Faster Online Elastic Degenerate String Matching

Kotaro Aoyama
Department of Electrical Engineering and Computer Science, Kyushu University, Japan
kotaro.aoyama@inf.kyushu-u.ac.jp

Yuto Nakashima
Department of Informatics, Kyushu University, Japan
yuto.nakashima@inf.kyushu-u.ac.jp

Tomohiro I
Frontier Research Academy for Young Researchers, Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp
https://orcid.org/0000-0001-9106-6192

Shunsuke Inenaga
Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp

Hideo Bannai
Department of Informatics, Kyushu University, Japan
bannai@inf.kyushu-u.ac.jp
https://orcid.org/0000-0002-6856-5185

Masayuki Takeda
Department of Informatics, Kyushu University, Japan
takeda@inf.kyushu-u.ac.jp

Abstract

An *Elastic-Degenerate String* [Iliopoulus et al., LATA 2017] is a sequence of sets of strings, which was recently proposed as a way to model a set of similar sequences. We give an online algorithm for the Elastic-Degenerate String Matching (EDSM) problem that runs in $O(nm\sqrt{m}\log m + N)$ time and $O(m)$ working space, where n is the number of elastic degenerate segments of the text, N is the total length of all strings in the text, and m is the length of the pattern. This improves the previous algorithm by Grossi et al. [CPM 2017] that runs in $O(nm^2 + N)$ time.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases elastic degenerate pattern matching, boolean convolution

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.9

Acknowledgements This work was supported by JSPS KAKENHI Grant Numbers JP17H06923 (YN), JP17H01697 (SI), JP16H02783 (HB), and JP25240003 (MT).

1 Introduction

The *degenerate* string matching problem [7, 1] is a variant of the string matching problem when a position in the text may contain uncertainties; the text string, called a degenerate string, can be regarded as a string over an extended alphabet that consists of non-empty subsets of the original alphabet. A string matches a degenerate string if the subset at each position of the degenerate string contains the character of the pattern at the corresponding position. The *Elastic Degenerate String Matching* (EDSM) problem, first proposed by

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).
Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 9; pp. 9:1–9:10
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Iliopoulos et al. [9], is a further generalization of this setting, where substrings of the text may contain uncertainties; the text string, called an elastic degenerate string (ED string), can be regarded as a sequence of non-empty sets of strings. A string matches an ED string if it is a substring of a string that can be obtained by taking a string from each position of the ED string, and concatenating them. The motivation behind these problems is in bioinformatics, where multiple genomic sequences from individuals of the same species can be obtained. Recently, rather than than considering a single reference sequence, it is increasingly more common to consider the multiple sequences [4], and ED strings is one way to model them.

Iliopoulos et al. [9], gave an (offline) algorithm which runs in $O(N + \alpha\gamma nm)$ time, where m is the length of the given pattern, n and N are respectively the length and total size of the given elastic-degenerate text, α and γ respectively represent the maximum number of strings in any elastic degenerate symbols and the largest number of elastic-degenerate symbols spanned by any occurrence of the pattern in the text.

The online version of the problem was considered by Grossi et al. [8], where they gave an algorithm which runs in $O(nm^2 + N)$ time. They also give an algorithm which runs in $O(N\lceil\frac{m}{w}\rceil)$ time, where w is the computer word size. Furthermore, Bernardini et al. [3] consider the EDSM problem with errors, and presented an on-line algorithm that runs in $O((k+1)^2 mG + (k+1)N)$ time and $O(m)$ space, where k is the number of allowed errors (insertion/deletion/substitution), and $n \leq G \leq N$ is the total size of the sets of subsets (i.e., the total number of strings) in the ED string. They also present a faster $O((k+1)(mG+N))$ time and $O(m)$ space algorithm when considering only substitution errors.

In this paper, we improve the first algorithm by Grossi et al., and give a faster online algorithm for EDSM that runs in $O(nm\sqrt{m\log m} + N)$ time and $O(m)$ working space, assuming that the alphabet size is constant, as in previous work. For the space complexity, we will also assume that the strings at each position of the ED string are given in lexicographically sorted order. We note that the algorithm can be considered better than that of Bernardini et al. (with $k = 0$), when each subset in the ED string may contain many strings, which could be the case as more sequences from many individuals become increasingly available.

2 Preliminaries

2.1 Strings

For any set Σ, an element of Σ^* is called a string over alphabet Σ. We will assume that $|\Sigma|$ is constant. The empty string is denoted by ε. For any string $w \in \Sigma^*$, if $w = xyz$ for (possibly empty) strings x, y, z, then, x, y, and z are respectively called a prefix, substring, suffix of w. The length of w is denoted by $|w|$. Let $\mathit{Pref}(w), \mathit{Sub}(w), \mathit{Suf}(w)$ respectively denote the set of prefixes, substrings, and suffixes of w. For any integers $1 \leq i \leq j \leq |w|$, $w[i]$ denotes the ith symbol of w, i.e., $w = w[1] \cdots w[|w|]$, and $w[i..j] = w[i] \cdots w[j]$ denotes a substring of w that starts at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$, $w[i..j] = w[1..j]$ when $i < 1$, and $w[i..j] = w[i..|w|]$ when $j > |w|$.

If $w = xu = vx$ for non-empty strings u, v and possibly empty x, then x is called a *border* of w. The *border array* of w is an array $B[1..|w|]$ of integers such that $B[i]$ stores the length of the longest border of $w[1..i]$. It is well known that the border array of w can be computed in linear time in an on-line fashion. Also, given the border array of w, the length of all borders of w can be computed in linear time.

2.2 Elastic-Degenerate Strings

Let $\tilde{\Sigma}$ denote the set of all finite non-empty subsets of Σ^* excluding $\{\varepsilon\}$. An Elastic-Degenerate string, or ED string, over alphabet Σ, is a string over $\tilde{\Sigma}$, i.e., an ED string is an element of $\tilde{\Sigma}^*$. Below is an example of an ED string over $\Sigma = \{A, C, T\}$.

▶ **Example 1** (Elastic-Degenerate String).

$$\tilde{T} = \left\{ \begin{array}{c} A, \\ C \end{array} \right\} \cdot \left\{ \begin{array}{c} C, \\ CA, \\ TACA \end{array} \right\} \cdot \left\{ \begin{array}{c} \varepsilon, \\ AC, \\ C \end{array} \right\} \cdot \left\{ \begin{array}{c} AT, \\ C \end{array} \right\}$$

Let $\tilde{T}$ denote an ED string of length n, i.e. $|\tilde{T}| = n$. We assume that for any $1 \leq i \leq n$, the set $\tilde{T}[i]$ is implemented as an array and can be accessed by an index, i.e., $\tilde{T}[i] = \{\tilde{T}[i][k] \mid k = 1, \ldots, |\tilde{T}[i]|\}$. We will also make the assumption that the strings in $\tilde{T}[i]$, i.e., $\tilde{T}[i][1], \ldots, \tilde{T}[i][|\tilde{T}[i]|]$, are sorted in lexicographic order. For any $\tilde{c} \in \tilde{\Sigma}$, $\|\tilde{c}\|$ denotes the total length of all strings in $\tilde{c}$, and for any ED string $\tilde{T}$, $\|\tilde{T}\|$ denotes the total length of all strings in all $\tilde{T}[i]$ for $i = 1, \ldots, |\tilde{T}|$, i.e., $\|\tilde{c}\| = \sum_{s \in \tilde{c}} |s|$ and $\|\tilde{T}\| = \sum_{i=1}^{n} \|\tilde{T}[i]\|$. We note that Grossi et al. define $|\varepsilon| = 1$ when computing $\|\tilde{T}\|$, but this only increases the value of $\|\tilde{T}\|$ by at most n (which is less than $\|\tilde{T}\|$, in our definition, since $\|\tilde{c}\| \geq 1$ for any $\tilde{c} \in \tilde{\Sigma}$) and thus does not affect the asymptotic complexities.

An ED string $\tilde{T}$ can be thought of as a representation of the set of strings $\mathcal{A}(\tilde{T}) = \tilde{T}[1] \times \cdots \times \tilde{T}[n]$, where $A \times B = \{xy \mid x \in A, y \in B\}$ for any sets of strings A and B. The string in Example 1 can be viewed as a representation of a of $2 \times 3 \times 3 \times 2 = 36$ strings.

For any ED string $\tilde{T}$ and string P, we say that P *matches* $\tilde{T}$ if

1. $|\tilde{T}| = 1$ and P is a substring of some $s \in \tilde{T}[1]$, or,
2. $|\tilde{T}| > 1$ and $P = p_1 \cdots p_{|\tilde{T}|}$ where p_1 is a suffix of some string in $\tilde{T}[1]$, $p_{|\tilde{T}|}$ is a prefix of some string in $\tilde{T}[|\tilde{T}|]$, and $p_i \in \tilde{T}[i]$ for all $1 < i < |\tilde{T}|$.

We say that an occurrence of P in $\tilde{T}$ starts at position i and ends at position j, if P matches $\tilde{T}[i..j]$. Below is the problem we solve.

▶ **Problem 2** (Elastic-Degenerate String Matching (EDSM) [8]). *Given a string P of length m, and an ED string $\tilde{T}$ of length n and size $N \geq m$, output all positions j in $\tilde{T}$ where at least one occurrence of P ends.*

We will measure space complexity in terms of working space, and exclude the input pattern and ED string, which we assume to be stored in read-only memory, as well as the space for output, which is write-only.

3 Tools

3.1 Suffix Trees

A *suffix tree* [11] $ST(w)$ of a string w is a compacted trie of all suffixes of $w\$$, where \$ is a special symbol that does not occur in w. In other words, the suffix tree of w is a rooted tree where each edge has string labels, where all and only suffixes of $w\$$ are represented in the concatenation of all labels on a root to leaf path. Furthermore, each internal node has at least two outgoing edges where the first character of the label of the outgoing edges are distinct. We assume that each leaf is labeled by an integer that represents the starting position of the suffix that corresponds to the root to leaf path. Although the total length of all labels in a suffix tree is not $O(|w|)$, each label can be represented in constant space, i.e., two integers representing positions in w, since any edge label is a substring of w. It is well known that the suffix tree of w can be constructed in $O(|w|)$ time for constant size alphabets [11] (as well as integer alphabets [6]).

For any node u in the suffix tree, let $\mathit{str}(u)$ denote the concatenation of all edge labels on the root to u path, and let $\mathit{len}(u) = |\mathit{str}(u)|$. Let $\mathit{parent}(u)$ denote the parent of u, and $\mathit{anc}(u)$, $\mathit{desc}(u)$ respectively, the set of nodes in the suffix tree that are ancestors and descendants of u, including u itself. A position in the suffix tree can be represented a a pair (u, d), where u is a node and $d \geq 0$ is an integer such that $d \leq |\mathit{str}(u)|$ and $d > |\mathit{str}(\mathit{parent}(u))|$ (if $\mathit{parent}(u)$ exists). For any substring s of w, the *locus* of s in $ST(w)$ is a position (u, d) in $ST(w)$ where $s = \mathit{str}(u)[1..d]$. For any string s, the locus of the longest prefix s' of s that is a substring of w can be obtained in $O(|s'|)$ time by simply traversing the suffix tree from the root.

We denote by $L(u)$, the set of integers that are labels on the leaf nodes that are descendants of u. Let $\mathit{Occ}(w, s)$ denote the set of occurrences of s in w, i.e. $\mathit{Occ}(w, s) = \{i \mid w[i..i + |s| - 1] = s\}$. Suffix trees can be used to compute this set efficiently, since $\mathit{Occ}(w, s) = L(v_s)$, where $(v_s, |s|)$ is the locus of s in $ST(w)$, if it exists, and $\mathit{Occ}(w, s) = \emptyset$ otherwise.

▶ **Lemma 3** ([11]). *Given the suffix tree $ST(w)$ of string w, $\mathit{Occ}(w, s)$ for any string s can be computed in $O(|s| + \mathit{Occ}(w, s))$ time.*

3.2 Sum Set and FFT

For any sets of integers A, B, we denote by $A \oplus B = \{a + b \mid a \in A, b \in B\}$ the sum set of A and B.

▶ **Lemma 4** (Efficient Computation of SumSet). *For any integer m and sets of integers $A, B \subseteq [1..m]$, $A \oplus B$ can be computed in $O(m \log m)$ time and $O(m)$ space.*

Proof. For any set $X \subseteq [1..m]$, let $I(X)$ denote an array of Boolean (`true` or `false`) values where $I(X)[i] = \mathtt{true}$ if and only if $i \in X$. For any $1 \leq i \leq m$, $I(A \oplus B)$ can be computed by the Boolean convolution:

$$I(A \oplus B)[i] = \bigvee_{j=1}^{m} (I(A)[i - j] \wedge I(B)[j]).$$

It is well known that these values can be done in total $O(m \log m)$ time and $O(m)$ space for all $1 \leq i \leq m$, using the Fast Fourier Transform [5]. The set $A \oplus B$ can easily be obtained in $O(m)$ time by scanning $I(A \oplus B)[1..m]$. ◀

4 Algorithm

We first give an overview of the algorithm of Grossi et al. [8] which our algorithm is based on, and then describe our improvements to it.

4.1 Overview of Algorithm

The algorithm is on-line, i.e., for each ED string position $i = 1, \ldots, n$, it outputs i as an answer if there is an occurrence of P that ends at i, i.e., P matches $\tilde{T}[l..i]$ for some $l \leq i$. Although the total number of strings in $\mathcal{A}(\tilde{T}[1..i])$, and thus the total number of occurrences of P in all these strings, can be exponential in i, the problem can be solved efficiently since we only consider whether there is an end of an occurrence of P in position i of the ED string. To this end, the key of the algorithm is to compute for each i, the set

$$\mathcal{S}_i^{\leq} = \{j \mid 1 < j \leq m, \exists s \in \mathcal{A}(\tilde{T}[1..i]) \text{ s.t. } P[1..j-1] \text{ is a suffix of } s\}$$

which represents the positions j in P such that $P[1..j-1]$ occurs as a suffix of some string in $\mathcal{A}(\tilde{T}[1..i])$. In other words, the set corresponds to potential positions j in P that can result in a match later, if $P[j..m]$ is a prefix of some string in $\mathcal{A}(\tilde{T}[i+1..n])$. For each i, the computation performs the following steps:

1. Determine whether P matches $\tilde{T}[i]$.
2. If $i > 1$, determine whether P matches $\tilde{T}[l..i]$ for some $l < i$.
3. Compute $\mathcal{S}_i^{\leq}$.

Position i is output as an ending position of an occurrence of P, if and only if a match is found in either Step 1 or 2. We basically follow previous work for computing Steps 1 and 2, but consider the space usage. Our main contribution is the improvement of Step 3. We note that the set $\mathcal{S}_i^{\leq}$, or more generally any subset of $\{1, \ldots, m\}$ can be represented as a bit array of size m, and determining membership, as well as adding/deleting elements, can be done in $O(1)$ time. Also, set union can be performed in $O(m)$ time.

4.2 Computing Step 1

In Step 1, we simply determine whether P is a substring of some $s \in \tilde{T}[i]$.

▶ **Lemma 5.** *Step 1 can be computed in total of $O(m + \|\tilde{T}\|)$ time for all $1 \leq i \leq n$, using $O(m)$ space.*

Proof. For Step 1, we can use a linear time pattern matching algorithm such as KMP [10]. Since the pattern does not change, the preprocessing of the pattern is done once in $O(m)$ time. The matching can be done in $O(\|\tilde{T}\|)$ time and $O(m)$ space. ◀

4.3 Computing Step 2

Steps 2 (as well as Step 3) is computed using the suffix tree $ST(P)$ of P, and also uses $\mathcal{S}_{i-1}^{\leq}$.

▶ **Lemma 6.** *Given $\mathcal{S}_{i-1}^{\leq}$, Step 2 can be computed in total of $O(m + \|\tilde{T}\|)$ time for all $1 \leq i \leq n$, using $O(m)$ space.*

Proof. We first construct the suffix tree $ST(P)$ of P, which takes $O(m)$ time and space. Since, by definition, a value $j \in \mathcal{S}_{i-1}^{\leq}$ if and only if $j > 1$ and $P[1..j-1]$ is a suffix of some $s \in \mathcal{A}(\tilde{T}[1..i-1])$, we have, as observed previously, an occurrence of P that ends at position i if and only if there exist $j \in \mathcal{S}_{i-1}^{\leq}$ and $t \in \tilde{T}[i]$ such that $P[j..m]$ is a prefix of t.

This can be checked as follows: For each string $t \in \tilde{T}[i]$, traverse the suffix tree from the root with t. We will detect such an occurrence of P, if, at any point in the traversal, we reach a position corresponding to a suffix $P[j..m]$ for some $j \in \mathcal{S}_{i-1}^{\leq}$, i.e., when we are at the locus (u, d) of a prefix $t' = t[1..d]$ of t during the traversal, either (1) u is a leaf that corresponds to the suffix $P[j..m]$ for some $j \in \mathcal{S}_{i-1}^{\leq}$, and $d = m - j + 1$, or, (2) u has an outgoing edge labeled by \$ that leads to a leaf that corresponds to the suffix $P[j..m]$ for some $j \in \mathcal{S}_{i-1}^{\leq}$. Since the traversal can be done in $O(|t|)$ time for each t, the total time for all $1 \leq i \leq n$ is $O(\|\tilde{T}\|)$. ◀

4.4 Computing Step 3

To compute $\mathcal{S}_i^{\leq}$, we will compute the two sets:

$$\begin{aligned}\mathcal{S}_i^{=} &= \{j \mid 1 < j \leq m, \exists s \in \tilde{T}[i] \text{ s.t. } P[1..j-1] \text{ is a suffix of } s\}\\ \mathcal{S}_i^{<} &= \{j + |t| : P[j..j+|t|-1] = t, j \in \mathcal{S}_{i-1}^{\leq}, t \in \tilde{T}[i], j + |t| \leq m\}.\end{aligned}$$

Then, it is clear that $\mathcal{S}_i^{\leq} = \mathcal{S}_i^{=} \cup \mathcal{S}_i^{<}$.

4.4.1 Computing $\mathcal{S}_i^{=}$

We first describe how to compute $\mathcal{S}_i^{=}$.

▶ **Lemma 7.** *$\mathcal{S}_i^{=}$ can be computed in total of $O(m + \|\tilde{T}\|)$ time for all $1 \leq i \leq n$, using $O(m)$ space.*

Proof. Consider the string $P\#\tilde{T}[i][k]$ for each $k = 1, \ldots, |\tilde{T}[i]|$, where # is a character that does not occur in P or $\tilde{T}[i][k]$. Then, it is easy to see that

$$\mathcal{S}_i^{=} = \{|b| + 1 \mid 1 \leq b < m, b \text{ is a proper border of } P\#\tilde{T}[i][k]\}.$$

Since the border array of a string can be computed in an on-line fashion and in linear time [10], the border array of $P\#$ can be computed in $O(m)$ time once. Furthermore, since the last element of the border array of $P\#$ is 0, the remaining elements of the border array of $P\#\tilde{T}[i][k]$, and thus, the lengths of all borders of $P\#\tilde{T}[i][k]$ can be computed in $O(|\tilde{T}[i][k]|)$ time for any $1 \leq k \leq |\tilde{T}[i]|$. Thus, the total time for computing $\mathcal{S}_i^{=}$ is $O(m + \|\tilde{T}\|)$.

We note that the above description is slightly different from that of Grossi et al. [8], where they describe the computation by computing the border array of the string

$$X = P\#_1\tilde{T}[i][1]\#_2 \cdots \#_{|\tilde{T}[i]|}\tilde{T}[i][|\tilde{T}[i]|].$$

Although Grossi et al. mention that the time and space complexities for the preprocessing (for computing the border array of P) are $O(m)$, they do not explicitly mention the space complexity of their matching algorithm. A naive implementation of their description would take $O(m + \max\{\|\tilde{T}[i]\| \mid 1 \leq i \leq n\})$ extra space for computing the border array of X, but this can easily be reduced to $O(m)$ extra space, since (1) we can compute the borders separately for each $\tilde{T}[i][k]$ as described above, and (2) $P\#\tilde{T}[i][k]$ can be replaced with $P\#Y$, where $Y = \tilde{T}[i][k][l - m + 2..l]$ and $l = |\tilde{T}[i][k]|$, to obtain the same result. ◀

4.4.2 Computing $\mathcal{S}_i^{<}$

We first describe how Grossi et al. compute $\mathcal{S}_i^{<}$. Basically, their algorithm is a fairly straightforward approach that uses $ST(P)$. For each $t \in \tilde{T}[i]$, compute the set $\{|t|\} \oplus (Occ(P, t) \cap \mathcal{S}_{i-1}^{\leq})$. Then, $\mathcal{S}_i^{<}$ is the union of this set for all $t \in \tilde{T}[i]$, i.e.,

$$\mathcal{S}_i^{<} = \bigcup_{t \in \tilde{T}[i]} \left(\{|t|\} \oplus (Occ(P, t) \cap \mathcal{S}_{i-1}^{\leq})\right). \tag{1}$$

By Lemma 3, each $\{|t|\} \oplus (Occ(P, t) \cap \mathcal{S}_{i-1}^{\leq})$ can be computed in $O(|t| + |Occ(P, t)|)$ time, and thus the total time is $O(\sum_{t \in \tilde{T}[i]}(|t| + |Occ(P, t)|)) = O(\|\tilde{T}[i]\| + \sum_{t \in \tilde{T}[i]} |Occ(P, t)|)$. Since all strings in $\tilde{T}[i]$ are distinct, $\sum_{t \in \tilde{T}[i]} |Occ(P, t)| = O(m^2)$, thus, giving an algorithm that computes $\mathcal{S}_i^{<}$ in $O(\|\tilde{T}[i]\| + m^2)$ time.

Next, we describe how to improve the running time. Our main idea is: rather than compute the sum set independently for each element $t \in \tilde{T}[i]$, we appropriately group together elements in $\tilde{T}[i]$ so that the efficient sum set computation of Lemma 4 can be used.

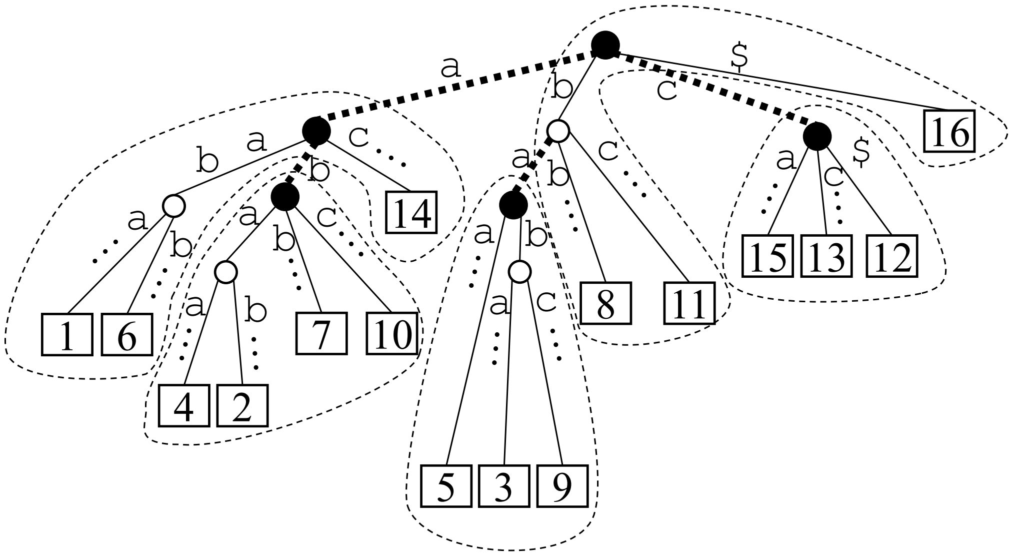

Figure 1 Example of a partition of $ST(P)$ for $P =$ aababaabbabccac with $\tau = 4$. There are 25 nodes (including leaves) in total. The black nodes represent the root of each component, and the dotted edges represent boundary edges. By construction, each component contains at least $\tau = 4$ nodes, so there are $5 \leq 25/4$ components, and any sub-component rooted at a node that is not a root of a component contains less than $\tau = 4$ nodes.

▶ **Lemma 8.** *$\mathcal{S}_i^<$ can be computed in $O(\|\tilde{T}[i]\| + m\sqrt{m \log m})$ time using $O(m)$ working space.*

Proof. To improve the running time, we first partition $ST(P)$ into subtrees as follows, similar to the micro-macro decomposition [2].

Let $\tau > 1$ be a parameter that will be chosen later. Define the weight $W(v)$ of a node v as

$$W(v) = \begin{cases} 1 & v \text{ is a leaf} \\ W'(v) & W'(v) < \tau \\ 0 & W'(v) \geq \tau \text{ or } v \text{ is the root.} \end{cases}$$

where $W'(v) = 1 + \sum_{u \in chldr(v)} W(u)$. The tree is partitioned into subtrees so that all nodes v such that $W(v) = 0$ are roots of the subtrees that comprise the partition, i.e., each incoming edge to a node v with $W(v) = 0$ is a boundary of the partition. Such an edge will be called a *boundary edge*. For all nodes v, it is clear that $W(v)$ — and thus the partition, can be computed in linear time by a post-order traversal on $ST(P)$. We will call each such subtree in the partition a *component*. For any node v, we denote by $C(v)$, the set of nodes that are descendants of v, including v itself, and are in the same component as v. Also, let $Cr(P)$ denote the set of all roots of components of $ST(P)$. Figure 1 shows an example of a partition. The important properties of such a partition are:

1. The total number of nodes (including leaves) contained in each component is $\Omega(\tau)$.
2. The number of components is bounded by $O(m/\tau)$.
3. For any node u that is not a root of a component, $|C(u)| = O(\tau)$.

Now, since $Occ(P, t) = L(v_t)$, where $(v_t, |t|)$ is the locus of t in $ST(P)$, we can rewrite Equation (1) as follows, by partitioning $L(v_t)$ according to the component that each leaf belongs to:

$$\begin{aligned}
&\bigcup_{t\in\tilde{T}[i]}\Big(\{|t|\}\oplus(Occ(P,t)\cap\mathcal{S}^{\le}_{i-1})\Big)\\
=\ &\left(\bigcup_{t\in\tilde{T}[i],v_t\not\in Cr(P)}\Big(\{|t|\}\oplus(L(v_t)\cap C(v_t)\cap\mathcal{S}^{\le}_{i-1})\Big)\right)\cup &(2)\\
&\left(\bigcup_{t\in\tilde{T}[i]}\ \bigcup_{v\in desc(v_t)\cap Cr(P)}\Big(\{|t|\}\oplus(L(v)\cap C(v)\cap\mathcal{S}^{\le}_{i-1})\Big)\right) &(3)
\end{aligned}$$

Furthermore, by grouping together all t that correspond to ancestors of each component when computing the sum set, we can rewrite Term (3) as follows:

$$\bigcup_{v\in Cr(P)}\Big(\{|t| : t\in\tilde{T}[i]\cap v_t\in anc(v)\}\oplus(L(v)\cap C(v)\cap\mathcal{S}^{\le}_{i-1})\Big) \tag{4}$$

First, we consider how to compute Term (2). For any $t\in\tilde{T}[i]$, its locus $(v_t,|t|)$ in $ST(P)$ can be computed in $O(|t|)$ time, if it exists (we can simply ignore any t that does not occur in P). Since we only consider t such that v_t is not a root of a component, $|C(v_t)|=O(\tau)$ (Property 3). Then, all elements in $L(v_t)\cap C(v_t)\cap\mathcal{S}^{\le}_{i-1}$ can be obtained by a simple traversal on $C(v_t)$, and thus $\{|t|\}\oplus(L(v_t)\cap C(v_t)\cap\mathcal{S}^{\le}_{i-1})$ can be obtained in $O(\tau)$ time. Note that this can also be bounded by the size of the subtree of $ST(P)$ rooted at v_t. Thus, the total time for this traversal for all $t\in\tilde{T}[i]$ is $O(\sum_{t\in\tilde{T}[i]}\min\{\tau,|T_{v_t}|\})$, where $|T_{v_t}|$ denotes the size of the subtree of $ST(P)$ rooted at v_t. Now, let $X_S=\{t\mid t\in\tilde{T}[i],|t|\le\tau\}, X_L=\tilde{T}[i]\setminus X_S=\{t\mid t\in\tilde{T}[i],|t|>\tau\}$, i.e., X_S is the set of strings in $\tilde{T}[i]$ shorter than or equal to τ, and X_L is the set of strings in $\tilde{T}[i]$ longer than τ. Then,

$$\begin{aligned}
\sum_{t\in\tilde{T}[i]}\min\{\tau,|T_{v_t}|\} &= \sum_{t\in X_S}\min\{\tau,|T_{v_t}|\}+\sum_{t\in X_L}\min\{\tau,|T_{v_t}|\}\\
&\le \sum_{t\in X_S}|T_{v_t}|+\sum_{t\in X_L}\tau\\
&= \sum_{\ell=1}^{\tau}\ \sum_{t\in X_S,|t|=\ell}|T_{v_t}|+\sum_{t\in X_L}\tau\\
&= O(\tau m+\|\tilde{T}[i]\|).
\end{aligned}$$

Here, the last inequality uses $\sum_{t\in X_S,|t|=\ell}|T_{v_t}|=O(m)$, which is true because all substrings in X_S are distinct, implying that all subtrees rooted at a given depth ℓ of the suffix tree are disjoint and therefore their total size is $O(m)$. Also, $\sum_{t\in X_L}\tau<\sum_{t\in X_L}|t|\le\|\tilde{T}[i]\|$. The total time for computing Term (2) is therefore $O(\tau m+\|\tilde{T}[i]\|)$.

Next, we consider how to compute Term (4). Notice that for any $v\in Cr(P)$, the size of set $\{|t| : t\in\tilde{T}[i]\cap v_t\in anc(v)\}$ is $O(m)$, since $len(v)\le m+1$, and can be obtained in $O(m)$ time provided that all loci of strings in $\tilde{T}[i]$ are marked on $ST(P)$. Also, $L(v)\cap C(v)\cap\mathcal{S}^{\le}_{i-1}$ can also be computed in $O(m)$ time. Since the sets can be computed in $O(m)$ time, the sum set can be computed in $O(m\log m)$ time using Lemma 4. The total time for all components is therefore $O(\frac{m}{\tau}m\log m)$ (Property 2).

From the above arguments, the total time for computing Equation (1) is $O(\|\tilde{T}[i]\|+\tau m+\frac{m^2}{\tau}\log m)$. By choosing $\tau=\sqrt{m\log m}$, we obtain $O(\|\tilde{T}[i]\|+m\sqrt{m\log m})$.

Concerning the space complexity, it is easy to implement the above algorithm in $O(m+|\tilde{T}[i]|)$ space. The term $|\tilde{T}[i]|$ exists because in the above description, we assumed that we had marked the locus of each $t\in\tilde{T}[i]$ on the suffix tree. If we assume that the strings

Algorithm 1: Pseudo code of algorithm for computing $\mathcal{S}_i^<$.

```
Input: $P$, $\tilde{T}[i]$, $\mathcal{S}_{i-1}^{\leq}$, $ST(P)$
Output: $\mathcal{S}_i^<$
// assumes $\tilde{T}[i]$ is lexicographically sorted.
S $= \emptyset$;
Function dfs(ancl, $k$, $(v_k, \ell_k)$, $v$):
    while $v_k = v$ do
        ancl.push($\ell_k$);
        $k \leftarrow k + 1$;
        $(v_k, \ell_k) \leftarrow$ locus of $\tilde{T}[i][k]$ ;        // (null, 0) if $k > |\tilde{T}[i]|$
    if $W(v) = 0$ then                                            // $v$ is a root of a component
        S $\leftarrow$ S $\cup$ (ancl $\oplus$ $(L(v) \cap C(v) \cap \mathcal{S}_{i-1}^{\leq})$);
    for $c \in chldr(v)$ do                       // in lexicographic order of children
        dfs(ancl, $k$, $(v_k, \ell_k)$, $c$);                           // recurse on child
        while ancl.top() $> len(c)$ do                  // discard visited descendants
            ancl.pop();
ancl $\leftarrow$ empty stack;
$r \leftarrow$ root of $ST(P)$;
$(v_1, \ell_1) \leftarrow$ locus of $\tilde{T}[i][1]$;
dfs(ancl, 1, $(v_1, \ell_1)$, $r$);
return S;
```

$\tilde{T}[i][1], \ldots, \tilde{T}[i][|\tilde{T}[i]|]$ are lexicographically sorted, we can reduce the space by doing the computation through a depth-first traversal on the suffix tree from left to right, during which we only maintain the loci on the path we are considering. The space requirement follows, since the length of this path is $O(m)$. This is illustrated in the pseudo-code shown in Algorithm 1. ◀

▶ **Theorem 9.** *Problem 2 can be solved in $O(N + nm^{1.5}\sqrt{\log m})$ time using $O(m)$ working space.*

Proof. For each $i = 1 \ldots, n$, all computations other than Step 3 take $O(\|\tilde{T}[i]\|)$ time, while Step 3 takes $O(\|\tilde{T}[i]\| + m\sqrt{m \log m})$ time. Thus, for all $1 \leq i \leq n$, the total time is $O(\sum_{i=1}^{n} \|\tilde{T}[i]\| + nm\sqrt{m \log m}) = O(N + nm\sqrt{m \log m})$. ◀

If we cannot assume that the strings in each $\tilde{T}[i]$ are sorted in lexicographic order, the space complexity becomes $O(m + \max_{i=1,\ldots,n} |\tilde{T}[i]|)$.

5 Conclusion

We present a new algorithm for the elastic degenerate string matching problem which runs in $O(nm\sqrt{m \log m} + N)$ time using $O(m)$ working space. While previous algorithms for the EDSM problem are basically applications of now "standard" string matching techniques, our algorithm applies a novel technique combining FFT and the suffix tree.

On a side note, it seems interesting that while Boolean convolution was the key technique that Fischer and Paterson [7] used to solve the degenerate pattern matching problem, we solve a generalized version of the problem with the same tool (Boolean convolution) but applying it in a different way.

References

1 Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987. doi:10.1137/0216067.

2 Stephen Alstrup, Jens Peter Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Information Processing Letters*, 64(4):161–164, 1997.

3 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval*, pages 74–90, Cham, 2017. Springer International Publishing.

4 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018. doi:10.1093/bib/bbw089.

5 James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. URL: http://www.jstor.org/stable/2003354.

6 Martin 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*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

7 Michael J. Fischer and Michael S. Paterson. String matching and other products. In Richard M. Karp, editor, *Complexity of Computation*, volume 7 of *SIAM-AMS Proceedings*, pages 113–125, 1974.

8 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-line pattern matching on similar texts. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, volume 78 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.9.

9 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate texts. In Frank Drewes, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, volume 10168 of *Lecture Notes in Computer Science*, pages 131–142, 2017. doi:10.1007/978-3-319-53733-7_9.

10 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

11 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.