On-line pattern matching on similar texts

Roberto Grossi[1], Costas S. Iliopoulos[2], Chang Liu[3], Nadia Pisanti[4], Solon P. Pissis[5], Ahmad Retha[6], Giovanna Rosone[7], Fatima Vayani[8], and Luca Versari[9]

1 **Department of Computer Science, University of Pisa, Italy and ERABLE Team, INRIA**
grossi@di.unipi.it

2 **Department of Informatics, King's College London, UK**
c.iliopoulos@kcl.ac.uk

3 **Department of Informatics, King's College London, UK**
chang.2.liu@kcl.ac.uk

4 **Department of Computer Science, University of Pisa, Italy and ERABLE Team, INRIA**
pisanti@di.unipi.it

5 **Department of Informatics, King's College London, UK**
solon.pissis@kcl.ac.uk

6 **Department of Informatics, King's College London, UK**
ahmad.retha@kcl.ac.uk

7 **Department of Computer Science, University of Pisa, Italy and Department of Mathematical and Computer Science, University of Palermo, Italy**
giovanna.rosone@unipi.it

8 **Department of Informatics, King's College London, UK**
fatima.vayani@kcl.ac.uk

9 **Scuola Normale Superiore, Pisa, Italy**
luca.versari@sns.it

Abstract

Pattern matching on a set of similar texts has received much attention, especially recently, mainly due to its application in cataloguing human genetic variation. In particular, many different algorithms have been proposed for the *off-line* version of this problem; that is, constructing a compressed index for a set of similar texts in order to answer pattern matching queries efficiently. However, the *on-line*, more fundamental, version of this problem is a rather undeveloped topic. Solutions to the on-line version can be beneficial for a number of reasons; for instance, efficient on-line solutions can be used in combination with partial indexes as practical trade-offs. We make here an attempt to close this gap via proposing two efficient algorithms for this problem. Notably, one of the algorithms requires time linear in the size of the texts' representation, for short patterns. Furthermore, experimental results confirm our theoretical findings in practical terms.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases string algorithms, pattern matching, degenerate strings, elastic-degenerate strings, on-line algorithms

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.07

28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017).
Editors: Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter; Article No. 07; pp. 07:1–07:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

It is possible to represent closely related sequences that have been aligned using a multiple sequence alignment (MSA) algorithm into one compacted form, that is able to represent the non-polymorphic sites (columns) of the MSA, as well as the polymorphic ones [10]. This representation compresses maximal sequences of non-polymorphic sites, while the polymorphic ones, containing substitutions, insertions, and deletions of letters, are represented as a set containing all possible variants observed at that location. Consider, for instance, the following:

```
ATGCAACGGGTA--TTTTA
ATGCAACGGGTATATTTTA
ATGCACCTGG----TTTTA
```

These sequences can be compacted into a single string $\tilde{T}$ containing some deterministic and some *non-deterministic* segments. Note that a non-deterministic segment is a finite set of deterministic strings and may contain an empty string ε corresponding to a deletion. The total number of segments is the *length* of $\tilde{T}$ and the total number of letters is the *size* of $\tilde{T}$.

$$\tilde{T} = \{\texttt{ATGCA}\} \cdot \left\{\begin{matrix}\texttt{A}\\ \texttt{C}\end{matrix}\right\} \cdot \{\texttt{C}\} \cdot \left\{\begin{matrix}\texttt{G}\\ \texttt{T}\end{matrix}\right\} \cdot \{\texttt{GG}\} \cdot \left\{\begin{matrix}\texttt{TA}\\ \texttt{TATA}\\ \varepsilon\end{matrix}\right\} \cdot \{\texttt{TTTTA}\}$$

This representation has been defined in [11] as an *elastic-degenerate* text. The natural problem that arises is finding all matches of a deterministic pattern P in text $\tilde{T}$. We call this the Elastic-Degenerate String Matching (*EDSM*) problem. The simplest version of this problem assumes that a degenerate segment can contain only single letters [9].

An elastic-degenerate text can represent, for example, a set of closely-related DNA sequences. For instance, a *pan-genome* [18, 24, 12, 21] is a reference sequence which is not just a single genome, but the result of an MSA of several of them that share large consensus regions and also exhibit differences at some positions. Recently, various data structures to store pan-genomes have been suggested [8, 4]. In particular, due to the application of cataloguing human genetic variation [23], there has been ample work in the literature on the *off-line* (indexing) version of the pattern matching problem [10, 14, 22, 15, 16]. In literature, there are also algorithms and applications for the problem of inferring motifs from degenerate input texts [20, 19]. However, to the best of our knowledge, the *on-line*, more fundamental, version of the *EDSM* problem has not been studied as much as indexing approaches. Solutions to the on-line version can be beneficial for a number of reasons: **(a)** efficient on-line solutions can be used in combination with partial indexes as practical trade-offs; **(b)** efficient on-line solutions for exact pattern matching can be applied for fast average-case approximate pattern matching, similar to standard strings [3]; **(c)** on-line solutions can be useful when one wants to search for a set of patterns in elastic-degenerate texts, similar to standard strings [1, 2].

Our Contributions. Let us denote by m the length of pattern P, by n the length of $\tilde{T}$, and by $N > m$ the size of $\tilde{T}$ (see Section 2 for definitions). In [11], an algorithm for solving the *EDSM* problem in time $\mathcal{O}(\alpha\gamma mn + N)$ and space $\mathcal{O}(N)$ was presented; where α and γ are parameters, respectively representing the maximum number of strings in any degenerate segment of the text and the maximum number of degenerate segments spanned by any occurrence of the pattern in the text. In this paper, we improve the state-of-the-art; we present two new algorithms to solve the same problem in an on-line manner. The first one requires time $\mathcal{O}(nm^2 + N)$ after a preprocessing stage with time and space $\mathcal{O}(m)$; the second algorithm requires time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ after a preprocessing stage with time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$, where w is the size of the computer word in the RAM model. Thus, the second algorithm requires time linear in the size of the texts' representation, for short patterns. Finally, we present experiments confirming our theoretical findings in practical terms.

2 Definitions

We begin with a few definitions, generally following [5]. An *alphabet* Σ is a non-empty finite set of letters of size $|\Sigma|$. A *(deterministic) string* on a given alphabet Σ is a finite sequence of letters of Σ. For this work, we assume that the alphabet is fixed, i.e. $|\Sigma| = \mathcal{O}(1)$. The *length* of a string x is denoted by $|x|$. For two positions i and j on x, we denote by $x[i\,..\,j] = x[i]\,..\,x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j (it is empty if $j < i$), and by ε we denote the *empty string*. The set of all strings on an alphabet Σ (including the empty string ε) is denoted by Σ^*. For any string $y = uxv$, where u and v are strings, if $u = \varepsilon$ then x is a *prefix* of y. Similarly, if $v = \varepsilon$ then x is a *suffix* of y. We say that x is a *proper factor* (resp. prefix/suffix) of y if x is a factor (resp. prefix/suffix) of y distinct from y. By $\mathcal{B}_{u,v}$ we denote the set containing all indices i, such that the prefix $u[0\,..\,i]$ of string u is also a suffix of string v.

▶ **Example 1.** Suppose we have two strings $u = \texttt{ATATG}$ and $v = \texttt{CATAT}$. Then $\mathcal{B}_{u,v} = \{1, 3\}$ because of prefix/suffix `AT` and prefix/suffix `ATAT`, respectively.

An *elastic-degenerate string* (ED string) $\tilde{X} = \tilde{X}[0]\tilde{X}[1]\ldots\tilde{X}[n-1]$, of *length* n, on an alphabet Σ, is a finite sequence of n *degenerate letters*. Every *degenerate letter* $\tilde{X}[i]$, for all $0 \leq i < n$, is a non-empty set of strings $\tilde{X}[i][j]$, with $0 \leq j < |\tilde{X}[i]|$, where each $\tilde{X}[i][j]$ is a deterministic string on Σ. The total *size* of $\tilde{X}$ is defined as

$$N = \sum_{i=0}^{n-1} \sum_{j=0}^{|\tilde{X}[i]|-1} |\tilde{X}[i][j]|.$$

Only for the purpose of computing N, $|\varepsilon| = 1$. We remark that, for an ED string $\tilde{X}$, the size and the length are two distinct concepts (see Example 2).

We say that a string Y *matches* an ED string $\tilde{X} = \tilde{X}[0]\ldots\tilde{X}[m'-1]$ of length $m' > 1$, denoted by $Y \approx \tilde{X}$, if and only if string Y can be decomposed into $y_0 \ldots y_{m'-1}$, $y_i \in \Sigma^*$, such that:

1. there exists a string $s \in \tilde{X}[0]$ such that a suffix of s is $y_0 \neq \varepsilon$;
2. if $m' > 2$, there exists $s \in \tilde{X}[i]$, for all $1 \leq i \leq m'-2$, such that $s = y_i$;
3. there exists a string $s \in \tilde{X}[m'-1]$ such that a prefix of s is $y_{m'-1} \neq \varepsilon$.

Note that, in the above definition, we require that both y_0 and $y_{m'-1}$ are non-empty to avoid spurious matches at the beginning or end of an occurrence. A string Y is said to have an *occurrence* ending at position j in an ED string $\tilde{T}$ if there exist $i < j$ such that $\tilde{T}[i]\ldots\tilde{T}[j] \approx Y$, or, if there exists $s \in \tilde{T}[j]$ such that Y occurs in s.

▶ **Example 2** (Running example)**.** Suppose we have a pattern $P = \texttt{ACACA}$, of length $m = 5$, and an ED string $\tilde{T}$, of length $n = 6$ and size $N = 18$; the first occurrence of P starts at position 1 and ends at position 2 of $\tilde{T}$; and the second one starts at position 2 and ends at position 4.

$$\tilde{T} = \{\ \texttt{C}\ \} \cdot \left\{ \begin{array}{c} \texttt{A} \\ \texttt{C} \end{array} \right\} \cdot \left\{ \begin{array}{c} \texttt{AC} \\ \texttt{ACC} \\ \texttt{CACA} \end{array} \right\} \cdot \left\{ \begin{array}{c} \texttt{C} \\ \varepsilon \end{array} \right\} \cdot \left\{ \begin{array}{c} \texttt{A} \\ \texttt{AC} \end{array} \right\} \cdot \{\ \texttt{C}\ \}$$

We are now in a position to formally define the main problem of this paper.

Elastic-Degenerate String Matching (*EDSM*)
Input: a string P, of length m, and an ED string $\tilde{T}$, of length n and size $N \geq m$
Output: all positions j in $\tilde{T}$ where at least one occurrence of P ends

3 Algorithmic Tools

The *suffix tree* $\mathcal{ST}_y$ of a string y, of length $n > 0$, is a compact trie representing all suffixes of y. The nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We let $\mathcal{P}(v)$ denote the *path-label* of a node v, that is, the concatenation of the edge labels along the path from the root to v. We say that v is path-labelled $\mathcal{P}(v)$. Node v is marked as a *terminal* node if its path-label is a suffix of y, that is, $\mathcal{P}(v) = y[i\,..\,n-1]$ for some $0 \leq i < n$. Note that v is also labelled with index i. Thus, each factor of y is uniquely represented by an explicit or an implicit node of $\mathcal{ST}_y$. More details on suffix trees can be found in [7, 5].

▶ Fact 3 ([6, 5]). Given a string y of length n, $\mathcal{ST}_y$ can be constructed in time and space $\mathcal{O}(n)$. Finding all Occ_x occurrences of a string x, of length m, in y can be performed in time $\mathcal{O}(m + Occ_x)$ using $\mathcal{ST}_y$.

A *border* of a non-empty string x is a proper factor of x that is both a prefix and a suffix of x. We introduce the function $\mathsf{border}(x)$ defined for every non-empty string x as the longest border of x. Let x be a string of length $m \geq 1$. We define the *border table* B: $\{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$ by $\mathsf{B}[k] = |\mathsf{border}(x[0\,..\,k])|$, for $k = 0, 1, \ldots, m-1$.

▶ Fact 4 ([13, 5]). Given a string x of length m, the border table of x can be computed on-line in time $\mathcal{O}(m)$. All borders of x can be specified within the same time complexity using the border table.

We remark that the border table and the notion of border refer to a proper prefix and a proper suffix of the same string, whereas the indexes in set $\mathcal{B}_{x,y}$ refer to a string which is a prefix of a string (x) and a suffix of another (y), and that is not necessarily proper.

▶ **Lemma 5.** *Given a string x, of length m, and the suffix tree $\mathcal{ST}_y$ of a string y, of length n, $\mathcal{B}_{x,y}$ can be computed in time $\mathcal{O}(m)$.*

Proof. By applying Fact 3, we traverse $\mathcal{ST}_y$ to find the terminal node v corresponding to the longest prefix of x, which is path-labelled $\mathcal{P}(v)$. While traversing $\mathcal{ST}_y$ with x, we add index $n - i - 1$ to $\mathcal{B}_{x,y}$ if we encounter a terminal node u, such that $\mathcal{P}(u) = y[i\,..\,n-1]$. The longest such prefix of x is of length at most m. No longer prefix of x can be a suffix of y as it does not occur in y. ◀

4 Algorithm

An ED string can always represent an *exponential* number of strings (per ending position), where the exact number is the product of the number of deterministic strings at previous positions. Searching a pattern in all these strings separately is thus not acceptable.

Main idea. Our algorithm has a preprocessing phase where we build the suffix tree of the pattern P (Line 2 in pseudocode below). Then, in an on-line manner, we scan $\tilde{T}$ from left to right and, for each $\tilde{T}[i]$, we:

1. memorise the prefixes of the pattern that occur as suffixes of some $s \in \tilde{T}[i]$ (Lines 5 & 12 in pseudocode);
2. check whether at $\tilde{T}[i]$ it is possible to extend a partial occurrence of the pattern which has started earlier in the ED text (Lines $13-16$ in pseudocode);
3. in both previous cases we finally check whether a full occurrence of P actually also ends in $\tilde{T}[i]$ (Lines $6-8$ & $17-22$ in pseudocode).

We perform these steps by computing and storing, for each $0 \leq i < n$, the list $\mathcal{L}_i$ of the rightmost positions of prefixes of P that occur at the end of $\tilde{T}[i]$.

Below, we formally present Algorithm EDSM that solves the *EDSM* in an on-line manner. Note that by $\textsc{Insert}(A, \mathcal{L})$, we denote the operation that inserts the elements of a set A into a linked-list $\mathcal{L}$.

```
1  Algorithm EDSM(P, m, T̃, n)
2      Construct ST_P;
3      L_0 ← EmptyList();
4      foreach S ∈ T̃[0] do
5          Compute B_{P,S} using the border table; Insert(B_{P,S}, L_0);
6          if |S| ≥ m then
7              Search P in S using KMP and
8              report 0 if P occurs in S and CheckDuplicate(0);
9      foreach i ∈ [1, n − 1] do
10         L_i ← EmptyList();
11         foreach S ∈ T̃[i] do
12             Compute B_{P,S} using the border table; Insert(B_{P,S}, L_i);
13             if |S| < m then
14                 Search S in P using ST_P; denote starting positions by A;
15                 foreach (p ∈ L_{i−1}, j ∈ A) such that p + 1 = j do
16                     Insert({p + |S|}, L_i);
17             if |S| ≥ m then
18                 Search P in S using KMP and
19                 report i if P occurs in S and CheckDuplicate(i);
20             Compute B_{S,P} using ST_P;
21             if there exists (p ∈ L_{i−1}, j ∈ B_{S,P}) such that p + j + 2 = m then
22                 Report i if CheckDuplicate(i);
```

Example 6 shows Steps (1) and (2) on the running example. The border table shown in Example 6 has to be computed for all text positions, leading to the overall complexity stated in Lemma 7.

▶ **Example 6** (Running example). Let us consider again $P = \texttt{ACACA}$ and $\tilde{T}$ of Example 2. Assume we have already computed $\mathcal{L}_0$ and $\mathcal{L}_1$, and we move to position $i = 2$, where at $\tilde{T}[i]$ we have three strings $\{S_0, S_1, S_2\}$, with $S_0 = \texttt{AC}$, $S_1 = \texttt{ACC}$, and $S_2 = \texttt{CACA}$. We generate the string $X_i = X_2 = P\$_0 S_0 \$_1 S_1 \$_2 S_2 = \texttt{ACACA}\$_0\texttt{AC}\$_1\texttt{ACC}\$_2\texttt{CACA}$ and build its border table B (Line 12 in pseudocode).

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X_2[k]$	A	C	A	C	A	$\$_0$	A	C	$\$_1$	A	C	C	$\$_2$	C	A	C	A
B$[k]$	0	0	1	2	3	0	1	2	0	1	2	0	0	0	1	2	3

In order to compute $\mathcal{B}_{P,S}$ (Line 12), we read $\mathsf{B}[7] = 2$, which gives the length of the longest string that is a prefix of P and a suffix of S_0. To check if there exist borders of length shorter than 2, we read $\mathsf{B}[2-1] = 0$, indicating that no shorter border exists. Therefore, we have $\mathcal{B}_{P,S_0} = \{1\}$. We then read $\mathsf{B}[11] = 0$, telling us that no prefix of P is a suffix of S_1, and hence $\mathcal{B}_{P,S_1} = \emptyset$. We read $\mathsf{B}[16] = 3$, which gives the length of the longest string that is a prefix of P and a suffix of S_2. To check if there exist shorter borders, we read $\mathsf{B}[3-1] = 1$, indicating that a shorter border of length 1 exists. Since $\mathsf{B}[1-1] = 0$, no shorter border exists. Therefore, we have $\mathcal{B}_{P,S_2} = \{0, 2\}$. This gives us a partial $\mathcal{L}_i = \{0, 1, 2\}$ for position $i = 2$ that concludes Step 1 for position $i = 2$ (INSERT$(\mathcal{B}_{P,S}, \mathcal{L}_i)$, Line 12). Further on, at Step 2, we will add position 4 to $\mathcal{L}_2$ by extending the occurrence of P that had started at $\tilde{T}[1]$. Putting everything together, we get $\mathcal{L}_2 = \{0, 1, 2, 4\}$ (INSERT$(\{p + |S|\}, \mathcal{L}_i)$, Lines $15 - 16$).

▶ **Lemma 7.** *Given P, of length m, and $\tilde{T}$, of length n and size N, the sets $\mathcal{B}_{P,S}$ with $S \in \tilde{T}[i]$, for all $i \in [0, n-1]$, can be computed in time $\mathcal{O}(N)$.*

Proof. For each position i, we generate a string $X_i = P\$_0 S_0 \$_1 S_1 \$_2 S_2 \dots \$_{k-1} S_{k-1}$, where $S_j \in \tilde{T}[i]$, $0 \le j < k$, and $\$_j$'s are distinct letters not in Σ. We build the border table B of string X_i. By traversing B from left to right we can compute sets $\mathcal{B}_{P,S_j}$. Specifically, for any string S_j, all borders that are suffixes of S_j and prefixes of P can be computed in time $\mathcal{O}(|S_j|)$, since there exist at most $|S_j|$ such borders. By Fact 4, we can build all border tables, and hence compute all $\mathcal{B}_{P,S_j}$, for all $S_j \in \tilde{T}[i]$, in time $\mathcal{O}(|P| + \sum_{j=0}^{k-1} |S_j|)$. Since the length and the total size of $\tilde{T}$ are n and N, respectively, sets $\mathcal{B}_{P,S_j}$ can be computed in time $\mathcal{O}(nm + N)$. By noting that the border table for P can be computed only once and that the border table computation can be done on-line (Fact 4), the whole computation is bounded by $\mathcal{O}(N)$. ◀

▶ **Lemma 8.** *Given P, $\mathcal{ST}_P$, and $\tilde{T}$ of length n and size N, the sets $\mathcal{B}_{S,P}$, $S \in \tilde{T}[i]$, for all $i \in [1, n-1]$, can be computed in time $\mathcal{O}(N)$.*

Proof. By Lemma 5, for any $S \in \tilde{T}[i]$, $|S| \le |P|$, $\mathcal{B}_{S,P}$ can be computed in time $\mathcal{O}(|S|)$ using $\mathcal{ST}_P$. Since the total size of $\tilde{T}$ is N, sets $\mathcal{B}_{S,P}$ can be computed in time $\mathcal{O}(N)$. ◀

▶ **Lemma 9.** *Lists $\mathcal{L}_i$, for all $i \in [0, n-1]$, in Algorithm EDSM can be computed in time $\mathcal{O}(nm^2 + N)$.*

Proof. List $\mathcal{L}_0$ consists of the elements of $\mathcal{B}_{P,S}$ for position 0, which by Lemma 7 can be done within time $\mathcal{O}(N)$. For pattern P of length m, there exist at most $\frac{m(m+1)}{2}$ factors. For the strings $S_j \in \tilde{T}[i]$, $|S_j| \le m$, $0 \le j < k$, we can find at most $\frac{m(m+1)}{2} = \mathcal{O}(m^2)$ occurrences in pattern P. By Fact 3, finding all occurrences can be done in time $\sum_{j=0}^{k-1}(|S_j| + Occ_{S_j})$, and this is bounded by $\mathcal{O}(nm^2 + N)$ for all positions i. This is because, by definition, no $S_j, S_{j'} \in \tilde{T}[i]$ exist such that $S_j = S_{j'}$. Each occurrence can cause only one extension from $\mathcal{L}_{i-1}$ to $\mathcal{L}_i$. To avoid duplicates in $\mathcal{L}_i$, we need to check if there exist more than one prefix extensions ending at the same position. Each check can be done in constant time using a bit vector of size m, which we set on only once per position i. Therefore, we can extend the prefixes in time $\mathcal{O}(m^2)$ for each position i, and in time $\mathcal{O}(nm^2)$ for the whole text $\tilde{T}$ of length n. By Lemma 7, sets $\mathcal{B}_{P,S}$ corresponding to new prefixes of pattern P which are suffixes of $\{S_0, S_1, \dots, S_{k-1}\}$ at position $\tilde{T}[i]$ can be found in time $\mathcal{O}(N)$. Merging new prefixes with the prefixes extended from $\mathcal{L}_{i-1}$ can be done in time $\mathcal{O}(m)$, since both are at most m. Therefore, lists $\mathcal{L}_i$, for all $i \in [0, n-1]$, in EDSM can be computed in time $\mathcal{O}(nm^2 + N)$. ◀

Example 10 shows Step (3) on our running example.

▶ **Example 10** (Running example)**.** Let us consider again $P = \texttt{ACACA}$ and $\tilde{T}$ of Example 2. For position $i = 4$, we have $\mathcal{L}_3 = \{1, 3\}$ and we have to compute $\mathcal{L}_4$. For $S_0 = \texttt{A}$, we have $\mathcal{B}_{\texttt{A},\texttt{ACACA}} = \{0\}$ (Line 20), so for $3 \in \mathcal{L}_3$, we have that $3 + 0 + 2 = 5 = m$ (Line 21). Hence, one occurrence of P has been found. Moreover, for $S_1 = \texttt{AC}$, we have $\mathcal{B}_{\texttt{AC},\texttt{ACACA}} = \{0, 1\}$ (Line 20), so for $3 \in \mathcal{L}_3$, we have that $3 + 0 + 2 = 5 = m$ (Line 21). Therefore, another occurrence of P has been found at the same position.

Since Algorithm EDSM reports all positions i in $\tilde{T}$ where at least one occurrence of P ends, and since more than one occurrence may end at the same position (as in Example 10), we need to avoid duplications. To this end, we can use a simple operation to check whether the current position i has already been reported (CheckDuplicate(i), Lines 8, 19, & 22).

▶ **Theorem 11.** *Algorithm EDSM solves the EDSM problem in an on-line manner in time $\mathcal{O}(nm^2 + N)$. Algorithm EDSM requires preprocessing time and space $\mathcal{O}(m)$.*

Proof. The correctness of the algorithm follows from the correctness of the KMP algorithm [13] if $|S| > m$, $S \in \tilde{T}[i]$, and from the combination of Lemmas 8 and 9, if $|S| \leq m$. By definition, we cannot have any other type of (ending) occurrence.

By Fact 3, the suffix tree $\mathcal{ST}_P$ can be computed in time and space $\mathcal{O}(m)$. By Lemma 9, lists $\mathcal{L}_i$, for all $i \in [0, n-1]$, can be computed in time $\mathcal{O}(nm^2 + N)$. By Lemma 8, sets $\mathcal{B}_{S,P}$ can be computed in time $\mathcal{O}(N)$. In case $|S| < m$, we use $\mathcal{L}_{i-1}$ and set $\mathcal{B}_{S,P}$ to find and report occurrence i in time $\mathcal{O}(m)$ using a bit vector of size m, which we initialise only once per position i. Finally, searching P in $S \in \tilde{T}[i]$, in case $|S| \geq m$, can be done in time $\mathcal{O}(|S|)$ using the KMP algorithm [13], which is bounded by $\mathcal{O}(N)$ for $\tilde{T}$ of total size N.

The algorithm reads a position i and reports whether i is an ending position of some occurrence of P, before reading position $i+1$. Therefore, Algorithm EDSM solves the *EDSM* problem in an on-line manner in time $\mathcal{O}(nm^2 + N)$, with preprocessing time and space $\mathcal{O}(m)$. ◀

5 Bit-Vector Algorithm

We introduce here Algorithm EDSM-BV, a *non-trivial* bit-vector version of Algorithm EDSM.

Main idea. The main idea of this algorithm is to simulate the previous algorithm using bit-level operations to maintain linked-lists $\mathcal{L}$ and do the matching. To this end, we also add a further preprocessing step to the suffix tree of the pattern. This augmented suffix tree allows us to retrieve a bit-vector representation of all occurrences of an $S \in \tilde{T}[i]$ in P in time linear in $|S|$. With this structure, we can use bit-level operations to compute $\mathcal{L}_i$ from $\mathcal{L}_{i-1}$.

We maintain a bit vector $\boldsymbol{B}$ of size m initialised with 0's, such that, for each position $0 \leq k < m$, $\boldsymbol{B}[k] = 1$ if and only if $P[0\,..\,k]$ has an occurrence ending at the current position of $\tilde{T}$. For each letter $c \in \Sigma$, we construct a bit vector $\boldsymbol{I}_c$ of size m initialised with 0's, such that for each position $0 < k < m-1$, $\boldsymbol{I}_c[k-1] = 1$, if and only if $P[k] = c$. We construct the suffix tree of P, denoted by $\mathcal{ST}_P$, and augment it with bit vectors of size m initialised with 0's for each explicit node as follows: for node u, we create bit vector $\boldsymbol{M}_u$ such that $\boldsymbol{M}_u[k-1] = 1$, if and only if the factor $\mathcal{P}(u)$ represented by node u occurs at position k in P, $0 < k < m-1$. The occurrences of $\mathcal{P}(u)$ can be found at terminal nodes in the subtree rooted at node u. We denote this augmented suffix tree of P by Occ-Vector$_P$. We wish to answer the following type of on-line queries: given a string α, if α is a factor of P, then Occ-Vector$_P(\alpha)$ finds the node w in $\mathcal{ST}_P$ which represents α, and returns a pointer to the

bit vector $\boldsymbol{M}_u$, where u is the first explicit node in the subtree rooted at w. Otherwise (if α is not a factor of P), $\mathsf{Occ\text{-}Vector}_P(\alpha)$ returns a pointer to a bit vector consisting of m 0's. This operation can be trivially realised in time $\mathcal{O}(|\alpha|)$. Note that both $\boldsymbol{I}_c$ and $\boldsymbol{M}_u$ are shifted one bit to the left with respect to the pattern position they refer to; this is just an optimisation that will save us a shift in the algorithm.

Below, we formally present Algorithm EDSM-BV that solves the *EDSM* problem in an on-line manner.

```
1  Algorithm EDSM-BV(P, m, T̃, n, Σ)
2      Construct I_c, for all c ∈ Σ, and Occ-Vector_P;
3      B[0..m−1] ← 0;
4      foreach S ∈ T̃[0] do
5          Compute B_{P,S} using the border table;
6          foreach b ∈ B_{P,S} do
7              B[b] = 1;
8          if |S| ≥ m then
9              Search P in S using KMP and
10             report 0 if P occurs in S and CheckDuplicate(0);
11     foreach i ∈ [1, n−1] do
12         B_1[0..m−1] ← 0;
13         foreach S ∈ T̃[i] do
14             Compute B_{P,S} using the border table;
15             foreach b ∈ B_{P,S} do
16                 B_1[b] = 1;
17             if |S| < m then
18                 B_2 ← B & Occ-Vector_P(S);
19                 B_1 ← B_1 | (B_2 ≫ |S|);
20             if |S| ≥ m then
21                 Search P in S using KMP and
22                 report i if P occurs in S and CheckDuplicate(i);
23             B_3 ← B;
24             foreach j ∈ [0, min{|S|, m−1} − 1] do
25                 B_3 ← B_3 & I_{S[j]};
26                 B_3 ← B_3 ≫ 1;
27                 if B_3[m−1] = 1 then
28                     Report i if CheckDuplicate(i);
29         B ← B_1;
```

In Algorithm EDSM-BV, at each iteration i, $\tilde{T}[i]$ is processed (Lines $11-29$) and, at the end, vector $\boldsymbol{B}$ stores indexes k such that $P[0\,..\,k]$ ends at position i.

▶ **Lemma 12.** *Bit vectors $\boldsymbol{I}_c$, for all $c \in \Sigma$, $\sigma = |\Sigma|$, can be constructed in time $\mathcal{O}(m+\sigma\cdot\lceil\frac{m}{w}\rceil)$ and space $\mathcal{O}(\sigma\cdot\lceil\frac{m}{w}\rceil)$. Occ-Vector$_P$ can be constructed in time and space $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$.*

Proof. For the bit vectors $\boldsymbol{I}_c$, we first read the alphabet and construct σ bit vectors of size m initialised with 0's. Then we only need to read the pattern once, and for each position $0 < k < m-1$ in the pattern such that $P[k] = c$, we set $\boldsymbol{I}_c[k-1] = 1$. Reading the pattern once and setting $\boldsymbol{I}_c$ costs time $\mathcal{O}(m)$, so in total we need time $\mathcal{O}(m+\sigma\cdot\lceil\frac{m}{w}\rceil)$ for the bit vectors $\boldsymbol{I}_c$. The space for each bit vector of size m is $\mathcal{O}(\lceil\frac{m}{w}\rceil)$, so in total $\mathcal{O}(\sigma\cdot\lceil\frac{m}{w}\rceil)$ space is required.

By Fact 3, $\mathcal{ST}_P$ can be constructed in time and space $\mathcal{O}(m)$. We traverse $\mathcal{ST}_P$ and allocate a bit vector $\boldsymbol{M}_u$ of size m initialised with 0's for every explicit node u we visit. If u is a terminal node representing suffix $P[k\,..\,m-1]$, we set $\boldsymbol{M}_u[k-1]=1$. If u is a non-terminal node, we set $\boldsymbol{M}_u[k-1]=1$ for all terminal nodes representing suffixes $P[k\,..\,m-1]$ in the subtree rooted at u, $0<k<m-1$. This can be realised by using an Or bitwise operation between the bit vectors of the children of node u. By applying this for all explicit nodes of $\mathcal{ST}_P$, we build Occ-Vector$_P$. We have exactly m terminal nodes, and no more than m non-terminal nodes in $\mathcal{ST}_P$, thus, the bit vectors $\boldsymbol{M}_u$ for $\mathcal{ST}_P$ can be constructed in time $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$. The space required for Occ-Vector$_P$ is $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$ since we have $\mathcal{O}(m)$ bit vectors and each bit vector requires space $\mathcal{O}(\lceil\frac{m}{w}\rceil)$. ◀

▶ **Theorem 13.** *Algorithm EDSM-BV solves the EDSM problem in an on-line manner in time $\mathcal{O}(N\cdot\lceil\frac{m}{w}\rceil)$. Algorithm EDSM-BV requires preprocessing time and space $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$.*

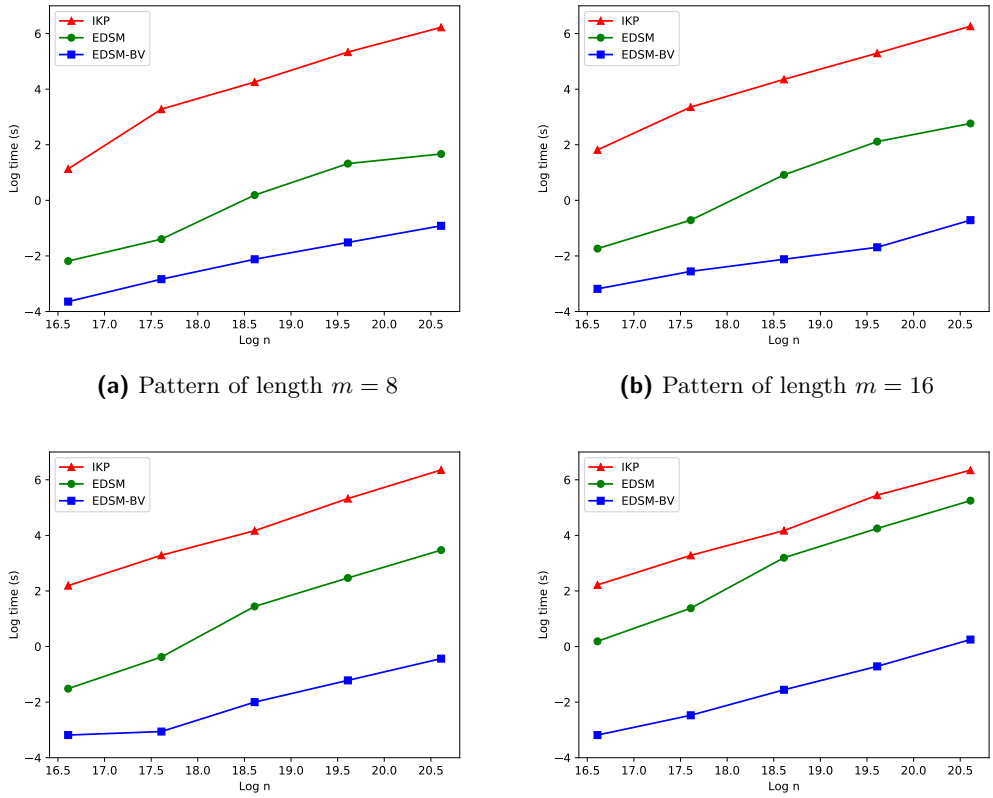
Proof. The correctness of the algorithm follows from the correctness of the KMP algorithm [13] if $|S|\geq m, S\in\tilde{T}[i]$. By the definition of bit vectors $\boldsymbol{I}_c$, we read each $S\in\tilde{T}[i]$, letter by letter, and try to extend the prefixes of P, position by position, using Shift-And bitwise operations [17]. When we reach the end of the bit vector $\boldsymbol{B}_3$, we may find an occurrence. No other occurrences can be found since we extend position by position, which means if we cannot reach the end of $\boldsymbol{B}_3$, we must have had at least one mismatch which prevents the extension.

By Lemma 12, the time and space for the preprocessing of Algorithm EDSM-BV is bounded by $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$. For each $S\in\tilde{T}[i]$, $|S|\geq m$, searching P in S can be done in time $\mathcal{O}(|S|)$ using the KMP algorithm [13], which is bounded by $\mathcal{O}(N)$ for all S. The Shift-And bitwise operation can be done in time $\mathcal{O}(\lceil\frac{m}{w}\rceil)$ [17], and it is repeated $|S|$ or $m-1$ times for each S to find an occurrence. Since we choose the minimum of $|S|$ and $m-1$, this time is bounded by $\mathcal{O}(|S|\cdot\lceil\frac{m}{w}\rceil)$, which is bounded by $\mathcal{O}(N\cdot\lceil\frac{m}{w}\rceil)$ for $\tilde{T}$. By Lemma 7, sets $\mathcal{B}_{P,S}$ can be computed in time $\mathcal{O}(N)$. Updating $\boldsymbol{B}$ for position $i=0$ and updating $\boldsymbol{B_1}$ for each position $i>0$ using sets $\mathcal{B}_{P,S}$ can be done in time $\mathcal{O}(N)$ for $\tilde{T}$. For each $S\in\tilde{T}[i]$, $|S|<m$, Occ-Vector$_P(S)$ requires time $\mathcal{O}(|S|)$ to return the corresponding bit vector, and updating $\boldsymbol{B_1}$ requires time $\mathcal{O}(\lceil\frac{m}{w}\rceil)$ using bit-level operations. Note that $\boldsymbol{B_1}$ needs only to be updated if $\boldsymbol{B}\neq 0$. So for all $\tilde{T}[i]$, the total time of this step can be bounded by $\mathcal{O}(N+N'\cdot\lceil\frac{m}{w}\rceil)$, where N' is the number of strings S such that $|S|<|P|$ and $\boldsymbol{B}\neq 0$. Since $N'\leq N$, this time is bounded by $\mathcal{O}(N\cdot\lceil\frac{m}{w}\rceil)$.

The algorithm reads a position i, and reports whether i is an ending position of some occurrence of P, before reading position $i+1$. Therefore, Algorithm EDSM-BV solves the *EDSM* problem in an on-line manner in time $\mathcal{O}(N\cdot\lceil\frac{m}{w}\rceil)$, with preprocessing time and space $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$. ◀

6 Experimental Results

We have implemented Algorithms EDSM and EDSM-BV in the C++ programming language. The implementation of the algorithm presented in [11], which we denote here by IKP, was taken from `https://github.com/Ritu-Kundu/ElDeS`. Recall that Algorithm IKP solves the *EDSM* problem in time $\mathcal{O}(\alpha\gamma mn+N)$ and space $\mathcal{O}(N)$; where α and γ are parameters, respectively representing the maximum number of strings in any degenerate position of the text and the maximum number of degenerate positions spanned by any occurrence of the pattern in the text. Note that Algorithm IKP outputs both the starting and ending positions of pattern occurrences, while the output of Algorithms EDSM and EDSM-BV is



(a) Pattern of length $m = 8$

(b) Pattern of length $m = 16$

(c) Pattern of length $m = 32$

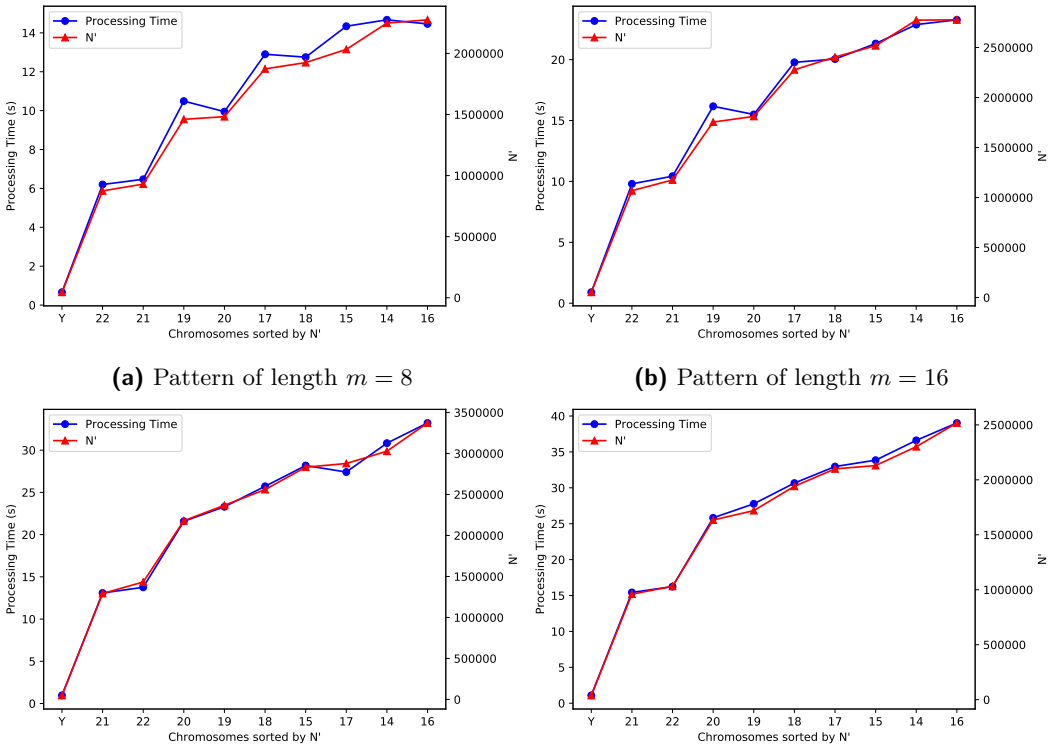
(d) Pattern of length $m = 64$

Figure 1 Elapsed time of EDSM, EDSM-BV, and IKP for synthetic ED texts of length n.

only the ending positions. All three programs were compiled with `g++` version 4.7.3 at optimisation level 3 (-O3). The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600S CPU at 2.8GHz and 8GB of RAM under 64-bit GNU/Linux. We compared the performance of EDSM, EDSM-BV, and IKP using synthetic data; as well as the performance of EDSM-BV—shown to be the fastest—using real data. The implementation of EDSM-BV is available at `https://github.com/webmasterar/edsm` under the terms of the GNU General Public License. The synthetic datasets referred to in this section are maintained at the same web-site.

Synthetic data. Synthetic ED texts were created randomly (uniform distribution over the DNA alphabet) with n ranging from $100,000$ to $1,600,000$; and the percentage of degenerate positions was set to 10%. For each degenerate position within the synthetic ED texts, the number of strings was chosen randomly, with an upper bound set to 10. The length of each string of a degenerate position was chosen randomly, with an upper bound again set to 10. Every non-degenerate position within the synthetic ED texts contained a single letter. Four different patterns of length $m = 8, 16, 32$, or 64 were given as input to all three programs, along with the aforementioned synthetic ED texts, resulting in four sets of output.

Our theoretical findings showing that Algorithms EDSM and EDSM-BV are asymptotically faster than Algorithm IKP are validated in practice by the results illustrated in Figure 1. Note



(a) Pattern of length $m = 8$

(b) Pattern of length $m = 16$

(c) Pattern of length $m = 32$

(d) Pattern of length $m = 64$

Figure 2 Processing time of EDSM-BV for real ED texts (Human chromosomes and variants).

that the axes are in $\log_2$ scale. In particular, the results confirm that Algorithm EDSM-BV, which is asymptotically the fastest for short patterns, is also the fastest in practice by up to two orders of magnitude. As for Algorithm EDSM, not surprisingly, we observe that, as m grows, the m^2 factor in its time complexity becomes more and more significant overall. Note that searching for much longer patterns *exactly* is not relevant in applications of interest, where errors (substitutions, insertions, and deletions) must be accommodated as m grows.

Real data. EDSM-BV was tested further using real-world datasets. Human genomic data was obtained from the 1,000 Genomes Project [23]. Specifically, data was obtained from Phase 3 of the project, in which the genomes of 2,504 individuals from 26 different populations were sequenced and aligned, producing a dataset which summarises the variation in the sample population. Files in Variant Call Format (VCF) include information about variations at each position in the reference genome, which makes the format ideal for our purposes. EDSM-BV was given a reference sequence (in FASTA format) and variation data (in VCF) for each of the ten smallest chromosomes as input, as well as synthetic patterns of length $m = 8, 16, 32$, or 64. The average percentage of degenerate positions across these chromosomes was approximately 3%; the average number of strings at degenerate positions was 2; and the average length of strings at degenerate positions was 1. The processing time of EDSM-BV was recorded; with *processing* we refer only to the actual CPU time used in executing the process—excluding the time to read the data in memory on-line. Chromosome 21, which is the smallest in length, has a VCF file of size 11.2GB. The results of this experiment are displayed in Figure 2.

The graphs in Figure 2 show, for the ten smallest chromosomes, a very clear *linear*

relationship between the time taken for EDSM-BV to run and N', the total number of strings $S \in \tilde{T}[i]$ such that $|S| < |P|$ and $\boldsymbol{B} \neq 0$, per chromosome. Recall that the total time required by EDSM-BV for updating bit vector $\boldsymbol{B}_1$ from $\boldsymbol{B}$ is $\mathcal{O}(N + N' \cdot \lceil \frac{m}{w} \rceil)$. This is the most time-consuming operation in practice as it searches for S in the suffix tree of P and then updates $\boldsymbol{B}_1$ using bit-level operations. Note that, the total time to process strings $S \in \tilde{T}[i]$, with $|S| > |P|$, using KMP is $\mathcal{O}(N)$, which becomes insignificant overall in practice.

7 Final Remarks

We have presented two efficient algorithms for on-line pattern matching on a set of similar texts. Notably, one of the algorithms requires time linear in the size of the texts' representation, for short patterns, that is $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$. The presented experimental results confirm our theoretical findings in practical terms.

Our immediate target is to apply these on-line solutions for fast average-case approximate pattern matching or for multiple pattern matching on a set of similar texts. An open problem is to either improve on the $\mathcal{O}(nm^2 + N)$-time algorithm or show conditional lower bounds.

Acknowledgements. We would like to thank Ritu Kundu (Department of Informatics, King's College London) for providing us with the implementation of Algorithm IKP [11]. Roberto Grossi, Nadia Pisanti, and Giovanna Rosone are partially supported by the project UniPi PRA_2017_44 ("Advanced computational methodologies for the analysis of biomedical data"). Costas S. Iliopoulos is partially supported by the Onassis Foundation. Ahmad Retha is supported by the Graduate Teaching Scholarship scheme of the Department of Informatics at King's College London. Giovanna Rosone is partially supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L. Fatima Vayani is supported by an EPSRC Grant (Doctoral Training Grant #EP/M506357/1).

References

1 Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Communications of the ACM 18(6), 333–340 (1975)
2 Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology 215(3), 403–410 (1990)
3 Baeza-Yates, R.A., Perleberg, C.H.: Fast and practical approximate string matching. Information Processing Letters 59(1), 21–27 (1996)
4 Baier, U., Beller, T., Ohlebusch, E.: Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. Bioinformatics 32(4), 497–504 (2016)
5 Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
6 Farach, M.: Optimal suffix tree construction with large alphabets. In: FOCS. pp. 137–143. IEEE Computer Society (1997)
7 Gusfield, D.: Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge University Press (1997)
8 Holley, G., Wittler, R., Stoye, J.: Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. Algorithms for Molecular Biology 11, 3 (2016)
9 Holub, J., Smyth, W.F., Wang, S.: Fast pattern-matching on indeterminate strings. Journal of Discrete Algorithms 6(1), 37–50 (2008)
10 Huang, L., Popic, V., Batzoglou, S.: Short read alignment with populations of genomes. Bioinformatics 29(13), 361–370 (2013)

11 Iliopoulos, C.S., Kundu, R., Pissis, S.P.: Efficient pattern matching in elastic-degenerate texts. In: LATA. LNCS, vol. 10168, pp. 131–142. Springer International Publishing (2017)

12 Kersey, P.J., Allen, J.E., Armean, I., Boddu, S., Bolt, B.J., Carvalho-Silva, D., Christensen, M., Davis, P., Falin, L.J., Grabmueller, C., Humphrey, J.C., Kerhornou, A., Khobova, J., Aranganathan, N.K., Langridge, N., Lowy, E., McDowall, M.D., Maheswari, U., Nuhn, M., Ong, C.K., Overduin, B., Paulini, M., Pedro, H., Perry, E., Spudich, G., Tapanari, E., Walts, B., Williams, G., Tello-Ruiz, M.K., Stein, J.C., Wei, S., Ware, D., Bolser, D.M., Howe, K.L., Kulesha, E., Lawson, D., Maslen, G., Staines, D.M.: Ensembl genomes 2016: more genomes, more complexity. Nucleic Acids Research 44(Database-Issue), 574–580 (2016)

13 Knuth, D.E., Jr., J.H.M., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350 (1977)

14 Maciuca, S., del Ojo Elias, C., McVean, G., Iqbal, Z.: A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In: WABI. LNBI, vol. 9838, pp. 222–233 (2016)

15 Na, J.C., Kim, H., Park, H., Lecroq, T., Léonard, M., Mouchard, L., Park, K.: FM-index of alignment: A compressed index for similar strings. Theoretical Computer Science 638, 159–170 (2016)

16 Navarro, G., Pereira, A.O.: Faster compressed suffix trees for repetitive collections. ACM Journal of Experimental Algorithmics 21(1), 1.8:1–1.8:38 (2016)

17 Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences. Cambridge University Press (2002)

18 Nguyen, N., Hickey, G., Zerbino, D.R., Raney, B.J., Earl, D., Armstrong, J., Kent, W.J., Haussler, D., Paten, B.: Building a pan-genome reference for a population. Journal of Computational Biology 22(5), 387–401 (2015)

19 Pisanti, N., Soldano, H., Carpentier, M., Pothier, J.: A relational extension of the notion of motifs: Application to the common 3D protein substructures searching problem. Journal of Computational Biology 16(12), 1635–1660 (2009)

20 Sagot, M., Viari, A., Pothier, J., Soldano, H.: Finding flexible patterns in a text: an application to three-dimensional molecular matching. Computer Applications in the Biosciences 11(1), 59–70 (1995)

21 Sheikhizadeh, S., Schranz, M.E., Akdel, M., de Ridder, D., Smit, S.: Pantools: representation, storage and exploration of pan-genomic data. Bioinformatics 32(17), 487–493 (2016)

22 Sirén, J.: Indexing variation graphs. In: ALENEX. pp. 13–27. SIAM (2017)

23 The 1000 Genomes Project Consortium: A global reference for human genetic variation. Nature 526(7571), 68–74 (2015)

24 The Computational Pan-Genomics Consortium: Computational pan-genomics: status, promises and challenges. Briefings in Bioinformatics pp. 1–18 (2016)