time by marking the locus of the suffix tree representing $P[1..k-1]$. If the condition is satisfied, then we output the beginning position j of the text factor that is k-Abelian equivalent to P. Then, we delete the position j from the corresponding bucket, and proceed to the next position by increasing x.

What remains is how to reduce the alphabet size of T. For this sake we replace *any* letter in T that exceeds cm with $cm+1$, where cm is the largest letter appearing in P. The resulting new text $\hat{T}$ is now a string of length n over the integer alphabet $[1..cm+1]$. For each t, the suffix tree of $\hat{T}[tm+1..(t+2)m]\$P$ can be constructed in $O(m)$ time and space, by the suffix tree construction algorithm for integer alphabets [15], or via any linear-time suffix array construction algorithm for integer alphabets and the LCP array. Note that all k-Abelian equivalent occurrences of P in the text are preserved in the new text $\hat{T}$. Thus, our algorithm runs in $O(n+m)$ time with $O(m)$ space. ☐

3.2 Online *k*-Abelian pattern matching problem

In this subsection, we consider the online version of Problem 1. In this variant of the problem, the authors assume that $\Sigma = [1..\sigma]$ with $\sigma \in O(m)$ [13][2].

The key idea of their algorithm is to use the following list L: Let $D_{k-1}(P)$ and $D_k(P)$ be the set of $(k-1)$-grams and k-grams that occur in P, respectively. Let f_1 be an array of length $|D_{k-1}(P)|$ such that $f_1[i]$ stores an occurrence of the lexicographically ith $(k-1)$-gram in $D_{k-1}(P)$. Similarly, let f_2 be an array of length $|D_k(P)|$ such that $f_2[j]$ stores an occurrence of the lexicographically jth k-gram in $D_k(P)$. Now the list L is defined as follows.

$$L = \{(i,a,j) \mid 1 \le i \le |D_{k-1}(P)|, 1 \le j \le |D_k(P)|, a \in \Sigma, f_1[i]a = f_2[j]\}.$$

While the original online algorithm by Ehlers et al. uses the suffix array and lcp array for P to implement L, in our explanation we use the suffix tree for P since it seems more intuitive and easier to follow[3]. Also, recall our offline algorithm of Theorem 1 as the method to follow can be seen as its online version. Let $\mathsf{STree}(P)$ denote the suffix tree for P.

Now one can regard L as the set of the edges of $\mathsf{STree}(P)$ that connect the (implicit or explicit) nodes of string depth $k-1$ to the nodes of string depth k. Now the basic strategy is the following. Let T' be the current text, a the next letter to be appended to T', and $T = T'a$. Suppose that we know the locus in $\mathsf{STree}(P)$ that represents the

[2] Ehlers et al. deal with the offline version of the problem in Section 3 and the online version in Section 4 in their paper [13]. While they write "As before, we assume that $\Sigma = [1..\sigma]$ with $\sigma \in O(m)$." in the beginning of Section 4, we cannot find such assumption in Section 3 or earlier in their paper.

[3] This variant of algorithm with suffix trees is also used by Ehlers et al. for online extended k-Abelian pattern matching (Section 5 of [13]).

suffix of T' of length $(k-1)$, which is the rightmost $(k-1)$-gram in T' (if it exists in the tree). Then, the task is to quickly find the out-going edge labeled a from this locus, since there we can find the locus of the suffix of T of length k in $\mathsf{STree}(P)$. This is a classical problem of implementing the set of out-going edges of a node of labeled trees, and a number of data structures can be used for this purpose. Ehlers et al. stated the following claim:

Claim (The 4th bound of Theorem 4 of [13]). Given a static pattern $P \in \Sigma^m$ over the integer alphabet $[1..\sigma]$, and a positive integer k, the online version of Problem 1 can be solved in $O(m)$ preprocessing time, $O(m)$ working space, and $O(\log\log\sigma)$ time per text letter.

The idea of the above claim is to use the van Emde Boas data structure [7]. However, it is well known that for an integer universe $U = [1..u]$ of size u, the van Emde Boas data structure of a set $S \subseteq U$ requires $\Theta(u)$ space regardless of the cardinality of S. In the above context, $u = \sigma$ since the universe here is the integer alphabet $[1..\sigma]$. This implies that this approach by Ehlers et al.'s must use $O(\sigma m)$ space, since there can be $O(m)$ nodes of string depth $k-1$ in the suffix tree. Thus the above claim does not hold.

Indeed, Ehlers et al. also proposed a simple array-based implementation of the branching edges (the 1st variant of Theorem 4 of [13]). When one can afford to using $O(\sigma m)$ space, then this simple array-based approach is faster since each edge can be accessed in $O(1)$ time. By the way, the 1st variant of Theorem 4 of [13] states that their preprocessing requires only $O(m)$ time. This is not the case, since this variant must use $O(\sigma m)$ time to construct all the arrays.

4 Extended k-Abelian pattern matching

Ehlers et al. also considered a slightly different notion of k-Abelian equivalence, called *extended k-Abelian equivalence.*

Definition 2 (extended k-Abelian equivalence). *For a positive integer k, two strings u and v of equal length said to be* extended k-Abelian equivalent *if their multi-sets of factors of length k coincide, i.e., both of the last two conditions (2)-(c) and (2)-(d) of having the same prefixes and suffixes are dropped from Definition 1.*

Example 2. Let $x = abaababbaab$ and $y = baabaabbaba$. For $k = 3$, x and y are not k-Abelian equivalent but extended k-Abelian equivalent, since they satisfy $|x| = |y| = 11 \geq 3$, $|x|_t = |y|_t$ for all strings $t \in \Sigma^3$ i.e. $|x|_{aaa} = |y|_{aaa} = 0, |x|_{aab} = |y|_{aab} = 2, |x|_{aba} = |y|_{aba} = 2, |x|_{abb} = |y|_{abb} = 1, |x|_{baa} = |y|_{baa} = 2, |x|_{bab} = |y|_{bab} = 1, |x|_{bba} = |y|_{bba} = 1, |x|_{bbb} = |y|_{bbb} = 0$, but their prefixes of length $k-1 = 2$ are not equal i.e. $x[1..2] = ab \neq ba = y[1..2]$.

Problem 2. Given a text T and a pattern P over an alphabet Σ and a positive integer k, locate all factors of T that are extended k-Abelian equivalent to P.

4.1 Errors in the previous work

They considered the online version of Problem 2 and claimed the following *real-time* bounds. Here, an online algorithm is called real-time if an $O(1)$ worst-case time is guaranteed per text symbol.

Claim (Theorem 6 of [13]). Given a static pattern $P \in \Sigma^m$ over the integer alphabet $[1..\sigma]$, and a positive integer k, the online version of Problem 2 can be solved in:

- $O(m \log k)$ preprocessing time, $O(\sigma m)$ working space, and $O(1)$ worst-case time per text letter;
- $O(m(\log\log m + \log k))$ preprocessing time, $O(m)$ working space, and $O(1)$ worst-case time per text letter;
- $O(m \log k)$ expected preprocessing time, $O(m)$ working space, and $O(1)$ worst-case time per text letter;
- $O(m \log k)$ preprocessing time, $O(m)$ working space, and $O(\log\log\sigma)$ worst-case time per text letter.

We note that the 4th variant is based on the same flawed argument with the van Emde Boas data structure as in Section 3.2, and thus it indeed requires $O(\sigma m)$ space. Hence the 1st variant is better, and we will ignore the 4th variant in the sequel. Also, since the 1st variant uses $O(\sigma m)$ space for preprocessing, there should be an additive σm term in the preprocessing time.

The $m \log k$ term that are common in the preprocessing time of the above claim comes from the next statement from [13]:

Claim (Lemma 5 of [13]). One can preprocess pattern P of length m in $O(m \log k)$ time and linear space such that, for each i and j with $j - i \leq k$, one can return in constant time the (explicit or implicit) node of $\mathsf{STree}(P)$ that corresponds to the factor $P[i..j]$.

Their claim relies on the result of Gawrychowski et al. [17] for the *constant-time weighted ancestor queries on suffix trees.*

A weighted tree is a rooted tree where an integer weight is assigned to each node, so that the weight of any node is strictly greater than the weight of its parent. A weighted ancestor query is, given a node V and an integer g, find the highest ancestor of V that has a weight at least g. In the context of suffix trees, the weight of a node is its string depth. Namely, Ehlers et al.'s approach [13] is to apply Gawrychowski et al.'s algorithm to the *truncated suffix tree* for P that consists only of the paths of string depths at most k. However, Gawrychowski et al.'s paper [17] does *not* consider construction time of their constant-time weighted ancestor data structure on suffix trees. Even on the (non-truncated) suffix tree for a string of length m, it seems rather challenging to construct the constant-time weighted ancestor data structure in $O(m \log m)$ time[4]. Hence, it is not known whether there exists an algorithm that satisfies the above claim (Lemma 5 of [13]), nor whether there exist algorithms that satisfy the other claim (Theorem 6 of [13]).

4.2 New real-time algorithms for extended k-Abelian matching

Here we propose some solutions for the online (and real-time) version of the extended k-Abelian pattern matching problem.

The basic framework of the approach by Ehlers et al. [13] is to compute the *k-gram matching statistics* for T against P, defined as follows: The k-gram matching statistics of T against P is the sequence of $|T| - k + 1$ integers $\ell_1, \ldots, \ell_{|T|}$ such that for each $1 \leq j \leq |T| - k + 1$ each ℓ_j is the length of the longest prefix of the k-gram

[4] One of the authors from [17] wondered that $O(m \log^3 m)$ or $O(m \log^4 m)$ construction time might be plausible [16], but any non-trivial construction algorithm is not known to date.

$T[j..j+k-1]$ that occurs in P. A k-gram $T[h..h+m-1]$ occurring at position h in T is extended k-Abelian equivalent to P iff $\ell_h = \ell_{h+1} = \cdots = \ell_{h+m-1} = k$. For each text position $1 \leq j \leq |T| - k + 1$, ℓ_j can be computed in $O(1)$ *amortized* time per text letter, using a similar technique to Ukkonen's online suffix tree construction [36], where traversals of "virtual" suffix links of implicit nodes are simulated by the suffix links of their explicit parents. The number of nodes that are visited in the simulation of each suffix link in $\mathsf{STree}(P)$ (and hence for each text letter) can be amortized constant [36], and this is basically what Theorem 5 of their paper [13] for non real-time solutions achieves.

To de-amortize the cost, Ehlers et al. [13] considered to use a weighted ancestor data structure instead of virtual suffix links. The idea is that one can find the locus pointed by a (virtual) suffix link with a weighted ancestor query from a corresponding leaf in the truncated suffix tree for P.

Instead of using the data structure by Gawrychowski et al. [17] whose construction time is unknown, one could use *level ancestor queries* on a limited class of weighted trees where the weight of each node is its node depth (not string depth). It is known that one can preprocess a tree with m nodes in $O(m)$ time so that level ancestor queries can be answered in $O(1)$ worst-case time [6]. We build this data structure on the full suffix tree $\mathsf{STree}(P)$. We also preprocess $\mathsf{STree}(P)$ such that for each $1 \leq i \leq m-k+1$ the leaf representing the suffix $P[i..m]$ has a pointer to its (implicit or explicit) ancestor of string depth k. These pointers can easily be precomputed in $O(m)$ time by a standard tree traversal, as was done in Section 3.1. Now suppose that we have just computed ℓ_j for $T[j..j+k-1]$ against P, and let i be one of the positions in P such that $P[i..i+\ell_j] = T[j..j+\ell_j]$. Then, for the weight $\ell_j - 1$ that represents the string depth we wish to jump up for the next text position $j+1$ in the text, we first take the pointer from the next leaf for $P[i+1..m]$, and from this leaf we binary search the nearest ancestor with weight $\ell_j - 1$ by level ancestor queries. Recall that this simulates the (virtual) suffix link traversal. If we can traverse with letter $T[j+\ell_j+1]$ from this locus of weight (i.e. string depth) $\ell_j - 1$, we are done for position $j+1$. Otherwise, we move to the next leaf for $P[i+2..m]$ and perform binary search for weight $\ell_j - 2$, and so forth. Since we need level ancestor queries only from nodes of string depth at most k (and hence node depth at most k), we can binary search the weighted ancestor with $O(\log k)$ level ancestor queries, in $O(\log k)$ worst-case time for each text letter.

Alternatively, we can use a weighted ancestor data structure that is designed for arbitrary weighted trees (i.e., not specialized for suffix trees). Kopelowitz and Lewenstein [30] showed that weighted ancestor queries on a weighted tree with m nodes can be reduced to a constant number of predecessor queries on a collection of predecessor data structures that maintain a total of $O(m)$ elements, where the number of elements in each predecessor data structure is bounded by the height of the tree. Insertions of new nodes can also be supported by a constant number of updates (insertions/deletions) in the collection of predecessor data structures. Therefore, if there is a dynamic predecessor data structure for a set of m integers over the universe $[1..u]$, that allows for queries/updates in $\mathrm{pred}(m,u)$ time and $O(m)$ space, then weighted ancestor queries on a weighted tree with m nodes with weights from $[1..u]$ can be answered in $O(\mathrm{pred}(m,u))$ time with $O(m)$ space (see Theorem 7.1 of [30]). We can plug-in the following linear-space dynamic predecessor data structures to the above result.

Lemma 1 ([4]). *There is a dynamic predecessor data structure for a set of up to m integers from the universe $[1..u]$ that uses $O(m)$ space and supports updates and queries in $O\Big(\min\Big\{\frac{(\log\log m)(\log\log u)}{\log\log\log u}, \sqrt{\frac{\log m}{\log\log m}}\Big\}\Big)$ worst-case time.*

Lemma 2 (y-fast trie [37] in conjunction with cuckoo hashing [34]). *There is a dynamic predecessor data structure for a set of up to m integers from the universe $[1..u]$ that uses $O(m)$ space and supports updates in $O(\log\log u)$ expected amortized time and predecessor queries in $O(\log\log u)$ worst-case time.*

In the current context we have $u = m$, since any node in $\mathsf{STree}(P)$ has string depth at most m. Note that $\min\Big\{\frac{(\log\log m)^2}{\log\log\log m}, \sqrt{\frac{\log m}{\log\log m}}\Big\} = \frac{(\log\log m)^2}{\log\log\log m}$.

Plugging these bounds where appropriate, we obtain the following:

Theorem 2 (Near real-time extended k-Abelian pattern matching). *Given a static pattern P of length m over the integer alphabet $[1..\sigma]$, and a positive integer k, the online version of Problem 2 can be solved in:*

- *$O(m\sigma)$ preprocessing time, $O(m\sigma)$ working space, and $O(\log k)$ worst case per text letter;*
- *$O(m\log\log m)$ preprocessing time, $O(m)$ working space, and $O(\log k)$ worst case per text letter;*
- *$O\Big(m\frac{(\log\log m)^2}{\log\log\log m}\Big)$ preprocessing time, $O(m)$ working space, and $O\Big(\frac{(\log\log m)^2}{\log\log\log m}\Big)$ worst-case time per text letter;*
- *$O(m\log\log m)$ expected preprocessing time, $O(m)$ working space, and $O(\log\log m)$ worst-case time per text letter.*

5 Conclusions and future work

In this paper, we pointed out some errors in the previous work by Ehlers et al. [13], provided a rigorous analysis on the complexities of some of the proposed algorithms by Ehlers et al. [13], and presented correct and alternative algorithms. For the offline k-Abelian pattern matching problem, we described that the algorithm by Ehlers et al. [13] indeed uses $O(n+m)$ space, and proposed a new offline algorithm which woks within $O(m)$ space. For the online k-Abelian pattern matching problem, we pointed out the abuse of the van Emde Boas data structure in Ehlers et al.'s algorithm and explained that this approach indeed uses $O(\sigma m)$ space. Finally, we pointed out that all the bounds claimed in [13] for the real-time extended k-Abelian pattern matching seem difficult to achieve, as these are heavily dependent on Gawrychowski et al.'s structure [17] of whose construction time is unknown. We proposed new alternative real-time algorithms for extended k-Abelian pattern matching with other data structures.

An interesting future work is to consider an efficient *indexing structure* for (extended) k-Abelian pattern matching. In a restricted case where both the alphabet size σ and k are fixed, then we can simply transform each k-gram in a given text T into a meta-letter, and transform T to a meta-string of length roughly kn. Since we have assumed that σ and k are fixed, $kn = O(n)$ and this meta-string is a string over an alphabet of size σ^k which can be seen as a constant as well. Thus we can use Amir et al.'s jumbled matching index [1] that works for any alphabet, or Kociumaka et al.'s jumbled matching index [29] that works for any constant-size alphabet. It would be interesting to develop an indexing structure that is specially designed for (extended) k-Abelian pattern matching with better complexities.

References

1. A. Amir, A. Butman, and E. Porat: *On the relationship between histogram indexing and block-mass indexing*. Philos. Trans. A. Math. Phys. Eng. Sci., 372(2016) 2014, p. 20130132.
2. A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein: *On hardness of jumbled indexing*, in Proc. ICALP 2014, 2014, pp. 114–125.
3. U. Baier: *Linear-time suffix sorting - A new approach for suffix array construction*, in Proc. CPM 2016, 2016, pp. 23:1–23:12.
4. P. Beame and F. E. Fich: *Optimal bounds for the predecessor problem and related problems*. J. Comput. Syst. Sci., 65(1) 2002, pp. 38–72.
5. M. A. Bender and M. Farach-Colton: *The LCA problem revisited*, in Proc. LATIN 2000, 2000, pp. 88–94.
6. M. A. Bender and M. Farach-Colton: *The level ancestor problem simplified*. Theor. Comput. Sci., 321(1) 2004, pp. 5–12.
7. P. v. E. Boas, R. Kaas, and E. Zijlstra: *Design and implementation of an efficient priority queue*. Mathematical Systems Theory, 10 1977, pp. 99–127.
8. P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták: *Algorithms for jumbled pattern matching in strings*. Int. J. Found. Comput. Sci., 23(2) 2012, pp. 357–374.
9. A. Butman, R. Eres, and G. M. Landau: *Scaled and permuted string matching*. Inf. Process. Lett., 92(6) 2004, pp. 293–297.
10. J. Cassaigne, J. Karhumäki, S. Puzynina, and M. A. Whiteland: *k-Abelian equivalence and rationality*. Fundam. Inform., 154(1-4) 2017, pp. 65–94.
11. T. M. Chan and M. Lewenstein: *Clustered integer 3SUM via additive combinatorics*, in Proc. STOC 2015, 2015, pp. 31–40.
12. F. Cicalese, G. Fici, and Z. Lipták: *Searching for jumbled patterns in strings*, in Proc. PSC 2009, 2009, pp. 105–117.
13. T. Ehlers, F. Manea, R. Mercas, and D. Nowotka: *k-Abelian pattern matching*. J. Discrete Algorithms, 34 2015, pp. 37–48.
14. P. Erdös: *Some unsolved problems*. Hungarian Academy of Sciences Mat. Kutató Intézet Közl, 6 1961, pp. 221–254.
15. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction*. J. ACM, 47(6) 2000, pp. 987–1011.
16. P. Gawrychowski: *Personal communication*, October 2017.
17. P. Gawrychowski, M. Lewenstein, and P. K. Nicholson: *Weighted ancestors in suffix trees*, in Proc. ESA 2014, 2014, pp. 455–466.
18. D. Hermelin, G. M. Landau, Y. Rabinovich, and O. Weimann: *Binary jumbled pattern matching via all-pairs shortest paths*. CoRR, abs/1401.2065 2014.
19. M. Huova and J. Karhumäki: *On the unavoidability of k-abelian squares in pure morphic words*. Journal of Integer Sequences, 16 2013, p. article 13.2.9.
20. M. Huova, J. Karhumäki, A. Saarela, and K. Saari: *Local squares, periodicity and finite automata*, in Rainbow of Computer Science - Dedicated to Hermann Maurer on the Occasion of His 70th Birthday, 2011, pp. 90–101.
21. M. Huova and A. Saarela: *Strongly k-Abelian repetitions*, in Proc. WORDS 2013, 2013, pp. 161–168.
22. J. Karhumäki, S. Puzynina, M. Rao, and M. A. Whiteland: *On cardinalities of k-abelian equivalence classes*. Theor. Comput. Sci., 658 2017, pp. 190–204.
23. J. Karhumäki, S. Puzynina, and A. Saarela: *Fine and Wilf's theorem for k-Abelian periods*. Int. J. Found. Comput. Sci., 24(7) 2013, pp. 1135–1152.
24. J. Karhumäki, A. Saarela, and L. Q. Zamboni: *On a generalization of Abelian equivalence and complexity of infinite words*. J. Comb. Theory, Ser. A, 120(8) 2013, pp. 2189–2206.
25. J. Karhumäki and M. A. Whiteland: *Regularity of k-Abelian equivalence classes of fixed cardinality*, in Adventures Between Lower Bounds and Higher Altitudes - Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday, 2018, pp. 49–62.
26. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction*. J. ACM, 53(6) 2006, pp. 918–936.
27. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Constructing suffix arrays in linear time*. J. Discrete Algorithms, 3(2-4) 2005, pp. 126–142.

28. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*. J. Discrete Algorithms, 3(2-4) 2005, pp. 143–156.
29. T. Kociumaka, J. Radoszewski, and W. Rytter: *Efficient indexes for jumbled pattern matching with constant-sized alphabet*. Algorithmica, 77(4) 2017, pp. 1194–1215.
30. T. Kopelowitz and M. Lewenstein: *Dynamic weighted ancestors*, in Proc. SODA 2007, 2007, pp. 565–574.
31. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings*. Inf. Process. Lett., 110(18-19) 2010, pp. 795–798.
32. 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 2012, pp. 5–9.
33. G. Nong, S. Zhang, and W. H. Chan: *Two efficient algorithms for linear time suffix array construction*. IEEE Trans. Computers, 60(10) 2011, pp. 1471–1484.
34. R. Pagh and F. F. Rodler: *Cuckoo hashing*. J. Algorithms, 51(2) 2004, pp. 122–144.
35. S. Sugimoto, N. Noda, S. Inenaga, H. Bannai, and M. Takeda: *Computing Abelian string regularities based on RLE*, in Proc. IWOCA 2017, 2017, pp. 420–431.
36. E. Ukkonen: *On-line construction of suffix trees*. Algorithmica, 14(3) 1995, pp. 249–260.
37. D. E. Willard: *Log-logarithmic worst-case range queries are possible in space $\Theta(N)$*. Inf. Process. Lett., 17(2) 1983, pp. 81–84.