mathematics

MDPI

Article

Order-Preserving Pattern Matching with Partition

Joong Chae Na [1,†], Youngjoon Kim [2,†], Seokchul Kang [2] and Jeong Seop Sim [2,*]

[1] Department of Computer Science and Engineering, Sejong University, Seoul 05006, Republic of Korea; jcna@sejong.ac.kr
[2] Department of Computer Engineering, Inha University, Incheon 22212, Republic of Korea; yjkim12130@gmail.com (Y.K.); stoneiron02@inha.edu (S.K.)
* Correspondence: jssim@inha.ac.kr
† These authors contributed equally to this work.

Abstract: Order-preserving pattern matching, which considers the relative orders of strings, can be applied to time-series data analysis. To perform a more meaningful analysis of time-series data, approximate criteria for the order-isomorphism are necessary, considering diverse types of errors. In this paper, we introduce a novel approximation criterion for the order-isomorphism, called the partitioned order-isomorphism. We then propose an efficient $O(n + sort(m))$-time algorithm for the order-preserving pattern matching problem considering the criterion of partition. A comparative experiment demonstrates that the proposed algorithm is more effective than the exact order-preserving pattern matching algorithm.

Keywords: order-preserving pattern matching; approximate order-preserving pattern matching; pattern matching with partition; string matching

MSC: 68W32

1. Introduction

Two strings of the same length are *order-isomorphic* when they have the same relative orders. For example, strings $X = (5, 9, 7)$ and $Y = (23, 51, 47)$ are order-isomorphic because the relative order of individual characters in both strings is the same as $(1, 3, 2)$. Given a text T of length n and a pattern P of length m, the order-preserving pattern matching (OPPM) problem is finding all substrings of T which are order-isomorphic to P. For example, given a text $T = (10, 23, 5, 3, 30, 8, 27, 15, 25, 12, 6, 17, 11, 4)$ and $P = (1, 8, 3, 7, 5, 6, 4, 2)$, P occurs at the fourth position of T because the underlined substring $(3, 30, 8, 27, 15, 25, 12, 6)$ of T is order-isomorphic to P (see Figure 1). The OPPM problem can be solved in $O(n + sort(m))$ time [1–3] based on the KMP algorithm [4], where $sort(m)$ is time for sorting P. Practical algorithms for the OPPM problem using the Horspool approach [5] and a filtration method have been studied [6–9]. Additionally, various related studies have been conducted, including research on order-preserving string regularities [10,11], OPPM with multiple patterns [1,12], parallelism in OPPM [12,13], order-preserving suffix trees [14], and OPPM in a tree and directed acyclic graphs [15].

Since, in OPPM, the relative order of characters is more important than their absolute values, it can be effectively applied to analyze various types of time-series data, such as stock prices, energy consumption, annual temperature variance, and music melodies [7,16–18]. For example, when there are two stocks in the same industry but with different face values, the actual price fluctuations of the two stocks over the same period may differ, but the pattern of change can be similar. In such a case, using order-isomorphism allows for detecting the same or similar patterns. However, it is not always easy to apply OPPM algorithms directly due to various factors such as differences in the measurement intervals of time-series data, noise occurring in the measurement process, errors in the data transformation process, etc. Additionally, even if this error occurs at just one position, it can

check for updates

Citation: Na, J.C.; Kim, Y.; Kang, S.; Sim, J.S. Order-Preserving Pattern Matching with Partition. *Mathematics* **2024**, *12*, 3381. https://doi.org/10.3390/math12213381

Academic Editors: Gang Hu, Lan Cheng and Guanqiu Qi

Received: 29 September 2024
Revised: 24 October 2024
Accepted: 28 October 2024
Published: 29 October 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

affect the overall relative order, which makes it difficult to detect in the reference data. For example, if the measurements of a certain time-series data are $(3, 30, 8, 27, 15, 25, 12, 6)$, the relative order of each measurement is $(1, 8, 3, 7, 5, 6, 4, 2)$. If the second value (character), 30, is changed to 24 due to a measurement or recording error, the relative order of each measurement changes significantly to $(1, 6, 3, 8, 5, 7, 4, 2)$, making it difficult to detect in the reference data or possibly leading to the detection of an incorrect position. Therefore, to perform a more meaningful pattern search, prediction, and analysis of time-series data, it is necessary to consider diverse types of errors.

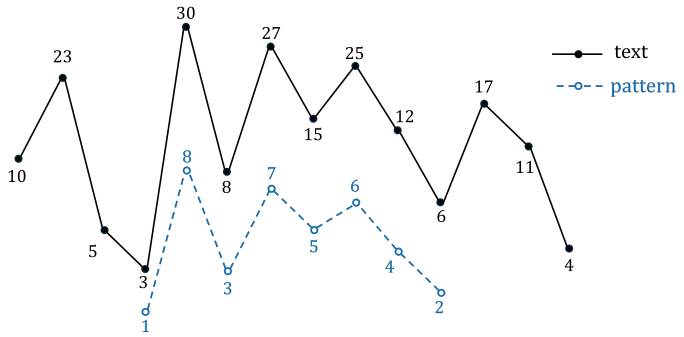


Figure 1. An OPPM example for $T = (10, 23, 5, 3, 30, 8, 27, 15, 25, 12, 6, 17, 11, 4)$ and $P = (1, 8, 3, 7, 5, 6, 4, 2)$.

Recently, approximate OPPM problems that allow errors in matching have been studied. Gawrychowski and Uznański [17] studied the OPPM problem with k-mismatches. Their algorithm finds occurrences of a pattern within a text where the relative order of elements is preserved, allowing up to k mismatches, with a time complexity of $O(n(\log\log m + k\log\log k))$. Russo et al. [18] studied OPPM problems on indeterminate strings, which considers the relative order of elements, even when some elements are not precisely determined. They presented an $O(rm\log r)$-time algorithm, where r is a bound on the number of uncertain characters per position. Kim et al. [19] defined a scaling approximation and presented algorithms for the OPPM problem with scaling, which considers the relative orders of cusps and the scale of lengths of strings between them. They presented an $O(n + m\log m)$-time algorithm to solve the OPPM problem with scaling, which is the same time bound as the best known exact OPPM algorithm.

In some cases in OPPM, it might be more effective to compare two strings part by part rather than as whole strings. Consider two strings $x = (1, 8, 3, 7, 5, 6, 4, 2)$ and $y = (3, 30, 8, 27, 15, 25, 12, 6)$ for example. The relative orders of these two strings are the same with $(1, 8, 3, 7, 5, 6, 4, 2)$. Assume that the second character of y is changed to 24 due to some error, as in the previous example. That is, let us assume y is changed to $y' = (3, 24, 8, 27, 15, 25, 12, 6)$. Then the relative orders of x and y' become quite different, with $(1, 8, 3, 7, 5, 6, 4, 2)$ and $(1, 6, 3, 8, 5, 7, 4, 2)$, respectively. Despite their overall differences in order, however, when x and y' are divided at the third position, the orders of the two divided parts of x and y' are the same with $(1, 3, 2)$ and $(5, 3, 4, 2, 1)$, respectively, i.e., the two respective parts match each other. See Figure 2.

The above approach can be applied to plagiarism detection in music melodies. OPPM can be used to detect parts of a full melody that exactly match a specific short melody to determine plagiarism. If one of the notes in the detected section is modified to a similar but different note, it may sound similar when played but might not be detected by OPPM. However, if the section is divided into two parts and OPPM is performed separately, the likelihood of detection increases. It can also be applied to stock price prediction. If the price movement of a particular stock over a certain period resembles the movement patterns of other stocks in the past, future changes can be predicted to some extent. However, as these patterns become longer, it becomes more challenging to find exact matches using OPPM. By dividing the pattern into two parts and performing OPPM separately, the chances of finding matching sections increase.

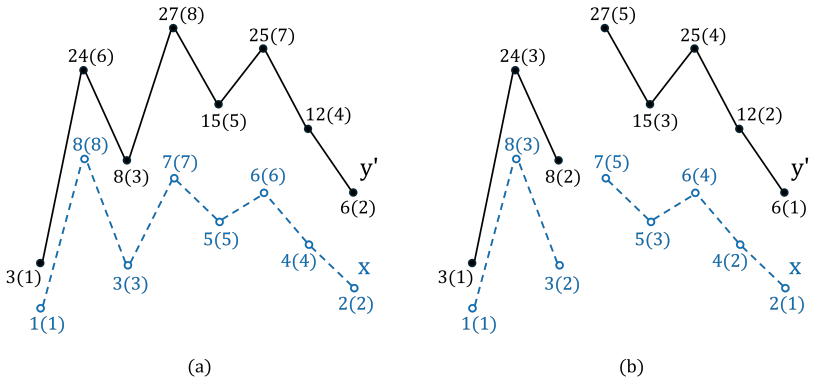


Figure 2. Illustration of the partitioned order-isomorphism (the numbers in parentheses represent the relative orders). For $x = (1, 8, 3, 7, 5, 6, 4, 2)$ and $y' = (3, 24, 8, 27, 15, 25, 12, 6)$, (**a**) x and y' are not order-isomorphic, but (**b**) when dividing them at the third position, two divided parts of x and y' match each other.

Our contributions are as follows:

- We introduce a novel approximation criterion for the order-isomorphism, called the partitioned order-isomorphism for the first time. In OPPM, unlike classical pattern matching, a change in just one character can significantly alter the overall ranking of the entire string. As a result, parts that could not be found when the two strings were compared directly may be detected when the strings are divided and compared in two parts.
- We present an efficient algorithm for the problem using the Z-function [10,20]. Even with the consideration of the approximation criterion, our algorithm achieves a time complexity of $O(n + sort(m))$, which is equivalent to the best-known exact OPPM algorithm.
- We demonstrate the effectiveness of our algorithm through comparative experiments using three types of real time-series data. The experiments are conducted by varying the lengths of patterns for each data set, which demonstrates that our algorithm is more effective than the OPPM algorithm.

This paper is organized as follows. In Section 2, we give some preliminary works. In Section 3, we first define the partitioned order-isomorphism and the OPPM problem with partition formally. And then we give an efficient algorithm for the OPPM problem with partition. In Section 4, we present experimental comparisons between the algorithm introduced in [1] and the algorithm proposed in this study. In Section 5, we conclude this research.

2. Related Works

Let Σ denote the set of characters where two characters can be compared in constant time. A string S is a sequence of characters drawn from Σ. We denote by $|S|$ the length of S and by $S[i]$ the ith character of S $(1 \leq i \leq |S|)$. The substring $S[i] \cdots S[j]$ $(1 \leq i, j \leq |S|)$ is denoted by $S[i..j]$, and $S[i..j]$ is an empty string when $i > j$. Substrings $S[1..i]$ and $S[i..|S|]$ $(1 \leq i \leq |S|)$ are called a prefix and a suffix of S, respectively. The concatenation of two strings X and Y is denoted by XY. For convenience, we assume that the characters in a string are all distinct as in [1].

The relative order (or rank) of a character $S[i]$ in a string S is defined as the number of characters in S that are smaller than or equal to $S[i]$, including itself. A natural way to represent the order relations of the characters in S is to enumerate their relative orders in S, which is called the *natural representation* of the order relations. For example, the natural representation of the string $(54, 12, 38, 69, 45, 22)$ is $(5, 1, 3, 6, 4, 2)$. Although the natural representation is intuitive and easy to understand, it is not suitable for pattern matching [1]. The *prefix representation* is an alternative way to represent order relations, where the relative

order of a character $S[i]$ is considered within $S[1..i]$ rather than in the entire string S. For example, the prefix representation of the string $(54, 12, 38, 69, 45, 22)$ is $(1, 1, 2, 4, 3, 2)$. It has been shown that there is a one-to-one mapping between the natural representation and the prefix representation [1].

We formally define the order-isomorphism of two strings. Two strings X and Y of the same length are *order-isomorphic*, denoted by $X \approx Y$, when $X[i] < X[j] \Leftrightarrow Y[i] < Y[j]$ for all $1 \leq i, j \leq |X|$. Obviously, the following lemma and corollary hold by the definition of the order-isomorphism.

Lemma 1. *For two strings X and Y of the same length, if X and Y are order-isomorphic, then substrings $X[i..j]$ and $Y[i..j]$ (for all $1 \leq i \leq j \leq |X|$) are also order-isomorphic.*

Corollary 1. *For two strings X and Y of the same length, if there exist substrings $X[i..j]$ and $Y[i..j]$ ($1 \leq i \leq j \leq |X|$) that are not order-isomorphic, then X and Y are not order-isomorphic.*

The order-isomorphism can be efficiently determined using the nearest neighbor representation [1–3,6], which is represented by the location tables $LMax_X$ and $LMin_X$.

$$LMax_X[i] = \begin{cases} j & \text{if } X[j] = \max\{X[k] : X[k] < X[i],\ 1 \leq k \leq i-1\} \\ -1 & \text{if there is no such } j \end{cases}$$

$$LMin_X[i] = \begin{cases} j & \text{if } X[j] = \min\{X[k] : X[k] > X[i],\ 1 \leq k \leq i-1\} \\ -1 & \text{if there is no such } j \end{cases}$$

That is, $LMax_X[i]$ and $LMin_X[i]$ are the nearest neighbors of $X[i]$ in the sorted sequence of characters in $X[1..i]$. Table 1 shows $LMax_X$ and $LMin_X$ for $X = (54, 12, 38, 69, 45, 22)$. The location tables can be computed in $O(|X| \log |X|)$ time using a sorting algorithm or an order-statistic tree. Using the location tables for X, we can determine the order-isomorphism of X and Y in $O(|X|)$ time by checking the following inequality for every position $1 \leq i \leq |X|$:

$$Y[LMax_X[i]] < Y[i] < Y[LMin_X[i]] \tag{1}$$

(if $LMax_X[i]$ or $LMin_X[i]$ is equal to -1, we assume the respective inequality is true). Moreover, we can find the longest prefix β of Y that is order-isomorphic to a prefix of X in $O(|\beta|)$ time by checking the above inequality in increasing order of i until we reach a position where the inequality is false (Algorithm 1). We call β the longest order-isomorphic prefix (loip) of X and Y, which is called the longest common prefix (lcp) in classical matching.

Table 1. Location tables and Z-function for $X = (54, 12, 38, 69, 45, 22)$.

i	1	2	3	4	5	6
$X[i]$	54	12	38	69	45	22
$LMax_X[i]$	-1	-1	2	1	3	2
$LMin_X[i]$	-1	1	1	-1	1	3
Z_X	6	1	1	2	2	1

For a string S, the Z-function $Z_S(i)$ ($1 \leq i \leq |S|$) represents the length of the longest prefix of $S[i..|S|]$ which is order-isomorphic to a prefix of S, i.e., the loip of $S[i..|S|]$ and $S[1..|S| - i + 1]$ [10]. It is a modified version for order-isomorphism of the Z-function introduced in [20] to solve classical string matching problems. With the nearest neighbor representation of S, Z_S can be computed in $O(|S|)$ time.

Algorithm 1 Computing loip

Input: two strings $X[1..m]$ and $Y[1..m]$, and an integer $k \geq 0$ such that $X[1..k] \approx Y[1..k]$.
output: length of the loip of X and Y.

```
1: procedure CompLoip (X, Y, k)
2:     for i ← k + 1 to m do                    ▷ Assume LMax_X and LMin_X are given.
3:         if LMax_X[i] ≠ −1 and Y[LMax_X[i]] ≥ Y[i] then
4:             return i − 1
5:         end if
6:         if LMin_X[i] ≠ −1 and Y[i] ≥ Y[LMin_X[i]] then
7:             return i − 1
8:         end if
9:     end for
10:    return m
11: end procedure
```

3. Order-Preserving Pattern Matching (OPPM) with Partition

In this section, we define the OPPM problem with partition and present an efficient algorithm for solving the problem.

3.1. Problem Definition

We define the partitioned order-isomorphism. Let us consider two strings $X = (5, 11, 18, 7, 3, 9)$ and $Y = (1, 2, 3, 5, 4, 6)$, which are not order-isomorphic. When we partition X (resp. Y) into two substrings $X[1..3]$ and $X[4..6]$ (resp. $Y[1..3]$ and $Y[4..6]$), the partitioned substrings of X are order-isomorphic to those of Y, i.e., $X[1..3] \approx Y[1..3]$ and $X[4..6] \approx Y[4..6]$. Then, we say that X is partitioned order-isomorphic to Y. That is, the *partitioned order-isomorphism* is formally defined as follows.

Definition 1. *For two strings X and Y of length m and a position t ($0 \leq t \leq m$), if $X[1..t] \approx Y[1..t]$ and $X[t+1..m] \approx Y[t+1..m]$, then X and Y are partitioned order-isomorphic at t, denoted by $X \approx^t_{\vdash} Y$. If there exists any position t such that $X \approx^t_{\vdash} Y$, then X and Y are partitioned order-isomorphic.*

In the example above, $X \approx^3_{\vdash} Y$ since $X[1..3] \approx Y[1..3]$ and $X[4..6] \approx Y[4..6]$. However, X is not partitioned order-isomorphic to $Z = (1, 2, 3, 4, 5, 6)$ because there exists no t such that $X \approx^t_{\vdash} Z$. We call position t a *partition position*. (Although we consider the cases when a string is partitioned into two substrings, Definition 1 can be extended to the cases of partitioning into more than two substrings.)

The following lemma shows that partition positions are consecutive.

Lemma 2. *For two strings X and Y of length m, let l_{max} be the largest l such that $X[1..l] \approx Y[1..l]$ ($1 \leq l \leq m$). Similarly, let r_{min} be the smallest r such that $X[r..m] \approx Y[r..m]$ ($1 \leq r \leq m$). Then, $X \approx^t_{\vdash} Y$ if and only if $r_{min} - 1 \leq t \leq l_{max}$.*

Proof. Note that l_{max} and r_{min} always exist since two strings of length 1 are order-isomorphic.
($\Rightarrow$) We show by contradiction that if $X \approx^t_{\vdash} Y$, then $r_{min} - 1 \leq t \leq l_{max}$. Suppose that $X \approx^t_{\vdash} Y$ for some t such that $t < r_{min} - 1$ or $t > l_{max}$. Consider the case when $t > l_{max}$. Since $X \approx^t_{\vdash} Y$, $X[1..t] \approx Y[1..t]$, which contradicts the definition of l_{max}. Similarly, a contradiction occurs when $t < r_{min} - 1$.
($\Leftarrow$) We show that $X \approx^t_{\vdash} Y$ if $r_{min} - 1 \leq t \leq l_{max}$. Since $X[1..l_{max}] \approx Y[1..l_{max}]$ and $t \leq l_{max}$, $X[1..t] \approx Y[1..t]$ by Lemma 1. Similarly, $X[t+1..m] \approx Y[t+1..m]$. Therefore, $X \approx^t_{\vdash} Y$. □

Corollary 2. *If $r_{min} - 1 > l_{max}$, then X and Y are not partitioned order-isomorphic.*

The *OPPM problem with partition* is to find all substrings of a text T that are partitioned order-isomorphic to a pattern P, formally defined as follows.

Problem 1. *The order-preserving pattern matching (OPPM) problem with partition.*
Input: A text T of length n and a pattern P of length m ($n \geq m$).
Output: Every pair $(i, [a : b])$ where i is a position in T ($1 \leq i \leq n - m + 1$) and $[a : b]$ is a position range in P ($0 \leq a \leq b \leq m$) such that $T[i..i + m - 1] \approx_{\vdash}^{t} P$ if and only if $a \leq t \leq b$.

For example, given $T = (13, 92, 34, 88, 77, 63, 37, 40, 70, 54, 35, 24, 50)$ and $P = (54, 12, 38, 69, 45, 22)$, the output is $(2, [3 : 3])$ and $(6, [2 : 5])$. See Table 2, which shows l_{max} and r_{min} for P and $T[i..i + m - 1]$ $(1 \leq i \leq n - m + 1)$.

Table 2. Given $T[1..13] = (13, 92, 34, 88, 77, 63, 37, 40, 70, 54, 35, 24, 50)$ and $P[1..6] = (54, 12, 38, 69, 45, 22)$, the values of l_{max} and r_{min} for P and $T[i..i + 5]$ $(1 \leq i \leq 8)$.

i	1	2	3	4	5	6	7	8
l_{max} $(= L_i)$	1	3	1	2	2	5	1	1
r_{min} $(= 7 - R_i)$	4	4	6	6	5	3	4	6
occurrence	-	$(2, [3 : 3])$	-	-	-	$(6, [2 : 5])$	-	-

3.2. An Algorithm for the OPPM Problem with Partition

Let T_i denote the substring of T of length m starting at position i, i.e., $T_i = T[i..i + m - 1]$ $(1 \leq i \leq n - m + 1)$. Let L_i and R_i be the lengths of the longest prefix and the longest suffix of T_i that are order-isomorphic to a prefix and a suffix of P, respectively. When we are given L_i's and R_i's for all $1 \leq i \leq n - m + 1$, we can solve Problem 1 in $O(n)$ time using Lemma 2 and Corollary 2 (note that $l_{max} = L_i$ and $r_{min} - 1 = m - R_i$ for T_i and P). See Algorithm 2.

Now we focus on the problem of computing all L_i's. (We omit the details of computing R_i's since it can be computed using the same way by considering reversed strings of T and P.) Note that L_i is the length of the loip of T_i and P. So we can compute each L_i in $O(m)$ time using the nearest neighbor representation of P (Algorithm 1), resulting in a total time complexity of $O(nm + m \log m)$, where $O(m \log m)$ represents the time required for computing the nearest neighbor representation $LMax_P$ and $LMin_P$ of P. More efficiently, we can compute all L_i's in $O((n + m) \log(n + m))$ time by concatenating P and T (say $S = PT$) and computing Z_S.

Algorithm 2 OPPM with partition

Input: a text $T[1..n]$ and a pattern $P[1..m]$
Output: all pairs $(i, [a : b])$ representing occurrences of Problem 1.

```
1: procedure OPPMwPartition(T, P)
2:     compute L_i and R_i for 1 ≤ i ≤ n − m + 1 using Algorithm 3
3:     for i ← 1 to n − m + 1 do
4:         if m − R_i ≤ L_i then
5:             report (i, [m − R_i : L_i]) as an occurrence
6:         end if
7:     end for
8: end procedure
```

However, we can improve the time complexity to $O(n + m \log m)$ time by computing Z_P and all L_i's separately. We first preprocess P for computing $LMax_P$, $LMin_P$, and Z_P. Then, we compute L_i in increasing order of i using Z_P and L_j $(1 \leq j < i)$, which is similar to the Z-algorithm in [20]. For any position k $(1 \leq k \leq n)$, the L-box starting at k is defined as the range $(k, k + L_k - 1)$ of length L_k. (Note that $L_k > 0$ because the first characters of two strings are trivially order-isomorphic.) For each i $(1 \leq i \leq n)$, let (l_i, r_i) be the L-box whose

end position is the rightmost among the L-boxes whose start position is less than or equal to i. (If more than one L-box has the rightmost end position, we may choose any L-box.)

Algorithm 3 Computing L_i's

input: a text $T[1..n]$ and a pattern $P[1..m]$
Output: all pairs $(i, [a : b])$ representing occurrences of Problem 1.

```
1:  procedure ComputeL(T, P)
2:      Compute LMax_P, LMin_P, and Z_P for P                ▷ Preprocessing P
3:      (l, r) ← (0, 0)                                      ▷ Initialize L-box (l_0, r_0)
4:      for i ← 1 to n − m+1 do
5:          if i > r then                                    ▷ Case 1
6:              L_i ← CompLoip(P, T[i..i + m − 1], 0)
7:              (l, r) ← (i, i + L_i − 1)                    ▷ Set (l_i, r_i) as the new L-box
8:          else                                             ▷ Case 2
9:              z ← Z_P(i − l + 1)
10:             if z = |T[i..r]| then                        ▷ Case 2(a)
11:                 L_i ← CompLoip(P, T[i..i + m − 1], z)
12:                 (l, r) ← (i, i + L_i − 1)                ▷ Set (l_i, r_i) as the new L-box
13:             else                                         ▷ Case 2(b) and 2(c)
14:                 L_i = min(z, |T[i..r]|)
15:                                  ▷ (l, r) remains unchanged, i.e., (l_i, r_i) = (l_{i−1}, r_{i−1})
16:             end if
17:         end if
18:     end for
19: end procedure
```

Algorithm 3 shows the pseudo-code of our algorithm for computing L_i's. At each iteration i ($1 \le i \le n - m + 1$), we compute L_i and (l_i, r_i) using (l_{i-1}, r_{i-1}). Initially, we set $(l_0, r_0) = (0, 0)$. The details of each iteration i are as follows:

1. If $i > r_{i-1}$, then compute L_i by explicitly checking the inequality (1) for T_i ($= T[i..i + m - 1]$) and P character by character, starting from the first position and continuing until either the inequality becomes false or the check reaches the last position. Since $i > r_{i-1}$, we set (l_i, r_i) to $(i, i + L_i - 1)$, i.e., the new L-box starting at position i.
2. If $i \le r_{i-1}$, then $T[i]$ is contained in the L-box $(l_{i-1}..r_{i-1})$. Let $\beta = T[i..r_{i-1}]$ and $z = Z_P(i - l_{i-1} + 1)$. Then, $\beta[1..\ell] \approx P[1..\ell]$ where $\ell = \min(|\beta|, z)$, i.e., it is guaranteed that L_i is at least $\min(|\beta|, z)$, which is proved in Lemma 3.
   (a) If $z = |\beta|$, then L_i can be larger than $z = r_{i-1} - i + 1$. Thus, we compute L_i by explicitly checking the inequality (1) for T_i and P character by character, starting from the position $z + 1$. Furthermore, we set (l_i, r_i) as $(i, i + L_i - 1)$, i.e., the new L-box starting at position i.
   (b) If $z < |\beta|$, then $L_i = z$ and $(l_i, r_i) = (l_{i-1}, r_{i-1})$.
   (c) If $z > |\beta|$, then $L_i = |\beta|$ and $(l_i, r_i) = (l_{i-1}, r_{i-1})$.

We illustrate how our algorithm computes L_i's for $T = (13, 92, 34, 88, 77, 63, 37, 40, 70, 54, 35, 24, 50)$ and $P = (54, 12, 38, 69, 45, 22)$. We first preprocess P (see Table 1 for $LMax_P$, $LMin_P$, and Z_P). Initially, $(l_0, r_0) = 0$.

- Iteration $i = 1$ (Case 1). We obtain $L_1 = 1$ by explicitly checking the inequality (1) for T_1 and P. Note that the inequality is false at position 2 since $LMin_P[2] = 1$ and $T[2](= 92) \not< T[1](= 13)$, i.e., $T[1..2] \not\approx P[1..2]$. Thus, the L-box starting at position 1 is $(1, 1)$, and it is set as (l_1, r_1) since $r_0 < 1$.
- Iteration $i = 2$ (Case 1). We compute $L_2 = 3$ explicitly and set $(l_2, r_2) = (2, 4)$. Note that $T_2[1..3] = (92, 34, 88) \approx P[1..3] = (54, 12, 38)$ but $T_2[1..4] = (92, 34, 88, 77) \not\approx P[1..4] = (54, 12, 38, 69)$.

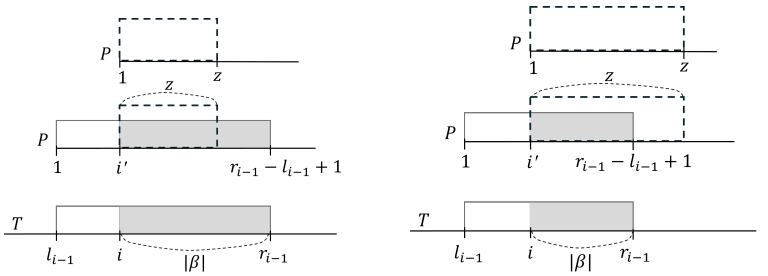
- Iteration $i = 3$ (Case 2(b)). We have $\beta = T[3..4]$, $z = Z_P(3-2+1) = Z_P(2) = 1$, and $\ell = 1$. Then, $T[3..3] = T_3[1..1] \approx P[1..1]$, but $T_3[1..2] = (34, 88) \not\approx P[1..2] = (54, 12)$, i.e., $L_3 = 1$, which we can know without checking the inequality (1) as proven in Lemma 3. The new L-box starting at position 3 is $(3, 3)$ and it is included in $(l_2, r_2) = (2, 4)$. Hence $(l_3, r_3) = (2, 4)$.
- Iteration $i = 4$ (Case 2(a)). We have $\beta = T[4..4]$ and $z = Z_P(4-2+1) = Z_P(3) = 1$. Since $|\beta| = z$, we obtain $L_4 = 2$ by explicitly checking the inequality (1) starting from position 2. Note that $T_4[1..2] = (88, 77) \approx P[1..2] = (54, 12)$, but $T_4[1..3] = (88, 77, 63) \not\approx P[1..3] = (54, 12, 38)$. Moreover, we set $(l_4, r_4) = (4, 5)$.
- Iteration $i = 5$ (Case 2(a)). We have $\beta = T[5..5]$ and $z = Z_P(5-4+1) = Z_P(2) = 1$. Similarly, we obtain $L_5 = 2$ and $(l_5, r_5) = (5, 6)$.
- Iteration $i = 6$ (Case 2(a)). We have $\beta = T[6..6]$ and $z = Z_P(6-5+1) = Z_P(2) = 1$. Similarly, we obtain $L_6 = 5$ and $(l_6, r_6) = (6, 10)$.
- Iteration $i = 7$ (Case 2(b)). We have $\beta = T[7..10]$ and $z = Z_P(7-6+1) = Z_P(2) = 1$. Similarly to iteration 3, we obtain $L_7 = 1$ and $(l_7, r_7) = (6, 10)$.
- Iteration $i = 8$ (Case 2(b)). We have $\beta = T[8..10]$ and $z = Z_P(8-6+1) = Z_P(3) = 1$. Similarly to iteration 7, we obtain $L_8 = 1$ and $(l_8, r_8) = (6, 10)$.

Lemma 3. *Algorithm 3 correctly computes L_i and (l_i, r_i) for all $1 \le i \le n - m + 1$.*

Proof. It can be proved by induction. Initially (when $i = 1$), since $i > r_0 = 0$ (Case 1), we compute L_i explicitly and set (l_1, r_1) as $(1, L_1)$, which is the only L-box whose start position is less than or equal to 1.

Now we consider an iteration where $i > 1$. In Case 1, we obtain correct L_i since we compute it by explicitly checking inequality (1). Since $L_i > 0$ (note that the first characters of two strings are trivially order-isomorphic), the end position $i + L_i - 1$ of the new L-box starting at i is greater than or equal to i. Additionally, since $i > r_{i-1}$ (by the condition of Case 1), $i + L_i - 1 > r_{i-1}$. Hence, $(i, i + L_i - 1)$ is the L-box whose end position is the rightmost among the L-boxes whose start position is less than or equal to i. Therefore, we compute (l_i, r_i) correctly.

In Case 2, $T[i]$ is contained in substring $T[l_{i-1}..r_{i-1}]$ such that $T[l_{i-1}..r_{i-1}] \approx P[1..r_{i-1} - l_{i-1} + 1]$, which means $T[i..r_{i-1}] \approx P[i - l_{i-1} + 1..r_{i-1} - l_{i-1} + 1]$ by Lemma 1. Let $\beta = T[i..r_{i-1}]$, $i' = i - l_{i-1} + 1$, and $z = Z_P(i')$ (see Figure 3). Then, $\beta \approx P[i'..i' + |\beta| - 1]$ and $P[i'..i' + z - 1] \approx P[1..z]$ (by the definition of z). Thus, $\beta[1..\ell] \approx P[1..\ell]$ where $\ell = \min(|\beta|, z)$, i.e., $L_i \ge \min(|\beta|, z)$.



(i) When $z < |T[i..r_{i-1}]|$ (ii) When $z > |T[i..r_{i-1}]|$

Figure 3. Illustration of several cases under the condition $i \le r_{i-1}$.

In Case 2(a), we obtain correct L_i since we compute it by explicitly checking inequality (1) starting from position $z + 1$. Note that we do not need to check the inequality for the position k $(1 \le k \le z)$ since $T_i[1..z] \approx P[1..z]$. Additionally, since $L_i \ge z = |\beta|$ $(= r_{i-1} - i + 1)$, it follows that $i + L_i - 1 \ge r_{i-1}$. Therefore, (l_i, r_i) should be $(i, i + L_i - 1)$.

In Case 2(b), $T_i[z + 1]$ is contained in β and $T_i[1..z + 1] \approx P[i'..i' + z]$ (see Figure 3(i)). Since $P[i'..i' + z] \not\approx P[1..z + 1]$ by the definition of $z = Z_P(i')$, it follows that $T_i[1..z + 1] \not\approx$

$P[1..z+1]$ and thus $L_i = z$. Moreover, since the end position $i+z-1$ of the new L-box starting at position i is to the left of r_{i-1} $(= i + |\beta| - 1)$, (l_i, r_i) is the same as (l_{i-1}, r_{i-1}).

In Case 2(c), $P[|\beta|+1]$ is contained in $P[1..z]$ and $P[1..|\beta|+1] \approx P[i'..i'+|\beta|]$ (see Figure 3(ii)). Since $P[i'..i'+|\beta|] \not\approx T_i[1..|\beta|+1]$ by the definition of β, it follows that $T_i[1..|\beta|+1] \not\approx P[1..|\beta|+1]$ and thus $L_i = |\beta|$. Additionally, the end position $i+|\beta|-1$ of the new L-box starting at position i is equal to r_{i-1}. Hence, (l_i, r_i) can be either (l_{i-1}, r_{i-1}) or $(i, i+L_i-1)$ (where (l_{i-1}, r_{i-1}) is chosen in our algorithm).

Therefore, Algorithm 3 computes L_i and (l_i, r_i) correctly in all cases. □

Lemma 4. *Given $LMax_P$, $LMin_P$, and Z_P, we can compute all L_i's $(1 \le i \le n-m+1)$ in $O(n)$ time.*

Proof. Each iteration takes constant time except for checking the inequality (1) in procedure CompLoip. The check ends when the inequality becomes false or the check reaches the last position. Let x_i be the number of times when the inequality is true during iteration i. If $x_i > 0$, r_i increases by x_i (and never decreases) from r_{i-1} (Cases 1 and 2(a)). Since $r_i \le n$, the total sum of x_i's for all iterations is at most n. Therefore, given $LMax_P$, $LMin_P$, and Z_P, all L_i's can be computed in $O(n)$ time. □

Theorem 1. *Given a text T of length n and a pattern P of length m, the OPPM problem with partition can be solved in $O(n + m\log m)$ time. If the characters of P can be sorted in linear time, the problem can be solved in $O(n+m)$ time.*

Proof. We can compute $LMax_P$ and $LMin_P$ in $O(m\log m)$ time. (If we can sort the characters of P in linear time, for example, such as when the characters are drawn from an integer alphabet, we can compute $LMax_P$ and $LMin_P$ in $O(m)$ time [2]). Then, Z_P and L_i's can be computed in $O(m)$ time and $O(n)$ time, respectively. Moreover, R_i's can be computed similarly using the reversed strings of P and T. For each position i $(1 \le i \le n-m+1)$ in Algorithm 2, given L_i and R_i, we can determine whether T_i is partitioned isomorphic to P and compute the range of partition positions in constant time. Therefore, we obtain the theorem. □

4. Experimental Results

The experimental environment was as follows: the operating system was Microsoft Windows 10 Education (64-bit), the CPU was an AMD Ryzen 9 5950X 16-Core, with 64 GB of RAM. The development tool was gcc 13.1.0, and the development language was C++. Three types of time-series data were used in the experiment: a power consumption index, particulate matter (PM2.5) levels, and the Dow Jones Index. The power consumption index consisted of data on the average household voltage per minute in Sceaux, France, from 00:00 on 16 December 2006, to 22:00 on 2 December 2008 [21]. The PM2.5 levels were recorded in Beijing at one-hour intervals from 00:00 on 2 January 2010, to 22:00 on 9 October 2014 [22]. The Dow Jones Index data included the daily closing prices of the Dow Jones Industrial Average from 2 May 1885, to 12 April 2019 [23]. For brevity, the power consumption index data are referred to as PCI, the particulate matter level data as PML, and the Dow Jones Index as DJI. We generated the experimental data as follows. First, the text was created by multiplying the real-valued time-series data by appropriate values to convert all values into integers. Since using a random string as the pattern could result in significantly fewer occurrences, a fixed-length substring starting from a random position in the converted text was designated as the pattern.

We compare the number of occurrences detected by our OPPM with partition algorithm (referred to as OPPM_PART) with those detected by the exact OPPM algorithm proposed by Kim et al. [1] (referred to as OPPM_EXACT). The lengths n of text T for PCI, PML, and DJI were 2,049,280, 41,757, and 38,104, respectively. Pattern P was generated by randomly extracting strings of lengths 6, 8, 10, 12, and 14 from T. The reason for selecting the pattern lengths in this way is that, due to the nature of OPPM, the longer the pattern

length, the fewer matching parts occur. In fact, when the pattern length exceeded 15, there were almost no occurrences.

Table 3 compares the number of occurrences for each algorithm, which are the averages obtained from executing the algorithm on 100 different patterns. As shown in the table, our algorithm, OPPM_PART, detects approximately 2.2 (for $m = 14$) to 51.0 (for $m = 8$) times more occurrences than OPPM_EXACT with PCI. With PML, OPPM_PART detects approximately 7.5 (for $m = 14$) to 35.3 (for $m = 10$) times more occurrences than OPPM_EXACT. With DJI, OPPM_PART detects approximately 1.9 (for $m = 14$) to 25.1 (for $m = 8$) times more occurrences than OPPM_EXACT. In all cases we experimented with, OPPM_PART detected significantly more meaningful parts than OPPM_EXACT.

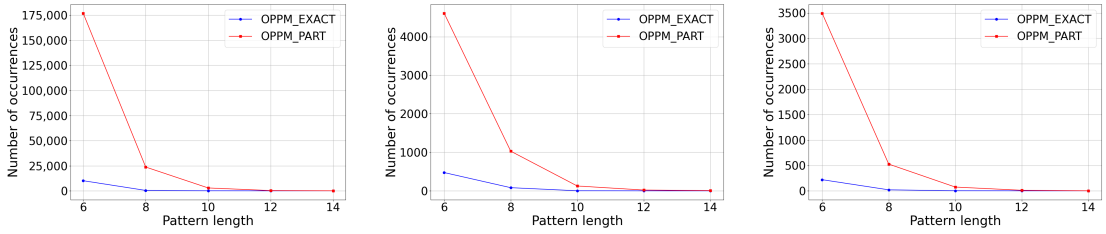
Table 4 compares the execution times for each algorithm, which are the averages obtained from running the algorithm on 100 different patterns. As shown in Table 4, OPPM_PART runs approximately 2.3 (for $m = 10$) to 2.5 (for $m = 6$) times slower than OPPM_EXACT with PCI. With PML, OPPM_PART runs approximately 2.3 (for $m = 14$) to 2.8 (for $m = 6$) times slower than OPPM_EXACT. With DJI, OPPM_PART runs approximately 2.2 (for $m = 14$) to 2.8 (for $m = 6$) times slower than OPPM_EXACT. In all the cases we experimented with, OPPM_PART was approximately twice as slow as OPPM_EXACT. Although both algorithms, OPPM_EXACT and OPPM_PART, have the same time complexity of $O(n + m \log m)$, OPPM_PART was slower due to the additional $O(n)$ time required for calculating L_i and R_i, as confirmed by additional experiments (not presented in a table). However, considering that the actual execution time is measured in microseconds, this increase in runtime is not significant (see Figure 4).

Table 3. Comparison of the number of occurrences for PCI, PML, and DJI data (averages of 100 times of executions).

Data	Algorithm	Pattern Length (m)				
		$m = 6$	$m = 8$	$m = 10$	$m = 12$	$m = 14$
PCI (n = 2,049,280)	OPPM_EXACT	10,138.7	465.5	80.9	40.7	34.2
	OPPM_PART	176,917.0	23,751.0	2927.8	315.9	76.8
PML (n = 41,757)	OPPM_EXACT	475.5	83.4	3.6	1.2	1.1
	OPPM_PART	4608.4	1030.1	127.2	22.1	8.3
DJI (n = 38,104)	OPPM_EXACT	220.7	21.0	3.1	1.3	1.0
	OPPM_PART	3495.5	527.5	75.7	12.1	1.9

Table 4. Comparison of execution times (microseconds) for PCI, PML, and DJI data (averages of 100 times of executions).

Data	Algorithm	Pattern Length (m)				
		$m = 6$	$m = 8$	$m = 10$	$m = 12$	$m = 14$
PCI (n = 2,049,280)	OPPM_EXACT	12,154.5	12,060.2	11,997.9	12,043.4	12,134.3
	OPPM_PART	29,979.6	28,583.7	28,102.4	28,472.5	28,552.8
PML (n = 41,757)	OPPM_EXACT	241.5	243.9	246.8	242.2	243.6
	OPPM_PART	672.4	629.5	620.7	563.9	549.3
DJI (n = 38,104)	OPPM_EXACT	226.9	231.7	232.9	231.5	235.2
	OPPM_PART	624.6	592.6	580.2	565.2	519.2



(**a**) PCI (**b**) PML (**c**) DJI

Figure 4. The number of occurrences according to pattern length.

5. Conclusions

We have introduced the concept of partitioned order-isomorphism as a novel criterion for approximate order-isomorphism. This approximation criterion is significant not only because it can be utilized in analyzing time-series data where errors may occur, but also because it provides a criterion for approximation that is challenging to address in classical pattern matching based solely on character value comparison. Using this criterion, we have defined the OPPM problem with partition and presented an efficient algorithm for solving it. One advantage of our criterion is that it allows for approximate matching without increasing the time complexity compared to the exact OPPM.

Our experimental results show that we can find more occurrences using our approximation criterion. As shown in the experimental results, our algorithm detected significantly more meaningful parts in all cases we tested. Therefore, we expect that our criterion will also enable more meaningful pattern search, prediction, and analysis in real-world applications. Moreover, as the pattern length increases, the difference of the number of occurrences between OPPM_EXACT and OPPM_PART decreases. This suggests that partitioning into more than two parts may be necessary for longer patterns.

Although our current focus is on partitioning a string into two parts, our definition can be extended to cases where the string is partitioned into three or more parts. However, developing an efficient algorithm for partitioning into even three parts is not straightforward. In the two-part case, we solved the problem efficiently by preprocessing T (i.e., computing L_i's and R_i's), since the start position of the left part and the end position of the right part are fixed. In contrast, when partitioning into three parts, neither the start nor the end position of the middle part is fixed, which can increase the complexity. We plan to explore this in future research.

Author Contributions: J.C.N. and Y.K. designed and analyzed the algorithm and wrote the draft of the paper. S.K. implemented and experimented with the algorithms. J.C.N. and J.S.S. reviewed and revised the paper. J.S.S. analyzed the algorithm, provided algorithmic support, and was the project manager. All authors read and approved the final manuscript.

Funding: This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean Government (MSIT) (No. RS-2022-00155915, Artificial Intelligence Convergence Innovation Human Resources Development (Inha University)) and by the INHA UNIVERSITY Research Grant.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare they have no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

OPPM	order-preserving pattern matching
KMP	Knuth–Morris–Pratt
loip	longest order-isomorphic prefix
lcp	longest common prefix
PCI	power consumption index data
PML	particulate matter level data
DJI	Dow Jones Index data
OPPM_EXACT	OPPM algorithm proposed in [1]
OPPM_PART	OPPM with partition algorithm proposed in this study

References

1. Kim, J.; Eades, P.; Fleischer, R.; Hong, S.; Iliopoulos, C.S.; Park, K.; Puglisi, S.J.; Tokuyama, T. Order-preserving matching. *Theor. Comput. Sci.* **2014**, *525*, 68–79. [CrossRef]
2. Kubica, M.; Kulczyński, T.; Radoszewski, J.; Rytter, W.; Waleń, T. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.* **2013**, *113*, 430–433. [CrossRef]

3. Kim, J.; Amir, A.; Na, J.C.; Park, K.; Sim, J.S. On Representations of Ternary Order Relations in Numeric Strings. *Math. Comput. Sci.* **2017**, *11*, 127–136. [CrossRef]
4. Knuth, D.E.; Morris, J.H.; Pratt, V.R. Fast Pattern Matching in Strings. *SIAM J. Comput.* **1977**, *6*, 323–350. 10.1137/0206024.
5. Horspool, R.N. Practical Fast Searching in Strings. *Softw. Pract. Exp.* **1980**, *10*, 501–506. [CrossRef]
6. Cho, S.; Na, J.C.; Park, K.; Sim, J.S. A fast algorithm for order-preserving pattern matching. *Inf. Process. Lett.* **2015**, *115*, 397–402. [CrossRef]
7. Chhabra, T.; Tarhio, J. A filtration method for order-preserving matching. *Inf. Process. Lett.* **2016**, *116*, 71–74. [CrossRef]
8. Na, J.C.; Lee, I. A simple heuristic for order-preserving matching. *IEICE Trans. Inf. Syst.* **2019**, *102*, 502–504.
9. Kim, Y.; Kim, Y.; Sim, J.S. An improved order-preserving pattern matching algorithm using fingerprints. *Mathematics* **2022**, *10*, 1954. [CrossRef]
10. Hasan, M.M.; Islam, A.S.M.S.; Rahman, M.S.; Rahman, M.S. Order preserving pattern matching revisited. *Pattern Recognit. Lett.* **2015**, *55*, 15–21. [CrossRef]
11. Gourdel, G.; Kociumaka, T.; Radoszewski, J.; Rytter, W.; Shur, A.M.; Walen, T. String periods in the order-preserving model. *Inf. Comput.* **2020**, *270*, 104463. [CrossRef]
12. Park, S.; Park, J.; Kim, Y.; Sim, J.S. Order-Preserving Multiple Pattern Matching in Parallel. *Appl. Sci.* **2023**, *13*, 5142. [CrossRef]
13. Chhabra, T.; Faro, S.; Külekci, M.O.; Tarhio, J. Engineering order-preserving pattern matching with SIMD parallelism. *Softw. Pract. Exp.* **2017**, *47*, 731–739. [CrossRef]
14. Crochemore, M.; Iliopoulos, C.S.; Kociumaka, T.; Kubica, M.; Langiu, A.; Pissis, S.P.; Radoszewski, J.; Rytter, W.; Waleń, T. Order-preserving indexing. *Theor. Comput. Sci.* **2016**, *638*, 122–135. [CrossRef]
15. Nakamura, T.; Inenaga, S.; Bannai, H.; Takeda, M. Order preserving pattern matching on trees and dags. In *International Symposium on String Processing and Information Retrieval*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 271–277.
16. Cantone, D.; Faro, S.; Külekci, M.O. The order-preserving pattern matching problem in practice. *Discret. Appl. Math.* **2020**, *274*, 11–25. [CrossRef]
17. Gawrychowski, P.; Uznański, P. Order-preserving pattern matching with k mismatches. *Theor. Comput. Sci.* **2016**, *638*, 136–144. [CrossRef]
18. Russo, L.M.S.; Costa, D.M.; Henriques, R.; Bannai, H.; Francisco, A.P. Order-preserving pattern matching indeterminate strings. *Inf. Comput.* **2022**, *289*, 104924. [CrossRef]
19. Kim, Y.; Kang, M.; Na, J.C.; Sim, J.S. Order-preserving pattern matching with scaling. *Inf. Process. Lett.* **2023**, *180*, 106333. [CrossRef]
20. Gusfield, D. *Algorithms on Strings, Trees, and Sequences—Computer Science and Computational Biology*; Cambridge University Press: Cambridge, UK, 1997. [CrossRef]
21. Hebrail, G.; Berard, A. Individual Household Electric Power Consumption Data Set. 2012. Available online: https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption (accessed on 26 September 2024).
22. Chen, S.X. Beijing PM2.5 Data Set. 2017. Available online: https://archive.ics.uci.edu/dataset/381/beijing+pm2+5+data (accessed on 26 September 2024).
23. Williamson, S. Daily Closing Values of the DJA in the United States, 1885 to Present, Measuring Worth. 2021. Available online: https://www.measuringworth.com/datasets/DJA/index.php (accessed on 26 September 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.