A fast pattern matching algorithm for highly similar sequences

Nadia Ben Nsira
Normandie University,
LITIS EA 4108,
Normastic FR3638,
IRIB, University of Rouen, France and
LaTICE,
University of Tunis El Manar, Tunisia
Email: nadia.bennsira@etu.univ-rouen.fr

Thierry Lecroq
Normandie University,
LITIS EA 4108,
Normastic FR3638,
IRIB, University of Rouen, France
Email: Thierry.Lecroq@univ-rouen.fr

Mourad Elloumi
LaTICE,
University of Tunis El Manar, Tunisia
Email: mourad.elloumi@gmail.com

***Abstract*—With the advent of NGS technologies there are more and more genomic sequences of individuals of the same species available. These sequences only differ by a very small amount. There is thus a strong need for efficient algorithms for performing fast pattern matching in such specific sets of sequences. In this paper we propose a very efficient algorithm that solves the on-line exact pattern matching problem in a set of highly similar DNA sequences. The algorithm we propose extends variants of the Boyer-Moore exact string matching algorithm. Experimental results show that our new algorithm exhibits the best performances in practice.**

I. Introduction

Since 2005 *Next Generation Sequencing* (NGS) technologies revolutionize the way to obtain genomic sequences. They allow to produce DNA sequences faster and cheaper than the traditional Sanger method. A direct consequence is that it is now possible to get a great number of sequences with a high degree of similarity. For instance, the 1000 genomes project[1] is aimed at producing a catalog of human genetic variations by sequencing a large amount of individual whole human genomes. This generates huge amounts of sequences (3 billion letters `A`, `C`, `G`, `T`) which are identical to more than 99% to the reference human genome. These sequences differ one from another by a few number of differences such as *substitutions* or *single nucleotide variants* (SNVs), *indels*, *copy number variations* (CNVs) or *translocations* to name a few.

There is thus a strong need for efficiently store, index and search in these data. In many situations, indexing these sequences will be possible. Thus pattern matching will be performed by using efficient and clever indexing data structures. However in some situations, indexing will not be possible (for instance due to a lack of storage space beyond the space necessary for storing the input sequences) and searching will need to be performed on-line. Thus it may be necessary to scan the whole sequences.

In this paper, we focus on offering an efficient solution that allows to find the exact occurrences of a given pattern of length m in a set of highly similar sequences. We propose a solution that follows a tight analysis of the Boyer-Moore exact string matching algorithm. We point out occurrences of the pattern by performing a left to right traversal over the reference sequence and at the same time we take into account variations contained in other sequences. Our approach makes a simplistic assumption that sequences include variations only of type *substitutions* and that there exists at most only one variation in a window of length m.

The rest of the paper is organized as follows. Section II presents related works. We set up notations and formalize the problem in Sect. III. In Sect. IV we give our new algorithm in Sect. V and its complexity analysis in Sect. VI. Experimental results are exhibited in Sect. VII. Finally we give our conclusions in Sect. VIII.

II. Related work

Recently there have been several works consisting in indexing sets of similar sequences. These works use the redundant information in order to reduce the memory space from the length of all the input sequences to the length of a single sequence called the reference sequence plus the number of variations between this reference sequence and all the other sequences.

Huang *et al.* [1] divide the input DNA sequences into common segments and non-common segments. If every sequence differs on m' positions from the reference, their data structure requires $O(n \log \sigma + m' \log m')$ bits where n is the length of the reference sequence and σ is the size of the alphabet. Their model is restricted to a specific type of similar sequences. Alatabbi *et al* [2] use the word level operations They build a suffix array together with an Aho-Corasick automaton [3] to store common segments and the non-common segments are converted into binary words using 2 bits per base. The use of the Aho-Corasick automaton induces a $\log n$ factor on the memory usage. In [4] a compressed index is proposed based on the Lempel-Ziv compression scheme [5]. Both [6] and [7] propose 2-level indexes for highly repetitive sequences. In [6] the authors implement an index based on suffix trees and traditional q-grams. The concept of the *suffix tree of alignment*

[1] http://www.1000genomes.org

978-1-4799-5669-2/14/$31.00 ©2014 IEEE

was proposed by [8]. It satisfies the same properties as the classical generalized suffix tree by adding a new one: common suffixes of two sequences are stored in an identical leaf. This result has been extended to the suffix array of an alignment [9].

All these results are concerned with off-line string matching but to the best of our knowledge there exists only one solution [10] for on-line string matching in a set of highly similar sequences. In this paper we will improve upon this first solution.

III. Preliminaries

In what follows, we consider a finite set Σ of σ symbols called an *alphabet*. In particular, Σ=$\{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$ for DNA sequences. A *string* or a *sequence* is a succession of zero or more symbols of the alphabet. The empty string is denoted by ε. The set of all non empty strings over Σ is denoted by Σ^+. All strings over the alphabet Σ are element of $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. The string w of length m is represented by $w[0\,..\,m-1]$ where $w[i] \in \Sigma$ and $0 \leq i \leq m-1$. The length of w is denoted by $|w|$.

A string x is a *factor* (substring) of y if there exist u and v such $y = uxv$, where $u, v, x, y \in \Sigma^*$. Let $0 \leq j \leq |y| - |x|$ be the starting position of x in y, thus $x = y[j\,..\,j + |x| - 1]$. A factor x is a *prefix* of y if $y = xv$, $v \in \Sigma^*$. Similarly a factor x is a *suffix* of y if $y = ux$, for $u \in \Sigma^*$. A factor u is a *border* of x, if it is both a prefix and a suffix of x; then there exist $v, w \in \Sigma^*$ such $x = vu = uw$. The longest common prefix and longest common suffix between two strings u and v are respectively denoted by *lcp*(u, v) and *lcsuff*(u, v). The Hamming distance between two strings u and v of the same length, denoted by *Ham*(u, v), is the number of positions where u and v have distinct symbols.

The exact pattern matching problem consists in finding all the occurrences of a pattern x of length m in a string y. That is, all possible j such that $x[i] = y[i + j]$ holds for all $0 \leq i \leq m-1$. This problem can be extended in a very interesting way by considering a set of sequences and find whether a given pattern occurs distributed horizontally where different parts of the pattern can be located in consecutive positions of different texts. More formally, given a set of r sequences $Y = \{y_0, y_1, \ldots, y_{r-1}\}$ of equal length n, point out all positions $0 \leq j \leq n - m + 1$, such that for $0 \leq i \leq m-1$ we have $x[i] = y_g[j + i]$ for some $g \in [0; r-1]$. This latter problem is known as distributed pattern matching [11].

Let M be a positive integer. From now on by variation we mean only substitution. A set of highly similar sequences is defined as follows.

Definition A set Y of r sequences $\{y_0, y_1, \ldots, y_{r-1}\}$ of the same length n over the alphabet Σ is a set of **highly similar sequences** if:

- y_0 is a reference sequence;
- the sequences $y_1, y_2, \ldots, y_{r-1}$ are represented by a set of variations over y_0: $Z = \{(\mathcal{G}, j, c) \mid c = y_g[j] \neq y_0[j]\}$ is the set of variations such that $1 \leq j \leq n-1$, $c \in \Sigma$ and $\mathcal{G} = \{g \mid 1 \leq g \leq r-1, c = y_g[j] \neq y_0[j]\} \neq \emptyset$;
- there does not exists in Z two triples with the same position: $\nexists (\mathcal{G}, j, c), (\mathcal{G}', j', c') \in Z$ such that $j = j'$;
- for every distinct $(\mathcal{G}, j, c), (\mathcal{G}', j', c') \in Z$ it holds that $|j - j'| > M$.

The set of highly similar sequences Y is represented by y_0 and Z. Note that several sequences can have the same variation at the same position. Furthermore, there is a restriction on the number of variations meaning that there can be at most one variation in a window of size M.

Our problem consists in finding all the occurrences of an arbitrary pattern x of length $m \leq M$ in the sequences of a set of highly similar sequences Y represented by y_0 and Z. This problem can be viewed as an hybrid between distributed pattern matching and approximate string matching with k mismatches [12].

A. The sliding window concept

When searching a given pattern x of length m in a string y, it is convenient to assume that the string y is examined with a *sliding window*. The window delimits the current factor in the string which is in general of the same length as x. It slides along the string y from beginning to end. When the sliding window is positioned on $y[j\,..\,j+m-1]$ a searching algorithm answers if an occurrence of x exists in the current window by comparing the pattern symbols with the aligned symbols of y in the window. This is called an **attempt** at position j. The window is first positioned on $y[0\,..\,m-1]$. Each attempt is followed by a shift (to the right) of the window until the right end of the window goes beyond the right end of the string y.

IV. The Extended Morris-Pratt method

We first recall the first algorithm [10] to solve the exact on-line searching in a set of highly similar sequences problem inspired from the Morris-Pratt (MP) algorithm [13].

It extends the *forward prefix scan* concept presented by MP (symbols of the sliding window are scanned from left to right at each attempt) and uses a sliding window mechanism on the set of highly similar sequences to scan the sequences from left to the right. For shifting the window, it considers borders at Hamming distance 0 (as in MP) and borders at Hamming distance 1. A general observation of the algorithm is as follows: given the pattern x of length m, it considers three cases when a prefix $x[0\,..\,i]$ for $0 \leq i \leq m-1$, is recognized when scanning the r sequences at position j:

1) $x[0\,..\,i] = y_0[j\,..\,j+i]$ and $\nexists (\mathcal{G}, k, c) \in Z$ such that $j \leq k \leq j+i$. This means that $x[0\,..\,i]$ matches on y_0 and there is no variation in all the other sequences in the current window. Then $x[0\,..\,i]$ matches equally in all sequences.
2) $x[0\,..\,i] = y_0[j\,..\,j+i]$ and $\exists (\mathcal{G}, k, c) \in Z$ such that $j \leq k \leq j+i$ meaning that there exists a variation in some sequences at a position between j and $j+i$ (Recall that there cannot exist two variations in a window of size $M \geq m$.) This also means that $x[0\,..\,i]$ matches all sequences except y_g with $g \in \mathcal{G}$.

3) $x[0\,..\,i] = y_g[j\,..\,j+i]$ and $\exists(\mathcal{G}, k, c) \in Z$ such that $j \leq k \leq j+i$ and $g \in \mathcal{G}$ at position k. Then $x[0\,..\,i]$ matches only sequences y_g with $g \in \mathcal{G}$.

If $x[i+1]$ does not match in any sequence of Y then a shift is performed using precomputed borders at Hamming distance at most 1 of $x[0\,..\,i]$ taking into account the three cases.

In [10] experiments show that this method is faster than applying a very fast exact single string matching algorithm on each sequence.

V. A NEW ALGORITHM

We now suggest a new algorithm called EXTENDEDFASTSEARCH that uses variants of the Boyer-Moore (BM) [14] algorithm along the same lines as the Fast-Search algorithm [15]. The Boyer-Moore algorithm is considered as one of the most efficient exact string matching algorithms (searching for all the exact occurrences of one pattern x in one string y). At each attempt it scans the symbols of the window from right to the left starting with its rightmost symbol. It shifts the current window to the right by using two precomputed arrays: the *good-suffix* and the *bad-character* shift arrays.

Let us consider an attempt at position j. In case of a mismatch between $x[i]$ and $y[j+i]$ (meaning that $x[i+1\,..\,m-1]$ matches $y[j+i+1\,..\,j+m-1]$):

- The bad-character shift consists in aligning the text symbol $y[j+i]$ with its rightmost occurrence in $x[0\,..\,m-1]$. If not present, then x can be safely shifted just past position $j+i$ of y.
- The good-suffix shift tries to align the factor $z = x[i+1\,..\,m-1] = y[j+i+1\,..\,j+m-1]$ with its rightmost occurrence in x that it is preceded by a symbol different from $x[i]$.

Note that the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the good-suffix shift and the bad-character shift.

The Fast-Search algorithm slightly modifies the Boyer-Moore algorithm. While comparing x with the current window $y[j\,..\,j+m-1]$ on y:

1) If a mismatch occurs immediately at the first symbol comparison, the shift is performed by using only a slightly modified bad-character shift: if $y[j+m-1] \neq x[m-1]$ the shift is computed in a way that the symbol $y[j+m-1]$ is aligned with its rightmost occurrence in $x[0\,..\,m-2]$. If such occurrence does not exist then the shift is performed just past the current window.
2) Otherwise, if a mismatch occurs at a position $i < m-1$ on x then the window is shifted by using only the good-suffix shift.

Extending the problem to the search for a pattern in a set of highly similar sequences, we need to consider re-occurrences of suffixes of the pattern with Hamming distance 0 and 1 for computing the good-suffix shift. We use the sliding window mechanism. We preserve the idea of bad-character shift when the mismatch occurs on the rightmost symbol of the window and consistently extend the good-suffix shift to our problem.

A. Searching phase

The EXTENDEDFASTSEARCH algorithm, given in Fig. 1, scans the given pattern going from right to left at each attempt.

The overview of the algorithm is as follows. While comparing a given pattern with the set of highly similar sequences:

- If a mismatch is encountered immediately on the rightmost symbol of the current window over the set of highly similar sequences (for that it needs to check in y_0 and in Z), then the algorithm performs a shift by the minimum value of the bad-character values of current mismatching symbols in the sequences.
- If a mismatch appears at some further comparison of the attempt, then the algorithm shifts safely by the minimum value between the classical good-suffix and a new good-suffix precomputed with pattern suffixes at Hamming distance 1.

Before detailing formally the procedure let us give some formal definitions.

As mentioned above shifts are precomputed according to positions and symbols in the given pattern.

The bad-character shifts are computed as in the BM algorithm and are stored in an array bc_x. For each symbol $c \in \Sigma$:

$$bc_x[c] = min\{0 \leq i < m \mid x[m-1-i] = c\} \cup \{m\}.$$

The good-suffix shifts are computed as in the BM algorithm and are stored in an array gs_x^0, defined for each position i by:

$$gs_x^0[i] = min\{k \quad | \quad x[i-k+1\,..\,m-1-k] = x[i+1\,..\,m-1] \text{ and } x[i-k] \neq x[i]\}.$$

This array gs_x^0 stores positions of the rightmost occurrences of suffixes of x at Hamming distance 0.

We now need to define a data structure for storing occurrences of suffixes of x at Hamming distance 1: the array gs_x^1 is defined for each position i by:

$$gs_x^1[i] = min\{k \quad | \quad Ham(\; x[i-k+1\,..\,m-1-k],\; x[i+1\,..\,m-1]) = 1 \text{ and } x[i-k] \neq x[i]\}.$$

Let us consider an attempt at position j over the set of similar sequences. The values of gs_x^0 and gs_x^1 satisfy two conditions to perform correct and safe shifts. If a mismatch at position i of x or a complete match of the pattern is encountered, then the first condition ensures that the recognized factor $v = y_0[j+i+1\,..\,j+m-1]$ (involving the case when the factor is recognized in some sequence(s) of the set Z) is aligned with its rightmost occurrence as a factor in the pattern if it is present. If such an occurrence is not present then the shift aligns the longest prefix of v with a matching prefix of x (still taking into account variations stored in Z). The second condition ensures that after shifting the symbol $b = y[j+i]$ is aligned with a symbol c that is different from $x[i]$ (that was just aligned with b before shifting). So if a mismatch occurs at position i in the pattern, then $gs_x^0[i]$ and $gs_x^1[i]$ shifts are precomputed in such a way to hold both conditions.

When scanning the pattern x from right to left such that j is the current window position over the sequences:

- If $x[m-1] \neq y_0[j+m-1]$ and $x[m-1] \neq c$ for all $c \in \Sigma$ such that $(\mathcal{G}, j+m-1, c) \in Z$. Then the algorithm shifts by $min\{bc_x[y_0[j+m-1]], bc_x[c]\}$.
- If $x[i+1\,..\,m-1]$ matches some sequences and $x[i] \neq y_0[j+i]$ with $0 \leq i \leq m-2$ and $x[i] \neq c$ for all $c \in \Sigma$ such that $(\mathcal{G}, j+m-1, c) \in Z$. Then the length of the shift of the window is given by $min\{gs_x^0[i], gs_x^1[i]\}$.

The EXTENDEDFASTSEARCH algorithm is described by the pseudo-code given in Fig. 1. The inputs of the algorithm are:

- the pattern x and its length m;
- the reference sequence y_0 and its length n;
- the list Z of variations in the other sequences, sorted in increasing order of the second components which are the positions on the sequences.

The fact that the list Z is stored in increasing order of the positions of the variations enables to access it through a window of bounded size during the two loops in lines 12-14 and 7-9 of algorithm EXTENDEDFASTSEARCH.

```
EXTENDEDFASTSEARCH(x, m, y_0, n, Z = {(G, j, c)})
▷ Input: m = |x| > 0, n = |y_0| > m, Z
▷ Output: Positions of occurrences of x in the set of sequences
▷ Precondition: There exists at most one position of
▷                variation in any window of length m
1  Begin
2    bc_x ← PREBMBC(x, m)
3    gs_x^0 ← PREBMGS(x, m)
4    gs_x^1 ← PREBMGSBIS(x, m)
5    j ← m − 1
6    while j < n − m do
7      while j < n − m and
           min{bc_x[y_0[j]], bc_x[c] ∀(G, j, c) ∈ Z} > 0 do
8        j ← j + min{bc_x[y_0[j]], bc_x[c] ∀(G, j, c) ∈ Z}
9      end while
10     if j < n − m then
11       i ← m − 2
12       while i ≥ 0 and (x[i] = y_0[i + j] or
             ∃(G, i + j, x[i]) ∈ Z) do
13         i ← i − 1
14       end while
15       if i < 0 then
16         OUTPUT(j)
17         i ← 0
18       end if
19       j ← j + min{gs_x^0[i], gs_x^1[i]}
20     end if
21   end while
22 End
```

Fig. 1. The searching algorithm.

B. Preprocessing phase

The preprocessing phase is independent from the sequences of Y and it consists in computing arrays storing the length of the shifts: bc_x for each symbol in the alphabet and gs_x^0 and gs_x^1 for each position in the pattern.

We consider in this section the preprocessing that shall undergo the pattern for computing the array gs_x^1. The reader is referred to [16] for details on PREBMBC(x, m) and PREBMGS(x, m) algorithms for computing bc_x and gs_x^0 arrays respectively.

For computing the array gs_x^1, we use the array $suff_x^1$ given by the following expression, for each $0 \leq i < m-1$:

$$suff_x^1[i] = \max\{\ell \mid Ham(\ x[i-\ell+1\,..\,i],\ x[m-\ell\,..\,m-1]) = 1\}.$$

This means that $suff_x^1[i]$ is the maximal length of suffixes of x with Hamming distance 1 that occur at the right position i on x.

Let us consider the array $suff_x^0$ such that: $suff_x^0[i] = lcsuff(x[0\,..\,i], x)$. The arrays $suff_x^0$ and $suff_x^1$ are computed as detailed in the pseudo-code given in Fig. 2.

```
SUFFIXESBIS(x, m)
▷ Input: m = |x| ≥ 0
▷ Output: suff_x^0, suff_x^1
1  Begin
2    suff_x^0[m − 1] ← m
3    g ← m − 1
4    for i ← m − 2 downto 0 step 1 do
5      if i > g and suff_x^0[i + m − 1 − f] ≠ i − g then
6        suff_x^0[i] ← min(suff_x^0[i + m − 1 − f], i − g)
7      else if i < g then
8        g ← i
9      end if
10     f ← i
11     while g ≥ 0 and x[g] = x[g + m − 1 − f] do
12       g ← g − 1
13     end while
14     suff_x^0[i] ← f − g
15     ℓ ← ℓ − suff_x^0[i] − 1
16     while j ≥ 0 and x[ℓ] = x[m − 1 − i + ℓ] do
17       ℓ ← ℓ − 1
18     end while
19     if ℓ ≠ i − suff_x^0[i] − 1 then
20       suff_x^1[i] ← i − ℓ
21     else suff_x^1[i] ← 0
22     end if
23   end if
24   end for
25   return suff_x^1
26 End
```

Fig. 2. Computation of $suff_x^1$ and $suff_x^0$.

We can now formulate the algorithm PREBMGSBIS(x, m) that computes gs_x^1 with the help of $suff_x^1$. It uses the following

relation:

$$gs_x^1[m-1-suff_x^1[i]] \leq m-1-i$$

for $0 \leq i \leq m-2$. The algorithm, depicted in Fig. 3 considers the values of i in increasing order (Lines 14 to 16). The previous part (Lines 5 to 13) deals with borders at Hamming distance 1 of x (detected with $suff_x^1[i] = i+1$).

```
PREBMGSBIS(x, m, suff_x^1)
  ▷ Input: m = |x| ≤ 0, suff_x^1
  ▷ Output: gs_x^1
    Begin
2     ▷ Initialization
2     for i ← 0 to m − 1 do
3     |  gs_x^1[i] ← m
4     end for
      ▷ Borders
5     k ← 0
6     for i ← m − 2 downto −1 step 1 do
7     |  if i = −1 or suff_x^1[i] = i + 1 then
8     |  |  while k < m − 1 − i do
9     |  |  |  gs_x^1[k] ← m − 1 − i
10    |  |  |  k ← k + 1
11    |  |  end while
12    |  end if
13    end for
      ▷ Re-occurrences of suffixes
14    for i ← 0 to m − 2 do
15    |  gs_x^1[m − 1 − suff_x^1[i]] ← m − 1 − i
16    end for
17    return gs_x^1
18  End
```

Fig. 3. Computation of gs_x^1 using $suff_x^1$.

VI. Complexity analysis

It turns out that the preprocessing phase, presented in this paper, is quadratic with respect to the length m of the given pattern x.

Proposition 1: Let σ be the size of the alphabet. The computation of bc_x can be done in $O(m+\sigma)$ time and space. Algorithm SUFFIXESBIS(x, m) computes $suff_x^0$ and $suff_x^1$ in $O(m^2)$ time and $O(m)$ space. The computation of gs_x^0 and gs_x^1 can be done in $O(m)$ time and space given $suff_x^0$ and $suff_x^1$. Then the precomputing algorithms applied to a given pattern of length m run in $O(m^2+\sigma)$ time and require $O(m+\sigma)$ extra space.

The computation of the array $suff_x^1$ requires to know the longest common suffix between any arbitrary pairs of position inside the pattern x. This can be done in linear time either by using a suffix tree data structure processed in order to answer lowest common ancestor (LCA) queries in constant time or by using a suffix array data structure enhanced with a longest common prefix array processed to answer range minimum queries (RMQ) in constant time. In practice suffix array and RMQs are faster [17]. However, for our purpose, building data structures such as suffix trees or enhanced suffix arrays for the pattern will be slower than algorithm SUFFIXESBIS(x, m).

The searching phase is cubic in the worst case.

Proposition 2: The algorithm EXTENDEDFASTSEARCH(x, m, y_0, n, Z) finds all the occurrences of x in the r sequences of Y represented by y_0 and Z in $O(mnr)$ time.

In the best case EXTENDEDFASTSEARCH(x, m, y_0, n, Z) can run in $O(n/m)$ time. We will now see that despite its very pessimistic worst case time complexity, it performs very well in practice.

VII. Experimental study

We compared four algorithms in terms of running times:

- An efficient exact single string matching algorithm namely FJS algorithm [18] that performs well for string matching in DNA sequences [19]. We ran it on each sequence successively one by one and then computed the sum of the obtained running times.
- A naive algorithm called EXTENDEDNAIVE that performs symbol comparisons from left to right at each attempt without any memory from one attempt to the next and applies a shift of length 1 after each attempt.
- Algorithm EXTENDEDMP [10].
- Algorithm EXTENDEDFASTSEARCH.

All algorithms have been consistently implemented in the C programming language and were run on a machine with 12 GB RAM and 4-core CPU with 2.27 GHz.

Note that all the algorithms implement the usual trick of adding a copy of x at the end of y_0 as a sentinel in order to test the end of the text only when an occurrence of x is found thus saving a lot of time over all the attempts.

One input sequence of length 150 MB have been randomly built with KISS [20] on DNA alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Then variations have been randomly generated every 500 positions. In practice, the set Z is implemented as an array with 3 lines (position of the variation j, symbol of the variation c and the set of sequences $\mathcal{G}$ involving the variation). This array is sorted in increasing order of the variation positions.

The tests consist in searching individually 100 patterns, randomly generated from the reference sequence for each pattern length, in the sets of simulated sequences. We then computed the average running times. We performed our tests with patterns of length 8, 16, 64 and 128 on sets Y from 2 to 6 sequences.

Results are shown in Fig. 4 to 7. The x-axis represents the number of sequences in Y while the y-axis represents the running times.

These results, on simulated data and with the given hardware settings, show that the EXTENDEDFASTSEARCH algorithm is faster than all the other algorithms for the 4 pattern lengths even for only two sequences (including the reference sequence).

Noteworthy, the curve given in Fig. 8 have a decreasing-line shape. It measure the running times of the algorithm EXTENDEDFASTSEARCH on different pattern lengths with 6

sequences in the set Y. As expected the algorithm speeds up as the patterns get longer since it performs longer shifts.

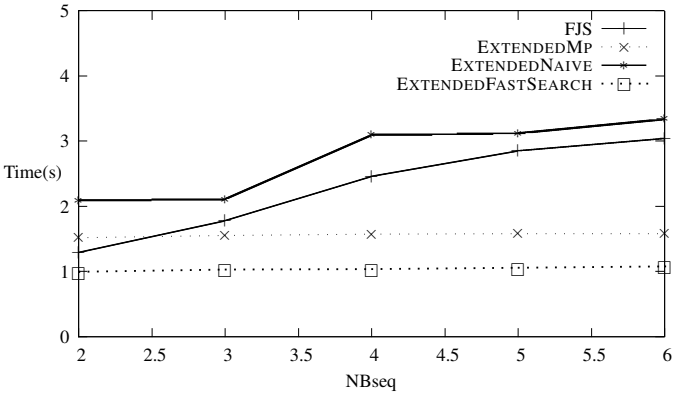

Fig. 4. Experimental results on patterns of length 8.

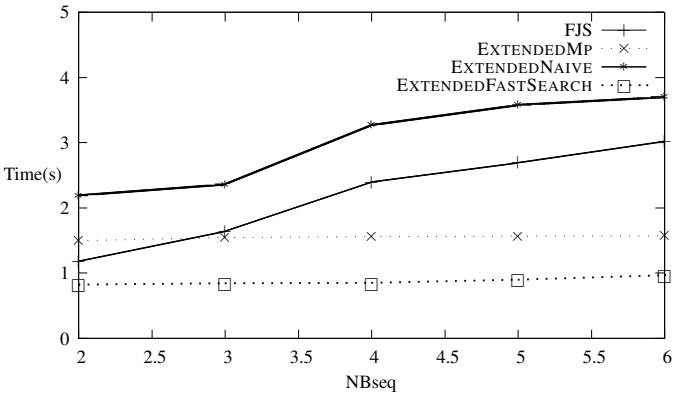

Fig. 5. Experimental results on patterns of length 16.

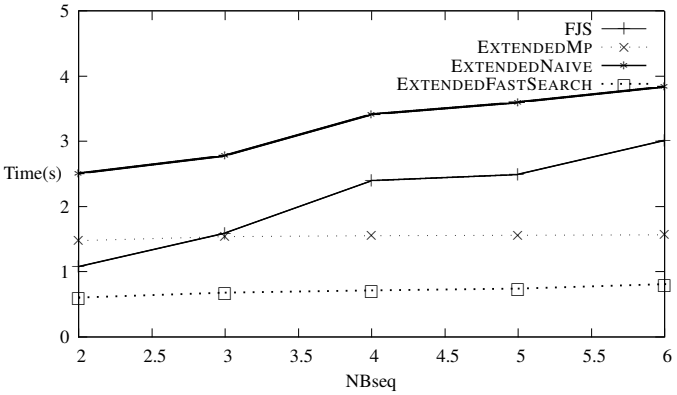

Fig. 6. Experimental results on patterns of length 64.

VIII. Conclusion

We have presented a new efficient algorithm called ExtendedFastSearch algorithm that extends variants of the Boyer-Moore exact pattern matching algorithm to solve the exact pattern matching problem on a set of highly similar sequences. This new algorithm outperforms existing methods moreover it gets more efficient when the pattern length increases.

However, our solution works with a particular model. First it is limited to a certain gap between consecutive variations. We will have to improve our algorithm to overcome this point. The next steps include to take into account any kind of variation between the reference sequence and the other sequences (not only substitutions) and to be able to perform approximate pattern matching. Then eventually we will have to consider pattern matching in compressed sequences. On a practical point of view this work constitutes an important step in order to perform pattern matching using the CRAM format [21].

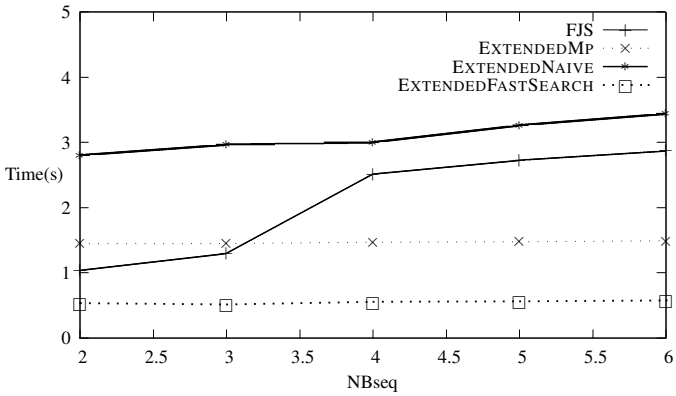

Fig. 7. Experimental results on patterns of length 128.

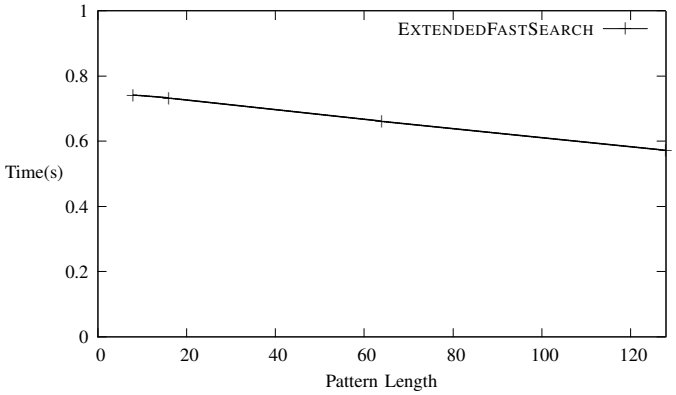

Fig. 8. Experimental results of ExtendedFastSearch algorithm for 6 sequences.

Acknowledgment

The authors thank the anonymous reviewers that greatly improved a first version of this paper.

References

[1] S. Huang, T. W. Lam, W.-K. Sung, S.-L. Tam, and S.-M. Yiu, "Indexing similar DNA sequences," in *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM*

2010), ser. Lecture Notes in Computer Science, B. Chen, Ed. Weihai, China: Springer, 2010, vol. 6124, pp. 180–190.

[2] A. Alatabbi, C. Barton, C. S. Iliopoulos, and L. Mouchard, "Querying highly similar structured sequences via binary encoding and word level operations," in *Proceedings of the International Workshop Artificial Intelligence Applications and Innovations (AIAI 2012) Part II*, ser. IFIP Advances in Information and Communication Technology, L. S. Iliadis, I. Maglogiannis, H. Papadopoulos, K. Karatzas, and S. Sioutas, Eds., vol. 382. Halkidiki, Greece: Springer, 2012, pp. 584–592.

[3] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[4] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung, "Fast relative Lempel-Ziv self-index for similar sequences," *Theoretical Computer Science*, vol. 532, pp. 14–30, 2014.

[5] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval," in *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE 2010)*, ser. Lecture Notes in Computer Science, E. Chávez and S. Lonardi, Eds., vol. 6393. Los Cabos, Mexico: Springer, 2010, pp. 201–206.

[6] X. Cao, S. C. Li, and A. K. H. Tung, "Indexing DNA sequences using q-grams," in *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA 2005)*, ser. Lecture Notes in Computer Science, L. Zhou, B. C. Ooi, and X. Meng, Eds., vol. 3453. Beijing, China: Springer, 2005, pp. 4–16.

[7] A. Alatabbi, C. Barton, and C. S. Iliopoulos, "On the repetitive collection indexing problem," in *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBM 2012)*, 2012, pp. 682–687.

[8] J. C. Na, H. Park, M. Crochemore, J. Holub, C. S. Iliopoulos, L. Mouchard, and K. Park, "Suffix tree of an alignment: An efficient index for similar data," in *Revised Selected Papers of the 24th International Workshop On Combinatorial Algorithms (IWOCA 2013)*, ser. Lecture Notes in Computer Science, T. Lecroq and L. Mouchard, Eds., vol. 8288. Rouen, France: Springer, 2013.

[9] J. C. Na, H. Park, S. Lee, M. Hong, T. Lecroq, L. Mouchard, and K. Park, "Suffix array of alignment: A practical index for similar data," in *Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE 2013)*, ser. Lecture Notes in Computer Science, M. L. Oren Kurland and E. Porat, Eds., vol. 8214. Jerusalem, Israel: Springer, 2013, pp. 243–254.

[10] N. Ben Nsira, T. Lecroq, and M. Elloumi, "On-line string matching in highly similar DNA sequences," in *Proceedings of the 2nd International Conference on Algorithms for Big Data*, ser. CEUR Workshop Proceedings, C. S. Iliopoulos and A. Langiu, Eds., vol. 1146. Palermo, Italy: CEUR-WS.org, 2014, pp. 16–22.

[11] C. S. Iliopoulos, L. Mouchard, and M. S. Rahman, "A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching," *Mathematics in Computer Science*, vol. 1, no. 4, pp. 557–569, 2008.

[12] A. Amir, M. Lewenstein, and E. Porat, "Faster algorithms for string matching with k mismatches," *Journal of Algorithms*, vol. 50, no. 2, pp. 257–275, 2004.

[13] J. H. Morris, Jr and V. R. Pratt, "A linear pattern-matching algorithm," University of California, Berkeley, Report 40, 1970.

[14] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[15] D. Cantone and S. Faro, "Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm," *Journal of Automata, Languages and Combinatorics*, vol. 10, no. 5/6, pp. 589–608, 2005.

[16] M. Crochemore, C. Hancart, and T. Lecroq, *Algorithms on strings*. Cambridge University Press, 2007.

[17] J. Fischer and V. Heun, "Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE," in *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, ser. Lecture Notes in Computer Science, M. Lewenstein and G. Valiente, Eds., vol. 4009. Barcelona, Spain: Springer, 2006, pp. 36–48.

[18] F. Franek, C. G. Jennings, and W. F. Smyth, "A simple fast hybrid pattern-matching algorithm," *Journal of Discrete Algorithms*, vol. 5, no. 4, pp. 682–695, 2007.

[19] S. Faro and T. Lecroq, "The exact online string matching problem: a review of the most recent results," *ACM Computing Surveys*, vol. 45, no. 2, p. 13, 2013.

[20] G. Marsaglia and A. Zaman, "The kiss generator," Tech. rep., Department of Statistics, University of Florida, Tech. Rep., 1993.

[21] M.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney, "Efficient storage of high throughput DNA sequencing data using reference-based compression," *Genome Research*, vol. 21, pp. 734–740, 2011.