Flexible and Efficient Algorithms for Abelian Matching in Genome Sequence

Simone Faro[1](✉) and Arianna Pavone[2]

[1] Dipartimento di Matematica e Informatica, Università di Catania, Viale Andrea Doria 6, 95125 Catania, Italy
faro@dmi.unict.it

[2] Dipartimento di Scienze Cognitive, Università di Messina, Via Concezione 6, 98122 Messina, Italy
apavone@unime.it

Abstract. Approximate matching in strings is a fundamental and challenging problem in computer science and in computational biology, and increasingly fast algorithms are highly demanded in many applications including text processing and DNA sequence analysis. Recently efficient solutions to specific approximate matching problems on genomic sequences have been designed using a filtering technique, based on the general abelian matching problem, which firstly locates the set of all candidate matching positions and then perform an additional verification test on the collected positions.

The *abelian pattern matching problem* consists in finding all substrings of a text which are permutations of a given pattern. In this paper we present a new class of algorithms based on a new efficient fingerprint computation approach, called *Heap-Counting*, which turns out to be fast, flexible and easy to be implemented. We prove that, when applied for searching short patterns on a DNA sequence, our solutions have a linear worst case time complexity. In addition we present an experimental evaluation which shows that our newly presented algorithms are among the most efficient and flexible solutions in practice for the abelian matching problem in DNA sequences.

Keywords: Approximate string matching · Abelian matching jumbled matching · Experimental algorithms

1 Introduction

Given a pattern x and a text y, the *abelian pattern matching* problem [10] (also known as *jumbled matching* [6,13] or *permutation matching* problem) is a well known special case of the approximate string matching problem and consists in finding all substrings of y, whose characters have the same multiplicities as in x, so that they could be converted into the input pattern just by permuting their characters.

I. Rojas et al. (Eds.): IWBBIO 2019, LNBI 11465, pp. 307–318, 2019.
https://doi.org/10.1007/978-3-030-17938-0_28

Example 1. For instance, assume that $y = ccgatacgcattgac$ is a text of length 15 and $x = accgta$ is a pattern of length 6, then x has two abelian occurrences in y, at positions 1 and 4, respectively, since both substrings *cgatac* and *tacgca* are permutations of the pattern.

This problem naturally finds applications in many areas, such as string alignment [3], SNP discovery [4], and also in the interpretation of mass spectrometry data [5]. We refer to the recent paper by Ghuman and Tarhio [13] for a detailed and broad list of applications of the abelian pattern matching problem.

More interestingly related with scope of this paper abelian matching finds application in the field of approximate string matching in computational biology, where algorithms for abelian matching are used as a filtering technique [2], usually referred to as *counting filter*, to speed up complex combinatorial searching problems. The basic idea is that in many approximation problems a substring of the text which is an occurrence of a given pattern, under a specific distance function, is also a permutation of it. For instance, the counting filter technique has been used solutions to the approximate string matching problem allowing for mismatches [16], differences [18], inversions [7] and translocations [15].

In this paper we are interested in the *online* version of the problem which assumes that the input pattern and text are given together for a single instant query, so that no preprocessing of the text is possible. Although its worst-case time complexity is well known to be $O(n)$, in the last few years much work has been made in order to speed up the performances of abelian matching algorithms in practice, and some very efficient algorithms have been presented, tuned for specific settings of the problem [9,13].

Specifically we present two algorithms based on a new efficient fingerprint computation approach, called *Heap-Counting*, which turns out to be fast, flexible and ease to be implemented, especially for the case of DNA sequences. The first algorithm is designed using a prefix based approach, while the second one uses a suffix based approach. We prove that both of them have a linear worst case time complexity.

In addition we present also two fast variants of the above algorithms, obtained by relaxing some algorithmic constraints, which, despite their quadratic worst case time complexity, turn out to be faster in some specific practical cases.

From our experimental results it turns out that our newly presented algorithms are among the most efficient and flexible solutions for the abelian matching problem in genomic sequences.

The paper is organized as follows. After introducing in Sect. 2 the relevant notations and describing in Sect. 3 the related literature, we present in Sect. 4 two new solutions of the online abelian pattern matching problem in strings, based on the Heap-Counting approach, and prove their correctness and their linear worst case time complexity. Then, in Sect. 5, we present a detailed experimental evaluation of the new presented algorithms, comparing them against the most effective solutions known in literature.

2 Basic Notions

Before entering into details we recall some basic notions and introduce some useful notations.

We represent a string x of length $|x| = m > 0$ as a finite array $x[0 \mathinner{..} m-1]$ of characters from a finite alphabet Σ of size σ. Thus, $x[i]$ will denote the $(i+1)$-st character of x, for $0 \leq i < m$, whereas $x[i \mathinner{..} j]$ will denote the substring of x contained between the $(i+1)$-st and the $(j+1)$-st characters of x.

Given a string x of length m, the occurrence function of x, $\rho_x : \Sigma \rightarrow \{0, \ldots, m\}$, associates each character of the alphabet with its number of occurrences in x. Formally, for each $c \in \Sigma$, we have:

$$\rho_x(c) = \big|\{i : x[i] = c\}\big|.$$

The *Parikh vector* [1,20] of x (denoted by pv_x and also known as *compomer* [5], *permutation pattern* [11], and *abelian pattern* [10]) is the vector of the multiplicities of the characters in x. More precisely, for each $c \in \Sigma$, we have

$$pv_x[c] = \big|\{i : 0 \leq i < m \text{ and } x[i] = c\}\big|.$$

In the following, the Parikh vector of the substring $x[i \mathinner{..} i+h-1]$ of x, of length h and starting at position i, will be denoted by $pv_{x(i,h)}$.

The procedure for computing the Parikh vector pv_x of a string x of length m needs an initialization of the vector which takes $O(\sigma)$ time, and an inspection of all characters of x which takes $O(m)$ time. Thus the Parikh vector can be computed in $O(m+\sigma)$ time.

In terms of Parikh vectors, the abelian pattern matching problem can be formally expressed as the problem of finding the set $\Gamma_{x,y}$ of positions in y, defined as

$$\Gamma_{x,y} = \big\{s : \ 0 \leq s \leq n-m \text{ and } pv_{y(s,m)} = pv_x\big\}.$$

3 Previous Results

For a pattern x of length m and a text y of length n over an alphabet Σ of size σ, the *online abelian pattern matching problem* can be solved in $\mathcal{O}(n)$ time and $\mathcal{O}(\sigma)$ space by using a naïve *prefix based approach* [10,16,18,19], which slides a window of size m over the text while updating in constant time the corresponding Parikh vector.

Specifically, for each position $0 \leq s < n-m$, and character $c \in \Sigma$, we have

$$pv_{y(s+1,m)}[c] = pv_{y(s,m)}[c] - \big|\{c\} \cap \{y[s]\}\big| + \big|\{c\} \cap \{y[s+m]\}\big|,$$

so that the vector $pv_{y(s+1,m)}$ can be computed from the vector $pv_{y(s,m)}$ by incrementing the value of $pv_{y(s,m)}[y[s+m]]$ and by decrementing the value of $pv_{y(s,m)}[y[s]]$. Thus, the test "$pv_{y(s+1,m)} = p_x$" can be easily performed in constant time. This is done by maintaining an error value e such that

$$e = \sum_{c \in \Sigma} \big|pv_x[c] - pv_{y(s,m)}[c]\big|, \mathit{for}\, 0 \leq s < n-m.$$

At the beginning of the algorithm the value of e is set to $\sum_{c\in\Sigma} \left|pv_x[c] - pv_{y(0,m)}[c]\right|$. Thus, when it becomes 0 an occurrence is reported.

A *suffix-based approach* to the problem has been presented in [10], as an adaptation of the Horspool string matching algorithm [17] to abelian pattern matching problem. Rather than reading the characters of the window from left to right, characters are read from right to left. Every time the reading restarts in a new window, starting at position s, the Parikh vector is initialized by setting $pv_{y(s,m)}[c] = 0$, for all $c \in \Sigma$. Then, during each attempt, as soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. An occurrence is reported when the whole window is inspected without reporting any frequency overflow. The resulting algorithm has an $\mathcal{O}(n(\sigma + m))$ worst-case time complexity but performs well in many practical cases, especially for long patterns and large alphabets.

Successive solutions to the problem tried to speed-up both the prefix-based and suffix-based approaches described above by applying techniques of algorithm engineering, where experimental evaluations play an important role.

In [8] Cantone and Faro presented the Bit-parallel Abelian Matcher (BAM), which applies bit-parallelism to enhance the suffix-based approach for the abelian pattern matching problem. It turns out to be very fast in practical cases. It has been also enhanced in [9] by reading 2-grams (BAM2), obtaining best results in most practical cases.

However, although the adaptive width of bit fields makes possible to handle longer patterns than a fixed width, the packing approach used in BAM can be applied only in the case where the whole Parikh vector fits into a single computer word. This makes the algorithm particularly suitable for small alphabets or short patterns, but not useful in the case of long patterns and large alphabets.

In [9], an attempt to adapt such strategy in the case of long patterns have been presented. The authors proposed the BAM-shared algorithm (BAMs for short) which uses a kind of alphabet reduction in order to make the bit-vector fit into a single computer word. This implies that the algorithm works using a filtering approach and, in case of a match, the candidate occurrence should be verified.

In [9] the authors presented also a simple and efficient bit-parallel suffix-based approach. Instead of packing the Parikh vector of a string, their algorithm, named Exact Backward for Large alphabets (EBL for short) maintains a bit vector B of size σ where, for each $c \in \Sigma$, $B[c]$ is set to 1 if the character c occurs in the pattern, and is set to 0 otherwise.

More recently, in [13], Ghuman and Tarhio enhanced the EBL suffix-based solution by using SIMD (Single Instruction Multiple Data) instructions. Their solution, named Equal-Any algorithm (EA for short), uses a SIMD load instruction for reading, at each iteration, the whole window of the text in one fell swoop and storing it in a computer word w. Experimental results show that such solution is 30% faster than previous algorithms for short English patterns. However, despite this results, it works only when the length of the pattern is less or equal to 16.

Some efficient variants of the prefix based approach have been also presented in the last few years. Among them in [15] Grabowsky *et al.* presented a more efficient prefix-based approach, which uses less branch conditions. In [9] Chhabra *et al.* presented a prefix-based solution named Exact Forward for Small alphabets (EFS for short) which applies the same packing strategy adopted by the BAM algorithm to the prefix-based approach, obtaining very competitive results in the case of short alphabets.

To delve into the problem we refer to a detailed analysis of the abelian pattern matching problem and of its solutions presented in [10] and in [14].

4 The Heap-Counting Approach

Let x and y be strings of length m and n, respectively, over a common alphabet Σ of size σ. As described above previous solutions for the abelian pattern matching problem maintain in constant time the symmetric difference, e, of the multisets of the characters occurring in the current text window and of those occurring in the pattern, respectively. Thus, when $e = 0$, a match is reported. Alternatively, they use a packed representation of the Parikh vector, where some kind of overflow sentinel is implemented in order to take track that the frequency of a given character has exceeded its corresponding value in the Parikh vector of the pattern. The aim is to perform the initialization of the Parikh vector in constant time and to perform vector updates in a very fast way.

Instead of using a structured representation of the Parikh vector of a string, fitting in a single computer word, our approach tries to map the multisets of our interest into natural numbers, using a heap-mapping function h that allows for very fast updates.

Specifically we suppose to have a function $h : \Sigma \rightarrow \mathbf{N}$ (the *heap-function*), that maps each character c of the alphabet Σ to a natural number, $h(c)$ indeed. Then, we assume that the multiset of a given string $w \in \Sigma^*$, of length m, can be univocally associated to a natural number, $h(w)$, using the following relation:

$$h(w) = \sum_{i=0}^{m-1} h(w[i]) \qquad (1)$$

The value $h(w)$ is called the *heap-value* of the string w. In this context a abelian match is found at position s of the text when the heap-value associated to the window starting at position s is equal to the heap-value of the pattern. This approach, when applicable, leads to two main advantages: the multisets of the characters occurring in string can be represented by a single numeric value, fitting in a single computer word; modifications and updates of such multisets can be done by means of simple integer additions.

Our heap-counting approach is based on the following elementary fact:

Let $\Sigma = \{c_0, c_1, \ldots, c_{\sigma-1}\}$ be an alphabet of size $|\Sigma| = \sigma$, let $m > 1$ be an integer, and let $h : \Sigma \rightarrow \mathbb{N}$ be the mapping $h(c_i) = m^i$, for $i = 0, \ldots, \sigma - 1$.

```
COMPUTE-HEAP-MAPPING($x$, $m$)
1.  for each $c \in \Sigma$ do $h[c] \leftarrow 0$
2.  $j \leftarrow 1$
3.  for $i \leftarrow 1$ to $m$ do
4.      if $h(x[i]) = 0$ then
5.          $h(x[i]) \leftarrow j$
6.          $j \leftarrow j \times m$
7.  return $h$

HEAP-COUNTING-ABELIAN-MATCHING($x$, $m$, $y$, $n$)
1.  $h \leftarrow$COMPUTE-HEAP-MAPPING($x$, $m$)
2.  $\delta \leftarrow \gamma \leftarrow 0$
3.  for $i \leftarrow 0$ to $m-1$ do
4.      $\delta \leftarrow \delta + h(x[i])$
5.      $\gamma_0 \leftarrow \gamma_0 + h(y[i])$
6.  if $\gamma_0 = \delta$ then OUTPUT(0)
7.  for $s \leftarrow 1$ to $n-m$ do
8.      $\gamma_s \leftarrow \gamma_{s-1} + h(y[s+m-1]) - h(y[s-1])$
9.      if $\gamma_s = \delta$ OUTPUT($s$)
```

Fig. 1. The pseudocode of the HEAP-COUNTING-ABELIAN-MATCHING for the online exact abelian matching problem, implemented using a prefix-based approach.

Then for any two distinct k-multicombinations (i.e., k-combinations with repetitions) φ_1 and φ_2 from the set Σ, with $1 \leq k \leq m$, we have

$$\sum_{c \in \varphi_1} h(c) \neq \sum_{c \in \varphi_2} h(c). \tag{2}$$

Example 2. Let $x = agcga$ be an input pattern of length 5 over the alphabet $\Sigma = \{a, c, g, t\}$ of size 4. Based on Lemma 4, the heap-function $h : \Sigma \rightarrow \mathbb{N}$, is defined as $h(a) = 1, h(c) = 5, h(g) = 25$, and $h(t) = 125$. The heap-value of x is then $h(x) = h(a) + h(g) + h(c) + h(g) + h(a) = 57$.

Let $\diamond$ be a character such that $\diamond \notin \Sigma$ and let $\Sigma_x \subseteq \Sigma$ be the set of all (and only) the characters occurring in x. We indicate with σ_x the size of the alphabet Σ_x. Plainly we have $\sigma_x \leq \min\{\sigma, m\}$, thus we can think to this transformation as a kind of alphabet reduction.

We define the *reduced text* $\bar{y}$, over Σ_x, as a version of the text y where each character $y[i]$, not included in Σ_x, is replaced with the special character $\diamond \notin \Sigma$. Since, in general, $\sigma_x < \sigma$ (especially in the case of short patterns), to process the reduced version of the text, instead of its original version, allows the heap function to be computed on a smaller domain, reducing therefore the size of the heap-values associated to any given string.

Example 3. Let $x = agcac$ be an input DNA sequence (the pattern) of length 5 and let $y =$*agtcagaccatcagata* be a text of length 17, both over the alphabet $\Sigma = \{a, g, c, t\}$ of size 4. We have that $\Sigma_x = \{a, c, d\}$. Thus the reduced version of y is $\bar{y} =$ ag◇c◇gacca◇caga◇◇

It can be proved that such alphabet reduction does not influence the output of any abelian pattern matching algorithm.

We are now ready to describe in details the two new algorithms based on the heap-counting approach. The algorithm described in Sect. 4.1 implements the heap-counting approach i a prefix-based algorithm, while the algorithm described in Sect. 4.2 uses a suffix-based mechanism.

4.1 A Prefix-Based Algorithm

Inspired by Lemma 4, the new algorithm precomputes the set Σ_x and the function $h : \Sigma_x \to \mathbb{N}$, defined as $h(c_i) = m^i$, for $i = 0, \ldots, \sigma_x - 1$, where m is the length of the pattern and σ_x is the size of Σ_x.

Figure 1 shows the Heap-Abelian-Matching algorithm and its the auxiliary procedure.

During the preprocessing phase (lines 1–7) the algorithm precomputes the heap-mapping function h (line 1) by means of procedure Compute-Heap-Mapping. Such procedure computes the mapping table over the alphabet $\Sigma_x \cup \{\diamond\}$ associating the value 1 with all characters not occurring in x, i.e. we set $h(\diamond) = 1$.

The heap values $\delta = h(x)$ and $\gamma_0 = h(y[0..m-1])$ are then precomputed in lines 2–5, Likewise, during the searching phase (lines 8–10), the heap value $\gamma_i = h(y[i \mathinner{..} i+m-1])$ is computed for each window $y[i \mathinner{..} i+m-1]$ of the text t, with $0 < i \leq n-m$. Specifically, starting from the heap value γ_{s-1}, the algorithm computes the heap value γ_s by using the relation $\gamma_s = \gamma_{s-1} - h(y[s-1]) + h(y[s+m-1])$ (line 8). Of course, in practical implementations of the algorithm it is possible to maintain a single value γ, corresponding to the heap value of the current window of the text.

The set $\Gamma_{x,y}$ of all occurrences in the text is then

$$\Gamma_{x,y} = \left\{ i \mid 0 \leq i \leq n-m \text{ and } \gamma_i = \delta \right\}$$

In order to compute the space and time complexity of the algorithm, it can be easily observed that the computation of the mapping h requires $O(m+\sigma)$-time and -space. Moreover observe that $\gamma_s = \gamma_{s-1} - h(y[s-1]) + h(y[s+m-1])$, so that γ_s can be computed in constant time from γ_{s-1}. Thus the set $\Gamma_{x,y}$ can be computed in $\mathcal{O}(n)$ worst case time. ■

From a practical point of view it is understood that for an architecture, say, at 64 bits, all operations will take place modulo 2^{64}. Thus, when m^{σ_x+1} exceeds 2^{64} we could have some collisions in the set of the heap values and an additional verification procedure should be run every time an occurrence is found. However it has been observed experimentally that, also in this specific cases, the collision problem for the heap function h is negligible.

4.2 A Suffix-Based Algorithm

In this section we extend the idea introduced in the previous section and present a backward version of the prefix-based algorithm described above which turns out to be more efficient in the case of long patterns or large alphabets. It shows a sub-linear behaviour in practice, while maintains the same worst case time complexity.

Figure 2 shows the pseudocode of the new algorithm, called Backward-Heap-Abelian-Matching, and its the auxiliary procedure.

```
COMPUTE-MEMBERSHIP-MAP($x$, $m$)
1.   for each $c \in \Sigma$ do $b(c) \leftarrow$ False
2.   for $i \leftarrow 1$ to $m$ do $b(x[i]) \leftarrow$ True
3.   return $b$

BACKWARD-HEAP-COUNTING-ABELIAN-MATCHING($x$, $m$, $y$, $n$)
1.   $h \leftarrow$COMPUTE-HEAP-MAPPING($x$, $m$)
2.   $b \leftarrow$COMPUTE-MEMBERSHIP-MAPPING($x$, $m$)
3.   $\delta \leftarrow 0$
4.   for $i \leftarrow 1$ to $m$ do $\delta \leftarrow \delta + h(x[i])$
5.   $y \leftarrow y.x$
6.   $s \leftarrow 0$
7.   while ( True ) do
8.       $\gamma \leftarrow -\delta$
9.       $j \leftarrow m - 1$
10.      while ($j \geq 0$) do
11.          if ($b(y[s + j])$) then
12.              $\gamma \leftarrow \gamma + h(y[s + j])$
13.              $j \leftarrow j - 1$
14.          else
15.              $\gamma \leftarrow -\delta$
16.              $s \leftarrow s + j + 1$
17.              $j \leftarrow m - 1$
18.      do
19.          if ($\gamma = 0$) then
20.              if ($s \leq n - m$) then OUTPUT($s$)
21.              else return
22.          if ($b(y[s + m])$ =False) then break
23.          $\gamma \leftarrow \gamma - h(y[s]) + h(y[s + m])$
24.          $s \leftarrow s + 1$
25.      while ( True )
26.      $s \leftarrow s + m + 1$
```

Fig. 2. The pseudocode of the Backward-Heap-Counting-Abelian-Matching for the online exact abelian matching problem, implemented using a suffix-based approach.

During the preprocessing phase (lines 1–6) the algorithm precomputes the heap-mapping function h (line 1) and the membership function b (line 2). We use procedure Compute-Heap-Mapping, and procedure Compute-Membership-Mapping, respectively.

The heap value $\delta = h(x)$ of the pattern x is then precomputed in lines 3–4. A copy of the pattern is then concatenated at the end of the pattern (line 5), as a sentinel, in order to avoid the window of the text to shift over the last position of the text.

The main cycle of the searching phase (line 7) is executed until the value of s becomes greater than $n - m$ (line 20). An iteration of the main cycle is divided into two additional cycles. The first cycle of line 10 performs a backward scanning of the current window of the text and stops when the whole window has been scanned or a character not occurring in Σ_x is encountered. The second cycle of line 18, starting from the heap value of the current window of the text, computes at each iteration the heap value of the next window in constant space using a forward scan. The second cycle stops when a character not occurring in Σ_x is encountered.

It can be proved that the algorithm Backward-Heap-Counting-Abelian-Matching computes all abelian occurrences of x in y with $O(\sigma + m + n)$-time and $O(\sigma + m)$-space complexity in the worst case.

4.3 Relaxed Filtering Variants

A simpler implementation of the above presented algorithms can be obtained by relaxing the heap-counting approach presented at the beginning of this section, in order to speed-up the computation of the heap values of a string and, as a consequence, to spud-up the searching phase of the algorithm.

Specifically we propose to use the natural predisposition of the characters of an alphabet to be treated as integer numbers. For instance, in many practical applications, input strings can be handled as sequences of ASCII characters. In such applications, characters can just be seen as the 8-bit integers corresponding to their ASCII code.

In this context, if we indicate with $\textsc{ascii}(c)$, the ASCII code of a character $c \in \Sigma$, we can set $h(c) = \textsc{ascii}(c)$. Thus the heap value of a string can be simply computed as the sum of the ASCII codes of its characters.

As a consequence the resulting algorithms works as a filtering algorithm. Indeed, when an occurrence is found we are not sure that the substring of the text which perform a match is a real permutation of the pattern. This implies that an additional verification phase is run for each candidate occurrences.

Plainly the resulting algorithms have an $O(\sigma + nm)$ worst case time complexity, since a verification procedure could be run for each position of the text.

5 Experimental Results

We report in this section the results of an extensive experimentation of the newly presented algorithms against the most efficient solutions known in literature for the online abelian pattern matching problem. In particular we have compared 11 algorithms divided in three groups: prefix-based, suffix-based and SIMD based algorithms. Specifically we compared the following 5 prefix based algorithms: the Window-Abelian-Matching (WM) [10,16,18,19]; the Grabowsky-Faro-Giaquinta (GFG) [15]; the Exact Forward form Small alphabets (EFS) [9]; the Heap-Counting-Abelian-Matching (HCAM) described in Sect. 4.1; the Heap-Filtering-Abelian-Matching (HFAM) described in Sect. 4.3.

We compared also the following 5 suffix based algorithms: the Backward-Window-Abelian-Matching (BWM) [10]; the Bit-parallel Abelian Matching algorithm (BAM) using 2-grams [8,9]; the Exact Backward for Large alphabets (EBL) [9]; the Backward-Heap-Counting-Abelian-Matching (BHCAM) described in Sect. 4.2; the Backward-Heap-Filtering-Abelian-Matching (BHFAM) described in Sect. 4.3.

In addition We compared also the Equal Any (EA), an efficient prefix based solution [13] implemented using SIMD instructions.

All algorithms have been implemented in C, and have been tested using the SMART tool [12] and executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 16 GB RAM 1600 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3.[1] Comparisons have been performed in terms of running times, including any preprocessing time.

For our tests we used a genome sequence provided by the research tool SMART, available online for download (for additional details about the sequences, see the details of the SMART tool [12]).

In the experimental evaluation, patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 1 \leq i \leq 8\}$. Thus at least one occurrence is reported for each algorithm execution. In all cases, the mean over the running times (expressed in hundredths of seconds) of 1000 runs has been reported.

Table 1 summarises the running times of our evaluations. The table is divided into four blocks. The first block presents the results relative to prefix based solutions, the second block presents the results for the suffix based algorithms, while the third block presents the results for the algorithm based on SIMD instructions. The newly presented algorithms have been marked with a star ($\star$) symbol. Best results among the two sets of algorithms have been bold-faced to ease their localization, while the overall best results have been also underlined. In addition we included in the last block the speedup (in percentage) obtained by our best newly presented algorithm against the best running time obtained by previous algorithms: positive percentages denote running times worsening, whereas negative values denote performance improvements. Percentages representing performance improvements have been bold-faced.

[1] The SMART tool is available online at https://smart-tool.github.io/smart/.

Table 1. Experimental results on a genome sequence.

	m	2	4	8	16	32	64
PREFIX BASED	WM	13.63	13.02	13.09	13.04	13.04	13.06
	GFG	20.47	20.19	20.42	20.41	20.47	20.30
	EFS	8.26	8.31	8.35	8.34	8.36	–
	HCAM ⋆	**6.97**	**6.86**	**6.92**	**6.93**	6.91	6.89
	HFAM ⋆	20.14	11.86	9.18	7.87	7.47	7.09
SUFFIX BASED	BWM	65.59	46.81	33.47	25.79	22.94	20.67
	BAM	**10.62**	**10.96**	13.20	11.87	10.69	9.85
	EBL	29.95	27.44	55.69	119.26	227.14	–
	BHCAM ⋆	26.82	18.03	**9.98**	**7.74**	**7.55**	**7.32**
	BHFAM ⋆	37.21	21.72	11.50	9.35	9.09	8.88
	Speed-up	+11.87%	+11.40%	+10.76%	+10.55%	**−12.12%**	**−29.16%**
SIMD	EA	**4.01**	**4.03**	**4.56**	**4.67**	–	–

Consider first the case of small alphabets, and specifically abelian string matching on strings over an alphabet of size $\sigma \leq 4$ (Table 1). From experimental results it turns out that prefix based solutions are more flexible and efficient than suffix based algorithms. This is because the shift advancements performed by suffix based solutions do not compensate the number of character inspections performed during each iteration. Thus, while prefix based algorithms maintain a linear behaviour which do not depend on the pattern length, suffix based solutions shown an increasing trend (or a slightly decreasing trend), while the length of the pattern increases, but with a very low performances on average. Specifically the HCAM algorithm obtains the best results only for $m \geq 16$, where it is approximately 10% slower than the EA algorithm, in the case of short patterns. However it remains always the best solution if compered with all other standard algorithm, with a gain from 11% to 35%. Among the suffix based solutions the BHCAM algorithm still remains the best choice in most cases, with a less sensible variance if compared with the HCAM algorithm.

6 Conclusions

In this paper we have introduced the heap-counting approach for the abelian pattern matching problem in strings and we have presented two new algorithms based on a prefix-based approach (HCAM) and on a suffix-based approach (BHCAM), respectively. We also presented two variants of these algorithms, based on a relaxed version of the heap-counting approach: the HFAM and BHFAM algorithms. From our experimental results it turns out that our approaches obtain good results when used for searching text over small alphabets, as the case of DNA sequences. The resulting algorithms turns out to be among the most effective in practical cases.

References

1. Amir, A., Apostolico, A., Landau, G.M., Satta, G.: Efficient text fingerprinting via Parikh mapping. J. Discrete Algorithms **1**(56), 409–421 (2003)
2. Baeza-Yates, R.A., Navarro, G.: New and faster filters for multiple approximate string matching. Random Struct. Algorithms **20**(1), 23–49 (2002)
3. Benson, G.: Composition alignment. In: Benson, G., Page, R.D.M. (eds.) WABI 2003. LNCS, vol. 2812, pp. 447–461. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39763-2_32
4. Böcker, S.: Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. Bioinformatics **23**(2), 5–12 (2007). https://doi.org/10.1093/bioinformatics/btl291
5. Böcker, S.: Sequencing from compomers: using mass spectrometry for DNA de novo sequencing of 200+ nt. J. Comput. Biol. **11**(6), 1110–1134 (2004)
6. Burcsi, P., Cicalese, F., Fici, G., Lipták, Z.: Algorithms for jumbled pattern matching in strings. Int. J. Found. Comput. Sci. **23**(2), 357–374 (2012)
7. Cantone, D., Cristofaro, S., Faro, S.: Efficient matching of biological sequences allowing for non-overlapping inversions. In: Giancarlo, R., Manzini, G. (eds.) CPM 2011. LNCS, vol. 6661, pp. 364–375. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21458-5_31
8. Cantone, D., Faro, S.: Efficient online Abelian pattern matching in strings by simulating reactive multi-automata. In: Holub, J., Zdarek, J. (eds.) Proceedings of the PSC 2014, pp. 30–42 (2014)
9. Chhabra, T., Ghuman, S.S., Tarhio, J.: Tuning algorithms for jumbled matching. In: Holub, J., Zdarek, J. (eds.) Proceedings of the PSC 2015, pp. 57–66 (2015)
10. Ejaz, E.: Abelian pattern matching in strings, Ph.D. thesis, Dortmund University of Technology (2010). http://d-nb.info/1007019956
11. Eres, R., Landau, G.M., Parida, L.: Permutation pattern discovery in biosequences. J. Comput. Biol. **11**(6), 1050–1060 (2004)
12. Faro, S., Lecroq, T., Borzì, S., Di Mauro, S., Maggio, A.: The string matching algorithms research tool. In: Proceeding of Stringology, pp. 99–111 (2016)
13. Ghuman, S.S., Tarhio, J.: Jumbled matching with SIMD. In: Holub, J., Zdarek, J. (eds.) Proceeding of the PSC 2016, pp. 114–124 (2016)
14. Ghuman, S.S.: Improved online algorithms for jumbled matching. Doctoral Dissertation 242/2017, Aalto University publication series, Aalto University, School of Science, Department of Computer Science (2017)
15. Grabowski, S., Faro, S., Giaquinta, E.: String matching with inversions and translocations in linear average time (most of the time). Inf. Process. Lett. **111**(11), 516–520 (2011)
16. Grossi, R., Luccio, F.: Simple and efficient string matching with k mismatches. Inf. Process. Lett. **33**(3), 113–120 (1989)
17. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. Exp. **10**(6), 501–506 (1980)
18. Jokinen, P., Tarhio, J., Ukkonen, E.: A comparison of approximate string matching algorithms. Softw. Pract. Exp. **26**(12), 1439–1458 (1996)
19. Navarro, G.: Multiple approximate string matching by counting. In: Baeza-Yates, R. (ed.) 1997 Proceeding of the 4th South American Workshop on String Processing, pp. 125–139 (1997)
20. Salomaa, A.: Counting (scattered) subwords. Bull. EATCS **81**, 165–179 (2003)