



HAL
open science

Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications

Esteban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P Pissis, Jakub Radoszewski, Michelle Sweering, Wiktor Zuba

► **To cite this version:**

Esteban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P Pissis, Jakub Radoszewski, et al.. Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications. 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023), 2023, Marne-la-Vallée, France. 10.4230/LIPIcs.CPM.2023.11 . hal-04365687

HAL Id: hal-04365687

<https://inria.hal.science/hal-04365687v1>

Submitted on 28 Dec 2023



HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.



L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.







Distributed under a Creative Commons Attribution 4.0 International License

Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications

Esteban Gabory  
CWI, Amsterdam, The Netherlands



Nadia Pisanti  
University of Pisa, Italy

Jakub Radoszewski  
Institute of Informatics,
University of Warsaw, Poland

Wiktor Zuba  
CWI, Amsterdam, The Netherlands

Moses Njagi Mwaniki  
University of Pisa, Italy

Solon P. Pissis  
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Michelle Sweering  
CWI, Amsterdam, The Netherlands

Abstract

An elastic-degenerate (ED) string T is a sequence of n sets $T[1], \dots, T[n]$ containing m strings in total whose cumulative length is N . We call n , m , and N the length, the cardinality and the size of T , respectively. The language of T is defined as $\mathcal{L}(T) = \{S_1 \cdots S_n : S_i \in T[i] \text{ for all } i \in [1, n]\}$. ED strings have been introduced to represent a set of closely-related DNA sequences, also known as a pangenome. The basic question we investigate here is: Given two ED strings, how fast can we check whether the two languages they represent have a nonempty intersection? We call the underlying problem the ED STRING INTERSECTION (EDSI) problem. For two ED strings T_1 and T_2 of lengths n_1 and n_2 , cardinalities m_1 and m_2 , and sizes N_1 and N_2 , respectively, we show the following:

- There is no $\mathcal{O}((N_1 N_2)^{1-\epsilon})$ -time algorithm, thus no $\mathcal{O}((N_1 m_2 + N_2 m_1)^{1-\epsilon})$ -time algorithm and no $\mathcal{O}((N_1 n_2 + N_2 n_1)^{1-\epsilon})$ -time algorithm, for any constant $\epsilon > 0$, for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Strong Exponential-Time Hypothesis is false.
- There is no combinatorial $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ -time algorithm, for any constant $\epsilon > 0$ and any function f , for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Boolean Matrix Multiplication conjecture is false.
- An $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ -time algorithm for outputting a compact (RLE) representation of the intersection language of two unary ED strings. In the case when T_1 and T_2 are given in a compact representation, we show that the problem is NP-complete.
- An $\mathcal{O}(N_1 m_2 + N_2 m_1)$ -time algorithm for EDSI.
- An $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$ -time algorithm for EDSI, where ω is the exponent of matrix multiplication; the $\tilde{\mathcal{O}}$ notation suppresses factors that are polylogarithmic in the input size.

We also show that the techniques we develop have applications outside of ED string comparison.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases elastic-degenerate string, sequence comparison, languages intersection, pangenome, acronym identification

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.11

Funding The work in this paper is supported in part: by the PANGAIA and ALPACA projects that have received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreements No 872539 and 956229, respectively; by the Netherlands Organisation for Scientific Research (NWO) through Gravitation-grant NETWORKS-024.002.003; and by PNRR ECS00000017 Tuscany Health Ecosystem Spoke 6 “Precision medicine & personalized healthcare”, funded by the European Commission under the NextGeneration EU programme.

Jakub Radoszewski: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.



© Esteban Gabory, Moses Njagi Mwaniki, Nadia Pisanti, Solon P. Pissis, Jakub Radoszewski, Michelle Sweering, and Wiktor Zuba;
licensed under Creative Commons License CC-BY 4.0

34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023).

Editors: Laurent Bulteau and Zsuzsanna Lipták; Article No. 11; pp. 11:1–11:20



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Sequence (or string) comparison is a fundamental task in computer science, with numerous applications in computational biology [24], signal processing [16], information retrieval [7], file comparison [25], pattern recognition [4], security [36], and elsewhere [37]. Given two or more sequences and a distance function, the task is to compare the sequences in order to infer or visualize their (dis)similarities [15].

Many sequence representations have been introduced over the years to account for *unknown* or *uncertain* letters, a phenomenon that often occurs in data that comes from experiments [8]. In the context of computational biology, for example, the IUPAC notation [27] is used to represent locations of a DNA sequence for which several alternative nucleotides are possible. This gives rise to the notion of *degenerate string* (or *indeterminate string*): a sequence of finite sets of *letters* [2]. When all sets are of size 1, we are in the special case of a *standard string* (or *deterministic string*). Degenerate strings can encode the consensus of a population of DNA sequences [17] in a gapless multiple sequence alignment (MSA). Iliopoulos et al. generalized this notion to also encode insertions and deletions (gaps) occurring in MSAs by introducing the notion of *elastic-degenerate string*: a sequence of finite sets of *strings* [26].

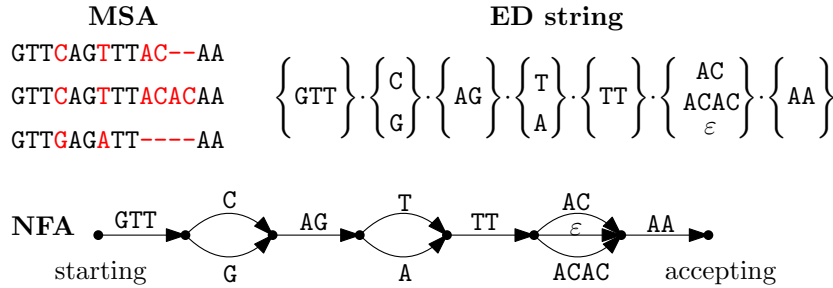
The main motivation to consider elastic-degenerate (ED) strings is that they can be used to represent a *pangenome*: a *collection* of closely-related genomic sequences that are meant to be analyzed together [42]. Several other, more powerful, pangenome representations have been proposed in the literature, mostly graph-based ones; see the comprehensive survey by Carletti et al. [12] or by Baaijens et al. [5]. Compared to these more powerful representations, ED strings have at least two algorithmic advantages, as they support: (i) fast and simple on-line string matching [23, 13]; and (ii) (deterministic) subquadratic string matching [3, 9, 10].

Our main goal here is to give the first algorithms and lower bounds for comparing two pangenomes represented by two ED strings.¹ We consider the most basic notion of matching, namely, to decide whether two ED strings, each encoding a language, have a nonempty intersection. Like with standard strings, algorithms for pairwise ED string comparison can serve as computational primitives for many analysis tasks (e.g., phylogeny reconstruction); lower bounds for pairwise ED string comparison can serve as meaningful lower bounds for more powerful pangenome representations such as, for instance, variation graphs [12].

Let us start with some basic definitions and notation. An *alphabet* Σ is a finite nonempty set of elements called *letters*. By Σ^* we denote the set of all strings over Σ including the *empty string* ε of length 0. For a string $S = S[1] \cdots S[n]$ over Σ , we call $n = |S|$ its *length*. The fragment $S[i..j]$ of S is an *occurrence* of the underlying *substring* $P = S[i] \cdots S[j]$. We also say that P occurs at *position* i in S . A *prefix* of S is a fragment of S of the form $S[1..j]$ and a *suffix* of S is a fragment of S of the form $S[i..n]$. An *elastic-degenerate string* (ED string, in short) T is a sequence $T = T[1] \cdots T[n]$ of n finite sets, where $T[i]$ is a subset of Σ^* . The total size of T is defined as $N = N_\varepsilon + \sum_{i=1}^n \sum_{S \in T[i]} |S|$, where N_ε is the total number of empty strings in T . By m we denote the total number of strings in all $T[i]$, i.e., $m = \sum_{i=1}^n |T[i]|$. We say that T has *length* $n = |T|$, *cardinality* m and *size* $N = ||T||$. An ED string T can be treated as a compacted nondeterministic finite automaton (NFA) with $n + 1$ states, numbered $1, \dots, n + 1$, and m transitions labeled by strings in Σ^* . State 1 is *starting* and state $n + 1$ is *accepting*. For each index $i \in [1, n]$ and string $S \in T[i]$, there is a

¹ Pangenome comparison is one of the central goals of two large EU funded projects on computational pangenomics: PANGAIA (<https://www.pangenome.eu/>) and ALPACA (<https://alpaca-itn.eu/>).

transition from state i to state $i + 1$ with label S ; inspect also Figure 1 for an example. The language $\mathcal{L}(T)$ generated by the ED string T is the language accepted by this compacted NFA. That is, $\mathcal{L}(T) = \{S_1 \cdots S_n : S_i \in T[i] \text{ for all } i \in [1, n]\}$.



■ **Figure 1** An example of an MSA and its corresponding (non-unique) ED string T of length $n = 7$, cardinality $m = 11$ and size $N = 20$, and the compacted NFA for T . The compacted NFA can be seen as a special case of an edge-labeled directed acyclic graph.

We next define the main problem in scope; inspect also Figure 2 for an example.

ED STRING INTERSECTION (EDSI)

Input: Two ED strings, T_1 of length n_1 , cardinality m_1 and size N_1 , and T_2 of length n_2 , cardinality m_2 and size N_2 .

Output: YES if $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection, NO otherwise.

| ED strings | Parameters | $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ |
|---|--------------------------------------|--|
| $T_1 = \left\{ \begin{matrix} \text{a} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{b} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{c} \\ \text{bcd} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{de} \\ \text{cde} \end{matrix} \right\}$ | $n_1 = 4$ $m_1 = 8$ $N_1 = 13$ | abcde accde |
| $T_2 = \left\{ \begin{matrix} \text{a} \\ \varepsilon \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{d} \\ \text{bcd} \\ \text{cc} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \text{e} \\ \text{de} \end{matrix} \right\}$ | $n_2 = 3$ $m_2 = 6$ $N_2 = 10$ | abcdde ade |

■ **Figure 2** An example of two ED strings T_1 and T_2 with their parameters and the intersection of their languages. In this instance, we see that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection.

Our Results. We assume that the ED strings are over an integer alphabet $[1, (N_1 + N_2)^{\mathcal{O}(1)}]$. We make the following specific contributions:

1. In Section 2.1, we give several conditional lower bounds. In particular, we show that there is no $\mathcal{O}((N_1 N_2)^{1-\epsilon})$ -time algorithm, thus no $\mathcal{O}((N_1 m_2 + N_2 m_1)^{1-\epsilon})$ -time algorithm and no $\mathcal{O}((N_1 n_2 + N_2 n_1)^{1-\epsilon})$ -time algorithm, for any constant $\epsilon > 0$, for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Strong Exponential-Time Hypothesis (SETH) [11] is false.
2. In Section 2.2, we present other conditional lower bounds. In particular, we show that there is no combinatorial $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ -time algorithm, for any constant $\epsilon > 0$ and any function f , for EDSI even when T_1 and T_2 are over a binary alphabet, unless the Boolean Matrix Multiplication (BMM) conjecture [1] is false.

3. In Section 3, we show an $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ -time algorithm for outputting a compact (RLE) representation of the intersection language of two unary ED strings. In the case when T_1 and T_2 are given in a compact representation, we show that the problem is NP-complete.
4. In Section 4.1, we show an $\mathcal{O}(N_1 m_2 + N_2 m_1)$ -time algorithm for EDSI.
5. In Section 4.2, we show an $\tilde{\mathcal{O}}(N_1^{\omega-1} n_2 + N_2^{\omega-1} n_1)$ -time algorithm for EDSI, where ω is the matrix multiplication exponent.

Interestingly, we show that the techniques we develop here have applications outside of ED string comparison. Given a sequence $P = P_1, \dots, P_n$ of n standard strings, we define an *acronym* of P as a string $A = A_1 \cdots A_n$, where A_i is a possibly empty prefix of P_i , for all $i \in [1, n]$. In the ACRONYM GENERATION (AG) problem, we are given a set D of k strings of total length K and a sequence P of n strings of total length N , and we are asked to say YES if and only if some acronym of P belongs to D . In Section 5, we show how our techniques for EDSI can be modified to solve AG in $\mathcal{O}(nK + N)$ time.

Related Work. Apart from its applications to pangenome comparison, EDSI is interesting theoretically on its own as a special case of regular expression (regex) matching. Regex is a basic notion in formal languages and automata theory. Regular expressions are commonly used in practical applications to define search patterns. Regex matching and membership testing are widely used as computational primitives in programming languages and text processing utilities (e.g., the widely-used `agrep`). The classic algorithm for solving these problems constructs and simulates an NFA corresponding to the regex, which gives an $\mathcal{O}(MN)$ running time, where M is the length of the pattern and N is the length of the text. Unfortunately, significantly faster solutions are unknown and unlikely [6]. However, much faster algorithms exist for many special cases of the problem: dictionary matching; wildcard matching; subset matching; and the word break problem; see [6] and references therein.

Special cases of EDSI have also been studied. First, let us consider the case when both T_1 and T_2 are degenerate strings. In this case, the problem is trivial: EDSI has a positive answer if and only if for every i , $T_1[i] \cap T_2[i]$ is nonempty. Alzamel et al. [2] studied the case when T_1 and T_2 are *generalized degenerate strings*: for any $i \in [1, n_1]$ and $j \in [1, n_2]$ all strings in $T_1[i]$ have the same length $\ell_{1,i}$ and all strings in $T_2[j]$ have the same length $\ell_{2,j}$. In the case of generalized degenerate strings, they showed that deciding if $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection can be done in $\mathcal{O}(N_1 + N_2)$ time. If T_2 is a standard string, i.e., an ED string with $m_2 = n_2 = 1$, then we can resort to the results of Bernardini et al. [10] for ED string matching. In particular: there is no combinatorial algorithm² for EDSI working in $\mathcal{O}(n_1 N_2^{1.5-\epsilon} + N_1)$ time unless the BMM conjecture¹ is false; and we can solve EDSI in $\tilde{\mathcal{O}}(n_1 N_2^{\omega-1} + N_1)$ time. Moreover, Gawrychowski et al. [21] provided a systematic study of the complexity of degenerate string comparison under different notions of matching: Cartesian tree matching; order-preserving matching; and parameterized matching.

Similar to ED strings (and to generalized degenerate strings) is the representation of pangenomes via *founder graphs*. The idea behind founder graphs is that a multiple alignment of few *founder sequences* can be used to approximate the input MSA, with the feature that each row of the MSA is a recombination of the founders. Like founder graphs, ED strings support the recombination of different rows of the MSA between consecutive columns. Unlike

² The notion of “combinatorial algorithm” is informal but widely used in the literature. Typically, we call an algorithm “combinatorial” if it does not call an oracle for ring matrix multiplication.

ED strings, for which no efficient index is probable [22] (and indeed their value is to enable fast on-line string matching), some subclasses of founder graphs are indexable, and a recent research line is devoted to constructing and indexing such structures [18, 35, 38, 39].

2 Conditional Lower Bounds

In this section, we show several conditional lower bounds for the EDSI problem. Bounds in the first group (see Section 2.1) are based on the popular Strong Exponential-Time Hypothesis (SETH) [11]; the second group of bounds (see Section 2.2) is based on another popular conjecture, the Boolean Matrix Multiplication (BMM) conjecture [1].

2.1 Lower Bounds Based on SETH

We are going to reduce the ORTHOGONAL VECTORS (OV, in short) problem to the EDSI problem. In the OV problem we are given a set $V = \{v^1, \dots, v^k\}$ of k binary vectors, each of length d , and we are asked to decide whether or not there are any two vectors in V which are orthogonal; i.e., the dot product of the two vectors is zero. The OV conjecture, implied by SETH (see [44]), is the following.

► **Conjecture 1** (OV conjecture [44]). *The OV problem for k binary vectors, each of length $d = \Theta(\log k)$, cannot be solved in $\mathcal{O}(k^{2-\epsilon})$ time, for any constant $\epsilon > 0$.*

We show the following reduction.

► **Theorem 2.** *Given any set $V = \{v^1, \dots, v^k\}$ of k binary vectors of length d , we can construct in linear time two ED strings T_1 and T_2 over a binary alphabet such that:*

- T_1 has length, cardinality, and size $\Theta(kd)$;
- T_2 has length $\Theta(\log k)$, cardinality $\Theta(k)$ and size $\Theta(kd)$; and
- V contains two orthogonal vectors if and only if T_1 and T_2 have a nonempty intersection.

Proof. Let $u^i = 1^d - v^i$ for all $i \in [1, k]$. For a length- d vector v and $j \in \{1, \dots, d\}$, by v_j we denote the j th component of v . We construct T_1 and T_2 as follows (see Example 3):³

$$T_1 = \prod_{i=1}^k \prod_{j=1}^d \{0, u_j^i\}, \quad T_2 = \prod_{i=0}^{\lfloor \log_2 k \rfloor} \{0^{d \cdot 2^i}, \varepsilon\} \cdot V \cdot \prod_{i=0}^{\lfloor \log_2 k \rfloor} \{0^{d \cdot 2^i}, \varepsilon\}.$$

We now show that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection if and only if there exists a pair of orthogonal vectors in V .

- Suppose v^a and v^b are orthogonal. Then for all $j \in [1, d]$, $v_j^b \in \{0, u_j^a\}$ and hence $v^b \in \prod_j \{0, u_j^a\}$. It follows that

$$0^{(a-1)d} v^b 0^{(k-a)d} \in 0^{(a-1)d} \prod_j \{0, u_j^a\} 0^{(k-a)d} \subseteq \mathcal{L}(T_1).$$

By decomposing $a-1 = \sum_{i \in S_{a-1}} 2^i$ and $k-a = \sum_{i \in S_{k-a}} 2^i$, where for any integer p , the set S_p contains the positions with a 1 in the binary representation of p , we find that

$$0^{(a-1)d} v^b 0^{(k-a)d} \in \prod_{i \in S_{a-1}} 0^{d \cdot 2^i} \cdot V \cdot \prod_{i \in S_{k-a}} 0^{d \cdot 2^i} \subseteq \mathcal{L}(T_2).$$

We conclude that $0^{(a-1)d} v^b 0^{(k-a)d} \in \mathcal{L}(T_1) \cap \mathcal{L}(T_2)$.

³ By the \prod notation we denote a sequence of concatenations of segments in an ED string.

- Conversely, suppose that $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection and consider a string $S \in \mathcal{L}(T_1) \cap \mathcal{L}(T_2)$. Let v^b be the vector from V which is chosen in T_2 when constructing S . The strings in the sets of T_2 all have length divisible by d . Thus v^b starts at an index $(a-1)d+1$ of string S for some integer a . Since $S \in \mathcal{L}(T_1)$, we have $v^b \in \prod_{j=1}^d \{0, u_j^a\}$. This implies that v^a and v^b are orthogonal.

Therefore, solving the orthogonal vectors problem for V is equivalent to checking whether $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$ have a nonempty intersection. ◀

► **Example 3.** Let $k = 3$, $d = 2$ and $V = \{v^1 = (1, 0), v^2 = (0, 1), v^3 = (1, 1)\}$.

$$\text{We have that } T_1 = \{0\} \cdot \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \cdot \{0\} \cdot \{0\} \cdot \{0\}$$

$$\text{and } T_2 = \begin{Bmatrix} 00 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 0000 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 10 \\ 01 \\ 11 \end{Bmatrix} \cdot \begin{Bmatrix} 00 \\ \varepsilon \end{Bmatrix} \cdot \begin{Bmatrix} 0000 \\ \varepsilon \end{Bmatrix}.$$

One can observe that each string from $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ corresponds to a pair of orthogonal vectors from V . For example, the string 010000 is in $\mathcal{L}(T_2)$ because $v^2 = (0, 1) \in V$. Since the vector $v^1 = (1, 0) \in V$ is orthogonal to v^2 , one also has $010000 \in \mathcal{L}(T_1)$. This is because the two first segments of T_1 are constructed to encode any vector which is orthogonal to v^1 .

Note that when $d = \Theta(\log k)$, the length n_1 , the cardinality m_1 and the size N_1 of T_1 are $\mathcal{O}(k \log k)$, whereas T_2 has length $n_2 = \mathcal{O}(\log k)$, cardinality $m_2 = \mathcal{O}(k)$ and size $N_2 = \mathcal{O}(k \log k)$. Moreover, both ED strings are over a binary alphabet $\Sigma = \{0, 1\}$. This implies various hardness results for EDSI. For example, we can see that, for any constant $\epsilon > 0$, and an alphabet Σ of size at least 2 the problem cannot be solved in

$$\mathcal{O}((N_1 + N_2 + n_1 + n_2)^{2-\epsilon} \cdot \text{poly}(n_2))$$

time, conditional on the OV conjecture. By using the fact that $n_1 \leq m_1 \leq N_1$ and $n_2 \leq m_2 \leq N_2$, we obtain the following bounds.

► **Corollary 4.** *For any constant $\epsilon > 0$, there exists no*

- $\mathcal{O}((N_1 N_2)^{1-\epsilon})$ -time
- $\mathcal{O}((N_1 m_2 + N_2 m_1)^{1-\epsilon})$ -time
- $\mathcal{O}((N_1 n_2 + N_2 n_1)^{1-\epsilon})$ -time

algorithm for the EDSI problem, unless the OV conjecture is false.

2.2 Combinatorial Lower Bounds Based on BMM Conjecture

In the TRIANGLE DETECTION (TD, in short) problem we are given three $D \times D$ Boolean matrices A, B, C and are to check if there are three indices $i, j, k \in [0, D)$ such that $A[i, j] = B[j, k] = C[k, i] = 1$. It is known that Boolean Matrix Multiplication (BMM) and TD either both have truly subcubic combinatorial algorithms or none of them do [45]. The BMM conjecture is stated as follows.

► **Conjecture 5** (BMM conjecture [1]). *Given two $D \times D$ Boolean matrices, there is no combinatorial algorithm for BMM working in $\mathcal{O}(D^{3-\epsilon})$ time, for any constant $\epsilon > 0$.*

Our construction is based on the construction of Bernardini et al. from [10] for ED string matching.

► **Theorem 6.** *If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ time, for any constant $\epsilon > 0$ and any function f , then there exists a truly subcubic combinatorial algorithm for TD.*

Proof. Let D be a positive integer and let A , B , and C be three $D \times D$ Boolean matrices. Further let $s \leq D$ be a positive integer to be set later. In the rest of the proof, we can assume that s divides D , up to adding α rows and columns containing only 0's to all three matrices, where α is the smallest non-negative representative of the equivalence class $-D \bmod s$.

Let us first construct an ED string $T_1 = X_1 X_2 X_3$ over a large alphabet with $n_1 = 3$, where each X_p , $p \in [1, n_1]$, contains a string for each occurrence of value 1 in A , B and C , respectively. Below i iterates over $[0, D)$, j and k over $[0, \frac{D}{s})$, and x and y iterate over $[0, s)$. Moreover, $x, y \in [0, s)$, v_i for $i \in [0, D)$, and $a, \$$ are all letters.

- If $A[i, x \cdot \frac{D}{s} + j] = 1$, then X_1 contains the string $v_i x a^j$;
- If $B[x \cdot \frac{D}{s} + j, y \cdot \frac{D}{s} + k] = 1$, then X_2 contains the string $a^{\frac{D}{s}-j} x \$ \$ y a^{\frac{D}{s}-k}$;
- If $C[y \cdot \frac{D}{s} + k, i] = 1$, then X_3 contains the string $a^k y v_i$;

The length of each string in each X_p is $\mathcal{O}(D/s)$ and the total number m_1 of strings is up to $3D^2$. Overall, $N_1 = \mathcal{O}(D^3/s)$.

We construct an ED string T_2 with $n_2 = 1$ containing the following strings:

$$P(i, x, y) = v_i x a^{\frac{D}{s}} x \$ \$ y a^{\frac{D}{s}} y v_i \quad \text{for every } x, y \in [0, s) \text{ and } i \in [0, D).$$

Each string has length $\mathcal{O}(D/s)$ and there are $m_2 = Ds^2$ strings, so $N_2 = \mathcal{O}(D^2 s)$.

We use the following fact.

► **Fact 7** ([10]). $P(i, x, y) \in \mathcal{L}(T_1)$ if and only if the following holds for some $j, k \in [0, D/s)$:

$$A[i, x \cdot \frac{D}{s} + j] = B[x \cdot \frac{D}{s} + j, y \cdot \frac{D}{s} + k] = C[y \cdot \frac{D}{s} + k, i] = 1.$$

We choose $s = \lfloor \sqrt{D} \rfloor$; then $N_1, N_2 = \mathcal{O}(D^{2.5})$ and $n_1, n_2 = \mathcal{O}(1)$. Then indeed an $\mathcal{O}((N_1 + N_2)^{1.2-\epsilon} f(n_1, n_2))$ -time algorithm for EDSI would yield an $\mathcal{O}(D^{3-2.5\epsilon})$ -time algorithm for the TD problem.

Note also that even though the size of the alphabet used above is $\Theta(s + D) = \Theta(D)$, we can encode all letters by equal-length binary strings blowing N_1 and N_2 up only by a factor of $\Theta(\log D)$ and, hence, obtain the same lower bound for a binary alphabet. ◀

Both m_1 and m_2 in the reduction are $\mathcal{O}(D^2)$, thus an $\mathcal{O}((m_1 + m_2)^{1.5-\epsilon} f(n_1, n_2))$ -time algorithm would yield an $\mathcal{O}(D^{3-2\epsilon})$ -time algorithm for TD; hence we obtain the following.

► **Corollary 8.** *If EDSI over a binary alphabet can be solved in $\mathcal{O}((N_1^{1.2} + N_2^{1.2} + m_1^{1.5} + m_2^{1.5})^{1-\epsilon} f(n_1, n_2))$ time, for any constant $\epsilon > 0$ and any function f , then there exists a truly subcubic combinatorial algorithm for TD.*

3 EDSI: The Unary Case

An ED string is called *unary* if it is over an alphabet of size 1. In this special case, if both T_1 and T_2 are over the same alphabet $\Sigma = \{a\}$, EDSI boils down to checking whether there exists any $b \geq 0$ such that a^b belongs to both $\mathcal{L}(T_1)$ and $\mathcal{L}(T_2)$.

Let T be a unary ED string of length n over alphabet $\Sigma = \{a\}$. We define the *compact representation* $R(T)$ of T as the following sequence of sets of integers:

$$\forall i \in [1, n] \quad R(T)[i] = \{b_{i,1}, b_{i,2}, \dots, b_{i,m_i}\} \iff T[i] = \{a^{b_{i,1}}, a^{b_{i,2}}, \dots, a^{b_{i,m_i}}\},$$

where $b_{i,j} \geq 0$ for all $i \in [1, n]$ and $j \in [1, m_i]$, the cardinality of T is $m = \sum_{i=1}^n m_i$, and its size is $N = N_\epsilon + \sum_{i=1}^n \sum_{j=1}^{m_i} b_{i,j}$, where N_ϵ is the total number of empty strings in T .

► **Theorem 9.** *If T_1 and T_2 are unary ED strings and each is given in a compact representation, the problem of deciding whether $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty is NP-complete.*

Proof. The problem is clearly in NP, as it is enough to guess a single element for each set in both T_1 and T_2 , and then simply check if the sums match in linear time. We show the NP-hardness through a reduction from the SUBSET SUM problem, which takes n integers b_1, b_2, \dots, b_n and an integer c , and asks whether there exist $x_i \in \{0, 1\}$, for all $i \in [1, n]$, such that $\sum_{i=1}^n x_i b_i = c$. SUBSET SUM is NP-complete [30] also for non-negative integers. For any instance of SUBSET SUM, we set $R(T_1)[i] = \{b_i, 0\}$ for all $i \in [1, n]$, $n_2 = 1$ and $R(T_2)[1] = \{c\}$. Then the answer to the SUBSET SUM instance is YES if and only if $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty. ◀

In what follows, we provide an algorithm which runs in polynomial time in the size of the two unary ED strings when the latter are given uncompact.

The set $\mathcal{L}(T)$ can be represented as a set $L(T) \subset \mathbb{N}$ such that $\mathcal{L}(T) = \{a^\ell : \ell \in L(T)\}$. The set $L(T)$ will be stored as a list (without repetitions). We will show how to efficiently compute $L(T_1)$ and $L(T_2)$. Then one can compute $L(T_1) \cap L(T_2)$ in $\mathcal{O}(N_1 + N_2)$ time, which allows, in particular, to check if $L(T_1) \cap L(T_2) = \emptyset$ (which is equivalent to $\mathcal{L}(T_1) \cap \mathcal{L}(T_2) = \emptyset$).

We show the computation for $L(T_1)$. The workhorse is an algorithm from the following Lemma 10 that allows to compute the set $L(X_1 X_2)$ of concatenation of two ED strings based on their sets $L(X_1), L(X_2)$.

► **Lemma 10.** *Let X_1 and X_2 be ED strings. Given $L(X_1)$ and $L(X_2)$ such that $t_1 = \max L(X_1)$ and $t_2 = \max L(X_2)$, we can compute $L(X_1 X_2)$ in $\mathcal{O}((t_1 + t_2) \log(t_1 + t_2))$ time.*

Proof. For two sets $A, B \subset \mathbb{N}$, by $A + B$ we denote the set $\{a + b : a \in A, b \in B\}$. We then have $L(X_1 X_2) = L(X_1) + L(X_2)$. Fast Fourier Transform (FFT) [14] can be used directly to compute $L(X_1) + L(X_2)$ in $\mathcal{O}((t_1 + t_2) \log(t_1 + t_2))$ time. ◀

► **Lemma 11.** *$L(T_1)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1)$ time.*

Proof. We apply the recursive algorithm described in Algorithm 1 to T_1 .

■ **Algorithm 1** Compute- $L(T[1] \cdots T[k])$.

```

if  $k = 1$  then
    Compute  $L(T[1])$  naïvely
 $i \leftarrow \lfloor k/2 \rfloor$ 
 $L_1 \leftarrow$  Compute- $L(T[1] \cdots T[i])$ 
 $L_2 \leftarrow$  Compute- $L(T[i+1] \cdots T[k])$ 
return  $L_1 + L_2$ 
    
```

Let $N_{1,i} = \sum_{x \in L(T_1)[i]} x$ and $t_{1,i} = \max L(T_1[i])$ for $i \in [1, n_1]$. Obviously, $t_{1,i} \leq N_{1,i}$.

We analyze the complexity of the recursion by levels. For the bottom level, $L(T_1[i])$ can be computed in $\mathcal{O}(N_{1,i})$ time for each $i \in [1, n_1]$, which sums up to $\mathcal{O}(N_1)$. For the remaining levels, we notice that $\max L(T_1[i] \cdots T_1[j]) = t_{1,i} + \dots + t_{1,j}$. On each level, the fragments of T_1 that are considered are disjoint. Thus, the complexity on each level via Lemma 10 is $\mathcal{O}((\sum_{i=1}^{n_1} t_{1,i}) \log(\sum_{i=1}^{n_1} t_{1,i})) = \mathcal{O}(N_1 \log N_1)$. The number of levels of recursion is $\mathcal{O}(\log n_1)$; the complexity follows. ◀

► **Theorem 12.** *If T_1 and T_2 are unary ED strings, then $L(T_1) \cap L(T_2)$ can be computed in $\mathcal{O}(N_1 \log N_1 \log n_1 + N_2 \log N_2 \log n_2)$ time.*

Proof. We use Lemma 11 to compute $L(T_1)$ and $L(T_2)$ in the required complexity. Then $L(T_1) \cap L(T_2)$ can be computed via bucket sort. ◀

4 EDSI: General Case

Assuming that the two ED strings, T_1 and T_2 , of total size $N_1 + N_2$ are over an integer alphabet $[1, (N_1 + N_2)^{\mathcal{O}(1)}]$, we can sort the suffixes of all strings in $T_1[i]$, for all $i \in [1, n_1]$, and the suffixes of all strings in $T_2[j]$, for all $j \in [1, n_2]$, in $\mathcal{O}(N_1 + N_2)$ time [19].

By $\text{LCP}(X, Y)$ let us denote the length of the longest common prefix of two strings X and Y . Given a string S over an integer alphabet, we can construct a data structure over S in $\mathcal{O}(|S|)$ time, so that when $i, j \in [1, |S|]$ are given to us on-line, we can determine $\text{LCP}(S[i..|S|], S[j..|S|])$ in $\mathcal{O}(1)$ time [15].

4.1 Compacted NFA Intersection

In this section we show an algorithm for computing a representation of the intersection of the languages of two ED strings using techniques from formal languages and automata theory.

► **Definition 13** (NFA). *A nondeterministic finite automaton (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states; Σ is an alphabet; $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ is a transition function, where $\mathcal{P}(Q)$ is the power set of Q ; $q_0 \in Q$ is the starting state; and $F \subseteq Q$ is the set of accepting states.*

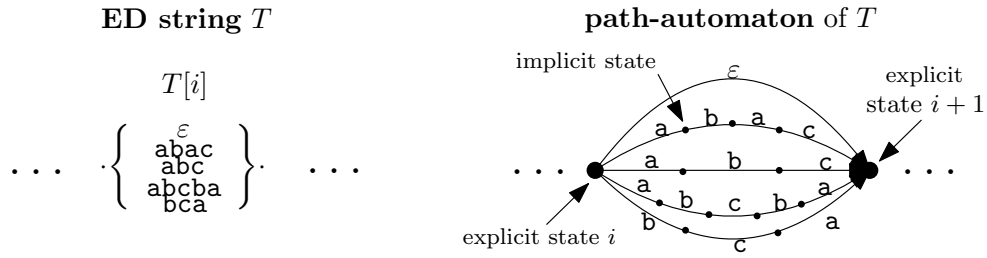
Using the folklore product automaton construction, one can check whether two NFA have a nonempty intersection in $\mathcal{O}(N_1 \cdot N_2)$ time, where N_1 and N_2 are the sizes of the two NFA [33]. We use a different, compacted representation of automata, which in some special cases allows a more efficient algorithm for computing and representing the intersection.

► **Definition 14** (Compacted NFA). *An extended transition is a transition function of the form $\delta^{ext} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$, where Q is a finite set of states, Σ^* is the set of strings over alphabet Σ , and $\mathcal{P}(Q)$ is the power set of Q . A compacted NFA is an NFA in which we allow extended transitions. Such an NFA can also be represented by a standard (uncompacted) NFA, where each extended transition is subdivided into standard one-letter transitions (and ε -transitions), $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$. The states of the compacted NFA are called explicit, while the states obtained due to these subdivisions are called implicit.*

Given a compacted NFA A with V explicit states and E extended transitions, we denote by V^u and E^u the number of states and transitions, respectively, of its uncompacted version A^u . Henceforth we assume that in the given NFA every state is reachable, and hence we have $V^u = \mathcal{O}(E^u)$ and $V = \mathcal{O}(E)$.

► **Lemma 15.** *Given two compacted NFA A_1 and A_2 , with V_1 and V_2 explicit states and E_1 and E_2 extended transitions, respectively, a compacted NFA representing the intersection of A_1 and A_2 with $\mathcal{O}(V_1^u V_2 + V_1 V_2^u)$ explicit states and $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$ extended transitions can be computed in $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$ time.*

Proof. We start by constructing an LCP data structure over the concatenation of all the labels of extended transitions of both NFA of total size $\mathcal{O}(E_1^u + E_2^u)$. It requires $\mathcal{O}(E_1^u + E_2^u)$ -time preprocessing and allows answering LCP queries on any two substrings of such labels in $\mathcal{O}(1)$ time.



■ **Figure 3** On the left: an ED string; on the right: the corresponding path-automaton.

We construct B a compacted NFA representing the intersection of A_1 and A_2 .

Every state of B is composed of a pair: an explicit state of one automaton and any explicit or implicit state of the other automaton (or equivalently a state of the uncompact version of the automaton). Thus the total number of explicit states of B is $\mathcal{O}(V_1^u V_2 + V_1 V_2^u)$.

We need to compute the extended transitions of B . For a state (u, v) we check every string pair (P, Q) , where P iterates over all extended transitions going out of u and Q iterates over all extended transitions going out of v (a transition going out of an implicit state is represented by a suffix of the transition it belongs to). For every pair (P, Q) we ask an $\text{LCP}(P, Q)$ query. If $\text{LCP}(P, Q)$ is equal to one of $|P|, |Q|$ (possibly both), we create an extended transition between (u, v) and the pair of states reachable through those transitions (if one of the transitions is strictly longer, we prune it to the right length, ending it at an implicit state of its input NFA). Otherwise such a transition does not lead to any explicit state of B and thus cannot be used to reach the accepting state; hence we ignore it.

Finally, the starting (resp. accepting) state of B corresponds to a pair of starting (resp. accepting) states of A_1 and A_2 .

Since any pair representing an explicit state of B contains an explicit state of A_1 or A_2 , the number of such transition pair checks (and hence also the number of the extended transitions of B) is $\mathcal{O}(E_1^u E_2 + E_1 E_2^u)$. Since each such check takes $\mathcal{O}(1)$ time, the construction complexity follows. Note that NFA B may contain unreachable states; such states can be removed afterwards in linear time. The algorithms' correctness follows from the observation that B^u is in fact the standard intersection automaton of A_1^u and A_2^u with some states, that do not belong to any path between the starting and the accepting states, removed. ◀

We next define the path-automaton of an ED string (inspect Figure 3 for an example).

► **Definition 16** (Path-automaton). *Let T be an ED string of length n , cardinality m , and size N . The path-automaton of T is the compacted NFA consisting of:*

- $V = n + 1$ explicit states, numbered from 1 through $n + 1$. State 1 is the starting state and state $n + 1$ is the accepting state. State $i \in [2, n]$ is the state in-between $T[i - 1]$ and $T[i]$.
- m_i extended transitions from state i to state $i + 1$ labeled with the strings in $T[i]$, for all $i \in [1, n]$, where $E = m = \sum_i m_i$.

The path-automaton of T accepts exactly $\mathcal{L}(T)$. The uncompact version of this path-automaton has $V^u = \mathcal{O}(N)$ states and $E^u = N$ transitions.

Lemma 15 thus implies the following result.

► **Corollary 17.** *The compacted NFA representing the intersection of two path-automata with $\mathcal{O}(N_1 n_2 + N_2 n_1)$ explicit states and $\mathcal{O}(N_1 m_2 + N_2 m_1)$ extended transitions can be constructed in $\mathcal{O}(N_1 m_2 + N_2 m_1)$ time.*

► **Theorem 18.** *EDSI can be solved in $\mathcal{O}(N_1m_2 + N_2m_1)$ time. If the answer is YES, a witness can be reported within the same time complexity.*

Proof. The path-automaton of an ED string of size N can be constructed in $\mathcal{O}(N)$ time. Given two ED strings, we can construct their path-automata in linear time and apply Corollary 17. By finding any path from the starting to the accepting state in linear time (if it exists), we obtain the result. ◀

Notice that the path-automata representing ED strings, as well as their intersection, are always acyclic, but may contain ε -transitions. In the following we are only interested in the graph underlying the path-automaton, that is the directed acyclic graph (DAG), where every *node* represents an explicit state and every labeled directed *edge* represents an extended transition of the path-automaton (inspect also Figure 1).

4.2 An $\tilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ -time Algorithm for EDSI

In this section, we start by showing a construction of the *intersection graph* computed by means of Lemma 15 in the case when the input is a pair of path-automata that allows an easier and more efficient implementation. The construction is then adapted to obtain an $\tilde{\mathcal{O}}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ -time algorithm for solving the EDSI problem.

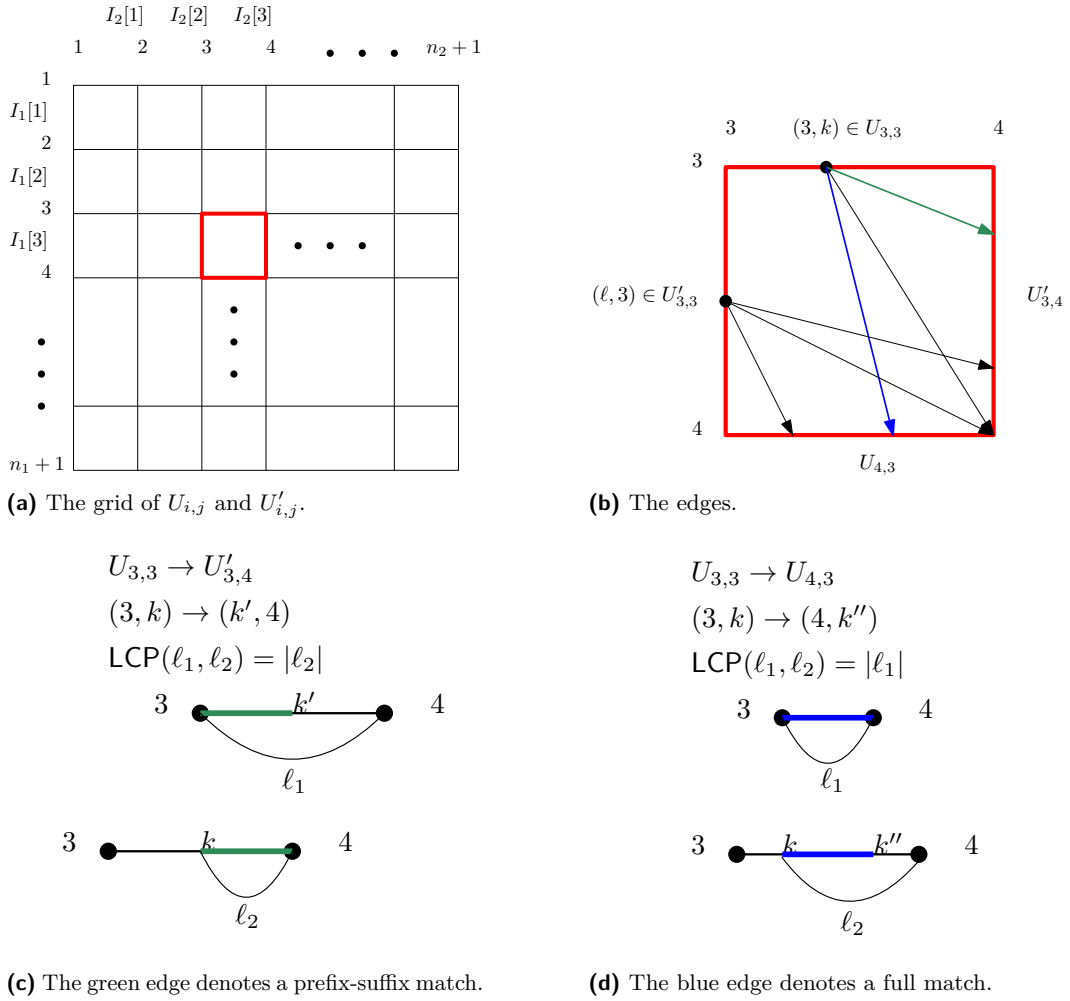
For $x \in \{1, 2\}$ by A_x we denote the compacted NFA (henceforth, graph A_x) representing the ED string T_x . By $I_x[i]$ we denote the set of implicit states (henceforth, implicit nodes) appearing on the extended transitions (henceforth, edges) between explicit states (henceforth, explicit nodes) i and $i + 1$. For convenience, the implicit nodes in the sets $I_x[1], \dots, I_x[n_x]$ can be numbered consecutively starting from $n_x + 2$.

Let $U_{i,j} = \{(i, k) : k \in \{j\} \cup I_2[j]\}$ and $U'_{i,j} = \{(k, j) : k \in \{i\} \cup I_1[i]\}$, for all $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$. As in the construction of Lemma 15, the union of all $U_{i,j}$ and $U'_{i,j}$ is the set of explicit nodes of the intersection graph that we construct; this can be represented graphically by a grid, where the horizontal and vertical lines correspond to $U_{i,j}$ and $U'_{i,j}$, respectively (inspect Figure 4a). In particular, we would like to compute the edges between these explicit nodes (inspect Figure 4b) in $\mathcal{O}(N_1m_2 + N_2m_1)$ time.

Consider an explicit node of the intersection graph; this node is represented by a pair of nodes: one from A_1 and one from A_2 . We need to consider two cases: explicit *vs* explicit node; or explicit *vs* implicit node. Without loss of generality, we consider the first node to be explicit. Let us denote this pair by $(i, k) \in U_{i,j}$, where i is an explicit node of A_1 and k is a node of A_2 . Let us further denote by ℓ_1 the label of one of the edges going from node i to node $i + 1$. For k , we have two cases. If k is explicit (i.e., $k = j$) then we denote by ℓ_2 the label of one of the edges going from k to $k + 1$. Otherwise (k is implicit), we denote by ℓ_2 the path label (concatenation of labels) from node k to node $j + 1$.

As noted in the proof of Lemma 15, an edge is constructed only if $\text{LCP}(\ell_1, \ell_2) = \min(|\ell_1|, |\ell_2|)$. If $\text{LCP}(\ell_1, \ell_2) = |\ell_2| < |\ell_1|$ (a prefix of a string in $T_1[i]$ is equal to the suffix of a string in $T_2[j]$ starting at the position corresponding to node $k \in \{j\} \cup I_2[j]$), the edge ends in a node from $U'_{i,j+1}$ (Figure 4c). If $\text{LCP}(\ell_1, \ell_2) = |\ell_1| < |\ell_2|$ (a whole string from $T_1[i]$ occurs in a string from $T_2[j]$ starting at the position corresponding to node $k \in \{j\} \cup I_2[j]$), the edge ends in a node from $U_{i+1,j}$ (Figure 4d). Otherwise ($\text{LCP}(\ell_1, \ell_2) = |\ell_1| = |\ell_2|$; the two strings are equal) the edge ends in $(i + 1, j + 1)$. Symmetrically (i.e., the second node is explicit), the edge going out of a node from $U'_{i,j}$ ends at a node from the same set $U'_{i,j+1} \cup U_{i+1,j} \cup \{(i + 1, j + 1)\}$ (inspect Figure 4b).

11:12 Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications



■ **Figure 4** An overview of the edges computed by the algorithm.

We next show how to construct the intersection graph by computing all such edges going out of $U_{i,j}$ or $U'_{i,j}$ in a *single batch* using suffix trees (inspect Figure 5 in Appendix A for an example). This construction allows an easier and more efficient implementation in comparison to the LCP data structure used in the general NFA intersection construction. Let us recall that $\|T\|$ denotes the size of the ED string T .

► **Lemma 19.** *For any $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$, we can construct all $K_{i,j}$ edges going out of nodes in $U_{i,j}$ in $\mathcal{O}(N_{1,i} + N_{2,j} + K_{i,j})$ time, where $N_{1,i} = \|T_1[i]\|$ and $N_{2,j} = \|T_2[j]\|$, using the generalized suffix tree of the strings in $T_2[j]$.*

Proof. We first construct the generalized suffix tree of the strings in $T_2[j]$ in $\mathcal{O}(N_{2,j})$ time [19]. We also mark each node corresponding to a suffix of a string in $T_2[j]$ with a T_2 -label. Each such node is also decorated with one or multiple starting positions, respectively, from one or multiple elements of $T_2[j]$ sharing the same suffix. For each branching node of the suffix tree, we construct a hash table, to ensure that any outgoing edge can be retrieved in constant time based on the first letter (the key) of its label. This can be done in $\mathcal{O}(N_{2,j})$ time with perfect hashing [20]. We next spell each string from $T_1[i]$ from the root of the suffix tree

making implicit nodes explicit or adding new ones if necessary to create the compacted trie of all those strings; and, finally, we mark the reached nodes of the suffix tree with a T_1 -label. Spelling all strings from $T_1[i]$ takes $\mathcal{O}(N_{1,i})$ time.

Every pair of different labels marking two nodes in an ancestor-descendant relationship corresponds to exactly one outgoing edge of the nodes in $U_{i,j}$: (i) if a node marked with a T_2 -label is an ancestor of a node marked with a T_1 -label, then the suffix of a string from $T_2[j]$ matches a prefix of a string from $T_1[i]$ forming an edge ending in $U'_{i,j+1}$; (ii) if a node marked with a T_1 -label is an ancestor of a node marked with a T_2 -label, then a string from $T_1[i]$ occurs in a string from $T_2[j]$ extending its prefix and forming an edge ending in $U_{i+1,j}$; (iii) if a node is marked with a T_1 -label and with a T_2 -label, then the suffix of a string from $T_2[j]$ matches a string from $T_1[i]$ forming an edge ending in $(i+1, j+1)$. After constructing the generalized suffix tree of $T_2[j]$ and spelling the strings from $T_1[i]$, it suffices to make a DFS traversal on the annotated tree to output all $K_{i,j}$ such pairs of nodes. ◀

► **Theorem 20.** *We can construct the intersection graph of T_1 and T_2 in $\mathcal{O}(N_1m_2 + N_2m_1)$ time using the suffix tree data structure and tree search traversals.⁴*

Proof. We apply Lemma 19 for $U_{i,j}$ and $U'_{i,j}$, for all $i \in [1, n_1 + 1]$ and $j \in [1, n_2 + 1]$. We have that the total number of nodes is $\sum_{i,j} \mathcal{O}(N_{1,i} + N_{2,j}) = \mathcal{O}(N_1n_2 + N_2n_1)$, and then the sum of all output edges is bounded by $\mathcal{O}(N_1m_2 + N_2m_1)$ by Corollary 17. ◀

Note that if we are interested only in checking whether the intersection is nonempty, and not in the computation of its graph representation, it suffices to check which of the nodes are *reachable* from the starting node, which may be more efficient as there are $\mathcal{O}(N_1n_2 + N_2n_1)$ explicit nodes in this graph.

Let X be the set of nodes of $U_{i,j}$ that are reachable from the starting node. From this set of nodes we need to compute two types of edges (inspect Figure 4b). The first type of edges, namely, the ones from X to $U'_{i,j+1} \cup \{(i+1, j+1)\}$ (green edges in Figure 4b) are computed by means of Lemma 21, which is similar to Lemma 19. For the second type of edges, namely, the ones from X to $U_{i+1,j} \cup \{(i+1, j+1)\}$ (blue edges in Figure 4b), we use a reduction to the so-called *active prefixes extension* problem [10] (Lemma 23).

► **Lemma 21.** *For any given $X \subseteq U_{i,j}$, we can compute the subset of $U'_{i,j+1} \cup \{(i+1, j+1)\}$ containing all and only the nodes that are reachable from the nodes of X in $\mathcal{O}(N_{1,i} + N_{2,j})$ time.*

Proof. In Lemma 19, the edges from nodes of $U_{i,j}$ to nodes of $U'_{i,j+1}$ come from a pair of nodes in the generalized suffix tree of $T_2[j]$: one marked with a T_1 -label and its ancestor marked with a T_2 -label. Notice that the T_2 -labels are in a correspondence with the elements of $U_{i,j}$ (the labels on a proper suffix of a string in T_2 are in a one-to-one correspondence with $U_{i,j} \setminus \{(i, j)\}$, and (i, j) corresponds to whole strings in $T_2[j]$), and hence we can trivially remove the T_2 -labels that do not correspond to the elements of X . Furthermore, we are not interested in the set of starting positions decorating a node with a T_2 -label; we are interested only in whether a node is T_2 -labeled or not (i.e., we do not care from which node of X the edge originates). Since the nodes marked with a T_1 -label have in total $N_{1,i}$ ancestors (including duplicates), we can compute the result of this case in $\mathcal{O}(N_{1,i} + N_{2,j})$ time in total. Finally, the node $(i+1, j+1)$ is reachable when a single node is marked with both a T_1 -label and a T_2 -label. This can be checked within the same time complexity. ◀

⁴ Our implementation of this algorithm can be found at <https://github.com/urbanslug/junctions>.

11:14 Comparing Elastic-Degenerate Strings: Algorithms, Lower Bounds, and Applications

The remaining edges (blue edges in Figure 4b) are dealt with via a reduction to the following problem:

ACTIVE PREFIXES EXTENSION (APE)

Input: A string P of length m , a bit vector W of size m , and a set \mathcal{S} of strings of total length N .

Output: A bit vector V of size m with $V[p] = 1$ if and only if there exists $P' \in \mathcal{S}$ and $p' \in [1, m]$, such that $P[1..p' - 1] \cdot P' = P[1..p - 1]$ and $W[p'] = 1$.

Bernardini et al. have shown the following result in [10].

► **Lemma 22** ([10]). *The APE problem can be solved in $\tilde{\mathcal{O}}(m^{\omega-1}) + \mathcal{O}(N)$ time, where ω is the matrix multiplication exponent.*

► **Lemma 23.** *For any given $X \subseteq U_{i,j}$, we can compute the subset of $U_{i+1,j} \cup \{(i+1, j+1)\}$ containing all and only the nodes that are reachable from the nodes of X in $\tilde{\mathcal{O}}(N_{1,i} + N_{2,j}^{\omega-1})$ time.*

Proof. The problems of computing the subset of $U_{i+1,j}$ reachable from X and the APE problem can be reduced to one another in linear time.

For the forward reduction, let us set $\mathcal{S} = T_1[i]$ and $P = \prod_{S \in T_2[j]} \S , where $\$$ is a letter outside of the alphabet of T_1 . This means that we order the strings in $T_2[j]$, in an arbitrary but fixed way. For a single string $\$S$ (where $S \in T_2[j]$), the positions from $S[1..|S| - 1]$ correspond to the implicit nodes (along the path spelling S) of $I_2[j]$, while the position with $\$$ corresponds to the explicit node j of A_2 and the one with $S[|S|]$ to the explicit node $j + 1$ of A_2 . Through this correspondence, we can construct two bit vectors W and V , each of them of size $|P|$, and whose positions are in correspondence with $\{j\} \cup I_2[j] \cup \{j + 1\}$ (note that this correspondence is not a bijection, as the explicit nodes j and $j + 1$ have several preimages when $|T_2[j]| \geq 2$). As $U_{i,j} \cup \{(i, j + 1)\}$ and $U_{i+1,j} \cup \{(i + 1, j + 1)\}$ are copies of $\{j\} \cup I_2[j] \cup \{j + 1\}$, we use the same correspondence to match positions between W and $U_{i,j} \cup \{(i, j + 1)\}$ and between V and $U_{i+1,j} \cup \{(i + 1, j + 1)\}$. Finally, we set $W[k] = 1$ if and only if the corresponding node of $U_{i,j}$ belongs to X (for k corresponding to $(i, j + 1)$, we set $W[k] = 0$ as such a node cannot belong to X). After solving APE, we have $V[k] = 1$ for some⁵ k corresponding to a node of $U_{i+1,j} \cup \{(i + 1, j + 1)\}$ if and only if this node is reachable from X .

In more detail, observe that since $\$$ does not belong to the alphabet of T_1 , a string S from $T_1[i]$ has to match a fragment of a string from $T_2[j]$ to set $V[k]$ to 1. This happens only if additionally $W[k - |S|] = 1$; both things happen at the same time exactly when: (i) there exists a node $(i, \ell) \in X$; (ii) there exists an edge from (i, ℓ) to $(i + 1, \ell')$; and (iii) the positions $k - |S|$ and k in P correspond to ℓ, ℓ' , respectively.

In the above reduction we have $|P| = \sum_{S \in T_2[j]} |S| + 1 = \mathcal{O}(N_{2,j})$, and $|\mathcal{S}| = N_{1,j}$, hence the lemma statement follows by Lemma 22.

For the reverse reduction, given an instance of APE, we encode it by setting $T_1[i] = \mathcal{S}$, $T_2[j] = \{P\}$ ($N_{1,i} = |\mathcal{S}|$, $N_{2,j} = |P|$) and X containing the nodes corresponding to positions k where $W[k] = 1$ (the last element of such X is potentially $(i + 1, j)$, but we do not care about this corner case of extending the prefix which is already the full string P).

This reduction shows that a more efficient solution to the problem of finding the endpoints of edges originating in X would result in a more efficient solution to the APE problem. ◀

⁵ Here, note that if the node is $(i + 1, j)$ or $(i + 1, j + 1)$, then a corresponding k is not unique, but at least one of them satisfy $V[k] = 1$.

► **Theorem 24.** *We can solve EDSI in $\tilde{O}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1)$ time, where ω is the matrix multiplication exponent. If the answer is YES, we can output a witness within the same time complexity.*

Proof. It suffices to set the starting node $(1, 1)$ as reachable, apply Lemmas 21 and 23, and their symmetric versions for $U'_{i,j}$, for each value of $(i, j) \in [1, n_1 + 1] \times [1, n_2 + 1]$ in lexicographical order, with X equal to the set of reachable nodes of $U_{i,j}$ (respectively of $U'_{i,j}$); and, finally, check whether node $(n_1 + 1, n_2 + 1)$ is set as reachable. We bound the total time complexity of the algorithm by:

$$\sum_{i,j} \tilde{O}(N_{1,i}^{\omega-1} + N_{2,j}^{\omega-1}) = \tilde{O}(n_2 \sum_i N_{1,i}^{\omega-1} + n_1 \sum_j N_{2,j}^{\omega-1}) \leq \tilde{O}(N_1^{\omega-1}n_2 + N_2^{\omega-1}n_1).$$

If $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ is nonempty, that is, if the node $(n_1 + 1, n_2 + 1)$ is set as reachable from node $(1, 1)$, then we can additionally output a witness of the intersection – a single string from $\mathcal{L}(T_1) \cap \mathcal{L}(T_2)$ – within the same time complexity. To do that we mimic the algorithm on the graph with reversed edges. This time, however, we do not mark all of the reachable nodes; we rather choose a single one that was also reachable from $(1, 1)$ in the forward direction. This way, the marked nodes form a single path from $(1, 1)$ to $(n_1 + 1, n_2 + 1)$. The witness is obtained by reading the labels on the edges of this path. ◀

5 Acronym Generation

In this section we study a problem on standard strings. Given a sequence $P = P_1, \dots, P_n$ of n strings we define an *acronym* of P as a string $A = A_1 \dots A_n$, where A_i is a (possibly empty) prefix of P_i , $i \in [1, n]$. We next formalize the ACRONYM GENERATION problem.

ACRONYM GENERATION (AG)

Input: A set D of k strings of total length K and a sequence $P = P_1, \dots, P_n$ of n strings of total length N .

Output: YES if some acronym of P is an element of D , NO otherwise.

The AG problem is underlying real-world information systems (e.g., see <https://acronymify.com/> or <https://acronym-generator.com/>) and existing approaches rely on brute-force algorithms or heuristics to address different variants of the problem [41, 40, 32, 34, 28, 43, 29, 31]. These algorithms usually accept a sequence P of $n \leq n_{\max}$ strings, for some small integer n_{\max} , which highlights the lack of efficient exact algorithms for generating acronyms. Here we show an exact polynomial-time algorithm to solve AG for any n .

We can encode AG by means of EDSI and modify the developed methods. Let $T_1[i]$, $i \in [1, n]$, be the set of all prefixes of P_i and further let $T_2[1] = D$. By using Theorem 18 or Corollary 17 we obtain an $\mathcal{O}(\sum_i |P_i|^2 k + KN) = \mathcal{O}(N^2 k + KN)$ -time algorithm, while using Theorem 24 we obtain an $\tilde{O}(N^{2\omega-2} + K^{\omega-1}n)$ -time algorithm, for solving the AG problem.

Since, however, all elements of set $T_1[i]$ are prefixes of a single string (P_i), we can obtain a more efficient graph representation of T_1 by joining nodes i and $i + 1$ with a single path labeled with P_i , with an additional ε edge between every (implicit) node of the path and node $i + 1$. As the size of the graph for T_1 is smaller ($\mathcal{O}(N)$ nodes and edges), by using Lemma 15 we obtain an $\mathcal{O}(Nk + KN) = \mathcal{O}(NK)$ -time algorithm for solving the AG problem.

The considered ED strings have additional strong properties however. $T_1[i]$'s are not just sets of prefixes of single strings, but sets of all their prefixes, while the length n_2 of T_2 is equal to 1. By employing these two properties we obtain the following improved result.

► **Theorem 25.** *AG can be solved in $\mathcal{O}(nK + N)$ time.*

Proof. The algorithm of Theorem 24 is based on finding out which elements of sets $U_{i,j}, U'_{i,j}$ are reachable; however, since $n_2 = 1$, the sets $U'_{i,j}$ are trivialized: by definition, a node from the middle of $T_1[i]$ cannot correspond to the starting or accepting node of the graph of T_2 (reading a letter in the first graph means also moving out of the starting node in the second one), hence the only possible reachable node of $U'_{i,j}$ is the explicit node (i, j) , which is also an element of $U_{i,j}$. More formally, a reachable node $(k, 1) \in U'_{i,1}$ must be equal to $(i, 1)$ as other such nodes can only be reached using some edge with a nonempty label. By symmetry, nodes from $U'_{i,2}$ other than $(i, 2)$ are not backwards reachable from the accepting node.

In Lemma 23, to compute the reachable nodes of $U_{i+1,j}$ knowing the reachable nodes of $U_{i,j}$, fast matrix multiplication is employed (Lemma 22), but in this special case a simpler method will be more effective. Let W_k be the string read between nodes k and 2 in the path-graph of T_2 . The crucial observation is: the edges going out of node $(i, k) \in U_{i,1} \cup \{(i, 2)\}$ for $k \neq 1$ end in nodes $(i+1, k')$ for $k' \in [k, k+l]$, where $l = \text{LCP}(P_i, W_k)$ as the strings from $T_1[i]$ matching the prefix of W_k are exactly all the prefixes of P_i of length at most l .

Hence to compute the reachable subset of $U_{i+1,1} \cup \{(i+1, 2)\}$, we can handle the edges going out of $(i, 1)$ separately in $\mathcal{O}(K + |P_i|)$ time by letter comparisons, then compute the $\text{LCP}(P_i, W_k)$ for all the reachable nodes (i, k) either using the LCP data structure, or with the use of the generalized suffix tree of $T_2[1] = D$ in $\mathcal{O}(K + |P_i|)$ total time, and finally, using a sweep line approach, compute the union of the obtained intervals in $\mathcal{O}(K)$ time. We answer YES if and only if node $(n+1, 2)$ is reachable.

Over all i values this gives an algorithm running in $\sum_i \mathcal{O}(K + |P_i|) = \mathcal{O}(nK + N)$ total time. Furthermore one is allowed to choose, for each $i \in [1, n]$, the minimal length x_i of the prefix of P_i (including length $x_i = 0$ if one wants to allow empty prefixes) used in the acronym (some strings should not be completely excluded from the acronym). The only modification to the algorithm in such a generalized case is replacing intervals $[k, k+l]$ by $[k+x_i, k+l]$, which does not influence the claimed complexity. ◀

► **Corollary 26.** *If the answer to the instance of the AG problem is YES, we can output all strings in D which are acronyms of P within $\mathcal{O}(nK + N)$ time.*

Proof. In the algorithm employed by Theorem 25 the reachable nodes of $U_{i,1} \cup \{(i, 2)\}$ are found. When the node $(i+1, 2)$ is the endpoint of an edge starting in node (i, k) for $k \neq 1$, then the path of the path-graph of T_2 containing node k is an acronym of P . If node $(i+1, 2)$ is reached directly from reachable node $(i, 1)$, then the whole prefix of P_i used to do that is in D , and hence is a standalone acronym of P . If for a path neither of the two cases qualifies, then it cannot be used to reach node $(n+1, 2)$, and hence is not an acronym of P .

If the generalization with minimal lengths of prefixes is applied, then the values of i used here are restricted to $[i', n]$, where i' is the largest value of i with a restriction $x_i > 0$: node $(i'-1, 2)$ cannot have an edge to node $(i', 2)$, and hence does not lie on a path from $(1, 1)$ to $(n+1, 2)$. ◀

Let us remark that although the main focus of real-world acronym generation systems is on the natural language parsing and interpretation of acronyms, our new algorithmic solution may inspire practical improvements in such systems or further algorithmic work.

References

- 1 Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 434–443. IEEE Computer Society, 2014. doi:10.1109/FOCS.2014.53.
- 2 Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Comparing degenerate strings. *Fundamenta Informaticae*, 175(1-4):41–58, 2020. doi:10.3233/FI-2020-1947.
- 3 Kotaro Aoyama, Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Faster online elastic degenerate string matching. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 – Qingdao, China*, volume 105 of *LIPICs*, pages 9:1–9:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.CPM.2018.9.
- 4 Lorraine A. K. Ayad, Carl Barton, and Solon P. Pissis. A faster and more accurate heuristic for cyclic edit distance computation. *Pattern Recognition Letters*, 88:81–87, 2017. doi:10.1016/j.patrec.2017.01.018.
- 5 Jasmijn A. Baaijens, Paola Bonizzoni, Christina Boucher, Gianluca Della Vedova, Yuri Pirola, Raffaella Rizzi, and Jouni Sirén. Computational graph pangenomics: a tutorial on data structures and their applications. *Nat. Comput.*, 21(1):81–108, 2022. doi:10.1007/s11047-022-09882-6.
- 6 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 457–466. IEEE Computer Society, 2016. doi:10.1109/FOCS.2016.56.
- 7 Ricardo Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval – the concepts and technology behind search, Second edition*. Pearson Education Ltd., Harlow, England, 2011. URL: <http://www.mir2ed.org/>.
- 8 Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering. Hide and mine in strings: Hardness and algorithms. In Claudia Plant, Haixun Wang, Alfredo Cuzzocrea, Carlo Zaniolo, and Xindong Wu, editors, *20th IEEE International Conference on Data Mining, ICDM 2020, Sorrento, Italy, November 17-20, 2020*, pages 924–929. IEEE, 2020. doi:10.1109/ICDM50108.2020.00103.
- 9 Giulia Bernardini, Paweł Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPICs*, pages 21:1–21:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ICALP.2019.21.
- 10 Giulia Bernardini, Paweł Gawrychowski, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Elastic-degenerate string matching via fast matrix multiplication. *SIAM Journal on Computing*, 51(3):549–576, 2022. doi:10.1137/20M1368033.
- 11 Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In Jianer Chen and Fedor V. Fomin, editors, *Parameterized and Exact Computation, 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10-11, 2009, Revised Selected Papers*, volume 5917 of *Lecture Notes in Computer Science*, pages 75–85. Springer, 2009. doi:10.1007/978-3-642-11269-0_6.
- 12 Vincenzo Carletti, Pasquale Foggia, Erik Garrison, Luca Greco, Pierluigi Ritrovato, and Mario Vento. Graph-based representations for supporting genome data analysis and visualization: Opportunities and challenges. In Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia, editors, *Graph-Based Representations in Pattern Recognition – 12th IAPR-TC-15 International Workshop, GBRPR 2019, Tours, France, June 19-21, 2019, Proceedings*, volume 11510 of *Lecture Notes in Computer Science*, pages 237–246. Springer, 2019. doi:10.1007/978-3-030-20081-7_23.

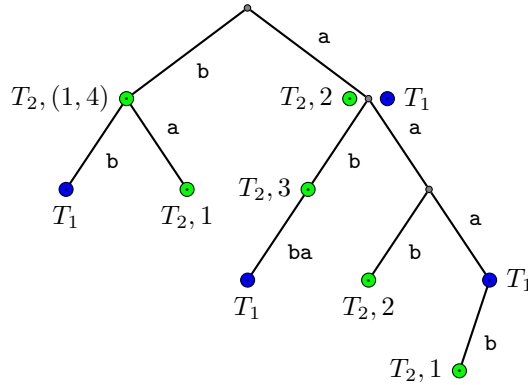
- 13 Aleksander Cisłak, Szymon Grabowski, and Jan Holub. SOPanG: online text searching over a pan-genome. *Bioinformatics*, 34(24):4290–4292, 2018. doi:10.1093/bioinformatics/bty506.
- 14 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- 15 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
- 16 N. Rex Dixon and Thomas B. Martin. *Automatic Speech and Speaker Recognition*. John Wiley & Sons, Inc., USA, 1979.
- 17 Esteban Domingo, Carlos García-Crespo, and Celia Perales. Historical perspective on the discovery of the quasispecies concept. *Annual Review of Virology*, 8(1):51–72, 2021. PMID: 34586874. doi:10.1146/annurev-virology-091919-105900.
- 18 Massimo Equi, Tuukka Norri, Jarno Alanko, Bastien Cazaux, Alexandru I. Tomescu, and Veli Mäkinen. Algorithms and complexity on indexing elastic founder graphs. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan*, volume 212 of *LIPICs*, pages 20:1–20:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.ISAAC.2021.20.
- 19 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS 1997, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.
- 20 Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.
- 21 Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau. On indeterminate strings matching. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPICs*, pages 14:1–14:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.CPM.2020.14.
- 22 Daniel Gibney. An efficient elastic-degenerate text index? Not likely. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval – 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 76–88. Springer, 2020. doi:10.1007/978-3-030-59212-7_6.
- 23 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 für Informatik, 2017. doi:10.4230/LIPICs.CPM.2017.9.
- 24 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.
- 25 Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978. doi:10.1145/359460.359467.
- 26 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate strings. *Information and Computation*, 279:104616, 2021. doi:10.1016/j.ic.2020.104616.
- 27 IUPAC-IUB Commission on Biochemical Nomenclature. Abbreviations and symbols for nucleic acids, polynucleotides, and their constituents. *Biochemistry*, 9(20):4022–4027, 1970. doi:10.1016/0022-2836(71)90319-6.

- 28 Kayla Jacobs, Alon Itai, and Shuly Wintner. Acronyms: identification, expansion and disambiguation. *Annals of Mathematics and Artificial Intelligence*, 88(5-6):517–532, 2020. doi:10.1007/s10472-018-9608-8.
- 29 Katrin Kirchhoff and Anne M. Turner. Unsupervised resolution of acronyms and abbreviations in nursing notes using document-level context models. In Cyril Grouin, Thierry Hamon, Aurélie Névél, and Pierre Zweigenbaum, editors, *Proceedings of the Seventh International Workshop on Health Text Mining and Information Analysis, Louhi@EMNLP 2016, Austin, TX, USA, November 5, 2016*, pages 52–60. Association for Computational Linguistics, 2016. doi:10.18653/v1/W16-6107.
- 30 Jon M. Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2006.
- 31 Divesh R. Kubal and Apurva Nagvenkar. Effective ensembling of transformer based language models for acronyms identification. In Amir Pouran Ben Veyseh, Franck Deroncourt, Thien Huu Nguyen, Walter Chang, and Leo Anthony Celi, editors, *Proceedings of the Workshop on Scientific Document Understanding co-located with 35th AAAI Conference on Artificial Intelligence, SDU@AAAI 2021, Virtual Event, February 9, 2021*, volume 2831 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: <http://ceur-ws.org/Vol-2831/paper24.pdf>.
- 32 Cheng-Ju Kuo, Maurice H. T. Ling, Kuan-Ting Lin, and Chun-Nan Hsu. BIOADI: a machine learning approach to identifying abbreviations and definitions in biological literature. *BMC Bioinformatics*, 10(S-15):7, 2009. doi:10.1186/1471-2105-10-S15-S7.
- 33 Mark V. Lawson. *Finite Automata*. Chapman and Hall/CRC, 2004.
- 34 Jie Liu, Caihua Liu, and Yalou Huang. Multi-granularity sequence labeling model for acronym expansion identification. *Information Sciences*, 378:462–474, 2017. doi:10.1016/j.ins.2016.06.045.
- 35 Veli Mäkinen, Bastien Cazaux, Massimo Equi, Tuukka Norri, and Alexandru I. Tomescu. Linear time construction of indexable founder block graphs. In Carl Kingsford and Nadia Pisanti, editors, *20th International Workshop on Algorithms in Bioinformatics, WABI 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 172 of *LIPICs*, pages 7:1–7:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.WABI.2020.7.
- 36 Udi Manber and Sun Wu. An algorithm for approximate membership checking with application to password security. *Information Processing Letters*, 50(4):191–197, 1994. doi:10.1016/0020-0190(94)00032-8.
- 37 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. doi:10.1145/375360.375365.
- 38 Nicola Rizzo and Veli Mäkinen. Indexable elastic founder graphs of minimum height. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic*, volume 223 of *LIPICs*, pages 19:1–19:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.CPM.2022.19.
- 39 Nicola Rizzo and Veli Mäkinen. Linear time construction of indexable elastic founder graphs. In Cristina Bazgan and Henning Fernau, editors, *Combinatorial Algorithms – 33rd International Workshop, IWOCA 2022, Trier, Germany, June 7-9, 2022, Proceedings*, volume 13270 of *Lecture Notes in Computer Science*, pages 480–493. Springer, 2022. doi:10.1007/978-3-031-06678-8_35.
- 40 Ariel S. Schwartz and Marti A. Hearst. A simple algorithm for identifying abbreviation definitions in biomedical text. In Russ B. Altman, A. Keith Dunker, Lawrence Hunter, and Teri E. Klein, editors, *Proceedings of the 8th Pacific Symposium on Biocomputing, PSB 2003, Lihue, Hawaii, USA, January 3-7, 2003*, pages 451–462, 2003. URL: <http://psb.stanford.edu/psb-online/proceedings/psb03/schwartz.pdf>.
- 41 Kazem Taghva and Jeff Gilbreth. Recognizing acronyms and their definitions. *International Journal on Document Analysis and Recognition*, 1(4):191–198, 1999. doi:10.1007/s100320050018.
- 42 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135, 2018.

- 43 Amir Pouran Ben Veyseh, Franck Dernoncourt, Quan Hung Tran, and Thien Huu Nguyen. What does this acronym mean? Introducing a new dataset for acronym identification and disambiguation. In Donia Scott, Núria Bel, and Chengqing Zong, editors, *Proceedings of the 28th International Conference on Computational Linguistics, COLING 2020, Barcelona, Spain (Online), December 8-13, 2020*, pages 3285–3301. International Committee on Computational Linguistics, 2020. doi:10.18653/v1/2020.coling-main.292.
- 44 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005. doi:10.1016/j.tcs.2005.09.023.
- 45 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 645–654. IEEE Computer Society, 2010. doi:10.1109/FOCS.2010.67.

A Omitted Figure

Figure 5 illustrates an example of the algorithm underlying Lemma 19.



■ **Figure 5** The annotated compacted trie constructed for $T_1[i] = \left\{ \begin{matrix} abba \\ aaa \\ bb \\ a \end{matrix} \right\}$ and $T_2[j] = \left\{ \begin{matrix} ba \\ aaab \\ b \end{matrix} \right\}$

in Lemma 19. The node corresponding to **b** has two T_2 labels and is an ancestor of the node corresponding to **bb** with a T_1 label; hence two corresponding edges to $U'_{i,j+1}$ are constructed. The node corresponding to **aaa** has a T_1 label and is an ancestor of the node corresponding to **aaab** with a T_2 label; hence a corresponding edge to $U_{i+1,j}$ is constructed. The node corresponding to **a** has both a T_1 and a T_2 label; hence a corresponding edge to $(i + 1, j + 1)$ is constructed.