Pattern Matching on Elastic-Degenerate Text with Errors

Giulia Bernardini, Nadia Pisanti, Solon Pissis, Giovanna Rosone

Pattern Matching on Elastic-Degenerate Text with Errors

Giulia Bernardini[1], Nadia Pisanti[2,3], Solon P. Pissis[4], and Giovanna Rosone[2,⋆]

[1] Department of Mathematics, University of Pisa, Pisa, Italy
[2] Department of Computer Science, University of Pisa, Pisa, Italy
[3] Erable Team, INRIA, Villeurbanne, France
[4] Department of Informatics, King's College London, London, UK

Abstract. An elastic-degenerate string is a sequence of n sets of strings of total length N. It has been introduced to represent a multiple alignment of several closely-related sequences (e.g. pan-genome) compactly. In this representation, substrings of these sequences that match exactly are collapsed, while in positions where the sequences differ, all possible variants observed at that location are listed. The natural problem that arises is finding all matches of a deterministic pattern of length m in an elastic-degenerate text. There exists an $\mathcal{O}(nm^2 + N)$-time algorithm to solve this problem on-line after a pre-processing stage with time and space $\mathcal{O}(m)$. In this paper, we study the same problem under the edit distance model and present an $\mathcal{O}(k^2mG + kN)$-time and $\mathcal{O}(m)$-space algorithm, where G is the total number of strings in the elastic-degenerate text and k is the maximum edit distance allowed. We also present a simple $\mathcal{O}(kmG + kN)$-time and $\mathcal{O}(m)$-space algorithm for Hamming distance.

Keywords: uncertain strings, elastic-degenerate strings, degenerate strings, pan-genome, pattern matching

1 Introduction

There is a growing interest in the notion of *pan-genome* [20]. In the last ten years, with faster and cheaper sequencing technologies, re-sequencing (that is, sequencing the genome of yet another individual of a species) became more and more a common task in modern genome analysis workflows. By now, a huge amount of genomic variations within the same population has been detected (e.g. in humans for medical applications, but not only), and this is only the beginning. With this, new challenges of functional annotation and comparative analysis have been raised. Traditionally, a single annotated *reference genome* is used as a control sequence. The reference genome is a representative example of the genomic sequence of a species. It serves as a reference text to which, for example, fragments of newly sequenced genomes of individuals are mapped. Although a single reference genome provides a good approximation of any individual genome,

⋆ Corresponding author

in loci with polymorphic variations, mapping and comparisons easily fail their purposes. This is where a multiple genome, i.e. a pan-genome, would be a better reference text [10].

In the literature, many different (compressed) representations and thus algorithms have been considered for pattern matching on a set of similar texts [4, 5, 16, 21, 6, 3, 12]. A natural representation of pan-genomes, or fragments of them, that we consider here are elastic-degenerate texts [11]. An *elastic-degenerate text* is a sequence which compactly represents a multiple alignment of several closely-related sequences. In this representation, substrings that match exactly are collapsed, while in positions where the sequences differ (by means of substitutions, insertions, and deletions of substrings), all possible variants observed at that location are listed. Elastic-degenerate texts correspond to the Variant Call Format (VCF), that is, the *standard* for storing gene sequence variations [19].

Consider, for example, the following *multiple sequence alignment* of three closely-related sequences:

```
GAAAGTGAGCA
GAGACAAA-CA
G--A-ACAGCA
```

These sequences can be compacted into the single elastic-degenerate string:

$$\tilde{T} = \{\texttt{G}\} \cdot \left\{ \begin{array}{c} \texttt{AA} \\ \texttt{AG} \\ \varepsilon \end{array} \right\} \cdot \{\texttt{A}\} \cdot \left\{ \begin{array}{c} \texttt{GTG} \\ \texttt{CAA} \\ \texttt{AC} \end{array} \right\} \cdot \{\texttt{A}\} \cdot \left\{ \begin{array}{c} \texttt{G} \\ \varepsilon \end{array} \right\} \cdot \{\texttt{CA}\}.$$

The total number of segments is the *length* of $\tilde{T}$ and the total number of letters is the *size* of $\tilde{T}$. 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 (sometimes called indeterminate) segment can contain only single letters [9].

Due to the application of cataloguing human genetic variation [19], there has been ample work in the literature on the *off-line* (indexing) version of the pattern matching problem [10, 17, 14, 18, 15]. 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 [2]; **(c)** on-line solutions can be useful when one wants to search for a few patterns in many degenerate texts similar to standard strings [1].

Previous Results. Let us denote by m the length of pattern P, by n the length of $\tilde{T}$, and by $N > m$ the size of $\tilde{T}$. A few results exist on the (exact) *EDSM* problem. 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 [7], two new algorithms to solve the same problem in an on-line manner[1] were presented: the first one requires time $\mathcal{O}(nm^2 + N)$ after a pre-processing stage with time and space $\mathcal{O}(m)$; the second requires time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ after a pre-processing 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.

Our Contribution. Since genomic sequences are endowed with polymorphisms and sequencing errors, the existence of an exact match can result into a strong assumption. The aim of this work is to generalize the studies of [11] and [7] for the exact case, allowing some approximation in the occurrences of the input pattern. We suggest a simple on-line $\mathcal{O}(kmG + kN)$-time and $\mathcal{O}(m)$-space algorithm, G being the total number of strings in $\tilde{T}$ and $k > 0$ the maximum number of allowed substitutions in a pattern's occurrence, that is nonzero *Hamming distance*. Our main contribution is an on-line $\mathcal{O}(k^2mG + kN)$-time and $\mathcal{O}(m)$-space algorithm where the type of edit operations allowed is extended to insertions and deletions as well, that is nonzero *edit distance*. These results are *good* in the sense that for *small* values of k the algorithms incur (essentially) no increase in time complexity with respect to the $\mathcal{O}(nm^2 + N)$-time and $\mathcal{O}(m)$-space algorithm presented in [7].

Structure of the Paper. Section 2 provides some preliminary definitions and facts as well as the formal statements of the problems we address. Section 3 describes our solution under the edit distance model, while Section 4 describes the algorithm under the Hamming distance model.

2 Preliminaries

An *alphabet* Σ is a non-empty finite set of letters of size $|\Sigma|$. We consider the case of a constant-sized alphabet, i.e. $|\Sigma| = \mathcal{O}(1)$. A *string* S on an alphabet Σ is a sequence of elements of Σ. The set of all strings on an alphabet Σ, including the *empty string* ε of length 0, is denoted by Σ^*. For any string S, we denote by $S[i \ldots j]$ the *substring* of S that *starts* at position i and *ends* at position j. In particular, $S[0 \ldots j]$ is the *prefix* of S that ends at position j, and $S[i \ldots |S| - 1]$ is the *suffix* of S that begins at position i, where $|S|$ denotes the *length* of S.

Definition 1 ([7]). *An* elastic-degenerate (ED) string $\tilde{T} = \tilde{T}[0]\tilde{T}[1] \ldots \tilde{T}[n-1]$ *of* length n *on alphabet* Σ*, is a finite sequence of* n *degenerate letters. Every* degenerate letter $\tilde{T}[i]$ *is a finite non-empty set of strings* $\tilde{T}[i][j] \in \Sigma^*$*, with* $0 \leq j < |\tilde{T}[i]|$*. The* size N *of* $\tilde{T}$ *is defined as*

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

assuming (for representation purposes only) that $|\varepsilon|=1$*.*

[1] On-line refers to the fact that the algorithm reads the elastic-degenerate text set-by-set in a serial manner.

Definition 2. *The* total number of strings *in $\tilde{T}$ is defined as $G = \sum_{i=0}^{n-1} |\tilde{T}[i]|$.*

Notice that $n \leq G \leq N$. A deterministic string is simply a string in Σ^*. The *Hamming distance* is defined between two deterministic strings of equal length as the number of positions at which the two strings have different letters. The *edit distance* between two deterministic strings is defined as the minimum total cost of a sequence of edit operations (that is, substitution, insertion, or deletion of a letter) required to transform one string into the other. Here we only count the number of edit operations, considering the cost of each to be 1. In [7] the authors give a definition of a match between a deterministic string P and an ED text $\tilde{T}$; here we extend their definition to deal with errors.

Definition 3. *Given an integer $k > 0$, we say that a string $P \in \Sigma^m$ k_H-*matches *(resp. k_E-*matches*) an ED string $\tilde{T} = \tilde{T}[0]\tilde{T}[1] \dots \tilde{T}[n-1]$ of length $n > 1$ if all of the following hold:*

- *there exists a non-empty suffix X of some string $S \in \tilde{T}[0]$;*
- *if $n > 2$, there exist strings $Y_1 \in \tilde{T}[1]$,...,$Y_t \in \tilde{T}[t]$, for $1 \leq t \leq n-2$;*
- *there exists a non-empty prefix Z of some string $S \in \tilde{T}[n-1]$;*
- *the Hamming (resp. edit) distance between P and $XY_1 \dots Y_t Z$ (note that $Y_1 \dots Y_t$ can be equal to ε) is no more than k.*

We say that P has a k_H*-occurrence* (resp. k_E-) ending at position j in an ED string $\tilde{T}$ of length n if either there exists a k_H-match (resp. k_E-) between P and $\tilde{T}[i \dots j]$ for some $0 \leq i < j \leq n-1$ or P is at Hamming (resp. edit) distance of at most k from a substring of some string $S \in \tilde{T}[j]$.

Example 4. (Running example) Consider $P = \texttt{GAACAA}$ of length $m = 6$. The following ED string has $n = 7$, $N = 20$, and $G = 12$. An 1_H-occurrence is underlined, and an 1_E-occurrences is overlined.

$$\tilde{T} = \{\underline{\overline{\texttt{G}}}\} \cdot \left\{ \begin{array}{c} \overline{\texttt{AA}} \\ \texttt{AG} \\ \underline{\varepsilon} \end{array} \right\} \cdot \{\underline{\overline{\texttt{A}}}\} \cdot \left\{ \begin{array}{c} \texttt{GTG} \\ \overline{\texttt{CAA}} \\ \underline{\texttt{AC}} \end{array} \right\} \cdot \{\underline{\texttt{A}}\} \cdot \left\{ \begin{array}{c} \texttt{G} \\ \underline{\varepsilon} \end{array} \right\} \cdot \{\underline{\texttt{C}}\texttt{A}\}$$

A *suffix tree* ST_X for a string X of length m is a tree data structure where edge-labels of paths from the root to the (terminal) node labelled i spell out suffix $X[i \dots m-1]$ of X. ST_X can be built in time and space $\mathcal{O}(m)$. The suffix tree can be generalized to represent the suffixes of a set of strings $\{X_1, \dots, X_n\}$ (denoted by $GST_{X_1,\dots,X_n}$) with time and space costs still linear in the length of the input strings (see [8], for details).

Given two strings X and Y and a pair (i, j), with $0 \leq i \leq |X| - 1$ and $0 \leq j \leq |Y| - 1$, the *longest common extension* at (i, j), denoted by $lce_{X,Y}(i, j)$, is the length of the longest substring of X starting at position i that matches a substring of Y starting at position j. For instance, for $X = \texttt{CGCGT}$ and $Y = \texttt{ACG}$, we have that $lce_{X,Y}(2, 1) = 2$, corresponding to the substring $\texttt{CG}$.

Fact 1 ([8]) *Given a string X and its ST_X, and a set of strings $W = \{Y_1, \ldots, Y_l\}$, it is possible to build the generalized suffix tree $GST_{X,W}$ extending ST_X, in time $\mathcal{O}(\sum_{h=1}^{l} |Y_h|)$. Moreover, given two strings X and Y of total length q, for each index pair (i, j), $lce_{X,Y}(i,j)$ queries can be computed in constant time per query, after a pre-processing of $GST_{X,Y}$ that takes time and space $\mathcal{O}(q)$.*

We will denote by $GST^*_{X,Y}$ such a pre-processed tree for answering *lce* queries. The time is ripe now to formally introduce the two problems considered here.

[k_E-**EDSM**] Elastic-Degenerate String Matching with Edit Distance:

Input: A deterministic pattern P of length m, an elastic-degenerate text $\tilde{T}$ of length n and size $N \geq m$, and an integer $0 < k < m$.
Output: Pairs (i, d), i being a position in $\tilde{T}$ where at least one k_E-occurrence of P ends and $d \leq k$ being the minimal number of errors (insertions, deletions and substitutions) for occurrence i.

[k_H-**EDSM**] Elastic-Degenerate String Matching with Hamming Distance:

Input: A deterministic pattern P of length m, an elastic-degenerate text $\tilde{T}$ of length n and size $N \geq m$, and an integer $0 < k < m$.
Output: Pairs (i, d), i being a position in $\tilde{T}$ where at least one k_H-occurrence of P ends and $d \leq k$ being the minimal number of mismatches for occurrence i.

3 An Algorithm for k_E-EDSM

In [7] the exact EDSM problem (that is, for $k = 0$) was solved in time $\mathcal{O}(m^2n+N)$. Allowing up to k substitutions, insertions, and deletions in the occurrences clearly entails a time-cost increase, but the solution proposed here manages to keep the time-cost growth limited, solving the k_E-EDSM problem in time $\mathcal{O}(k^2mG + kN)$, G being the total number of strings in the ED text. At a high level, the k_E-EDSM algorithm (pseudocode shown below) works as follows.

Pre-processing phase: the suffix tree for the pattern P is built (line 1 in pseudocode).
Searching phase: in an on-line manner, the text $\tilde{T}$ is scanned from left to right and, for each $\tilde{T}[i]$:

(1) It finds the prefixes of P that have a k_E-match ending at $\tilde{T}[i]$; if there exists an $S \in \tilde{T}[i]$ that is long enough, it also searches for k_E-occurrences of P that start and end at position i (lines 6 and 16);
(2) It tries to extend at $\tilde{T}[i]$ a partial k_E-occurrence of the pattern which has started earlier in the ED text (lines 23 and 30);
(3) In both previous cases, if the occurrence of P also ends in $\tilde{T}[i]$, then it outputs position i; otherwise it stores the prefixes of P extended at $\tilde{T}[i]$ (lines 7-9, 17-19, 24-26, 31-32).

```
$k_E$-EDSM($P$,$m$,$\tilde{T}$,$n$,$k$)
1  Build $ST_P$;
2  for $j = 0$ to $m-1$ do $V_c[j] \leftarrow \infty$;
3  $L_c \leftarrow \emptyset$;
4  Build $GST^*_{P,\tilde{T}[0]}$;
5  forall $S \in \tilde{T}[0]$ do
6      $L' \leftarrow \emptyset$; $L' \leftarrow k_E$-BORDERS($P, m, S, |S|, GST^*_{P,\tilde{T}[0]}, k$);
7      forall $(j,d) \in L'$ do
8          if $j = m-1 \wedge d < V_c[m-1]$ then $V_c[m-1] = d$;
9          else INSERT($L_c$,$(j,d)$,$V_c$);
10 if $V_c[m-1] \neq \infty$ then report $(0, V_c[m-1])$;
11 for $i = 1$ to $n-1$ do
12     $L_p \leftarrow L_c$; $L_c \leftarrow \emptyset$;
13     $V_p \leftarrow V_c$; for $j = 0$ to $m-1$ do $V_c[j] \leftarrow \infty$;
14     Build $GST^*_{P,\tilde{T}[i]}$ ;
15     forall $S \in \tilde{T}[i]$ do
16         $L' \leftarrow \emptyset$; $L' \leftarrow k_E$-BORDERS($P, m, S, |S|, GST^*_{P,\tilde{T}[i]}, k$);
17         forall $(j,d) \in L'$ do
18             if $j = m-1 \wedge d < V_c[m-1]$ then $V_c[m-1] = d$;
19             else INSERT($L_c$,$(j,d)$,$V_c$);
20         if $|S| < m$ then
21             forall $p \in L_p$ do
22                     $L' \leftarrow \emptyset$;
23                     $L' \leftarrow k_E$-EXTEND($p+1$,$P$,$m$,$S$,$|S|$,$GST^*_{P,\tilde{T}[i]}$,$k - V_p[p]$);
24                     forall $(j,d) \in L'$ do
25                         if $j = m-1 \wedge d + V_p[p] < V_c[m-1]$ then
                            $V_c[m-1] = d + V_p[p]$;
26                         else INSERT($L_c$,$(j, d + V_p[p])$, $V_c$);
27         if $|S| \geq m$ then
28             forall $p \in L_p$ do
29                     $L' \leftarrow \emptyset$;
30                     $L' \leftarrow k_E$-EXTEND($p+1$,$P$,$m$,$S$,$|S|$,$GST^*_{P,\tilde{T}[i]}$,$k - V_p[p]$);
31                     forall $(j,d) \in L'$ do
32                         if $j = m-1 \wedge d + V_p[p] < V_c[m-1]$ then
                            $V_c[m-1] = d + V_p[p]$;
33     if $V_c[m-1] \neq \infty$ then report $(i, V_c[m-1])$;
```

Step (1) of algorithm k_E-EDSM is accomplished using algorithm k_E-BORDERS described in Section 3.1. Step (2) is implemented by algorithm k_E-EXTEND described in Section 3.2.

The following lemma follows directly from Fact 1.

Lemma 5. *Given P of length m and $\tilde{T}$ of length n and size N, building $GST^*_{P,\tilde{T}[i]}$ for all $\tilde{T}[i]$'s takes total time $\mathcal{O}(mn + N)$.*

Besides ST_P (built once as a pre-processing step) and $GST^*_{P,\tilde{T}[i]}$ (built for all $\tilde{T}[i]$'s), the algorithm uses the following data structures:

L' A list re-initialized to $\emptyset$ for each $S \in \tilde{T}[i]$: contains pairs (j, d) storing the rightmost position j of P such that a k_E-match of $P[0 \dots j]$ ends at S with edit distance d. L' is filled in by k_E-BORDERS and k_E-EXTEND.

V_c A vector of size $|P|$ re-initialized for each $\tilde{T}[i]$ (c stands for *current* position) to $V_c[j] = \infty$ for all j's: $V_c[j]$ contains the lowest number of errors for a partial k_E-occurrence of $P[0 \dots j]$. For each pair (j, d) in L', if $V_c[j] < d$ then $V_c[j]$ is updated with d by the function INSERT. $V_c[j] = \infty$ denotes that a partial k_E-occurrence of $P[0 \dots j]$ has not yet been found.

L_c A list re-initialized to $\emptyset$ for each $\tilde{T}[i]$: contains the rightmost positions of all the prefixes of P found ending at $\tilde{T}[i]$. It is filled in by function INSERT for each rightmost position j where $V_c[j]$ turns into a value $\neq \infty$.

L_p A list where at the beginning of each iteration i for $\tilde{T}[i]$, the L_c list for $i - 1$ is copied. L_p thus stores prefixes of P that ended at the previous position (p stands for *previous* position).

V_p Similarly, in V_p the vector V_c of the previous position is copied.

Algorithm k_E-EDSM needs to report each position i in $\tilde{T}$ where some k_E-occurrence of P ends with edit distance d, d being the minimal such value for position i. To this aim, the last position of V_c can be updated with the following criterion: each time an occurrence of P ending at $\tilde{T}[i]$, $(m - 1, d)$, is found, if $V_c[m - 1] > d$ then we set $V_c[m - 1] = d$. After all $S \in \tilde{T}[i]$ have been examined, if $V_c[m - 1] \neq \infty$, the algorithm outputs the pair $(i, V_c[m - 1])$.

3.1 Algorithm k_E-BORDERS

For each i and for each $S \in \tilde{T}[i]$, Step (1) of the algorithm needs to find all prefixes of P that are at distance at most k from a suffix of some $S \in \tilde{T}[i]$, as well as k_E-occurrences of P that start and end at position i if S is long enough. To this end, we use and modify the Landau-Vishkin algorithm [13]. We first recall some relevant definitions from [8] concerning the dynamic programming table.

Given an m by q dynamic programming table (m rows, q columns), the *main diagonal* consists of cells (h, h) for $0 \leq h \leq \min\{m - 1, q - 1\}$. The diagonals above the main diagonal are numbered 1 through $(q - 1)$; the diagonal starting in cell $(0, h)$ is diagonal h. The diagonals below the main diagonal are numbered -1 through $-(m - 1)$; the diagonal starting in cell $(h, 0)$ is diagonal $-h$. A *d-path* in the dynamic programming table is a path that starts in row zero and specifies a total of exactly d errors (insertions, deletions and substitutions). A d-path is *farthest reaching in diagonal h* if it is a d-path that ends in diagonal h, and the index of its ending column c is $\geq$ to the ending column of any other d-path ending in diagonal h.

Algorithm k_E-BORDERS takes as input a pattern P of length m, a string $S \in \tilde{T}[i]$ of length q, the $GST^*_{P,\tilde{T}[i]}$ and the upper bound k for edit distance; it outputs pairs (j, d), where j is the rightmost position of the prefix of P that is at distance d from a suffix of S, with the minimal value of d reported for each j. In order to fulfill this task, at a high level, the algorithm executes the following steps on a table having P at the rows and S at the columns:

(1a) For each diagonal $0 \leq h \leq q-1$ it finds $lce_{P,S}(0, h)$. This specifies the end column of the farthest reaching 0-path on each diagonal from 0 to $q-1$.

(1b) For each $1 \leq d \leq k$, it finds the farthest reaching d-path on diagonal h, for each $-d \leq h \leq q-1$. This path is found from the farthest reaching $(d-1)$-paths on diagonals $(h-1)$, h and $(h+1)$.

(1c) If a d-path reaches the last row of the dynamic programming table, then a k_E-occurrence of P with edit distance d that starts and ends at position i has been found, and the algorithm reports $(m-1, d)$; if a d-path reaches the end of S in row r, then the prefix of P ending at $P[r]$ is at distance d from a suffix of S, and the algorithm reports (r, d).

In Step (1b), the farthest reaching d-path on diagonal h can be found by computing and comparing the following three particular paths that end on diagonal h:

R_1**:** Consists of the farthest reaching $(d-1)$-path on diagonal $h+1$, followed by a vertical edge to diagonal h, and then by the maximal extension along diagonal h that corresponds to identical substrings. Function R_1 takes as input the length $|X|$ of a string X, whose letters spell the rows of the dynamic table, the length $|Y|$ of a string Y, whose letters spell the columns, $GST^*_{X,Y}$ and the pair row-column (r, c) where the farthest reaching $(d-1)$-path on diagonal $h+1$ ends. It outputs pair (r_1, c_1) where path R_1 ends. This path represents a letter insertion in X.

R_2**:** Consists of the dual case of R_1 with a horizontal edge representing a letter deletion in X.

R_3**:** Consists of the farthest reaching $(d-1)$-path on diagonal h followed by a diagonal edge, and then by the maximal extension along diagonal h that corresponds to identical substrings. Function R_3 takes as input the length $|X|$ of a string X, whose letters spell the rows of the dynamic table, the length $|Y|$ of a string Y, whose letters spell the columns, $GST^*_{X,Y}$ and the pair row-column (r, c) where the farthest reaching $(d-1)$-path on diagonal h ends. It outputs pair (r_3, c_3) where path R_3 ends. This path represents a letter substitution.

Fact 2 ([8]) *The farthest reaching path on diagonal h is the path among R_1, R_2 or R_3 that extends the farthest along diagonal h.*

In each one of the iterations in k_E-BORDERS, a diagonal is associated with two variables *pFRP* and *cFRP*, storing the column reached by the farthest reaching path (FRP) in the previous and in the current iteration, respectively.

```
INSERT($L$,($j,d$),$V$)
1 if $V[j] > d$ then
2     if $V[j] = \infty$ then Insert $j$ in $L$;
3     $V[j] \leftarrow d$;
```

```
$R_1(|X|, |Y|, GST^*_{X,Y}, r, c)$
1 if $-1 \leq r \leq |X| - 1 \wedge -1 \leq c \leq |Y| - 1$ then
2     $l \leftarrow lce_{X,Y}(r+2, c+1)$;
3     $c_1 \leftarrow c + l$;
4     $r_1 \leftarrow r + 1 + l$;
5     return $(r_1, c_1)$
6 else return $(r, c)$;
```

```
$R_2(|X|, |Y|, GST^*_{X,Y}, r, c)$
1 if $-1 \leq r \leq |X| - 1 \wedge -1 \leq c \leq |Y| - 1$ then
2     $l \leftarrow lce_{X,Y}(r+1, c+2)$;
3     $c_2 \leftarrow c + 1 + l$;
4     $r_2 \leftarrow r + l$;
5     return $(r_2, c_2)$
6 else return $(r, c)$;
```

```
$R_3(|X|, |Y|, GST^*_{X,Y}, r, c)$
1 if $-1 \leq r \leq |X| - 1 \wedge -1 \leq c \leq |Y| - 1$ then
2     $l \leftarrow lce_{X,Y}(r+2, c+2)$;
3     $c_3 \leftarrow c + 1 + l$;
4     $r_3 \leftarrow r + 1 + l$;
5     return $(r_3, c_3)$
6 else return $(r, c)$;
```

In algorithm k_E-BORDERS, at most $k+q$ diagonals need to be taken into account: the algorithm first finds the *lce*'s between $P[0]$ and $S[j]$, for all $0 \leq j \leq q-1$, and hence it initializes q diagonals; after this, for each successive step (there are at most k of them), it widens to the left one diagonal at a time, because an initial deletion can be added; therefore, it will consider at most $k+q$ diagonals.

Lemma 6. *Given P of length m, $\tilde{T}$ of length n and size N, the $GST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, and an integer $0 < k < m$, k_E-BORDERS finds all prefixes of P that are at edit distance at most k from suffixes of $S \in \tilde{T}[i]$ and the k_E-occurrences of P that start and end at position i, in time $\mathcal{O}(k^2G + kN)$, G being the total number of strings in $\tilde{T}$.*

Proof. For a string $S \in \tilde{T}[i]$, for each $0 \leq d \leq k$ and each diagonal $-k \leq h \leq |S|-1$, the k_E-BORDERS algorithm must retrieve the end of three $(d-1)$-paths (constant-time operations) and compute the path extension along the diagonal via a constant-time *lce* query (Fact 1). It thus takes time $\mathcal{O}(k^2 + k|S|)$ to find all prefixes of P that are at distance at most k from suffixes of S; the k_E-occurrences of P that start and end at position i are computed within the same complexity. The total time is $\mathcal{O}(k^2|\tilde{T}[i]| + k\sum_{j=0}^{|\tilde{T}[i]|-1}|S|)$, for all $S \in \tilde{T}[i]$. Since the size of $\tilde{T}$ is N and the total number of strings in $\tilde{T}$ is G, the result follows. □

3.2 Algorithm k_E-EXTEND

In Step (2), algorithm k_E-EDSM needs to extend each partial k_E-occurrence that has started earlier in $\tilde{T}$. That is, at text position $\tilde{T}[i]$, for each $p \in L_p$ and

```
$k_E$-BORDERS$(P, m, S, q, GST^*_{P,\tilde{T}[i]}, k)$

for $h = -(k+1)$ to $-1$ do $cFRP(h) \leftarrow h - 1$;
for $h = 0$ to $q - 1$ do
    $l \leftarrow lce_{P,S}(0, h)$;
    $cFRP(h) \leftarrow l - 1 + h$;
    if $l + h = q$ then report $(l - 1, 0)$;
    else
        if $l = m$ then report $(m - 1, 0)$;
for $d = 1$ to $k$ do
    for $h = -d$ to $q - 1$ do $pFRP(h) \leftarrow cFRP(h)$;
    for $h = -d$ to $q - 1$ do
        $(r_1, c_1) \leftarrow R_1(|P|, |S|, GST^*_{P,\tilde{T}[i]}, pFRP(h+1) - (h+1), pFRP(h+1))$;
        $(r_2, c_2) \leftarrow R_2(|P|, |S|, GST^*_{P,\tilde{T}[i]}, pFRP(h-1) - (h-1), pFRP(h-1))$;
        $(r_3, c_3) \leftarrow R_3(|P|, |S|, GST^*_{P,\tilde{T}[i]}, pFRP(h) - h, pFRP(h))$;
        $cFRP(h) \leftarrow \max\{c_1, c_2, c_3\}$;
        if $\max\{r_1, r_2, r_3\} = m - 1$ then report $(m - 1, d)$;
        if $\max\{c_1, c_2, c_3\} = q - 1$ then report $(q - 1 - h, d)$;
```

for each string $S \in \tilde{T}[i]$, we try to extend $P[p+1 \ldots m-1]$ with S. Once again, we modify the Landau-Vishkin algorithm [13] to our purpose: it suffices to look for the FRPs starting at the desired position only.

k_E-EXTEND takes as input a pattern P of length m, a string $S \in \tilde{T}[i]$ of length q, the $GST^*_{P,\tilde{T}[i]}$, the upper bound k for edit distance and the position j in P where the extension should start; it outputs a list of distinct pairs (h, d), where h is the index of P where the extension ends, and d is the minimum additional number of errors introduced by the extension. Algorithm k_E-EXTEND somehow performs a task which is the *dual* of that of k_E-BORDERS, as (i) it builds a $q \times (m - 1 - j)$ DP table (rather than a $m \times q$ table) and (ii) instead of searching for occurrences of P starting anywhere within S, k_E-EXTEND checks whether the whole S can extend the prefix $P[0 \ldots j-1]$ detected at the previous text position, and whether a prefix of S matches the suffix of P starting at $P[j]$ (and hence the whole P has been found). At a high level, the algorithm executes the following steps:

(2a) It finds $lce_{S,P}(0, j)$ specifying the end column of the farthest reaching 0-path on diagonal 0 (as it builds a DP table for S and $P[j \ldots m-1]$).

(2b) For each $1 \leq d \leq k$, it finds the farthest reaching d-path on diagonal h, for each $-d \leq h \leq d$. This path is found from the farthest reaching $(d-1)$-paths on diagonals $(h-1)$, h and $(h+1)$.

(2c) If a d-path reaches the last row of the dynamic programming table in column c, then an occurrence of the whole S with edit distance d has been found, and the algorithm reports $(c + j, d)$, $c + j$ being the position in P where the occurrence ends; if a d-path reaches the end of P, then a prefix of

S is at distance d from a suffix of P starting at position j, and the algorithm reports $(m-1, d)$.

```
$k_E$-EXTEND$(j, P, m, S, q, GST^*_{P,\tilde{T}[i]}, k)$
1  if $S = \varepsilon$ then
2     for $d = 0$ to $k$ do report $(j+d, d)$;
3  else
4     for $h = -(k+1)$ to $k+1$ do $cFRP(h) \leftarrow h - k - 1$;
5     $l \leftarrow lce_{S,P}(0, j)$;
6     $cFRP(0) \leftarrow l - 1$;
7     if $l = q$ then report $(l + j - 1, 0)$;
8     for $d = 1$ to $k$ do
9        for $h = -d$ to $d$ do $pFRP(h) \leftarrow cFRP(h)$;
10       for $h = -d$ to $d$ do
11          $(r_1, c_1) \leftarrow R_1(|S|, m-1-j, GST^*_{P,\tilde{T}[i]}, pFRP(h+1)-(h+1), pFRP(h+1))$;
12          $(r_2, c_2) \leftarrow R_2(|S|, m-1-j, GST^*_{P,\tilde{T}[i]}, pFRP(h-1)-(h-1), pFRP(h-1))$;
13          $(r_3, c_3) \leftarrow R_3(|S|, m-1-j, GST^*_{P,\tilde{T}[i]}, pFRP(h)-h, pFRP(h))$;
14          $cFRP(h) \leftarrow \max\{c_1, c_2, c_3\}$;
15          if $\max\{r_1, r_2, r_3\} = q - 1$ then report $(cFRP(h) + j, d)$;
16          if $\max\{c_1, c_2, c_3\} = m - 1 - j$ then report $(m - 1, d)$;
```

Lemma 7. *Given a prefix of P, a string $S \in \tilde{T}[i]$, the $GST^*_{P,\tilde{T}[i]}$, and an integer $0 < k < m$, k_E-EXTEND extends the prefix of P with S in time $\mathcal{O}(k^2)$.*

Proof. The k_E-EXTEND algorithm does k iterations: at iteration d, for each diagonal $-d \leq h \leq d$, the end of three paths must be retrieved (constant-time operations) and the path extension along diagonal h must be computed via a constant-time *lce* query (Fact 1). The overall time for the extension is then bounded by $\mathcal{O}(1 + 2 + \cdots + (2k+1))=\mathcal{O}(k^2)$. □

The following lemma summarizes the time complexity of k_E-EDSM.

Lemma 8. *Given P of length m, $\tilde{T}$ of length n and total size N, and an integer $0 < k < m$, algorithm k_E-EDSM solves the k_E-EDSM problem, in an on-line manner, in time $\mathcal{O}(k^2 mG + kN)$, G being the total number of strings in $\tilde{T}$.*

Proof. At the i-th iteration, algorithm k_E-EDSM tries to extend each $j \in L_p$ with each string $S \in \tilde{T}[i]$. By Lemma 5, building $GST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, requires time $\mathcal{O}(mn+N)$. By Lemma 7, extending a single prefix with a string S costs time $\mathcal{O}(k^2)$; in L_p there are at most $|P| = m$ prefixes; then to extend all of them with a single string S requires time $\mathcal{O}(mk^2)$. In $\tilde{T}[i]$ there are $|\tilde{T}[i]|$ strings, so the time cost rises to $\mathcal{O}(|\tilde{T}[i]|mk^2)$ for each $\tilde{T}[i]$, leading to an overall time cost of $\mathcal{O}(k^2mG)$ to perform extensions. By Lemma 6, all prefixes of P that are at distance at most k from suffixes of S and the k_E-occurrences of P that start and end at position i can be found in time $\mathcal{O}(k^2G + kN)$; the overall time complexity for the whole k_E-EDSM algorithm is then $\mathcal{O}(mn + N + k^2mG + k^2G + kN) = \mathcal{O}(k^2mG + kN)$. The algorithm is on-line in the sense that any occurrence of the pattern ending at position i is reported before reading $\tilde{T}[i+1]$. □

Theorem 9. *The k_E-EDSM problem can be solved on-line in time $\mathcal{O}(k^2mG + kN)$ and space $\mathcal{O}(m)$.*

Proof. In order to obtain the space bound $\mathcal{O}(m)$, it is necessary to modify algorithm k_E-EDSM. The proposed method works as follows: each string $S \in \tilde{T}[i]$ is (conceptually) divided into windows of size $2m$ (except for the last one, whose length is $\leq m$) overlapping by m. Let W_j be the j-th window in S, $1 \leq j \leq \frac{|S|}{m}$. Instead of building $GST^*_{P,\tilde{T}[i]}$ for each degenerate letter $\tilde{T}[i]$, the algorithm now builds GST^*_{P,W_j} for each $1 \leq j \leq \frac{|S|}{m}$ and for each $S \in \tilde{T}[i]$: since the windows are of size $2m$, this can be done in both time and space $\mathcal{O}(m)$. Both algorithms k_E-BORDERS and k_E-EXTEND require space linear in the size of the string that spell the columns of the dynamic programming table, that is either P (in extensions) or a window of size $2m$ (in borders). Each list (L_c, L_p, L') and each vector (V_c, V_p) requires space $\mathcal{O}(m)$, so the overall required space is actually $\mathcal{O}(m)$.

The time bound is not affected by these modifications of the algorithm: the maximum number of windows in $\tilde{T}[i]$, in fact, is $\max\{|\tilde{T}[i]|, \frac{N_i}{m}\}$, where $N_i = \sum_{j=0}^{|\tilde{T}[i]|-1} |\tilde{T}[i][j]|$. This means that it takes time $\mathcal{O}(m|\tilde{T}[i]|)$ or $\mathcal{O}(m\frac{N_i}{m}) = O(N_i)$ to build and pre-process every suffix tree for $\tilde{T}[i]$. Algorithm k_E-BORDERS requires time $\mathcal{O}(k^2 + km) = \mathcal{O}(km)$ (because $k < m$) for each window: again, this must be multiplied by the number of windows in $\tilde{T}[i]$, so the time is $\max\{\mathcal{O}(km|\tilde{T}[i]|), \mathcal{O}(kN_i))\}$ for $\tilde{T}[i]$. Coming to algorithm k_E-EXTEND, nothing changes, as prefixes of P can only be extended by prefixes of S, so it suffices to consider one window for each S: it still requires time $\mathcal{O}(k^2mG)$ over the whole ED text. Summing up all these considerations, it is clear that the overall time is

$$\mathcal{O}(\sum_{i=0}^{n-1}[\max\{m|\tilde{T}[i]|, N_i\} + \max\{km|\tilde{T}[i]|, kN_i\}] + k^2mG) =$$
$$= \mathcal{O}(\sum_{i=0}^{n-1}[\max\{km|\tilde{T}[i]|, kN_i\}] + k^2mG)$$

which is clearly bounded by $\mathcal{O}(k^2mG + kN)$. □

4 An Algorithm for k_H-EDSM

The overall structure of algorithm k_H-EDSM (pseudocode not shown) is the same as k_E-EDSM. The two algorithms differ in the functions used to perform Step

(1) (k_H-BORDERS rather than k_E-BORDERS) and Step (2) (k_H-EXTEND rather than k_E-EXTEND). The new functions take as input the same parameters as the old ones and, like them, they both return lists of pairs (j, d) (pseudocode shown below). Unlike k_E-BORDERS and k_E-EXTEND, with k_H-BORDERS and k_H-EXTEND such pairs now represent (partial) occurrences of P in $\tilde{T}$ with Hamming distance.

```
$k_H$-BORDERS($P$,$m$,$S$,$q$,$GST^*_{P,\tilde{T}[i]}$,$k$)
1  for $h = 0$ to $q - 1$ do
2      $count \leftarrow 0$;
3      $j \leftarrow 0$;
4      $h' \leftarrow h$;
5      while $count \leq k$ do
6          $l \leftarrow lce_{P,S}(j, h')$;
7          if $h' + l = q$ then report $(q - h - 1, count)$ ;
8          else
9              if $h' + l + 1 = q \wedge count + 1 \leq k$ then report $(q - h, count + 1)$ ;
10             else
11                 if $j + l = m$ then report $(m - 1, count)$ ;
12                 else
13                     if $j + l + 1 = m \wedge count + 1 \leq k$ then report
                        $(m - 1, count + 1)$ ;
14                     else
15                         $count \leftarrow count + 1$;
16                         $j \leftarrow j + l + 1$;
17                         $h' \leftarrow h' + l + 1$;
```

At the i-th iteration, for all $S \in \tilde{T}[i]$ and any position h in S, k_H-BORDERS determines whether a prefix of P is at distance at most k from the suffix of S starting at position h via executing up to $k+1$ *lce* queries in the following manner: computing $l = lce_{P,S}(0, h)$, it finds out that $P[0 \dots l-1]$ and $S[h \dots h+l-1]$ match exactly and $P[l] \neq S[h+l]$. It can then skip one position in both strings (the mismatch $P[l] \neq S[h+l]$), increasing the error-counter by 1, and compute the $lce_{P,S}(l+1, h+l+1)$. This process is performed up to $k+1$ times, until either (i) the end of S is reached, and then a prefix of P is at distance at most k from the suffix of S starting at h (lines 7-12 in pseudocode); or (ii) the end of P is reached, then a k_H-occurrence of P has been found (lines 13-17 in pseudocode). If the end of S nor the end of P are reached, then more than k mismatches are required, and the algorithm continues with the next position (that is, $h+1$) in S.

The following lemma gives the total cost of all the calls of algorithm k_H-BORDERS in k_H-EDSM.

Lemma 10. *Given P of length m, $\tilde{T}$ of length n and size N, the $GST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, and an integer $0 < k < m$, k_H-BORDERS finds all prefixes of*

k_H-EXTEND(j,P,m,S,q,$GST^*_{P,\tilde{T}[i]}$,k)

```
if $S = \varepsilon$ then report $(j, 0)$;
else
    $count \leftarrow 0$;
    $h \leftarrow 0$;
    $j' \leftarrow j$;
    while $count \leq k$ do
        $l \leftarrow lce_{P,S}(i', j)$;
        if $h + l = q$ then report $(j' + l - 1, count)$ ;
        else
            if $h + l + 1 = q \wedge count + 1 \leq k$ then report $(j' + l, count + 1)$ ;
            else
                if $j' + l = m$ then report $(m - 1, count)$ ;
                else
                    if $j' + l + 1 = m \wedge count + 1 \leq k$ then report
                     $(m - 1, count + 1)$ ;
                    else
                        $count \leftarrow count + 1$;
                        $h \leftarrow h + l + 1$;
                        $j' \leftarrow j' + l + 1$;
```

P that are at Hamming distance at most k from suffixes of $S \in \tilde{T}[i]$ and the k_H-occurrences of P that start and end at position i, in time $\mathcal{O}(kN)$.

Proof. For any position h in S, the k_H-BORDERS algorithm finds the prefix of P that is at distance at most k from the suffix of S starting at position h in time $\mathcal{O}(k)$ by performing up to $k + 1$ *lce* queries (Fact 1). Over all positions of S, the method therefore requires time $\mathcal{O}(k|S|)$. Doing this for all $S \in \tilde{T}[i]$ and for all $i \in [0, n - 1]$ leads to the result. □

At the i-th iteration, for each partial k_H-occurrence of P started earlier (represented by $j \in L_p$ similar to algorithm k_E-EDSM) k_H-EXTEND tries to extend it with a string from the current text position. To this end, for each string $S \in \tilde{T}[i]$, it checks whether some partial occurrence can be extended with the whole S starting from position j of P, or whether a full k_H-occurrence can be obtained by considering only a prefix of S for the extension. The algorithm therefore executes up to $k + 1$ *lce* queries with the same possible outcomes and consequences mentioned for k_H-BORDERS.

The following lemma gives the total cost of all the calls of algorithm k_H-EXTEND in k_H-EDSM.

Lemma 11. *Given P of length m, $\tilde{T}$ of length n and size N, the $GST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n - 1]$, and an integer $0 < k < m$, k_H-EXTEND finds all the extensions of prefixes of P required by k_H-EDSM in time $\mathcal{O}(kmG)$, G being the total number of strings in $\tilde{T}$.*

Proof. Algorithm k_H-EXTEND determines in time $\mathcal{O}(k)$ whether a partial k_H-occurrence of P can be extended by S by performing up to $k+1$ constant-time *lce* queries (Fact 1); checking whether a full k_H-occurrence is obtained by considering only a prefix of S for the extension can be performed within the same complexity. Since P has m different prefixes, extending all of them costs $\mathcal{O}(km)$ per each string S. Given that there are G such strings, the overall time is $\mathcal{O}(kmG)$. □

Lemma 12. *Given P of length m, $\tilde{T}$ of length n and total size N, and an integer $0 < k < m$, algorithm k_H-EDSM solves the k_H-EDSM problem, in an on-line manner, in time $\mathcal{O}(kmG + kN)$, G being the total number of strings in $\tilde{T}$.*

Proof. At the i-th iteration, algorithm k_H-EDSM tries to extend each $j \in L_p$ with each string $S \in \tilde{T}[i]$. By Lemma 5, building $GST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, requires time $\mathcal{O}(mn + N)$. By Lemma 11, extending prefixes of P stored in L_p with each string $S \in \tilde{T}[i]$ has an overall time cost of $\mathcal{O}(kmG)$. By Lemma 10, finding prefixes of P that are at distance at most k from suffixes of $S \in \tilde{T}[i]$ and the k_H-occurrences of P that start and end at position i takes time $\mathcal{O}(kN)$ in total. Summing up, the overall time complexity for the whole k_H-EDSM algorithm is then $\mathcal{O}(mn + N + kmG + kN) = \mathcal{O}(kmG + kN)$, as $G \geq n$. The algorithm is on-line in the sense that any occurrence of the pattern ending at position i is reported before reading $\tilde{T}[i+1]$. □

The proof of Theorem 9 suggests a way in which algorithm k_E-EDSM can be run on-line in space $\mathcal{O}(m)$; it should be straightforward to see that a similar modification of algorithm k_H-EDSM leads to the following result.

Theorem 13. *The k_H-EDSM problem can be solved on-line in time $\mathcal{O}(kmG + kN)$ and space $\mathcal{O}(m)$.*

Acknowledgements

NP and GR are partially supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L. GB, NP, and GR are partially supported by the project UniPi PRA_2017_44 ("Advanced computational methodologies for the analysis of biomedical data"). NP, SPP, and GR are partially supported by the Royal Society project IE 161274 ("Processing uncertain sequences: combinatorics and applications").

References

1. 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)
2. Baeza-Yates, R.A., Perleberg, C.H.: Fast and practical approximate string matching. Inf. Process. Lett. 59(1), 21–27 (1996)

3. Barton, C., Liu, C., Pissis, S.P.: On-line pattern matching on uncertain sequences and applications. In: COCOA. LNCS, vol. 10043, pp. 547–562. Springer International Publishing (2016)
4. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: SODA. pp. 373–389. SIAM (2011)
5. Gagie, T., Gawrychowski, P., Puglisi, S.J.: Faster approximate pattern matching in compressed repetitive texts. In: ISAAC. LNCS, vol. 7074, pp. 653–662. Springer (2011)
6. Gagie, T., Puglisi, S.J.: Searching and indexing genomic databases via kernelization. Frontiers in Bioengineering and Biotechnology 3, 12 (2015)
7. Grossi, R., Iliopoulos, C.S., Liu, C., Pisanti, N., Pissis, S.P., Retha, A., Rosone, G., Vayani, F., Versari, L.: On-Line Pattern Matching on Similar Texts. In: CPM. LIPIcs, vol. 78, pp. 9:1–9:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2017)
8. Gusfield, D.: Algorithms on strings, trees, and sequences. Cambridge University Press New York, New York (1997)
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. Kociumaka, T., Pissis, S.P., Radoszewski, J.: Pattern Matching and Consensus Problems on Weighted Sequences and Profiles. In: ISAAC. LIPIcs, vol. 64, pp. 46:1–46:12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2016)
13. Landau, G., Vishkin, U.: Introducing efficient parallelism into approximate string matching and a new serial algorithm. In: STOC. pp. 220–230. ACM (1986)
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. LNCS, vol. 9838, pp. 222–233. Springer (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. Theor. Comput. Sci. 638, 159–170 (2016)
16. Navarro, G.: Indexing highly repetitive collections. In: IWOCA. LNCS, vol. 7643, pp. 274–279. Springer (2012)
17. 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)
18. Sirén, J.: Indexing variation graphs. In: ALENEX. pp. 13–27. SIAM (2017)
19. The 1000 Genomes Project Consortium: A global reference for human genetic variation. Nature 526(7571), 68–74 (2015)
20. The Computational Pan-Genomics Consortium: Computational pan-genomics: status, promises and challenges. Briefings in Bioinformatics pp. 1–18 (2016)
21. Wandelt, S., Leser, U.: String searching in referentially compressed genomes. In: KDIR. pp. 95–102. SciTePress (2012)