Approximate pattern matching on elastic-degenerate text

Giulia Bernardini, Nadia Pisanti, Solon Pissis, Giovanna Rosone

Approximate Pattern Matching on Elastic-Degenerate Text

Giulia Bernardini[a], Nadia Pisanti[b,d], Solon P. Pissis[c,d], Giovanna Rosone[b]

[a] *Department of Informatics, Systems and Communication (DISCo), University of Milano - Bicocca, Italy*
[b] *Department of Computer Science, University of Pisa, Italy*
[c] *ERABLE Team, INRIA, France*
[d] *CWI, Amsterdam, The Netherlands*

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 a non-combinatorial $\mathcal{O}(nm^{1.381} + N)$-time algorithm to solve this problem on-line [1]. 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 solving the problem under Hamming distance.

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

Email addresses: giulia.bernardini@unimib.it (Giulia Bernardini), pisanti@di.unipi.it (Nadia Pisanti), solon.pissis@cwi.nl (Solon P. Pissis), giovanna.rosone@unipi.it (Giovanna Rosone)

1. Introduction

There is a growing interest in the notion of *pan-genome* [2]. 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, in loci with polymorphic variations, mapping and sequence comparison often fail their purposes. This is where a multiple genome, *i.e.,* a pan-genome, would be a better reference text [3].

In the literature, many different (compressed) representations and thus algorithms have been considered for pattern matching on a set of similar texts [4, 5, 6, 7, 8, 9, 10]. 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 in so-called *degenerate segments*. In the literature, other similar types of uncertain sequences have also been considered, namely *degenerate strings*, where each degenerate segment can contain only single letters (see [12, 13, 14, 15, 16] and references therein) and *generalised degenerate strings* [17], where each degenerate segment contains strings of the same length. Elastic-degenerate texts are more general than both the previous objects, and they correspond to the Variant Call Format (VCF), that is, the *standard* for storing gene sequence variations [18]. A tool EDSO (available at `https://github.com/webmasterar/edso`) for creating elastic-degenerate texts from VCF files was also made available in [19].

Consider, for example, the following *multiple sequence alignment* of three

closely-related sequences:

```
GAAACAAAACA
GAGAGTGA-CA
G--A-ACAACA
```

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

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

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 [20].

Due to the application of cataloguing human genetic variation [18], there has been ample work in the literature on the *off-line* (indexing) version of the pattern matching problem [3, 21, 22, 23, 24]. The *on-line*, more fundamental, version of the *EDSM* problem has not been studied as much as indexing approaches. The motivation for considering the on-line version of the problem is to remove the hardship of building disk-based indexes or rebuilding them with every update in the sequences. Indexes are often unwieldy, take a lot of time and space to build, and require lots of disk space to be stored. Their usage is therefore only convenient when the data is static or changes very infrequently. Solutions to the on-line version can thus 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 [25]; **(c)** on-line solutions can be useful when one wants to search for a few patterns in many degenerate texts similar to standard strings such as protein or DNA sequences [26].

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 [27], 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 word-RAM model. Later, in [28] a new on-line algorithm was proposed and the running time was improved to $\mathcal{O}(nm^{1.5}\sqrt{\log m} + N)$; in [19] the authors extended the algorithm of [27] through adding the ability to search for multiple patterns simultaneously, achieving time $\mathcal{O}(N\lceil M/w \rceil)$ with pre-processing time and space $\mathcal{O}(M)$, where M is the total length of the patterns; a bit-parallel method, that matches the same $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ time complexity as [27] by simpler means, was also proposed in [29], leading to a fast and practical implementation of the algorithm. A bit-parallel algorithm to align a sequence to a graph was also presented in [30]. Finally, [1] provides a conditional lower bound for the *EDSM* problem. The authors show that any combinatorial algorithm that solves the problem in $\mathcal{O}(nm^{1.5-\varepsilon} + N)$ time, for any $\varepsilon > 0$, refutes the Boolean Matrix Multiplication conjecture. Notably, the authors also present a non-combinatorial $\mathcal{O}(nm^{1.381} + N)$-time algorithm that solves the *EDSM* problem by applying Fast Matrix Multiplication.

Our Contribution. Since genomic sequences are endowed with polymorphisms and sequencing errors, the existence of an exact occurrence can result into a strong assumption. The aim of this work is to generalize the studies of [11] and [27] 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 [27]

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

for the exact case. A preliminary version of this paper appeared in [31].

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 for constant-sized alphabets under the edit distance model, while Section 4 describes the algorithm under the Hamming distance model for constant-sized alphabets. Section 5 extends these algorithms to work for general integer alphabets. We conclude in Section 6.

2. Preliminaries

An *alphabet* Σ is a non-empty finite set of letters of size $|\Sigma|$. We start by considering the case of a constant-sized alphabet, *i.e.,* $|\Sigma| = \mathcal{O}(1)$, and then extend all the results to general integer alphabets in Section 5. 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 $S[i \dots j]$ the *substring* of S that *starts* at position i and *ends* at position j. In particular, $S[0 \dots j]$ is the *prefix* of S that ends at position j, and $S[i \dots |S| - 1]$ is the *suffix* of S that begins at position i, where $|S|$ denotes the *length* of S.

Definition 2.1 ([27])**.** An *elastic-degenerate (ED) string* of *length* n on alphabet Σ, $\tilde{T} = \tilde{T}[0]\tilde{T}[1] \dots \tilde{T}[n-1]$, 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. 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 [27] the authors give a definition of an exact match between a deterministic string P and an ED string $\tilde{T}$; here we extend their definition to deal with errors.

Definition 2.2. Given an integer $k > 0$, we say that a string $P \in \Sigma^m$ *k_H-matches* (resp. *k_E-*) 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 $Y_1 \in \tilde{T}[1],\dots,Y_t \in \tilde{T}[t]$, 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 $XY_1\dots Y_tZ$ (note that $Y_1\dots Y_t$ can be ε) is no more than k.

We say that P has a *k_H-occurrence* (resp. *k_E-*) ending at position j in $\tilde{T}$ if either there exists a k_H-match (resp. k_E-) between P and $\tilde{T}[i]\dots\tilde{T}[j]$ 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 $S \in \tilde{T}[j]$. We say that P has a *partial* k_H-occurrence (resp. k_E-) $P[0\dots\ell]$, for some $\ell < m-1$, ending at position j of $\tilde{T}$ if $P[0\dots\ell]$ k_H-matches (resp. k_E-) $\tilde{T}[i]\dots\tilde{T}[j]$, for some $0 \leq i \leq j$.

Example 2.3 (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 of P is underlined, and an 1_E-occurrences of P is overlined.

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

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. For constant-sized alphabets, 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 $ST_{X_1,\dots,X_n}$) with time and space costs still linear in the length of the input strings (see [32], 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, when $X = \texttt{CGCGT}$ and $Y = \texttt{ACG}$, $lce_{X,Y}(2,1) = 2$, corresponding to the substring $\texttt{CG}$. We define $lce_{X,Y}(i,j) = 0$ when either $i \notin \{0,1,\ldots,|X|-1\}$ or $j \notin \{0,1,\ldots,|Y|-1\}$.

Fact 1 ([32]). *Given a string X, its ST_X, and a set of strings $W = \{Y_1,\ldots,Y_l\}$ over a constant-sized alphabet, it is possible to build the generalized suffix tree $ST_{X,W}$ 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 $ST_{X,Y}$ that takes time and space $\mathcal{O}(q)$.*

We will denote by $ST^*_{X,Y}$ 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 under the edit distance model:

Input: A deterministic pattern P of length m, an ED text $\tilde{T}$ of length n and size $N \geq m$, 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 (substitutions, insertions and deletions) for occurrence i.

[k_H-**EDSM**] Elastic-Degenerate String Matching under the Hamming distance model:

Input: A deterministic pattern P of length m, an ED text $\tilde{T}$ of length n and size $N \geq m$, 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 substitutions for occurrence i.

3. An Algorithm for k_E-EDSM

In [28] the exact *EDSM* problem (that is, for $k = 0$) was solved in time $\mathcal{O}(nm^{1.5}\sqrt{\log m} + N)$. 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. We assume a constant-sized alphabet. At a high level, the k_E-EDSM algorithm (pseudocode shown below) works as follows.

Pre-processing phase: build the suffix tree for the pattern P.

Searching phase: in an on-line manner, scan the text $\tilde{T}$ from left to right and, for each $\tilde{T}[i]$:

(1) Find the prefixes of P that are at edit distance at most k from any suffix of some $S \in \tilde{T}[i]$; if there exists an $S \in \tilde{T}[i]$ that is long enough, also search for k_E-occurrences of P that start and end at position i (lines 3 and 13 of the pseudocode)

(2) Try to extend at $\tilde{T}[i]$ each partial k_E-occurrence of P which has started earlier in $\tilde{T}$ (line 19)

(3) In both previous cases, if a full k_E-occurrence of P also ends at $\tilde{T}[i]$, then output position i; otherwise store the prefixes of P extended at $\tilde{T}[i]$ (lines 4-7, 14-17, 20-28)

Step (1) of algorithm k_E-EDSM is implemented by algorithm k_E-BORD described in Section 3.1. Step (2) is implemented by algorithm k_E-EXT described in Section 3.2.

The following lemma follows directly from Fact 1.

Lemma 3.1. *Given P of length m and $\tilde{T}$ of length n and size N, to build $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, requires total time $\mathcal{O}(N)$.*

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

- L' - a list that temporarily stores the output of functions k_E-BORD and k_E-EXT. It is re-initialized to $\emptyset$ (lines 3, 13 and 19) for each $S \in \tilde{T}[i]$ before executing either of the functions.
- V_c - a vector of size $|P|$ such that $V_c[j]$ contains the lowest number of errors for a partial k_E-occurrence of $P[0 \dots j]$ ending at $\tilde{T}[i]$. For each pair position - edit operations (j, d) in L', if $V_c[j] < d$ then $V_c[j]$ is updated with d by the function INSERT. V_c (c stands for *current*) is re-initialized

k_E-EDSM(P,ST_P,$\tilde{T}$,n,k)

$V_c[0 \ldots |P|-1] \leftarrow \infty$; Build $ST^*_{P,\tilde{T}[0]}$;
forall $S \in \tilde{T}[0]$ **do**
  $L' \leftarrow \emptyset$; $L' \leftarrow k_E$-BORD$(P, S, ST^*_{P,\tilde{T}[0]}, k)$;
  forall $(j, d) \in L'$ **do**
    if $j = |P| - 1$ **then**
      if $d < V_c[|P| - 1]$ **then** $V_c[|P| - 1] \leftarrow d$;
    else INSERT(L_c,(j, d),V_c);
if $V_c[|P| - 1] \neq \infty$ **then report** $(0, V_c[|P| - 1])$;
for $i = 1$ **to** $n - 1$ **do**
  $L_p \leftarrow L_c$; $L_c \leftarrow \emptyset$; $V_p \leftarrow V_c$;
  $V_c[0 \ldots |P| - 1] \leftarrow \infty$; Build $ST^*_{P,\tilde{T}[i]}$;
  forall $S \in \tilde{T}[i]$ **do**
    $L' \leftarrow \emptyset$; $L' \leftarrow k_E$-BORD$(P, S, ST^*_{P,\tilde{T}[i]}, k)$;
    forall $(j, d) \in L'$ **do**
      if $j = |P| - 1$ **then**
        if $d < V_c[|P| - 1]$ **then** $V_c[|P| - 1] \leftarrow d$;
      else INSERT(L_c,(j, d),V_c);
    forall $p \in L_p$ **do**
      $L' \leftarrow \emptyset$; $L' \leftarrow k_E$-EXT$(p + 1$,P,S,$ST^*_{P,\tilde{T}[i]}$,$k - V_p[p])$;
      forall $(j, d) \in L'$ **do**
        if $j = |P| - 1$ **then**
          if $d + V_p[p] < V_c[|P| - 1]$ **then** $V_c[|P| - 1] \leftarrow d + V_p[p]$;
        else INSERT(L_c,$(j, d + V_p[p])$, V_c);
  if $V_c[|P| - 1] \neq \infty \vee V_p[|P| - 1] < k$ **then**
    report $(i, \min\{V_c[|P| - 1], V_p[|P| - 1] + 1\})$
  if $V_p[|P| - 1] + \min_{S \in \tilde{T}[i]} |S| < k$ **then**
    if $V_p[|P| - 1] + \min_{S \in \tilde{T}[i]} |S| < V_c[|P| - 1]$ **then**
      $V_c[|P| - 1] \leftarrow V_p[|P| - 1] + \min_{S \in \tilde{T}[i]} |S|$

to $V_c[j] = \infty$ (line 11) for all j's each time a new degenerate segment $\tilde{T}[i]$ is read: $V_c[j] = \infty$ denotes that a partial k_E-occurrence of $P[0 \dots j]$ ending at $\tilde{T}[i]$ has not yet been found.

L_c - a list that contains the rightmost position of the prefixes of P found in L'. It is filled in by function INSERT for each prefix $P[0 \dots j]$ where $V_c[j]$ turns into a value $\neq \infty$. Before reading a new degenerate segment $\tilde{T}[i]$, it is copied into L_p and re-initialized to $\emptyset$ (line 10).

L_p - a list where the L_c list, filled in at iteration $i-1$, is copied at the beginning of each iteration i (line 10). L_p thus stores prefixes of P found in L' during the previous iteration (p stands for *previous*).

V_p - similarly, V_p stores a copy of the vector V_c of the previous position.

Algorithm k_E-EDSM needs to report each position i in $\tilde{T}$ where some k_E-occurrence of P ends with edit distance $d \leq k$, d being the minimal such value for position i. To this aim, the last position of V_c is updated with the following criterion: each time we find an occurrence of P ending at $\tilde{T}[i]$, corresponding to pair $(m-1, d)$, if $V_c[m-1] > d$ then we set $V_c[m-1] = d$ (lines 6, 16 and 22). After all $S \in \tilde{T}[i]$ have been examined, if either $V_c[m-1] \neq \infty$ or $V_p[m-1] < k$ (*i.e.,* an occurrence of P at the previous position implies an occurrence at the current one by deleting a letter in any $S \in \tilde{T}[i]$: see Example 3.2) the algorithm outputs position i together with the minimum between $V_c[m-1]$ and $V_p[m-1]+1$ (lines 24-25). If an occurrence of P at $i-1$ can lead to an occurrence at $i+1$ by deleting a whole string $S \in \tilde{T}[i]$ and a letter of any string in $\tilde{T}[i+1]$, *i.e.,* if $V_p[m-1] + \min_{S \in \tilde{T}[i]} |S| < k$, and if this value is smaller than $V_c[m-1]$, it eventually updates $V_c[m-1]$ (lines 26-28).

Example 3.2 (Running example)**.** Consider text $\tilde{T}$ and pattern $P = \texttt{GAACAA}$ of Example 2.3. The k_E-occurrence of P beginning at position 0 and ending at position 5 of $\tilde{T}$ with edit distance 0 implies an occurrence of P ending at position 6 with 1 deletion (namely, letter `C`).

3.1. Algorithm k_E-BORD

For each i and for each $S \in \tilde{T}[i]$, Step (1) of the algorithm finds the prefixes of P that are at distance at most k from any suffix of S, 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 [33]. We first recall some relevant definitions concerning the dynamic programming table [32].

Given an $m \times q$ 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 one are numbered -1 through $-(m-1)$; the diagonal starting in cell $(h, 0)$ is numbered $-h$. A *d-path* in the dynamic programming table is a path that starts in row zero and specifies a total of exactly d edit operations (substitutions, insertions, and deletions). 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-BORD takes as input a pattern P, a string $S \in \tilde{T}[i]$, $ST^*_{P,\tilde{T}[i]}$ 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 $d \leq k$ 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 $0 \leq h \leq |S| - 1$ it finds $lce_{P,S}(0, h)$. This specifies the end column of the farthest reaching 0-path on each diagonal from 0 to $|S| - 1$.

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

(1c) Any d-path that reaches the last row of the dynamic programming table indicates a k_E-occurrence of P with edit distance d that starts and ends at position i, thus the algorithm reports $(|P| - 1, d)$; any d-path that reaches the end of S in row r denotes that 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 is found by computing and comparing the following three particular paths that end on diagonal h:

R_i - 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_i takes as input the length $|X|$ of a string X, whose letters spell the rows

of the dynamic programming table, the length $|Y|$ of a string Y, whose letters spell the columns, $ST^*_{X,Y}$ and the pair row-column (r, c) where the farthest reaching $(d-1)$-path on diagonal $h+1$ ends. It outputs pair (r_i, c_i) where path R_i ends. This path represents a letter insertion in X.

R_d - consists of the dual case of R_i with a horizontal edge representing a letter deletion in X.

R_s - 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_s takes as input the length $|X|$ of a string X, whose letters spell the rows of the dynamic programming table, the length $|Y|$ of a string Y, whose letters spell the columns, $ST^*_{X,Y}$ and the pair row-column (r, c) where the farthest reaching $(d-1)$-path on diagonal h ends. It outputs pair (r_s, c_s) where path R_s ends. This path represents a letter substitution.

All such functions output $(-\infty, -\infty)$ if it is not possible to derive a path from the given parameters (*e.g.,* if r or c exceed the input dimension).

INSERT(L,(j,d),V)

```
1 if V[j] > d then
2     if V[j] = ∞ then Insert j
        in L;
3     V[j] ← d;
```

$R_i(|X|, |Y|, ST^*_{X,Y}, r, c)$

```
1 if −1 ≤ r ≤ |X| − 2 ∧ −1 ≤
    c ≤ |Y| − 1 then
2     ℓ ← lce_{X,Y}(r + 2, c + 1);
3     r_i ← r + 1 + ℓ;
4     c_i ← c + ℓ;
5     return (r_i, c_i)
6 else return (−∞, −∞);
```

Fact 2 ([32])**.** *The farthest reaching path on diagonal h is the path among R_i, R_d or R_s that extends the farthest along h.*

In each one of the iterations in k_E-BORD, a diagonal h is associated with two variables $F_p(h)$ and $F_c(h)$, storing the column reached by the farthest reaching path (FRP) on h in the previous and in the current iteration, respectively. We define $F_p(h) = F_c(h) = -\infty$ when $h \notin \{-(|P|-1), \ldots, |S|-1\}$. Notice that at most $k + |S|$ diagonals will be taken into account: the algorithm first finds the *lce*'s between $P[0]$ and $S[j]$, for all $0 \leq j \leq |S|-1$, and hence it initializes

$R_d(|X|, |Y|, ST^*_{X,Y}, r, c)$

1 **if** $-1 \leq r \leq |X| - 1 \wedge -1 \leq c \leq |Y| - 2$ **then**
2 $\ell \leftarrow lce_{X,Y}(r + 1, c + 2)$;
3 $r_d \leftarrow r + \ell$;
4 $c_d \leftarrow c + 1 + \ell$;
5 **return** (r_d, c_d)
6 **else return** $(-\infty, -\infty)$;

$R_s(|X|, |Y|, ST^*_{X,Y}, r, c)$

1 **if** $-1 \leq r \leq |X| - 2 \wedge -1 \leq c \leq |Y| - 2$ **then**
2 $\ell \leftarrow lce_{X,Y}(r + 2, c + 2)$;
3 $r_s \leftarrow r + 1 + \ell$;
4 $c_s \leftarrow c + 1 + \ell$;
5 **return** (r_s, c_s)
6 **else return** $(-\infty, -\infty)$;

$|S|$ 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 $k+|S|$ diagonals. The only difference between algorithm k_E-BORD and the algorithm by Landau and Vishkin [33] is that k_E-BORD outputs pairs (ℓ, d) corresponding to FRPs that reach the last *column* of the DP table, in addition to the ones corresponding to FRPs that reach the last row. By construction, these additional pairs correspond to k_E-matches between prefixes of P and suffixes of S. The correctness of the Landau-Vishkin algorithm thus directly implies the following lemma:

Lemma 3.3. *Algorithm k_E-BORD is correct.*

The next lemma provides the time complexity of applying k_E-BORD to every $S \in \tilde{T}[i]$, for all $i = 0, \ldots, n - 1$.

Lemma 3.4. *Given P of length m, $\tilde{T}$ of length n and size N, $ST^*_{P,\tilde{T}[i]}$ for all $i \in [0, n - 1]$, and an integer $0 < k < m$, k_E-BORD finds the minimal edit distance $\leq k$ between the prefixes of P and any suffix of $S \in \tilde{T}[i]$, as well as 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-BORD algorithm retrieves the end of three $(d - 1)$-paths (constant-time operations) and computes 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 the prefixes of P that are at distance at most k from any suffix 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

k_E-BORD$(P, S, ST^*_{P,\tilde{T}[i]}, k)$

for $h = -(k+1)$ **to** -1 **do** $F_c(h) \leftarrow h - 1$;
for $h = 0$ **to** $|S| - 1$ **do**
  $\ell \leftarrow lce_{P,S}(0, h)$;
  $F_c(h) \leftarrow \ell - 1 + h$;
  if $\ell + h = |S|$ **then report** $(\ell - 1, 0)$;
  else
    if $\ell = |P|$ **then report** $(|P| - 1, 0)$;
for $d = 1$ **to** k **do**
  for $h = -(k+1)$ **to** $|S| - 1$ **do** $F_p(h) \leftarrow F_c(h)$;
  for $h = -d$ **to** $|S| - 1$ **do**
    $(r_i, c_i) \leftarrow R_i(|P|, |S|, ST^*_{P,\tilde{T}[i]}, F_p(h+1) - (h+1), F_p(h+1))$;
    $(r_d, c_d) \leftarrow R_d(|P|, |S|, ST^*_{P,\tilde{T}[i]}, F_p(h-1) - (h-1), F_p(h-1))$;
    $(r_s, c_s) \leftarrow R_s(|P|, |S|, ST^*_{P,\tilde{T}[i]}, F_p(h) - h, F_p(h))$;
    if $\max\{c_i, c_d, c_s\} > -\infty$ **then** $F_c(h) \leftarrow \max\{c_i, c_d, c_s\}$;
    else $F_c(h) \leftarrow F_p(h)$;
    if $\max\{r_i, r_d, r_s\} = |P| - 1$ **then report** $(|P| - 1, d)$;
    if $\max\{c_i, c_d, c_s\} = |S| - 1$ **then report** $(|S| - 1 - h, d)$;

$S \in \tilde{T}[i]$. Since the size of $\tilde{T}$ is N and the total number of strings in $\tilde{T}$ is G, the result follows. □

Example 3.5 (Running example)**.** Let us consider again text $\tilde{T}$ and pattern $P =$ `GAACAA` of Example 2.3, and let $k = 1$. Suppose we already executed iteration 0, and we move to position $i = 1$, where we need to find the suffixes of all $S \in \tilde{T}[1]$ that are at edit distance at most 1 from some prefix of P. Consider then $S =$ `AA` $\in \tilde{T}[1]$. The borders at edit distance 1 are the following:

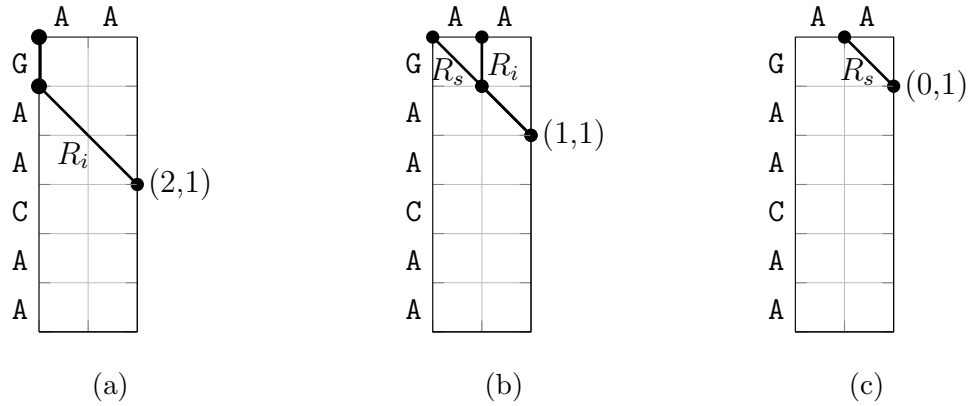
`P:`	`GAACAA`	`GAACAA`	`GAACAA`
`S:`	`-AA`	`AA`	`AA`
Output:	$(2, 1)$	$(1, 1)$	$(0, 1)$

To find them, Algorithm k_E-BORD$(P, S, ST^*_{P,\tilde{T}[1]}, 1)$ executes the following steps:

$d = 0$**:** find 0-paths on diagonals 0, 1 via *lce* queries. $lce_{P,S}(0,0) = lce_{P,S}(0,1) = 0$, thus $F_c(-2) = -3$, $F_c(-1) = -2$, $F_c(0) = -1$, $F_c(1) = 0$.

$d = 1$**:** compute farthest reaching 1-paths for diagonals -1, 0, 1 with $F_p(-2) = -3$, $F_p(-1) = -2$, $F_p(0) = -1$ and $F_p(1) = 0$ (Figure 3).

This results in $L_c = \{0, 1, 2\}$ and $V_c = [1, 1, 1, \infty, \infty, \infty]$.



(a) (b) (c)

Figure 3: (3a) diagonal $h = -1$: $R_i(|P|, |S|, ST^*_{P,S}, -1, -1)$ returns $(r_i, c_i) = (2, 1)$ (as $lce_{P,S}(1, 0) = 2$), hence $F_c(-1) = 1$, the path exhausts S and k_E-BORD returns pair $(2, 1)$. (3b) diagonal 0: both $R_i(|P|, |S|, ST^*_{P,S}, -1, 0)$ and $R_s(|P|, |S|, ST^*_{P,S}, -1, -1)$ return $(1, 1)$ (as $lce_{P,S}(1, 1) = 1$), thus they reach the end of S and k_E-BORD returns pair $(1, 1)$. (3c) diagonal 1: $R_s(|P|, |S|, ST^*_{P,S}, -1, 0)$ returns $(0, 1)$ (as $lce_{P,S}(1, 2) = 0$), the path consumes the whole S and k_E-BORD returns pair $(0, 1)$.

3.2. Algorithm k_E-EXT

In Step (2), algorithm k_E-EDSM tries to extend each partial k_E-occurrence that has started earlier in $\tilde{T}$. That is, at position i, for each $p \in L_p$ and for each string $S \in \tilde{T}[i]$, we try to extend $P[0 \dots p]$ with S. Once again, we modify the Landau-Vishkin algorithm [33] to our purpose: it suffices to look for the FRPs starting at the desired position only.

k_E-EXT takes as input a pattern P, a string $S \in \tilde{T}[i]$, the $ST^*_{P,\tilde{T}[i]}$, 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 edit operations introduced by the extension. Algorithm k_E-EXT performs a task

similar to that of k_E-BORD: (i) it builds a $|S| \times |P|$ DP table (rather than a $|P| \times |S|$ table) and (ii) instead of searching for occurrences of P starting anywhere within S, k_E-EXT checks whether the whole of S can extend the prefix $P[0 \dots j-1]$ detected at the previous text position or whether a prefix of S matches the suffix of P starting at $P[j]$ (and hence a k_E-occurrence of P has been found). In order to fulfill this task, at a high level, the algorithm executes the following steps on a table having S at the rows and P at the columns:

(2a) It finds $lce_{S,P}(0, j)$ specifying the end column of the farthest reaching 0-path on diagonal j. The value of the end column of the farthest reaching 0-path for the rest of the diagonals from $j-(k+1)$ to $j+k+1$ is set to $-\infty$ by default. This initialization ensures that any FRP on the other diagonals will originate from diagonal j.

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

(2c) Any d-path that reaches the last row of the dynamic programming table in column c denotes an occurrence of the whole S with edit distance d, and the algorithm reports (c, d), c being the position in P where this extension ends; any d-path that reaches the end of P denotes that a prefix of S is at distance d from a suffix of P starting at position j, and the algorithm reports $(|P|-1, d)$.

Example 3.6 (Running example)**.** Let us continue our running example with pattern P=`GAACAA` and text $\tilde{T}$ of Example 2.3; let again $k = 1$, and let us consider $i = 1$. After computing the borders as hinted in Example 3.5, we need to extend previous partial k_E-occurrences of P with the strings in $\tilde{T}[1]$. Consider thus $S =$`AA` $\in \tilde{T}[1]$, $L_p = \{0, 1\}$, $V_p = [0, 1, \infty, \infty, \infty, \infty]$. We try to extend $P[0]$ with S and up to $k - V_p[0] = 1$ extra errors. k_E-EXT$(1, P, S, ST^*_{P,\tilde{T}[1]}, 1)$ performs the following steps:

$d = 0$: find a 0-path on diagonal $j = 1$: Since $lce_{S,P}(0, 1) = 2$, the value $F_c(1) = 2$ is updated and the algorithm reports pair $(2, 0)$ (see Figure 4a). The value of the rest of $F_c(h)$ for h from $j - (k + 1) = -1$ to $j + k + 1 = 3$ is set to $-\infty$ by default.

k_E-EXT$(j, P, S, ST^*_{P,\tilde{T}[i]}, k)$

1 **if** $S = \varepsilon$ **then**
2 **for** $d = 0$ **to** k **do report** $(j + d, d)$;
3 **else**
4 **for** $h = j - (k + 1)$ **to** $j + k + 1$ **do** $F_c(h) \leftarrow -\infty$;
5 $\ell \leftarrow lce_{S,P}(0, j)$;
6 $F_c(j) \leftarrow \ell - 1 + j$;
7 **if** $\ell = |S|$ **then report** $(\ell + j - 1, 0)$;
8 **for** $d = 1$ **to** k **do**
9 **for** $h = j - (k + 1)$ **to** $j + k + 1$ **do** $F_p(h) \leftarrow F_c(h)$;
10 **for** $h = j - d$ **to** $j + d$ **do**
11 $(r_i, c_i) \leftarrow R_i(|S|, |P|, ST^*_{P,\tilde{T}[i]}, F_p(h + 1) - (h + 1), F_p(h + 1))$;
12 $(r_d, c_d) \leftarrow R_d(|S|, |P|, ST^*_{P,\tilde{T}[i]}, F_p(h - 1) - (h - 1), F_p(h - 1))$;
13 $(r_s, c_s) \leftarrow R_s(|S|, |P|, ST^*_{P,\tilde{T}[i]}, F_p(h) - h, F_p(h))$;
14 **if** $\max\{c_i, c_d, c_s\} > -\infty$ **then** $F_c(h) \leftarrow \max\{c_i, c_d, c_s\}$;
15 **else** $F_c(h) \leftarrow F_p(h)$;
16 **if** $\max\{r_i, r_d, r_s\} = |S| - 1$ **then report** $(F_c(h), d)$;
17 **if** $\max\{c_i, c_d, c_s\} = |P| - 1$ **then report** $(|P| - 1, d)$;

$d = 1$: compute FRPs for diagonals $j - d = 0, j = 1, j + d = 2$. Since the 0-FRP on diagonal 1 reaches the last row of the DP table already, it is not possible to extend it to 1-paths on lower diagonals: indeed, R_i, R_d and R_s all return $(-\infty, -\infty)$ for both diagonals 0 and 1. It is possible to extend it to a 1-path on diagonal 2 though, as shown in Figure 4b.

This results in $L_c = \{0, 1, 2, 3\}$ and $V_c = [1, 1, 0, 1, \infty, \infty]$.

It is easy to see that the correctness of the Landau-Vishkin algorithm directly implies the correctness of k_E-EXT, providing the following lemma.

Lemma 3.7. *Algorithm k_E-EXT is correct.*

Lemma 3.8. *Given a prefix of P, a string $S \in \tilde{T}[i]$, $ST^*_{P,\tilde{T}[i]}$, and an integer $0 < k < m$, k_E-EXT extends the prefix of P with S in time $\mathcal{O}(k^2)$.*

Proof. The k_E-EXT 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

Figure 4: (4a) diagonal $j = 1$: $lce_{S,P}(0, 1) = 2$, thus the 0-FRP reaches the last row of the table and k_E-EXT correctly returns pair $(2, 0)$.
(4b) diagonal $j+d = 2$: $R_d(|S|, |P|, ST^*_{P,S}, 1, 2)$ returns $(r_d, c_d) = (1, 3)$ (as $lce_{S,P}(2, 4) = 0$): since $r_d = |S| - 1$, this path reaches the last row of the DP table, and k_E-EXT correctly returns pair $(3, 1)$.

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 $\mathcal{O}(1 + 3 + \cdots + (2k + 1))=\mathcal{O}(k^2)$. □

Correctness. As for the correctness of algorithm k_E-EDSM, Lemmas 3.3 and 3.7 ensure that borders and extensions are correctly computed; we further observe that, by storing just the minimum edit distance for every partial k_E-occurrence of P at a certain position $\tilde{T}[i]$, we do not miss any occurrence of P nor report spurious occurrences. It is easy to find examples where, should we store a single value different from the minimum, we would either fail to report an occurrence (in case we stored a greater value), or report a spurious one (if we stored a lower value). On the other hand, any additional distance value beyond the minimum would be redundant according to the following observation: assume $P[0 \dots \ell]$ has two partial k_E-occurrences at $\tilde{T}[i]$ with distances, respectively, d and $d' > d$. If $P[\ell + 1 \dots m - 1]$ matches a prefix of some string in $\tilde{T}[i + 1]$ with e errors and $e + d' \leq k$, then also $e + d \leq k$. Therefore, it suffices to store distance d associated to $P[0 \dots \ell]$ to output an occurrence of P (or an extended partial one) at $\tilde{T}[i + 1]$.

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

Lemma 3.9. *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 on-line in time $\mathcal{O}(k^2mG + kN)$, G being the total number of strings in $\tilde{T}$.*

Proof. At the i-th iteration, algorithm k_E-EDSM tries to extend each $p \in L_p$ with each string $S \in \tilde{T}[i]$. By Lemma 3.1, to build $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, requires time $\mathcal{O}(N)$. By Lemma 3.8, to extend a single prefix with a string S

can be done in time $\mathcal{O}(k^2)$. Since in L_p there are at most $|P| = m$ prefixes, to extend them all 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 3.4, the prefixes of P that are at distance at most k from any suffix of S as well as 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 $\mathcal{O}(N + k^2mG + k^2G + kN) = \mathcal{O}(k^2mG + kN)$. The algorithm is on-line in the sense that any occurrence of the pattern ending at position i is reported before reading $\tilde{T}[i + 1]$. □

We thus have the following result.

Theorem 3.10. *The k_E-EDSM problem can be solved on-line in time $\mathcal{O}(k^2mG+kN)$ and space $\mathcal{O}(m)$ for constant-sized alphabets.*

Proof. To obtain the space bound $\mathcal{O}(m)$, we need to slightly modify Algorithm k_E-EDSM in the following way: 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, $1 \leq j \leq \lceil \frac{|S|}{m} \rceil$. Instead of building $ST^*_{P,\tilde{T}[i]}$ for each degenerate letter $\tilde{T}[i]$, the algorithm now builds ST^*_{P,W_j} for each $1 \leq j \leq \lceil \frac{|S|}{m} \rceil$ 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-BORD and k_E-EXT 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 $\max\{|\tilde{T}[i]|, \lceil \frac{N_i}{m} \rceil\}$, 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-BORD 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-EXT, 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, 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 $\mathcal{O}(k^2mG + kN)$. □

To sum up, the following example shows a full iteration of k_E-EDSM.

Example 3.11 (Running example)**.** Consider the usual pattern P=GAACAA and text $\tilde{T}$ of Example 2.3, $k = 1$, $i = 1$. Examples 3.5 and 3.6 considered string $S = \texttt{AA} \in \tilde{T}[1]$ to compute borders and extensions respectively, so that $L_c = \{0, 1, 2, 3\}$ and $V_c = [1, 1, 0, 1, \infty, \infty]$ so far: consider now $S = \texttt{AG} \in \tilde{T}[1]$. k_E-BORD$(P, S, ST'_{P,\tilde{T}[1]}1)$ returns pair $(0, 0)$: since 0 already belongs to L_c and $V_c[0] = 1 > 0$, we set $V_c[0] = 0$, so that $L_c = \{0, 1, 2, 3\}$ and $V_c = [0, 1, 0, 1, \infty, \infty]$. Now k_E-EXT$(1, P, S, ST'_{P,\tilde{T}[1]}1)$ returns $(2, 1)$: 2 is already in the list, but since $V_c[2] = 0 < 1$ we leave it as it is.
k_E-EXT$(2, P, S, ST'_{P,\tilde{T}[1]}0)$ does not provide any additional extensions (as it is not possible to extend $P[0, 1] = \texttt{GA}$ with $S = \texttt{AG}$ and no additional edit operations), so we move to $S = \varepsilon \in \tilde{T}[1]$. Of course the empty string can only be used to extend the prefixes already matched at $\tilde{T}[0]$: in this case, k_E-EXT$(1, P, S, ST'_{P,\tilde{T}[1]}1)$ outputs pairs $(1, 0)$ and $(1, 1)$, k_E-EXT$(2, P, S, ST'_{P,\tilde{T}[1]}0)$ reports pair $(2, 1)$, which are all stored in L_c and V_c already. The whole iteration thus ends with $L_c = \{0, 1, 2, 3\}$ and $V_c = [0, 1, 0, 1, \infty, \infty]$.

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. We assume a constant-sized alphabet. The two algorithms differ in the functions used to perform Step (1) (k_H-BORD rather than k_E-BORD) and Step (2) (k_H-EXT rather than k_E-EXT). 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-BORD and k_E-EXT, with k_H-BORD and k_H-EXT such pairs now represent partial k_H-occurrences of P in $\tilde{T}$.

k_H-BORD(P,S,$ST^*_{P,\tilde{T}[i]}$,k)

for $h = 0$ **to** $|S| - 1$ **do**
  $d \leftarrow 0$; $j \leftarrow 0$; $h' \leftarrow h$;
  while $d \leq k$ **do**
    $\ell \leftarrow lce_{P,S}(j, h')$;
    if $h' + \ell = |S|$ **then report** $(|S| - h - 1, d)$;
    else
      if $h' + \ell + 1 = |S| \wedge d + 1 \leq k$ **then report** $(|S| - h, d + 1)$;
      else
        if $j + \ell = |P|$ **then report** $(|P| - 1, d)$;
        else
          if $j + \ell + 1 = |P| \wedge d + 1 \leq k$ **then report** $(|P| - 1, d + 1)$;
          else $d \leftarrow d + 1$; $j \leftarrow j + \ell + 1$; $h' \leftarrow h' + \ell + 1$;

k_H-EXT(j,P,S,$ST^*_{P,\tilde{T}[i]}$,k)

if $S = \varepsilon$ **then report** $(j, 0)$;
else
  $d \leftarrow 0$; $h \leftarrow 0$; $j' \leftarrow j$;
  while $d \leq k$ **do**
    $\ell \leftarrow lce_{S,P}(h, j')$;
    if $h + \ell = |S|$ **then report** $(j' + \ell - 1, d)$;
    else
      if $h + \ell + 1 = |S| \wedge d + 1 \leq k$ **then report** $(j' + \ell, d + 1)$;
      else
        if $j' + \ell = |P|$ **then report** $(|P| - 1, d)$;
        else
          if $j' + \ell + 1 = |P| \wedge d + 1 \leq k$ **then report** $(|P| - 1, d + 1)$;
          else $d \leftarrow d + 1$; $h \leftarrow h + \ell + 1$; $j' \leftarrow j' + \ell + 1$;

At the i-th iteration, for each $S \in \tilde{T}[i]$ and any position h in S, k_H-BORD 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 $\ell = lce_{P,S}(0,h)$, it finds out that $P[0 \dots \ell-1]$ and $S[h \dots h+\ell-1]$ match exactly and $P[\ell] \neq S[h+\ell]$. It can then skip one position in both strings (the mismatch $P[\ell] \neq S[h+\ell]$), increasing the error-counter d by 1, and compute the $lce_{P,S}(\ell+1, h+\ell+1)$. 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 substitutions 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-BORD in k_H-EDSM.

Lemma 4.1. *Given P of length m, $\tilde{T}$ of length n and size N, the $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, and an integer $0 < k < m$, k_H-BORD finds the minimal Hamming distance $\leq k$ between the prefixes of P and any suffix of $S \in \tilde{T}[i]$, as well as 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-BORD 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 $p \in L_p$ similar to algorithm k_E-EDSM) k_H-EXT 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 k_H-occurrence can be extended with the whole S starting from position $j = p+1$ 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-BORD.

Lemma 4.2. *Given P of length m, $\tilde{T}$ of length n and size N, the $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, and an integer $0 < k < m$, k_H-EXT 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-EXT 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. Since there are G such strings, the overall time is $\mathcal{O}(kmG)$. □

Lemma 4.3. *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 on-line 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 $p \in L_p$ with each string $S \in \tilde{T}[i]$. By Lemma 3.1, building $ST^*_{P,\tilde{T}[i]}$, for all $i \in [0, n-1]$, requires time $\mathcal{O}(N)$. By Lemma 4.2, 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 4.1, the prefixes of P that are at distance at most k from any suffix of S as well as the k_H-occurrences of P that start and end at position i can be found in time $\mathcal{O}(kN)$ in total. Summing up, the overall time complexity for the whole k_H-EDSM algorithm is then $\mathcal{O}(N + kmG + kN) = \mathcal{O}(kmG + kN)$, 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 3.10 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 4.4. *The k_H-EDSM problem can be solved on-line in time $\mathcal{O}(kmG + kN)$ and space $\mathcal{O}(m)$ for constant-sized alphabets.*

5. Extension to General Integer Alphabets

The algorithms presented in the previous sections are designed for constant-sized alphabets only: a straightforward switch to the general integer alphabets case would entail an increase in the time required to build the suffix trees, and hence in the complexity of the algorithm. In this section we show how to extend our results to the case of general integer alphabets, while

maintaining the same time and space complexity. We obtain this by using perfect hashing [34] to build the suffix tree of a window of length (at most) $2m$ in $\mathcal{O}(m)$ time for general integer alphabets. The procedure consists of a preprocessing phase followed by the proper construction of the suffix tree.

Preprocessing: We hash the letters of pattern P using perfect hashing. For each key, we assign a rank value from $\{1, \ldots, m\}$. This takes $\mathcal{O}(m)$ (expected) time and space [34].

Construction: When reading a window W of length (at most) $2m$ of the text we look up its letters using the hash table constructed during the preprocessing phase. If a letter is in the hash table we replace it in W by its rank value; otherwise we replace it by rank $m+1$. This operation takes $\mathcal{O}(1)$ time [34]. We can now construct the suffix tree of W in $\mathcal{O}(m)$ time and $\mathcal{O}(m)$ space using Farach's suffix tree construction algorithm [35]. This is because string W is over $\{1, \ldots, m+1\}$.

We thus have the following lemma.

Lemma 5.1. *Given P of length m and $\tilde{T}$ of length n and size N, to build ST^*_{P,W_j} for each window W_j of length $2m$, $1 \leq j \leq \lceil \frac{|S|}{m} \rceil$, and for each $S \in \tilde{T}[i]$, for all $i \in [0, n-1]$, requires total time $\mathcal{O}(N)$ for general integer alphabets.*

By plugging this lemma into the algorithms of, respectively, Section 3 and Section 4, we obtain the following results.

Theorem 5.2. *The k_E-EDSM problem can be solved on-line in time $\mathcal{O}(k^2mG + kN)$ and space $\mathcal{O}(m)$ for general integer alphabets.*

Theorem 5.3. *The k_H-EDSM problem can be solved on-line in time $\mathcal{O}(kmG + kN)$ and space $\mathcal{O}(m)$ for general integer alphabets.*

6. Final Remarks

In this paper we introduced two algorithms for finding all approximate matches of a pattern P of length m in an ED text $\tilde{T}$ of length n and size N: an $\mathcal{O}(kmG+kN)$-time algorithm for Hamming distance; and an $\mathcal{O}(k^2mG+kN)$-time algorithm for edit distance, where G is the total number of strings in $\tilde{T}$ and k is the maximum distance allowed. Both algorithms are on-line, their working space is $\mathcal{O}(m)$, and they work for general integer alphabets.

There are at least two directions for future work. The first one is to improve the time complexity for these problems by perhaps removing the dependency on parameter G. The second direction is to develop algorithms for searching multiple patterns simultaneously under the approximate setting.

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] G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, G. Rosone, Even faster elastic-degenerate string matching via fast matrix multiplication, in: C. Baier, I. Chatzigiannakis, P. Flocchini, S. Leonardi (Eds.), 46th International Colloquium on Automata, Languages and Programming, ICALP 2019, Patras, Greece, 8-12 July 2019, Vol. 132 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, pp. 16:1 – 16:15.
URL `https://doi.org/10.4230/LIPIcs.ICALP.2019.16`

[2] The Computational Pan-Genomics Consortium, Computational pan-genomics: status, promises and challenges, Briefings in Bioinformatics 19 (1) (2018) 118–135. doi:10.1093/bib/bbw089.
URL `https://doi.org/10.1093/bib/bbw089`

[3] L. Huang, V. Popic, S. Batzoglou, Short read alignment with populations of genomes, Bioinformatics 29 (13) (2013) 361–370. doi:10.1093/bioinformatics/btt215.
URL `https://doi.org/10.1093/bioinformatics/btt215`

[4] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, O. Weimann, Random access to grammar-compressed strings and trees, SIAM J. Comput. 44 (3) (2015) 513–539. doi:10.1137/130936889.
URL `https://doi.org/10.1137/130936889`

[5] T. Gagie, P. Gawrychowski, S. J. Puglisi, Faster approximate pattern matching in compressed repetitive texts, in: T. Asano, S. Nakano, Y. Okamoto, O. Watanabe (Eds.), Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings, Vol. 7074 of Lecture Notes in Computer Science, Springer, 2011, pp. 653–662. doi:10.1007/978-3-642-25591-5_67.
URL https://doi.org/10.1007/978-3-642-25591-5_67

[6] G. Navarro, Indexing highly repetitive collections, in: S. Arumugam, W. F. Smyth (Eds.), Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers, Vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 274–279. doi:10.1007/978-3-642-35926-2_29.
URL https://doi.org/10.1007/978-3-642-35926-2_29

[7] S. Wandelt, U. Leser, String searching in referentially compressed genomes, in: A. L. N. Fred, J. Filipe (Eds.), KDIR 2012 - Proceedings of the International Conference on Knowledge Discovery and Information Retrieval, Barcelona, Spain, 4 - 7 October, 2012, SciTePress, 2012, pp. 95–102.

[8] T. Gagie, S. J. Puglisi, Searching and indexing genomic databases via kernelization, Frontiers in Bioengineering and Biotechnology 3 (2015) 12. doi:10.3389/fbioe.2015.00012.
URL https://www.frontiersin.org/article/10.3389/fbioe.2015.00012

[9] C. Barton, C. Liu, S. P. Pissis, On-line pattern matching on uncertain sequences and applications, in: T. H. Chan, M. Li, L. Wang (Eds.), Combinatorial Optimization and Applications - 10th International Conference, COCOA 2016, Hong Kong, China, December 16-18, 2016, Proceedings, Vol. 10043 of Lecture Notes in Computer Science, Springer, 2016, pp. 547–562. doi:10.1007/978-3-319-48749-6_40.
URL https://doi.org/10.1007/978-3-319-48749-6_40

[10] T. Kociumaka, S. P. Pissis, J. Radoszewski, Pattern matching and consensus problems on weighted sequences and profiles, in: S. Hong (Ed.), 27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia, Vol. 64 of LIPIcs,

Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 46:1–46:12. doi:10.4230/LIPIcs.ISAAC.2016.46.
URL https://doi.org/10.4230/LIPIcs.ISAAC.2016.46

[11] C. S. Iliopoulos, R. Kundu, S. P. Pissis, Efficient pattern matching in elastic-degenerate texts, in: F. Drewes, C. Martín-Vide, B. Truthe (Eds.), Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings, Vol. 10168 of Lecture Notes in Computer Science, 2017, pp. 131–142. doi:10.1007/978-3-319-53733-7_9.
URL https://doi.org/10.1007/978-3-319-53733-7_9

[12] K. R. Abrahamson, Generalized string matching, SIAM J. Comput. 16 (6) (1987) 1039–1051. doi:10.1137/0216067.
URL https://doi.org/10.1137/0216067

[13] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, T. Walen, Covering problems for partial words and for indeterminate strings, Theor. Comput. Sci. 698 (2017) 25–39. doi:10.1016/j.tcs.2017.05.026.
URL https://doi.org/10.1016/j.tcs.2017.05.026

[14] C. S. Iliopoulos, J. Radoszewski, Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties, in: R. Grossi, M. Lewenstein (Eds.), 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel, Vol. 54 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 8:1–8:12. doi:10.4230/LIPIcs.CPM.2016.8.
URL https://doi.org/10.4230/LIPIcs.CPM.2016.8

[15] N. Pisanti, H. Soldano, M. Carpentier, J. Pothier, A relational extension of the notion of motifs: Application to the common 3d protein substructures searching problem, Journal of Computational Biology 16 (12) (2009) 1635–1660. doi:10.1089/cmb.2008.0019.
URL https://doi.org/10.1089/cmb.2008.0019

[16] H. Soldano, A. Viari, M. Champesme, Searching for flexible repeated patterns using a non-transitive similarity relation, Pattern Recognition Letters 16 (3) (1995) 233–246. doi:10.1016/0167-8655(94)00095-K.
URL https://doi.org/10.1016/0167-8655(94)00095-K

[17] M. Alzamel, L. A. K. Ayad, G. Bernardini, R. Grossi, C. S. Iliopoulos, N. Pisanti, S. P. Pissis, G. Rosone, Degenerate string comparison and applications, in: L. Parida, E. Ukkonen (Eds.), 18th International Workshop on Algorithms in Bioinformatics, WABI 2018, August 20-22, 2018, Helsinki, Finland, Vol. 113 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 21:1–21:14. doi:10.4230/LIPIcs.WABI.2018.21.
URL https://doi.org/10.4230/LIPIcs.WABI.2018.21

[18] The 1000 Genomes Project Consortium, A global reference for human genetic variation, Nature 526 (7571) (2015) 68–74. doi:10.1038/nature15393.
URL https://www.nature.com/articles/nature15393

[19] S. P. Pissis, A. Retha, Dictionary matching in elastic-degenerate texts with applications in searching VCF files on-line, in: G. D'Angelo (Ed.), 17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L'Aquila, Italy, Vol. 103 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 16:1–16:14. doi:10.4230/LIPIcs.SEA.2018.16.
URL https://doi.org/10.4230/LIPIcs.SEA.2018.16

[20] J. Holub, W. F. Smyth, S. Wang, Fast pattern-matching on indeterminate strings, J. Discrete Algorithms 6 (1) (2008) 37–50. doi:10.1016/j.jda.2006.10.003.
URL https://doi.org/10.1016/j.jda.2006.10.003

[21] R. Rahn, D. Weese, K. Reinert, Journaled string tree - a scalable data structure for analyzing thousands of similar genomes on your laptop, Bioinformatics 30 (24) (2014) 3499–3505. doi:10.1093/bioinformatics/btu438.
URL https://doi.org/10.1093/bioinformatics/btu438

[22] S. Maciuca, C. del Ojo Elias, G. McVean, Z. Iqbal, A natural encoding of genetic variation in a burrows-wheeler transform to enable mapping and genome inference, in: M. C. Frith, C. N. S. Pedersen (Eds.), Algorithms in Bioinformatics - 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings, Vol. 9838 of Lecture Notes in Computer Science, Springer, 2016, pp. 222–233. doi:10.1007/978-3-319-43681-4_18.
URL https://doi.org/10.1007/978-3-319-43681-4_18

[23] J. Sirén, Indexing variation graphs, in: S. P. Fekete, V. Ramachandran (Eds.), Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017., SIAM, 2017, pp. 13–27. doi:10.1137/1.9781611974768.2.
URL https://doi.org/10.1137/1.9781611974768.2

[24] J. C. Na, H. Kim, S. Min, H. Park, T. Lecroq, M. Léonard, L. Mouchard, K. Park, Fm-index of alignment with gaps, Theor. Comput. Sci. 710 (2018) 148–157. doi:10.1016/j.tcs.2017.02.020.
URL https://doi.org/10.1016/j.tcs.2017.02.020

[25] R. A. Baeza-Yates, C. H. Perleberg, Fast and practical approximate string matching, Inf. Process. Lett. 59 (1) (1996) 21–27. doi:10.1016/0020-0190(96)00083-X.
URL https://doi.org/10.1016/0020-0190(96)00083-X

[26] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, Basic local alignment search tool, Journal of Molecular Biology 215 (3) (1990) 403–410. doi:https://doi.org/10.1016/S0022-2836(05)80360-2.
URL http://www.sciencedirect.com/science/article/pii/S0022283605803602

[27] R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani, L. Versari, On-line pattern matching on similar texts, in: J. Kärkkäinen, J. Radoszewski, W. Rytter (Eds.), 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland, Vol. 78 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, pp. 9:1–9:14. doi:10.4230/LIPIcs.CPM.2017.9.
URL https://doi.org/10.4230/LIPIcs.CPM.2017.9

[28] K. Aoyama, Y. Nakashima, T. I, S. Inenaga, H. Bannai, M. Takeda, Faster online elastic degenerate string matching, in: G. Navarro, D. Sankoff, B. Zhu (Eds.), Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China, Vol. 105 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 9:1–9:10. doi:10.4230/LIPIcs.CPM.2018.9.
URL https://doi.org/10.4230/LIPIcs.CPM.2018.9

[29] A. Cisłak, S. Grabowski, J. Holub, Sopang: online text searching over a pan-genome, Bioinformatics 34 (24) (2018) 4290–4292. doi:10.1093/bioinformatics/bty506.
URL http://dx.doi.org/10.1093/bioinformatics/bty506

[30] M. Rautiainen, V. Makinen, T. Marschall, Bit-parallel sequence-to-graph alignment, Bioinformaticsdoi:10.1093/bioinformatics/btz162.
URL https://doi.org/10.1093/bioinformatics/btz162

[31] G. Bernardini, N. Pisanti, S. P. Pissis, G. Rosone, Pattern matching on elastic-degenerate text with errors, in: G. Fici, M. Sciortino, R. Venturini (Eds.), String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings, Vol. 10508 of Lecture Notes in Computer Science, Springer, 2017, pp. 74–90. doi:10.1007/978-3-319-67428-5_7.
URL https://doi.org/10.1007/978-3-319-67428-5_7

[32] D. Gusfield, Algorithms on strings, trees, and sequences, Cambridge University Press New York, New York, 1997.

[33] G. M. Landau, U. Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in: J. Hartmanis (Ed.), Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA, ACM, 1986, pp. 220–230. doi:10.1145/12130.12152.
URL https://doi.org/10.1145/12130.12152

[34] M. L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with $\mathcal{O}(1)$ worst case access time, J. ACM 31 (3) (1984) 538–544. doi:10.1145/828.1884.

[35] M. Farach, Optimal suffix tree construction with large alphabets, in: 38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997, IEEE Computer Society, 1997, pp. 137–143. doi:10.1109/SFCS.1997.646102.
URL https://doi.org/10.1109/SFCS.1997.646102