Math.Comput.Sci. (2017) 11:113–126
DOI 10.1007/s11786-016-0280-2

Mathematics in Computer Science

On-line String Matching in Highly Similar DNA Sequences

Nadia Ben Nsira · Mourad Elloumi · Thierry Lecroq

Received: 22 April 2014 / Revised: 4 March 2015 / Accepted: 10 April 2015 / Published online: 2 February 2017

Abstract We consider the problem of on-line exact string matching of a pattern in a set of highly similar sequences. This can be useful in cases where indexing the sequences is not feasible. We present a preliminary study by restricting the problem for a specific case where we adapt the classical Morris-Pratt and Knuth-Morris-Pratt algorithms to consider borders with errors. We give an original algorithm for computing borders at Hamming distance 1. We exhibit experimental results showing that our algorithms are much faster than searching for the pattern in each sequences with a very fast on-line exact string matching algorithm.

Keywords Algorithm design · Algorithms on strings · Pattern matching · String matching · String · Border

Mathematics Subject Classification 68W32

1 Introduction

High-throughput sequencing or *Next Generation Sequencing* (NGS) technologies allow to produce a great amount of DNA sequences with a high rate of similarity. For instance, the 1000 genomes project[1] aimed at sequencing a large amount of individual whole human genomes. This generates massive amounts of sequences (3 billion letters A, C, G, T) which are identical more than 99% to the reference human genome. The generated data form a collection of sequences where each differs 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.

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

N. B. Nsira · M. Elloumi
LaTICE, University of Tunis El Manar, Tunis, Tunisia
e-mail: nadia.bennsira@etu.univ-rouen.fr

M. Elloumi
e-mail: mourad.elloumi@gmail.com

N. B. Nsira · T. Lecroq (✉)
LITIS EA 4108, Normastic FR3638, IRIB, University of Rouen, Normandie University, Rouen, France
e-mail: Thierry.Lecroq@univ-rouen.fr
URL: monge.univ-mlv.fr/~lecroq/lec_en.html

With the large mass of available data, storing, indexing and support for fast pattern matching have become important research topics.

Pattern matching can be carried out in two ways: off-line by using an index or on-line when indexing is not possible. Although the first kind of solutions seems to be more suitable, the index issue might not seem significant in some cases even if it is compressed. The main problem we can face is to have insufficient storage space to build the index. Thus one may have to scan the whole sequence rather than index it.

In this paper, we focus on offering a preliminary study that allows to find the exact occurrences of a given pattern in a set of highly similar sequences. We propose two solutions that follows a tight analysis of the Morris-Pratt [16] and Knuth-Morris-Pratt [6] algorithms. 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 where m is the length of the pattern.

The rest of the paper is organized as follows. Section 2 presents related works. We set up notations and formalize the problem in Sect. 3. We give an extension of the Morris-Pratt algorithm in Sect. 4. An extension of the Knuth-Morris-Pratt algorithm is given in Sect. 5 while experimental results are exhibited in Sect. 6. Finally we give our conclusions in Sect. 7.

2 Related Works

Storing genetic sequences of many individual of the same species is a major challenge in biological research. Basic structures usually store redundant information which lead to a memory requirement proportional to the total length of input data. Recently, several works focusing on indexing similar sequences were implemented to allow building data structures taking advantage from high similarity between the considered data. These works aim to reduce the memory requirement from the length of all input sequences to the length of a single sequence (reference sequence) plus the number of variations.

Huang et al. [12] propose a solution which assumes that the input set of DNA sequences can be divided into common segments and non-common segments. Their solution assumes that every sequence differs on m' positions from the reference and the designed 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. Though this data structure greatly reduces the memory usage and allows fast pattern matching, the adopted model is restricted to a specific type of similar sequences. In [2] a solution based on the use of word level operations on bit vectors is presented. In a similar way as in [12], the general scheme of this technique stores the entire of a reference sequence with only differences between the remaining sequences. The authors build a suffix array together with an Aho-Corasick automaton [1] to store identical segments and the non-common segments are converted into a binary word using 2 bits per base. Due to the use of the Aho-Corasick the memory usage depends on a $\log n$ factor. In [7] a compressed index is proposed based on the Lempel-Ziv compression scheme [15]. Both [4,13] propose two level indexes for highly repetitive sequences. In [4] the authors implement an index based on suffix tree and traditional q-grams. The concept of the *suffix tree of alignment* was proposed by [17]. 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 [18].

All these results are concerned with off-line string matching. To the best of our knowledge there exists only one recent solution for on-line string matching in a set of highly similar sequences, called *journaled stirng tree (JST)* provided by Rahn et al. [21]. In brief, the approach: (1) extends on-line algorithms based on sequential scans of sequences while exploiting the high similarity: all sequences are simultaneously scanned from left to right instead of iterating them sequentially, (2) succinct representation that gives a referentially compressed version of sequences so-called *journaled strings*: each reference position at which at least one sequence has a variation is stored and a bitvector is used to denote which sequences contain the variation. The provided datatype *JST* can be traversed similarly to a simple for-loop over all sequences. The search algorithm requires, as well as classical

on-line algorithms, information about the length of a fixed-size window of last seen characters called *context*, i.e when scanning a sequence its current state depends on the context. During the traversal the method constructs the required sequence contexts using the bitvectors and respective journaled strings and presents them to the algorithm. The algorithm can, in addition, signal after processing whether it needs the information for which sequences the presented sequence context is valid, i.e. when a string matching algorithm found a match. The results showed a better performance and memory consumptions compared with the sequential processing of the same number of sequences.

3 Preliminaries

In what follows, we consider a finite *alphabet* $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$ 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 $0 \leq j \leq |y| - |x| + 1$ 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 *reverse* of the string x is denoted by $x^{\sim}$. The longest common prefix between two strings u and v is denoted by $lcp(u, v)$. The array $pref_x^0$ is the array of prefixes of the pattern x such that $pref_x^0[i]$ stores the length of the longest prefix of x starting at position i, for $0 \leq i \leq m-1$: $pref_x^0[i] = |lcp(x, x[i\,..\,m-1])|$.

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. Formally

$$Ham(u, v) = card\{k \mid 0 \leq k < |u| = |v| \text{ and } u[k] \neq v[k]\}.$$

The exact pattern matching problem consists in finding all the occurrences of a pattern x of length m in a string y of length n: that is, all possible $0 \leq j \leq n - m + 1$ such that $y[i + j] = x[i]$ 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 sequences $Y = \{y_0, \ldots, y_{r-1}\}$ 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 [14].

The problem we focus on in this paper is formally defined as follows. Let $y_0, y_1, \ldots, y_{r-1}$ be r highly similar sequences with the same length n defined over the alphabet Σ. Let y_0 be the reference sequence of length n. The sequences $y_1, y_2, \ldots, y_{r-1}$ are represented by variations over y_0. Thus, we consider the set $Z = \{(g, j, c)\}$, such that $c = y_g[j] \neq y_0[j]$ for all $0 \leq j \leq n-1$ where $1 \leq g \leq r-1$ and $c \in \Sigma$. Furthermore, for $(g, j, c), (g', j', c') \in Z$ we have either that $j = j'$ and $g \neq g'$ or that $|j - j'| > M$ for some integer M. We wish to find all occurrences of an arbitrary pattern x of length $m \leq M$ in y_g where $0 \leq g \leq r - 1$. This problem can be viewed as an hybrid between distributed pattern matching and approximate string matching with k mismatches [3]. This is closely related to the CRAM format reported in [11].

4 Extension of the Morris-Pratt

We offer an algorithm to solve the problem described above in the same fashion as the Morris-Pratt (MP) algorithm [16] using a sliding window mechanism to scan the sequences. Hence we need to preprocess the query pattern before the searching phase. We adopt the same strategy of *forward prefix scan* presented by MP by extending the problem to the search in highly similar data. For that we need to consider borders at Hamming distance 0 (as in MP) and borders at Hamming distance 1.

Given the pattern x of length m, we consider 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:

Case 1 $x[0\,..\,i] = y_0[j\,..\,j+i]$ and $\nexists(g,k,c) \in Z$ 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.

Case 2 $x[0\,..\,i] = y_0[j\,..\,j+i]$ and $\exists(g,k,c) \in Z$ such that $j \leq k \leq j+i$. This means that $x[0\,..\,i]$ matches all sequences except y_g.

Case 3 $x[0\,..\,i] = y_g[j\,..\,j+i]$ and $\exists(g,k,c) \in Z$ such that $j \leq k \leq j+i$. Then $x[0\,..\,i]$ matches only sequence(s) y_g.

Borders at Hamming distance 0 are computed as in the MP algorithm and stored in an array called *mpNext*. For borders at Hamming distance 1 we will use an array called B: for each $0 \leq i \leq m-1$, $B[i]$ contains the suffix starting positions of borders of $x[0\,..\,i]$ with one substitution. More formally $B[i] = \{i' \mid Ham(x[0\,..\,i-i'], x[i'\,..\,i]) = 1\}$.

4.1 Searching Phase

The searching phase consists in scanning the sequences from beginning to the end. A general situation is the following: a prefix $x[0\,..\,i]$ of the pattern matches at least one sequence of Z at right position j. Then position $i+j+1$ is scanned and a decision has to be made in accordance with the three cases (Fig. 1).

Case 1 If $x[i+1] = y_0[j+1]$ then if there is no variation at position $j+i+1$ in the other sequences we remain in Case 1 otherwise we move to Case 2. If $x[i+1] \neq y_0[j+1]$ then if there is no variation in the other sequences at position $j+1$ then a shift is performed as in the MP algorithm otherwise if the variant symbol

```
SEARCH(x, m, y_0, n, Z = {(g, j_p, c_p)}, mpNext, pref_x^0, B)
  ▷ Input: |x| > 0, |y_0| > m, Z
  ▷ Output: right position of occurrences of x
  ▷ Auxiliary: case = 1 , i = 0, j = 0
1 Begin
2   while j < n do
3     if case = 1 then
4       if ∃{(g, j_p, c_p)} ∈ Z | j_p = j then
5         while i > −1 and y_0[j] ≠ x[i] and c_p ≠ x[i]) do
6           i ← mpNext[i]
7         end while
8         if i > −1 then
9           i_p ← i
10          if x[i] = y_0[j] then
11            case ← 2
12          else case ← 3
13          end if
14        end if
15      else while i > −1 and y_0[j] ≠ x[i] do
16          i ← mpNext
17        end while
18      end if
19    else while i > −1 and y_0[j] ≠ x[i] do
20        (i, case) ← SHIFT(x, i, j, case, mpNext, pref_x^0, B, i_p, c_p, j_p)
21      end while
22    end if
23    i ← i + 1
24    if i = m then
25      OUTPUT(j − i)
26      (i, case) ← SHIFT(x, i, j, case, mpNext, pref_x^0, B, i_p, c_p, j_p)
27    end if
28    j ← j + 1
29  end while
30 End
```

Fig. 1 Algorithm searching for a query pattern x in a set $Z = \{y_0, \ldots, y_{r-1}\}$

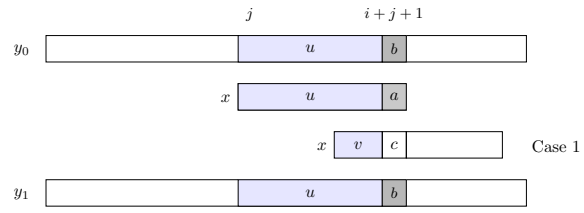

Fig. 2 Shift in Case 1

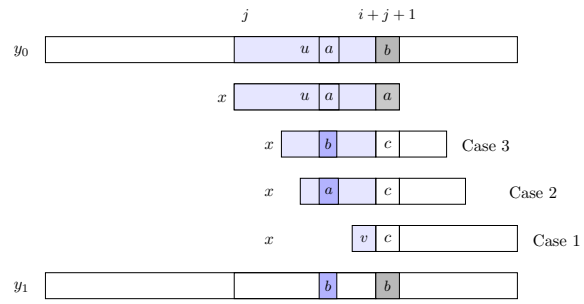

Fig. 3 Shift in Case 2

c is equal to $x[i+1]$ we move to Case 3 otherwise a shift has to be performed using *mpNext* and B (see Fig. 2).

Case 2 If $x[i+1] = y_0[j+1]$ then we remain in Case 2. If $x[i+1] \neq y_0[j+1]$ then a shift has to be performed using B (see Fig. 3).

Case 3 If $x[i+1] = y_0[j+1]$ then we remain in Case 3. If $x[i+1] \neq y_0[j+1]$ then a shift has to be performed using B (see Fig. 4).

When a shift is performed using B, the case can change according to the length of the shift and the position of the variation.

The algorithm SEARCH given in Fig. 1 describes formally the searching phase and the transition from one case to another. At the beginning the *case* is set to 1. When comparing $x[i]$ with the facing character $y_0[j]$, we check if there exists $(g, j, c_p) \in Z$ where $c_p = y_g[j]$. If this is true line 9 keeps the position of the variation in x in a variable i_p, then lines 10–12 update the *case* to 2 if $c_p \neq x[i]$ or to 3 otherwise. The loop in lines 5–6 performs a shift by $mpNext[i]$ either if $x[i]$ does not correspond to the current character in y_0 or to the current variation c_p or if the shift by *mpNext* returns -1. If there is no variation in Z that contains the current position in y_0 and $y_0[j] \neq x[i]$, then we shift by $mpNext[i]$ in lines 15–16. Once the value of the *case* is updated in lines 19–21 or if an occurrence of x is reported in lines 24–26 we compute a new valid shift according to the values of i_p, j_p, c_p (verify the condition of Proposition 2 and the situation in Fig. 8 below, thus to verify if a new border can be computed). Note that we consider the minimal position that satisfies the condition of Proposition 2, in order to perform a safe shift and to avoid missing any occurrence of x.

Theorem 1 *The algorithm* SEARCH *runs in* $O(n)$ *time.*

Proof Assuming that elements of Z are stored in increasing order of the second components which are the positions on the sequences and assuming that each shift is performed in constant time, the complexity of the algorithm SEARCH is dominated by the loops in lines 5, 15 and 19 that perform the symbol comparisons. Along the same line as for the MP algorithm, when a symbol at position $0 \leq j \leq n$ of the sequences is processed it is either a match or a mismatch. When it is a match it is never processed again. There can thus be at most n matches. When it is a mismatch a shift is performed. There can be at most n shifts and thus at most n mismatches. Altogether this gives a linear bound on the number of comparisons.

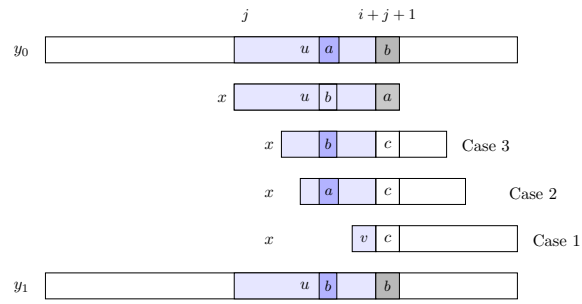

Fig. 4 Shift in Case 3

4.2 Preprocessing Phase

The preprocessing phase consists in computing the arrays used for computing the shifts at the end of each attempt during the searching phase. These computations only depend on the pattern x and can thus be performed prior to the searching phase. These arrays store positions of borders for each prefix of the pattern. Borders at Hamming distance 0 are computed as in the MP algorithm and stored in an array called *mpNext*. For computing borders at Hamming distance 1 that are stored in the array B, we use the array $pref_x^0$ and the following $pref_x^{\sim}$ array.

The array $pref_x^{\sim}$ stores, for each position i, the length of the longest common prefix starting at position i when reading the pattern from right to left at each position $i' < i$ where

$$lcp(x[0\,..\,i]^{\sim}, x[0\,..\,i']^{\sim}) \neq 0.$$

It is defined for all $0 \leq i \leq m-1$ and $i' < i$ by

$$pref_x^{\sim}[i] = \{(i', \ell) \mid i' < i, x[i] = x[i'] \text{ and } 0 < \ell = |lcp(x[0\,..\,i]^{\sim},\ x[0\,..\,i']^{\sim}|)\}.$$

Then $pref_x^{\sim}[i+1]$ can be easily computed from $pref_x^{\sim}[i]$ as follows (see Fig. 7):

$$pref_x^{\sim}[i+1] = \{(i',\ \ell+1) \mid x[i'] = x[i+1] \text{ and } \exists (i'-1) \in pref_x^{\sim}[i]\} \cup \{(i', 1) \mid \nexists (i'-1) \in pref_x^{\sim}[i]\}.$$

From the definition above one can notice that $pref_x^{\sim}$ is computed based on occurrences of letters. Along with this need we use arrays where we store positions of occurrences of letters:

- $last[c]$ stores the rightest occurrence of c in x, $\forall c \in \Sigma$;
- $prev[k]$ stores the position of the previous occurrence of the letter $x[k]$ on the left.

An example of arrays *prev* and *last* is given in Fig. 6. The algorithm ComputePrev given Fig. 5 computes array *last* and *prev*.

Recall that $0 \leq i \leq m$, $B[i]$ contains the suffix starting positions of borders of $x[0\,..\,i]$ with one substitution. The next proposition shows how to compute the array B using arrays $pref_x^0$ and $pref_x^{\sim}$ (Fig. 7).

Proposition 2 *For* $1 \leq i \leq m-1$, *if* $\exists (i-i', \ell) \in pref_x^{\sim}[i]$ *and* $i - i' = pref_x^0[i'] + \ell$ *and* $i - i' < i' + pref_x^0[i']$ *then* $x[0\,..\,i-i']$ *is a border of* $x[0\,..\,i' + pref_x^0[i'] - 1]x[pref_x^0[i']]x[i' + pref_x^0[i'] + 1\,..\,i]$.

Proof Since $i - i' < i' + pref_x^0[i']$ then $x[0\,..\,i-i']$ is a prefix of $x[0\,..\,i' + pref_x^0[i'] - 1]$ and thus of $x[0\,..\,i' + pref_x^0[i'] - 1]x[pref_x^0[i']]x[i' + pref_x^0[i'] + 1\,..\,i]$. By definition of $pref_x^0[i']$ we have $x[i'\,..\,i' + pref_x^0[i'] - 1] = x[0\,..\,pref_x^0[i'] - 1]$ and by definition of $pref_x^{\sim}[i]$ and by the fact that $\ell = i - i' - pref_x^0[i']$ we have $x[i' + pref_x^0[i'] + 1\,..\,i] = x[pref_x^0[i'] + 1\,..\,i - i']$. Thus $x[0\,..\,i-i']$ is a suffix of $x[0\,..\,i' + pref_x^0[i'] - 1]x[pref_x^0[i']]x[i' + pref_x^0[i'] + 1\,..\,i]$. Thus $x[0\,..\,i-i']$ is a border of $x[0\,..\,i' + pref_x^0[i'] - 1]x[pref_x^0[i']]x[i' + pref_x^0[i'] + 1\,..\,i]$.

```
ComputePrev(x, m)
 ▷ Input: |x| > 0
 ▷ Output: The array prev
 1  Begin
 2     prev[0] ← −1
 3     last[c] ← −1,  ∀c ∈ Σ
 4     last[x[0]] ← 0
 5     for i ← 1 to m − 1 do
 6     |  prev[i] ← last[x[i]]
 7     |  last[x[i]] ← i
 8     end for
 9     return prev
10  End
```

Fig. 5 Algorithm computing arrays *last* and *prev*

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$x[k]$	A	T	A	A	T	A	G	A	C	A	A	T	A	C
$prev[k]$	−1	−1	0	2	1	3	−1	5	−1	7	9	4	10	8

c	A	C	G	T
$last[c]$	12	13	6	11

Fig. 6 Arrays *last* and *prev* for $x =$ ATAATAGACAATAC and $\Sigma =$ {A, C, G, T}

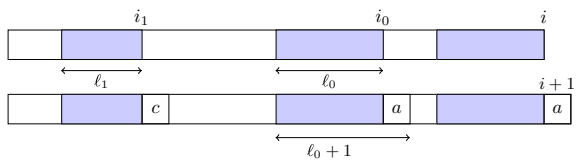

Fig. 7 Relation for computing $pref_x^{\sim}[i+1]$ from $pref_x^{\sim}[i]$: $(i_0, \ell_0), (i_1, \ell_1) \in pref_x^{\sim}[i] \mid x[i_0+1] = x[i+1]$ and $x[i_1+1] \neq x[i+1]$, then $(i_0+1, \ell_0+1) \in pref_x^{\sim}[i+1]$

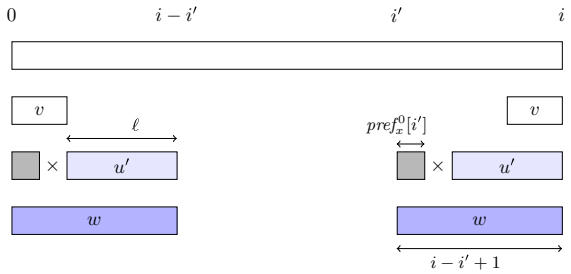

Fig. 8 Situation of Proposition 2: v is the longest border at Hamming distance 0 of $x[0\,..\,i]$ and w is a longer border at Hamming distance 1

Figure 8 illustrates the situation of Proposition 2 and an example is shown in Fig. 9.

Then $B[i] = \{i' \mid \exists (i-i', \ell) \in pref_x^{\sim}[i]$ and $i - i' = pref_x^0[i'] + \ell\} \cup \{i \mid pref_x^0[i] = 0\}$. An example is depicted in Fig. 10.

Array B is computed by the algorithm ComputeB given in Fig. 11. Since elements of the array $pref_x^{\sim}$ can be computed incrementally ($pref_x^{\sim}[i]$ can be computed only with $pref_x^{\sim}[i-1]$ and since $B[i]$ depends on $pref_x^{\sim}[i]$ and not on $pref_x^{\sim}[j]$, for $j < i$ with $1 \leq i \leq m-1$, elements of $pref_x^{\sim}$ are not stored only $pref_x^{\sim}[i-1]$ and $pref_x^{\sim}[i]$ are stored in variables P_1 and P_2 respectively. P_1 and P_2 are lists of pairs: $p.pos$ gives the first element of a pair p, $p.length$ gives the second element of a pair p, $first(P)$ returns the first element of the list P, $next(P)$ removes

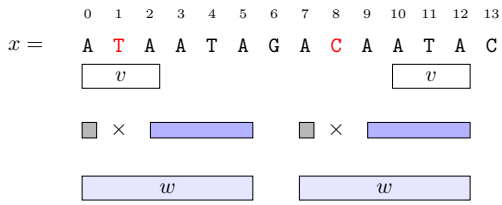

Fig. 9 Example illustrating the result of Proposition 2: ATA is the longest border at Hamming distance 0 of ATAATAGACAATA and ATAATA is a longer border at Hamming distance 1

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$x[i]$	A	T	A	A	T	A	G	A	C	A	A	T	A	C
$pref_x^0[i]$	13	0	1	3	0	1	0	1	0	1	3	0	1	0
$pref_x^{\sim}[i]$			L_2	L_3	L_4	L_5	$\emptyset$	L_7	$\emptyset$	L_9	L_{10}	L_{11}	L_{12}	L_{13}
$B[i]$	$\emptyset$	$\{1\}$	$\emptyset$	$\emptyset$	$\{4\}$	$\emptyset$	$\{6\}$	$\{5\}$	$\{8\}$	$\{7\}$	$\{7\}$	$\{7,11\}$	$\{7\}$	$\{13\}$

where

$L_2 = \{(0,1)\}$, $L_3 = \{(2,1)(0,1)\}$, $L_4 = \{(1,2)\}$, $L_5 = \{(3,1),(2,3),(0,1)\}$,
$L_7 = \{(5,1)(3,1),(2,1),(0,1)\}$, $L_9 = \{(7,1),(5,1)(3,1),(2,1),(0,1)\}$,
$L_{10} = \{(9,1),(7,1),(5,1)(3,2),(2,1),(0,1)\}$, $L_{11} = \{(4,3),(1,2)\}$,
$L_{12} = \{(10,1),(9,1),(7,1),(5,4)(3,1),(2,3),(0,1)\}$, $L_{13} = \{(8,2)\}$.

Fig. 10 Arrays $pref_x^0$, $pref_x^{\sim}$ and B for $x =$ ATAATAGACAATAC

the next element of a list P and $P.append(p)$ adds the pair p at the end of the list P. Lines 6–14 and 23 compute elements of the array $pref_x^{\sim}[i]$ according to the relation described in Fig. 7. Lines 15-16 check if the condition of Proposition 2 is verified for the position i. If it is then the corresponding position is added to $B[i]$. In the case where $pref_x^0[i] = 0$, i is added to $B[i]$, since it is trivial that if we substitute $x[i]$ to $x[0]$ we obtain a new border of the prefix $x[0\,..\,i]$ at Hamming distance 1 of length $1 > pref_x^0[i] = 0$.

Proposition 3 *The algorithm ComputeB computes the array B in $O(m^2)$ time.*

Proof The loops in lines 6–22 scans all the elements in the array $pref_x^{\sim}$. Since the total number of elements in the array $pref_x^{\sim}$ is $O(m^2)$ the time complexity of the algorithm *ComputeB* is also $O(m^2)$.

The algorithm SHIFT can now be presented in Fig. 12. It operates in constant time.

5 Extension of Knuth-Morris-Pratt Algorithm

In the classical exact string matching problem when searching for all the occurrences of a single pattern x of length m in a single sequence y of length n, in the case of a mismatch between pattern symbol $x[i]$ and sequence symbol $y[i+j]$ when scanning the window from left to right, the Knuth-Morris-Pratt (KMP) algorithm performs a shift according to the longest border of $x[0\,..\,i-1]$ followed by a symbol different from $x[i]$. For that it uses a precomputed array kmp_x^0 defined as follows:

$kmp_x^0[i] = \max\{j \mid x[0\,..\,j-1]$ is a border of $x[0\,..\,i-1]$ and $x[j] \neq x[i]\} \cup \{-1\}$

(see Fig. 13).

```
ComputeB(x, m, prev, pref_x^0)
 1  for i ← 1 to m − 1 do
 2  |  j ← prev[i]
 3  |  P_1 ← ∅
 4  |  P_2 ← ∅
 5  |  while j ≠ −1 do
 6  |  |  while P_1 ≠ ∅ and first(P_1).pos >= j do
 7  |  |  |  P_1 ← next(P_1)
 8  |  |  end while
 9  |  |  if P_1 ≠ ∅ and first(P_1).pos = j − 1 then
10  |  |  |  ℓ ← first(P_1).length + 1
11  |  |  else ℓ ← 1
12  |  |  end if
13  |  |  P_2 ← P_2.append((j, ℓ))
14  |  |  if pref_x^0[i − j] = j − ℓ and 2j < i + pref_x^0[i − j] then
15  |  |  |  B[i] ← B[i] ∪ {i − j}
16  |  |  end if
17  |  |  if pref_x^0[i] = 0 then
18  |  |  |  B[i] ← B[i] ∪ {i}
19  |  |  end if
20  |  |  j ← prev[j]
21  |  end while
22  |  P_1 ← P_2
23  end for
24  return B
```

Fig. 11 Algorithm computing array B

```
Shift(x, i, j, case, mpNext, pref_x^0, B, i_p, c_p, j_p)
▷ Input: the latest updated value of case in search phase, last retrieved values of i_p, c_p, j_p
 1  Begin
 2    if case = 1 then
 3    |  i ← mpNext[i]
 4    else if ∃i′ ∈ B[i − 1] | i′ + pref_x^0[i′] = i_p and x[pref_x^0[i′]] = c_p then
 5    |  |  if i − i′ > mpNext[i] then
 6    |  |  |  case ← 3
 7    |  |  |  i ← i − i′
 8    |  |  else i ← mpNext[i]
 9    |  |  |  if j − i > j_p then
10    |  |  |  |  case ← 1
11    |  |  |  end if
12    |  |  end if
13    |  else i ← mpNext[i]
14    |  |  if j − i > j_p then
15    |  |  |  case ← 1
16    |  |  end if
17    |  end if
18    end if
19    return (i, case)
20  End
```

Fig. 12 Algorithm computing a shift at the end of each attempt

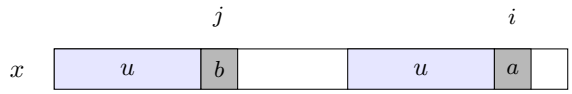

Fig. 13 $kmp_x^0[i] = \max\{j \mid x[0\,..\,j-1]$ is a border of $x[0\,..\,i-1]$ and $x[j] \neq x[i]\} \cup \{-1\}$

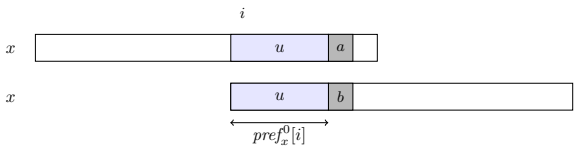

Fig. 14 $kmp_x^0[i + pref_x^0[i]] \geq pref_x^0[i]$

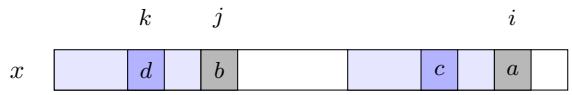

Fig. 15 $kmp_x^1[i] = \{(k, j) \mid x[0\,..\,j-1]$ is a border at Hamming distance 1 of $x[0\,..\,i-1]$, $x[k] \neq x[i-j+k]$ and $x[j] \neq x[i]\}$

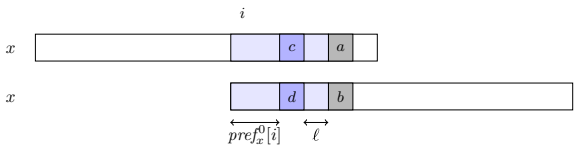

Fig. 16 $(pref_x^0[i], pref_x^0[i] + 1 + \ell) \in kmp_x^1[i + pref_x^0[i] + 1 + \ell]$ where $\ell = |lcp(x[pref_x^0[i] + 1], x[i + pref_x^0[i] + 1])|$

The values of the array can be easily computed with the help of the array $pref_x^0$ using the following relation:

$$kmp_x^0[i + pref_x^0[i]] \geq pref_x^0[i]$$

(see Fig. 14).

For searching a pattern in a set of highly similar sequences we need to consider borders at Hamming distance 0 as in the KMP algorithm and borders at Hamming distance 1. For that we use an array called kmp_x^1 defined as follows:

$$kmp_x^1[i] = \{(k, j) \mid x[0\,..\,j-1] \text{ is a border at Hamming distance 1 of } x[0\,..\,i-1], \\ x[k] \neq x[i-j+k] \text{ and } x[j] \neq x[i]\}$$

(see Fig. 15).

The values of this array can be computed using the following relation:

$$(pref_x^0[i], pref_x^0[i] + 1 + \ell) \in kmp_x^1[i + pref_x^0[i] + 1 + \ell]$$

where $\ell = |lcp(x[pref_x^0[i] + 1], x[i + pref_x^0[i] + 1])|$.

According to the relation defined above and Fig. 16, we need to answer to Longest Common Prefix (LCP) queries. These queries can be answered in constant time using:

- either a suffix tree of x and Lowest Common Ancestor queries;
- or a suffix array of x and Range Minimun Queries.

For our purpose, instead of building a costly data structure, we choose to extend the algorithm PREFIXES(x, m) of [5] to the case of prefixes at Hamming distance 1. We will built an array $pref_x^1$ that stores the length of the

```
PrefixesBis(x, m)
▷ Input: m = |x| ≥ 0
▷ Output: pref_x^0, pref_x^1
Begin
  pref_x^0[0] ← m
  g ← 0
  for i ← 1 to m − 1 do
    if i < g and pref_x^0[i − f] ≠ g − i then
      pref_x^0[i] ← min(pref_x^0[i − f], g − i)
    else if i > g then
      g ← i
    end if
    f ← i
    while x[g] = x[g − f] do
      g ← g + 1
    end while
    pref_x^0[i] ← g − f
    j ← pref_x^0[i] + 1
    while i + j ≤ m and x[j] = x[i + j] do
      j ← j + 1
    end while
    if j ≠ pref_x^0[i] + 1 then
      pref_x^1[i] ← j
    else pref_x^1[i] ← 0
    end if
  end if
  end for
  return pref_x^1
End
```

Fig. 17 Computation of $pref_x^1$ and $pref_x^0$

longest common prefix at Hamming distance 1 starting at each position of the pattern x. The array $pref_x^1$ is defined as follows, for each $0 < i \leq m-1$:

$$pref_x^1[i] = \max\{\ell \mid Ham(x[0\,..\,\ell-1], x[i\,..\,i+\ell-1]) = 1\}$$

or equivalently,

$$pref_x^1[i] = pref_x^0[i] + 1 + |lcp(x[pref_x^0[i]+1], x[i + pref_x^0[i]+1])|.$$

The reader is referred to [5] for details on Prefixes(x, m) algorithm for computing $pref_x^0$. The algorithm PrefixesBis(x, m) in Fig. 17 is an adaptation of Prefixes(x, m) that computes both the arrays $pref_x^0$ and $pref_x^1$.

Then kmp_x^1 can be computed using arrays $pref_x^0$ and $pref_x^1$ with the help of the following lemma.

Lemma 4 $(pref_x^0[i], pref_x^1[i]) \in kmp_x^1[i + pref_x^1[i]]$ *for* $1 \leq i \leq m$.

Proof or $1 \leq i \leq m$, by definition of $pref_x^0$ we have $x[pref_x^0[i]] \neq x[i + pref_x^0[i]]$ and by definition of $pref_x^1[i]$ we have $Ham(x[0\,..\,pref_x^1[i]-1], x[i\,..\,i + pref_x^1[i]-1]) = 1$ and $x[pref_x^1[i]] \neq x[i + pref_x^1[i]]$ thus $(pref_x^0[i], pref_x^1[i]) \in kmp_x^1[i + pref_x^1[i]]$.

```
PreKmpBis(x, m, pref_x^0, pref_x^1)
▷ Input: m = |x| ≤ 0, pref_x^0, pref_x^1
▷ Output: kmp_x^1
Begin
    ▷ Initialization
    for i ← 0 to m do
    |  kmp_x^1[i] ← ∅
    end for
    ▷ Borders at Hamming distance 1
    for i ← m − 1 to 1 do
    |  kmp_x^1[i + pref_x^1[i]] ← kmp_x^1[i + pref_x^1[i]] ∪ (pref_x^0[i], pref_x^1[i])
    end for
    return kmp_x^1
End
```

Fig. 18 Computation of kmp_x^1 using $pref_x^1$

The algorithm PreKmpBis$(x, m, pref_x^0, pref_x^1)$ in Fig. 18 computes the arrays kmp_x^1.

Proposition 5 *The algorithm* PreKmpBis$(x, m, pref_x^0, pref_x^1)$ *computes the array* kmp_x^1 *in time and space* $O(m)$ *for a given string* x *of length* m *and arrays* $pref_x^0$ *and* $pref_x^1$,

Proof The correctness comes from Lemma 4. The time complexity comes from the fact that the two loops run $O(m)$ times and that all the other instructions run in constant time. The space complexity comes from the fact that apart from x and the arrays $pref_x^0$, $pref_x^1$ and kmp_x^1 the algorithm only needs variable i.

We have the following result.

Theorem 6 *The algorithm* Search *using* kmp_x^1 *instead of mpNext,* $pref_x^0$ *and* B *runs in* $O(n)$ *time.*

Proof The proof is similar to the one of Theorem 1.

6 Experiments

We compared our solution with the FJS algorithm [10] which is one of the most efficient exact string matching algorithm in this setting (see [9]) and with a naive algorithm performing shifts of length 1.

We performed experiments on two kinds of data: pseudo-random data and real data. For the patterns, we randomly selected them from the reference text with varying length from 8 to 128. For each pattern length, we repeated tests 100 times and computed the average searching time. Experiments were conducted on a machine with 12 GB RAM and 4-core CPU with 2.27 GHz.

One input sequence of length 150 MB have been pseudo-randomly built with the WELL [19] generator on DNA alphabet {A, C, G, T}. Then variations have been pseudo-randomly generated every 500 positions with WELL. In practice, the set Z is implemented as an array with 3 lines (position of the variation, symbol of the variation and the set of sequences involving the variation). This array is sorted in increasing order of the variation positions.

For real data we used as a reference sequence the human chromosome 7 from the build hg19 (Human Genome version 19). It has around 152Mbp. For the variations, we used all the SNV (homozygous and heterozygous) from the exomes of five patients suffering from autosomal dominant early-onset Alzheimer disease [20] discarding indels.

As mentioned above our algorithm takes into account the reference sequence with positions of variations. We ran the FJS algorithm on each sequence successively. Our goal is to demonstrate that from a certain number of sequences, it is more efficient to use our solution than a classical exact string matching solution. Results are shown in Figs. 19 and 20 only for pseudo-random data when searching for a pattern of length 8 and for real data when searching for a pattern of length 64. The results in the other tested cases are similar. These results show that the behaviour of the algorithms is similar between pseudo-random data and real date and that our solutions are faster

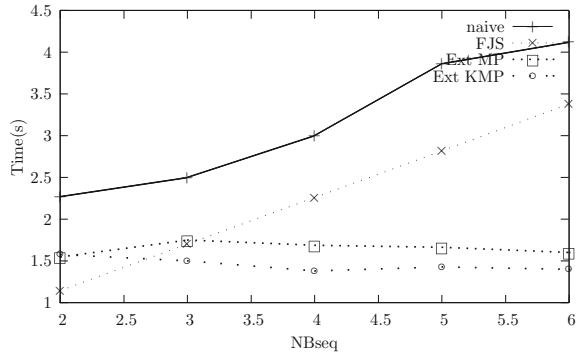

Fig. 19 Experimental results on pseudo-random data when searching for a pattern of length 8

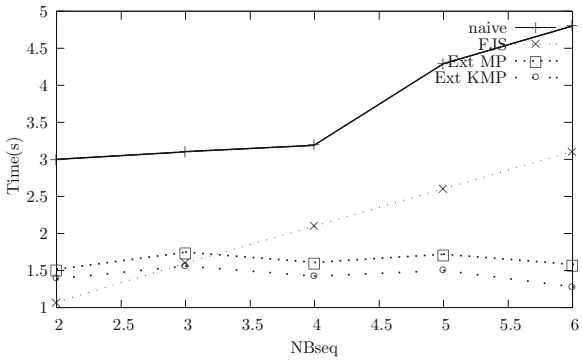

Fig. 20 Experimental results on real data when searching for a pattern of length 64

(in our settings) when considering more that 3 highly similar sequences. These results also show that the extension of the KMP algorithm is faster than the extension of the MP algorithm.

7 Discussion and Conclusions

We presented two new algorithms for searching a single pattern in a set of similar sequences. The design of the algorithm follows a tight analysis of the Morris-Pratt and Knuth-Morris-Pratt algorithms. Recall that we use a suitable representation of data such that we take into account a reference sequence with only positions of variations of the other sequences. The searching time complexity of our algorithms depends on the size of the reference text. However, our solutions work with a particular model, since we are limited to a certain gap between consecutive variations. Further research directions include to overcome this limitation. On a practical point of view it would be interested to use variants of the Backward-Oracle-Matching algorithm [8] for greater efficiency. The next steps include to take into account any kind of variation between the reference sequence and the other sequences, to be able to perform approximate pattern matching and to consider performing such pattern matching in similar sequences in a compressed form.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Alatabbi, A., Barton, C., Iliopoulos, C.S., Mouchard, L.: Querying highly similar structured sequences via binary encoding and word level operations. In: Iliadis, L.S., Maglogiannis, I., Papadopoulos, H., Karatzas, K., Sioutas, S. (eds.) Proceedings of the

International Workshop Artificial Intelligence Applications and Innovations (AIAI 2012) Part II, IFIP Advances in Information and Communication Technology, vol. 382, pp. 584–592. Springer, Halkidiki, Greece (2012)
3. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with *k* mismatches. J. Algorithms **50**(2), 257–275 (2004)
4. Cao, X., Li, S.C., Tung, A.K.H.: Indexing DNA sequences using q-grams. In: Zhou, L., Ooi, B.C., Meng, X. (eds.) Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA 2005), Lecture Notes in Computer Science, vol. 3453, pp. 4–16. Springer, Beijing, China (2005)
5. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)
6. Knuth, D.E., Morris, J.H., Jr., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
7. Do, H.H., Jansson, J., Sadakane, K., Sung, W.-K.: Fast relative Lempel-Ziv self-index for similar sequences. Theor. Comput. Sci. **532**, 14–30 (2014)
8. Faro, S., Lecroq, T.: Efficient variants of the Backward-Oracle-Matching algorithm. In: Holub, J., Zdàrek, J. (eds.) Proceedings of the Prague Stringology Conference 2008, pp. 146–160. Czech Republic, Prague (2008)
9. Faro, S., Lecroq, T.: The exact online string matching problem: a review of the most recent results. ACM Comput. Surv. **45**(2), 13 (2013)
10. Franek, F., Jennings, C.G., Smyth, W.F.: A simple fast hybrid pattern-matching algorithm. J. Discrete Algorithms **5**(4), 682–695 (2007)
11. Fritz, M.-Y., Leinonen, R., Cochrane, G., Birney, E.: Efficient storage of high throughput DNA sequencing data using reference-based compression. Genome Res. **21**, 734–740 (2011)
12. Huang, S., Lam, T.W., Sung, W.-K., Tam, S.-L., Yiu, S.-M.: Indexing similar DNA sequences. In: Chen, B. (ed.) Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM 2010), Lecture Notes in Computer Science, vol. 6124, pp. 180–190. Springer, Weihai, China (2010)
13. Iliopoulos, C.S., Alatabbi, A., Barton, C.: On the repetitive collection indexing problem. In: Proceedings of the 2012 IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW), pp. 682–687. IEEE Computer Society (2012)
14. Iliopoulos, C.S., Mouchard, L., Rahman, M.S.: A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching. Math. Comput. Sci. **1**(4), 557–569 (2008)
15. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In: Chávez, E., Lonardi, S. (eds.) Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE 2010), Lecture Notes in Computer Science, vol. 6393, pp. 201–206. Springer, Los Cabos, Mexico (2010)
16. Morris, J.H., Jr, Pratt, V.R.: A linear pattern-matching algorithm. Report 40, University of California, Berkeley (1970)
17. Na, J.C., Park, H., Crochemore, M., Holub, J., Iliopoulos, C.S., Mouchard, L., Park, K.: Suffix tree of an alignment: an efficient index for similar data. In: Lecroq, T., Mouchard, L., (eds.) Revised Selected Papers of the 24th International Workshop On Combinatorial Algorithms (IWOCA 2013), vol. 8288 of Lecture Notes in Computer Science, Rouen, France, Springer (2013)
18. Na, J.C., Park, H., Lee, S., Hong, M., Lecroq, T., Mouchard, L., Park, K.: Suffix array of alignment: a practical index for similar data. In: Oren Kurland, M.L., Porat, E. (eds.) Proceedings of the 20th International Symposium on String Processing and Information Retrieval (SPIRE 2013), Lecture Notes in Computer Science, vol. 8214, pp. 243–254. Springer, Jerusalem, Israel (2013)
19. Panneton, F., L'Ecuyer, P., Matsumoto, M.: Improved long-period generators based on linear recurrences modulo 2. ACM Trans. Math. Softw. **32**(1), 1–16 (2006)
20. Pottier, C., Hannequin, D., Coutant, S., Rovelet-Lecrux, A., Wallon, D., Paquet, C., Bombois, S., Pariente, J., Thomas-Anterion, C., Michon, A., Croisile, B., Etcharry-Bouyx, F., Berr, C., Dartigues, J.-F., Amouyel, P., Dauchel, H., Boutoleau-Bretonnière, C., Frebourg, T., Lambert, J.C., Campion, D., PHRC GMAJ collaborators: High frequency of potentially causative SORL1 mutations in autosomal dominant early-onset alzheimer disease. Mol. Psychiatry **17**(9), 875–879 (2012)
21. Rahn, R., Weese, D., Reinert, K.: Journaled string tree—a scalable data structure for analyzing thousands of similar genomes on your laptop. Bioinformatics **30**(24), 3499–3505 (2014)