View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

On Indeterminate Strings Matching

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Samah Ghazawi
Department of Computer Science, University of Haifa, Israel

Gad M. Landau
Department of Computer Science, University of Haifa, Israel
Department of Computer Science and Engineering,
NYU Tandon School of Engineering, Brooklyn, NY, USA

Abstract

Given two indeterminate equal-length strings p and t with a set of characters per position in both strings, we obtain a determinate string p_w from p and a determinate string t_w from t by choosing one character per position. Then, we say that p and t match when p_w and t_w match for some choice of the characters. While the most standard notion of a match for determinate strings is that they are simply identical, in certain applications it is more appropriate to use other definitions, with the prime examples being parameterized matching, order-preserving matching, and the recently introduced Cartesian tree matching. We provide a systematic study of the complexity of string matching for indeterminate equal-length strings, for different notions of matching. We use n to denote the length of both strings, and r to be an upper-bound on the number of uncertain characters per position. First, we provide the first polynomial time algorithm for the Cartesian tree version that runs in deterministic $\mathcal{O}(n \log^2 n)$ and expected $\mathcal{O}(n \log n \log \log n)$ time using $\mathcal{O}(n \log n)$ space, for constant r. Second, we establish NP-hardness of the order-preserving version for $r = 2$, thus solving a question explicitly stated by Henriques et al. [CPM 2018], who showed hardness for $r = 3$. Third, we establish NP-hardness of the parameterized version for $r = 2$. As both parameterized and order-preserving indeterminate matching reduce to the standard determinate matching for $r = 1$, this provides a complete classification for these three variants.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases string matching, indeterminate strings, Cartesian trees, order-preserving matching, parameterized matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.14

Funding *Samah Ghazawi*: Partially supported by the Israel Science Foundation (ISF) grant 1475/18. *Gad M. Landau*: Partially supported by the Israel Science Foundation (ISF) grant 1475/18, and the United States-Israel Binational Science Foundation (BSF) grant No. 2018141.

1 Introduction

String matching, in the sense of comparing two equal-length strings, is one of the fundamental problems in computer science with multiple practical applications. While exact matching is trivial to solve in optimal linear time by comparing the strings character-by-character, for many of the applications it seems more appropriate to work with some kind of approximate matching. Prime examples include string matching with swaps [2], parameterized string matching [6], string matching with gaps [9], jumbled string matching [10], string matching with don't cares [29], and edit distance [32]. In all of such problems, one needs to first precisely define when do two strings match.

Parameterized matching is a classical notion motivated by finding identical sections of code [3, 4, 5, 6, 19, 34]. Formally, two strings p and t of length n are a parameterized match when for every $i, j \in \{1, \ldots, n\}$, $p[i] = p[j]$ iff $t[i] = t[j]$. This is denoted by $p \sim_= t$.

© Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau;
licensed under Creative Commons License CC-BY
31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020).
Editors: Inge Li Gørtz and Oren Weimann; Article No. 14; pp. 14:1–14:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Order-preserving matching is a more recent but already well-studied notion motivated by stock price analysis and musical melody matching [11, 16, 17, 25, 26]. Formally, two strings p and t of length n are a order-preserving match when for every $i, j \in \{1, \ldots, n\}$, $p[i] \leq p[j]$ iff $t[i] \leq t[j]$. This is denoted by $p \sim_{\leq} t$.

Very recently, a different notion called Cartesian tree matching has been proposed [28]. The Cartesian tree of a given string p ($CT(p)$), first defined in [31], is constructed according to the following rules:

- If p is an empty string, $CT(p)$ is an empty tree.
- If $p[1...n]$ is not empty and $p[i]$ is the leftmost minimum value in p, $CT(p)$ is the tree with $p[i]$ being the root, $CT(p[1...i-1])$ the left subtree, and $CT(p[i+1...n])$ the right subtree.

Even though the most well-known applications of Cartesian trees are probably in designing space-efficient structures for finding the minimum in a range, they can be also used to compare strings. Similarly to order-preserving matching, this notion is motivated by applications concerned with time-series data such as stock price analysis, and has gained considerable attention during the last year [7, 18, 30]. Formally, two strings p and t of equal length n are a Cartesian tree match when their Cartesian trees $CT(p)$ and $CT(t)$ are identical, i.e. $CT(P)$ and $CT(t)$ have the same shape while the labels on the nodes may differ. This is denoted by $p \sim_C t$.

We consider the complexity of string matching for indeterminate strings defined as follows.

▶ **Definition 1.** *An indeterminate string is a sequence of sets of characters $p[1]p[2]...p[n]$, where $p[i] \subseteq \mathbb{N}$. Each position is specified by writing $p[i] = a_1|...|a_r$, such that $a_\ell \in \mathbb{N}$, which means that we can choose $p[i]$ to be any a_ℓ.*

Indeterminate strings were studied earlier, among others, covering problems for indeterminate strings [1, 14] and indeterminate strings in graph theory [20, 12, 27]. Indeterminate string matching was investigated lately from different angles [8, 13, 15, 23, 22, 24]. It provides a convenient formalism for compactly capturing situations in which there are some uncertainties concerning characters at some positions. Indeed, an indeterminate string p of length n describes r^n determinate strings. We write $\tilde{p}$ to denote the set of all such strings, and p_w when referring to a single determinate string described by p.

First, we consider the complexity of Cartesian tree matching for indeterminate strings defined as follows.

Problem: CARTESIAN TREE MATCHING OF INDETERMINATE STRINGS (CTMIS)
Input: Two indeterminate strings p and t of length n with up to r of uncertain characters per position.
Output: Are there determinate strings $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that p_w Cartesian tree matches t_w?

A naive solution to the CTMIS would be to apply the solution of [28] to each $t_w \in \tilde{t}$ and $p_w \in \tilde{p}$ in $\mathcal{O}(n^2 r^n)$ time. In Section 2 we provide the first polynomial algorithm for this problem that works in $\mathcal{O}(n \log^2 n)$ time and $\mathcal{O}(n \log n)$ space, assuming that r is constant. Additionally, in the Word RAM model of computation we further improve the time complexity to expected $\mathcal{O}(n \log n \log\log n)$.

▶ **Example 2.** Consider the following indeterminate strings:

$$p = (2|4|7, 2|5|6, 1|4|8, 4|7|8, 3|10|16)$$
$$t = (2|7|10, 5|20|31, 10|17|25, 0|9|11, 1|8|18).$$

Figure 1 The Cartesian trees of $p_w = (7, 2, 8, 4, 16)$ and $t_w = (10, 5, 17, 9, 18)$.

$p_w = (7, 2, 8, 4, 16)$ and $t_w = (10, 5, 17, 9, 18)$ define the same Cartesian tree, see Figure 1. Therefore, we say that $p \sim_C t$. Note that p and t define other matching or non-matching Cartesian trees.

Second, we consider the complexity of order-preserving matching for indeterminate strings defined as follows.

Problem: ORDER-PRESERVING MATCHING OF INDETERMINATE STRINGS (OPMIS)
Input: Two indeterminate strings p and t of length n with up to r uncertain characters per position.
Output: Are there determinate strings $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that p_w order-preserving matches t_w?

Henriques et al. [21] proved that OPMIS is NP-hard for $r = 3$. As for $r = 1$ there is a simple linear-time algorithm, this left $r = 2$ as the only open case (CPM version of the paper [21] claims a polynomial time algorithm for this case, but this has been clarified in the arXiv version [13]). In Section 4 we provide a different reduction that establishes NP-hardness of OPMIS already for $r = 2$, thus fully resolving the complexity of this problem and answering an open question explicitly stated by Costa et al [13]. In contrast with the previous work, our reduction exploits the order between elements instead of just their equality, and is more involved.

Third, we consider the complexity of parameterized matching for indeterminate strings defined as follows.

Problem: PARAMETERIZED MATCHING OF INDETERMINATE STRINGS (PMIS)
Input: Two indeterminate strings p and t of length n with up to r uncertain characters per position.
Output: Are there determinate strings $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that p_w parameterized matches t_w?

NP-hardness proof by Henriques et al. [21] implicitly shows hardness of PMIS for $r = 3$. This, again, leaves $r = 2$ as the only open case. In Section 5 we provide a reduction that establishes NP-hardness of PMIS for $r = 2$.

2 CTMIS in $\mathcal{O}(n^3)$ Time and $\mathcal{O}(n^2)$ Space

In this section, we describe a warm-up solution for the CTMIS problem. The input is two equal-length indeterminate strings p and t with two uncertain characters per position, and the output is whether $p \sim_C t$ or not. The solution can be generalized to any constant value of r in a straightforward manner. We will assume that both p and t consists of distinct values, which can be always ensured by an appropriate perturbation.

First, note that for each index i, we have $p[i] = a_i|a'_i$ and $t[i] = b_i|b'_i$, hence each i defines a set consisting 4 pairs $\{(a_i, b_i), (a_i, b'_i), (a'_i, b_i), (a'_i, b'_i)\}$ (called thresholds) denoted by $\textsc{Thresholds}(i)$. The main idea of the algorithm is to determine for each index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$:

1. for which indices k we have $p[k, i] \sim_C t[k, i]$ with the roots x_i and y_i, respectively.
2. for which indices j we have $p[i, j] \sim_C t[i, j]$ with the roots x_i and y_i, respectively.

Consider an interval $[k, i]$, the reasoning for an interval $[i, j]$ is similar. We have $p[k, i] \sim_C t[k, i]$ with the roots x_i and y_i iff there exists an index ℓ and a threshold $(x_\ell, y_\ell) \in \textsc{Thresholds}(\ell)$ where $k \le \ell \le i-1$, $x_i < x_\ell$ and $y_i < y_\ell$ such that $p[k, \ell] \sim_C t[k, \ell]$ and $p[\ell, i-1] \sim_C t[\ell, i-1]$.

We process all possible intervals $[k, i]$ and $[i, j]$ in an increasing order of their lengths using dynamic programming. For each index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$ we compute the answer for all left intervals $[k, i-1]$ and all right intervals $[i+1, j]$, see Figure 2. We define two types of states and associate a boolean value with each of them as follows:

Left states $L_{k,i}(x_i, y_i) = \texttt{true}$ iff $p[k, i] \sim_C t[k, i]$ with the roots x_i and y_i, respectively.
Right states $R_{i,j}(x_i, y_i) = \texttt{true}$ iff $p[i, j] \sim_C t[i, j]$ with the roots x_i and y_i, respectively.

$$p = (a_1|a'_1, \dots, \overbrace{a_k|a'_k, \dots, a_{i-1}|a'_{i-1}}^{\text{left Cartesian subtree } CT(p[k,i-1])}, a_i|a'_i, \overbrace{a_{i+1}|a'_{i+1}, \dots, a_j|a'_j}^{\text{right Cartesian subtree } CT(p[i+1,j])}, \dots, a_n|a'_n)$$

$$t = (b_1|b'_1, \dots, \overbrace{b_k|b'_k, \dots, b_{i-1}|b'_{i-1}}^{\text{left Cartesian subtree } CT(t[k,i-1])}, b_i|b'_i, \overbrace{b_{i+1}|b'_{i+1}, \dots, b_j|b'_j}^{\text{right Cartesian subtree } CT(t[i+1,j])}, \dots, b_n|b'_n)$$

Figure 2 An interval $[k, j]$ of the strings p and t with the root at index i, defining left and right Cartesian subtrees.

▶ **Example 3.** Let $p = (4|7, 2|6, 1|8, 3|20, 10|16)$ and $t = (2|10, 20|31, 10|17, 0|11, 8|18)$. Considering index 3 as a possible root for the Cartesian tree in both strings, the thresholds defined by index 3 are $(x_3, y_3) \in \{(1, 10), (8, 10), (1, 17), (8, 17)\}$. The right states are $R_{3,4}(x_3, y_3)$ and $R_{3,5}(x_3, y_3)$. The left states are $L_{2,3}(x_3, y_3)$ and $L_{1,3}(x_3, y_3)$. Some of their corresponding boolean values are as follows:

1. $R_{3,4}(1, 10) = \texttt{true}$ for $p[3, 4] = (1, 3)$ and $t[3, 4] = (10, 11)$.
2. $R_{3,4}(8, 10) = \texttt{true}$ for $p[3, 4] = (8, 20)$ and $t[3, 4] = (10, 11)$.
3. $L_{2,3}(1, 10) = \texttt{true}$ for $p[2, 3] = (6, 1)$ and $t[2, 3] = (31, 10)$.
4. $L_{1,3}(1, 10) = \texttt{true}$ for $p[1, 3] = (4, 6, 1)$ and $t[2, 3] = (2, 31, 10)$.

From the definition of a Cartesian tree we directly obtain the following proposition illustrated in Figure 3.

▶ **Proposition 4.**

(a) $R_{i,j}(x_i, y_i) = \texttt{true}$ *iff* $\exists \ell \in [i+1, j]$ *such that* $R_{\ell,j}(x_\ell, y_\ell) = \texttt{true}$ *and* $L_{i+1,\ell}(x_\ell, y_\ell) = \texttt{true}$ *where* $x_\ell > x_i$ *and* $y_\ell > y_i$.

(b) $L_{k,i}(x_i, y_i) = \texttt{true}$ *iff* $\exists \ell' \in [k, i-1]$ *such that* $R_{\ell',i-1}(x_{\ell'}, y_{\ell'}) = \texttt{true}$ *and* $L_{k,\ell'}(x_{\ell'}, y_{\ell'}) = \texttt{true}$ *where* $x_{\ell'} > x_i$ *and* $y_{\ell'} > y_i$.

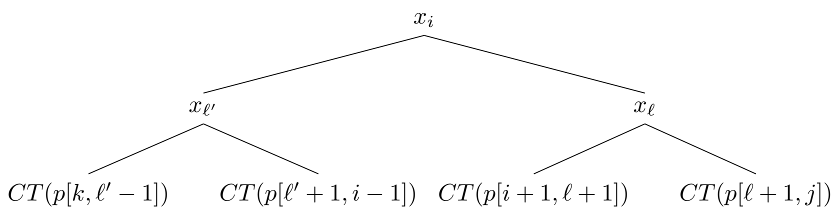

Figure 3 The Cartesian tree $CT(p[k, j])$ with the root at index i. Note that, the Cartesian tree $CT(t[k, j])$ is identical to the Cartesian tree above with the proper values y_i, $y_{\ell'}$ and y_ℓ on the nodes, and with the proper subtrees $CT(t[k, \ell' - 1])$, $CT(t[\ell' + 1, i - 1])$, $CT(t[i + 1, \ell + 1])$ and $CT(t[\ell + 1, j])$. Moreover, in both Cartesian trees, the left Cartesian subtrees correspond to the left state $L_{k,i}(x_i, y_i)$, while the right Cartesian subtrees correspond to the right state $R_{i,j}(x_i, y_i)$.

Recall that we apply dynamic programming in an increasing order of the lengths of the intervals. Therefore, the states $R_{\ell,j}(x_\ell, y_\ell)$ and $L_{i+1,\ell}(x_\ell, y_\ell)$ from Proposition 4(a) are computed before the state $R_{i,j}(x_i, y_i)$. Similarly, the states $R_{\ell',i-1}(x_{\ell'}, y_{\ell'})$ and $L_{k,\ell'}(x_{\ell'}, y_{\ell'})$ are computed before the state $L_{k,i}(x_i, y_i)$. Therefore, for every interval we can simply consider all relevant ℓ and ℓ', access their corresponding states, and update the answer. Finally, after having processed all the intervals, we conclude that $p \sim_C t$ iff there exists an index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$ such that $L_{1,i}(x_i, y_i) = \texttt{true}$ and $R_{i,n}(x_i, y_i) = \texttt{true}$.

▶ **Example 5.** Let $p = (4|7, 2|6, 1|8, 3|20, 10|16)$ and $t = (2|10, 20|31, 10|17, 0|11, 8|18)$ as in the previous example above. We have $L_{1,3}(1, 10) = \texttt{true}$ for $p[1, 3] = (4, 6, 1)$ and $t[2, 3] = (2, 31, 10)$, and $R_{3,5}(1, 10) = \texttt{true}$ for $p[3, 5] = (1, 3, 16)$ and $t[3, 5] = (10, 11, 18)$. Hence, $p \sim_C t$ with the roots 1 and 10 respectively.

Time complexity. For each state $L_{k,i}(x_i, y_i)$ and $R_{i,j}(x_i, y_i)$ we consider $\mathcal{O}(n)$ relevant indices ℓ and ℓ', respectively. Each such index is processed in constant time, thus the overall time complexity is $\mathcal{O}(n^3)$. The space complexity is bounded by the number of states processed in the dynamic programming, which is $\mathcal{O}(n^2)$.

3 CTMIS in $\mathcal{O}(n \log^2 n)$ Time and $\mathcal{O}(n \log n)$ Space

In this section we present an efficient solution for the CTMIS problem that builds on the slower algorithm presented in the previous section.

The input is two equal-length indeterminate strings p and t with 2 uncertain characters per position, and the output is whether $p \sim_C t$, or not. The solution can be generalized to any constant value of r in a straightforward manner. The main idea of the algorithm is to find, for each index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$, the largest matching Cartesian trees with the root in both trees being x_i and y_i at index i, respectively. As in the previous algorithm, we consider each index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$ separately. However, now instead of computing the answer for all intervals $[k, i]$ and $[i, j]$ we use the following definition.

▶ **Definition 6.** *For an index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$:*

- $\min_L(i,x_i,y_i)$ *denotes the smallest index such that* $p[\min_L(i,x_i,y_i), i] \sim_C t[\min_L(i,x_i,y_i), i]$,
- $\max_R(i,x_i,y_i)$ *denotes the largest index such that* $p[i, \max_R(i,x_i,y_i)] \sim_C t[i, \max_R(i,x_i,y_i)]$,

with the root in both trees being x_i and y_i at index i, respectively.

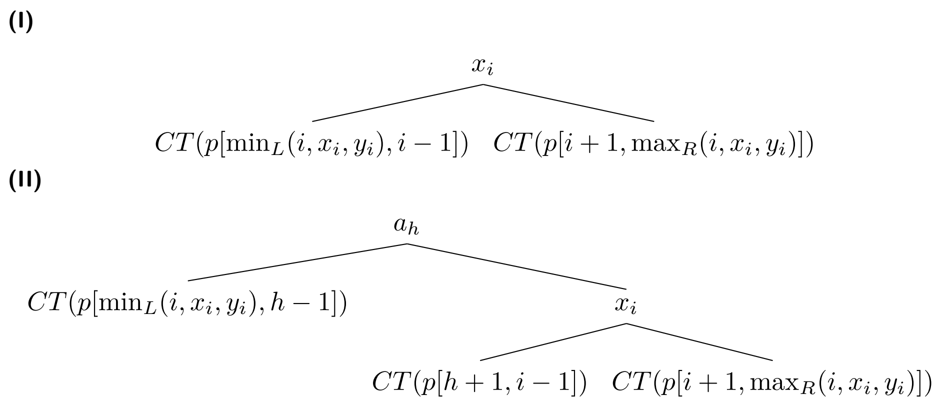

Figure 4 Consider the strings:

$$p = (a_1|a'_1, \ldots, a_{\min_L(i,x_i,y_i)}|a'_{\min_L(i,x_i,y_i)}, \ldots, a_h|a'_h, \ldots, a_i|a'_i, \ldots, a_{\max_R(i,x_i,y_i)}|a'_{\max_R(i,x_i,y_i)}, \ldots, a_n|a'_n)$$
$$t = (b_1|b'_1, \ldots, b_{\min_L(i,x_i,y_i)}|b'_{\min_L(i,x_i,y_i)}, \ldots, b_h|b'_h, \ldots, b_i|b'_i, \ldots, b_{\max_R(i,x_i,y_i)}|b'_{\max_R(i,x_i,y_i)}, \ldots, b_n|b'_n)$$

assuming $a_h < x_i < a'_h$, the figure illustrates (I) the Cartesian tree of the substring $p[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)]$ with x_i as a root when choosing a'_h at index h, and (II) the Cartesian tree of the substring $p[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)]$ with a_h as the root after changing a'_h to a_h at index h. Note that, assuming $b_h < y_i < b'_h$, the Cartesian trees of $t[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)]$ are identical to the Cartesian trees in (I) and (II) above with the proper roots and the proper Cartesian subtrees.

Computing $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ fully describes the situation, as the above definition together with the definition of a Cartesian tree matching directly imply the following:

- $p[\ell, i] \sim_C t[\ell, i]$ iff $\min_L(i, x_i, y_i) \leq \ell \leq i$.
- $p[i, r] \sim_C t[i, r]$ iff $i \leq r \leq \max_R(i, x_i, y_i)$.

We also note that $p[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)] \sim_C t[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i)]$ due to a Cartesian tree with the root in both trees being x_i and y_i at index i, respectively. Consequently, $p \sim_C t$ iff there exists an index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$ such that $\min_L(i, x_i, y_i) = 1$ and $\max_R(i, x_i, y_i) = n$. Thus, in the remaining part of this section we focus on efficiently computing the values of $\min_L$ and $\max_R$.

Our algorithm is based on the following observation. Consider an index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$, and assume that $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ have been already computed. Then, the following holds:

1. for any index $h \in [\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$ and a threshold $(x_h, y_h) \in \textsc{Thresholds}(h)$ such that $x_h < x_i$ and $y_h < y_i$, the index $\max_R(i, x_i, y_i)$ is a potential candidate for $\max_R(h, x_h, y_h)$.
2. for any index $h \in [\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$ and a threshold $(x_h, y_h) \in \textsc{Thresholds}(h)$ such that $x_h < x_i$ and $y_h < y_i$, the index $\min_L(i, x_i, y_i)$ is a potential candidate for $\min_L(h, x_h, y_h)$.

Each index h and a threshold $(x_h, y_h) \in \textsc{Thresholds}(h)$ might be considered for several indices i and thresholds $(x_i, y_i) \in \textsc{Thresholds}(i)$ in the above statement, hence we might have several potential candidates for $\min_L(h, x_h, y_h)$ and $\max_R(h, x_h, y_h)$. By the definition of a Cartesian tree, one of these potential candidates corresponds to the sought $\min_L(h, x_h, y_h)$ and $\max_R(h, x_h, y_h)$ as defined above if they are not equal to h. See Figure 4.

The high-level description of the algorithm is as follows. Please see Algorithm 1 for the pseudocode. We iterate over all indices i and thresholds $(x_i, y_i) \in \textsc{Thresholds}(i)$ in a specific order that will be precisely defined later. For each index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$ we aim to:

Step 1 Compute efficiently the indices $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ (See Definition 6 above).

Step 2 Add for all indices $h \in [\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$ and a threshold $(x_h, y_h) \in \textsc{Thresholds}(h)$ such that $x_h < x_i$ and $y_h < y_i$, the index $\max_R(i, x_i, y_i)$ as a potential candidate $\max_R(h, x_h, y_h)$. Add for all indices $h \in [\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$ and a threshold $(x_h, y_h) \in \textsc{Thresholds}(h)$ such that $x_h < x_i$ and $y_h < y_i$, the index $\min_L(i, x_i, y_i)$ as a potential candidate $\min_L(h, x_h, y_h)$.

We need to ensure that, for any index i and a threshold $(x_i, y_i) \in \textsc{Thresholds}(i)$, and any index h and a threshold $(x_h, y_h) \in \textsc{Thresholds}(h)$ such that $x_h < x_i$ and $y_h < y_i$, $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ are already computed when we are considering threshold $(x_h, y_h) \in \textsc{Thresholds}(h)$. This will be guaranteed by the algorithm as explained below.

The algorithm considers all indices i and thresholds $(x_i, y_i) \in \textsc{Thresholds}(i)$ in the reverse lexicographical order, that is, the decreasing order of x_i and, if there is a tie, the decreasing order of y_i. Before we explain how to implement **Step 1** and **Step 2** efficiently, we define the necessary data structures. We maintain a balanced binary search tree T_y on the values of y_i, and identify y_i with its corresponding node of T_y. In each node u of T_y we have its associated secondary trees $T_{\min}(u)$ and $T_{\max}(u)$. Each $T_{\min}(u)$ and $T_{\max}(u)$ stores a collection of intervals $[\ell, r]$. The update adds a new interval $[\ell, r]$ to the collection. The query in $T_{\min}(u)$ for i finds the smallest ℓ such that $[\ell, r]$ containing i belongs to the collection, while the query in $T_{\max}(u)$ finds the largest r. By symmetry, it is enough to explain how to implement $T_{\min}(u)$. We maintain the following invariant: there are no two intervals $[\ell, r]$ and $[\ell', r']$ such that $[\ell, r] \subseteq [\ell', r']$. Clearly, such $[\ell, r]$ is not an answer to any query. Note that this implies that if we sort all the remaining intervals $[\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_s, r_s]$ so that $\ell_1 < \ell_2 < \ldots < \ell_s$ then we also have $r_1 < r_2 < \ldots < r_s$. This gives us a linear order on the intervals, and so we can maintain them in any balanced binary search tree. After adding the new interval $[\ell, r]$ to the collection, we can check if it is not contained in any of the already existing intervals, and if so find the already existing intervals that should be removed, with standard operations on the balanced binary search tree.

Now we explain how to implement **Step 1** and **Step 2** efficiently using T_y and the secondary structures associated with its nodes. Let i and $(x_i, y_i) \in \textsc{Thresholds}(i)$ be the index and the threshold we are currently considering. We begin our discussion with **Step 2** and therefore assume that we already computed $\min_L(i, x_i, y_i)$ and $\max_R(i, x_i, y_i)$ for this threshold. Note that all thresholds $(x_h, y_h) \in \textsc{Thresholds}(h)$ such that $x_i < x_h$ and $y_i < y_h$ have been already processed. Moreover, all thresholds $(x_c, y_c) \in \textsc{Thresholds}(c)$ such that $x_i < x_c$ have been already processed and will not be considered in the future, so we don't need to be concerned with updating their answer. Hence, in **Step 2** we update all thresholds $(x_h, y_h) \in \textsc{Thresholds}(h)$ such that $y_h < y_i$, regardless of the value of x_h. To this end, we consider every ancestor y_h of y_i such that $y_h < y_i$, plus the node y_i itself, and add the interval $[\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$ or $[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$ to their corresponding $T_{\max}$ and $T_{\min}$, respectively. To implement **Step 1**, we consider every ancestor y_c of y_i such that $y_c > y_i$, plus the node y_i itself, and we query their corresponding $T_{\min}$ and $T_{\max}$. It can be readily verified that by the choice of which ancestors are updated, this is enough to implicitly consider every $y_h > y_i$, as such y_h must have updated one of the ancestors y_c.

Algorithm 1 CTMIS in $\mathcal{O}(n \log^2 n)$ time and $\mathcal{O}(n \log n)$ space.

Data: indeterminate length-n strings p and t with 2 uncertain characters per position.
Output: Does $p \sim_C t$ hold?

THRESHOLDS $\leftarrow \{(x_i, y_i) \mid x_i \in p[i]$ and $y_i \in t[i]$ for some $i = 1, 2, \ldots, n\}$
Build a balanced binary search tree T_y on the values of y_i
foreach node $u \in T_y$ **do**
  Create secondary balanced search trees $T_{\min}(u)$ and $T_{\max}(u)$ of intervals $[\ell, r]$
  ▷ `$T_{\min}(u)$ is ordered by $\ell$ while $T_{\max}(u)$ is ordered by $r$`
foreach $(x_i, y_i) \in$ THRESHOLDS by decreasing order of x_i **do**
  Let $u \in T_y$ be the node satisfying VALUE$(u) = y_i$
  ▷ VALUE(u) returns the corresponding y_i of a node $u \in T_y$.
  $\min_L(i, x_i, y_i) \leftarrow i$
  $\max_R(i, x_i, y_i) \leftarrow i$
  ▷ Step 1: Query the potential candidates structures.
  foreach v an ancestor of u in T_y **do** ▷ including u itself
    if VALUE$(v) > y_i$ **then**
      if $\min\{\ell \mid [\ell, r] \in T_{\min}(v)$ and $i \in [\ell, r]\} < \min_L(i, x_i, y_i)$ **then**
        $\min_L(i, x_i, y_i) \leftarrow \min\{\ell \mid [\ell, r] \in T_{\min}(v)$ and $i \in [\ell, r]\}$
      if $\max\{r \mid [\ell, r] \in T_{\max}(v)$ and $i \in [\ell, r]\} > \max_R(i, x_i, y_i)$ **then**
        $\max_R(i, x_i, y_i) \leftarrow \max\{r \mid [\ell, r] \in T_{\max}(v)$ and $i \in [\ell, r]\}$
  if $\min_L(i, x_i, y_i) = 1$ **and** $\max_R(i, x_i, y_i) = n$ **then**
    return true
  ▷ Step 2: Update the potential candidates structures.
  foreach v an ancestor of u in T_y **do** ▷ including u itself
    if VALUE$(v) < y_i$ **then**
      if $[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1] \nsubseteq [\ell, r]$ for all $[\ell, r] \in T_{\min}(v)$ **then**
        Add $[\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$ to $T_{\min}(v)$
        Remove from $T_{\min}(v)$ every $[\ell', r'] \subseteq [\min_L(i, x_i, y_i), \max_R(i, x_i, y_i) + 1]$
      if $[\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)] \nsubseteq [\ell, r]$ for all $[\ell, r] \in T_{\max}(v)$ **then**
        Add $[\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$ to $T_{\max}(v)$
        Remove from $T_{\max}(v)$ every $[\ell', r'] \subseteq [\min_L(i, x_i, y_i) - 1, \max_R(i, x_i, y_i)]$
return false

Time complexity. The time complexity of the algorithm is $\mathcal{O}(n \log^2 n)$. First, we need to sort the $4n$ thresholds in $\mathcal{O}(n \log n)$ time. Each of these thresholds is processed by considering $\mathcal{O}(\log n)$ nodes of T_y. At each of these nodes u we spend $\mathcal{O}(\log n)$ amortized time to update and query $T_{\min}(u)$ and $T_{\max}(u)$. Furthermore, the space complexity is $\mathcal{O}(n \log n)$, because each interval appears in $\mathcal{O}(\log n)$ secondary structures. Instead of using balanced binary search trees with $\mathcal{O}(\log n)$ query and update time for the secondary structures, we can plug in any predecessor structure that stores a collection of s integers from $\{1, 2, \ldots, n\}$ in $\mathcal{O}(s)$ space with expected $\mathcal{O}(\log \log n)$ query and update time [33].

4 Order-Preserving Matching of Indeterminate Strings

Given two indeterminate strings p and t of equal-length n with at most 2 uncertain characters per position, we want to check if there exist $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that $p_w \sim_{\leq} t_w$. The goal of this section is to prove that this is NP-hard by reducing checking satisfiability of a 3-CNF formula.

We start with rephrasing the question in a graph-theoretical language. Let Σ_p and Σ_t be the sets of characters that occur in p and t, respectively. We consider a complete undirected bipartite graph G with Σ_p corresponding to the nodes on the one side and Σ_t corresponding to the nodes on the other side. We claim that there exist $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that $p_w \sim_{\leq} t_w$ iff there exists a non-crossing matching M in G, where non-crossing means that we cannot have two edges $(x, y), (x', y')$ such that $x < x'$ but $y' < y$, such that the following holds for every position $i = 1, 2, \ldots, n$:

- $p[i] = x$ **and** $t[i] = y$: $(x, y) \in M$,
- $p[i] = x_1|x_2$ **and** $t[i] = y$: $(x_1, y) \in M$ or $(x_2, y) \in M$
- $p[i] = x$ **and** $t[i] = y_1|y_2$: $(x, y_1) \in M$ or $(x, y_2) \in M$,
- $p[i] = x_1|x_2$ **and** $t[i] = y_1|y_2$: $(x_1, y_1) \in M$ or $(x_1, y_2) \in M$ or $(x_2, y_1) \in M$ or $(x_2, y_2) \in M$.

The proof is straightforward.

We consider a 3-CNF formula ϕ on n variables $1, 2, \ldots, n$ and m clauses. We reduce checking satisfiability of ϕ to finding a non-crossing matching M with some additional constraints in a complete undirected bipartite graph G. Each constraint is of the form $M \cap X \times Y \neq \emptyset$, for some subsets of the nodes X and Y such that $|X|, |Y| \leq 2$, or $X \times Y$ for short. As long as the size of G and the number of constraints is polynomial, this will establish NP-hardness of our problem, as we can create two strings p and t and encode each constraint by setting up some $p[i]$ and $t[i]$ appropriately.

We start with creating nodes $1, 2, \ldots, n$ on the left side and $1, 2, 3, 4, \ldots, 2n$ on the right side of G. We add a constraint $\{i\} \times \{2i-1, 2i\}$, for every $i = 1, 2, \ldots, n$. We add a constraint $\{2n+1\} \times \{2n+1\}$. For every $k = 1, 2, \ldots, m$, we consider the k-th clause $(\ell_{k,1} \vee \ell_{k,2} \vee \ell_{k,3})$, where $\ell_{k,1}, \ell_{k,2}, \ell_{k,3}$ are literals. Let $s = 2n + 2 + 5(k-1)$. We add the following constraints: $\{s\} \times \{s, s+1\}$, $\{s+2, s+3\} \times \{s+3\}$, $\{s, s+2\} \times \{s+1, s+3\}$. This is illustrated in Figure 5. Then we add a constraint for every literal:

1. If $\ell_{k,1} = x$ then we add $\{x, s\} \times \{2x, s\}$, and if $\ell_{k,1} = \neg x$ then we add $\{x, s\} \times \{2x-1, s\}$.
2. If $\ell_{k,2} = y$ then we add $\{y, s+1\} \times \{2y, s+2\}$, and if $\ell_{k,2} = \neg y$ then we add $\{x, s+1\} \times \{2y-1, s+2\}$.
3. If $\ell_{k,3} = z$ then we add $\{z, s+3\} \times \{2z, s+3\}$, and if $\ell_{k,3} = \neg z$ then we add $\{z, s+3\} \times \{2z-1, s+3\}$.

Due to the constraint $\{2n+1\} \times \{2n+1\}$, a variable constraint $\{v, a\} \times \{2v-1, b\}$ translates into $(v, 2v-1) \in M$ or $(a, b) \in M$. Similarly, $\{v, a\} \times \{2v, b\}$ translates into $(v, 2v) \in M$ or $(a, b) \in M$.

We need to prove that ϕ is satisfiable iff there exists a non-crossing matching M in G that respects all the constraints.

First, assume that ϕ is satisfiable and fix a satisfying valuation of all the variables. We obtain M by first adding $(v, 2v-1)$ or $(v, 2v)$ to M depending on whether v is set to false or true, respectively. We also add $(2n+1, 2n+1)$ to M. Then, we proceed as follows for the k-th clause. For concreteness assume that the clause is $(x \vee y \vee z)$, the argument is symmetric for the other cases. If x is set to false then we add (s, s) to M. If y is set to false then we add $(s+1, s+2)$ to M. Finally, if z is set to false then we add $(s+3, s+3)$ to M.

Because at least one of x, y, z is set to true, at least one of these three edges is not in M. If $(s+1, s+2) \notin M$ then we add $(s+2, s+1)$ to M. If $(s, s) \notin M$ then we add $(s, s+1)$ to M. Finally, if $(s+3, s+3) \notin M$ then we add $(s+2, s+3)$ to M. In all cases, the constraints corresponding to the k-clause are fulfilled. Due to how we compose the gadgets, M being a non-crossing matching in every gadget implies that M is a non-crossing matching in the whole G.

Second, assume that we have a non-crossing matching M in G. For every $v = 1, 2, \ldots, n$, M contains exactly one of the edges $(v, 2v-1), (v, 2v)$. We set v to false if $(v, 2v-1) \in M$ and to true if $(v, 2v) \in M$. We must have $(2n+1, 2n+1) \in M$. We need to verify that every clause is satisfied by the obtain valuation of the variables. Again, for concreteness assume that the clause is $(x \vee y \vee z)$. We cannot have all edges $(s, s), (s+1, s+2), (s+3, s+3)$ in M, as in such case the constraint $\{s, s+2\} \times \{s+1, s+3\}$ cannot be fulfilled. If $(s, s) \notin M$ then due to the constraint $\{x, s\} \times \{2x, s\}$ we must have $(x, 2x) \in M$, so x is set to true. If $(s+1, s+2) \notin M$ then due to the constraint $\{y, s+1\} \times \{2y, s+2\}$ we must have $(y, 2y) \in M$, so y is set to true. Finally, if $(s+3, s+3) \notin M$ then due to the constraint $\{z, s+3\} \times \{2z, s+3\}$ we must have $(z, 2z) \in M$, so y is set to true. So, one of the variable x, y, z is set to true, making the clause satisfied.

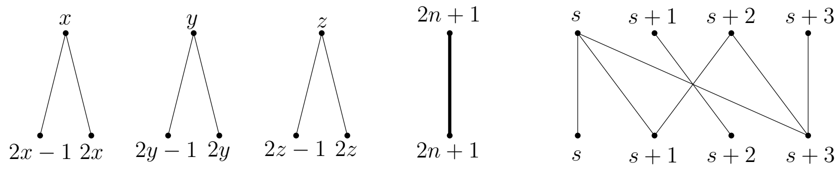

Figure 5 Gadget created for the k-th clause concerning variables x, y, z.

5 Parameterized Matching of Indeterminate Strings

Given two indeterminate strings p and t of equal-length n with at most 2 uncertain characters per position, we want to check if there exist $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that $p_w \sim_= t_w$. The goal of this section is to prove that this is NP-hard by reducing checking if a given undirected graph has a vertex cover consisting of at most k vertices.

As in the previous section, we start with rephrasing the question in a graph-theoretical language. Let Σ_p and Σ_t be the sets of characters that occur in p and t, respectively. We consider a complete undirected bipartite graph G with Σ_p corresponding to the nodes on the one side and Σ_t corresponding to the nodes on the other side. We claim that there exist $p_w \in \tilde{p}$ and $t_w \in \tilde{t}$ such that $p_w \sim_= t_w$ iff there exists a matching M in G, such that the following holds for every position $i = 1, 2, \ldots, n$:

- $p[i] = x$ **and** $t[i] = y$: $(x, y) \in M$,
- $p[i] = x_1 | x_2$ **and** $t[i] = y$: $(x_1, y) \in M$ or $(x_2, y) \in M$
- $p[i] = x$ **and** $t[i] = y_1 | y_2$: $(x, y_1) \in M$ or $(x, y_2) \in M$,
- $p[i] = x_1 | x_2$ **and** $t[i] = y_1 | y_2$: $(x_1, y_1) \in M$ or $(x_1, y_2) \in M$ or $(x_2, y_1) \in M$ or $(x_2, y_2) \in M$.

The proof is straightforward.

We consider an undirected graph H on n vertices $V = \{1, 2, \ldots, n\}$ and m edges E together with a parameter $k \leq n$. We reduce checking if there is a subset S of k vertices of H such that for every edge $(u, v) \in E$ we have $u \in S$ or $v \in S$ to finding a matching M in a complete undirected bipartite graph G that respects a number of constraints of the form $M \cap X \times Y \neq \emptyset$, for $|X|, |Y| \leq 2$, or $X \times Y$ for short. As long as the size of G and the number of constraints is polynomial, this will establish NP-hardness of our problem, as we can create two strings p and t and encode each constraint by setting up some $p[i]$ and $t[i]$ appropriately.

We start with creating nodes $1, 2, \ldots, n$ on the left side and $1, 2, \ldots, n$ on the right side of G. We add a constraint $\{u, v\} \times \{u, v\}$ for every $(u, v) \in E$. For every $i = 1, 2, \ldots, n$, $(i, j) \in M$ for some $j \in \{1, 2, \ldots, n\}$ corresponds to including i in the sought vertex cover. The remaining part of H is constructed as to guarantee that there are at least k nodes $i \in \{1, 2, \ldots, n\}$ such that $(i, j) \in M$ for some $j \neq \{1, 2, \ldots, n\}$. To this end, we design a gadget G_{2s} with the following property:

1. there are distinguished $2s$ nodes $v_1, v_2, \ldots, v_{2s}$ on the left side, each v_i is incident to a unique edge e_i,
2. there are also some additional internal nodes on the left and on the right and some constraints that concern both the internal and the distinguished nodes,
3. if none of the edges e_i belongs to M then it is not possible to satisfy the constraints of G_{2s},
4. for any nonempty subset S of distinguished nodes, it is possible to select some of the edges with both endpoints being internal nodes in such a way that, together with the edges e_i for $i \in S$, they satisfy all constraints of G_{2s}.

We will first show that G_4 exists, and then explain how to obtain $G_{2(s+1)}$ from G_{2s}.

▶ **Lemma 7.** *G_4 with the sought properties exists.*

Proof. G_4 consists of nodes v_1, v_2, v_3, v_4 and internal nodes v'_1, v'_2, v'_3, v'_4 and x, y, z. We set $e_i = (v_i, v'_i)$ for $i = 1, 2, 3, 4$ and create the following constraints: $\{v_1, x\} \times \{v'_1, y\}$, $\{v_2, x\} \times \{v'_2, y\}$, $\{v_3, z\} \times \{v'_3, y\}$, $\{v_4, z\} \times \{v'_4, y\}$ and $\{y\} \times \{x, z\}$. See Figure 6.

Assume that none of the edges e_i belongs to M. By symmetry, we can assume that $(x, y) \in M$. But then we must have $(v'_3, z), (v'_4, z) \in M$, which is impossible.

Let S be a nonempty set of distinguished nodes. By symmetry, we can assume that $v_1 \in S$. Then, we include $(y, z) \in M$ and if $e_2 \notin S$ we also include $(v'_2, x) \in M$. ◀

▶ **Lemma 8.** *$G_{2(s+1)}$ with the sought properties can be obtained in polynomial time from G_{2s} with the sought properties.*

Proof. We take a copy of G_{2s}, let its distinguished nodes and their corresponding edges be $v_1, v_2, \ldots, v_{2s}$ and $e_1, e_2, \ldots, e_{2s}$. We also take a copy of G_4, let its distinguished nodes and their corresponding edges be u_1, u_2, u_3, u_4 and f_1, f_2, f_3, f_4. To obtain $G_{2(s+1)}$ we identify v_{2s} with u_1 and add a constraint that enforces including e_{2s} or f_1. in M. The distinguished nodes and their corresponding edges of $G_{2(s+1)}$ are $v_1, v_2, \ldots, v_{2s-1}, u_2, u_3, u_4$ and $e_1, e_2, \ldots, e_{2s-1}, f_2, f_3, f_4$.

Assume that none of the edges $e_1, e_2, \ldots, e_{2s-1}, f_2, f_3, f_4$ belongs to M. Either $e_{2s} \notin M$ and we obtain that none of the edges $e_1, e_2, \ldots, e_{2s}$ belongs to M, or $f_1 \notin M$ and none of the edges f_1, f_2, f_3, f_4 belongs to M. In either case we obtain a contradiction by the construction of G_{2s} or G_4.

Let S be a nonempty set of distinguished nodes and assume that $v_1 \in S$ (other cases are essentially the same). We set $S' = S \cap \{v_1, v_2, \ldots, v_{2s-1}\}$ and $S'' = (S \cap \{u_2, u_3, u_4\}) \cup \{u_1\}$. Then $S', S'' \neq \emptyset$, and by assumption we can select some of the edges with both endpoints being internal nodes of G_{2s} or G_4 in such a way that, together with the edges e_i for $i \in S'$ and f_j for $j \in S''$, they satisfy all constraints of G_{2s} and G_4. Additionally, the constraint that enforces including e_{2s} or f_1 is satisfied by taking f_1. So, by selecting the edges with both endpoints being internal nodes of G_{2s} or G_4 together with f_1 we obtain a set of edges with both endpoints being internal nodes of $G_{2(s+1)}$ that, together with the edges associated with the nodes in S, satisfy all constraints of $G_{2(s+1)}$ as required. ◀

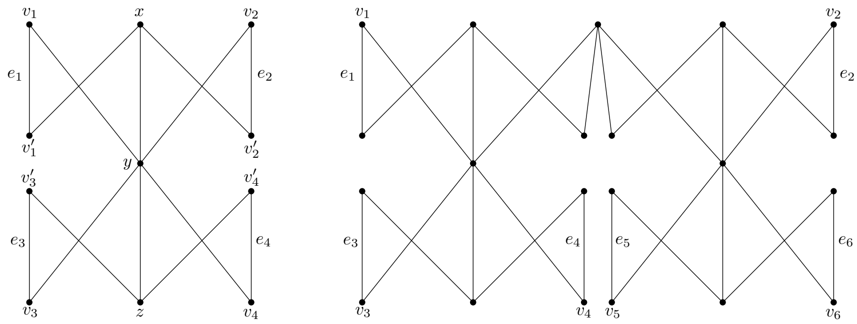

Figure 6 Gadgets G_4 (left) and G_8 (right).

With the gadget G_{2s} in hand, we are ready to complete the reduction. By duplicating the graph H we can assume that $n = 2s$. We add $n - k$ copies of the gadget G_{2s} to G. Let $v_1, v_2, \dots, v_{2s}$ be the distinguished nodes of one such copy. We identify v_i with the node i on the left side of G. This guarantees that for each gadget we must have a unique node i such that $(i,1),(i,2),\dots,(i,n) \notin M$. We claim that the resulting graph G has a matching that satisfies all the constrains if and only if H admits a vertex cover of cardinality at most k. In one direction, consider the set C consisting of all nodes $i \in \{1,2,\dots,n\}$ such that $(i,1),(i,2),\dots,(i,n) \notin M$. By the properties of G_{2s}, $|C| \le k$. We need to argue that C is a vertex cover. Consider any $(u,v) \in E$. Due to the constraint $\{u,v\} \times \{u,v\}$, one of the edges $(u,u),(u,v),(v,u),(v,v)$ must belong to M. But then either u or v cannot be matched to any node not belonging to $\{1,2,\dots,n\}$, so $u \in C$ or $v \in C$ as required. In other direction, let C be a vertex cover of H of cardinality at most k. For every $i \in C$, we include the edge (i,i) in M. This clearly satisfies every constraint $\{u,v\} \times \{u,v\}$ by C being a vertex cover. Then, for every copy of G_{2s} we choose a unique node $i \notin C$ (that is not matched to any other node yet) and use the properties of G_{2s} to add its internal edges to M in such a way that, together with the edge associated to i, they satisfy all the constraints.

References

1 A. Alatabbi, A. S. M. Sohidull Islam, M. S. Rahman, R. J. Simpson, and W. F. Smyth. Enhanced covers of regular & indeterminate strings using prefix tables. *Automata, Languages and Combinatorics*, 21(3):131–147, 2016.

2 A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein. Pattern matching with swaps. *Journal of Algorithms*, 37(2):247–266, 2000.

3 A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, 1994.

4 A. Apostolico, Péter L. Erdös, and M. Lewenstein. Parameterized matching with mismatches. *Discrete Algorithms*, 5(1):135–140, 2007.

5 B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *25th STOC*, pages 71–80, 1993.

6 B. S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.

7 M. Bataa, S. G. Park, A. Amir, G. M. Landau, and K. Park. Finding periods in cartesian tree matching. In *30th IWOCA*, volume 11638, pages 70–84, 2019.

8 G. Bernardini, P. Gawrychowski, N. Pisanti, S. P. Pissis, and G. Rosone. Even faster elastic-degenerate string matching via fast matrix multiplication. In *46th ICALP*, pages 21:1–21:15, 2019.

9 P. Bille, I. Li Gørtz, H. W. Vildhøj, and D. K. Wind. String matching with variable length gaps. *Theoretical Computer Science*, 443:385–394, October 2010.

10 P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták. Algorithms for jumbled pattern matching in strings. *International Journal of Foundations of Computer Science*, 23(02):357–374, 2012.

11 S. Cho, J. C. Na, K. Park, and J. S. Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397–402, 2015.

12 M. Christodoulakis, P. J. Ryan, W. F. Smyth, and S. Wang. Indeterminate strings, prefix arrays & undirected graphs. *Theoretical Computer Science*, 600:34–48, 2015.

13 D. Costa, L. M. S. Russo, R. Henriques, H. Bannai, and A. P. Francisco. Order-preserving pattern matching indeterminate strings. In *30th CPM*, 2019. `arXiv:1905.02589`.

14 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń. Covering problems for partial words and for indeterminate strings. In *25th ISAAC*, pages 220–232, 2014.

15 J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, É. Prieur-Gaston, and B. Watson. Efficient pattern matching in degenerate strings with the burrows-wheeler transform. *Information Processing Letters*, 147, 2017.

16 P. Gawrychowski and P. Uznański. Order-preserving pattern matching with k mismatches. *Theoretical Computer Science*, 638:136–144, 2016.

17 G. Gourdel, T. Kociumaka, J. Radoszewski, W. Rytter, A. Shur, and T. Waleń. String periods in the order-preserving model. In *35th STACS*, volume 96, pages 1–16, 2018.

18 G. Gu, S. Song, S. Faro, T. Lecroq, and K. Park. Fast multiple pattern cartesian tree matching. In *14th WALCOM*, pages 107–119, 2020.

19 C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. *ACM Transactions on Algorithms (TALG)*, 3(3):29–44, 2007.

20 J. Helling, P. J. Ryan, W. F. Smyth, and M. Soltys. Constructing an indeterminate string from its associated graph. *Theoretical Computer Science*, 710, March 2017.

21 Rui Henriques, Alexandre P. Francisco, Luís M. S. Russo, and Hideo Bannai. Order-preserving pattern matching indeterminate strings. In *29th CPM*, volume 105, pages 2:1–2:15, 2018.

22 J. Holub and W. F. Smyth. Algorithms on indeterminate strings. In *14th AWOCA*, pages 36–45, 2003.

23 J. Holub, W. F. Smyth, and S. Wang. Fast pattern-matching on indeterminate strings. *Discrete Algorithms*, 6(1):37–50, 2008.

24 C. Iliopoulos, R. Kundu, and S. Pissis. Efficient pattern matching in elastic-degenerate strings. In *11th LATA*, pages 131–142, 2017.

25 J. Kim, P. Eades, R. Fleischer, S. H. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.

26 M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.

27 R. McIntyre and M. Soltys. An improved upper bound and algorithm for clique covers. *J. Discrete Algorithms*, 48:42–56, 2018.

28 S. G. Park, A. Amir, G. M. Landau, and K. Park. Cartesian tree matching and indexing. In *30th CPM*, pages 16:1–16:14, 2019.

29 M. S. Rahman and C. S. Iliopoulos. Pattern matching algorithms with don't cares. In *33th SOFSEM*, pages 116–126, 2007.

30 S. Song, C. Ryu, S. Faro, T. Lecroq, and K. Park. Fast cartesian tree matching. In *26th SPIRE*, pages 124–137, 2019.

31 J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

32 R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

33 D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

34 B. Zeidman. Software v. software. *IEEE Spectrum*, 47:32–53, 2010.