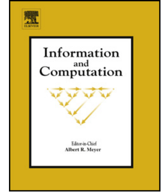
Information and Computation 289 (2022) 104924

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco



# Order-preserving pattern matching indeterminate strings

Luís M.S. Russo [a,*], Diogo Costa [a], Rui Henriques [a], Hideo Bannai [b], Alexandre P. Francisco [a]

[a] INESC-ID and Instituto Superior Técnico, Universidade de Lisboa, Portugal
[b] Department of Computer Science, Kyushu University, Japan

ARTICLE INFO

*Article history:*
Received 26 June 2020
Received in revised form 4 October 2021
Accepted 25 May 2022
Available online 30 May 2022

*Keywords:*
Order-preserving pattern matching
Indeterminate string analysis
Generic pattern matching
Satisfiability

ABSTRACT

Given a pattern $p$ of size $m$ and a text $t$, the problem of order-preserving pattern matching (OPPM) is to find all substrings of $t$ that satisfy one of the orderings defined by $p$. This problem has applications on time series analysis. However given its strict nature this model is unable to deal with indetermination, thus limiting its application to noisy time series. In this paper we introduce indeterminate characters to alleviate this limitation. We then propose two polynomial time algorithms. If the indetermination is limited to $p$ confirming one occurrence can be computed in $O(rm \lg r)$ time, where $r$ is a bound on the number of uncertain characters per position. If the indetermination alternates, but does not occur at the same position in $t$ and $p$, we present an algorithm that requires $O(rm(m + \log r))$ time. We also show that the general problem is NP-hard and provide a polynomial size boolean formula.

© 2022 Elsevier Inc. All rights reserved.

## 1. Introduction

Given a pattern string $p$ and a text string $t$, the exact order preserving pattern matching (OPPM) problem is to find all substrings of $t$ with the same relative orders as $p$. The problem is applicable to strings with characters drawn from numeric or ordinal alphabets. Illustrating (with 0-based indexes), given $p$=(1,5,3,3) and $t = (5, 1, 4, 2, 2, 5, 2, 4)$, substring $t[1..4] = (1, 4, 2, 2)$ is reported since it satisfies the character orders in $p$, $p[0] \leq p[2] = p[3] \leq p[1]$. Despite its relevance, the OPPM problem has limited potential since it prevents the specification of errors, uncertainties or don't care characters within the text.

Indeterminate strings allow uncertainties between two or more characters per position. Given indeterminate strings $p$ and $t$, the problem of indeterminate order preserving pattern matching ($\mu$OPPM) is to find all substrings of $t$ with an assignment of values that satisfy the orders defined by $p$. For instance, let $p = (1, 2|5, 3, 3)$ and $t = (5, 0, 1, 2|1, 2, 5, 2|3, 3|4)$. The substrings $t[1..4]$ and $t[4..7]$ are reported since there is an assignment of values that preserve either $p[0] < p[1] < p[2] = p[3]$ or $p[0] < p[2] = p[3] < p[1]$ orderings: respectively $t[1..4] = (0, 1, 2, 2)$ and $t[4..7] = (2, 5, 3, 3)$.

Order-preserving pattern matching captures the structural isomorphism of strings, therefore having a wide-range of relevant applications in the analysis of financial times series, musical sheets, physiological signals and biological sequences [1–3]. Uncertainties often occur in these domains. In this context, although the OPPM problem is already a generalisation of the traditional pattern matching problem, the need to further handle localised errors is essential to deal with noisy strings

* Corresponding author.
*E-mail address:* luis.russo@tecnico.ulisboa.pt (L.M.S. Russo).

https://doi.org/10.1016/j.ic.2022.104924
0890-5401/© 2022 Elsevier Inc. All rights reserved.

[4]. For instance, given the stochasticity of gene regulation (or markets), the discovery of order-preserving patterns in gene expression (or financial) time series needs to account for uncertainties [5,6]. Numerical indexes of amino-acids (representing physiochemical and biochemical properties) are subjected to errors, which makes the analysis of protein sequences difficult [7]. Another example are ordinal strings obtained from the discretization of numerical strings, often having two uncertain characters in positions where the original values are near a discretization boundary [4].

Let $m$ and $n$ be the length of the pattern $p$ and text $t$, respectively. The exact OPPM problem has a linear solution on the text length $O(n + m \lg m)$ based on the Knuth-Morris-Pratt algorithm [8,2,9]. Alternative algorithms for the OPPM problem have also been proposed [10–12]. Contrasting with the large attention given to the resolution of the OPPM problem, there are no polynomial-time algorithms to solve the $\mu$OPPM problem, as far as we know. Naive algorithms for $\mu$OPPM assess all possible pattern and text assignments, bounded by $O(nr^m)$ when considering up to $r$ uncertain characters per position.

This work discusses the first polynomial time algorithms able to answer restricted instances of the $\mu$OPPM problem. Accordingly, the contributions are organised as follows:

- In Section 3 we show that finding an order-preserving match between an indeterminate string of length $m$ and a determinate string with the same length, if it exists, is possible in $O(rm \lg r)$ time based on their monotonic properties. We also show that an $O(rm(m + \log r))$ time algorithm exists when the indeterminate positions in $p$ and $t$ do not coincide. Both these algorithms are extended to the general case where $t$ is larger than $p$, by using filtration techniques.
- In Section 4 we show that the general $\mu$OPPM problem is **NP**-hard as soon as we allow for 3 indeterminacies to occur simultaneously in $p$ and $t$. In this Section we also provide a polynomial size boolean formula that encodes a general $\mu$OPPM instance, thus allowing this problem to be handled by SAT-solvers.

A preliminary version of this work was presented at the Annual Symposium on Combinatorial Pattern Matching (CPM) [13]. In this paper, we present significant new developments on those initial results.

## 2. Background

Let $\Sigma$ be a totally ordered alphabet and an element of $\Sigma^*$ be a string. The length of a string $w$ is denoted by $|w|$. The empty string $\varepsilon$ is a string of length 0. For a string $w = xyz$, $x$, $y$ and $z$ are called a prefix, substring, and suffix of $w$, respectively. The $i$-th character of a string $w$ is denoted by $w[i]$ for each $0 \le i < |w|$. For a string $w$ and integers $0 \le i \le j < |w|$, $w[i..j]$ denotes the substring of $w$ from position $i$ to position $j$. For convenience, let $w[i..j] = \varepsilon$ when $i > j$.

Given strings $x$ and $y$ with equal length $m$, $y$ is said to order-preserving match $x$ [8], denoted by $x \approx y$, if the order relation between the characters of $x$ and $y$ is the same, i.e., $x[i] \diamond x[j] \Leftrightarrow y[i] \diamond y[j]$ for any $0 \le i, j < m$, and any relation $\diamond \in \{<, =, >\}$. A non-empty pattern string $p$ is said to order-preserving match (*op-match* in short) a non-empty text string $t$ if and only if there is a position $i$ in $t$ such that $p \approx t[i - |p| + 1..i]$. The *order-preserving pattern matching* (OPPM) problem is to find all such text positions.

### 2.1. The problem

Given a totally ordered alphabet $\Sigma$, an indeterminate string is a sequence of disjunctive sets of characters $x[0]x[1]..x[n-1]$ where $x[i] \subseteq \Sigma$. Each position is given by $x[i] = \{\sigma_1, \ldots, \sigma_r\}$ where $r \ge 1$ and $\sigma_i \in \Sigma$.

Given an indeterminate string $x$, a *valid assignment* $\$x$ is a (determinate) string with a single character at position $i$, denoted $\$x[i]$, contained in the $x[i]$ set of characters, i.e. $\$x[0] \in x[0], \ldots, \$x[m-1] \in x[m-1]$. For instance, the indeterminate string $(1|3, 3|4, 2|3, 1|2)$ has $2^4$ valid assignments.

Given a determinate string $x$ of length $m$, an indeterminate string $y$ of equal length is said to be *order-preserving* against $x$, identically denoted by $x \approx y$, if there is a valid assignment $\$y$ such that the relative orders of the characters in $x$ and $\$y$ are the same, i.e., $x[i] \diamond x[j] \Leftrightarrow \$y[i] \diamond \$y[j]$ for any $0 \le i, j < m$ and any relation $\diamond \in \{<, =, >\}$. Given two indeterminate strings $x$ and $y$, with length $m$, $y$ preserves the orders of $x$, $x \approx y$, if there exists $\$y$ in $y$ that respects the orders of an assignment $\$x$ in $x$.

A non-empty indeterminate pattern string $p$ is said to order-preserving match (*op-match* in short) a non-empty indeterminate text string $t$ if and only if there is a position $i$ in $t$ such that $p \approx t[i - |p| + 1..i]$. The *order-preserving pattern matching with character uncertainties* ($\mu$OPPM) problem is to find all such text positions.

To understand the complexity of the $\mu$OPPM problem, let us consider a solution that is a naive application of state-of-the-art OPPM techniques. In particular we may use the algorithm by Kubica et al. [8], which requires $O(n+q)$ time, where $q = m$ for alphabets of size $m^{O(1)}$ ($q = m \lg m$ otherwise). Given a determinate string $x$ of length $m$, an integer $i$ ($0 \le i < m$) is said in the context of this work to be an *order-preserving border* of $x$ if $x[0..i-1] \approx x[m-i..m-1]$. In this context, given a pattern string $p$, the orders between the characters of $p$ are used to linearly infer the order borders. The order borders can then be used within the Knuth-Morris-Pratt algorithm to find op-matches against a text string $t$ in linear time [8].

Given an indeterminate string $p$ of length $m$ and a determinate string $t$ of length $n$, we consider all the possible assignments of $p$. Note that there may be $O(r^m)$ such assignments. For each such assignment we can apply the algorithm by Kubica et al., thus yielding an $O((n + q)r^m)$ time solution. This complexity is further increased when indetermination is also considered in the text, thus stressing the need for more efficient solutions.

### 2.2. Related work

The exact OPPM problem is well-studied in the literature. Kubica et al. [8], Kim et al. [2] and Cho et al. [9] presented linear time solutions on the text length by combining order-borders, rank-based prefixes and grammars with the Knuth–Morris–Pratt (KMP) algorithm [14]. Cho et al. [10], Belazzougui et al. [11], and Chhabra et al. [12] presented $O(nm)$ algorithms that show a sublinear average complexity by either combining bad character heuristics with the Boyer–Moore algorithm [15] or applying filtration strategies. Recently, Chhabra et al. [16] proposed further principles to solve OPPM using word-size packed string matching instructions to enhance efficiency.

In the context of numeric strings, multiple relaxations to the exact pattern matching problem have been pursued to guarantee that approximate matches are retrieved. In norm matching [17–20], matches between numeric strings occur if a given distance threshold $f(x, y) \leq \theta$ is satisfied. In $(\delta,\gamma)$-matching [21–27], characters is at most $\delta$ and the sum of differences is at most $\gamma$.

In the context of nominal strings, variants of the pattern matching task have also been extensively studied to allow for don't care symbols in the pattern [28–30], transposition-invariant [25], parameterised matching [31,32], less than matching [33], swapped matching [34,35], gaps [36–38], overlap matching [39], and function matching [40,41].

Despite the relevance of the aforementioned contributions to answer the exact order-preserving pattern matching and generic pattern matching, they cannot be straightforwardly extended to efficiently answer the $\mu$OPPM problem.

## 3. Polynomial $\mu$OPPM cases

Section 3.1 introduces the first efficient algorithm to solve the $\mu$OPPM problem when one string is indeterminate. Section 4 discusses the existence of efficient solvers when both strings are indeterminate. Section 3.2 introduces a polynomial time, algorithm for the Alternate-$\mu$OPPM, a subproblem of $\mu$OPPM where both strings may have indeterminate characters, but never in the same position.

Section 3.3 finishes by discussing how to OPPM search a text that is larger than the pattern, using filtration techniques to skip sure misses.

### 3.1. $O(mr \lg r)$ time $\mu$OPPM with one determinate string

Given a determinate string $x$, of length $m$, we can start by sorting $x$ so that its characters form a non-decreasing sequence. We will represent the sorted sequence as $x_\pi$, where $\pi$ is the permutation that transforms $x$ into $x_\pi$.

For example, given $x = (1, 4, 3, 1)$ we have that $x[0] = x[3] < x[2] < x[1]$ and therefore $\pi = (0, 3, 2, 1)$ and $x_\pi = (1, 1, 3, 4)$.

Now let us focus on $y$, which we assume also has size $m$. First let us consider the simpler case when $y$ is also determinate. To verify if $x \approx y$ it is enough to verify the $m - 1$ order relations of its sorted characters. For example, if $y = (2, 5, 4, 3)$ then there is no op-match as not all the order relations are preserved in $y$, since $y[0] < y[3] < y[2] < y[1]$. In particular we have $x[0] = x[3]$ but $y[0] < y[3]$. As illustrated by this example, an effective process to obtain this conclusion is to permute $y$ according to $\pi$ into $y_\pi$ and compare the resulting strings. In this example we obtained $x_\pi = (1, 1, 3, 4)$ and $y_\pi = (2, 3, 4, 5)$. Notice that both are sorted into non-decreasing. However whereas this is expected of $x_\pi$, in the case of $y_\pi$ it is a fortunate coincidence. This condition is actually necessary for an op-match to occur. However, as seen in this example, it is not sufficient. The issue being that whereas the first two characters of $x_\pi$ are equal the same does not happen in $y_\pi$. Hence the decisive test to guarantee an op-match is to compare the sequence of order relations in $x_\pi$ with the sequence of order relations in $y_\pi$. To obtain this sequence we compare the consecutive values in $x_\pi[i]$, i.e., we compare $x_\pi[i]$ with $x_\pi[i + 1]$. The same is done for $y_\pi$. In this case these sequences are $(=, <, <)$, for $x_\pi$, and $(<, <, <)$ for $y_\pi$. Since these sequences differ, namely in the first relation, there is no op-match between $x$ and $y$. Moreover notice that because $x_\pi$ is a non-decreasing sequence the only possible relations in the order sequence for $x_\pi$ are equality $(=)$ and smaller than $(<)$. For $y_\pi$ the greater than relation may occur, which makes it a mismatch guarantee.

Let us now focus on generalising this approach to the case where $y$ is an indeterminate string. Let $x_\pi$ and $y_\pi$ be the permuted strings, as before, but note that now $y_\pi$ contains indeterminate characters. Consider The example where $x = (4, 1, 4, 2)$ and $y = (2|7, 2, 7|8, 1|4|8)$. By ordering $x$ we obtain $x_\pi = (1, 2, 4, 4)$. Applying the same permutation $\pi = (1, 3, 0, 2)$ on $y$ yields $y_\pi = (2, 1|4|8, 2|7, 7|8)$. We can still produce the order sequence for $x_\pi$, which in this case is $(<, <, =)$. However due to the indeterminate nature of the characters in $y$ there is no meaningful way to produce a similar sequence for $y_\pi$. In essence the problem is that $y_\pi$ simultaneously represents several sequences at the same time. A naive approach would be to generate all those sequences and examine them one by one. This process can easily become overwhelming, as illustrated in Table 1. The problem is that there may be a huge number of such sequences and only a few are relevant. In this example only one sequence corresponds to an op-match.

Let us then focus on how to determine the matching sequence without having to enumerate all the possible sequences represented by $y_\pi$. As noticed before the order sequence of $x_\pi$ contains only relations that are equalities or smaller than relations. For now let us ignore equalities. The process of removing equalities reduces our example to $x_\pi = (1, 2, 4)$ and $y_\pi = (2, 1|4|8, 7)$. We will explain how after the exposition of the main algorithm.

**Table 1**
All possible determinate strings and order relation sequences for $y_\pi = (2, 1|4|8, 2|7, 7|8)$.

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 1 | 2 | 7 | $>$ | $<$ | $<$ |
| 2 | 1 | 2 | 8 | $>$ | $<$ | $<$ |
| 2 | 1 | 7 | 7 | $>$ | $<$ | $=$ |
| 2 | 1 | 7 | 8 | $>$ | $<$ | $<$ |
| 2 | 4 | 2 | 7 | $<$ | $<$ | $<$ |
| 2 | 4 | 2 | 8 | $<$ | $>$ | $<$ |
| 2 | 4 | 7 | 7 | $<$ | $<$ | $=$ |
| 2 | 4 | 7 | 8 | $<$ | $<$ | $<$ |
| 2 | 8 | 2 | 7 | $<$ | $>$ | $<$ |
| 2 | 8 | 2 | 8 | $<$ | $>$ | $<$ |
| 2 | 8 | 7 | 7 | $<$ | $>$ | $=$ |
| 2 | 8 | 7 | 8 | $<$ | $>$ | $<$ |

When all the relations are inequalities determining if an op-match exists reduces to verifying if it is possible to obtain an increasing sub-sequence from $y_\pi$. The following Lemma summaries this observation.

**Lemma 1.** *Given a determinate string $x$ and an indeterminate string $y$, let $x_\pi$ be a non-decreasing sort of $x$ and $y_\pi$ the result of applying the same permutation to $y$. Then $x$ and $y$ have an op-match if and only if there is valid assignment of $y_\pi$ that is an increasing sequence.*

**Proof.** ($\Rightarrow$) If $x$ and $y$ are an op-match then there is a valid assignment $\$y$ of $y$ that is an op-match with $x$. Hence applying $\pi$ to $x$ and $\$y$ also yields an op-match, meaning that $x_\pi$ and $\$y_\pi$ are also an op-match. Now $x_\pi$ is an increasing sequence so $\$y_\pi$ most also be an increasing sequence, which moreover is a valid assignment of $y_\pi$.

($\Leftarrow$) If there is valid assignment $\$y_\pi$ of $y_\pi$ that is an increasing sequence then $y_\pi$ and $x_\pi$ are an op-match, since both are strictly increasing sequences. Finally we can apply the inverse permutation $\pi^{-1}$ to $\$y_\pi$ which yields a valid assignment of $y$ that is op-match with $x$. $\square$

Hence let us now focus on describing a polynomial algorithm for verifying if there is valid assignment of $y_\pi$ that is an increasing sequence. Our first non-trivial approach consists in reducing this problem to the longest increasing subsequence (LIS) problem. For this reduction to work appropriately, we do not want that an indeterminate character maps to more than one element of the LIS. This is assured by sorting the possible elements of an indeterminate character in decreasing order. In our example we have $y_\pi = (2, 1|4|8, 7)$. The resulting integer sequence is $z = (2, 8, 4, 1, 7)$, where the numbers $8, 4, 1$ stand for $y_\pi[1] = \{1, 4, 8\}$. Polynomial algorithms for computing the LIS are readily available [42]. The LIS of sequence of $n$ numbers can be computed in $O(n \lg n)$ time and $O(n)$ space. This gives us an $O(mr \lg(mr))$ time and $O(mr)$ space algorithm for the $\mu$OPPM problem when one string is determinate. This approach has the advantage that information from all the increasing sub-sequences can be obtained from this data structure. This information reveals all the ways a op-match may exist and about longest partial op-matches, which are matches that preserve most relations but not all.

Still for the exact $\mu$OPPM problem we are interested in determining whether the LIS is big enough. More precisely the LIS only guarantees a match when its size is $|y_\pi|$. In our running example there is only an op-match if the LIS is of size 3, which is the case because the LIS of $z = (\mathbf{2}, 8, \mathbf{4}, 1, \mathbf{7})$ is $(2, 4, 7)$. Hence there is a faster and more straightforward way to obtain a valid assignment of $y_\pi$ that is an increasing sequence. It is enough to maintain a minimum counter (`nextMin`) that gets updated as it processes the indeterminate characters of $y_\pi$. At each character $y_\pi[i]$ this counter is updated to the smallest possible value that is strictly larger than its current value.

Let us illustrate this process with our current example. We initialise `nextMin` to $-\infty$ and process $y_\pi[0] = \{2\}$. This updates `nextMin` to 2. Next we process $y_\pi[1] = \{1, 4, 8\}$ and update `nextMin` to 4, because 1 is smaller than 2. Finally `nextMin` gets updated to 7 by processing $y_\pi[2] = \{7\}$. The resulting sequence is again $(2, 4, 7)$ as expected. This process is only a linear scan and requires $O(mr)$ time, thus reducing the performance of the resulting algorithm to $O(mr \lg r)$. The corresponding pseudo-code is given in Algorithm 1.

**Theorem 2.** *It is possible to determine if strings $x$ and $y$, of size $m$, where $x$ is determinate and $y$ is indeterminate, are a $\mu$OPPM in $O(rm \lg r)$ time and $O(mr)$ space.*

**Proof.** First we will explain the procedure for handling equalities. The time necessary for this process is the only time that was not yet accounted by our explanation.

Consider again our initial example with $x_\pi = (1, 2, 4, 4)$ and $y_\pi = (2, 1|4|8, 2|7, 7|8)$. In this case the character 4 occurs twice at the end of $x_\pi$. We can handle equalities by contracting the sequence $x_\pi$, so that consecutive equal characters get removed. We represent the resulting sequence by $x'_\pi = (1, 2, 4)$. To enforce this restriction on $y_\pi$ we instead intersect the corresponding sets, in this case we intersect the last two sets $y_\pi[2]$ and $y_\pi[3]$. The resulting sequence is denoted $y'_\pi$. We thus obtain $y'_\pi[2] = y_\pi[2] \cap y_\pi[3] = \{2, 7\} \cap \{7, 8\} = \{7\}$. This greatly simplifies the problem as now the number

**Table 2**
Alternate-$\mu$OPPM instance with $r = 2$.

| $i$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $p$ | 1 | 6\|7 | 5 | 3 | 1\|4 |
| $t$ | 8\|10 | 19 | 2\|17 | 15\|30 | 16 |

of sequences represented by $y'_\pi = (2, 1|4|8, 7)$ is much smaller. We have only three sequences, instead of the twelve. The sequences are $(2, 1, 7)$, $(2, 4, 7)$ and $(2, 8, 7)$. Here $(2, 4, 7)$ is the desired sequence, as now we are searching for a strictly increasing sequence, given that equalities were removed. We can now apply the procedure above to $x'_\pi$ and $y'_\pi$.

The only time that was not accounted so far was the time to compute intersections. For this purpose we can sort both sets $y_\pi[j]$ and $y_\pi[i]$ and intercept then by a process similar to merging on mergesort. Essentially we scan both strings at the same time increasing the pointer of the smallest value. An element is selected for the interception if it is being considered in both lists at the same time. This procedure requires $O(r \lg r)$ time per intersection. Alternatively we can use binary search trees which require the same time or hash tables which have even better best and average case performance but may be worse in the worst case. For simplicity we assume $O(r \lg r)$ time per intersection. □

### *3.2. Polynomial time alternate-$\mu$OPPM*

In this section we define Alternate-$\mu$OPPM as the subproblem of $\mu$OPPM where both strings ($x$ and $y$, interchangeable) may have indeterminate characters, but never in the same position (an example is shown in Table 2). In this section we assume that $p = x$ and $t = y$, the case where $|t| > |p|$ is discussed in Section 3.3. We show that Alternate-$\mu$OPPM is polynomial in both the number of indeterminacies $r$ (which may be different in each position and string) and the length of the strings $m$.

We present a Dynamic Programming solution that merges two subproblems (of the kind in Section 3.1) in $O(m^2 \times r)$ time and space. For simplicity, we will assume that $r$ is the same in every position. It is always possible to add repeats up to the maximum $r$ when they are not.

*Intuition.* We partition the strings $p$ and $t$ (see example in Table 2) into two subproblems: one subproblem where only $p$ has indeterminacies and $t$ has fixed letters ($p_{ind}$ and $t_{fix}$, see Table 3), and another where only $t$ has indeterminacies and $p$ has fixed letters ($t_{ind}$ and $p_{fix}$, see Table 4). Then, we sort the subproblems by $t_{fix}$ and $p_{fix}$, respectively. Finally, we use a DP algorithm to merge the two by extending our solution one position at a time, choosing the leftmost available position of either $p_{ind}$ or $t_{ind}$ until all the positions have been used. It is important to notice that this DP algorithm assumes that the problems does not contain equality restrictions. In particular this means that the numbers in the fixed parts are all distinct, in each string. If this is not the case we proceed as in Section 3.1 by intercepting the corresponding sets of indeterminate characters. We illustrate this situation in Table 5. Note that on this table the number 3 occurs twice in $p_{fix}$, to be precise we have that $p_{fix}[1] = p_{fix}[2] = 3$. This means that we should intercept the sets $t_{ind}[1]$ and $t_{ind}[2]$. In this case this yields $\{7, 15, 30\} \cap \{15, 30, 55\} = \{15, 30\}$. This reduces Table 5 to Table 4, to which our DP algorithm can now be applied. The time for this pre-processing step is $O(rm \log r)$ because we need to sort the indeterminate characters to obtain linear time interception, as seen in Algorithm 1. We are also assuming the sorted condition of the indeterminate sets in the remaining algorithm.

**Algorithm 1** $O(mr \lg r)$ $\mu$OPPM algorithm with one determinate string.

```
Require: determinate x, indeterminate y (|x| = |y| = m)
 1: π ← sortedIndexes(x)                    ▷ O(m) if |Σ| = O(m); O(m lg m) otherwise
 2: x_π ← permute(x, π)                     ▷ O(m)
 3: y_π ← permute(y, π)                     ▷ O(mr)
 4: j ← 0
 5: y'_π[0] ← y_π[0]
 6: for i = 1 to m − 1 do                   ▷ O(mr lg r)
 7:     if x_π[i] = x_π[i − 1] then         ▷ O(r lg r)
 8:         y'_π[j] ← y'_π[j] ∩ y_π[i]
 9:     else
10:         j ← j + 1
11:         y'_π[j] ← y_π[i]
12:     end if
13: end for
14: s ← |y'_π|
15: nextMin ← −∞
16: for i = 0 to s − 1 do                   ▷ O(mr)
17:     nextMin ← min{a | a ∈ y'_π[i], a > nextMin}   ▷ O(r)
18:     if ∄ nextMin then return false
19:     end if
20: end for
21: return true
```

**Table 3**
Sorted $p_{ind}$ and $t_{fix}$ partition of Table 2.

| $i$ | 0 | 1 |
|---|---|---|
| $p_{ind}$ | 1\|4 | 6\|7 |
| $t_{fix}$ | 16 | 19 |

**Table 4**
Sorted $t_{ind}$ and $p_{fix}$ partition of Table 2.

| $i$ | 0 | 1 | 2 |
|---|---|---|---|
| $p_{fix}$ | 1 | 3 | 5 |
| $t_{ind}$ | 8\|10 | 15\|30 | 2\|17 |

**Table 5**
Sorted $t_{ind}$ and $p_{fix}$ partition of Table 2.

| $i$ | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| $p_{fix}$ | 1 | **3** | **3** | 5 |
| $t_{ind}$ | 8\|10 | 7\|**15**\|**30** | **15**\|**30**\|55 | 2\|17 |

*Table.* Let $m_p$ and $m_t$ be the lengths of $p_{ind}$ (and $t_{fix}$) and $t_{ind}$ (and $p_{fix}$), respectively. In our example, we have $m_p = 2$ and $m_t = 3$ as shown in Tables 2, 3 and 4. We fill a table $T$ with dimensions $\mathrm{T}[m_p + 1, m_t + 1, r, 2]$. Because we are interested in an exact match (rather than a longest match), each entry $\mathrm{T}[i, j, k, l]$ only needs to be filled with a boolean value that indicates whether a match of length $i + j$ exists, where the first $i$ positions of $p_{ind}$ and the first $j$ positions of $t_{ind}$ have been chosen, and the last extension was done by choosing the indeterminate character at index $k$ of string $l \in \{p_{ind}, t_{tind}\}$. We represent these boolean values by using a bit $b \in \{0, 1\}$, where 0 corresponds to false and 1 to true. The problem has a solution if and only if $\exists_{k,l} : \mathrm{T}[m_p, m_t, k, l] = 1$. We can recover the solution by backtracking in the table, as we use 1 to represent the existence of a match.

*Algorithm.* To initialise the table, we set $\mathrm{T}[0, 1, k, t_{ind}]$ and $\mathrm{T}[1, 0, k, p_{ind}]$ to 1, for all valid $k$. All other entries start with 0. To fill the remaining entries, we use the following equalities:

$$T[i, j, k, p_{ind}] = \begin{cases} 1, & \text{if } \exists_{k'} \in [0..r-1] : \textbf{checkPP}(i, j, k, k') \\ 1, & \text{if } \exists_{k'} \in [0..r-1] : \textbf{checkTP}(i, j, k, k') \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$T[i, j, k, t_{ind}] = \begin{cases} 1, & \text{if } \exists_{k'} \in [0..r-1] : \textbf{checkPT}(i, j, k, k') \\ 1, & \text{if } \exists_{k'} \in [0..r-1] : \textbf{checkTT}(i, j, k, k') \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

where:

Note that although the condition in Equations (1) and (2) is that $\exists k'$, in reality we only need the minimum such $k'$. We can maintain an auxiliary table $\mathrm{A}[m_p + 1, m_t + 1, l]$ to store and retrieve it in constant time. $A$ is initialised with $\infty$.

**Algorithm 2** Try to extend solution through $p_{ind}$ after $p_{ind}$.

$k' = \mathrm{A}[i-1, j, p_{ind}]$;
checkPP$(i, j, k, k')$
**return** $\mathrm{T}[i-1, j, k', 0] = 1$ and $p_{ind}[i-1, k] > p_{ind}[i-2, k']$

**Algorithm 3** Try to extend solution through $p_{ind}$ after $t_{ind}$.

$k' = \mathrm{A}[i-1, j, t_{ind}]$;
checkTP$(i, j, k, k')$
**return** $\mathrm{T}[i-1, j, k', 1] = 1$ and $p_{ind}[i-1, k] > p_{fix}[j-1]$ and $t_{fix}[i-1] > t_{ind}[j-1, k']$

**Algorithm 4** Try to extend solution through $t_{ind}$ after $p_{ind}$.

$k' = \mathrm{A}[i, j-1, p_{ind}]$;
checkPT$(i, j, k, k')$
**return** $\mathrm{T}[i, j-1, k', 0] = 1$ and $t_{ind}[j-1, k] > t_{fix}[i-1]$ and $p_{fix}[j-1] > p_{ind}[i-1, k']$

**Algorithm 5** Try to extend solution through $t_{ind}$ after $t_{ind}$.

$k' = \mathrm{A}[i, j-1, t_{ind}]$;
checkTT$(i, j, k, k')$
**return** $\mathrm{T}[i, j-1, k', 1] = 1$ and $t_{ind}[j-1, k] > t_{ind}[j-2, k']$

**Table 6**
Dynamic Programming table $T$ for the instance shown in Table 2. The top table corresponds to T[$i, j, 0, p_{ind}$], second table to T[$i, j, 0, t_{ind}$], the third table to T[$i, j, 1, p_{ind}$] and the bottom table to T[$i, j, 1, t_{ind}$].

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | **1** |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | **1** | **1** | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | **1** | 0 |
| 2 | 1 | 1 | 1 | 1 |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | **1** |
| 2 | 0 | 0 | 0 | 0 |

**Table 7**
Auxiliary Table $A$ for the instance shown in Table 2. The top table stores values A[$i, j, p_{ind}$] and the bottom table stores values A[$i, j, t_{ind}$].

| A[$i, j, p_{ind}$] | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ |
| 1 | 0 | 1 | **1** | $\infty$ |
| 2 | 0 | 0 | 0 | **0** |

| A[$i, j, t_{ind}$] | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | **0** | **0** | 1 |
| 1 | $\infty$ | $\infty$ | $\infty$ | **1** |
| 2 | $\infty$ | $\infty$ | $\infty$ | $\infty$ |

Whenever we fill any entry T[$i, j, k, l$], we set A[$i, j, l$] $= min(k, A[i, j, l])$, thus building it in the same bounds as the original table. We can retrieve $k'$ in constant time using the Obtain line in functions **checkPP**, **checkPT**, **checkTP**, or **checkTT** described in Algorithms 2, 3, 4, and 5, respectively.

*Retrieving the solution.* We start by finding some $k, l$ such that T[$m_p, m_t, k, l$] $= 1$. At T[$i, j, k, p_{ind}$], we retrieve the previous solution at T[$i-1, j, k', p_{ind}$] or T[$i-1, j, k', t_{ind}$]; at T[$i, j, k, t_{ind}$] we retrieve it at T[$i, j-1, k', p_{ind}$] or T[$i, j-1, k', t_{ind}$]. In each case, we need to check both. We can find one correct $k'$ in constant time by checking the two respective entries of the auxiliary table $A$. We repeat the process $i + j$ times, until we construct the whole solution and reach $(i, j) = (0, 0)$.

*Time and Space.* The dynamic programming table $T$, described above, uses $O(m_p \times m_t \times r \times 2) \leq O(m^2 \times r)$ space, as $m = m_p + m_t$. We also have an auxiliary table $A$ that is strictly smaller than $T$. Time-wise, we take constant time to fill each entry (we have to check two entries of the matrix, do three character comparisons, and one check on the auxiliary table to retrieve the minimum $k'$), hence the algorithm will also take $O(m^2 \times r)$ time.

**Theorem 3.** *Two strings of size m can be verified to Alternate-μOPPM in $O(rm(m + \lg r))$ time and $O(m^2)$ space.*

**Proof.** We start by solving equalities by intercepting the corresponding sets in $O(rm \log r)$ time. We then sort two pairs of strings of size $m_p$ and $m_t$ in $O(m_p \times \lg(m_p))$ and $O(m_t \times \lg(m_t))$. For the dynamic programming we fill the two tables, $T$ and $A$ of size at most $O(m_p \times m_t \times r \times 2)$, each entry taking constant time. We can retrieve the solution by $O(m_p + m_t)$ accesses. The overall algorithm, then, takes $O(m_p \times m_t \times r \times 2)$ and is polynomial in both time and space.

To reduce the space to $O(m^2)$ notice that table $A$ requires only this amount of space. Also note that when $k' = A[i-1, j, 0]$ then $T[i-1, j, k', 0] = 1$, i.e., the table $A$ can be used to obtain all the values of $T$ that correspond to true. Hence table $A$ can be computed without using table $T$. Table $T$ was given for presentation purposes as its boolean values are simpler than the index positions stored by table $A$. □

*Example.* The dynamic programming table $T$ of the example of Table 2 is shown in Table 6. The respective auxiliary table $A$ is shown in Table 7. We show here some computations of interest to elucidate how the algorithm works.

**Table 8**
Solution of the example of Table 2.

| i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| p | 1 | 3 | 1\|**4** | 5 | **6**\|7 |
| t | **8**\|10 | **15**\|30 | 16 | 2\|**17** | 19 |

Consider, for example, positions T$[0, 3, 0, t_{ind}]$ and T$[0, 3, 1, t_{ind}]$ in Table 6. Because $i = 0$, we are merely trying to compare $t_{ind}[2]$ with $t_{ind}[1]$, by choosing the smallest $k$ at $t_{ind}[2]$. First, we note that T$[0, 2, 0, t_{ind}]$= 1, i.e., it is possible to form an increasing sequence by using only the first letter in the first two indeterminate sets of $t_{ind}$. These letters are respectively $t_{ind}[0, 0] = 8 < 15 = t_{ind}[1, 0]$. However T$[0, 3, 0, t_{ind}]$= 0, whereas T$[0, 3, 1, t_{ind}]$= 1. The value T$[0, 3, 0, t_{ind}]$ corresponds to the sequence formed by using only the first letter in the first three indeterminate sets of $t_{ind}$ in this case we now have $t_{ind}[1, 0] = 15 \geq t_{ind}[2, 0] = 2$. Hence the sequence $8, 15, 2$ is not increasing and so T$[0, 3, 0, t_{ind}]$= 0. On the other hand if we are allowed to choose any of the two letters in the indeterminate sets we can fix this situation, in particular by using $t_{ind}[1, 0] = 15 < t_{ind}[2, 1] = 17$. We then obtain the sequence $8, 15, 17$, which is increasing and therefore T$[0, 3, 1, t_{ind}]$= 1.

Let us now see how to recover a solution. The entries in **bold** are the ones we follow, here, part of the solution. To start with, note that both T$[2, 3, 0, t_{ind}]$ and T$[2, 3, 1, t_{ind}]$ are set to 0, so we know that our solution must end with a choice in $p_{ind}$. We can see that both choices of $k$ work in $p_{ind}[2]$, consistent with the solution working with either $p_{ind}[1] = 6$ or $p_{ind}[1] = 7$. We choose $p_{ind}[1] = 6$, the one with the minimum value of $k$ (cf. A$[2, 3, p_{ind}]$= 0). Then, we must check whether the solution comes from T$[2 - 1, 3, k', p_{ind}]$, or from T$[2 - 1, 3, k'', t_{ind}]$, for the respective minimum $k'$ and $k''$. The only entry of T$[1, 3, k, l]$ set to 1 is T$[1, 3, 1, t_{ind}]$, corresponding to choosing $t_{ind}[2] = 17$ (cf. A$[1, 3, t_{ind}]$= 1). Because the last choice was through $t_{ind}$, we move to $(i, j - 1) = (1, 2)$. Once again, only one entry is set to 1, T$[1, 2, 1, p_{ind}]$, corresponding to choosing $p_{ind}[0] = 4$ (cf. A$[1, 2, p_{ind}]$= 1) and we move to $(i, j) = (0, 2)$. At this point, we know that all remaining choices come from $t_{ind}$, but we will continue following the algorithm. Both T$[0, 2, 0, t_{ind}]$ and T$[0, 2, 1, t_{ind}]$ are set to 1, but we only need the minimum $k$, which is 0 (cf. A$[0, 2, t_{ind}]$= 0), corresponding to choosing $t_{ind}[1] = 15$. Finally, we consider at $(i, j) = (0, 1)$. Once again, both T$[0, 1, 0, t_{ind}]$ and T$[0, 1, 1, t_{ind}]$ are set to 1, and we choose the minimum $k$, 0 (cf. A$[0, 1, t_{ind}]$= 0), corresponding to $t_{ind}[0] = 8$. Having reached $(i, j) = (0, 0)$, we have a complete solution. The solution is shown in Table 8, with the chosen values in **bold**.

### *3.3. Handling larger texts*

In this Section we consider the case where the pattern $p$ and the text $t$ are not necessarily the same size. The next lemma summarises the strait-forward approach of testing $p$ against every position of $t$, by considering a smaller sub-string.

**Lemma 4.** *Given an indeterminate pattern string $p$ of length $m$ and a determinate text string of length $n$, the $\mu$OPPM problem can be solved in $O(nmr \lg r)$ time.*

**Proof.** From Theorem 2, verifying if two strings of length $m$ op-match can be done in $O(mr \lg r)$ time (indetermination in one string). We repeat this procedure for at most $n - m + 1$ sub-strings of $t$. □

Note that only one of the strings needs to be determine so the same approach applies when $p$ is determinate and $t$ is indeterminate. Moreover the same procedure can be applied with the algorithm in Theorem 3.

This Lemma further triggers the research question "*Is $O(nmr \lg r)$ a tight bound to solve the $\mu$OPPM?*", here left as an open research question.

Irrespectively of the answer, the analysis of the average complexity is of complementary relevance. State-of-the-art research on the exact OPPM problem shows that the average performance of some $O(nm)$ worst case time algorithms can outperform linear time algorithms [12,43,44].

Motivated by the evidence gathered by these works, we suggest the use of filtration procedures to improve the average complexity of the proposed $\mu$OPPM algorithm while still preserving its complexity bounds. A filtration procedure encodes the input pattern and text, and relies on this encoding to efficiently find positions in the text with a high likelihood to op-match a given pattern. Despite the diversity of string encodings, simplistic binary encodings are considered to be the state-of-the-art in OPPM research [12,43]. In accordance with Chhabra et al. [12], a pattern $p$ can be mapped into a binary string $p'$ expressing increases (1), equalities (0) and decreases (0) between subsequent positions. By searching for exact pattern matches of $p'$ in an analogously transformed text string $t'$, we guarantee that the verification of whether $p[0..m-1]$ and $t[i..i+m-1]$ orders are preserved is only performed when exact binary matches occur. Illustrating, given $p = (3, 1, 2, 4)$ and $t = (2, 4, 3, 5, 7, 1, 4, 8)$, then $p' = (1, 0, 1, 1)$ and $t' = (1, 1, 0, 1, 1, 0, 1, 1)$, revealing two matches $t'[1..4]$ and $t'[4..7]$: one spurious match $t[1..4]$ and one true match $t[4..7]$.

When handling indeterminate strings the concept of increase, equality and decrease needs to be redefined. Given an indeterminate string $x$, consider $x'[i] = 1$ if $max(x[i]) < min(x[i + 1])$, $x'[i] = 0$ if $min(x[i]) \geq max(x[i + 1])$, and $x'[i] = *$ otherwise. Under this encoding, the pattern matching problem is identical under the additional guard that a character in $p'$ always matches a do not care position, $t'[i] = *$, and vice-versa. Illustrating, given $p = (6, 2|3, 5)$ and

**Table 9**
$\mu$OPPM instance corresponding to the 3CNF-SAT formula $(z_1 \vee \neg z_2 \vee z_3) \wedge (\neg z_1 \vee z_2 \vee z_4)$.

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Formula | $z_1$ | $z_2$ | $z_3$ | $z_4$ | $c_1$ | $c_2$ |
| Pattern | 1 | 2 | 3 | 4 | 1\|2\|3 | 1\|2\|4 |
| Text | 1\|2 | 3\|4 | 5\|6 | 7\|8 | 2\|3\|6 | 1\|4\|8 |

$t = (3|4, 5, 6|8, 6|7, 3, 5, 4|6, 7|8, 4)$, then $p' = (0, 1)$ and $t' = (1, 1, *, 0, 1, *, 1, 0)$, leading to one true match $t[3..5]$ – e.g. $t[3..5] = (6, 3, 5)$ – and one spurious match $t[5..7]$. The $*$ characters play the role of wild cards. Efficient algorithms for this problem were described by Gusfield [45, Sec 9.3]. If there are $w$ wild cards in $p'$ the algorithm requires $O(w)$ time to check a position in $t'$. Hence scanning $t'$ for all the wild card matches of $p'$ takes $O(nw)$ time.

The properties of the proposed encoding guarantee that the wild card matches of $p'$ in $t'$ cannot skip any op-match of $p$ in $t$. Thus, when combining the premises of Lemma 4 with the previous observation, we guarantee that the computed $\mu$OPPM solution is sound.

The application of this simple filtration procedure prevents the recurring $O(mr\lg r)$ verifications $n-m+1$ times. Instead, the complexity of the proposed method to solve the $\mu$OPPM problem becomes $O(nw + dmr\lg r + n)$ (when one string is indeterminate) where $d$ is the number of wild card matches, which is potentially smaller than $n$. According to previous work on exact OPPM with filtration procedures [12], SBNDM2 and SBNDM4 algorithms [46] (Boyer-Moore variants) were suggested to match binary encodings. In the presence of small patterns, Fast Shift-Or (FSO) [47] can be alternatively applied [12].

A similar approach can be used for alternate $\mu$OPPM, in which case $w$ must count the total amount of wild cards, in $p$ and $t$, and the $O(w)$ worst case bound per position becomes worse. In this case we note that the trivial $O(m)$ bound must also apply, but the average case may be significantly better.

## 4. General cases

In this Section we start by showing that general $\mu$OPPM is **NP**-hard, Section 4.1. We then show how to encode an instance into a boolean formula, of polynomial size, so that it can be tackled with a SAT solver.

### 4.1. $\mu$OPPM is NP-hard

In this section, we define $\{3, 3\}$-$\mu$OPPM as the subproblem of $\mu$OPPM where both the pattern and the text have indeterminate characters in any position (with at least one position having three or more indeterminate characters in both pattern and text) and prove it **NP**-hard (thus proving the same for general $\mu$OPPM). We do this with a direct reduction from 3CNF-SAT, first presenting the construction and then the proof of equivalence between the two instances. The construction is similar to the one by Bose et al. for the permutation matching problem [48].

*Construction.* To ease the description of the construction itself, we start by describing how we represent an instance of 3CNF-SAT. First, we assume that every literal and clause has some ordering. We have a set $V$ of literals, and a set $C$ of clauses. Each clause $c$ is represented by two tuples, $(z_{c,0}, z_{c,1}, z_{c,2})$ and $(l_{c,0}, l_{c,1}, l_{c,2})$. $z_{c,i} \in \{0, \dots, |V|-1\}$ represents the index of literal $i$ of clause $c$; $l_{c,i} \in \{0, 1\}$ represents the value of the literal $i$ in clause $c$, having the value of 0 for positive literals and 1 for negative literals. For example, the clause $(v_1 \vee \neg v_2, \vee v_5)$ would be represented by the two tuples $z = (1, 2, 5)$ and $l = (0, 1, 0)$.

Although the designations of text or pattern are interchangeable in this section, we will use pattern for the simpler string (with less indeterminacies) and text for the more complicated string (with more indeterminacies). We use $p$ and $t$ for the pattern and text, respectively, or $s$ when they are interchangeable.

Both text and pattern have two parts, one representing literals and the other representing clauses. Each literal, and clause, has a single position in each string to represent it, dividing $s$ into $s_V = s[0..|V|-1]$ and $s_C = s[|V|..|V|+|C|-1]$. In $p_V$, we have a simple sequence of literals given by their indexes, so $p[i] = i+1$, for $i \in \{0, \dots, |V|-1\}$; in $t_V$ we have a similar sequence, but each literal takes one of two variable values to represent an assignment of true or false, so $t[i] = 2 \times (i+1)$ or $2 \times (i+1) - 1$. We choose the larger value to represent the assignment of true. In $s_C$, each position has three indeterminacies, corresponding to the three variables of the clause. In $p_C$, we choose one of the three literals of the respective clause. For clause $c$, with literals $v_1, v_2, v_5$ (regardless of their value being positive or negative), its position in $p$, $p[|V|+c] = 1|2|5$. In $t_C$, as in $p_C$ we choose one of the literals, but now the *value* of the literal must match the clause. More precisely for a positive literal it should be an even value and for a negative literal it should be an odd value, i.e., for $z_i$ it should be $2i - 0$ and for $\neg z_i$ it should be $2i - 1$. For example for clause $c$, $(v_1 \vee \neg v_2 \vee v_5)$, $t[|V|+c] = 2 \times 1 - 0 \mid 2 \times 2 - 1 \mid 2 \times 5 - 0 = 2|3|10$. An example of this construction is shown in Table 9.

**Lemma 5.** *The construction above takes polynomial time.*

**Proof.** It is easy to see that, assuming that variables and clauses are numbered, we can simply scan the formula once to construct our two strings in linear time. □

**Lemma 6.** *The initial 3CNF-SAT clause is satisfiable if and only if there is an $\mu$OPPM match between the two constructed strings.*

**Proof.** We start by showing how solving the $\mu$OPPM instance solves the initial 3CNF-SAT instance. To solve $\mu$OPPM, we need to choose exactly one value for each position in $p$ and $t$ that leads to two order-isomorphic strings. To extract the solution, we can limit ourselves to look at the initial part of $t$, $t[0, |V| - 1]$, which sets the value of each literal.

First, note that $p$ function is to maintain consistency between the values of literals chosen in $t$. By choosing only literals in $p$, and not their values, we force equality between all such literals. Because of order-isomorphism, this equality must be kept in $t$, forcing a valid solution to use a single value for each literal (since different values match in $p$ but mismatch in $t$). If we choose a literal to be positive/negative at some position in $t$, we force the value of that literal to be positive/negative at every position in $t$.

Now, we focus on $t_C$. Every clause has exactly one position in $t_C$, and each of these positions have three choices of value, matching only the three values that satisfy a clause. Because we must choose one value in each position to solve our $\mu$OPPM instance, we must choose one value that satisfies each clause, for every clause.

Putting these two properties together, to solve $\mu$OPPM we must choose a literal value that satisfies each clause *and* those literals must have consistent values. This establishes the equivalence between the solutions of the two instances.

We can easily extract the solution from $\mu$OPPM to 3CNF-SAT by checking whether the values in $t_V$ are even or odd, true or false, respectively. There is a unique solution to 3CNF-SAT given an $\mu$OPPM solution.

To extract the solution from 3CNF-SAT to $\mu$OPPM, we take the values assigned to each variable and choose the respective values in $t_V$. Then, we need to choose values for $p_C$ and $t_C$, which can easily be done by choosing any of the literals that satisfies its respective clause. There may be multiple $\mu$OPPM solutions for a given 3CNF-SAT solution. □

**Theorem 7.** *$\{3, 3\}$-$\mu$OPPM is **NP**-hard.*

**Proof.** Using Lemmas 5 and 6 we show that 3CNF-SAT $\leq_p$ $\{3, 3\}$-$\mu$OPPM by constructing an instance of $\mu$OPPM in polynomial time. The solutions can also be retrieved and translated in polynomial time. □

**Theorem 8.** *$\mu$OPPM is **NP**-hard.*

**Proof.** Since $\{3, 3\}$-$\mu$OPPM is a particular case of $\mu$OPPM, and it is **NP**-hard, then $\mu$OPPM is **NP**-hard. □

### 4.2. Encoding $\mu$OPPM with indeterminate pattern and text in SAT

To solve the general $\mu$-OPPM with two indeterminate strings, we will present a simple SAT encoding of this problem. First, we build a visual representation of the problem in the form of a graph, which can be seen as finding an independent set in an $m$-partite graph. This leads to a straightforward and conceptually simple formulation of $\mu$OPPM in terms of SAT clauses.

*Constructing the graph.* We build an $m$-partite graph, where each layer corresponds to a position in the strings.

Each layer has $r_x \times r_y$ nodes, where $r_x$ ($r_y$) is the number of indeterminate characters in string $x$ ($y$, respectively). This leads to a total of $m \times r^2$ nodes in the graph. We use $r$ throughout this section to indicate the maximum value of indeterminate characters at any position, in either string.

We add an edge between every pair of nodes in each layer, leading to $r^2 \times m$ edges. This will enforce an *at most one* character choice per string and position in the encoding. These are cardinality constraints.

Then, for every pair of layers, and every pair of nodes in the different layers, we add an edge if the order between the character choices in string $x$ does not match the ones in string $y$. This leads to, at most, $(m \times r^2)^2$ edges. These are conflict constraints.

In total, our graph has $m \times r^2$ nodes and $m^2 \times r^4 + m \times r^2$ edges, for a total size of $O(m^2 \times r^4)$. Fig. 1 shows an example of this step for the strings $x = (2, 2|4, 0)$ and $y = (2, 0|1|3, 1|3)$. The edges within layers are not shown. The only solution is given by $\$x = (2, 4, 0)$ and $\$y = (2, 3, 1)$, and can be seen in the graph as the only independent set (of size 3, which is the number of layers).

*SAT encoding.* The $\mu$OPPM problem is trivially reduced to finding an independent set (of size $m$) in the graph $G$ described above. Every edge $e$ represents a violation of the order-isomorphism between two positions, and we must choose exactly one node per layer (character per string position).

This can be encoded in SAT. Our variables correspond to the nodes in the graph, given by $z_{i,\$x[i],\$y[i]}$ where $i$ is the layer, $\$x[i]$ is the choice of $x[i]$'s character and $\$y[i]$ is the choice of $y[i]$'s character. We first add a clause of size $O(r^2)$ for each layer of the graph, to choose *at least one* node per layer. These are cardinality clauses. Then, for every edge (from node $\$x[i], \$y[i]$ to node $\$x[j], \$y[j]$) on the graph, we add a clause $(\neg z_{i,\$x[i],\$y[i]} \vee \neg z_{j,\$x[j],\$y[j]})$ to prevent choosing both nodes (since they are incompatible). These include both cardinality and conflict clauses.

In total, we have a formula with $O(m \times r^2)$ variables and $O(m^2 \times r^4)$ clauses. It may also be worthwhile to note that only $m$ of those clauses have more than 2 variables. The resulting formula is of the form:

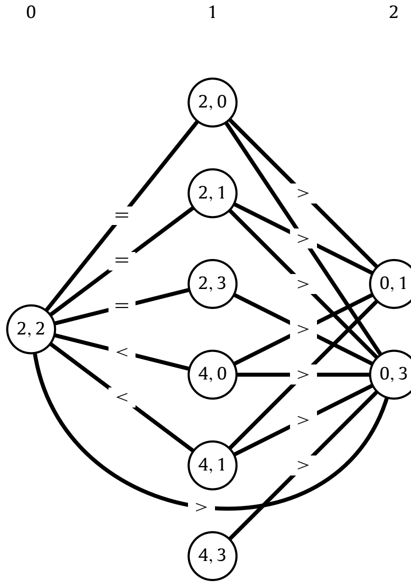



**Fig. 1.** Conflicts when matching $x = (2, 2|4, 0)$ against $y = (2, 0|1|3, 1|3)$. In each node, the left character corresponds to string $x$, and the right character to string $y$. The edges represent a conflict, with labels by a mismatch between the order in $x$ and $y$.

$$
\begin{aligned}
\phi = & \bigwedge_{i=0}^{m-1} \Big( \bigvee_{\substack{\$y[i] \in y[i] \\ \$x[i] \in x[i]}} z_{i,\$x[i],\$y[i]} \Big) \\
& \wedge \bigwedge_{e \in G} \left( \neg z_{i,\$x[i],\$y[i]} \vee \neg z_{j,\$x[j],\$y[j]} \right)
\end{aligned}
\tag{3}
$$

**Theorem 9.** *The $\mu$OPPM can be reduced to SAT with the encoding above.*

**Proof.** If $x \approx y$ then $\phi$ is satisfiable, and if $x$ does not op-match $y$ then $\phi$ is not satisfiable. ($\Rightarrow$) When $x$ op-matches $y$, there is an assignment of values in $x$ and $y$ such that $\$x \approx \$y$. At each position $i$, and given the assignments $\$x[i]$ and $\$y[i]$, we set the SAT variable $z_{i,\$x[i],\$y[i]}$ to true. Because every clause derives from a conflict (or selecting multiple characters per position) and none is present (by assumption), the formula $\phi$ is satisfied. ($\Leftarrow$) When $x$ does not match $y$, there is no assignment of values $\$x \in x$ and $\$y \in y$ such that $\$x \approx \$y$. Any choice of character violates either a conflict clause (by assumption, because there is no valid match), or a cardinality clause (setting to true less/more than one variable per position $i$), thus making $\phi$ formula unsatisfiable. $\square$

Given the unique properties of the proposed SAT encoding, effective backtracking in accordance with the clauses in the first line of (3), as well as dedicated conflict pruning principles derived from remaining clauses in (3), can be considered to optimise efficient SAT solvers to solve the $\mu$OPPM problem.

## 5. Discussion and conclusion

We studied the $\mu$OPPM problem according to the number and position of the indeterminate characters. We have shown that, for any number of indeterminacies, $\mu$OPPM has a polynomial-time algorithm for indeterminate characters in a single string (Section 3.1), or in both strings, but never in both strings at the same position (Section 3.2). For indeterminate characters in both strings at the same position, we have also shown that for *at least three* indeterminacies (at select positions), the problem is **NP**-hard (Section 4).

There is a gap in between these two results. For the strings where there are *at most two* indeterminate characters in both strings at the same position. This issue is not settled by this paper. Interestingly this question was very recently studied by Gawrychowski, Ghazawi, and Landau [49] which established that indeed the problem is **NP**-hard even with only two indeterminate characters.[1] This result hence means that we now have efficient algorithms for the polynomial cases and the remaining cases are known to be **NP**-hard.

[1] See section 4.

In conclusion, this work addressed the relevant but not yet studied problem of finding order-preserving pattern matches on indeterminate strings ($\mu$OPPM). We showed that the problem has a linear time and space solution when one string is indeterminate. In addition, the $\mu$OPPM problem (when both strings are indeterminate) was mapped into a satisfiability formula of polynomial size and two simple types of clauses in order to study efficient solvers for the $\mu$OPPM problem. Moreover the $\mu$OPPM problem was shown to be **NP**-hard in general. Finally, we showed that solvers of the $\mu$OPPM problem can be boosted in the presence of filtration procedures and we identified a still open problem in what concerns the computational complexity of the $\mu$OPPM problem when restricted to at most two indeterminate characters in both strings at the same position.

## Declaration of competing interest

All authors have participated in (a) conception and design, or analysis and interpretation of the data; (b) drafting the article or revising it critically for important intellectual content; and (c) approval of the final version.

This manuscript has not been submitted to, nor is under review at, another journal or other publishing venue.

The authors have no affiliation with any organization with a direct or indirect financial interest in the subject matter discussed in the manuscript

## Acknowledgments

We thank an anonymous reviewer for pointing out the space improvement in Theorem 3.

This work was developed in the context of a secondment granted by the BIRDS MASC RISE project funded in part by EU H2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement no.690941. The work reported in this article was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 and projects, PTDC/CCI-BIO/29676/2017, TUBITAK/0004/2014, SAICTPAC/0021/2015.

## References

[1] X. Ge, Pattern matching in financial time series data, in: Final Project Report for ICS, Vol. 278, 1998.
[2] J. Kim, P. Eades, R. Fleischer, S.-H. Hong, C.S. Iliopoulos, K. Park, S.J. Puglisi, T. Tokuyama, Order-preserving matching, Theor. Comput. Sci. 525 (2014) 68–79.
[3] R. Henriques, A. Paiva, Seven principles to mine flexible behavior from physiological signals for effective emotion recognition and description in affective interactions, in: PhyCS, 2014, pp. 75–82.
[4] R. Henriques, Learning from High-Dimensional Data using Local Descriptive Models, Ph.D. thesis, Instituto Superior Tecnico, Universidade de Lisboa, Lisboa, 2016.
[5] R. Henriques, S.C. Madeira, Bicspam: flexible biclustering using sequential patterns, BMC Bioinform. 15 (2014) 130.
[6] R. Henriques, C. Antunes, S. Madeira, Methods for the efficient discovery of large item-indexable sequential patterns, in: New Frontiers in Mining Complex Patterns, in: LNCS, vol. 8399, Springer International Publishing, 2014, pp. 100–116.
[7] S. Kawashima, M. Kanehisa, Aaindex: amino acid index database, Nucleic Acids Res. 28 (2000) 374.
[8] M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, T. Waleń, A linear time algorithm for consecutive permutation pattern matching, Inf. Process. Lett. 113 (2013) 430–433.
[9] S. Cho, J.C. Na, K. Park, J.S. Sim, A fast algorithm for order-preserving pattern matching, Inf. Process. Lett. 115 (2015) 397–402.
[10] S. Cho, J.C. Na, K. Park, J.S. Sim, Fast order-preserving pattern matching, in: Combinatorial Optimization and Applications, Springer, 2013, pp. 295–305.
[11] D. Belazzougui, A. Pierrot, M. Raffinot, S. Vialette, Single and multiple consecutive permutation motif search, in: Int. Symposium on Algorithms and Computation, Springer, 2013, pp. 66–77.
[12] T. Chhabra, J. Tarhio, A filtration method for order-preserving matching, Inf. Process. Lett. 116 (2016) 71–74.
[13] R. Henriques, A.P. Francisco, L.M.S. Russo, H. Bannai, Order-preserving pattern matching indeterminate strings, in: Proceedings of the Symposium on Combinatorial Pattern Matching (CPM), 2018.
[14] D.E. Knuth, J.H. Morris Jr, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 323–350.
[15] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Commun. ACM 20 (1977) 762–772.
[16] T. Chhabra, S. Faro, M.O. Külekci, J. Tarhio, Engineering order-preserving pattern matching with simd parallelism, Softw. Pract. Exp. 47 (2017) 731–739.
[17] A. Amir, O. Lipsky, E. Porat, J. Umanski, Approximate Matching in the l1 Metric, CPM, vol. 5, Springer, 2005, pp. 91–103.
[18] O. Lipsky, E. Porat, Approximate matching in the l$\infty$ metric, Inf. Process. Lett. 105 (2008) 138–140.
[19] A. Amir, Y. Aumann, P. Indyk, A. Levy, E. Porat, Efficient computations of $l1$ and $l\infty$ rearrangement distances, Theor. Comput. Sci. 410 (2009) 4382–4390.
[20] E. Porat, K. Efremenko, Approximating general metric distances between a pattern and a text, in: ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2008, pp. 419–427.
[21] E. Cambouropoulos, M. Crochemore, C. Iliopoulos, L. Mouchard, Y. Pinzon, Algorithms for computing approximate repetitions in musical sequences, Int. J. Comput. Math. 79 (2002) 1135–1148.
[22] M. Crochemore, C.S. Iliopoulos, T. Lecroq, W. Plandowski, W. Rytter, Three heuristics for delta-matching: delta-bm algorithms, in: CPM, Springer, 2002, pp. 178–189.
[23] R. Clifford, C. Iliopoulos, Approximate string matching for music analysis, Soft Computing 8 (2004) 597–603.
[24] P. Clifford, R. Clifford, C. Iliopoulos, Faster algorithms for $\delta$, $\gamma$-matching and related problems, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 2005, pp. 68–78.
[25] I. Lee, R. Clifford, S.-R. Kim, Algorithms on extended $(\delta, \gamma)$-matching, Comput. Sci. Appl., ICCSA 2006 (2006) 1137–1142.
[26] I. Lee, J. Mendivelso, Y.J. Pinzón, $\delta\gamma$–parameterized matching, in: International Symposium on String Processing and Information Retrieval, Springer, 2008, pp. 236–248.
[27] J. Mendivelso, I. Lee, Y.J. Pinzón, Approximate function matching under $\delta$-and $\gamma$-distances, in: SPIRE, Springer, 2012, pp. 348–359.
[28] J. Holub, W. Smyth, S. Wang, Fast pattern-matching on indeterminate strings, J. Discret. Algorithms 6 (2008) 37–50. Selected papers from AWOCA 2005.
[29] R. Cole, C. Iliopoulos, T. Lecroq, W. Plandowski, W. Rytter, On special families of morphisms related to $\delta$-matching and don't care symbols, Inf. Process. Lett. 85 (2003) 227–233.

[30] A. Apostolico, Algorithms and Theory of Computation Handbook, Chapman & Hall/CRC, 2010, chapter 15.
[31] B.S. Baker, A theory of parameterized pattern matching: algorithms and applications, in: ACM Symposium on Theory of Computing, ACM, 1993, pp. 71–80.
[32] A. Amir, M. Farach, S. Muthukrishnan, Alphabet dependence in parameterized matching, Inf. Process. Lett. 49 (1994) 111–115.
[33] A. Amir, M. Farach, Efficient 2-dimensional approximate matching of half-rectangular figures, Inf. Comput. 118 (1995) 1–11.
[34] A. Amir, Y. Aumann, G.M. Landau, M. Lewenstein, N. Lewenstein, Pattern matching with swaps, J. Algorithms 37 (2000) 247–266.
[35] S. Muthukrishnan, New results and open problems related to non-standard stringology, in: Combinatorial Pattern Matching, Springer, 1995, pp. 298–317.
[36] D. Cantone, S. Cristofaro, S. Faro, An efficient algorithm for $\delta$-approximate matching with $\alpha$-bounded gaps in musical sequences, in: IW on Experimental and Efficient Algorithms, Springer, 2005, pp. 428–439.
[37] D. Cantone, S. Cristofaro, S. Faro, On tuning the $(\delta, \alpha)$-sequential-sampling algorithm for $\delta$-approximate matching with alpha-bounded gaps in musical sequences, in: ISMIR, 2005, pp. 454–459.
[38] K. Fredriksson, S. Grabowski, Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance, Inf. Retr. 11 (2008) 335–357.
[39] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, E. Porat, Overlap matching, Inf. Comput. 181 (2003) 57–74.
[40] A. Amir, Y. Aumann, M. Lewenstein, E. Porat, Function matching, SIAM J. Comput. 35 (2006) 1007–1022.
[41] A. Amir, I. Nor, Generalized function matching, J. Discret. Algorithms 5 (2007) 514–523. Selected papers from Ad Hoc Now 2005.
[42] M.L. Fredman, On computing the length of longest increasing subsequences, Discrete Math. 11 (1975) 29–35.
[43] D. Cantone, S. Faro, M.O. Külekci, An efficient skip-search approach to the order-preserving pattern matching problem, in: Stringology, 2015, pp. 22–35.
[44] T. Chhabra, M.O. Külekci, J. Tarhio, Alternative algorithms for order-preserving matching, in: Stringology, 2015, pp. 36–46.
[45] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.
[46] B. Ďurian, J. Holub, H. Peltola, J. Tarhio, Improving practical exact string matching, Inf. Process. Lett. 110 (2010) 148–152.
[47] K. Fredriksson, S. Grabowski, Practical and Optimal String Matching, SPIRE, vol. 3772, Springer, 2005, pp. 376–387.
[48] P. Bose, J.F. Buss, A. Lubiw, Pattern matching for permutations, Inf. Process. Lett. 65 (1998) 277–283.
[49] P. Gawrychowski, S. Ghazawi, G.M. Landau, On indeterminate strings matching, in: I.L. Gørtz, O. Weimann (Eds.), 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020), in: Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, vol. 161, 2020, pp. 14:1–14:14.