A Bitwise Approach to SCER Matching in Indeterminate Strings

Simone Faro
Department of Mathematics and Computer Science, University of Catania, Italy

Dominik Köppl
University of Yamanashi, Kofu, Japan

Thierry Lecroq
Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie Univ, LITIS UR 4108, CNRS NormaSTIC FR 3638, IRIB, Rouen, France

Francesco Pio Marino
Department of Mathematics and Computer Science, University of Catania, Italy
Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie Univ, LITIS UR 4108, CNRS NormaSTIC FR 3638, IRIB, Rouen, France

Abstract

We study the problem of matching a determinate pattern against an indeterminate text of the same length n, where each text position is a set of possible characters drawn from an alphabet Σ of size σ. We study this matching problem under the order-preserving and parameterized matching setting. For that, we encode character sets by bit expressions using sum-free sequences. This encoding enables constant-time character comparisons and avoids explicit set operations. We present an optimal $\mathcal{O}(n)$ time algorithm for order-preserving matching and an $\mathcal{O}(n + (\sigma_p^x \cdot \sigma_p^y)\sqrt{\sigma_p^x + \sigma_p^y})$ time algorithm for parameterized matching, where σ_p^x and σ_p^y denote the number of distinct parameterized symbols in the pattern and the text, respectively. The proposed techniques significantly reduce overhead while maintaining exactness, offering practical performance improvements for pattern matching under uncertainty. Additionally, we extend the parameterized matching framework to allow mismatches, for which we present an algorithm with time complexity $\mathcal{O}(\sigma^2 n \log n \ + \ n\sigma^2\sqrt{\sigma}\log(n\sigma))$.

2012 ACM Subject Classification Information systems → Information retrieval

Keywords and phrases string matching, indeterminate strings, SCER matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.21

Supplementary Material *Software (Source Code)*: `https://github.com/marfra99x/SCER-ISM` [7] archived at `swh:1:dir:c33fa1f2fdf35f4030601b03c54ad27d5931dfd0`

Funding *Dominik Köppl*: JSPS KAKENHI Grant Number 25K21150.

1 Introduction

The problem of *string matching* is a foundational challenge in theoretical computer science, with widespread applications in areas such as *bioinformatics*, *text retrieval*, and *pattern recognition* [5, 12]. In its classical formulation, the problem assumes deterministic input, where each position in a string contains a single, fixed character drawn from a finite alphabet. Under this assumption, a match between two strings is defined as exact equality at every position.

However, in many real-world scenarios, this assumption does not hold. Strings may exhibit uncertainty – so-called *indeterminate strings* – in which some positions contain a *set* of possible characters rather than a unique one. This type of uncertainty naturally arises in biological sequence analysis (e.g., ambiguous nucleotides), noisy or corrupted text data, and symbolic pattern recognition, motivating the need for more expressive and robust matching models [19].

A standard approach to matching indeterminate strings is to interpret a match as the existence of a consistent assignment of characters to each position such that the two strings become identical. When the alphabet is small, an efficient implementation strategy is to encode characters and character classes using bit-parallelism. In this setting, the alphabet Σ is mapped to a set of bit masks, and each symbol is represented as a binary vector indicating the presence or absence of characters. For instance, in the DNA alphabet $\{A, C, G, T\}$, each character can be encoded using a 4-bit mask, and indeterminate symbols correspond to bitwise combinations of these masks. This representation allows character comparisons to be performed using constant-time bitwise operations, making it highly effective in practice [17].

An alternative encoding strategy is based on number-theoretic techniques, where each character is associated with a distinct prime number and indeterminate symbols are represented as products of primes. Matching can then be tested via greatest common divisor (GCD) computations. While conceptually elegant, this approach is generally impractical due to the high computational cost of arithmetic operations and the exponential growth of prime products, restricting its applicability to very small alphabets [6].

Beyond exact matching, indeterminate string matching has been studied under more general equivalence relations that capture richer notions of similarity. A particularly important class is that of *substring consistent equivalence relations* (SCERs). An equivalence relation $\approx$ is said to be substring consistent if, for any two strings x and y such that $x \approx y$, it holds that (i) $|x| = |y|$, and (ii) $x[i..j] \approx y[i..j]$ for all substrings. This property ensures that matching is preserved under substring extraction and is satisfied by several well-known matching paradigms.

Notable examples of SCER-based matching include *parameterized matching*, *order-preserving matching*, and variants of palindrome pattern matching. These models have been widely studied due to their applicability in domains where structural or relative relationships between symbols are more important than their exact identities.

1.1 Related Work

Efficient algorithms for string matching over indeterminate strings have been extensively studied. A widely used technique for small alphabets is bit-parallel encoding, where characters and character classes are represented as bit masks. This approach enables constant-time set operations and has been successfully applied in practical domains such as bioinformatics and text processing [17].

Alternative encoding methods based on prime numbers have also been proposed. In this framework, each character is assigned a distinct prime number, and indeterminate symbols are represented as products of primes. Matching is then reduced to testing divisibility via GCD computations. Although mathematically appealing, this approach suffers from significant practical limitations due to the cost of arithmetic operations and the rapid growth of encoded values [6].

The study of indeterminate string matching under equivalence relations has led to the development of the SCER framework. Several important string matching paradigms fall within this class, including parameterized matching [2, 10], order-preserving matching [13, 9], and palindrome-related matching problems [15].

Henriques *et al.* [13] introduced the first SCER-based approach to indeterminate string matching by investigating the order-preserving pattern matching problem. They showed that an indeterminate pattern of length m can be matched against a determinate string in $\mathcal{O}(mr \log r)$ time, where r is the maximum number of symbols per indeterminate position, by exploiting structural properties of the value domains.

More recently, Gawrychowski *et al.* [11] studied the problem of matching two indeterminate strings under SCERs. They presented the first polynomial-time algorithm for the Cartesian tree variant of the problem, achieving deterministic $\mathcal{O}(m \log^2 m)$ time and expected $\mathcal{O}(m \log m \log\log m)$ time, using $\mathcal{O}(m \log m)$ space, for constant r.

Furthermore, they established the NP-hardness of the order-preserving variant for $r = 2$, resolving an open question, and showed that the parameterized version is NP-hard for $r \geq 2$. Since both problems reduce to classical string matching when $r = 1$, these results provide a complete complexity classification for these SCER-based matching problems.

1.2 Our Contribution

In this work, we focus on the problem of matching a determinate pattern string x against an indeterminate text string y, both of length n, under SCER-based matching models. In particular, we study both order-preserving and parameterized matching, and further extend the latter to support mismatches.

Our contributions can be summarized as follows:

- We present an $\mathcal{O}(n)$-time algorithm for order-preserving matching between a determinate pattern and an indeterminate text.
- We establish that our approach admits an efficient parallelization, analyzing its work and span in the PRAM model, while leaving practical aspects for future investigation.
- We develop an algorithm for parameterized matching with running time $\mathcal{O}\big(n + (\sigma_p^x \cdot \sigma_p^y)\sqrt{\sigma_p^x + \sigma_p^y}\big)$, where σ_p^x and σ_p^y denote the number of distinct parameterized symbols in the pattern and text, respectively.
- We extend the parameterized matching framework to support mismatches, providing an efficient solution in the indeterminate setting.

The remainder of the paper is organized as follows. In Section 2, we introduce the necessary definitions and notation. Section 3 presents our linear-time algorithm for order-preserving matching. Section 4 describes our parameterized matching algorithm, while Section 5 extends it to support mismatches. Section 6 presents our experimental results. Finally, Section 7 concludes the paper and outlines directions for future work.

2 Preliminaries

This section introduces the notation, definitions, and mathematical foundations required for our approach. We begin by defining indeterminate strings and their matching criteria, followed by an introduction to sum-free sequences (an increasing sequence of numbers $a_1 < a_2 < \cdots < a_n$ is *sum-free* if no term a_k can be expressed as the sum of any subset of its preceding terms $a_1, \ldots, a_{k-1}$; e.g., the powers of two $1, 2, 4, 8, \ldots$) and their computational properties, which form the basis for our efficient encoding and search mechanisms.

Let Σ be a finite and totally ordered alphabet. A determinate string u of length n is defined as: $u = u[1]u[2]\ldots u[n], \quad u[i] \in \Sigma, \quad \forall i \in [1, n]$. An *indeterminate string* w of length n is a string whose symbols are taken from the power set of Σ, that is $w = w[1]w[2]\ldots w[n], \quad w[i] \subseteq \Sigma, \quad \forall i \in [1, n]$. Since each position can be a singleton set, indeterminate strings are a generalization of determinate strings.

We define *assignment* of an indeterminate string y is a string $\hat{y}$ such that $\hat{y}[i] \in y[i]$.

For an indeterminate symbol $w[i] \subseteq \Sigma$, we denote its elements in increasing order as $w[i] = \langle w[i, 1], w[i, 2], \ldots, w[i, |w[i]|]\rangle, \quad w[i, 1] < w[i, 2] < \cdots < w[i, |w[i]|]$. Equivalently, each element can be characterized as $w[i, j] = \{c \in w[i] : |\{d \in w[i] : d \leq c\}| = j\}$, so in particular $w[i, 1]$ is the minimum of $w[i]$. This notation allows us to explicitly refer to the j-th element of $w[i]$.

This perspective also highlights a connection with *2D-strings* [3], since each position of an indeterminate string can be regarded as a (sorted) list of possible symbols.

A common technique for handling indeterminate strings is to use a *bitwise encoding*, where each character position is represented by a bitmask indicating which symbols are present [18]. This allows set operations, such as membership testing, to be performed efficiently. Specifically, each symbol $a \in \Sigma$ is assigned a unique bit position, and the presence of a set of characters at a given position is represented as a bitwise OR operation over their corresponding bit values. Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be an alphabet of size σ. We define a mapping $\mathcal{M} : \Sigma \to \mathbb{N}$ that assigns a distinct power of two to each symbol: $\mathcal{M}(a_i) = 2^{i-1}, \quad \forall i \in [1, \sigma]$, similarly given a determinate string $y[1..n]$ we define $\mathcal{M}(y) = [\mathcal{M}(y[1]), \mathcal{M}(y[2]), \ldots, \mathcal{M}(y[n])]$. Then, an indeterminate letter $w[i] \subseteq \Sigma$ is encoded by the mapping $\bar{\mathcal{M}} : 2^\Sigma \to \mathbb{N}$ as: $\bar{\mathcal{M}}(w[i]) = \sum_{a \in w[i]} \mathcal{M}(a)$, i.e. the k-th bit is set to 1 if and only if the k-th symbol of Σ is contained in $w[i]$. Likewise, we can generalize $\bar{\mathcal{M}}$ to strings. Using this encoding, a match between an indeterminate text and an indeterminate pattern at position j can be determined using a bitwise AND operation: $\bar{\mathcal{M}}(x[i]) \cap \bar{\mathcal{M}}(y[j+i-1]) \neq 0, \quad \forall i \in [1, m]$. This formulation allows for fast comparisons and eliminates the need for explicit set intersection checks.

We define the symbol $\wedge$ to denote the bitwise AND operation between two binary strings of equal length. Given $a, b \in \{0,1\}^n$, the result of $a \wedge b$ is the binary string $c \in \{0,1\}^n$ such that $c_i = a_i \cdot b_i$ for all $i \in [1, n]$.

Throughout this work, we assume the standard word-RAM model with word size $w = \Omega(\log n)$. Under this assumption, bitwise operations on machine words can be performed in constant time when the alphabet size σ is bounded by w. For larger alphabets, an additional $\log \sigma$ factor is required to account for set operations.

2.1 Order Preserving Pattern Matching

Two strings u and v of the same length over Σ are said to be *order-equivalent*, written $u \approx v$, if $u[i] < u[j] \iff v[i] < v[j]$ for all pairs of positions i, j on the strings. For example, $\langle 5, 2, 9, 4, 3 \rangle \approx \langle 6, 1, 7, 5, 2 \rangle$, which shows in particular that their central value is the largest in both strings.

The order preserving pattern matching (OPPM) problem is naturally defined as follows: given a pattern $x \in \Sigma^*$ and a text $y \in \Sigma^*$, find all the substrings of y that are order-equivalent to x. For example, $\langle 5, 2, 9, 4, 3 \rangle$ occurs equivalently at position 2 on $\langle 4, \underline{6, 1, 7, 5, 2}, 9, 8, 3 \rangle$ but nowhere else. For instance, it does not appear at position 5 because $5 < 8$, while in the pattern the corresponding values satisfy $5 > 4$.

Given a determinate string x of length m, an indeterminate string y of equal length is said to be *order-preserving* with respect to x, denoted $x \approx y$, if there exists a determinate assignment $\hat{y}$ of y such that the relative orders of the symbols in x and $\hat{y}$ coincide, that is,

$$x[i] \leq x[j] \iff \hat{y}[i] \leq \hat{y}[j], \quad \text{for all } 1 \leq i, j \leq m.$$

2.2 Parameterized Pattern Matching

Parameterized pattern matching is a SCER-based model in which symbols from a parameterized alphabet Σ_p may be dynamically renamed to match symbols in a text, as long as symbolic consistency is preserved. Let $\Sigma = \Sigma_s \cup \Sigma_p$, where Σ_s contains fixed symbols and Σ_p contains parameterized ones. Given a pattern $x \in \Sigma^n$ and a text $y \in \Sigma^n$, a parameterized match occurs if there exists a bijective mapping $f : \Sigma_p(x) \to \Sigma_p(y)$, where $\Sigma_p(x)$ and $\Sigma_p(y)$ denote the subsets of parameterized symbols that actually appear in x and y, respectively.

The mapping f must preserve equality: if two positions in x contain the same parameter symbol, then their mapped counterparts in y must also be equal – and vice versa. Fixed symbols from Σ_s must match exactly, without transformation.

▶ **Example 1.** Let the fixed alphabet be $\Sigma_s = \{\mathtt{a}, \mathtt{b}\}$ and the parameterized alphabet be $\Sigma_p = \{\mathtt{X}, \mathtt{Y}\}$. Consider the pattern $x = \mathtt{a\ X\ b\ X\ a}$ and the text $y = \mathtt{a\ Y\ b\ Y\ a}$, both of length 5. A parameterized match occurs under the bijective mapping $f(\mathtt{X}) = \mathtt{Y}$, where fixed symbols `a` and `b` match exactly, and the repeated parameter symbol `X` is consistently mapped to `Y`. Since the mapping is bijective, it preserves the symbolic structure of x, and the transformed pattern equals y, so the match is valid.

Given a determinate pattern x of length m, an indeterminate text y of equal length is said to *parameterized-match* x if there exists a determinate assignment $\hat{y}$ of y and a bijective mapping $f : \Sigma_p(x) \to \Sigma_p(\hat{y})$ such that:

- for all positions i, if $x[i] \in \Sigma_s$, then $\hat{y}[i] = x[i]$;
- for all positions i, j such that $x[i], x[j] \in \Sigma_p$, we have $x[i] = x[j] \iff f(\hat{y}[i]) = f(\hat{y}[j])$.

3 Order-Preserving Matching in Indeterminate Strings

Let $x = x[1]x[2]\dots x[n]$ and $y = y[1]y[2]\dots y[n]$ be two strings of equal length, drawn from a finite ordered alphabet $\Sigma = \{a_1, a_2, \dots, a_\sigma\}$ of size σ. Each symbol a_i is assigned a unique numerical value through the mapping: $\mathcal{M} : \Sigma \to \mathbb{N}, \quad \mathcal{M}(a_i) = 2^{i-1}, \quad \text{for } 1 \le i \le \sigma$. This encoding ensures that each determinate character is represented as a distinct power of two.

For an indeterminate string y, given the presence of indeterminate positions i, where a character may belong to a subset of Σ, the numerical representation is given by: $\bar{\mathcal{M}}(y[i]) = \sum_{c \in y[i]} \mathcal{M}(c)$.

This representation ensures that determinate characters correspond to single powers of two, while indeterminate positions are encoded as sums of distinct powers of two. Since the powers of two form a sum-free sequence, each distinct subset of characters maps to a unique integer representation. This property allows for efficient encoding and comparison of indeterminate positions using bitwise operations. Thus, in this model, the string x is determinate, which means that each position contains exactly one character, while y is indeterminate, allowing each position to contain a subset of characters from Σ.

An order-preserving match occurs when the text y matches x, such that the relative order of the characters in x is preserved in y. This formulation enables pattern matching in settings where the uncertainty in y must be taken into account while maintaining structural consistency.

Indeterminate OPPM
Input: A determinate pattern $x \in \Sigma^n$ and an indeterminate text y of length n, where each $y[i] \subseteq \Sigma$.
Output: **true** if there exists an assignment $\hat{y}$ of y such that x order-preserving matches $\hat{y}$; **false** otherwise.

To formalize this notion, we define a rank function $\rho : \Sigma^n \to [1..n]^n$ that assigns an order-preserving rank to each character of a string. A ranking is order-preserving if the ranked string order-preserving matches with the original string. The rank function $\rho(x)$ is defined as $\rho(x) = (r_1, r_2, \dots, r_n), \quad \text{where} \quad r_i = |\{j \mid x[j] \le x[i]\}|$. $\rho(x)$ defines a permutation, whose inverse $\rho(x)^{-1} = [p_1, \dots, p_n]$ gives the indices of x in increasing order with respect to their ranks.

Since ρ preserves order relations, determining an order-preserving match in our setting reduces to verifying whether the ranks of the positions $\rho(x)^{-1} = \rho(\hat{y})^{-1}$ is the same, for some assignment $\hat{y}$ of y. The algorithm processes the text y according to the order induced by the rank vector $\rho(x)$ of the pattern x. For each position i in this order, it checks whether $y[i]$ is determinate or indeterminate.

▶ **Lemma 2.** *A positive integer t is a power of two if and only if the following bitwise condition holds: $t \wedge (t-1) = 0$.*

Proof. Suppose t is a power of two. Then there exists an integer $k \geq 0$ such that $t = 2^k$. The binary representation of t consists of a single bit set at position k, with all lower bits set to 0: $t_i = (100\ldots000)_2$. Subtracting 1 results in a number where all bits below position k are set to 1, and the bit at position k is set to 0: $t-1 = (011\ldots111)_2$. Computing the bitwise AND operation, we obtain: $t \wedge (t-1) = (100\ldots000)_2 \wedge (011\ldots111)_2 = (000\ldots000)_2 = 0$. Thus, the condition holds for all powers of two.

Conversely, suppose t is not a power of two. Then, in its binary representation, t contains at least two bits set to 1. The binary representation of $t-1$ follows from the standard property that subtracting 1 from a number in binary flips all trailing 0s to 1s and the rightmost 1 to 0. Consequently, at least one bit set in t remains set in $t-1$, ensuring that the bitwise AND operation satisfies: $t \wedge (t-1) \neq 0$. Thus, the given condition is violated for all non-powers of two.

Since we have proven both directions, the statement follows. ◀

▶ **Example 3.** Consider the numbers 4 and 6 and their binary representations:

$4 = (0100)_2, \quad 4-1 = 3 = (0011)_2$. Applying the bitwise AND operation: $4 \wedge 3 = (0100)_2 \wedge (0011)_2 = (0000)_2 = 0$. Thus, the condition holds, confirming that 4 is a power of two. $6 = (0110)_2, \quad 6-1 = 5 = (0101)_2$. Applying the bitwise AND operation: $6 \wedge 5 = (0110)_2 \wedge (0101)_2 = (0100)_2 \neq 0$. Since the result is nonzero, the condition fails, confirming that 6 is not a power of two.

If $y[i]$ is determinate, it must respect the order constraints, which means that it should be greater than or equal to the previously selected value. If $y[i]$ is indeterminate, a greedy selection strategy is used to extract the smallest valid value that is at least as large as the previous selection.

In order to formalize the greedy procedure, we define a sequence of *selected values*. Suppose the positions $p_1, p_2, \ldots, p_{i-1}$ from $\rho(x)^{-1} = [p_1, \ldots, p_n]$ have already been processed. Then the algorithm has chosen values $y[p_1, j_{p_1}] \leq y[p_2, j_{p_2}] \leq \cdots \leq y[p_{i-1}, j_{p_{i-1}}]$, according to the following procedure: set $j_{p_1} = 1$ (because $y[p_1, j_{p_1}] = \min_j y[p_1, j] = y[p_1, 1]$), and for each $k \in [2..i-1]$ we set j_{p_k} such that $y[p_k, j_{p_k}] = \min\{\, y[p_k, j] \mid y[p_k, j] \geq y[p_{k-1}, j_{p_{k-1}}] \,\}$.

The element $v_{k-1} = y[p_{k-1}, j_{p_{k-1}}]$ is called the *selected value* at step $k-1$. At step i, the algorithm checks that $y[p_i]$ contains at least one candidate greater than or equal to the selected value v_{i-1}.

▶ **Lemma 4.** *Given a previously selected value p, the operation $y[i] \wedge (-p)$ removes all values in $y[i]$ that are smaller than p, where $-p$ is the two complement representation of p.*

Proof. In binary representation, the two's complement of p is given by $-p = \bar{p} + 1$, where $\bar{p}$ represents the bitwise negation of p. This transformation results in a binary number in which all bits up to and including the least significant 1 of p remain 0, while all higher bits are set to 1.

Performing the bitwise AND operation between $y[i]$ and $-p$ ensures that any bits in $y[i]$ that correspond to positions below the least significant 1 in p are set to 0, while bits at positions equal to or higher than p remain unchanged. Consequently, all values in $y[i]$ smaller than p are removed, and only values at least as large as p are retained. ◀

▶ **Example 5.** Let $t = 22$ be an integer value, and $p = 4$. Their binary representations are $t = 22 = (10110)_2$, $p = 4 = (00100)_2$. The two's complement of p is computed as $-p = (11100)_2$. Applying the bitwise AND operation yields $t \wedge (-p) = (10110)_2 \wedge (11100)_2 = (10100)_2 = 20$. The result 20 retains only values that are at least $p = 4$, ensuring that all smaller values are removed.

▶ **Lemma 6.** *Given an integer t, the smallest remaining nonzero bit is selected using the operation $t \wedge -t$. This operation isolates the least significant set bit of t in constant time.*

Proof. The expression $-t$ represents the two's complement of t, which is given by $-t = \bar{t} + 1$. The two's complement negation flips all bits of t up to the least significant set bit, which is then incremented by 1. This ensures that $-t$ has exactly one bit set at the position of the least significant 1 in t and all lower bits reset to 0.

Applying the bitwise AND operation, $t \wedge -t$, retains only the least significant set bit in t, as all higher bits are set to 0. ◀

▶ **Example 7.** Let $t = 20$, whose binary representation is $t = (10100)_2$. Computing the two's complement: $-t = (01100)_2$. Applying the bitwise AND operation: $t \wedge -t = (10100)_2 \wedge (01100)_2 = (00100)_2 = 4$. Thus, 4 isolates the least significant set bit of t, ensuring that the smallest valid value possible is selected.

Due to Lemma 6 the smallest valid match can be found in constant time.

Given a determinate pattern x and an indeterminate text y, both of length n, we first compute $\rho(x)^{-1} = [p_1, p_2, \ldots, p_n]$. This vector captures the relative ordering of characters in x.

When analyzing $y[p_i]$ after $y[p_{i-1}]$, we define $t = \bar{\mathcal{M}}(y[p_i])$, and let v_{i-1} denote the value previously selected (with $v_0 = 0$). The selected value from $y[p_i]$ must be strictly greater than v_{i-1}.

- If $y[r_i]$ is determinate, we verify that t is a power of two (as stated in Lemma 2) and that it preserves the increasing order with respect to the previously selected value, namely $t > v_{i-1}$. If either condition fails, the match is rejected; otherwise, we set $v_i = t$ and proceed with the comparison.
- If $y[r_i]$ is indeterminate, we filter out all values of t that are smaller than or equal to v_{i-1} by computing $t \leftarrow t \wedge (-v_{i-1})$ (using Lemma 4). If $t = 0$, no valid value remains, and the match fails. Otherwise, we select the smallest valid value using Lemma 6 and assign it to v_i.

If all positions satisfy the above conditions without failure, then x order-preserving matches y in the bitwise-compatible encoding model.

▶ **Example 8.** Let x and y be two strings such that $u = \mathcal{M}(x) = (8, 2, 1, 16)$ and $v = \mathcal{M}(y) = (49, 12, 25, 64)$. Computing the rank function for u, we find that $u[1] = 8$ has rank 3, $u[2] = 2$ has rank 2, $u[3] = 1$ has rank 1, and $u[4] = 16$ has rank 4, yielding $\rho(u) = (3, 2, 1, 4)$. Processing v in order of increasing rank $\rho(u)$, the first value examined is $v[3] = 25 = 11001_2$, which is indeterminate. Since it is the first selection, we take the smallest power of two in $v[3]$, which is $25 \wedge -25 = 11001_2 \wedge 00111_2 = 00001_2 = 1$. Thus, we select 1. The next value, $v[2] = 12 = 1100_2$, is also indeterminate. Since the previous selection was 1, we

remove values smaller than 1 using $12 \wedge (-1) = 1100_2 \wedge 1111_2 = 1100_2 = 12$. The smallest remaining power of two is $12 \wedge -12 = 1100_2 \wedge 0100_2 = 0100_2 = 4$. Thus, we select 4. The third value, $v[1] = 49 = 110001_2$, is indeterminate. Since the previous selection was 4, we remove values smaller than 4 using $49 \wedge (-4) = 110001_2 \wedge 111100_2 = 110000_2$. The smallest remaining power of two is $110000_2 \wedge -110000_2 = 010000_2 = 16$. Thus, we select 16. Finally, $v[4] = 64 = 1000000_2$ is determinate and greater than 16, so we select 64. The final sequence selected is therefore $\hat{v} = (16, 4, 1, 64)$, which correctly satisfies $\rho(\hat{v}) = \rho(x)$, confirming an order-preserving match.

Algorithm 1 checks whether an indeterminate string y can be matched to a pattern x while preserving order. The algorithm first sorts the positions of x by rank (line 2), then iterates through them in that order. At each step, it checks whether the corresponding entry in y is a determinate or indeterminate character (line 8). If it is indeterminate, the algorithm selects the smallest valid candidate that preserves the ordering (line 14). Lines 9 and 15 verify whether the ordering constraints are violated; if so, the algorithm terminates early. The helper functions isPowerOfTwo and findSmallestValid ensure that only valid candidates are considered.

Algorithm 1 Order-Preserving Matching Algorithm for Indeterminate Strings.

```
1:  function isPowerOfTwo(k)
2:      return (k & (k − 1)) = 0
```

```
1:  function findSmallestValid(t, p)
2:      t ← t & (−p)                          ▷ Filter out values smaller than p
3:      return t & (−t)                       ▷ Select the smallest remaining value
```

```
1:  function IndeterminateOPPM(x, y, n)
2:      ρ ← sortedIndexes(x)                  ▷ Positions sorted by ρ(x)
3:      selected ← array of size n initialized to 0
4:      predecessor ← 0
5:      for i ← 0 to n − 1 do
6:          idx ← ρ[i]
7:          t ← y[idx]
8:          if isPowerOfTwo(t) then
9:              if t ≤ predecessor then
10:                 return false              ▷ Order violated
11:             selected[idx] ← t
12:             predecessor ← t
13:         else
14:             f ← findSmallestValid(t, predecessor)
15:             if f = 0 then
16:                 return false              ▷ No valid value remains
17:             selected[idx] ← f
18:             predecessor ← f
19:     return true
```

The correctness of this method follows from the properties of rank transformation and greedy selection strategy. Since the selection process respects the ordering constraints at each step, any alternative valid sequence must be lexicographically dominated by the computed selection. If no valid selection exists at any step, then no order-preserving match can exist.

The efficiency of the approach is ensured by the constant-time operations used to determine, filter, and select the smallest remaining bit. The general complexity consists of computing the rank transformation in $O(n)$ time and processing each position of y in constant time, leading to an optimal time complexity of $O(n)$. The pseudo code of this method is shown in Algo. 1.

It is worth noticing that the structure of the proposed OPPM algorithm lends itself to parallelization in a straightforward way. Each determinate character imposes a strict ordering constraint, which partitions the rank vector into independent segments. Each thread starts using the determinate value as lower bound and selects the smallest valid values greedily. This yields fine-grained, scalable parallelism with very limited synchronization overhead. We formalize this in Theorem 9, which establishes the parallel work and time bounds.

▶ **Theorem 9.** *Let x be a determinate pattern and y an indeterminate text of length n. Let d be the number of determinate positions in y, and let $s = d + 1$. For any number of processors $p \le s$, the OPPM on (x, y) can be solved with $\mathcal{O}(n)$ total work and $\mathcal{O}(\ell + \log p)$ parallel time, where ℓ is the length of the longest segment. If the $s = d + 1$ segments are balanced so that $\ell = \mathcal{O}(n/p)$, the running time improves to $\mathcal{O}(n/p + \log p)$. Synchronization consists of a constant-time hand-off of one boundary value per segment and the final combination of p Boolean flags, which incurs $\mathcal{O}(p)$ work.*

Proof. Let $D = \{i_1 < i_2 < \cdots < i_d\} \subseteq [1..n]$ be the set of determinate positions of y, and let these positions induce $s = d + 1$ contiguous segments on the rank order of x, denoted $S_1, \ldots, S_s$. Segment S_1 contains all positions up to i_1 in the rank order; for $j \in \{2, \ldots, s-1\}$, S_j contains the positions strictly between i_{j-1} and i_j; and S_s contains the positions after i_d. Let $p \le s$ be the number of processors. We assign each S_j to one distinct processor (if $p < s$, we assign multiple segments to some processors by any balanced partition of $\{S_j\}$).

Independence. By the definition of the OPPM procedure, the selected value at the last position of S_{j-1} provides a single lower bound for all choices within S_j, while choices internal to S_j do not affect $S_{j'}$ with $j' > j$ except through the final selected value of S_j. Hence, conditioned on the single boundary value passed from S_{j-1}, all operations within S_j are independent of other segments. The first segment S_1 has no such lower bound.

Work. Each position of y is processed a constant number of times (consistent with the word-RAM assumption where bitwise operations and tests such as power-of-two, masking, and least-significant-bit selection are unit cost). The synchronization adds at most $O(p)$ additional operations: a single-word hand-off per segment boundary and the p Boolean flags for the final reduction. Therefore, the total work of the parallel algorithm is $\mathcal{O}(n+p) = \mathcal{O}(n)$.

Span / parallel time. Let L_j denote the number of positions of y handled in S_j, with $\sum_{j=1}^{s} L_j = n$. With a balanced assignment over p processors, the maximum load per processor is $\max_j L_j = \mathcal{O}(n/p)$, so the parallel time due to local computations is $\mathcal{O}(n/p)$. The only inter-segment dependency is the hand-off of the single boundary value from S_{j-1} to S_j for each $j = 2, \ldots, s$, which are constant-time in the word-RAM model, and the final Boolean reduction, which has span $\mathcal{O}(\log p)$. Thus the total parallel time is $\mathcal{O}(n/p + \log p)$.

Putting everything together, the parallel running time is $T_p = \mathcal{O}\left(\frac{n}{p} + \log p\right)$, with total work $\mathcal{O}(n + p) = \mathcal{O}(n)$. Since $p \le s$, the attainable speedup is upper-bounded by s, as at most one processor can be assigned per segment. ◀

▶ Remark 10. On a CRCW PRAM the p Boolean flags can be combined by a single concurrent AND in $O(1)$ time. On the more natural CREW PRAM, each processor writes its Boolean value into an array $B[1..p]$ and the flags are combined using a tournament tree, which takes $\mathcal{O}(\log p)$ time and $\mathcal{O}(p)$ work.

▶ **Example 11.** Consider the indeterminate text $y = (16,\ 6,\ 4,\ 10,\ 40,\ 73,\ 72)$ with binary decompositions $y = (16,\ 4+2,\ 4,\ 8+2,\ 32+8,\ 64+8+1,\ 64+8)$. The determinate entries (powers of two) are at positions 1 and 3. Assume the pattern x induces the rank order $\rho(x) = (5,\ 2,\ 3,\ 4,\ 6,\ 1,\ 7)$. With $p = d = 2$, this yields three segments: $S_1 = \{73, 6, 4\}$, $S_2 = \{10, 16\}$, $S_3 = \{40, 72\}$.

In S_1 (no lower bound) we pick 1 from 73, then 2 from 6, then the determinate 4. In S_2 (lower bound 4) we pick 8 from 10, then 16. In S_3 (lower bound 16) we pick 32 from 40, then 64 from 72. Thus the selected assignment is $\hat{y} = (16,\ 2,\ 4,\ 8,\ 32,\ 1,\ 64)$, with $\rho(\hat{y}) = (5,\ 2,\ 3,\ 4,\ 6,\ 1,\ 7) = \rho(x)$, so y order-preserving matches x. The segments are independent once boundaries (determinate characters) are fixed, and can thus be verified in parallel.

▶ Remark 12. An equivalent way to express the OPPM problem is through permutations. Let $\pi = \rho(x)$ be the rank permutation induced by the determinate pattern x and define $Z[i] = y[\pi^{-1}(i)]$ for each $i \in [1..n]$. Then checking whether x order-preserving matches y amounts to verifying that Z itself forms a non-decreasing sequence (depending on whether ties are allowed). The parallel algorithm can be viewed as partitioning Z into independent segments at the determinate positions.

3.1 Space Complexity

Previous approaches to order-preserving matching on indeterminate strings, such as the one presented in [13], require $\mathcal{O}(mr)$ space, where m is the length of the pattern and r is the maximum number of symbols per indeterminate position.

In contrast, our method does not require any additional space to store the bitwise representation, as all encoding and comparisons can be performed online. Furthermore, assuming that the pattern and the text have the same length, the result of the matching process can be stored in-place, that is, directly within the memory allocated for the text itself. However, to extend the method to handle patterns and texts of different lengths, an additional linear size buffer in the length of the pattern may be required to store the final solution.

4 Parameterized Matching in Indeterminate Strings

Let x be a determinate string drawn from an alphabet Σ^x of size σ^x, partitioned into two disjoint subsets: a fixed alphabet Σ_s and a parameterized alphabet Σ_p^x, such that $\Sigma^x = \Sigma_s \cup \Sigma_p^x$, with $|\Sigma_s| = \sigma_s$ and $|\Sigma_p^x| = \sigma_p^x$.

Let y be an indeterminate string of the same length as x, drawn from an alphabet Σ^y of size σ^y, also partitioned into the same fixed alphabet Σ_s and a parameterized alphabet Σ_p^y, such that $\Sigma^y = \Sigma_s \cup \Sigma_p^y$, with $|\Sigma_p^y| = \sigma_p^y$.

Parameterized matching allows dynamic mappings from characters in x that belong to Σ_p^x to characters in y that belong to Σ_p^y, provided the structural consistency of x is preserved. Thus, to define an injective function from Σ_p^x to Σ_p^y, it is necessary that $\sigma_p^x \geq \sigma_p^y$.

Additionally, we consider that y may include indeterminate positions, where each $y[i]$ represents a subset of Σ^y. To support efficient matching under these conditions, we use a bitwise encoding scheme in which each character is represented by a bitmask. This allows filtering via bitwise operations.

INDETERMINATE PARAMETERIZED MATCHING
Input: A determinate pattern $x \in (\Sigma_s \cup \Sigma_p^x)^n$ and an indeterminate text y of length n, where each $y[i] \subseteq \Sigma_s \cup \Sigma_p^y$.
Output: **true** if there exist an assignment $\hat{y}$ of y and an injective mapping $f : \Sigma_p^x \to \Sigma_p^y$ such that, for every position $i \in [1..n]$, fixed symbols match exactly and parameterized symbols are matched through f; **false** otherwise.

Initially, each parameterized character $c \in \Sigma_p^x$ is assigned a bitmask value: $\texttt{mask}[c] = 2^{\sigma_p^y} - 1$ which permits it to map to any symbol in Σ_p^y. The algorithm proceeds by scanning the string x and refining these bitmasks using the corresponding entries in y. When $y[i]$ is a fixed character, it must match exactly $x[i]$; if not, the match fails immediately. If $y[i]$ is parameterized, the characters in $y[i]$ are encoded as a bitmask, and this is intersected with the current $\texttt{mask}[x[i]]$ via: $\texttt{mask}[x[i]] = \texttt{mask}[x[i]] \wedge \mathcal{M}(y[i])$, removing any characters from the mapping that do not appear in $y[i]$.

A parameterized match is only possible if every character in Σ_p^x retains at least one valid mapping after filtering, i.e., $\texttt{mask}[c] > 0$ for all $c \in \Sigma_p^x$. However, this condition does not guarantee that a valid one-to-one mapping exists.

▶ **Example 13.** Let $\Sigma_p^x = \{\texttt{E}, \texttt{F}\}$ and $\Sigma_p^y = \{\texttt{X}, \texttt{Y}, \texttt{Z}\}$ and $\Sigma_s = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, with $x =$ `a E b F` and $y =$ `a {Y, Z} b {X, Z}`. Initially, all parameterized characters are assigned the bitmask $(111)_2$. The fixed characters `a` and `b` match exactly at their positions. For `E`, the corresponding $y_2 = \{\texttt{Y}, \texttt{Z}\}$ is encoded as $(011)_2$, and the mask is updated via $\texttt{mask}[\texttt{E}] = \texttt{mask}[\texttt{E}] \wedge (011)_2$. Similarly, for `F`, with $y_4 = \{\texttt{X}, \texttt{Z}\}$ encoded as $(101)_2$, the update is $\texttt{mask}[\texttt{F}] = \texttt{mask}[\texttt{F}] \wedge (101)_2$. After this step, both $\texttt{mask}[\texttt{E}]$ and $\texttt{mask}[\texttt{F}]$ remain non-zero. Still, since they share a common mapping candidate belonging to the parameterized-alphabet (e.g., `Z`), uniqueness must be verified.

To enforce uniqueness, we reduce the problem to a bipartite graph matching instance. We construct a bipartite graph $G = (U, W, E)$, where U is the set of parameterized characters in the pattern x, and W is the set of parameterized characters in the text y. We assume $|V| = |U| + |W|$. An edge exists between $u \in U$ and $w \in W$ if the bitmask associated with u contains a 1 at the position corresponding to w, indicating compatibility.

A valid match exists if and only if there is a maximum matching in G that covers all nodes in U, ensuring a unique assignment for each parameterized symbol in the pattern. This can be efficiently computed using the Hopcroft-Karp algorithm, which runs in time $\mathcal{O}(|E|\sqrt{|V|})$ in the general case [14].

In our setting, we have $|U| = \sigma_p^x$ and $|W| = \sigma_p^y$, so the total number of vertices is $|V| = |U| + |W| = \sigma_p^x + \sigma_p^y$. The number of edges $|E|$ is determined by the bitmask encoding and is upper-bounded by $\sigma_p^x \cdot \sigma_p^y$ in the case of full compatibility.

Therefore, the Hopcroft-Karp algorithm runs in time $\mathcal{O}(|E|\sqrt{\sigma_p^x + \sigma_p^y})$ in the general case, and up to $\mathcal{O}((\sigma_p^x \cdot \sigma_p^y)\sqrt{\sigma_p^x + \sigma_p^y})$ in the worst case. Notably, in the fully dense case where G is a complete bipartite graph, the classical bound from Hopcroft and Karp applies, yielding a worst-case complexity of $\mathcal{O}(|V|^{2.5})$ [14].

In practice, however, this step is often significantly faster due to the sparsity of the compatibility graph and the typically low degree of ambiguity. The pseudo-code for this procedure is provided in Algo. 2, which we call the *Parameterized Indeterminate String Matching* (PISM).

We now analyze the computational complexity of the proposed parameterized matching algorithm. The procedure consists of three main steps: filtering the parameterized symbols using bitwise operations, constructing a bipartite graph based on the remaining valid mappings, and computing a maximum matching to enforce one-to-one assignments.

The filtering phase takes $\mathcal{O}(n)$ time due to constant-time bitwise operations per position. Constructing the bipartite graph from the bitmask representations requires $\mathcal{O}(\sigma_p^x \sigma_p^y)$ time in the worst case. The final matching step, performed via the Hopcroft–Karp algorithm, runs in $\mathcal{O}((\sigma_p^x \cdot \sigma_p^y)\sqrt{\sigma_p^x + \sigma_p^y})$ time.

Consequently, the overall time complexity of the algorithm is: $\mathcal{O}(n + (\sigma_p^x \cdot \sigma_p^y)\sqrt{\sigma_p^x + \sigma_p^y})$. The space complexity depends primarily on the representation of the bipartite graph. Using an adjacency matrix yields a worst-case space usage of $\mathcal{O}(\sigma_p^x \sigma_p^y)$. Alternatively, an adjacency list may be employed to reduce space consumption in sparse instances, although the asymptotic bound remains the same.

Algorithm 2 implements parameterized matching between a determinate pattern x and an indeterminate text y under the alphabets $\Sigma^x = \Sigma_s \cup \Sigma_p^x$ and $\Sigma^y = \Sigma_s \cup \Sigma_p^y$. It first initializes masks for all parameterized symbols (lines 2–3). Then, in the presence of indeterminate characters, it updates these masks based on the observed matches between x and y (line 9). Finally, a bipartite graph is constructed from the masks (line 10), and the existence of a full matching (line 11) determines whether x matches y under the parameterized model.

Algorithm 2 Parameterized Matching Algorithm.

```
1:  function PISM(x, y, Σ_p^x, Σ_p^y, Σ_s)
2:      for all c ∈ Σ_p^x do
3:          mask[c] ← all 1s of size |Σ_p^y|
4:      for i = 1 to |x| do
5:          if x[i] ∈ Σ_s then
6:              if y[i] ≠ x[i] then
7:                  return false
8:          else
9:              mask[x[i]] ← mask[x[i]] & BITMASK(y[i])
10:     Construct bipartite graph G = (U, V, E) from mask
11:     return ISFULLMATCHING(G)
```

5 Allowing Mismatches in the Parameterized Matching

The problem of parameterized matching with mismatches [1] has been recently revisited by Saha et al. [20]. In their formulation, given two deterministic strings x and y of equal length n, a weighted bipartite graph $G = (U \cup V, E)$ is constructed, where $U = V = \Sigma_p$. Each edge $(a, b) \in E$ is weighted by the number of positions i such that $x_i = a$ and $y_i = b$. The minimum number of mismatches is then $n_p - \mathrm{mwm}(G)$, where $\mathrm{mwm}(G)$ denotes the maximum weighted matching of G and $n_p = \left|\{\, i \in [1..n] \mid x[i] \in \Sigma_p^x \,\}\right|$. This reduction enables efficient algorithms for approximate parameterized matching, including cases with a bounded number of mismatches. We now extend this framework to the case of a determinate string x and an *indeterminate* string y, formally defined as follows:

INDETERMINATE PARAMETERIZED MATCHING WITH MISMATCHES
Input: A determinate pattern $x \in (\Sigma_s \cup \Sigma_p^x)^n$ and an indeterminate text y of length n, where each $y[i] \subseteq \Sigma_s \cup \Sigma_p^y$.
Output: The minimum number of mismatches over all assignments $\hat{y}$ of y and injective mappings $f : \Sigma_p^x \to \Sigma_p^y$.

▶ **Definition 14** (Parameterized Matching with Mismatches on Indeterminate Texts)**.** *Given a determinate pattern x of length n and an indeterminate text $y = \langle y[1], y[2], \ldots, y[n] \rangle$, where each $y[i] \subseteq \Sigma$, the* parameterized matching with mismatches *problem consists in finding a bijective mapping $f : \Sigma_p^x \to \Sigma_p^y$ and a determinate assignment $\hat{y}$ of y minimizing the number of positions $i \in [1, n]$ such that $x[i] \neq f(\hat{y}[i])$. Equivalently, the minimum number of mismatches is given by $d(x, y) = \min_{f,\,\hat{y}} \left| \{\, i \in [1, n] \mid x[i] \neq f(\hat{y}[i]) \,\} \right|$. The pair $(f, \hat{y})$ achieving this minimum defines an optimal parameterized alignment between x and y.*

In the exact setting, we initialized each mask with all ones and progressively refined it via bitwise AND operations, thereby filtering out incompatible candidates until only consistent bijections remained. To allow mismatches, this logic is inverted: since each $y[i]$ is a *set* of possible symbols, the goal is to *accumulate all compatibilities* rather than restrict them. We therefore initialize every parameterized symbol with an all-zeros mask and update it by bitwise OR: $\texttt{mask}[x[i]] = \texttt{mask}[x[i]] \vee \mathcal{M}(y[i])$, so that all candidates compatible with $y[i]$ are preserved. The OR-masks specify adjacency in the bipartite graph but not edge multiplicities. To capture frequencies, we introduce an auxiliary array μ of size $|\Sigma_p|$ such that $\mu[c] = \left| \{\, i \mid c \in y[i] \,\} \right|$, counting how many times c appears in the subsets of y. Consequently, for each pair $(a, b) \in \Sigma_p^x \times \Sigma_p^y$, the weight of edge (a, b) equals the number of positions i with $x[i] = a$ and $b \in y[i]$. This generalizes the deterministic case, where all edges had unit weight. The resulting bipartite graph is thus defined by OR-masks (adjacency) and the μ array (weights). The maximum weighted matching of this graph again determines the optimal alignment, and the minimum number of mismatches is $n_p - \mathrm{mwm}(G)$.

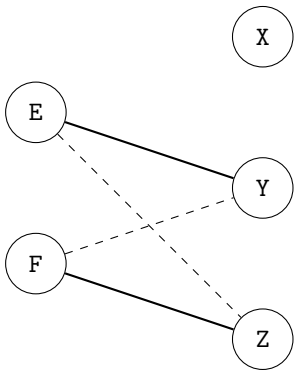
▶ **Example 15.** Let $\Sigma_p^x = \{\texttt{E}, \texttt{F}\}$ and $\Sigma_p^y = \{\texttt{X}, \texttt{Y}, \texttt{Z}\}$, and let $\Sigma_s = \{\texttt{a}, \texttt{b}, \texttt{c}\}$. Consider the determinate pattern $x = \texttt{a E b F E}$ and the indeterminate text $y =$ `a {Y} b {Y, Z} {Z}`. In this example, the number of parameterized positions is $n_p = 3$, corresponding to positions 2, 4, and 5 of both strings.

In contrast to the exact setting, we initialize all masks to $(000)_2$ and update them using bitwise OR. Encoding $\texttt{X}, \texttt{Y}, \texttt{Z}$ as $(100)_2$, $(010)_2$, and $(001)_2$, respectively, the sets $\{\texttt{Y}\}$, $\{\texttt{Y}, \texttt{Z}\}$, and $\{\texttt{Z}\}$ are encoded as $(010)_2$, $(011)_2$, and $(001)_2$. Thus, we obtain

$$\texttt{mask}[E] = (000)_2 \vee (010)_2 \vee (001)_2 = (011)_2,$$
$$\texttt{mask}[F] = (000)_2 \vee (011)_2 = (011)_2.$$

Hence, both `E` and `F` are compatible with $\{\texttt{Y}, \texttt{Z}\}$.

Next, we compute the edge weights of the bipartite graph. Each edge (a, b) is weighted by the number of positions i such that $x[i] = a$ and $b \in y[i]$. Therefore, we obtain the bipartite graph shown below, where the left side corresponds to Σ_p^x and the right side to Σ_p^y. All edges have unit weight.



The thick edges indicate one possible maximum weighted matching, while dashed edges represent alternative compatible assignments.

A maximum weighted matching selects two disjoint edges, for instance $(\mathtt{E}, \mathtt{Y})$ and $(\mathtt{F}, \mathtt{Z})$ with total weight 2. Since $n_p = 3$, the minimum number of mismatches is $d(x, y) = n_p - \mathrm{mwm}(G) = 3 - 2 = 1$. We now verify this bound explicitly.

The pattern contains three non-static positions, namely `E` at positions 2 and 5, and `F` at position 4. Under the bijection $f(\mathtt{E}) = \mathtt{Y}, \quad f(\mathtt{F}) = \mathtt{Z}$, position 2 matches, position 4 matches, but position 5 does not, since $y[5] = \{\mathtt{Z}\}$ does not contain `Y`. Similarly, under the bijection $f(\mathtt{E}) = \mathtt{Z}, \quad f(\mathtt{F}) = \mathtt{Y}$, position 5 matches, position 4 matches, but position 2 does not, since $y[2] = \{\mathtt{Y}\}$ does not contain `Z`. Therefore, exactly one mismatch is unavoidable, and $d(x, y) = 1$.

This example shows how OR-masks encode local compatibility, while the weighted matching enforces global consistency through a bijection, leading to an optimal alignment.

▶ **Lemma 16.** *The OR-mask and frequency array μ for an indeterminate text y of length n can be constructed in $O(n)$ time.*

Since only the edge weights differ while the graph structure is preserved, the algorithm of Saha et al. [20] remains directly applicable, with time complexity $\mathcal{O}(\sigma^2 n \log n + n\sigma^2\sqrt{\sigma}\,\log(n\sigma))$. Constructing the OR-masks and μ requires only $\mathcal{O}(n)$ preprocessing, so the dominant cost remains the FFT-based polynomial multiplication [4] and the maximum weighted bipartite matching, yielding the same asymptotic complexity as in the deterministic case.

6 Experimental Results

The proposed algorithms were implemented in C.[1] Experiments were conducted on a MacBook Pro equipped with a 2.7 GHz Intel Core i7-8559U (4 cores), with 16 GB RAM. All algorithms were compiled using the `-O3` optimization flag.

Performance was measured in terms of search execution time (milliseconds) on randomly generated texts. We report two complementary experiments, shown as the two plots of Fig. 1.

Within the context of *order-preserving pattern matching (OPPM)*, we compare two approaches: `OLD`, referring to the algorithm proposed by Henriques et al. [13]; and `NEW`, representing the bitwise-compatible method introduced in this work. For the *parameterized string matching* setting, we evaluate the `PISM` algorithm, designed to support efficient bit-parallelism in handling variable symbol classes and the *Mismatches Parameterized Indeterminate String Matching* (MPISM)[2]. In the left plot, the x–axis varies according to the specific setting. For OPPM, it represents the number of *uncertain characters per position*, denoted by r. For PISM, it corresponds to the size of the parameterized alphabet of the text, σ_p^y, which captures the diversity of symbol classes. For MPISM, we fix three different sizes of the parameterized alphabet $|\Sigma_p| = \{2, 8, 32\}$ (small, medium and large) and instead vary the number of allowed mismatches in the input to assess performance. Additionally, in the right plot, all parameters are fixed and only the text length n varies, taking values between 10^4 and 10^7. The results in Fig. 1 show a substantial and consistent speedup of `NEW` over `OLD`

[1] Source code available at `https://github.com/marfra99x/SCER-ISM`.

[2] Although MPISM is described in the paper using a fast FFT-based procedure, in our experiments we employ the Hungarian algorithm [16] for maximum weighted bipartite matching, as it offers a simpler and more straightforward implementation. Nonetheless, the FFT-based approach outlined in the paper could yield faster results.

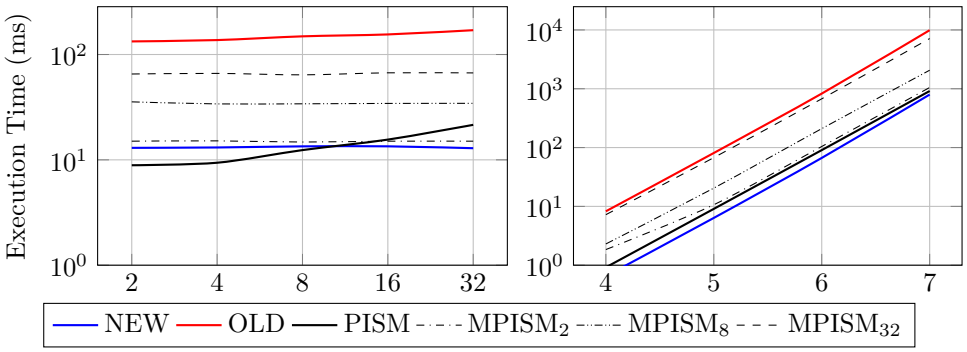


Figure 1 Average execution time (in milliseconds, log-scale on the y-axis) of the compared algorithms on randomly generated texts. **Left plot:** execution time as the main input parameter varies. For `NEW` and `OLD` (OPPM), the x-axis shows the number of uncertain characters per position r (values 2–32). For `PISM`, it shows the size of the parameterized alphabet of the text σ_p^y. For `MPISM`, results are reported for $|\Sigma_p| \in \{2, 8, 32\}$ while varying the number of allowed mismatches. **Right plot:** execution time as a function of the text length n (from 10^4 to 10^7), with all other parameters fixed. Overall, `NEW` consistently outperforms `OLD`, while `PISM` and `MPISM` highlight the effect of increasing alphabet size and mismatch allowance on execution time.

across all tested regimes. On average, our method outperforms the prior state-of-the-art by over 92%, confirming the benefits of the bit-parallel encoding. `PISM` maintains competitive performance while `MPISM` displays the expected growth as σ_p^y and n increase, reflecting the underlying matching complexities.

These results confirm the effectiveness of our algorithms not only in theoretical terms but also in practice, particularly for large-scale texts where traditional set-based comparisons become a performance bottleneck. The combination of sum-free sequence encoding and greedy selection strategies leads to measurable improvements in runtime performance.

7 Conclusions and Future Work

In this work, we introduced efficient algorithms for matching a determinate pattern against an indeterminate text under two notable substring consistent equivalence relations (SCERs): order-preserving and parameterized matching. By leveraging a bitwise-compatible encoding based on sum-free sequences, our approach enables constant-time character comparisons and compact set representations. The resulting techniques achieve an optimal time complexity of $O(n)$ for order-preserving matching and $\mathcal{O}(n + (\sigma_p^x \cdot \sigma_p^y)\sqrt{\sigma_p^x + \sigma_p^y})$ for parameterized matching, where σ_p^x and σ_p^y denote the number of distinct parameterized symbols, in the pattern and text, respectively.

The efficiency of our approach can be further enhanced through pre-sampling strategies. Sampling techniques have proven effective in reducing redundant search spaces and filtering unpromising regions in various exact and approximate matching tasks [8]. Applying such techniques in the context of indeterminate matching may enable early pruning and tighter complexity bounds, especially in large-scale or high-ambiguity datasets.

In addition, we extended the parameterized framework to handle mismatches in the presence of indeterminate positions, we showed that the construction naturally reduces to the same weighted bipartite graph formulation of Saha et al. [20]. As a result, their FFT-based computation and matching algorithm carry over unchanged, with only a linear preprocessing overhead. This demonstrates that approximate parameterized matching can also be tackled

efficiently in the indeterminate setting, further broadening the scope of tractable problems under SCERs. Future directions include extending this asymmetric framework to other SCER models, integrating adaptive sampling mechanisms, and optimizing for real-time or streaming contexts where input uncertainty dynamically evolves.

References

1 Alberto Apostolico, Peter Erdős, and Moshe Lewenstein. Parameterized matching with mismatches. *J. Discrete Algorithms*, 5:135–140, 2007. doi:10.1016/j.jda.2006.03.014.

2 Brenda S. Baker. A theory of parameterized pattern matching: algorithms and applications. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 71–80. ACM, 1993. doi:10.1145/167088.167115.

3 R.S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977. doi:10.1016/0020-0190(77)90017-5.

4 Elbert Oran Brigham. *The Fast Fourier Transform*. Prentice-Hall, Englewood Cliffs, N.J., 1974.

5 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

6 Hossein Dehghani, Thierry Lecroq, Neerja Mhaskar, and William F. Smyth. Practical KMP/BM style pattern-matching on indeterminate strings. *Discrete Applied Mathematics*, 370:22–33, 2025. doi:10.1016/J.DAM.2025.02.032.

7 Simone Faro, Dominik Köppl, Thierry Lecroq, and Francesco Pio Marino. marfra99x/SCER-ISM. Software, swhId: swh:1:dir:c33fa1f2fdf35f4030601b03c54ad27d5931dfd0 (visited on 2026-05-27). URL: https://github.com/marfra99x/SCER-ISM, doi:10.4230/artifacts.26165.

8 Simone Faro, Francesco Pio Marino, and Arianna Pavone. Efficient online string matching based on characters distance text sampling. *Algorithmica*, 82(11):3390–3412, 2020. doi:10.1007/S00453-020-00732-4.

9 Simone Faro, Francesco Pio Marino, and Arianna Pavone. Improved characters distance sampling for online and offline text searching. *Theoretical Computer Science*, 946:113684, 2023. doi:10.1016/J.TCS.2022.12.034.

10 Simone Faro, Francesco Pio Marino, Arianna Pavone, and Antonio Scardace. Towards an efficient text sampling approach for exact and approximate matching. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021*, pages 75–89. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021. URL: http://www.stringology.org/event/2021/p07.html.

11 Paweł Gawrychowski, Samah Ghazawi, and Gad M. Landau. On indeterminate strings matching. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, volume 161 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.14.

12 Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

13 Rui Henriques, Alexandre P. Francisco, Luís M. S. Russo, and Hideo Bannai. Order-preserving pattern matching indeterminate strings. In Gonzalo Navarro, David Sankoff, and Binhai Zhu, editors, *Annual Symposium on Combinatorial Pattern Matching, CPM 2018, July 2-4, 2018 - Qingdao, China*, volume 105 of *LIPIcs*, pages 2:1–2:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.CPM.2018.2.

14 John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. doi:10.1137/0202019.

15 Tomohiro I, Shunsuke Inenaga, and Masayuki Takeda. Palindrome pattern matching. *Theoretical Computer Science*, 483:162–170, 2013. doi:10.1016/J.TCS.2012.01.047.

16 Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. doi:10.1002/nav.3800020109.

17 Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002. doi:10.1017/CBO9781316135228.

18 Petr Procházka and Jan Holub. On-line searching in IUPAC nucleotide sequences. In Elisabetta De Maria, Ana L. N. Fred, and Hugo Gamboa, editors, *Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2019) - Volume 3: BIOINFORMATICS, Prague, Czech Republic, February 22-24, 2019*, pages 66–77. SciTePress, 2019. doi:10.5220/0007382900660077.

19 M. Sohel Rahman, Costas S. Iliopoulos, and Laurent Mouchard. Pattern matching in degenerate DNA/RNA sequences. In M. Kaykobad and Md. Saidur Rahman, editors, *Workshop on Algorithms and Computation 2007 - Proceedings of First WALCOM, 12 February 2007, Dhaka, Bangladesh*, pages 109–120. Bangladesh Academy of Sciences (BAS), 2007.

20 Apurba Saha, Iftekhar Hakim Kaowsar, Mahdi Hasnat Siyam, and M. Sohel Rahman. Algorithms for parameterized string matching with mismatches. *arXiv*, 2024. doi:10.48550/arXiv.2412.00222.