Segment and Fenwick Trees for Approximate Order Preserving Matching

Rafael Niquefa[1], Juan Mendivelso[2,3](✉), Germán Hernández[4], and Yoan Pinzón[5]

[1] Facultad de Ciencias e Ingeniería, Politecnico Grancolombiano Institucion Universitaria, Calle 57 # 3 - 00 Este, i Tower Bogotá, Bogotá, Colombia
rniquefa@poligran.edu.co

[2] Departamento de Matemáticas, Universidad Nacional de Colombia, Bogotá, Colombia
jcmendivelsom@unal.edu.co

[3] Facultad de Matemáticas e Ingenierías, Fundación Universitaria Konrad Lorenz, Bogotá, Colombia

[4] Departamento de Ingeniería de Sistemas e Industrial, Universidad Nacional de Colombia, Bogotá, Colombia

[5] Departamento de Electrónica y Ciencias de la Computación, Pontificia Universidad Javeriana, Cali, Colombia

Abstract. In this paper we combine two string searching related problems: the approximate string matching under parameters δ and γ, and the order preserving matching problem. Order-preserving matching regards the internal structure of the strings rather than their absolute values while matching under δ and γ distances permit a level of error. We formally define the $\delta\gamma$–order-preserving matching problem. We designed two algorithms for it based on the segment tree and the Fenwick tree, respectively. Also, we design and implement in C++ and an experimental setup to compare these algorithms with the naive solution and the *updateBA* algorithm introduced in [22]. The data structure based algorithms show better experimental performance due to their better lower bound of $\Omega(n \lg n)$ complexity.

Keywords: String searching · Strings similarity metric · Fenwick tree · Binary indexed tree · Segment tree

1 Introduction

Stringology is the branch of computer science that is dedicated to the study of problems in which sequences are involved. One of the main problems of interest in stringology is *string matching*, which consists of finding the occurrences of a pattern within a text. Formally, the input of a string matching algorithm is a text T, of length n, and a pattern P, of length m. Both the text and the pattern are formed by the concatenation of symbols of a given alphabet Σ. This alphabet for the vast majority of practical applications can be considered as an ordered set of different symbols. The output of a pattern matching algorithm is the list of positions in the text T where the pattern P is found.

The strings will be considered throughout the paper as indexed from 0. A notation generally used to represent substrings in a string, and which we will adopt in this paper, is the following: Let $T_{0\ldots n-1}$ represent a length-n string defined over Σ. The symbol at the position i of a string T is denoted as T_i. Also, $T_{i\ldots j}$ represents the substring of the text T from the position i to the position j, i.e. $T_{i\ldots j} = T_i T_{i+1} \cdots T_j$, where it is assumed that $0 \leq i \leq j < n$. In particular, we are interested in each length-m substring that starts at position i of the text, i.e. $T_{i\ldots i+m-1}$, $0 \leq i \leq n-m$, which we call *text window* and denote as T^i in the rest of the paper. Then, the output of the exact string matching problem should list all the positions i, $0 \leq i \leq n-m$, such that $P_j = T_{i+j}$ for all $0 \leq j \leq m-1$.

In this paper, two variants of the problem of exact search of patterns were combined: the $\delta\gamma$–matching problem and the order preserving matching problem. Both of them consider integer alphabets. The *$\delta\gamma$–matching* problem consists of finding all the text windows in T for which $\max_{0\leq j\leq m-1} |P_j - T_{i+j}| \leq \delta$ and $\sum_{j=0}^{m-1} |P_j - T_{i+j}| \leq \gamma$. We can see that δ limits the individual error of each position while γ limits the total error. Then, $\delta\gamma$–matching has applications in bioinformatics, computer vision and music information retrieval, to name some. Cambouropoulos et al. [3] was perhaps the first to mention this problem motivated by Crawford's work et al. [6]. Recently, it has been used to make more flexible other string matching paradigms such as parameterized matching [17,18], function matching [19] and jumbled matching [20,21].

On the other hand, *order-preserving matching* considers the order relations within the numeric strings rather than the approximation of their values. Specifically, the output of this problem is the set of text windows whose natural representation match the natural representation of the pattern. The natural representation of a string is a string composed by the rankings of each symbol in such string. In particular, the ranking of symbol T_i of string $T_{0\ldots n-1}$ is:

$$rank_T(i) = 1 + |\{T_j < T_i : 0 \leq j, i < n \wedge i \neq j\}| + |\{T_j = T_i : j < i\}|.$$

Then, the natural representation of T is $nr(T) = rank_T(0) rank_T(1) \cdots rank_T(n-1)$. Therefore, order preserving matching consists of finding all the text windows T^i such that $nr(P) = nr(T^i)$. Note that this problem is interested in matching the internal structure of the strings rather than their values. Then, it has important applications in music information retrieval and stock market analysis. Specifically, in music information retrieval, one may be interested in finding matches between relative pitches; similarly, in stock market analysis the variation pattern of the share prices may be more interesting than the actual values of the prices [15]. Since Kim et al. [15] and Kubica et al. [16] defined the problem, it has gained great attention from several other researchers [4,5,7–10,14].

Despite the extensive work on order-preserving matching, the only approximate variant in previous literature, to the best of our knowledge, was recently proposed by Uznański and Gawrychowski [12]. In particular, they allow k mismatches between the pattern and each text window. Then, they regard the number of mismatches but not their magnitude. In this paper, we propose a different approach to approximate order-preserving matching that bounds the magnitude of the mismatches through the $\delta\gamma$- distance. Specifically, δ is a bound between the ranking of each character in the pattern and its corresponding character in the text window; likewise, γ is a bound on the sum of all such differences in ranking. Thus, δ and γ respectively restrict the magnitude of the error individually and globally across the strings. We define $\delta\gamma$*–order-preserving matching* as the problem of finding all the text windows in T that match the pattern P under this new paradigm.

We first defined the notion of $\delta\gamma$–order preserving matching in [22]. Now, in this paper we provide a more formal definition in Sect. 2. In Sect. 3, we present two new algorithms for this problem: one based on segment trees and the other based on Fenwick trees. In Sect. 4, we describe the experiment performed to compare the algorithms with a naive solution and the algorithm *updateBA* introduced in [22]. The data structures based algorithms experimentally outperformed the other two. Finally, the concluding remarks are presented in Sect. 5.

2 Definition of $\delta\gamma$–Order Preserving Matching Problem ($\delta\gamma$–OPMP)

The motivation to define $\delta\gamma$–order-preserving matching stems from the observation that the application areas of order-preserving matching, mainly stock market analysis and music information retrieval, require to search for occurrences of the pattern that may not be exact but rather have slight modifications in the magnitude of the rankings. For example, let us assume that the text T presented in Fig. 1 is a sequence of stock prices and that we want to determine whether it contains similar occurrences of the pattern P (also shown in this figure). Under the exact order-preserving matching paradigm, there are no matches, but there are similar occurrences at positions and 1 and 11. In particular, $T_{1...8}$ and $T_{11...18}$ are similar, regarding order structure, to the pattern. This similarity can be seen even more clearly if we consider natural representations of these strings (also shown in Fig. 1).

Next we will formally define the $\delta\gamma$*–order-preserving match*, and with that definition we will define the $\delta\gamma$*–order-preserving matching ($\delta\gamma$–OPMP).*

Definition 1 ($\delta\gamma$–order-preserving match). *Let $X = X_{0...m-1}$ and $Y = Y_{0...m-1}$ be two equal-length strings defined over Σ_σ. Also, let δ, γ be two given numbers ($\delta, \gamma \in \mathbb{N}$). Strings X and Y are said to $\delta\gamma$–order-preserving match, denoted as $X \overset{\delta\gamma}{\leftrightsquigarrow} Y$,* ***iff*** *$nr(X) \overset{\delta\gamma}{\cong} nr(Y)$.*

Example 1. Given $\delta = 2$, $\gamma = 6$, $X = \langle 10, 15,\ 19, 12,\ 11, 18,\ 23, 22\rangle$ and $Y = \langle 14, 17,\ 20, 18,\ 12, 15,\ 23, 22\rangle$, $X \overset{\delta\gamma}{\leftrightsquigarrow} Y$ as $nr(X) = \langle 1, 4,\ 6, 3,\ 2, 5,\ 8, 7\rangle$, $nr(Y) = \langle 2, 4,\ 6, 5,\ 1, 3,\ 8, 7\rangle$ and $nr(X) \overset{\delta\gamma}{\cong} nr(Y)$.

Problem 1 ($\delta\gamma$–order-preserving matching ($\delta\gamma$–OPMP)). Let $P = P_{0...m-1}$ be a pattern string and $T = T_{0...n-1}$ be a text string, both defined over Σ_σ. Also, let

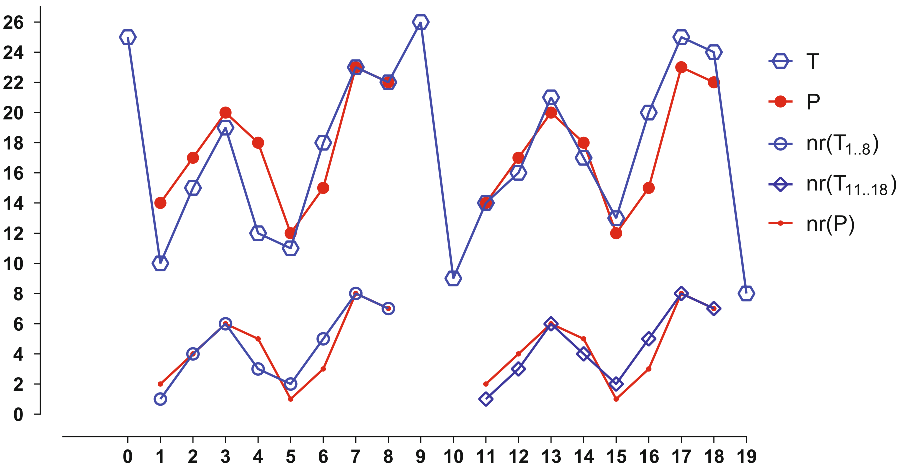


Fig. 1. Order preserving matching under $\delta\gamma$ approximation example.

δ, γ be two given numbers ($\delta, \gamma \in \mathbb{N}$). The $\delta\gamma$–*order-preserving matching problem* is to calculate the set of all indices i, $0 \leq i \leq n - m$, satisfying the condition $P \overset{\delta\gamma}{\leadsto} T^i$. From now on $\delta\gamma$–**OPMP**.

3 Algorithms for the $\delta\gamma$–OPMP

In this section, we present two algorithms that solve the $\delta\gamma$–Order preserving matching problem ($\delta\gamma$–OPMP): one that makes use of segment trees (Sect. 3.1) and the other utilizes Fenwick trees (Sect. 3.2).

3.1 Segment Tree Based Algorithm (*segtreeBA*)

The segment tree data structure is a powerful data structure with applications in many areas like in computational geometry [1,2] and graph theory. The segment tree data structure uses the divide and conquer approach to answer queries in ranges of an underlying array A. Every node in a Segment Tree is assigned a range and will contain the answer to the query for that specific range. We will use the segment tree data structure to solve the range minimum query (RMQ) problem, which consists in finding the index of the minimum value of the array in a given range, and we will be able to change elements of the array. Building a segment tree to solve the RMQ problem for an array A of length $|A|$ takes $O(|A|)$ space and time. The update and query operations both take $O(\lg |A|)$.

Based on this data structure, we propose the algorithm called *settreeBA* (see Fig. 2). It first calculates the natural representation of the pattern P (line 1 in Fig. 2). Then, it iterates over all possible position and tries to find $\delta\gamma$-order preserving matches in every one of them. The process of finding a match at position i in T is as follows: First the algorithm finds the smallest number

Algorithm 1: $\delta\gamma$–OPMP **segtreeBA**

```
Input: $P = P_{0...m-1}, T = T_{0...n-1}, \delta, \gamma$
Output: $\{i \in \{0, \dots, n-m\} : T^i \overset{\delta\gamma}{\leftrightsquigarrow} P\}$
1. Create as Array: $P^{nr} \leftarrow nr(P)$
2. Create as Array of size $m$: $oldValue$,$changedIndex$
3. Create as Segment Tree: $minIndex \leftarrow buildSegTree(T, 0, n-1)$
4. Define: $curDelta$,$curGamma$,$rank$,$idxT$,$idxP$,$nChanges$ as integers
5. $nChanges \leftarrow 0$
6. for $i = 0 \rightarrow n - m$ do
7.    for $rank = 1 \rightarrow m$ do
8.       $idxT \leftarrow querySegTree(minIndex, i, i + m - 1)$
9.       $idxP \leftarrow idxT - i$
10.      $curDelta \leftarrow |rank - P^{nr}_{idxP}|$
11.      $curGamma \leftarrow curGamma + curDelta$
12.      if $curDelta > delta \vee curGamma > gamma$ then break loop
13.      $changedIndex_nChanges \leftarrow idxT$
14.      $oldValue_nChanges \leftarrow T_{idxT}$
15.      $nChanges \leftarrow nChanges + 1$
16.      $updateSegTree(minIndex, idxT, \infty)$
17.   for $c = 0 \rightarrow nChanges - 1$ do
18.      $updateSegTree(minIndex, changedIndex_c, oldValue_c)$
19.   if $rank > m$ then report $i$
20.   $nChanges \leftarrow 0$
```

Fig. 2. Segment tree based algorithm: *segtreeBA*.

in the interval $\left[i, i+m-1\right]$ (line 8); this value has the rank 1 in the sliding window T^i. It then uses the natural representation of P to check the δ and γ restrictions for the rank 1 in the window T^i. Then it prepares the segment tree for the next iteration; this is done by changing the smallest value in the interval $\left[i, i+m-1\right]$ to infinity, so in the next iteration of the first inner loop the operation $querySegTree(minIndex, i, i+m-1)$ finds the second smallest value in the same interval. This process is done for all the rankings from 1 to m.

In the second inner loop (lines 17 and 18 in Fig. 2), the values of T in the interval $\left[i, i+m-1\right]$ must be changed so that, in the next window, those contain the original values of T and no infinity. The arrays $oldValue$ and $changedIndex$ help in the process of restoring the segment tree. We are going to adapt the standard operations of the segment tree to this solution as follows:

- $buildSegTree(T, 0, n-1)$: Builds a segment tree with T_0, T_1 , ... , T_{n-1} and returns the root node. The complexity is $O(n)$.
- $updateSegTree(minIndex, i, x)$: Sets T_i to x. The complexity is $O(\lg n)$.
- $querySegTree(minIndex, i, j)$: Returns the index of the minimum value among T_i, T_{i+1} , ... , T_j. If there are several minimum values, the leftmost (smallest index) is chosen. The complexity is $O(\lg n)$.

The complexity of *segtreeBA* can be computed as follows: In line 1 in Fig. 2, the algorithm creates the natural representation of the pattern with cost

$\Theta(m \lg m)$. In line 2 it creates two arrays of size m in $\Theta(m)$. In line 3 a segment tree is created in $\Theta(n)$. Then in the main loop it iterates over all $n-m+1$ candidates. For each candidate it finds the elements with ranks from 1 to m using the segment tree. Finding the position of each rank in the window costs $O(\lg n)$. After each rank position is found, the algorithm checks if the $\delta\gamma$ restrictions holds for the current window (lines 10 to 12). If so, it continues with the next rank; if not, the algorithm breaks the inner loop and continues with the next search window (line 12).

Due to the fact that the segment tree is used to find the smallest element in an interval, the algorithm must *mark* as ∞ the position of each rank. Then, in the next iteration, the next smallest element that is found, is the next rank. These changes are done in $O(\lg n)$ time (lines 13 to 16). Reversing those changes costs $O(m \lg n)$ (lines 17 to 18). In fact, the inner loops (lines 7 to 20) have a combined complexity of $O(m \lg n)$, but also have a lower bound of $\Omega(\lg n)$. The lower bound of this algorithm is then $\Omega(n \lg n)$, because in many cases it does not perform the m comparisons cost $O(\lg n)$. The total complexity of the algorithm is then $O(n + n \lg n + m \lg m + (n-m+1)(m \lg n)) = O(nm \lg n)$, but with a lower bound of $\Omega(n \lg n)$.

3.2 Fenwick Tree Based Algorithm (*bitBA*)

The Binary Indexed Tree (*BIT*) or Fenwick tree, proposed by Peter M. Fenwick in 1994 [11], is a data structure that can be used to maintain and query cumulative frequencies. In particular, it is mainly used to efficiently calculate prefix sums in an array of numbers. Based on this data structure, we propose the algorithm called *bitBA* (see Fig. 3). The BIT data structure could be considered then as an abstraction of an integer array of size n indexed from 1, i.e., a bit encapsulate $A = A_1 A_2 \cdots A_n$. The version we are going to use has two operations:

- $sumUpTo(tree, i)$: Returns $A_1 + A_2 + \ldots + A_i$. The complexity is $O(\lg n)$.
- $addAt(tree, i, x)$: Sums x to A_i. The complexity is $O(\lg n)$.

The algorithm has a preprocessing phase in which the data structures needed to solve the $\delta\gamma$–OPMP are created. This is done with a complexity of $\Theta(n + n \lg n + m \lg m)$. The term n is due to the creation of the BIT. The term $n \lg n$ is due to the creation of T^{nr} and the term $m \lg m$ is due to the creation of P^{nr}. In the searching phase, it iterates over all possible positions in the text T to find the existing matches. For each position i to be considered, the algorithm uses the BIT to get the rank of every symbol in the searching window $T_{i \ldots i+m-1}$, and then each rank in the window is compared with each rank in P^{nr} to check if T^i is a $\delta\gamma$–order preserving match. This operation is evaluated using the function $isAMatch(P, T^i, \delta, \gamma)$; in particular, this function returns true ***iff*** $P \overset{\delta\gamma}{\leftrightsquigarrow} T^i$ and this takes $O(m \lg m + m)$.

Each rank calculation using the BIT costs $O(\lg n)$. Then the total complexity of the algorithm is $O(n\lg n+m\lg m+(n-m+1)(m\lg n)) = O(nm\lg n)$. Similar to *segtreeBA*, *bitBA* has a lower bound of $\Omega(n\lg n)$ because, in many cases, *bitBA* does not perform the m comparisons of cost $O(\lg n)$. The total complexity of *bitBA* is then $O(n+n\lg n+m\lg m+(n-m+1)(m\lg n)) = O(nm\lg n)$, but with a lower bound of $\Omega(n\lg n)$.

In the preprocessing phase, the algorithm first creates the natural representations of the pattern P and the text T (P^{nr} and T^{nr}, respectively). Then, it creates a BIT which is an encapsulation of an array with n positions numbered from 1 to n. Then assigns 1 the positions T_0^{nr}, T_1^{nr}, $\ldots T_{m-2}^{nr}$ (Lines 1 to 5 in Fig. 3). In the searching phase, for each candidate position i, the algorithm computes the rank of each symbol $T_{i+j}, 0 \leq j \leq m-1$ using $sumUpTo(i+j)$. After checking if there is a match at position i, the BIT must be updated in each iteration to consider symbol T_{i+m} (line 7 in Fig. 3). And the BIT must be updated so it does not consider the position i in the next search window (line 9 in Fig. 3).

```
Algorithm 2: $\delta\gamma$–OPMP bitBA
Input: $P = P_{0\ldots m-1}, T = T_{0\ldots n-1}, \delta, \gamma, \Sigma_\sigma$
Output: $\{i \in \{0,\ldots,n-m\} : T^i \overset{\delta\gamma}{\leadsto} P\}$
1. Create as Array: $T^{nr} \leftarrow nr(T)$
2. Create as Array: $P^{nr} \leftarrow nr(P)$
3. Create as Array of size n: $bit$
4. for $i = 0 \to m-2$ do
5.     $addAt(bit, T_i^{nr}, 1)$
6. for $i = 0 \to n-m$ do
7.     $addAt(bit, T_{i+m-1}^{nr}, 1)$
8.     $isAMatch(i, bit, T^{nr}, P^{nr}, \delta, \gamma)$ then report $i$
9.     $addAt(bit, T_i^{nr}, -1)$
```

Fig. 3. BIT based algorithm: *bitBA*.

4 Experiments

In this section, we describe the experimental setup we designed to evaluate the performance of the proposed algorithms. We compare our algorithms with two baseline algorithms: The naive algorithm, which we call *naiveA*, and *updateBA*, presented in [22]. The former, whose time complexity is $\Theta(nm\ \lg m)$, considers all possible positions in the text and, for each one of them, verifies if there is a match in $\Theta(m\lg m+m)$ time. The latter algorithm, whose time complexity is $\Theta(nm)$, is based on linear update and verification.

We present the experimental framework (Sect. 4.1) and describe the data generation (Sect. 4.2). Then, we discuss the results obtained (Sect. 4.3). Finally we show the results of the experiments directed to detect how the algorithms *segtreeBA* and *bitBA* behave when in all the experiment instances the worst case came up (Sect. 4.4).

4.1 Experimental Setup

Here we describe the hardware and software used for the experiments. Then, we show how we vary the input parameters.

Hardware and Software. All the algorithms were implemented using C++. The computer used for the experiments was a Lenovo ThinkPad with a processor Intel(R) Core(TM) i7 4600u CPU @ 2.10 GHz 2.69 GHz and installed RAM memory of 8 GB. The computer was running 64-bit Linux Ubuntu 14.04.5 LTS. The C++ compiler version was g++ (Ubuntu 4.8.4-2ubuntu1 14.04.3) 4.8.4.

Parameters. It is clear that the defined problem has several parameters. They may change depending on the area of study in which the problem and string searching algorithms are applied. To show how our solution behaves with different configuration of the given parameters, we perform five types of experiments. In each experiment, we vary one of the given parameters n, m, δ, γ and σ, and let the other four parameters fixed at a given value. We chose the fixed values after several attempts via try and error to find values that produced results varying from no matches to matches near the value of n. For each experiment type, we performed five different experiments and took the median as the value to plot, making the median of five experiments the representative value for a experiment configuration of values n m, δ, γ and σ. The variation of the parameter values for each experiment type is presented in Table 1.

Table 1. Experimental values of n m, δ, γ and σ.

	Varying n	Varying m	Varying δ	Varying γ	Varying σ
n	$[3000, 60000] \Delta n = 3000$	10000	10000	10000	10000
m	40	$[30, 600]$ $\Delta m = 30$	40	40	40
δ	10	10	$[0, 228]$ $\Delta\delta = 12$	10	10
γ	60	60	60	$[0, 570]$ $\Delta\gamma = 30$	60
σ	100	100	100	100	$[12, 240] \Delta\sigma = 12$

4.2 Random Data Generation

An experiment consists of two stages. The first stage is the pseudo-random generation of a text T of length n and the pattern P of length m. The second stage is the execution of the algorithms on the generated strings P and T. The random generation of each character of both the pattern P and the text T is done by calling a function that pseudo-randomly and selects a number between 1 and σ with the same probability for each number to be selected, i.e., all symbols have the same probability to appear in a position and for that reason, the count of each symbol on a generated string will be similar to the quantities of the others symbols in the alphabet.

4.3 Experimental Results and Analysis

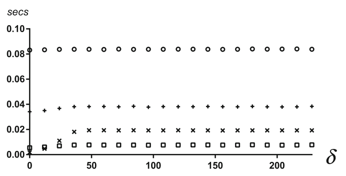
The first result to highlight is the fact that, in every experiment, the naive algorithm always has the worst performance, as expected. The results shown in Figs. 4a, b and c show that the size of the alphabet and the parameters δ and γ have practically no impact on the execution time of any of the algorithms, they all show nearly constant time behavior. Figures 4d and f verify the theoretical complexity analysis that states that n and m are the parameters that really determine the growth in the execution time of all the algorithms. In Fig. 4d, m is a constant and n is a variable while in Fig. 4f, n is a constant and m is a variable. Also, for a clearer illustration, in Figs. 4e and g we only show the best two algorithms of Figs. 4d and f, respectively, on the same data. It is important to notice that, under these conditions, the graphs are expected to be linear and the experiments verify that.

In the figures where we show the result of varying the parameter n and the parameter m, (Figs. 4d–g), we can see that the best two algorithms are the based on data structures (*segtreBA* and *bitBA*). This despite the fact that these two algorithms have a higher upper bound on their complexities in relation with the first two algorithms (*naiveA* and *updateBA*). This result can be explained by the fact that the lower bound on the data structure based algorithms is considerably lower in comparison with the other two. The lower bound of the data structures based algorithms is $\Omega(n \lg n)$ and the lower bound of the *naiveA* and *updateBA* is the same as their upper bound which is $\Theta(nm \lg m)$ and $\Theta(nm)$ respectively. This can be understood by taking into account that the first two algorithms check for a match after a natural representation of every window is completely obtained; on the contrary, data structure based algorithms break the calculation of a given natural representation of a window if at some point the δ or γ restriction do not hold.

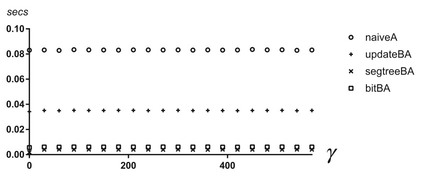
Given the result of the experiments, it is safe to say that the algorithms based on data structures are faster in most cases, especially if they are going to be used in applications where very few matches are expected to appear. This is due to their lower bound of complexity. We test two different implementations of the segment tree data structure: one based on classes and pointers, and the other based on an array. Finally we chose the array based as representative for the segment tree based solution and the experiments plots show their results. The array–based segment tree is almost twice time faster than the classes–based implementation.

4.4 Worst Case Experiments on *seegtreBA* and *bitBA*

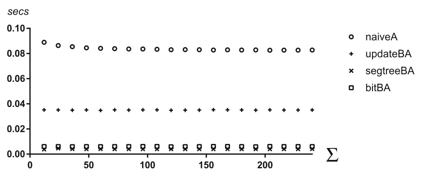
Taking into account that the first two algorithms, *naiveA* and *updateBA* both have complexities in θ–notation, i.e. their worst case is the same as their best case, the experiments described so far are enough for their experimental analysis. For the data structures based algorithms a more particular kind of experiment is needed, i.e. the worst case experimental analysis. For this algorithms the worst



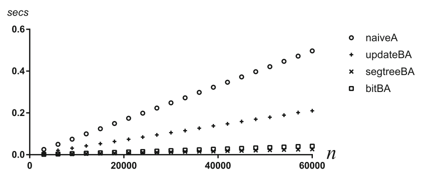
(a) Varying the parameter δ.



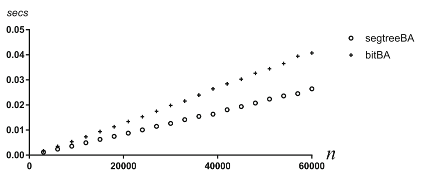
(b) Varying the parameter γ.



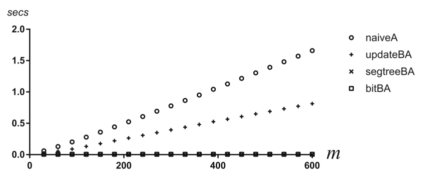
(c) Varying alphabet size $|\Sigma|$.



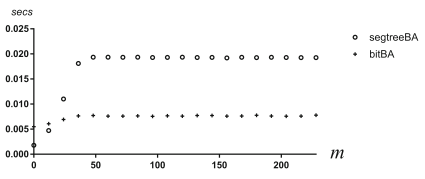
(d) Varying the text size n.



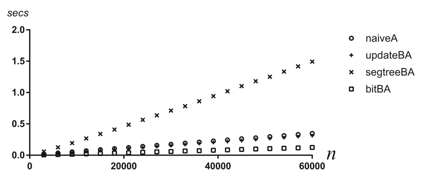
(e) Varying n: $bitBA$ vs $segtreeBA$.



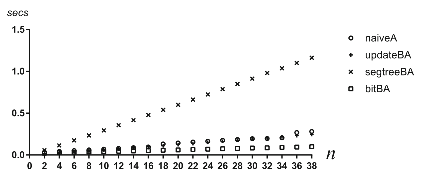
(f) Varying the pattern size m.



(g) Varying m: $bitBA$ vs $segtreeBA$.



(h) Varying n in the worst case.



(i) Varying m in the worst case.

Fig. 4. Experimental results of comparing the four algorithm by varying different parameters.

case is when there is a match in every candidate position. An easy way to generate data for the worst case is when all the symbols in both the pattern P and the text T are the same. Other way to generate worst cases scenarios for this two algorithms is when either both P and T are strictly increasing or both are strictly decreasing. Results from this experiments show a fast degradation in experimental performance of the *segtreeBA* algorithm, but a very slow degradation of the *bitBA* algorithm. Results of this last experiments are shown in Figs. 4h and i.

5 Conclusions and Future Work

We define a new variant of the string matching problem, the $\delta\gamma$–order preserving matching problem ($\delta\gamma$–OPMP). This new variant provides the possibility of searching a pattern according to the relative order of the symbols as the order preserving matching problem. But we also gives more flexibility to the search allowing error in the individual ranking comparisons due to the parameter δ. And also the proposed problem gives a bound for the global error in the comparison of a pattern against a text window by γ. This new variant has at least the same applications as the order preserving matching problem.

We designed two algorithms: one based on the segment tree and another based on the Fenwick tree. They both have complexities of $O(nm \lg n)$ for their worst case and a $\Omega(n \lg n)$ lower bound. We implemented in C++ these algorithms and compared them with a naive solution and the *updateBA* algorithm introduced in [22]. Their theoretical time complexity is $\Theta(nm \lg m)$. Our experimental results on randomly generated data show that in many cases, given the uniformly data generation, the proposed algorithms work faster than the *naiveA* and the *updateBA*. One question that remains open is if an algorithm with better worst case time complexity than $O(nm)$ can be designed; other question that also remains open is that if an algorithm with better lower bound than $\Omega(n \lg n)$ can be obtained.

We show experimental results on the worst cases of the *bitBA* and *segtreeBA*. We conclude that the degradation in performance in the *segtreeBA* algorithm is much more notorious than the degradation of *bitBA*. A question remains, and is if we can device an experimental setup where the best worst–case algorithm, *updateBA* experimentally beats the other three algorithms. Given the theory behind the big O notation, we can say that this experimental setup exist.

Our future work on this problem will be focused on the applications of $\delta\gamma$–order preserving matching. Specifically, we are interested in experiments with real data that verify some applications of this problem in music and finance. In music we can use it to search for a portion of a melody inside another music score; in finance, we can look for similar change patterns in the price of stock actions. It is important to notice that for specific applications more efficient algorithms could be designed based on the particularities of the chosen field (alphabet, language, etc.). Considering such particularities will also be part of our future work.

References

1. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: More Geometric Data Structures, pp. 219–241. Springer, Heidelberg (2008)
2. Brass, P.: Advanced Data Structures. Cambridge University Press, Cambridge (2008). Cambridge books online
3. Cambouropoulos, E., Crochemore, M., Iliopoulos, C., Mouchard, L., Pinzon, Y.: Algorithms for computing approximate repetitions in musical sequences. Int. J. Comput. Math. **79**(11), 1135–1148 (2002)
4. Chhabra, T., Kulekci, M.O., Tarhio, J.: Alternative algorithms for order-preserving matching. In: Holub, J., Žďárek, J. (eds.) Proceedings of the Prague Stringology Conference 2015, pp. 36–46. Czech Technical University in Prague, Prague, Czech Republic (2015)
5. Chhabra, T., Tarhio, J.: Order-preserving matching with filtration. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 307–314. Springer, Cham (2014). doi:10.1007/978-3-319-07959-2_26
6. Crawford, T., Iliopoulos, C.S., Raman, R.: String-matching techniques for musical similarity and melodic recognition. Comput. Musicol. **11**, 71–100 (1998)
7. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kubica, M., Langiu, A., Pissis, S.P., Radoszewski, J., Rytter, W., Waleń, T.: Order-Preserving Incomplete Suffix Trees and Order-Preserving Indexes, pp. 84–95. Springer, Cham (2013)
8. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Kubica, M., Langiu, A., Pissis, S.P., Radoszewski, J., Rytter, W., Walen, T.: Order-preserving suffix trees and their algorithmic applications. CoRR abs/1303.6872 (2013)
9. 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. **638**(C), 122–135 (2016)
10. Faro, S., Külekci, M.O.: Efficient algorithms for the order preserving pattern matching problem. CoRR abs/1501.04001 (2015)
11. Fenwick, P.M.: A new data structure for cumulative frequency tables. Softw. Pract. Exp. **24**, 327–336 (1994)
12. Gawrychowski, P., Uznański, P.: Order-preserving pattern matching with k mismatches. Theor. Comput. Sci. **638**, 136–144 (2016)
13. Hasan, M.M., Islam, A., Rahman, M.S., Rahman, M.S.: Order Preserving Prefix Tables, pp. 111–116. Springer, Cham (2014)
14. Hasan, M.M., Islam, A., Rahman, M.S., Rahman, M.: Order preserving pattern matching revisited. Pattern Recogn. Lett. **55**(C), 15–21 (2015)
15. Kim, J., Eades, P., Fleischer, R., Hong, S.H., Iliopoulos, C.S., Park, K., Puglisi, S.J., Tokuyama, T.: Order-preserving matching. Theor. Comput. Sci. **525**, 68–79 (2014). Advances in Stringology
16. Kubica, M., Kulczyński, T., Radoszewski, J., Rytter, W.: WaleÅĎ, T.: A linear time algorithm for consecutive permutation pattern matching. Information Processing Letters **113**(12), 430–433 (2013)
17. Lee, I., Mendivelso, J., Pinzón, Y.J.: $\delta\gamma$-Parameterized Matching, pp. 236–248. Springer, Heidelberg (2009)
18. Mendivelso, J.: Definition and solution of a new string searching variant termed $\delta\gamma$-parameterized matching. Master's thesis, National University of Colombia, Bogota, Colombia (2010)
19. Mendivelso, J., Lee, I., Pinzón, Y.J.: Approximate Function Matching under δ- and γ- Distances, pp. 348–359. Springer, Heidelberg (2012)

20. Mendivelso, J., Pino, C., Niño, L.F., Pinzón, Y.: Approximate Abelian Periods to Find Motifs in Biological Sequences, pp. 121–130. Springer, Cham (2015)
21. Mendivelso, J., Pinzón, Y.: A novel approach to approximate parikh matching for comparing composition in biological sequences. In: Proceedings of the 6th International Conference on Bioinformatics and Computational Biology (BICoB 2014) (2014)
22. Niquefa, R., Mendivelso, J., Hernández, G., Pinzón, Y.: Order preserving matching under $\delta\gamma$-approximation. In: Congreso Internacional de Ciencias Básicas e Ingeniería (2017)