Information and Computation ••• (••••) ••••••

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Efficient pattern matching in elastic-degenerate strings [☆]

Costas S. Iliopoulos [a], Ritu Kundu [b,*], Solon P. Pissis [c]

[a] *Department of Informatics, King's College London, London, UK*
[b] *School of Computing, National University of Singapore, Singapore, Singapore*
[c] *CWI, Amsterdam, the Netherlands*

ARTICLE INFO

Article history:
Received 10 June 2017
Received in revised form 14 January 2020
Accepted 16 January 2020
Available online xxxx

Keywords:
Algorithms on strings
Degenerate strings
Indeterminate strings
Elastic-degenerate strings
Gapped strings

ABSTRACT

Motivated by applications in bioinformatics and image searching, in what follows, we study the classic pattern matching problem in the context of elastic-degenerate strings: the generalised notion of gapped strings. An *elastic-degenerate string* can be seen as an ordered collection of k strings interleaved by $k-1$ *elastic-degenerate symbols*, where each such elastic-degenerate symbol corresponds to a set of two or more variable-length strings. We present efficient algorithms for two variants of the pattern matching problem on elastic-degenerate strings: first, for a solid pattern and an elastic-degenerate text; second, for an elastic-degenerate pattern and a solid text. A proof-of-concept implementation of the former is provided.

© 2020 Elsevier Inc. All rights reserved.

1. Introduction

Uncertainty in sequential data (strings) can be modelled using various representations. One such representation is a *degenerate string*, which is defined by the existence of one or more positions that are represented by sets of symbols from an alphabet Σ, unlike a solid (or deterministic, standard) string characterised by a single symbol at each position. For instance, $\left[\begin{smallmatrix}\mathtt{a}\\\mathtt{b}\end{smallmatrix}\right]\mathtt{ac}\left[\begin{smallmatrix}\mathtt{b}\\\mathtt{c}\end{smallmatrix}\right]\mathtt{a}\left[\begin{smallmatrix}\mathtt{a}\\\mathtt{b}\\\mathtt{c}\end{smallmatrix}\right]$ is a degenerate string of length 6 over $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$; and $\mathtt{abaababa}$ is a solid string of length 8 over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. When a string is solid, we simply refer to it as string.

A more restrictive variant of degenerate strings – which allows at a given position a subset consisting of either a single letter or all the letters of Σ – was proposed by Fischer and Paterson in their seminal work [2]. For example, $\mathtt{ab}\Diamond\mathtt{ac}$ is an instance of a string of length 5 where the third position carries a *hole* or *don't care* or *wildcard* symbol (usually represented by $\Diamond$ or $*$) which can match any letter from the alphabet including itself. This restrictive model has been called "partial words" or "strings with wild cards/holes/don't cares" in recent years; [3] presents a comprehensive survey on partial words.

A *gapped string* is another way to capture uncertainty: it is an ordered collection of standard strings separated by variable-length gaps defined by an ordered collection of intervals [4]. Simply, a gapped string P can be represented as follows [5]: $P = P_1 *^{a_1,b_1} P_2 *^{a_2,b_2} P_3 \cdots *^{a_{\ell-1},b_{\ell-1}} P_\ell$, where $*$ is a *wildcard* symbol (also called *don't care* symbol or *hole*) that matches any symbol from alphabet Σ; $\forall i \in [1, \ell]$ each P_i is a string over Σ; and $\forall i \in [1, \ell-1]$ each pair (a_i, b_i) represents the gap (minimum and maximum number of wildcard symbols, respectively) between two consecutive strings P_i and P_{i+1}.

[☆] This is an extended version of an article presented at the Language and Automata Theory and Applications - 11th International Conference, LATA 2017 [1].
[*] Corresponding author.
E-mail addresses: costas.iliopoulos@kcl.ac.uk (C.S. Iliopoulos), dcsritu@nus.edu.sg (R. Kundu), solon.pissis@cwi.nl (S.P. Pissis).

https://doi.org/10.1016/j.ic.2020.104616
0890-5401/© 2020 Elsevier Inc. All rights reserved.

In [1], we introduced another representation to encapsulate uncertainty in sequential data—which we call *elastic-degenerate strings*—by extending and combining the ideas of gapped strings and degenerate strings. An *elastic-degenerate string* is a string where an *elastic-degenerate symbol* can occur at one or more positions; each such symbol corresponds to a set of two or more variable-length strings. Another way to visualise an elastic-degenerate string is to see it as an ordered collection of k strings interleaved by $k-1$ elastic-degenerate symbols. For instance, $\texttt{bc}\begin{bmatrix}\texttt{ab}\\\texttt{aab}\\\texttt{aca}\end{bmatrix}\texttt{ca}\begin{bmatrix}\texttt{abcab}\\\texttt{cba}\end{bmatrix}\texttt{bb}$ is an example of an elastic-degenerate string over $\Sigma = \{\texttt{a},\texttt{b},\texttt{c}\}$, where $k = 3$.

This generalisation of the concept of *degeneracy* is motivated by several data mining problems [6] which can be reduced to the core task of discovering occurrences of one or more patterns in a text that can best be described as an ordered collection of strings interleaved by sets of variable-length strings.

More specifically, in genomics an important class of problems is to study within-species genetic variation; state-of-the-art solutions for this class comprises matching (*mapping*) short strings (called *reads*) to a longer genomic sequence (canonical *reference genome* obtained through assembly). Owing to the high diversity among biologically relevant genomic regions in many organisms, the population level complexities cannot be captured by the linear structure of a reference genome [7]. Consequently, the recent research trend has shifted towards using alternative representations of genomic sequence for population-based genome assembly [8–11]. One such representation that encodes a set of related genomes with variations in the reference genome itself (called Population Reference Genome in [11]), can essentially be seen as an elastic-degenerate string.

The problem of pattern matching and discovery in the context of degenerate, partial, and gapped strings has been studied extensively: some of the efficient and practical algorithms for pattern matching on degenerate strings developed over the last decade can be found in [12–16]; [3] presents a comprehensive survey on partial words; [17] and references therein provide an overview of combinatorial algorithms for problems involving gapped strings. However, the precise identification of allowed strings (with varying lengths) in a gap makes the matching problem in the context of elastic-degenerate strings, algorithmically more challenging. Nevertheless, since we introduced the notion in [1], several works have been added to the literature for solving the problem of finding patterns in elastic-degenerate texts involving a range of settings [18–23]; see [24] for a summary.

Another practical relevance of this generalisation lies in the nature of real data and the querying process in many applications where occurrences of an elastic-degenerate pattern in a set are to be found. For example, to expedite the mapping, a set of related reads with small variations can be encoded as an elastic-degenerate string. In addition, matching of elastic-degenerate patterns in the given data also finds application in areas such as computer vision or image processing; one such example can be an image-search query, looking for a pattern in which some portions remain unchanged but are interspersed by areas which might be occluded with different objects.

In this article, we extend the work presented in [1] that studied one aspect of the classic pattern matching problem in the context of elastic-degenerate strings, namely, the pattern matching problem given a solid pattern and an elastic-degenerate text. Here, we also study the converse notion—pattern matching given an elastic-degenerate pattern and a solid text—and present the first algorithm (based on a graph-theoretic approach using dynamic programming technique) to solve the problem.

The rest of the article is organised in the following format: The vocabulary, notions, and tools that will be used in this article are introduced in the next section. The formal problem definitions and algorithms (along with their analysis) of the two variants of the pattern matching problem for elastic-degenerate string are described in Section 3 and Section 4, respectively. Finally, the conclusions drawn are delineated in Section 5.

2. Preliminaries

The following two subsections lay the groundwork for the rest of the manuscript.

2.1. Terminology

We begin with basic definitions and notation. We think of a *string* X of *length* $n = |X|$ as an array $X[1..n]$, where every $X[i]$, $1 \leq i \leq n$, is a *symbol* drawn from some fixed *alphabet* Σ of size $|\Sigma| = \mathcal{O}(1)$. The *empty string* of length 0 is denoted by ε. Σ^* denotes the set of all strings over alphabet Σ including the empty string ε. A string Y is a *factor* of a string X if there exist two strings U and V, such that $X = UYV$. We say that there is an *occurrence* of Y in X, or simply, that Y *occurs* in X, when Y is a factor of X. The starting position of an occurrence, say i, is called *head* of the occurrence and its ending position $i + |Y|$ - 1 is called *tail* of the occurrence. Note that an empty string occurs at each position in a given string. Consider the strings X, Y, U, and V, such that $X = UYV$. If $U = \varepsilon$, then Y is a *prefix* of X. If $V = \varepsilon$, then Y is a *suffix* of X.

A *degenerate symbol* $\tilde{\sigma}$ over an alphabet Σ is a non-empty subset of Σ, i.e., $\tilde{\sigma} \subseteq \Sigma$ and $\tilde{\sigma} \neq \emptyset$. $|\tilde{\sigma}|$ denotes the size of the set and we have $1 \leq |\tilde{\sigma}| \leq |\Sigma|$. A *degenerate string* is built over the potential $2^{|\Sigma|} - 1$ non-empty subsets of symbols of Σ. In other words, a degenerate string $\tilde{X} = \tilde{X}[1..n]$ is a string such that every $\tilde{X}[i]$ is a degenerate symbol, $1 \leq i \leq n$. If $|\tilde{x}[i]| = 1$, that is, $\tilde{X}[i]$ represents a single symbol of Σ, we say that $\tilde{X}[i]$ is a *solid* symbol and i is a *solid position*. Otherwise $\tilde{X}[i]$ and i are said to be a *non-solid symbol* and a *non-solid position*, respectively. For example, $\begin{bmatrix}\texttt{a}\\\texttt{b}\end{bmatrix}\texttt{ac}\begin{bmatrix}\texttt{b}\\\texttt{c}\end{bmatrix}\texttt{a}\begin{bmatrix}\texttt{a}\\\texttt{b}\\\texttt{c}\end{bmatrix}$ is a degenerate string of length 6 over $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. A string consisting of only solid symbols is called a *solid string* or, simply, a string.

Here, we briefly describe the terminology building the concept of elastic-degeneracy in strings (most of which has been established in [1]).

Definition 1 *(Seed: S).* A *seed* S is a (possibly empty) string over Σ.

Definition 2 *(Elastic-degenerate symbol: ξ).* An *elastic-degenerate symbol* ξ, over a given alphabet Σ, is a set of two or more strings over Σ (i.e. $\xi \subseteq \Sigma^*$ and $|\xi| > 1$). An elastic-degenerate symbol ξ is denoted by $\begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_{|\xi|} \end{bmatrix}$, where each $E_i,\ 1 \le i \le |\xi|$, is a solid string. The minimum (resp. maximum) length in ξ, denoted by $|\xi|_{min}$ (resp. $|\xi|_{max}$), is the length of the shortest (resp. longest) string in the set.

Definition 3 *(Elastic-degenerate string: $\hat{X}$).* An *elastic-degenerate string* $\hat{X}$, over a given alphabet Σ, is a sequence $S_1\xi_1S_2\xi_2S_3\ldots S_{k-1}\xi_{k-1}S_k$, where $S_i,\ 1 \le i \le k$, is a seed and $\xi_i,\ 1 \le i \le k-1$ is an elastic-degenerate symbol.

An elastic degenerate string $\hat{X}$ can be visualised as follows:

$$\hat{X} = S_1 \begin{bmatrix} E_{1,1} \\ E_{1,2} \\ \vdots \\ E_{1,|\xi_1|} \end{bmatrix} S_2 \begin{bmatrix} E_{2,1} \\ E_{2,2} \\ \vdots \\ E_{2,|\xi_2|} \end{bmatrix} S_3 \ldots S_{k-1} \begin{bmatrix} E_{k-1,1} \\ E_{k-1,2} \\ \vdots \\ E_{k-1,|\xi_{k-1}|} \end{bmatrix} S_k.$$

Example 2.1. $\hat{X} = \texttt{abbc} \begin{bmatrix} \texttt{ab} \\ \texttt{aab} \\ \texttt{acca} \end{bmatrix} \texttt{cca} \begin{bmatrix} \texttt{aabcab} \\ \texttt{cba} \end{bmatrix} \texttt{bb}$ is an elastic-degenerate string, where we have the following:

- Three seeds: $S_1 = \texttt{abbc}$, $S_2 = \texttt{cca}$, and $S_3 = \texttt{bb}$.
- Two elastic-degenerate symbols:
  $\xi_1 = \begin{bmatrix} \texttt{ab} \\ \texttt{aab} \\ \texttt{acca} \end{bmatrix}$ and $\xi_2 = \begin{bmatrix} \texttt{aabcab} \\ \texttt{cba} \end{bmatrix}$.
- For ξ_1: $E_{1,1} = \texttt{ab}$, $E_{1,2} = \texttt{aab}$, $E_{1,3} = \texttt{acca}$; minimum length is 2 (length of $E_{1,1}$); and maximum length is 4 (length of $E_{1,3}$).
- For ξ_2: $E_{2,1} = \texttt{aabcab}$, $E_{2,2} = \texttt{cba}$; minimum length is 3 (length of $E_{2,2}$); and maximum length is 6 (length of $E_{2,1}$).

Observe the use of $\hat{X}$ to distinguish an elastic-degenerate string from a solid string X or a degenerate string $\tilde{X}$. In the following, we define a few characteristics of a given elastic-degenerate string $\hat{X}$ with k seeds.

Definition 4 *(Total size: $\|\hat{X}\|$).* The *total size* of $\hat{X}$, denoted by $\|\hat{X}\|$, is defined as the sum of the total length of its seeds and the total length of all the strings in each of its elastic-degenerate symbols: $\|\hat{X}\| = \sum_{i=1}^{k} |S_i| + \sum_{i=1}^{k-1} \sum_{j=1}^{|\xi_i|} |E_{i,j}|$.

Definition 5 *(Length: $|\hat{X}|$).* The *length* of $\hat{X}$, denoted by $|\hat{X}|$, is defined as the sum of the total length of its seeds and the total number of its elastic-degenerate symbols: $|\hat{X}| = \sum_{i=1}^{k} |S_i| + k - 1$.

Informally, the total number of positions in $\hat{X}$ is its length considering an elastic-degenerate symbol to occupy only one position. Intuitively, a position belonging to some seed will be called a *solid position* and that of an elastic-degenerate symbol will be called an *elastic-degenerate position*. In Example 2.1, the total length of the seeds is 9; hence, $\|\hat{X}\| = 9 + (2 + 3 + 4) + (6 + 3) = 27$, while $|\hat{X}| = 9 + 2 = 11$. The first a occurs at (solid) position 1, followed by b at (solid) position 2 and so on; ξ_1 and ξ_2 are at (elastic-degenerate positions) 5 and 9, respectively; the last b is at (solid) position 11.

Definition 6 *(Possibility-set: $\Re$).* For the elastic-degenerate string $\hat{X} = S_1\xi_1S_2\xi_2S_3\ldots S_{k-1}\xi_{k-1}S_k$, its possibility-set $\Re$ is defined as

$$\Re = \{S_1E_{1,r_1}S_2E_{2,r_2}\ldots E_{k-1,r_{k-1}}S_k\} \quad \forall r_i, 1 \le i \le k-1 \text{ such that } 1 \le r_i \le |\xi_i|.$$

ARTICLE IN PRESS

Informally, the *possibility-set* $\Re$ of $\hat{X}$ is the set of all possible solid strings obtained from $\hat{X}$. A solid string can be obtained by replacing each of the elastic-degenerate symbols with one of its constituent strings. In Example 2.1, $\Re =$ $\{\texttt{abbc}\underline{\texttt{ab}}\texttt{cca}\underline{\texttt{aabcab}}\texttt{bb}, \texttt{abbc}\underline{\texttt{ab}}\texttt{cca}\underline{\texttt{cba}}\texttt{bb}, \texttt{abbc}\underline{\texttt{aab}}\texttt{cca}\underline{\texttt{aabcab}}\texttt{bb}, \texttt{abb}\underline{\texttt{aab}}\texttt{cca}\underline{\texttt{cba}}\texttt{bb}, \texttt{abbc}\underline{\texttt{acca}}\texttt{cca}\underline{\texttt{aabcab}}\texttt{bb},$ $\texttt{abbc}\underline{\texttt{acca}}\texttt{cca}\underline{\texttt{cba}}\texttt{bb}\}$. Note that constituent strings replacing the elastic-degenerate symbols have been underlined for clarity.

Definition 7 *(Elasticity: Δ). Elasticity* (denoted as Δ) of $\hat{X}$ is the difference between the lengths of the shortest and the longest string in its possibility-set $\Re$.

As the shortest (or longest) string of $\Re$ is obtained by replacing each elastic-degenerate symbol with its shortest (or longest) constituent string,

$$\Delta = \sum_{i=1}^{k-1} |\xi_i|_{max} - \sum_{i=1}^{k-1} |\xi_i|_{min}$$

In Example 2.1, the total minimum length of the elastic-degenerate symbols is 5 (2+3); the total maximum length of the elastic-degenerate symbols is 10 (4+6). Hence, $\Delta = 10 - 5 = 5$.

Now we will introduce the notion of *elastic cardinality* which is based on the prefix/suffix nesting relationship amongst strings in an elastic-degenerate symbol.

Definition 8 *(Elastic cardinality: μ).* The *prefix (suffix) cardinality* μ_{pi} (μ_{si}) is defined as the largest number of the prefixes (suffixes) of a string in an elastic-degenerate symbol ξ_i that appear as the other constituent strings within the same ξ_i. The *elastic cardinality* of $\hat{X}$, denoted by μ, will be the maximum prefix or suffix cardinality amongst all its elastic-degenerate symbols (i.e. $\mu = \max(\max(\mu_{pi}), \max(\mu_{si}))$, $1 \le i \le k-1$).

Note that μ will at least be 1 since each string is a prefix (suffix) of itself. Also note that $\mu > 1$ for $\hat{X}$ may cause different combinations (i.e. replacement of elastic-degenerate symbols by one of its string elements) to result in the same solid string. To further clarify the concept, consider another example as follows:

Example 2.2. For $\hat{X} = \texttt{bc}\begin{bmatrix}\texttt{abc}\\\texttt{ab}\\\texttt{d}\\\texttt{de}\\\texttt{dee}\end{bmatrix}\texttt{ccc}\begin{bmatrix}\texttt{ab}\\\texttt{cba}\\\texttt{ba}\\\varepsilon\end{bmatrix}\texttt{ba}$, we have the following:

- For elastic-degenerate symbol ξ_1: $\mu_{p1} = 3$ because three prefixes of $E_{1,5}$ (dee) appear in ξ_1: $E_{1,5}$ itself, $E_{1,3}$ (d) and $E_{1,4}$ (de). Although $E_{1,1}$ (abc), apart from itself, also has another prefix as a constituent of ξ_1, i.e. $E_{1,2}$ (ab), but we have to take the largest number. As no string has any of its suffix present as a constituent, $\mu_{s1} = 1$.
- For elastic-degenerate symbol ξ_2: ξ_1: $\mu_{ps} = 3$ because two suffixes of $E_{2,2}$ (cba) appear as $E_{2,3}$ (ba) and $E_{2,4}$ (ε). Although $E_{2,4}(\varepsilon)$ is also a suffix of every string in ξ_2, but we have to take the largest number. As every string in ξ_2 has $E_{2,4}$ (ε) as its prefix, $\mu_{p2} = 2$.
- $\hat{X}$ has elastic cardinality $\mu = \max(3, 1, 3, 2) = 3$.
- Two combinations - $\texttt{bc}\underline{\texttt{ab}}\texttt{ccc}\underline{\texttt{cba}}\texttt{ba}$ and $\texttt{bc}\underline{\texttt{abc}}\texttt{ccc}\underline{\texttt{ba}}\texttt{ba}$ - yield the same solid string.

2.2. *Algorithmic tools*

In this section, we briefly introduce two fundamental data structures, which support a wide variety of string matching operations, and a well-known pattern matching algorithm. These tools will be used extensively by the proposed algorithm.

2.2.1. *Suffix tree*

The *suffix tree* $\mathbb{S}(X)$ of a non-empty string X of length n is a compact trie representing all the suffixes of X such that $\mathbb{S}(X)$ has n leaves labelled from 1 to n. The construction of the suffix tree $\mathbb{S}(X)$ for string X of length n over a fixed-sized alphabet takes $\mathcal{O}(n)$ time and space using one of the algorithms in [25–27]. Once the suffix tree of X has been constructed, it can be used to support queries that return all *Occ* occurrences of a given string (called pattern) of length m in time $\mathcal{O}(m + Occ)$. In addition, the *longest common ancestor* (LCA) of any two leaves of $\mathbb{S}(X)$, thus the length of the *longest common prefix* (LCP) of any two suffixes of X, can be computed in constant time after a linear-time pre-processing [28,29]. A *generalised suffix tree* is a suffix tree for a set of strings [30,31]. For a general introduction to suffix trees, see [32].

2.2.2. *KMP algorithm and failure function*

Knuth, Morris, and Pratt (KMP) introduced a linear-time algorithm for pattern matching in [33]; that is, an algorithm for finding all occurrences of a pattern P in a text T. It pre-processes P by computing a *failure function* f such that $f(i)$ is defined as the length of the longest prefix of P that is a suffix of $P[1..i]$. By using the failure function, it achieves an optimal search time of $\mathcal{O}(n)$ after $\mathcal{O}(m)$-time pre-processing, where n is the length of T and m is the length of P.

2.2.3. Aho-Corasick Automaton

The *Aho-Corasick automaton* [34] of a set of strings $\mathcal{P}$, denoted by $\mathcal{AC}(\mathcal{P})$, is the minimal partial deterministic finite automaton accepting the set of all strings having a string of $\mathcal{P}$ as a suffix (see [35, Section 7.1] for more description and for efficient construction). The construction of the suffix automaton $\mathcal{AC}(\mathcal{P})$ can be done in linear time and space [32] independent of the alphabet size. Once the automaton $\mathcal{AC}(\mathcal{P})$ has been constructed, searching a text T for occurrences of the patterns in $\mathcal{P}$ can be realised in time linear in the length of T for a fixed alphabet; such a problem is known as the dictionary matching problem. We can slightly modify the search algorithm so that it reports the head (rather than the tail) of an occurrence; this modified version has been used in the rest of the article.

3. Solid pattern and elastic-degenerate text

Before formalising the problem, we define *matching* and *occurrence* in the context of a solid pattern and an elastic-degenerate text.

Definition 9 *(Matching).* An elastic-degenerate string $\hat{X}$ with k seeds and a solid string Y are said to match, denoted by $\hat{X} \simeq Y$, if, and only if, there exists a solid string $S = S_1 E_{1,r_1} S_2 E_{2,r_2} \ldots E_{k-1,r_{k-1}} S_k$, $1 \le r_i \le |\xi_i|$, obtained from $\hat{X}$ (i.e. $S \in \Re$ of $\hat{X}$), such that $S = UYV$, where $U, V \in \Sigma^*$, satisfying:

$$\begin{cases} U = \varepsilon, V = \varepsilon & \text{if } S_1 \ne \varepsilon, S_k \ne \varepsilon \\ E_{1,r_1} \ne \varepsilon, V = \varepsilon, U \text{ is a prefix of } E_{1,r_1} & \text{if } S_1 = \varepsilon, S_k \ne \varepsilon \\ E_{k-1,r_{k-1}} \ne \varepsilon, U = \varepsilon, V \text{ is a suffix of } E_{k-1,r_{k-1}} & \text{if } S_1 \ne \varepsilon, S_k = \varepsilon \\ E_{1,r_1} \ne \varepsilon, U \text{ is a prefix of } E_{1,r_1}, & \\ \quad E_{k-1,r_{k-1}} \ne \varepsilon, V \text{ a suffix of } E_{k-1,r_{k-1}} & \text{if } S_1 = \varepsilon, S_k = \varepsilon. \end{cases}$$

Informally, we say that $\hat{X}$ and Y match such that Y starts at the first position of $\hat{X}$ if the position is solid or as a suffix of one of its non-empty strings if it is elastic-degenerate; and Y ends at the last position of $\hat{X}$ if the position is solid or as a prefix of one of its non-empty strings if it is elastic-degenerate.

Example 3.1. Consider $\hat{X}$ as given in Example 2.1. For string $Y =$ `abbcabccacbabb` we have that $\hat{X} \simeq Y$.

Definition 10 *(Occurrence).* In an elastic-degenerate string (text) $\hat{T}$, a solid string (pattern) P is said to have an occurrence starting and ending at positions i and j respectively, if $P \simeq \hat{T}[i..j]$. An occurrence is represented as the pair of starting position i (head) and ending position j (tail).

For consistency with the intuitive meaning of an occurrence, we say that P occurs at the position of some elastic-degenerate symbol (say ξ_i) of $\hat{T}$, if it is a factor of any of the constituent strings of ξ_i.

Example 3.2. Consider a pattern $P =$ `cabbcb` and a text $\hat{T}$ as follows:

$$\texttt{aacabbcbbc} \begin{bmatrix} \texttt{a} \\ \texttt{aab} \\ \texttt{acca} \end{bmatrix} \texttt{bb} \begin{bmatrix} \texttt{c} \\ \texttt{acabbcbb} \\ \texttt{cba} \end{bmatrix} \texttt{bacabbc} \begin{bmatrix} \texttt{b} \\ \texttt{cabb} \\ \texttt{bbc} \\ \texttt{aacabb} \end{bmatrix} \texttt{cbc}.$$

All the occurrences of P in $\hat{T}$ are given below.

Occurrence:	(3, 8)	(10,15)	(11,14)	(11,15)	(14,14)	(17,22)	(22,24)
Strings chosen:	-	ξ_1: $\underline{\texttt{a}}$ ξ_2: $\underline{\texttt{c}}$	ξ_1: $\texttt{ac}\underline{\texttt{ca}}$ ξ_2: $\underline{\texttt{cb}}\texttt{a}$	ξ_1: $\texttt{ac}\underline{\texttt{ca}}$ ξ_2: $\underline{\texttt{c}}$	ξ_2: $\texttt{a}\underline{\texttt{cabbcb}}\texttt{b}$	ξ_3: $\underline{\texttt{b}}$ or ξ_3: $\underline{\texttt{b}}\texttt{bc}$	ξ_3: $\underline{\texttt{cabb}}$ or ξ_3: $\texttt{aa}\underline{\texttt{cabb}}$

Note that more than one occurrence of P can start at the same starting position but their ending positions are different: for instance, $(11, 14)$ and $(11, 15)$ in Example 3.2. Also, note that different strings in the same elastic-degenerate symbols can lead to the same occurrence: for instance, the same pair of head and tail as happened for occurrences $(17, 22)$ and $(22, 24)$ in Example 3.2.

Example 3.3. Here we illustrate the case where an elastic-degenerate string has an empty seed (S_2). Consider a pattern $P =$ `babbcb` and a text $\hat{T}$ as follows:

$$\hat{T} = \texttt{ab}\begin{bmatrix}\texttt{bcab}\\\texttt{abb}\end{bmatrix}\begin{bmatrix}\texttt{ab}\\\texttt{cbb}\\\texttt{abc}\end{bmatrix}\texttt{cca}\begin{bmatrix}\texttt{bb}\\\texttt{cb}\end{bmatrix}\texttt{ca}.$$

There is an occurrence of P at $(2, 4)$ of $\hat{T}$.

3.1. Problem definition

Pattern Matching in Elastic-Degenerate Texts
Input: An elastic-degenerate text $\hat{T} = S_1\xi_1 S_2 \ldots \xi_{k-1}S_k$ of length n and total size N, a pattern P of length $m < N$.
Output: All the occurrences of P in $\hat{T}$.

By definition, all the occurrences of the pattern P in the text $\hat{T}$ fall under the following cases:

1. P entirely lies in some seed.
2. P entirely lies in some string of an elastic-degenerate symbol.
3. P spans across one or more elastic-degenerate symbols. This can further be seen as:
   (a) P starts in some seed.
   (b) P starts in some string of an elastic-degenerate symbol.

For instance, consider Example 3.2: the occurrences $(3, 8)$ and $(14, 14)$ fall into Case 1 and Case 2, respectively; $(10, 15)$ and $(17, 22)$ fall into Case 3(a); $(11, 14)$, $(11, 15)$, and $(22, 24)$ fall into Case 3(b).

3.2. Algorithm

Note that a naïve solution to this problem would be to find the pattern occurrences in the possibility-set $\Re$ of $\hat{T}$ using the KMP algorithm; this time is exponential in the number of elastic-degenerate symbols (as the number of strings in $\Re$ is $\mathcal{O}(q^{k-1})$ where q is the maximum number of constituent strings in any ξ). In this section, we present an efficient algorithm that makes use of the KMP algorithm and the suffix tree data structure. Clearly, the KMP algorithm can easily report the occurrences corresponding to the Cases 1 and 2. Case 3 requires some additional processing and data structures. The algorithm works in two stages, outlined below.

3.2.1. Stage 1: pre-processing

Pre-process the pattern P to compute its failure-function as required by the KMP algorithm. In addition, create the generalised suffix tree $\mathbb{S}_S$ for the set $\{P, S_1, S_2, \ldots, S_k\}$ of strings corresponding to all the seeds of $\hat{T}$, as well as the generalised suffix tree $\mathbb{S}_\xi$ for the set $\{P\} \cup \xi_1 \cup \xi_2 \cup \ldots \cup \xi_{k-1}$ of strings corresponding to all the strings in each of the elastic-degenerate symbols of $\hat{T}$. Furthermore, pre-process these two suffix trees so as to answer LCA queries in constant time.

3.2.2. Stage 2: search

Start searching the pattern P in the text $\hat{T}$ using the KMP algorithm, comparing the symbols and using the failure function to shift the pattern on a mismatch. The starting position of an occurrence being tested may be either solid or elastic-degenerate; we call the two types of occurrences as *Type* 1 and *Type* 2, respectively. We consider the two types separately as follows.

Type 1: Solid starting position Consider a situation when an occurrence starting from a position (say *pos*) that lies in some seed S_i is being tested. Proceed normally comparing the corresponding symbols of P and S_i; and shifting the pattern using failure function on a mismatch. As soon as the elastic-degenerate symbol ξ_i is encountered (suppose corresponding position in the pattern is p), suspend the KMP algorithm (for this test). Check each of the strings of ξ_i (i.e. $E_{i,j}$) whether or not it occurs in the pattern at position p, using LCA queries on $\mathbb{S}_\xi$, and *tick* (mark) the tails of the found occurrences. This can be realised by maintaining a boolean array of size m, which we denote by $\mathbb{T}_i$.

Next, Procedure 1 (given formally below) is executed. Each ticked position of $\mathbb{T}_i$ is tried to extend by testing whether S_{i+1} occurs adjacent to it (using LCA queries on $\mathbb{S}_S$). For each such found occurrence of S_{i+1}, occurrences of strings of ξ_{i+1} are checked using the suffix tree $\mathbb{S}_\xi$ and their tails are ticked in $\mathbb{T}_{i+1}$. The procedure will then be repeated for $\mathbb{T}_{i+1}$; this continues recursively until there is no tail marked in some call.

Once the process ends (reporting all the occurrences of P starting from *pos*, if any), the failure function corresponding to the position where the KMP algorithm was suspended (i.e. p) is used to shift the pattern and the KMP algorithm resumes. It is to be noted that an occurrence of P is implied if the length returned by the LCA query between the pattern starting from some ticked-tail t and either of the following hits the boundary of the pattern:

- some seed S_i;
- any string $E_{i,j}$ of some elastic-degenerate symbol ξ_i.

Procedure 1: Procedure to extend ticked tails in a given $\mathbb{T}_i$ and reporting the occurrences found, if any.

```
Extend($\mathbb{T}_i$)
    $isNonEmpty \leftarrow false$;
    forall indices t of $\mathbb{T}_i$ which are ticked do
        $l_s \leftarrow |$ LCA($P[t+1..m]$, $S_{i+1}[1..|S_{i+1}|]$) $|$;
        if $(l_s + t) = m$ then                                        // Pattern ends
            Report the occurrence;
        else if $l_s = |S_{i+1}|$ then                                  // $S_{i+1}$ occurs here
            $e \leftarrow t + |S_{i+1}|$;
            forall $E_{i+1,j}$ in $\xi_{i+1}$ do
                $l_e \leftarrow |$ LCA($P[e+1..m]$, $E_{i+1,j}[1..|E_{i+1,j}|]$) $|$;
                if $(l_e + e) = m$ then                                // Pattern ends
                    Report the occurrence (if not reported already);
                else if $l_e = |E_{i+1,j}|$ then                        // $E_{i+1,j}$ occurs here
                    Mark $e + |E_{i+1,j}| - 1$ in $\mathbb{T}_{i+1}$;
                    $isNonEmpty \leftarrow true$;
    if $isNonEmpty$ then
        Extend($\mathbb{T}_{i+1}$);
```

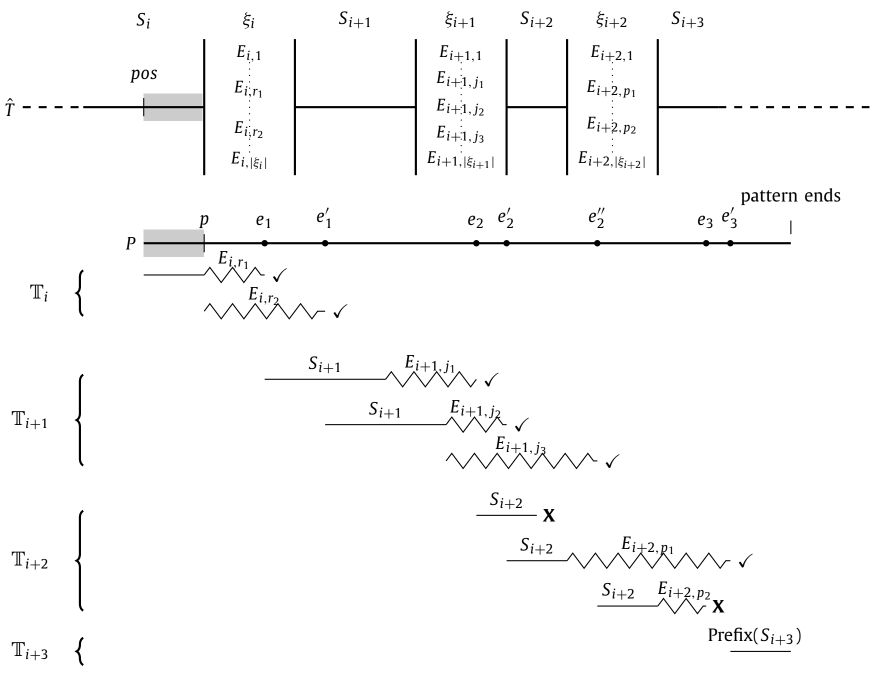

Fig. 1. An illustration of how the algorithm works for Type 1 occurrences. Strings in elastic-degenerate symbols are shown as zigzag, while solid lines depict the seeds. Symbol **X** denotes that this path could not be extended further while the symbol ✓ represents a ticked tail.

Fig. 1 elucidates the description given above.

Type 2: Elastic-Degenerate starting position Consider a situation when the starting position of an occurrence to be tested is an elastic-degenerate symbol ξ_i. This case can be processed in a similar fashion as the one described for Type 1, with the only difference being the manner in which tails are ticked initially.

Begin by applying the KMP algorithm for each $E_{i,j}$ to achieve two purposes: finding the occurrences of P in $E_{i,j}$ and ticking the last position of $E_{i,j}$ for which a prefix of P appears as a suffix of $E_{i,j}$. The ticked tails obtained in that way are then extended by Procedure 1 recursively and occurrences are reported. After the Procedure 1 ends, the KMP algorithm resumes and the testing starts at the beginning of the seed S_{i+1}.

3.3. Analysis

In the following, we discuss the correctness of the algorithm and analyse its space and time complexity.

3.3.1. Correctness

Consider an occurrence (i, j). If the occurrence falls under the Case 1 (resp. Case 2) then $j = i + m - 1$ (resp. $j = i$) for some fixed i. Thus, the number of occurrences falling under either Case 1 or Case 2 is bounded by $\mathcal{O}(n)$. On the other hand, for occurrences under Case 3, let parameter γ represent the maximum number of elastic-degenerate symbols spanned by any occurrence (i, j). Note that γ captures the possibility that the elastic-degenerate symbols contain empty strings. As there can be maximum m prefixes going past an elastic-degenerate position, the number of occurrences per starting position i are bounded by $\mathcal{O}(\gamma m)$. Thus the total number of distinct occurrences (i, j) is bounded by $\mathcal{O}(\gamma mn)$.

The correctness of the presented algorithm is straightforward as every starting position of the text is being tested for potential occurrences exhaustively. While the occurrences corresponding to the Cases 1 and 3(a) are covered by Type 1, Type 2 investigates all occurrences associated with Case 2 and Case 3(b). Thus all the occurrences of P in $\hat{T}$ are reported.

3.3.2. *Space complexity*

The space required by both the failure-function and the ticked tails array is $\mathcal{O}(m)$. The suffix tree $\mathbb{S}_S$ uses $\mathcal{O}(m+\sum_{i=1}^{k}|S_i|)$ space and the suffix tree $\mathbb{S}_\xi$ uses $\mathcal{O}(m+\sum_{i=1}^{k-1}\sum_{j=1}^{|\xi_i|}|E_{i,j}|)$ space. This leads to the total space required to be $\mathcal{O}(N)$, as $\sum_{i=1}^{k}|S_i| + \sum_{i=1}^{k-1}\sum_{j=1}^{|\xi_i|}|E_{i,j}| = N$ and $m < N$.

3.3.3. *Time complexity*

The time taken by the pre-processing stage is $\mathcal{O}(N)$ as the failure function can be computed in $\mathcal{O}(m)$ time and construction of both the suffix trees (along with their pre-processing required to answer LCA queries in constant time) can be done in $\mathcal{O}(N)$ time.

The search stage uses the KMP algorithm over each seed and each string of every elastic-degenerate symbol in the text to report the occurrences for Case 1 and Case 2; and to search the beginning of the occurrence for Case 3. Thus the time consumed by the KMP algorithm is $\mathcal{O}(\sum_{i=1}^{k}|S_i| + \sum_{i=1}^{k-1}\sum_{j=1}^{|\xi_i|}|E_{i,j}|) = \mathcal{O}(N)$.

Procedure 1 can be analysed as follows. Intuitively, for every ticked position in the pattern (which can at most be m), an LCA query is used to find whether the corresponding seed occurs at the ticked position or not; a found such occurrence is then tried to extend by another LCA query with each of the strings in the following elastic-degenerate symbol. Let parameter α represent the maximum number of strings in any elastic-degenerate symbol of the text. This extension step for each ticked position will be carried out at most α times. More specifically, the outer loop of the procedure runs m times and the inner one takes $\mathcal{O}(\alpha)$ time, as each LCA query takes constant time. Thus, each recursive call requires $\mathcal{O}(m\alpha)$ time. The number of recursive calls depends on the number of the elastic-degenerate symbols spanned by the occurrence of P being tested. In other words, if an occurrence spans across i elastic-degenerate symbols, there will be i recursive calls to the procedure. If γ is the maximum such i, Procedure 1 executes in $\mathcal{O}(\alpha\gamma m)$ time in total for each starting position.

Initial ticking of the tails in Type 1 needs $\mathcal{O}(\alpha)$ time. For Type 2, initial ticking is done by KMP algorithm (already accounted above). In the worst case, Procedure 1 will be called from each of the n starting positions of the text, leading to an overall time-complexity of the algorithm to be $\mathcal{O}(N + \alpha\gamma mn)$. In other words, the algorithm takes $\mathcal{O}(N + \alpha\gamma mn)$ time to find and report $\mathcal{O}(\gamma mn)$ number of possible occurrences of the pattern.

4. Elastic-degenerate pattern and solid text

We begin by defining *occurrence* in the context of an elastic-degenerate pattern and a solid text.

Definition 11 *(Occurrence).* An elastic-degenerate string $\hat{P}$ is said to occur in a solid string T at position i, if $T[i..j] \simeq \hat{X}$, where $i < j$; the occurrence is represented as a pair of its starting and ending positions (or head and tail).

An occurrence (i, j) simply means that the substring starting at position i in T and ending at position j is the same as one of the solid strings obtained from $\hat{P}$.

Example 4.1. Consider $\hat{X}$ as given in Example 2.1. In a given text

$$T = \texttt{ccababbcabccaaabcabbbaacacabbcaabccacbabbaacaaa},$$

there are two occurrences (overlined) of $\hat{X}$: $(5, 21)$ and $(27, 41)$

$$\texttt{ccab}\overline{\texttt{abbc}\underline{\texttt{ab}}\texttt{cca}\underline{\texttt{aabcab}}\texttt{b}}\texttt{baacac}\overline{\texttt{abbc}\underline{\texttt{aab}}\texttt{cca}\underline{\texttt{cba}}}\texttt{bbaacaaa}$$

Notice that more than one solid string obtained from $\hat{P}$ (if $\mu > 1$) can match the substring starting at a position but their ending positions will be different.

4.1. Problem definition

Finding Occurrences of an Elastic-Degenerate Pattern in a Solid Text
Input: A solid text T of length n; an elastic-degenerate pattern $\hat{P} = S_1 \xi_1 S_2 .. \xi_{k-1} S_k$—of total size m, elastic cardinality μ, and elasticity Δ—where each $\xi_i = \{E_{i,j}\}, \ 1 \le j \le |\xi_i|$.
Output: All pairs (s, e) such that there is an occurrence of $\hat{P}$ in T starting at position s and ending at position e.

A simple (brute-force) approach may use the Aho-Corasick Automaton of the strings in $\Re$ of $\hat{P}$ to find their occurrences in T. As the total number of all the strings in $\Re$ is $\mathcal{O}(\prod_{i=1}^{k-1} |\xi_i|)$, this approach will result in time complexity that is exponential in k. We present, in the following, a much more efficient solution.

4.2. Algorithm: basic idea

Every occurrence of $\hat{P}$ in T will start from an occurrence of S_1. Furthermore, finding an occurrence of $\hat{P}$ can be seen as iteratively incrementing its partial match seed by seed. The most straightforward approach, better than the naïve (brute-force) solution, is to independently find the occurrences of all the components (S_i and strings of ξ_i); followed by dynamic programming method to validate those occurrences so as to extend the partial matches starting from an occurrence of the first seed.

A significant efficiency can be achieved by using memoization if the problem is decomposed into sub-problems in the following way: Let $\Im_l$ be the set of ending positions of all the occurrences of the partial pattern up to seed S_l in the text. If $\mathbb{O}_{S_1}$ is the set of tails of all the occurrences of S_1 in the text, we begin with $\Im_1 = \mathbb{O}_{S_1}$ and use the following recursion to build the solution level by level such that $\Im_k$ contains all the distinct ending positions of the occurrences of the complete pattern:

$$\Im_l = \bigcup_{j \in \Im_{l-1}} extend(j, l-1)$$

where $extend(j, i)$ is a function which tries to extend the ending position j of seed S_i as follows: for each $E_{i,r}, 1 \le r \le |\xi_i|$ that starts next to j such that S_{i+1} occurs at the tail of this $E_{i,r}$, add the tail of this occurrence of S_{i+1} to the result. For reporting the starting positions corresponding to each ending position in $\Im_k$, back-pointers are required such that if j at level $l-1$ has been extended to i at level l then there is pointer from i to j.

This level-wise decomposition is conceptually the same as building a *layered graph*[1] (divided into levels) such that the vertices at each level i are subset of tails of occurrences of the seed S_i in the text and edges exist only between the vertices of adjacent levels (more specifically, from vertices of level i to vertices of level $i-1$). In the rest of the article, we will call a level l an *internal level* if $1 < l < k$ and an *external level* if $l = 1$ or $l = k$.

Our algorithm utilises the dynamic programming technique but explicitly creates the graph for organising and maintaining the search information. Moreover, we incorporate two simple ideas to speed up the graph construction and reporting stages as presented below.

Rarest seed as anchor: We note that every occurrence of $\hat{P}$ in T must have an occurrence of every seed. Instead of starting from level 1 (corresponding to the occurrences of the first seed), we can initiate building the graph from the level corresponding to the seed with the minimum number of occurrences in the text. We will call such a seed an *anchor*.

Adaptive graph traversal: For reporting, traversing the graph can be made adaptive to the number of distinct starting and ending positions of the pattern in the text as follows:

$$\text{Traversal direction} = \begin{cases} \text{bottom-up (i.e. from level } k \text{ to level 1)} & \text{if } |\Im_k| \le |\Im_1| \\ \text{top-down (i.e. from level 1 to level } k) & \text{otherwise.} \end{cases}$$

Forward as well as back chaining (pointers) strategy is required in order to attain this flexibility of the traversal. In addition, reporting can be done efficiently (as compared to a straightforward depth-first search) by utilising word-level parallelism if we maintain a bit-vector (at each vertex) of reachable nodes at level k (or level 1) for the bottom-up (or top-down) traversal. Fig. 2 illustrates the graph construction and simplification ideas.

It is worth pointing out that our algorithm bears superficial similarity to the implicit graph construction and traversal for reporting as used in [5] for gapped strings. Apart from the above-mentioned features to improve efficiency, there are other significant distinctions owing to the fundamental difference between a gapped and an elastic-degenerate pattern (which, in turn, gives different valid occurrences of a seed) and the definition of an occurrence itself (there can be many occurrences within the same starting and ending positions in [5], thus each enumerated path is an occurrence).

[1] A layered graph has the vertices partitioned into levels such that edges can exist only between the vertices of adjacent levels.

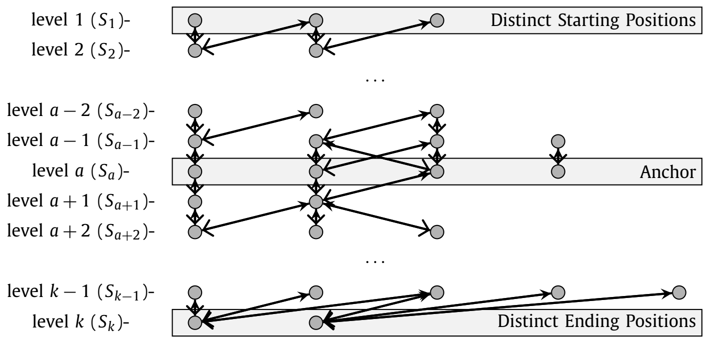

Fig. 2. An illustration of the graph constructed from the anchor (forward and backward edges have been differentiated using different arrowheads). Level a is chosen as an anchor because the seed S_a has minimum number of occurrences in the text. Graph traversal will be bottom-up because $|\Im_k| \leq |\Im_1|$.

4.3. Algorithm: details

The following steps describe our solution:

Stage 1: Pre-processing We build two Aho-Corasick Automata of the set $\mathcal{S}$ and set $E = \{E_{i,j}\}\ \forall i, j$; denoted by $\mathcal{AC}(\mathcal{S})$ and $\mathcal{AC}(E)$, respectively.

Stage 2: Computing occurrences of the components (i.e. all seeds and all the constituent strings of each elastic-degenerate symbol) Using the automaton $\mathcal{AC}(\mathcal{S})$, we compute the occurrences of the seeds in the text T and store them in a boolean table $\mathbf{Occ}(\mathcal{S})$ such that $\mathbf{Occ}(\mathcal{S})[i, j]$ is true if j is the head of the seed S_i. Let S_a be the seed with the minimum number of occurrences (to be used as the anchor).

Similarly, the occurrences of the strings making elastic-degenerate symbols (i.e. $E_{i,j}$) can be computed using automaton $\mathcal{AC}(E)$ and stored in a table, $\mathbf{Occ}(E)$, of k rows and n columns. Each cell $\mathbf{Occ}(E)[i, j]$ contains the list of lengths of the strings of ξ_i (i.e. $|E_{i,r}|, 1 \leq r \leq |\xi_i|$) that have heads (if $i \geq a$, tails otherwise) at position j in the text. In other words, to enable building the graph from the anchor a, the elastic-degenerate symbols before the anchor seed will record their occurrences at the corresponding ending positions while those after the anchor will mark them at the starting positions.

Stage 3: Graph construction The graph is constructed level by level starting from the occurrences of the anchor seed. A vertex at level l represents a tail (ending position in the text) of an occurrence of the seed S_l; it contains *forward edges* (pointers) to its children at level $l+1, \forall 1 < l \leq k$ and *backward edges* (pointers) to its parent at level $l-1, \forall 1 \leq l < k$. Each level is maintained as an associative array of the vertices with corresponding text-index as the key. Edges are maintained as a list of pointers within a vertex.

We follow Procedure 2 to build the graph iteratively, starting from the anchor-level, extending in both, forward and backward directions; making use of the tables $\mathbf{Occ}(\mathcal{S})$ and $\mathbf{Occ}(E)$. Clearly, when the procedure ends, $\Im_k$ and $\Im_1$ will respectively contain all the distinct ending positions and starting positions (if any) of $\hat{P}$ in the text.

Stage 4: Reporting Note that if we were to report only the distinct starting and ending positions of all the occurrences (rather than specifically reporting each pair corresponding to an occurrence), we can stop at the previous stage. For reporting each occurrence, we need to traverse the graph since each vertex (say $v_{1,i}$) at level 1 reachable from a vertex at level k (say $v_{k,j}$) makes an occurrence—($v_{1,i}.key, v_{k,j}.key$)—and vice-versa. Either bottom-up (if $|\Im_1| > |\Im_k|$) or top-down (if $|\Im_1| \leq |\Im_k|$) traversal of the simplified graph is executed. A bit-vector of length $\mathbb{Z}$ is maintained at each vertex where $\mathbb{Z} = \min(|\Im_1|, |\Im_k|)$; each bit represents reachability to a vertex at the starting level (i.e. level k for bottom-up and 1 for top-down) of the traversal. Initially the bit-vectors at each vertex is set to all zeroes at all levels except the starting level where the bit of the vertex itself is set to 1. In the bottom-up (or top-down) traversal, taking union of bit-vectors of the children (or parents) updates the corresponding bit-vectors at the previous (or next) level; proceeding level wise in this fashion eventually gives the ending positions (or starting positions) corresponding to each starting position (or ending position) at level 1 (or k).

4.4. Analysis

Let $\mathbb{O}_{comp}$, $\mathbb{O}_S$, and $\mathbb{O}_{S_a}$ be the number of occurrences of the components, all the seeds, and the anchor (rarest seed) of the elastic-degenerate pattern $\hat{P}$, respectively. The following lemmas establish the size of the graph which is conducive to the analysis of space and time complexity of the algorithm.

Lemma 4.2. *The number of vertices of the graph is* $\mathcal{O}(\min(\mathbb{O}_S, k\Delta\mathbb{O}_{S_a}))$.

Procedure 2: Construct the graph starting from the anchor-level a, extending in both, forward and backward directions; making use of the tables $\mathbf{Occ}(\mathcal{S})$ and $\mathbf{Occ}(E)$.

```
Construct ($a$, $\mathbf{Occ}(\mathcal{S})$, $\mathbf{Occ}(E)$)
    Add the tails of all occurrences of $S_a$ to $\mathfrak{I}_a$;
    /* Construct the graph to level $k$ : Forward                    */
    for $l \leftarrow a$ to $k-1$ do
        forall $v$ in $\mathfrak{I}_l$ do
            $i \leftarrow v.key$
            forall $j$ in $\mathbf{Occ}(E)[l, i+1]$ do
                if $\mathbf{Occ}(\mathcal{S})[l, i+j+1]$ is $True$ then
                    $e \leftarrow i+j+|S_{l+1}|$
                    Add vertex corresponding to $e$ to $\mathfrak{I}_{l+1}$, if not already there
                    Add backward edge from $e$ to $v$
                    Add forward edge from $v$ to $e$
    /* Construct the graph to level 1 : Backward                     */
    for $l \leftarrow a$ to 2 do
        forall $v$ in $\mathfrak{I}_l$ do
            $i \leftarrow v.key - |S_a|$
            forall $j$ in $\mathbf{Occ}(E)[l-1, i]$ do
                $e \leftarrow i-j$
                if $\mathbf{Occ}(\mathcal{S})[l-1, e-|S_{l-1}|+1]$ is $True$ then
                    Add vertex corresponding to $e$ to $\mathfrak{I}_{l-1}$, if not already there
                    Add forward edge from $e$ to $v$
                    Add backward edge from $v$ to $e$
```

Proof. Let δ_i be the difference in the lengths of the shortest and the longest string within an elastic-degenerate symbol ξ_i and Δ_i be the difference in the lengths of the shortest and the longest partial string up to seed s_{i+1}. $\Delta_i = \sum_{j=1}^{i-1} \delta_j$. It implies $\Delta = \Delta_k > \Delta_i \; \forall 1 \le i < k$. Each partial string up to seed i will have Δ ending positions (with a fixed starting position) in the worst case. Analogously, for a fixed anchor seed, there are Δ starting positions and Δ ending positions extending to its left and right, respectively. Thus, at each level the maximum number of vertices are $\Delta \mathbb{O}_{S_a}$, but a vertex is only added if there is a corresponding occurrence of the seed. Hence, the total number of vertices of the graph is bounded by the minimum of the two: $\mathbb{O}_S$ and $k\Delta\mathbb{O}_{S_a}$. □

Lemma 4.3. *The number of edges of the graph is $\mathcal{O}(\mu\mathbb{V})$ where $\mathbb{V}$ is the number of the vertices in the graph.*

Proof. Each cell of $\mathbf{Occ}(E)$ contains the list of the lengths of string in the corresponding elastic-degenerate symbol occurring at the corresponding position; the size of such a list is upper-bounded by μ. For each vertex at level (say l), an edge is added for each length in the list contained in the corresponding cell of $\mathbf{Occ}(E)$ if followed by an occurrence of the seed S_{l+1}. As a consequence, for $\mathbb{V}$ vertices in the graph, the number of edges are $\mathcal{O}(\mu\mathbb{V})$. □

4.4.1. Space complexity

The total space consumed by the two automata is $\mathcal{O}(m)$. $\mathcal{O}(kn)$ space is required by the boolean matrix $\mathbf{Occ}(\mathcal{S})$. A cell in the table $\mathbf{Occ}(E)$ requires $\mathcal{O}(\mu)$ space, leading to the total space taken by the table to be $\mathcal{O}(kn\mu)$. The vertices at each level are maintained as an associative map; if direct addressing model is used, $\mathcal{O}(kn)$ space is needed for the vertices of the graph. Each vertex stores $\mathcal{O}(\mu)$ pointers representing edges, thus the graph takes $\mathcal{O}(kn\mu)$ space. Consequently, the total space used by the algorithm is $\mathcal{O}(m + k\mu n)$.

4.4.2. Time complexity

Constructing both $\mathcal{AC}(\mathcal{S})$ and $\mathcal{AC}(E)$ takes $\mathcal{O}(m)$ time. $\mathcal{O}(n + \mathbb{O}_{comp})$ time is required for finding all the occurrences of the components.

Adding an edge (given two vertices) takes constant time. Furthermore, checking the presence of a vertex and adding a new vertex can be done in constant time, assuming a direct addressing associative map model. Consequently, the graph construction procedure of Stage 2 runs in $\mathcal{O}(\mu\mathbb{V})$ time where $\mathbb{V}$ is the number of vertices in the graph. Note that the number $\mathbb{V}$ of the vertices of the graph is less than or equal to the number $\mu\mathbb{V}$ of the edges by Lemma 4.3.

Graph simplification in the subsequent stage requires $\mathcal{O}(\mu\mathbb{V})$ time; followed by $\mathcal{O}(\mu\mathbb{V}\mathbb{Z}/w)$ time for reporting the occurrences, where w is the size of the computer word and $\mathbb{Z}$ is the number of distinct starting or ending positions, whichever is minimum.

Overall, $\mathcal{O}(n + m + \mathbb{O}_{comp} + \mu\mathbb{V}\mathbb{Z}/w)$ time is taken where $\mathbb{V} = \min(\mathbb{O}_S, k\Delta\mathbb{O}_{S_a})$ by Lemma 4.2. Although the worst-case time complexity is certainly better than that of the naïve approach, it can still be $\mathcal{O}(m + k\mu n^2/w)$ (as $\mathbb{O}_{comp}$ can be μkn; $\mathbb{Z}$ can be n; and $\mathbb{V}$ can be kn). However, real data is mostly such that the $\mathbb{Z} \lll n$ and $\mathbb{V} \lll n$ (because the number of occurrences of the pattern and seeds are not that frequent). Therefore, the algorithm is expected to be fast in practice.

5. Conclusion

Motivated by applications in bioinformatics and image-searching, we extended the notion of gapped strings to elastic-degenerate strings. In particular, we presented efficient algorithms for two variants of the pattern matching problem in the context of elastic-degenerate strings: first, for a solid pattern and an elastic-degenerate text; second, for an elastic-degenerate pattern and a solid text.

The presented algorithm for the first variant runs in $\mathcal{O}(N + \alpha\gamma mn)$ time; where m is the length of the given pattern; n and N are the length and total size of the given elastic-degenerate text, respectively; α and γ are parameters, respectively representing the maximum number of strings in any elastic-degenerate symbol of the text and the maximum number of elastic-degenerate symbols spanned by any occurrence of the pattern in the text. Note that in applications involving gene sequence variation data, α represents the number of sequences in the multiple sequence alignment of the similar sequences and γ represents the number of genetic variation-sites falling in a full occurrence. The values of these parameters can be small in practice, and so the presented algorithm is expected to work very fast in practice. The implementation of this algorithm is available at https://github.com/Ritu-Kundu/ElDeS.

The algorithm for the second variant takes $\mathcal{O}(m + k\mu n^2/w)$ time, where n is the length of the given text, m is the total size of the given elastic-degenerate pattern, k is the number of seeds, μ is the elastic cardinality of the pattern, and w is the size of the computer word.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] 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, Proceedings, LATA 2017, Umeå, Sweden, March 6-9, 2017, in: Lecture Notes in Computer Science, vol. 10168, 2017, pp. 131–142.

[2] M. Fischer, M. Paterson, String-Matching and Other Products, MAC Technical Memorandum, Mass. Inst. of Technology, Project MAC, 1974, https://books.google.co.uk/books?id=aSa4HAAACAAJ.

[3] F. Blanchet-Sadri, Algorithmic combinatorics on partial words, Int. J. Found. Comput. Sci. 23 (06) (2012) 1189–1206, https://doi.org/10.1142/S0129054112400473, arXiv: https://www.worldscientific.com/doi/pdf/10.1142/S0129054112400473, https://www.worldscientific.com/doi/abs/10.1142/S0129054112400473.

[4] M. Crochemore, M.-F. Sagot, Motifs in sequences: localization and extraction, in: Compact Handbook of Computational Biology, Marcel Dekker, New York, 2004, pp. 47–97.

[5] M.S. Rahman, C.S. Iliopoulos, I. Lee, M. Mohamed, W.F. Smyth, Finding patterns with variable length gaps or don't cares, in: Computing and Combinatorics: 12th Annual International Conference, Proceedings, COCOON 2006, Taipei, Taiwan, August 15-18, 2006, vol. 4112, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 146–155.

[6] Y. Li, J. Bailey, L. Kulik, J. Pei, Efficient matching of substrings in uncertain sequences, in: M.J. Zaki, Z. Obradovic, P. Tan, A. Banerjee, C. Kamath, S. Parthasarathy (Eds.), Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014, SIAM, 2014, pp. 767–775.

[7] Y. Liu, M. Koyutürk, S. Maxwell, M. Xiang, M. Veigl, R.S. Cooper, B.O. Tayo, L. Li, T. LaFramboise, Z. Wang, X. Zhu, M.R. Chance, Discovery of common sequences absent in the human reference genome using pooled samples from next generation sequencing, BMC Genomics 15 (1) (2014) 685, https://doi.org/10.1186/1471-2164-15-685.

[8] L. Huang, V. Popic, S. Batzoglou, Short read alignment with populations of genomes, Bioinformatics 29 (13) (2013) i361–i370, https://doi.org/10.1093/bioinformatics/btt215, arXiv: http://bioinformatics.oxfordjournals.org/content/29/13/i361.full.pdf+html, http://bioinformatics.oxfordjournals.org/content/29/13/i361.abstract.

[9] D.M. Church, V.A. Schneider, K.M. Steinberg, M.C. Schatz, A.R. Quinlan, C.-S. Chin, P.A. Kitts, B. Aken, G.T. Marth, M.M. Hoffman, J. Herrero, M.L.Z. Mendoza, R. Durbin, P. Flicek, Extending reference assembly models, Genome Biol. 16 (1) (2015) 13, https://doi.org/10.1186/s13059-015-0587-3.

[10] A. Dilthey, C. Cox, Z. Iqbal, M.R. Nelson, G. McVean, Improved genome inference in the MHC using a population reference graph, Nat. Genet. 47 (6) (2015) 682–688, https://doi.org/10.1038/ng.3257, technical report.

[11] 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, Proceedings, WABI 2016, Aarhus, Denmark, August 22-24, 2016, in: Lecture Notes in Computer Science, vol. 9838, Springer, 2016, pp. 222–233.

[12] J. Holub, W. Smyth, S. Wang, Fast pattern-matching on indeterminate strings, J. Discret. Algorithms 6 (1) (2008) 37–50, https://doi.org/10.1016/j.jda.2006.10.003, selected papers from AWOCA 2005, http://www.sciencedirect.com/science/article/pii/S1570866706000967.

[13] C.S. Iliopoulos, L. Mouchard, M.S. Rahman, A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching, Math. Comput. Sci. 1 (4) (2008) 557–569, https://doi.org/10.1007/s11786-007-0029-z.

[14] W.F. Smyth, S. Wang, An adaptive hybrid pattern-matching algorithm on indeterminate strings, Int. J. Found. Comput. Sci. 20 (06) (2009) 985–1004, https://doi.org/10.1142/S0129054109007005, arXiv: https://www.worldscientific.com/doi/pdf/10.1142/S0129054109007005, https://www.worldscientific.com/doi/abs/10.1142/S0129054109007005.

[15] M. Crochemore, C.S. Iliopoulos, R. Kundu, M. Mohamed, F. Vayani, Linear algorithm for conservative degenerate pattern matching, Eng. Appl. Artif. Intell. 51 (2016) 109–114, https://doi.org/10.1016/j.engappai.2016.01.009, http://www.sciencedirect.com/science/article/pii/S0952197616000130.

[16] J. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, E. Prieur-Gaston, B. Watson, Efficient pattern matching in degenerate strings with the burrows–wheeler transform, Inf. Process. Lett. 147 (2019) 82–87, https://doi.org/10.1016/j.ipl.2019.03.003, http://www.sciencedirect.com/science/article/pii/S0020019019300535.

[17] S.P. Pissis, MoTeX-II: structured MoTif eXtraction from large-scale datasets, BMC Bioinform. 15 (1) (2014) 235, https://doi.org/10.1186/1471-2105-15-235.

[18] 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, in: LIPIcs, vol. 78, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 9.

[19] K. Aoyama, Y. Nakashima, I. Tomohiro, 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, in: LIPIcs, vol. 105, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 9.

[20] 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, L'Aquila, Italy, June 27–29, 2018, in: LIPIcs, vol. 103, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 16.

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

[22] G. Bernardini, N. Pisanti, S.P. Pissis, G. Rosone, Approximate pattern matching on elastic-degenerate text, Theor. Comput. Sci. (2020), https://doi.org/10.1016/j.tcs.2019.08.012, http://www.sciencedirect.com/science/article/pii/S0304397519305018.

[23] 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, July 9-12, 2019, Patras, Greece, in: LIPIcs, vol. 132, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 21.

[24] R. Kundu, Algorithmic Advances in Handling Uncertainty and Regularity in Strings, Ph.D. thesis, King's College London, 2019.

[25] P. Weiner, Linear pattern matching algorithms, in: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory, Institute of Electrical Electronics Engineer, 1973, pp. 1–11.

[26] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.

[27] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[28] H.T. Harel, R.E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355.

[29] B. Schieber, U. Vishkin, On finding lowest common ancestors: simplification and parallelization, SIAM J. Comput. 17 (6) (1988) 1253–1262, https://doi.org/10.1137/0217079.

[30] A. Amir, M. Farach, Z. Galil, R. Giancarlo, K. Park, Dynamic dictionary matching, J. Comput. Syst. Sci. 49 (2) (1994) 208–222, https://doi.org/10.1016/S0022-0000(05)80047-9, http://www.sciencedirect.com/science/article/pii/S0022000005800479.

[31] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, NY, USA, 1997.

[32] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007, 392 pages.

[33] D.E. Knuth, J. James H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350, https://doi.org/10.1137/0206024.

[34] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340, https://doi.org/10.1145/360825.360855.

[35] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.