k-Abelian Pattern Matching *

Thorsten Ehlers, Florin Manea, Robert Mercaş, and Dirk Nowotka

Christian-Albrechts-Universität zu Kiel, Institut für Informatik,
D-24098 Kiel, Germany, {the,flm,rgm,dn}@informatik.uni-kiel.de

Abstract. Two words are called k-abelian equivalent, if they share the same multiplicities for all factors of length at most k. We present an optimal linear time algorithm for identifying all occurrences of factors in a text that are k-abelian equivalent to some pattern P. Moreover, an optimal algorithm for finding the largest k for which two words are k-abelian equivalent is given. Solutions for various online versions of the k-abelian pattern matching problem are also proposed.

1 Introduction

The notion of k-abelian equivalence generalises the concepts of both the identity and the abelian equivalence of two words. Two words are called k-abelian equivalent, if they share the same multiplicities for all factors of length at most k. It is straightforward to see that two words of length n are identical if they are n-equivalent and they are abelian equivalent, if they are 1-abelian equivalent.

The notion of k-abelian equivalence was introduced in [1] and since then has captured more and more attention. The concept has been investigated with respect to repetitions [2–4], to periodicity properties [5], as well as to the complexity functions of the equivalence classes it determines [6]. In particular, it has been shown that in most situations, the concept oscillates between the two limit cases given above, identifying itself with one or the other.

Pattern matching is one of the most basic and well studied algorithmic problems: given a text T and a pattern P, we are interested in finding one or all occurrences of P in T. Besides the many obvious applications of pattern matching for finding a specific fragment in a larger sequential data structure, practical applications often emphasise the approximate variants of pattern matching like in the context of computational molecular biology [7]. Approximate pattern matching problems aim to find occurrences of factors of the text T that are equivalent to the pattern P by some given equivalence relation. In this paper, we investigate the approximate pattern matching problem with respect to k-abelian equivalence.

After fixing our notation, we show in Section 2 that the identification of all factors of a text T which are k-abelian equivalent to some pattern P can be done

* T. Ehlers is supported by the BMBF grant 01IS110355. F. Manea is supported by the *DFG* grant 596676. R. Mercaş is supported by the *DFG* grant 582014. D. Nowotka is supported by the *DFG Heisenberg* grant 590179.

in linear time with respect to the length of the text and the pattern, just as in the special cases of identity and abelian equivalence (Theorem 2). Moreover, we also show that identifying the largest k for which two given words are k-abelian equivalent takes time linear in the length of the words (Theorem 3). In Section 3 we investigate the pattern matching problem for k-abelian equivalences in the setting of online algorithms, and propose a series of real-time solutions of this problem (Theorem 4). Section 4 studies the same problem for an extended form of k-abelian equivalence. Finally, in Section 5 we give experimental results and discuss the problem of building index structures for k-abelian pattern matching. For the detailed proofs of the results in this paper please see the Appendix.

Preliminaries. An *alphabet*, i.e., Σ, is a finite set of symbols. Let $\sigma = |\Sigma|$ denote its cardinality and take $\Sigma = \{1, \ldots, \sigma\}$; at times we will use the letter $0 \notin \Sigma$. By ε we denote the *empty symbol*. A *word* w is a finite sequence of letters from Σ. We denote by $|w|$ its *length* and by $|w|_u$ the number of occurrences of u in w.

The set of all words over Σ is denoted by Σ^*, while the set of all words of length n is denoted by Σ^n for any positive integer n. The *catenation* of two words u and v is the word uv obtained by adding to the right of u the letters of v. For a factorization $w = uxv$, we say that x is a *factor* of w. Whenever u is empty, x is a *prefix* of w, i.e., $x \leq_p w$, and when v is empty, x is a *suffix* of w. For w of length n, and the numbers $i \leq j \in \{1, \ldots, n\}$, we denote by $w[i]$ the i^{th} symbol of w and by $w[i..j]$ the factor $w[i] \cdots w[j]$; clearly, $w = w[1..n] = w[1] \cdots w[n]$.

Considering the lexicographical order on Σ^*, for words u and v, we say that u is lexicographically smaller than v, i.e., $u \leq_{lex} v$, if either u is a prefix of v or there exist $a, b \in \Sigma$ such that $a < b$, $wa \leq_p u$, and $wb \leq_p v$ for some word w.

Let w be a word over Σ. The Parikh vector of the word w is an array $\pi_w[\cdot]$ with σ components, where $\pi_w[a] = |w|_a$ for all $a \in \Sigma$.

Words u and v are *abelian equivalent* if $|u|_a = |v|_a$ for all letters $a \in \Sigma$. That is, two words are *abelian equivalent* over Σ iff they have the same Parikh vector.

We say that u and v are *k-abelian equivalent*, i.e., $u \equiv_k v$, if either $u = v$ or $|u|, |v| \geq k$, $|u|_t = |v|_t$ for every $t \in \Sigma^k$, $u[1..k-1] = v[1..k-1]$ and $u[n-k+2..n] = v[n-k+2..n]$. According to [6], the suffix-equality requirement can be in fact dropped. An equivalent definition is that $u \equiv_k v$ if $|u|_t = |v|_t$ for every word t of length at most k. A *k-abelian n^{th} power* is a word $u_1 \ldots u_n$, where $u_1, \ldots, u_n$ are pairwise k-abelian equivalent. Obviously, 1-abelian equivalence is the same as abelian equivalence, while equality is equivalent to ∞-abelian equivalence.

Recall that a multi-set represents a set together with the multiplicity of each of the elements (i.e., both elements and multiplicities are present). We say that two words are *extended-k-abelian equivalent* if their multi-sets of factors of length k coincide (the condition of having the same prefixes is dropped).

In this paper, we solve a series of algorithmic problems related to k-abelian equivalence. The algorithms we propose use the RAM with logarithmic word size model. We also assume that whenever we are given as input of our problems a word w of length n, the alphabet of w is in fact included in $\{1, \ldots, n\}$ (i.e., $\sigma = |\Sigma| \leq n$). This is a common assumption in algorithmics on words and can be in fact replaced with a more general assumption, namely that Σ can be

sorted in linear time by radix sort (see, e.g., the discussion in [8]). Clearly, for all results proved for integer alphabets our reasoning holds canonically for constant alphabets (i.e., with $\sigma \in \mathcal{O}(1)$), as well. Finally, note that most pattern matching problems that we deal with require searching for a word P inside another word T; generally, P is called pattern and T is called text.

The following result is well known.

Theorem 1. *For a pattern $P \in \Sigma^m$ and a text $T \in \Sigma^n$, we can identify all factors of T that are abelian equivalent to P in $\mathcal{O}(n+m)$.*

Remark 1. The above result can be adapted to identify all the length $|P|$ factors P' of T that contain the same letters as P (not necessarily with same multiplicity as in P, so not abelian equivalent), and $\sum_{a \in \Sigma} |\pi_P[a] - \pi_{P'}[a]| \leq \Delta$, for some Δ.

We conclude the preliminaries section with a series of data structures.

For a string u of length n, over an alphabet $\Sigma \subseteq \{1, \ldots, n\}$, we define a suffix-array data structure that contains two arrays Suf_u, which is a permutation of $\{1, \ldots, n\}$, and lcp_u, with n elements from $\{0, \ldots, n-1\}$. Called the suffix array of u, Suf_u is defined such that $Suf_u[i] = j$ iff $u[j..n]$ is the i^{th} suffix of u, in the lexicographical order. The following lemma is straightforward.

Lemma 1. *Let $w \in \Sigma^n$. If for $1 \leq i < j \leq n$ and $u \in \Sigma^*$ we have $u \leq_p w[Suf_w[i]..n]$ and $u \leq_p w[Suf_w[j]..n]$, then $u \leq_p w[Suf_w[\ell]..n]$ for any $i \leq \ell \leq j$.*

The array lcp_u is defined by $lcp_u[1] = 1$ and $lcp_u[r]$ is the length of the longest common prefix of the suffixes found on positions r and $r-1$ in the suffix array, i.e., $u[Suf_u[r-1]..n]$ and $u[Suf_u[r]..n]$. Both arrays Suf_u and lcp_u can be constructed in $\mathcal{O}(n)$ time (see [8], and the references therein). Moreover, lcp_u can be processed in $\mathcal{O}(n)$ time to produce a more general data structure that enables us to return in constant time the answer to longest common prefix (or, for short, LCP-) queries "$LCP(i,j)$: What is the length of the longest common prefix of $u[i..n]$ and $u[j..n]$?".

2 Offline k-abelian pattern matching

The first step of our algorithms is to define the k-encoding of a word. For $w \in \Sigma^n$, we define the word $\#(w,k)$ of length $n-k+1$ as follows:

- let $S = \{w[i+1..i+k] \mid 0 \leq i \leq n-k\}$ be the set of length k factors of w;
- sort S lexicographically and associate with each factor $w[i+1..i+k]$ its rank (position) in the sorted set, i.e., $rank(i+1)$ for $0 \leq i \leq n-k$;
- let $\#(w,k)[i] = rank(i)$ for $1 \leq i \leq n-k+1$.

Clearly, $\#(w,k)$ is defined over an alphabet included in $\{1, \ldots, n-k+1\}$. Moreover, w is uniquely defined by the set S and the word $\#(w,k)$.

It is important to note that $\#(w,k)$ can be computed in linear time.

Lemma 2. *Let $w \in \Sigma^n$ with $\sigma \leq n$. We can compute $\#(w,k)$ in $\mathcal{O}(n)$ time.*

Proof. We determine the ranks $rank(i)$ of the factors $w[i+1..i+k]$ in the set $S = \{w[i+1..i+k] \mid 0 \leq i \leq n-k\}$ by identifying in the suffix array of w the contiguous groups of suffixes that share a common prefix of length k, and then assigning to each of these groups (from left to right) consecutive numbers, starting with 1; the suffixes of length less than k are not taken into account. □

The following lemma, although straightforward, is essential to our algorithms.

Lemma 3. *Let $w_1, w_2 \in \Sigma^n$. If $w_1 \equiv_k w_2$ for some integer k, then $w_1[1..k-1] = w_2[1..k-1]$, $w_1[n-k+2..n] = w_2[n-k+2..n]$, and $\#(w_1, k) \equiv_1 \#(w_2, k)$.*

Proof. The equality $\#(w_1, k) \equiv_1 \#(w_2, k)$ follows from the fact that w_1 and w_2 have the same factors of length k, with the same multiplicities. □

If we take $w_1 = 1236$ and $w_2 = 1456$, both over the alphabet $\{1, \ldots, 6\}$, for $k = 1$ we have $w_1[1..k-1] = w_2[1..k-1] = \varepsilon$ and $\#(w_1, 1) = 1234 = \#(w_2, 1)$, but w_1 is not abelian equivalent to w_2. Hence, the converse implication of the lemma does not necessarily hold. In order for the converse to hold as well, we need to check that the two words have the same set of factors of length k.

We can test in linear time the k-abelian equivalence of two words:

Lemma 4. *Let $w_1, w_2 \in \Sigma^n$ and k be an integer with $1 \leq k \leq n$. We can decide whether $w_1 \equiv_k w_2$ in $\mathcal{O}(n)$ time.*

Proof. We construct $w = w_1 0 w_2$, and compute Suf_w. Next, we compute $\#(w, k)$ of length $2n-k+2$, and set $w'_1 = \#(w, k)[1..n-k+1]$ and $w'_2 = \#(w, k)[n+2..2n-k+2]$(the two encodings are done using the ranking of w, and disconsider all letters of $\#(w, k)$ that contain a 0). Hence, w'_1 and w'_2 contain the same letters if w_1 and w_2 have the same multi-set of factors of length k. Note that w'_1 and w'_2 are computed in linear time, as they only require the computation of $\#(w, k)$. Finally, we remark that $w_1 \equiv_k w_2$ iff $w_1[1..k-1] = w_2[1..k-1]$ and $w'_1 \equiv_1 w'_2$. This last equality can be tested in linear time, by Theorem 1. □

Lemma 4 together with its proof suggests a simple way to transform and solve in linear time the k-abelian pattern matching problem following the classical abelian pattern matching problem, solved in Theorem 1.

Problem 1. Given a text T and some pattern P, over an alphabet Σ, find all factors of T that are k-abelian equivalent to P.

Theorem 2. *Given a text $T \in \Sigma^n$ and some pattern $P \in \Sigma^m$, we can find all factors of T that are k-abelian equivalent to P in time $\mathcal{O}(n+m)$.*

Proof. As in the proof of Lemma 4, we construct the word $w = T0P$, and the encoding $\#(w, k) = w'$. Let $T' = w'[1..n-k+1]$ and $P' = w'[n+2..n+m-k+2]$. Also, build LCP-data structures for $T0P$. Now, for any $i > 0$, a factor $T[i..i+m-1]$ is k-abelian equivalent to P iff $T[i..i+k-2] = P[1..k-1]$ (tested in $\mathcal{O}(1)$ time using LCP-queries), and $T'[i..i+m-1] \equiv_1 P'$.

Therefore, it is enough to find in linear time all the positions i of T' where factors that are abelian equivalent to P' occur, and then check, for each of them, whether $T[i..i+k-2] = P[1..k-1]$. All the positions fulfilling these conditions correspond to positions in T where a factor k-abelian equivalent to P occurs. □

Remark 2. Since Problem 1 is reducible to the classical abelian pattern matching problem, by Remark 1, our algorithm can be adapted to produce in linear time all the factors P' of length $|P|$ of T that contain the same factors of length k as P (not necessarily in the same numbers, so P' is not necessarily k-abelian equivalent to P) such that $\sum_{t\in\Sigma^k} \mid |P'|_t - |P|_t \mid \leq \Delta$, for some Δ.

Using the same reduction, we can extend a result from [9]:

Corollary 1. *For a word $w \in \Sigma^n$ and a positive integer k, we can identify all factors of w that are k-abelian powers in $\Theta((n-k+1)^2)$ time.*

The results shown so far help us answer a related, bit more difficult problem.

Problem 2. For words $u, v \in \Sigma^n$, find the largest integer k such that $u \equiv_k v$.

The immediate approach to solve this problem is to look through all possible k for the largest value such that $u \equiv_k v$. With the search for k implemented as a binary search, this approach takes $\mathcal{O}(n \log n)$ time, using the solution described in Lemma 4. However, this problem can also be solved in linear time.

Theorem 3. *Given two words $u, v \in \Sigma^n$, we can find the greatest positive integer k such that $u \equiv_k v$ in linear time $\mathcal{O}(n)$.*

Proof. As before, we construct $w = u0v$, the Suf_w and LCP_w data structures.

Due to Lemma 1, if there exists a positive integer k such that $u \equiv_k v$, then the suffixes of both u and v that share a common prefix of length at least k are grouped together in Suf_w. Also, if k is maximum such that $u \equiv_k v$, then the suffixes of length at most $k-1$ of u and v coincide, and if we truncate the suffixes of u and v to length k, then we should obtain the same multi-set for each word.

Following this remark, we split the suffix array of w into two separate new arrays: one contains suffixes that correspond to u and the other one the suffixes corresponding to v, each with exactly $n-k+1$ elements. Using the two arrays, we compute in linear time the maximum ℓ_1, resp. ℓ_2, such that the suffixes of length ℓ_1-1, resp. prefixes of length ℓ_2-1, of u and v coincide. We take $\ell = \min\{\ell_1, \ell_2\}$.

Further, going simultaneously through the sorted suffixes of u and v we compute the longest common prefix of the i^{th} suffix of u (in lexicographic order) and of the i^{th} suffix of v. Let ℓ' be the minimum value for which there exists i such that the i^{th} suffix of u shares a common prefix of length exactly ℓ' with the i^{th} suffix of v (the suffixes are again counted in lexicographical order), but both suffixes have length at least $\ell'+1$. The value k we were looking for is $\min\{\ell', \ell\}$.

Therefore, to solve Problem 2, we first compute in linear time the values ℓ and ℓ', and then return the value k we were looking for as $\min\{\ell, \ell'\}$. The whole algorithm takes $\mathcal{O}(n)$ time, clearly, and produces the desired output. □

3 Real-time k-abelian pattern matching

So far we discussed static problems, i.e., in Problem 1 both P and T are given at the beginning and we process them both in order to solve the problem. Now

consider a different type of problem: we are given $P \in \Sigma^m$ and k, while T is read letter by letter (i.e., in an online manner). We want to preprocess P so we can tell, after each new letter, whether the prefix of T read so far ends with a factor k-abelian equivalent to P. We assume that $\Sigma = \{1, \ldots, \sigma\}$ with $\sigma \in \mathcal{O}(m)$.

Problem 3. Preprocess a pattern P and an integer k such that when given a text T, in letter by letter manner, to answer at each moment, efficiently, whether the prefix of T read so far ends with a factor k-abelian equivalent to P.

In general, an algorithm for such a problem is called online algorithm. For simplicity, when discussing this type of problem, we call the time needed to tell whether the prefix of T ends with a factor k-equivalent to P *query time*, while the time needed to preprocess P is called *preprocessing time*. If a solution has constant query time, then its algorithm is called *real-time*.

First, note that for $k = 1$, the result of Theorem 1 holds for the real-time version of Problem 3, as well (see the proof of Theorem 1 in the Appendix).

The solution for $k > 1$ is based on the k-encoding strategy used already in the previous sections. We consider the sets of letters $\{1, \ldots, \ell_1\}$ of $\#(P, k-1)$ and $\{1, \ldots, \ell_2\}$ of $\#(P, k)$, where $\ell_1 \leq \ell_2 \leq m-k+2$. Recall that $i \in \{1, \ldots, \ell_1\}$ (resp., $i \in \{1, \ldots, \ell_2\}$) is the rank of a factor of length $k-1$ (resp., k) of P, in the lexicographically ordered set of all factors of length $k-1$ (resp., k) of P. Let $f_1[i]$ (resp., $f_2[i]$) be the length $k-1$ (resp., k) factor of P, whose position in the lexicographically ordered set of factors of length $k-1$ (resp., k) of P is i.

Further, we compute the list of triples $L = \{(i, a, j) \mid 1 \leq i \leq \ell_1, 1 \leq j \leq \ell_2, a \in \Sigma$, and $f_1[i]a = f_2[j]\}$. The suffixes of P that share the same prefix of length $k-1$ form a contiguous subarray of the suffix array of P, according to Lemma 1. Moreover, each of these groups can be split into several subgroups, that come one after the other in the suffix array, based on the letter following the common prefix. Accordingly, these subgroups correspond to the groups of suffixes that share a common prefix of length k, ordered lexicographically. Computing the subgroups corresponding to a group of suffixes takes linear time, in the size of the group, thus $\mathcal{O}(m)$, altogether. Therefore, we can compute all elements of L in linear time, as well, and collect them in a linked list, for instance.

Alternatively, one can see L as the set of the (explicit or implicit) edges that connect (explicit or implicit) nodes of depth $k-1$ to (explicit or implicit) nodes of depth k in the suffix tree constructed for P.

Now, we discuss several ways of implementing L so we can efficiently test whether there is a triple $(i, a, \cdot)$ in L, and, if so, quickly find its third component.

One way is to implement a $m \times \sigma$ table $M[\cdot][\cdot]$, where $M[i][a] = j$ iff $(i, a, j) \in L$. In this case, both operations mentioned above take constant time, while constructing this data structure takes $\mathcal{O}(m\sigma)$ time and space; however, the table M is sparse, so such an implementation of L is not practical.

As each component of the triples of L is a number between 1 and m, and L has at most m elements, we can also use perfect hashing to construct a hash table with satellite information and a dictionary search data structure, useful to do the above mentioned operations in $\mathcal{O}(1)$ time. The construction of these

data structures can be done in $\mathcal{O}(m \log\log m)$ time deterministically (see [10, Theorem 1]) or in $\mathcal{O}(m)$ expected time, while the table itself takes $\mathcal{O}(m)$ space. Using these structures the two operations mentioned above take $\mathcal{O}(1)$ time.

Finally, allowing more than $\mathcal{O}(1)$ time for the operations on L, another implementation can be used. For each i we define a data structure in which the pairs (a, j), with $(i, a, j) \in L$, are stored. Assume that these pairs (a, j) are ordered by their first component, i.e., a. Now, each of the operations on L can be seen as a predecessor search among the pairs stored in the data structure associated to i, where the key on which the sorting/searching is done is the first component of the stored pairs. There are at most σ such components, so we can construct a van Emde Boas tree containing these pairs [11]. Since the time needed to construct such a tree is t_i (the number of triples having i on the first position) for each i, the time needed to construct the trees for all i's is $\mathcal{O}(m)$, and they can be stored in $\mathcal{O}(m)$ space. Further, doing predecessor search in each structure takes $\mathcal{O}(\log\log\sigma)$ time per query.

Once L constructed, we compute in $\mathcal{O}(m)$ time, using Suf_P and L, the values $suf[j]$ upper bounded by ℓ_1, such that $suf[j] = i$ iff $f_2[j] = af_1[i]$ for some $a \in \Sigma$.

As a first step for the real-time k-abelian pattern matching problem, using a real-time pattern matching algorithm, e.g., [12], each time we read a new letter of T we report whether $P[1..k-1]$ is a suffix of the prefix of T read so far.

Assume now that before reading the newest symbol of T, denoted a, the longest suffix of the text we already read, of length at most $|P|$ and whose factors of length k are all factors of P, was P' with $\#(P', k) = j_1 \dots j_{m'-k} j_{m'-k+1}$ for some $m' \leq m$. Let $i = suf[j_{m'-k+1}]$, and check whether a triple $(i, a, \cdot)$ is in L.

If so, we return its third component, say j. The suffix of T becomes P'', with $\#(P'', k) = j_1 \dots j_{m'-k} j_{m'-k+1} j$, if $m' < m$, or $\#(P'', k) = j_2 \dots j_{m'-k} j_{m'-k+1} j$, otherwise. Using real-time abelian pattern matching we check whether $\#(P'', k)$ is abelian equivalent to $\#(P, k)$. If yes, we can decide in constant time whether the prefixes of length $k-1$ of P and P'' coincide, and, hence whether $P'' \equiv_k P$.

When no $(i, a, \cdot)$ is in L, we restart the procedure above when we find a new occurrence of $P[1..k-1]$, by reading the next letter and taking $i = \#(P, k-1)[1]$.

Therefore, the time we spend for each read letter is upper bounded by the time needed to find a triple in L. In conclusion, we obtain the following result.

Theorem 4. *Given a pattern $P \in \Sigma^m$ for $|\Sigma| = \sigma$, and a positive integer k, the online k-abelian pattern matching problem can be solved in:*

- $\mathcal{O}(m\sigma)$ *preprocessing time,* $\mathcal{O}(m\sigma)$ *space, and* $\mathcal{O}(1)$ *query time.*
- $\mathcal{O}(m \log\log m)$ *preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(1)$ *query time.*
- $\mathcal{O}(m)$ *expected preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(1)$ *query time.*
- $\mathcal{O}(m)$ *preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(\log\log\sigma)$ *query time.*

4 Online extended-k-abelian pattern matching

In this section, we consider the following more general problem.

Problem 4. Preprocess a pattern P and an integer k such that given a text T, in letter by letter manner, to answer at each moment, efficiently, whether the prefix of T read so far ends with a factor extended-k-abelian equivalent to P.

Although the idea used to solve Problem 3 cannot be directly applied in this setting, the strategy stays the same: for every letter of the text T we read we check if the suffix of length P can be encoded using the letters of $\#(P, k)$, and use a real-time abelian pattern matching algorithm to tell whether this suffix is extended-k-abelian equivalent to P. For this we maintain and update for each new letter read, a succinct representation of the longest factor of P, with length at most k, that is a suffix of the prefix of T read so far.

In addition to the previously mentioned data structures, we now also construct in $\mathcal{O}(m)$ time the suffix tree of P together with the suffix links (see, e.g., [13]). We say that a node of the suffix tree of P corresponds to the factor $P[i..j]$ iff the path from the root to that node is labelled with $P[i..j]$. For the succinct representation of the longest factor of P with length at most k, that is a suffix of the prefix of T read so far, we use the (explicit or implicit) node of the suffix-tree of P that corresponds to this factor, together with its length; when that node is implicit, we store the lowest explicit ancestor of this node.

For simplicity, we store the edges of the suffix tree of P using perfect hashing; we can check in $\mathcal{O}(1)$ time whether a node is the source of an edge labelled with a certain letter, and simultaneously get the respective target node (when it exists). Also, the longest factor of P with length at most k, that is a suffix of the prefix of T read so far is called *the P-suffix of that prefix of T*, or *the current P-suffix.*

Online Algorithm. We first remark that if $P[i..i+\ell-1]$ is the P-suffix of $T[1..j]$, then the P-suffix of $T[1..j+1]$ has length at most $\ell+1$. Therefore, when updating the representation of the current P-suffix, we just have to find the longest suffix of the previous P-suffix that can be extended by the letter $T[j+1]$.

We show first how to compute the P-suffix of $T[1..j+1]$ when the succinct representation defined above of the P-suffix of $T[1..j]$, namely $X = P[i..i+\ell-1]$, is known. If $\ell = k$ we use the approach in the previous section. If $\ell < k$ and $a = T[j+1]$, we first try to extend X with a, and see whether the new string Xa is a factor of P (an edge whose label starts with a leaves the node corresponding to X in the suffix tree of P). If yes, then Xa becomes the current P-suffix and we update the succinct representation of the factor of P according to the node where the aforementioned edge leads. If not, then we try extending the factor $X[2..\ell]$ with a. To compute its corresponding node in the suffix tree, we take the lowest explicit ancestor N_1 of the node N corresponding to X, and follow its suffix link. This takes us to an explicit node N_2 that corresponds to a prefix of $X[2..\ell]$, which is not necessarily the lowest explicit ancestor of the node corresponding to that factor. Thus, we use the letters that labelled the path from N_1 to N (i.e., the remaining suffix of $X[2..\ell]$) to advance in the suffix tree from N_2, until we reach the node N_3 corresponding to $X[2..\ell]$; we also compute in the same time the lowest explicit ancestor of N_3. Then we check again if this node is the source of an edge whose label begins with a. If yes, then we found the current P-suffix;

this is $P[i+1..i+\ell-1]a$, and we, therefore, also have its representation. If not, we repeat the procedure for the factor $X[3..\ell]$, now a suffix of $X[2..\ell]$, and so on.

It is simpler to compute the total time spent executing all the above algorithm, than upper bound each of the steps. Notice that each symbol of T is used only once to go through an edge of the tree. Thus, in total, we make at most $\mathcal{O}(n)$ steps for this action and only constant time for each of the other steps, which are executed at most $\mathcal{O}(n)$ times. Therefore, the overall complexity of maintaining the succinct representation of the longest suffix of T that is a factor of P with length at most k, is $\mathcal{O}(n)$. Using this, together with the approach of the previous section (for $\ell = k$), we get that Problem 4 can be solved in time $\mathcal{O}(n)$ and space $\mathcal{O}(m)$, to which the time and space needed to preprocess the pattern P should be added (either $\mathcal{O}(m \log\log m)$ in a deterministic implementation of the perfect hashing, or $\mathcal{O}(m)$ expected). The space needed remains $\mathcal{O}(m)$.

Other implementations of the set of edges of the suffix tree lead, as previously discussed, to other time complexities: with preprocessing time and space $\mathcal{O}(m\sigma)$ (resp. $\mathcal{O}(m)$) the algorithm runs in $\mathcal{O}(n)$ (resp. $\mathcal{O}(n \log\log \sigma)$) time.

Real-time Algorithm. For a real-time algorithm, the idea is to report only the factors of T that are extended-k-abelian equivalent to P, thus not update the information after each new read letter, but have it ready whenever a length k factor of P is found. For this we also need the following data structures from [14]:

Lemma 5. *We can preprocess a word $P \in \Sigma^m$ in $\mathcal{O}(m \log k)$ time and linear space $\mathcal{O}(m)$ such that, for each i and j with $j - i \leq k$, we can return in constant time the (explicit or implicit) node of the suffix tree of P corresponding to $P[i..j]$.*

Using this we skip the search of the tree for nodes corresponding to the suffixes of the current P-suffix. However, for a constant upper bound on the time needed to perform the update of the representation of the P-suffix that we try to maintain, we still have to deal with the unknown number of suffixes of the current P-suffix of $T[1..j]$, to determine the P-suffix of $T[1..j+1]$, when a new letter is read.

First, we maintain a queue of the letters read from T. In each step enqueue the new letter, and perform two more checks: 1) check whether the factor of P whose representation is stored can be extended by the first element in the queue; if yes, 2) update the P-suffix and its representation accordingly, delete the first letter from the queue, and repeat the previous check with the current P-suffix and its representation (also performing the eventual update of the P-suffix and of the queue); if no, 2′) using the $\mathcal{O}(1)$ query in Lemma 5, get the node for the longest proper suffix of the factor of P whose representation we had, and check whether the first letter of the queue extends it.

Since when the current P-suffix has length ℓ and it cannot be extended the number of suffixes of factors of P we have to check until reaching again a P-suffix of length ℓ equals the number of letters read between these two moments, we use the lazy update algorithm described above to update the succinct representation of the current P-suffix, and output that we did not find a factor of T that is extended-k-abelian equivalent to P as long as its length is not k; when we reach a length k factor, the queue is empty and the succinct representation is that of

the current P-suffix of the read prefix. Then we proceed just as in the case of the algorithm for Theorem 4, until the length decreases, and repeat the procedure.

Again, the previous discussion on the implementation of the suffix tree applies. Thus, for a real-time algorithm, the preprocessing uses $\mathcal{O}(m)$ space and the time needed is $\mathcal{O}(m(\log k + \log\log m))$ deterministically, or $\mathcal{O}(m \log k)$ expected.

5 Further Remarks

Experiments. We tested the algorithm for Theorem 2 on the E.coli and Vibrio cholera genomes with text sizes $4,638,690$ and $1,109,333$, respectively, for $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Having in mind the DNA sequencing process, one may see our pattern matching problem in the following way: given a template sequence P, we want to find out whether the long sequence T contains other sequences that may produce the same reads (of length k). For exemplification purposes, the pattern P was chosen so that $m = 100$ and $P = T[i..i+m-1]$ for random $i \in \{1, \ldots, n-m+1\}$. The values of k were chosen as multiples of 3, considering the normal length of a codon, with the assumption that removing an entire amino acid does not change the structure of a protein as much as the removal of one of the nucleotides from the translating RNA. We looked, as suggested in Remark 2, for factors P' of T such that the sum, over all factors of length k, of the absolute values of the differences between the number of occurrences of a factor in P' and those of the same factor in P is upper bounded by some Δ. For each values of k and Δ, we ran 100 tests. As P is a factor of T, each test finds at least one match. Fig. 1 reports the average number of matches except for this one occurrence.

Δ	E.coli			Vibrio cholera		
	k=3	k=6	k=9	k=3	k=6	k=9
0	0.04	0.04	0.04	0.01	0	0
2	0.04	0.04	0.04	0.01	0	0
4	0.04	0.04	0.04	0.02	0	0
8	0.06	0.04	0.04	0.03	0	0
16	0.1	0.05	0.04	0.11	0	0
32	0.16	0.05	0.05	0.27	0	0
64	353.2	0.05	0.05	98.26	0	0
128	18576.09	0.04	0.05	4742.9	0.02	0
256	18715.59	7.32	0.05	4747.54	1.45	0

Fig. 1. Experiments with 100 runs each

k	E.coli	Vibrio cholera	$\sigma^k + k$
1	5	5	5
2	18	18	18
3	67	67	67
4	260	260	260
5	1,011	1,029	1,029
6	4,102	4,102	4,102
7	16,390	16,389	16,391
8	65,371	65,106	65,544
9	256,559	234,324	262,153

Fig. 2. Alphabet size for $\#(w, k)$

For $k \geq 6$ the number of matches is quite low. Interestingly, the number of matches for $\Delta = 256$ is not too far from the expected number of matches, if these tests were run on random data. In that case, as $m \geq 2k-2$ and Δ is sufficiently large, the problem is degraded to finding exact matches on two factors of length $k-1$ each. Hence, the probability for a match is upper bounded by σ^{-2k+2}.

Additionally, we tested the extended version of k-abelian matching (Fig. 3). Here, we ignored the requirement of equal prefixes and suffixes of length $k-1$.

Please note that for $\Delta = 256$ there is a match on every factor of T. Again, we found only few matches for small values of Δ. A reason for this may be the increased number of different length k factors which appear, in close numbers, in the chosen texts. In Fig. 2 we give the number of different ranks that appeared in $\#(w, k)$, together with an upper bound $\sigma^k + k$. This bound is due to the

Δ	E.coli			Vibrio cholera		
	k=3	k=6	k=9	k=3	k=6	k=9
0	0.07	0.04	0.04	0.02	0	0
2	2.34	2.12	2.12	2.19	2	2
4	4.74	4.2	4.2	4.41	4	4
8	9.38	8.36	8.36	9.09	8.01	8
16	20.44	16.76	16.68	19.38	16.05	16
32	48.06	33.74	33.47	46.05	32.21	32.01
64	78747.81	68.12	67.19	22283	64.98	64.02
128	4594162.28	139.14	135.21	1106909.91	139.68	131.16
256	4638591	4638591	4638591	1108151	1108151	1108151

Fig. 3. Disregarding matches on prefixes and suffixes, 100 runs each

fact that there are at most σ^k factors of length k, and additionally k factors containing 0. The number of occurrences are quite close to the upper bound.

A motivation for the very rare occurrences of some pattern in an arbitrary text comes also from an analytical analysis of the probability of a match in the latter setting; given the dependencies among the letters of $\#(T, k)$, the problem can be seen in terms of a Markov source of order $k - 1$. Another, more loose view of the problem, would be in the form of the string matching over reduced set of patterns problem, when the set of independent length k factors occurring at positions $kj + 1$ in the pattern, for $j \geq 0$, is the one that we look for within a factor of length $|P|$ in the text. However, this second model is not tight as it does not consider the fact that each factor of length k is influenced by the preceding $k - 1$ factors of length k. For more details, we recommend [15, Sect. 7.2–7.3].

Index structures. A problem worth considering in this context, and that was recently considered in the context of abelian pattern matching [16, 17], is that of building index structures for k-abelian pattern matching. Basically, now we are given a positive integer k and a text T, and we want to preprocess the text such that we can answer quickly queries in which we are given a pattern P and have to report whether T has a factor that is k-abelian to P. Recall our assumption that the alphabet of T and of the query-patterns is integer.

Generally, we can approach the problem as follows. Following the solution of the online pattern matching problem, we consider the sets of letters $\{1, \ldots, \ell_1\}$ of $\#(T, k-1)$ and $\{1, \ldots, \ell_2\}$ of $\#(T, k)$, where $\ell_1 \leq \ell_2 + 1 \leq n - k + 2$. Let $f_1[i]$ (resp., $f_2[i]$) be the factor of length $k-1$ (resp., k) of T, whose rank in the lexicographically ordered set of all factors of length $k-1$ (resp., k) of T is i. We construct Suf_T and the list $L = \{(i, a, j) \mid 1 \leq i \leq \ell_1, 1 \leq j \leq \ell_2, a \in \Sigma$, and $f_1[i]a = f_2[j]\}$, and implement L such that we can test efficiently whether there is a triple $(i, a, \cdot)$ in L, and, if so, to find efficiently its third component.

Assume we use perfect hashing to store the edges of the suffix tree of T and the list L. Then, given a pattern $P \in \Sigma^m$, we find in time $\mathcal{O}(k)$ an occurrence of $P[1..k-1]$ in T (if none exists, then no factor of T is k-abelian equivalent to P). Next, reading $P[k..m]$ letter by letter, and checking the list L, we can produce $\#(P, k)$ in $\mathcal{O}(m-k)$ time. Hence the problem is reduced to producing an index of $\#(T, k)$, useful to check efficiently whether a factor abelian equivalent to $\#(P, k)$ occurs in $\#(T, k)$. Clearly, the classical abelian matching approach can be slightly adapted so we can check whether the length $k-1$ prefix of P matches the length $k-1$ prefix of the factor identified in T. Again, the preprocessing time depends on the implementation of the suffix tree of T and the list L (see the

discussions in the previous sections); the query time is obtained by adding up the time needed to locate $P[1..k-1]$ in T, then to compute $\#(P,k)$, and, finally, to answer the abelian pattern matching query for the text $\#(T,k)$ and pattern $\#(P,k)$. Such an abelian pattern matching query is answered in $\mathcal{O}(n-k+1)$ time, for $\#(T,k)$ and $\#(P,k)$ over an integer alphabet. Unfortunately, not much is known about building indexes for abelian pattern matching when the alphabet is integer. However, it is our hope that since the letters of $\#(T,k)$ or $\#(P,k)$ do not occur in an arbitrary order (the alphabet they are defined on is not a random integer alphabet), we could solve the indexing problem faster than the naive approach. It is worth mentioning that $\#(T,k)$ and $\#(P,k)$ are words over large alphabets, even if one assumes that Σ is constant (the letters of the encodings are words in Σ^k, which is not of constant size when k is not a constant).

References

1. Huova, M., Karhumäki, J., Saarela, A., Saari, K.: Local squares, periodicity and finite automata. In: Rainbow of Computer Science. Springer (2011) 90–101
2. Huova, M., Karhumäki, J., Saarela, A.: Problems in between words and abelian words: k-abelian avoidability. Theor. Comput. Sci. **454** (2012) 172–177
3. Mercaş, R., Saarela, A.: 3-abelian cubes are avoidable on binary alphabets. In: DLT 17. LNCS 7907 (2013) 374–383
4. Rao, M.: On some generalizations of abelian power avoidability. Preprint (2013)
5. Karhumäki, J., Puzynina, S., Saarela, A.: Fine and Wilf's theorem for k-abelian periods. In: DLT 16. LNCS 7410 (2012) 296–307
6. Karhumäki, J., Saarela, A., Zamboni, L.Q.: On a generalization of abelian equivalence and complexity of infinite words. J. Combin. Theory Ser. A **120**(8) (2013) 2189 – 2206
7. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York (1997)
8. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. Journal of the ACM **53** (2006) 918–936
9. Cummings, L.J., Smyth, W.F.: Weak repetitions in strings. J. Combin. Math. Combin. Comput **24** (1997) 33–48
10. Ružić, M.: Constructing efficient dictionaries in close to sorting time. In: Automata, Languages and Programming. LNCS 5125. Springer (2008) 84–95
11. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: SFCS 16, IEEE Computer Society (1975) 75–84
12. Breslauer, D., Grossi, R., Mignosi, F.: Simple real-time constant-space string matching. In: CPM 22. LNCS 6661. Springer (2011) 173–183
13. Maaß, M.G.: Computing suffix links for suffix trees and arrays. Inf. Process. Lett. **101**(6) (2007) 250–254
14. Gawrychowski, P., Lewenstein, M., Nicholson, P.K.: Weighted level ancestors in suffix trees. Preprint (2014)
15. Lothaire, M.: Applied Combinatorics on Words. Cambridge University Press, Cambridge, New York (2005)
16. Kociumaka, T., Radoszewski, J., Rytter, W.: Efficient indexes for jumbled pattern matching with constant-sized alphabet. In: ESA 21. LNCS 8125 (2013) 625–636
17. Gagie, T., Hermelin, D., Landau, G.M., Weimann, O.: Binary jumbled pattern matching on trees and tree-like structures. In: ESA 21. LNCS 8125 (2013) 517–528

6 Appendix

Proof of Theorem 1
Say that $|P| = m$ and $|T| = n$. We define two arrays π_1 and π_2, each with σ elements. The first array π_1 is the Parikh vector of P, while π_2 is initially the Parikh vector of $T[1..m]$.

Define $\delta = \sum_{a \in \Sigma} |\pi_1[a] - \pi_2[a]|$. If $\delta = 0$, then we conclude that $P \equiv_1 T[1..m]$.

Further, we read the word T letter by letter, from left to right, starting with the $m + 1^{th}$ letter. The variable δ is updated as follows when the k^{th} letter of T is read, for $k > m$. If the k^{th} letter of T is a and the $(k-m)^{th}$ letter of T was b, then we decrease $\pi_2[b]$ by 1 and increase $\pi_2[a]$ by 1; thus, the value of δ is updated accordingly (basically, we just have to subtract from δ the initial values of $|\pi_1[a] - \pi_2[a]|$ and $|\pi_1[b] - \pi_2[b]|$ and then add the updated values). Clearly, if after the k^{th} letter of T was read and the updates were performed we have $\delta = 0$, then $P \equiv_1 T[k - m + 1..k]$.

Clearly, this process takes $\mathcal{O}(n + m)$ time. □

Proof of Lemma 2
We compute the suffix array Suf_w and the array lcp_w for the word w. As a consequence, we can determine the ranks $rank(i)$ of the corresponding factors $w[i + 1..i + k]$ in the set

$$S = \{w[i + 1..i + k] \mid 0 \leq i \leq n - k\},$$

as follows.

By Lemma 1, the suffixes of w that begin with the same factor of length k form a contiguous group in Suf_w. Thus, we identify in the suffix array of w the contiguous groups of suffixes that share a common prefix of length k, and then assign to each of these groups (as they occur when traversing the suffix array of w from left to right) consecutive natural numbers, starting with 1; the suffixes of length less than k are not taken into account.

Formally, we start with $r = 0$ and $i = 1$. While $|w[Suf[i]..n]| < k$ we increase the value of i by 1. Assume we reached a value of i such that $|w[Suf[i]..n]| \geq k$. In this case, we increase r by 1, we define $rank(Suf[i]) = r$, and, then, increase i with 1. While $lcp_w[i] \geq k$, we assign $rank(Suf[i]) = r$. As soon as we reached an i such that $|w[Suf[i]..n]| < k$ or $lcp_w[i] < k$, we restart the process with the current values of r and i. □

Proof of Lemma 4
We construct the word $w = w_1 0 w_2$, and we compute its suffix array. Next, we compute $\#(w, k)$; this word has length $2n - k + 2$.

We now set

$$w'_1 = \#(w, k)[1..n - k + 1], \text{ and}$$
$$w'_2 = \#(w, k)[n + 2..2n - k + 2].$$

Intuitively, w'_1 is an encoding of w_1 that is very similar to the encoding $\#(w_1, k)$, only that in assigning ranks to the suffixes of w_1 we took into account also the

suffixes of w_2; similarly, w'_2 is an encoding of w_2 when taking into account also the suffixes of w_1; moreover, both encodings disconsider all letters of $\#(w,k)$ that contain a 0. Accordingly, the fact that w'_1 and w'_2 contain exactly the same letters is equivalent to them having exactly the same multi-set of factors of length k. Note that w'_1 and w'_2 can be computed in linear time, as they only require the computation of $\#(w,k)$.

Finally, we remark that $w_1 \equiv_k w_2$ if and only if $w_1[1..k-1] = w_2[1..k-1]$, $w_1[n-k+1..n] = w_2[n-k+1..n]$, and $w'_1 \equiv_1 w'_2$. This last equality can be clearly tested in linear time, using for example Theorem 1. □

Proof of Corollary 1

Just as before, we first create the encoding $\#(w,k)$ associated to w as well as the Suf_w and LCP_w structures.

Next, using the result of [9], we find all abelian powers in $\#(w,k)$ in $\Theta((n-k+1)^2)$. Note that these comprise of all abelian repetitions that have length at least k. For each abelian repetition u, using LCP_w queries, check if in w, for u', the factor of w that has been encoded by u, all of the consecutive occurrences of its abelian root share both the same prefixes, as well as the suffixes of length $k-1$ match. This is done in constant time per query, thus we still need only $\mathcal{O}((n-k+1)^2)$ time. Every time the answer is positive, we add u' to a set S consisting of all k-abelian repetitions. Please note that in fact our prefix and suffix checks are quite easy as the algorithm proposed in [9] consists in fact of a linked list of abelian squares. Thus, our check is performed for each of these abelian squares in constant time, summing up to $\Theta((n-k+1)^2)$.

Finally, we have to take care of all k-abelian powers that have length less than k. However, these comprise of all the classical powers. In [*Crochemore, An Optimal Algorithm for Computing the Repetitions in a Word. Inf. Process. Lett., 12* **(5)***, 1981, 244–250*], an optimal $\mathcal{O}(n\log(n))$ algorithm doing this has been given. Moreover, since all powers are given in the form (i,p,e), where i is the position where the repetition occurs, p it's period, and e its exponent, choosing the ones that have length less than k is trivial as we only need to check if $k-1 \leq p \cdot e$. Therefore, by adding to S all of those that fulfil this property proves our result. □

Proof of Theorem 3

Just like before, we first construct the word $w = u0v$, its suffix array Suf_w, and LCP_w data structures.

Note that, due to Lemma 1, if there exists a positive integer k such that $u \equiv_k v$, then the suffixes of both u and v that share a common prefix of length at least k are grouped together (i.e., occur on consecutive positions) in Suf_w. Also, if k is maximum such that $u \equiv_k v$, then the suffixes of length at most $k-1$ of u and v coincide, and if we truncate the suffixes of u and v to length k, then we should obtain exactly the same multi-set for each of the two words.

Following this remark, we proceed as follows. We first split the suffix array of w into two separate new arrays: one of them contains suffixes that correspond to u and the other one contains the suffixes corresponding to v. Each of the

two arrays has exactly $n-k+1$ elements. Looking at these two arrays, we can compute in linear time the maximum ℓ_1 such that the suffixes of length ℓ_1-1 of u and v coincide;then we compute, still in linear time, a value ℓ_2 such that the prefixes of length ℓ_2-1 of u and v coincide. We take $\ell=\min\{\ell_1,\ell_2\}$. The maximum value k such that $u\equiv_k v$ is at most ℓ, by the definition of k-abelian equivalence. Further, going simultaneously through the sorted suffixes of u and v we compute the longest common prefix of the i^{th} suffix of u (in lexicographic order) and of the i^{th} suffix of v; the value obtained each time upper bounds, just like ℓ, the value k that we are looking for. These values can be computed in linear time, using the two arrays we constructed in order to access in constant time the lexicographically i^{th} suffixes of the two words, and LCP queries for $w=u0v$ to actually compute their longest prefixes.

If the lexicographically i^{th} suffix of u equals the lexicographically i^{th} suffix of v, for all i, then $u\equiv_n v$. For the sake of generality, let us assume this is not the case. Let ℓ' be the minimum value for which there exists i such that the i^{th} suffix of u shares a common prefix of length exactly ℓ' with the i^{th} suffix of v (the suffixes are again counted in lexicographical order), but both suffixes have length at least $\ell'+1$. We claim that the value k we were looking for is $\min\{\ell',\ell\}$.

Indeed, it is not hard to see that $u\equiv_s v$ for all positive integers $s\leq\min\{\ell',\ell\}$. If $\ell\leq\ell'$, and if $s>\ell$, then there exists s' such that $s-1\geq s'>\ell-1$ and the suffixes or prefixes of length s' of u and v are not equal. This is in contradiction with the definition of $\equiv_s$. If $\ell>\ell'$, and if $s>\ell$, then there exists i such that the prefix of length s of the lexicographically i^{th} suffix of u does not coincide with the prefix of same length of the lexicographically i^{th} suffix of v, so by truncating the suffixes of u and v to length s, we do not obtain exactly the same multi-set. This is yet another contradiction to the definition of $\equiv_s$.

An algorithm that solves Problem 2 works, thus, as follows. We first compute in linear time the values ℓ and ℓ', and then return the value k we were looking for as $\min\{\ell,\ell'\}$. The whole algorithm takes $\mathcal{O}(n)$ time, clearly, and produces the desired output, by the previous arguments. □

Proof of Theorem 4

First, note that for $k=1$, the result of Theorem 1 holds for the real-time version of Problem 3, as well (see the proof of Theorem 1 in the Appendix).

The solution for $k>1$ is based on the k-encoding strategy used already in the previous sections. We consider the sets of letters $\{1,\ldots,\ell_1\}$ of $\#(P,k-1)$ and $\{1,\ldots,\ell_2\}$ of $\#(P,k)$, where $\ell_1\leq\ell_2+1\leq m-k+2$. Recall that $i\in\{1,\ldots,\ell_1\}$ (respectively, $i\in\{1,\ldots,\ell_2\}$) is the rank of a factor of length $k-1$ (respectively, k) of P, in the lexicographically ordered set of all factors of length $k-1$ (respectively, k) of P. Let $f_1[i]$ (respectively, $f_2[i]$) be the length $k-1$ (respectively, k) factor of P, whose position in the lexicographically ordered set of factors of length $k-1$ (respectively, k) of P is i.

Further, we compute the list of triples

$$L=\{(i,a,j)\mid 1\leq i\leq\ell_1,1\leq j\leq\ell_2,a\in\Sigma,\text{ and }f_1[i]a=f_2[j]\}.$$

The suffixes of P that share the same prefix of length $k-1$ form a contiguous subarray of the suffix array of P, according to Lemma 1. Moreover, each of these groups can be split into several subgroups, that come one after the other in the suffix array, based on the letter following the common prefix. Accordingly, these subgroups correspond to the groups of suffixes that share a common prefix of length k, ordered lexicographically. Computing the subgroups corresponding to a group of suffixes takes linear time, in the size of the group, thus $\mathcal{O}(m)$, altogether. Therefore, we can compute all elements of L in linear time, as well, and collect them in a linked list, for instance.

Alternatively, one can see L as the set of the (explicit or implicit) edges that connect (explicit or implicit) nodes of depth $k-1$ to (explicit or implicit) nodes of depth k in the suffix tree constructed for P.

Now, we discuss several ways of implementing L so we can efficiently test whether there is a triple $(i, a, \cdot)$ in L, and, if so, quickly find its third component.

One way is to implement an $m \times \sigma$ table $M[\cdot][\cdot]$, where $M[i][a] = j$ if and only if $(i, a, j) \in L$. In this case, both operations mentioned above take constant time, while constructing this data structure takes $\mathcal{O}(m\sigma)$ time and space.

As the components of all the triples of L are numbers between 1 and m, and L has at most m elements, we can also use perfect hashing to construct a hash table and a dictionary search data structure, useful to do the above mentioned operations in $\mathcal{O}(1)$ time. The construction of these data structures can be done in $\mathcal{O}(m \log\log m)$ time deterministically (see [10, Theorem 1]) or in $\mathcal{O}(m)$ expected time, while the table itself occupies $\mathcal{O}(m)$ space. Basically, they are used as follows. In the dictionary we store all the pairs (i, a) such that there is a triple (i, a, j) in L. We also store an m-positions array $W[\cdot]$ such that $W[k] = j$ if and only if the hash code of (i, a) is k and $(i, a, j) \in L$. Using these structures it is immediate how the two operations mentioned above are executed in constant time.

Finally, allowing more than constant time for the operations to be performed on L (which translates in having a near-real-time algorithm solving the k-abelian pattern matching problem), another implementation can be used. For each i we keep a data structure in which the pairs (a, j), with $(i, a, j) \in L$, are stored. We can assume that we get these pairs (a, j) ordered by their first component, namely a. Now, each of the operations to be performed on L can be seen as a predecessor search among the pairs stored in the data structure associated to i, where the key on which the sorting/searching is done is the first component of the stored pairs. As these components are at most σ, we can construct a van Emde Boas tree containing the aforementioned pairs [11]. Since the time needed to construct such a tree is t_i (the number of triples having i on the first position) for each i, the time needed to construct the trees for all i's is $\mathcal{O}(m)$, and they can be stored in $\mathcal{O}(m)$ space. Further, doing predecessor search in each of these structures takes $\mathcal{O}(\log\log\sigma)$ time per query.

Once L constructed, we also compute for each j with $1 \leq j \leq \ell_2$ the positive values $suf[j]$ and $pref[j]$, both upper bounded by ℓ_1, such that $suf[j] = i$ if and only if $f_2[j] = af_1[i]$ for some $a \in \Sigma$, while $pref[j] = i$ if and only if

$f_2[j] = f_1[i]a$. All these values are easily computed in linear time, using the suffix array of P, and the list L.

Now, we explain how the real-time k-abelian pattern matching problem is solved. As a first step, each time we read a new letter of T we report whether the prefix of length $k-1$ of P, namely $P[1..k-1]$, is a suffix of the prefix of T read so far. For this we can use a real-time pattern matching algorithm, e.g., [12].

Next, assume that, before reading the new symbol of T, the longest suffix of the part of T we already read, with length at most $|P|$ and whose factors of length k are all factors of P, was P' such that $\#(P', k) = j_1 \dots j_{m'-k} j_{m'-k+1}$, for some $m' \leq m$. Assume that now we read the letter a. We take $i = suf[j_{m'-k+1}]$, and check whether a triple $(i, a, \cdot)$ exists in L.

If so, we return its third component, say j. The suffix of T becomes P'', with

$$\#(P'', k) = \begin{cases} j_1 \dots j_{m'-k} j_{m'-k+1} j, \text{ if } m' < m, \text{ or} \\ j_2 \dots j_{m'-k} j_{m'-k+1} j, \text{ otherwise.} \end{cases}$$

Using real-time abelian pattern matching we check whether $\#(P'', k)$ is abelian equivalent to $\#(P, k)$ or not. If yes, we can decide in $\mathcal{O}(1)$ time whether the prefixes of length $k-1$ of P and P'' coincide, and, hence whether $P'' \equiv_k P$.

When no $(i, a, \cdot)$ is in L, we restart the procedure above when we find a new occurrence of $P[1..k-1]$, by reading the next letter and taking $i = \#(P, k-1)[1]$, where $[1]$ is the next read letter.

Therefore, the time we spend for each read letter is upper bounded by the time needed to find a triple in L. □

Results obtained in Section 4

We first give an online algorithm that needs $\mathcal{O}(|P|)$ space, and takes $\mathcal{O}(|T|)$ time to read the entire text $T \in \Sigma^n$ and report the occurrences of factors that are extended-k-abelian equivalent to $P \in \Sigma^m$, thus having constant amortised query time. The preprocessing time is either $\mathcal{O}(m \log\log m)$ in a deterministic framework, or expected $\mathcal{O}(m)$ in a randomised setting. This algorithm is neither real-time nor near-real-time. We then present a real-time algorithm, with a slightly increased preprocessing time but with the same $\mathcal{O}(m)$ space complexity.

We emphasise that the approach used to solve Problem 3 cannot be used directly to solve this new problem, as the restriction that each of the factors of T we are looking for starts with $P[1..k-1]$ does not hold anymore. However, the general idea stays the same: we identify the longest suffix of the part of the text T read so far that can be encoded using the letters of $\#(P, k)$, and use a real-time abelian pattern matching algorithm to tell whether this suffix is extended-k-abelian equivalent to P. To obtain this, we maintain, and update for each letter of T we read, a succinct representation of the longest factor of P, with length at most k, that is a suffix of the prefix of T read so far. In addition to the data structures we constructed for P in the previous sections, we now also construct in $\mathcal{O}(m)$ time the suffix-tree of P together with the suffix links (see, e.g., [13]). Intuitively, the main data-structure of the algorithm becomes now the suffix-tree of P (with the nodes of depth more than k trimmed), replacing the

set L, that only consisted in a stripe of edges of the respective suffix tree (the edges connecting nodes of depth $k-1$ to the ones of depth k).

We say that a node of the suffix tree of P corresponds to the factor $P[i..j]$ iff the path from the root to that node is labelled with $P[i..j]$. For the succinct representation of the longest factor of P with length at most k, that is a suffix of the prefix of T read so far, we use the (explicit or implicit) node of the suffix-tree of P that corresponds to this factor, together with its length; when that node is implicit, we store the lowest explicit ancestor of this node. The main part of our algorithm is to maintain this information efficiently.

For simplicity, assume that we store the edges of the suffix tree of P using perfect hashing, so we can check in constant time whether a node is the source of an edge labelled with a certain letter, and simultaneously get the target node of that edge (when it really exists). Also, the longest factor of P with length at most k, that is a suffix of the prefix of T read so far is called, in the following, the P-suffix of that prefix of T, or the current P-suffix.

Theorem 5. *Given a pattern $P \in \Sigma^m$ for $|\Sigma| = \sigma$, and a positive integer k, the online extended-k-abelian pattern matching problem can be solved in:*

- $\mathcal{O}(m\sigma)$ *preprocessing time,* $\mathcal{O}(m\sigma)$ *space, and* $\mathcal{O}(1)$ *amortised query time.*
- $\mathcal{O}(m \log\log m)$ *preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(1)$ *amortised query time.*
- $\mathcal{O}(m)$ *expected preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(1)$ *amortised query time.*
- $\mathcal{O}(m)$ *preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(\log\log\sigma)$ *amortised query time.*

Proof. The first remark we make is that if $P[i..i+\ell-1]$ is the P-suffix of $T[1..j]$, then the P-suffix of $T[1..j+1]$ has length at most $\ell+1$. Therefore, when updating the representation of the current P-suffix, we just have to find the longest suffix of the previous P-suffix that can be extended by the letter $T[j+1]$.

We show in the following how to compute the P-suffix of $T[1..j+1]$ when we know the succinct representation defined above of the P-suffix of $T[1..j]$. Say that the latter factor is $P[i..i+\ell-1]$ with $\ell < k$, and let $a = T[j+1]$; when $\ell = k$ we can essentially use the approach in the previous section. We first try to extend $P[i..i+\ell-1]$ with the letter a, and see whether the newly obtained string $P[i..i+\ell-1]a$ is a factor of P; this is done by checking whether there is an edge whose label starts with a leaving the node corresponding to $P[i..i+\ell-1]$ in the suffix tree of P. If yes, then $P[i..i+\ell-1]a$ becomes the current P-suffix and we just update the succinct representation of the factor of P, according to the node where the aforementioned edge leads. If not, then we have to try extending the factor $P[i+1..i+\ell-1]$ with a. But first, we have to compute its corresponding node in the suffix tree. For this, we take the lowest explicit ancestor N_1 of the node N corresponding to $P[i..i+\ell-1]$, and follow its suffix link. This takes us to an explicit node N_2 that corresponds to a prefix of $P[i+1..i+\ell-1]$ which is not necessarily the lowest explicit ancestor of the node corresponding to that factor. Thus, we use the letters that labelled the path from N_1 to N (so, basically, the remaining suffix of $P[i+1..i+\ell-1]$) to advance in the suffix tree from N_2, until we reach the node N_3 corresponding to $P[i+1..i+\ell-1]$; we also find the lowest explicit ancestor of N_3. Then we check again if this node is the source of

an edge whose label starts with a. If yes, then we found the current P-suffix; this is $P[i+1..i+\ell-1]a$, and we, therefore, also have its representation. If not, we repeat the procedure for the factor $P[i+2..i+\ell-1]$, which is now a suffix of $P[i+1..i+\ell-1]$, and so on.

We can summarise our approach in the following meta-steps (when $\ell < k$):

1. Let N be the node of the suffix tree corresponding to $P[i..i+\ell-1]$, let N_1 be the lowest explicit ancestor of N.
2. If there is an implicit or explicit node M such that the (explicit of implicit) edge from N to M has the label starting with $a = T[j+1]$, then $T[1..j+1]$ has the suffix $P[i..i+\ell-1]a$ which is a factor of P, whose corresponding node in the suffix tree is M. Moreover, the lowest explicit ancestor of M is either M itself (if it is explicit) or the lowest explicit ancestor of N, otherwise.
3. Otherwise, let N_2 be the target-node of the suffix link of N_1. Assume that the path from root to N_2 is labelled with $P[i+1..s]$. While the (explicit or implicit) edge leaving N_2 and labelled with $P[s+1]Y$, for some word Y, is of length at most $(\ell-1)-(s-i+2)$, we advance along that edge, set N_2 to be the implicit node we reach, update s in order to have $P[i+1..s]$ the label of the path from root to N_2, and repeat the procedure. When this is no longer possible, the current N_2 is the lowest explicit ancestor of the node corresponding to $P[i+1..i+\ell-1]$; this latter node, called N_3 can be immediately discovered, by selecting the appropriate edge from those leaving N_2 and returning the implicit node found at the right length along that edge.
4. After updating $i = i+1$ and $N = N_3$, we restart the process from step 2.

It is rather hard to put a precise upper bound on the time needed to perform the four meta-steps described above. However, it is simpler to compute the total time spent executing all these steps (during the entire computation). The third step of the above is the only one that cannot be executed in constant time. Thus let us see how much time we spend executing the third step in the entire computation. We notice that, actually, each symbol of T is used only once to go through an edge of the tree: indeed, if $P[i+1..s]$ labels the path from root to N_2, then the path from root to N_1 was labelled with $P[i..s]$. So, the total time spent by our algorithm executing the operations in step 3 is $\mathcal{O}(n)$. Now, the rest of the meta-steps are executed at most $\mathcal{O}(n)$ times. Indeed, assume that $T[j-\ell+1..j]$ is the suffix of T equal to $P[i..i+\ell-1]$, and $T[j+2-\ell'..j+1]$ is the longest suffix of $T[1..j+1]$ that equals a factor of P of length $\ell' \leq k$; we have $\ell' \leq \ell+1$, so the following holds:

$$n-(j-\ell+1)+n-j > n-(j+2-\ell')+n-(j+1).$$

This value is maximal at the beginning of the algorithm, when it equals $2n-1$; then it decreases in every execution of the meta-steps $2-4$. This shows that these steps are executed $\mathcal{O}(n)$ times. Thus, the overall complexity of maintaining the succinct representation of the longest suffix of T that is a factor of P with length at most k, is $\mathcal{O}(n)$. Using this, together with the approach in the previous section (when $\ell = k$), we get that Problem 4 can be solved in time $\mathcal{O}(n)$ and space $\mathcal{O}(m)$,

to which the time and space needed to preprocess the pattern P should be added (and this time is either $\mathcal{O}(m \log\log m)$ in a deterministic implementation of the perfect hashing used to store the edges of the suffix tree of P or, alternatively, expected linear time $\mathcal{O}(m)$). The space needed by our algorithm remains $\mathcal{O}(m)$.

Other implementations of the set of edges of the suffix tree lead, as discussed in the previous section, to other time complexities. With preprocessing time and space $\mathcal{O}(m\sigma)$, in which the incidence matrix of the tree is constructed and stored, the time needed to maintain the representation of the current suffix and to do the pattern matching stays $\mathcal{O}(n)$. With preprocessing time and space $\mathcal{O}(m)$, in which the edges are stored using van Emde Boas trees, the matching algorithm runs in $\mathcal{O}(n \log\log \sigma)$ time. Both these results show that for constant alphabets, Problem 4 can be solved in linear time $\mathcal{O}(n+m)$ and space $\mathcal{O}(m)$. □

The above algorithm was not even a near-real-time algorithm, as we could not bound the time needed to update the succinct representation of the current suffix of T. Now we propose a slightly different implementation of the strategy above, that leads to a real-time algorithm.

Theorem 6. *Given a pattern $P \in \Sigma^m$ for $|\Sigma| = \sigma$, and an integer $k > 0$, the (near-) real-time extended-k-abelian pattern matching problem can be solved in:*

- $\mathcal{O}(m(\sigma + \log k))$ *preprocessing time,* $\mathcal{O}(m\sigma)$ *space, and* $\mathcal{O}(1)$ *query time.*
- $\mathcal{O}(m(\log\log m + \log k))$ *preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(1)$ *query time.*
- $\mathcal{O}(m \log k)$ *expected preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(1)$ *query time.*
- $\mathcal{O}(m \log k)$ *preprocessing time,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(\log\log k)$ *query time.*

Proof. The main idea is to only report the factors of T that are extended-k-abelian equivalent to P. Therefore, we will not have the information we wanted to maintain updated at each step anymore (that is, immediately after a new letter of T is read), but we will have it (and be able to check it) whenever it is really needed: when a factor of length k of P is found (that is, a desired factor of T might appear).

We can easily see that the data structures constructed in Lemma 5 help us get rid of the part where we search the tree for nodes corresponding to the suffixes of the current P-suffix. Now we get this information in constant time, by one query. However, this is not enough to get a constant upper bound on the time needed to perform the update of the representation of the P-suffix that we try to maintain. When reading a letter $T[j+1]$ we still have to go through an unknown number of suffixes of the current P-suffix of $T[1..j]$, to determine the P-suffix of $T[1..j+1]$. Just like before, the time needed to check these prefixes amortises to constant time when the entire text is read. However, here we are interested in the worst case behaviour.

Fortunately, we can employ now a simple idea. We do not need to have always an updated representation of the current P-suffix. We only want to have it up to date when we reached the case when the current P-suffix has length exactly k, so we can use the ideas from real-time k-abelian matching. We further explain how this can be achieved.

We keep a queue of the letters read from T. Also, we memorise the node in the suffix tree corresponding to factor X of P (which is supposed to match the current P-suffix when there are no more letters in the queue); initially (that is, before any letter of T was read), X corresponds to the empty string. In each step when a new letter of T is read, we put it at the end of the queue, and also perform exactly two other checks (which can be implemented in constant or near-constant time, see the discussion below). First, we check whether X can be extended by the first letter from the queue (as previously described, this means checking whether there is an edge labelled with the respective letter leaving the node corresponding to X in the suffix tree). If yes, we update both X and the P-suffix and their representation accordingly (i.e., save the node corresponding to the respective factor of P extended with a), delete the first letter from the queue, and do another similar check with the current content of the queue and the current X and its representation. Otherwise, we get the node corresponding to $X[2..|X|]$ (obviously this is a factor of P, hence we can use the data structures in Lemma 5); this suffix becomes then the current suffix of T that we want to extend with the first letter of the queue, so it will be saved under the name X; we check whether this can be done or not. After these two checks, we can move on and read the next letter of T.

A special case is when we have that X and the current P-suffix have length k. To see this, just note that if at some point we had that X (the longest factor of P which was a suffix of the prefix of T read till that point) had length ℓ, and X could not be extended, then the number of suffixes of factors of P (more precisely, suffixes of the different words X we produce in our search) we had to check until reaching again a situation where the current X has length ℓ equals the number of letter of T read between these two moments.

Therefore, we use the lazy update algorithm we described above to update the succinct representation of the P-suffix of the part of T we just read, and output that we did not find a factor of T that is extended-k-abelian equivalent to P as long as the current P-suffix does not have length k; when we reached a factor of length k, the queue will be empty and the succinct representation will be exactly that of the current P-suffix of the prefix of T we read so far. Then we can proceed exactly as in the case of the real-time algorithm used for the conventional k-abelian equivalence. When we reach again the case of a suffix of length less than $k-1$, we have to repeat the procedure.

As we do only a constant number of checks for each letter of T we read, the algorithm is real-time if we can check in constant time whether a given edge exists in the suffix tree or near-real-time if this check is done in near-constant time. Again, the discussion regarding the implementation of the suffix tree from the previous section applies. We stress that in the case of a real-time algorithm, the preprocessing takes either $\mathcal{O}(m \log k + m \log\log m)$ time in a deterministic setting, or $\mathcal{O}(m \log k)$ expected time. In both cases, the space used is $\mathcal{O}(m)$. □