UNIVERSIDAD NACIONAL DE COLOMBIA

Definition and solution of a new approximate variant of the order preserving matching problem

Rafael Alberto Niquefa Velásquez

National University of Colombia
Engineering Faculty, System and Computing Engineering
Bogotá, Colombia
2017

Definition and solution of a new approximate variant of the order preserving matching problem

Rafael Alberto Niquefa Velásquez

Thesis work submitted as a partial requirement to opt for degree of:
Msc in System and Computing Engineering

Advisor:
Juan Mendivelso, Ph.D

Co-advisor:
Germán Hernández, Ph.D

Research Line:
Theoretical Computer Science
Research Group:
ALGOS: Algorithms and Combinatory Research Group

National University of Colombia
Engineering Faculty
Bogotá, Colombia
2017

Dedicated to:

Juan Mendivelso for his continuous guidance, support and patience. To Germán Hernandez and Yoán Pinzón, the two greatest people I have ever had as teachers, for their advices and humanity.

Acknowledgment To my advisor Juan Mendivelso for his continuous feedback, advices and teachings in the writing process of this thesis. To Diego Niquefa, my brother and undergrad student in System and Computer Engineering of the National University of Colombia - Bogotá, for his valuable ideas and advices related to the algorithms and experimental setup used. To Gabriela Rojas, graduated from the Conservatory of Music of the National University of Colombia, Bogotá, for her advices and inputs from a musical perspective in the possible applications and examples of the work in this thesis.

Abstract

In this thesis 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 and implement in C++ four algorithms that solve the proposed problem and an experimental setup to compare them. The first algorithm is the naive algorithm with complexity $\Theta(nm \lg m)$ time. The second has a complexity of $\Theta(nm)$ time. The third and four algorithms are based on the segment tree and Fenwick tree data structures, respectively, and both have $O(nm \log n)$ time complexities. The data structure based algorithms show better experimental performance due to their better lower bound of $\Omega(n \lg n)$ complexity. We show applications in music and finance.

Keywords: String searching, Experimental algorithm analysis, Strings similarity metric, String searching algorithms, Fenwick tree, Binary indexed tree, Segment tree

Resumen

En esta tesis se combinan dos problemas de búsqueda de cadenas: la búsqueda aproximada de cadenas bajo parámetros $\delta\gamma$, y el emparejamiento con preservación de orden. Uno permite un nivel de error en la búsqueda, mientras que el otro considera la estructura interna de las cadenas en lugar de sus valores absolutos. Se define formalmente el **Emparejamiento con preservación de orden bajo distancias** $\delta\gamma$. Se diseñaron e implementaron en C++ cuatro algoritmos que resuelven el problema, y una configuración experimental para compararlos. El algoritmo más simple, tiene complejidad $O(nm \lg m)$. El segundo tiene una complejidad de $O(nm)$. El tercero y el cuarto se basan en estructuras de datos: árbol de segmentos y árbol de fenwick respectivamente. Ambos tienen complejidad $O(nm \lg n)$. Los resultados experimentales muestran que los algoritmos basados en estructuras de datos tiene un mejor rendimiento en muchos casos. El de mejor rendimiento experimental es del basado en el árbol Fenwick, seguido por el basado en árboles de segmentos. Estos resultados se pueden explicar debido a su complejidad $\Omega(n \lg n)$. Se muestran aplicaciones en música y finanzas.

Palabras clave: Búsqueda de cadenas, Análisis experimental de algoritmos, Métrica de similitud de cadenas, Árbol de Fenwick, Árbol indexado binario, Árbol de segmentos.

Contents

List of Tables

List of Figures

1. Introduction

Stringology is the branch of computer science that is dedicated to the study of algorithms, data structures and techniques related to the definition and solution of problems in which sequences are involved. One of the main problems of interest in stringology is *string matching* (also called *string pattern matching* or simply *pattern matching*), which consists in finding the occurrences of a pattern within a text, possibly according to certain conditions or characteristics depending on the variation of the problem. 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. It may well be the binary alphabet $\{0, 1\}$, the alphabet in bioinformatics $\{A, G, T, C\}$, the alphabet given by the Spanish language or any other language. It can also be the ASCII or UNICODE character alphabet. The strings will be considered throughout the document as indexed from 0.

The output of a pattern matching algorithm can be: (i) a boolean that indicates whether or not the pattern appears in the text; (ii) a number that indicates the position of the first occurrence of the pattern in the text; or (iii) the list of positions in the text T where the pattern P is found. In this thesis we will consider the problem with output (iii). A notation generally used to represent substrings in a string, and which we will adopt in this document, is the following: Let $T_{0...n-1}$ represent a length-n string defined over Σ. The symbol at the position i of a string T is denoted as T_i. Also, $T_{i...j}$ represents the substring of the text T from the position i to the position j, i.e. $T_{i...j} = T_i T_{i+1} \cdots T_j$, 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. $T_{i...i+m-1}$, $0 \leq i \leq n - m$, which we call *text window* and denote as T^i in the rest of the document. 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 $0 \leq j \leq m - 1$. For example, for the text $T = GATTACATTACATTACA$ and the pattern $P = TTA$, the answer given by exact pattern matching algorithms would be $\{2, 7, 12\}$ since the pattern TTA is found at the positions 2, 7 and 12 of the text T. For information on string matching algorithms, see for example: [16, 5].

In this thesis, two variants of the problem of exact search of patterns were combined: the $\delta\gamma$–matching problem and the order preserving matching problem. First, in Section 1.1 we

describe these two variants. Then, in Section 1.2 we define the problem considered in this thesis. The objectives are outlined Section1.3. Finally, the structure of the document is presented in Section 1.4.

1.1. Background

Exact string matching does not support all the applications. Therefore, many variants of the problem have been defined to tackle specific problems. For instance, in some areas the alphabet is drawn from a set of integer values. These integer strings are normally found in cipher text, financial data, meteorology data, image data, and music data, to name some. In such numeric strings, it would be unrealistic and ineffective to search for exact occurrences of a pattern but rather ought to search for similar instances of it. Then, some variants of the problem have been defined, including *$\delta\gamma$-matching* and *order-preserving matching*.

The *$\delta\gamma$–matching* problem consists of finding all the text windows in T for which: (i) the distance to the corresponding symbols in P is at most δ; and (ii) the sum of such distances is at most γ. In other words, the output of this problem is the set of positions i such that $|P_j - T_{i+j}| \leq \delta$, $0 \leq j \leq m-1$, 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. Numerous algorithms have been design and tested to resolve $\delta\gamma$–matching (see for instance [12, 26, 30, 21, 13, 25, 47, 20]). The approaches of these solutions make use of different techniques such as bit parallelism, dynamic programming, and heuristics based on occurrences (see [54] for a survey on solutions and related problems of $\delta\gamma$–matching). Recently, it has been used to make more flexible other string matching paradigms such as parameterized matching [48, 54], function matching [55] and jumbled matching [56, 57]. Cambouropoulos *et al.* [12] was perhaps the first to mention this algorithm motivated by Crawford's work *et al.* [24]. Variations of these works have been made to allow wildcards or also known as *do not care symbols* [22, 6], *transposition-invariant* [47] and gaps [14, 15, 35]. δ–matching and $\delta\gamma$–matching are also related to other string similarity metrics like L_1 and L_∞ also known as Manhattan distance and Chevyshev distance. For review of recent work in this area, see *e.g.* [3, 4, 49, 52, 50, 51, 31].

On the other hand, *order-preserving matching* considers the order relations within the numeric strings rather than the approximation of their values. In particular, 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, denoted as $rank_T(i)$, of string $T_{0\ldots n-1}$ is $1 + |\{T_j < T_i : 0 \leq j, i < n \wedge i \neq j\}| + |\{T_j = T_i : j < i\}|$. With definition of

the *rank* of a character in a string, we can define the natural representation of a string, denoted as *nr(T)* for any string T, as the concatenation of the ranks in the strings, i.e. $nr(T) = rank_T(0)rank_T(1)\cdots rank_T(n-1)$. And with the natural representation of a string of integers, we can define when a *order-preserving match* (**OPM**) occurs: Two string X and Y, both of the same length, have an OPM *iff* $nr(X) = nr(Y)$. In a similar fashion, we can say there is a OPM of a pattern $P_{0\ldots m-1}$ with T^i ***iff*** $nr(P) = nr(T^i)$.

Then, order-preserving matching consists of finding every text window in T such that its natural representation matches the natural representation of P. 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 [44]. The order preserving matching problem can be considered an evolution of studies in combinatorial patterns of permutations, although those had a different approach, in which they worked on avoiding patterns (see for example [2, 9, 11, 37, 38, 41, 45, 53, 58]).

Since Kim et al. [44] and Kubica et al. [46] defined the problem, it has gained great attention from several other researchers [28, 27, 18, 32, 29, 40, 17, 40, 39]. Some approaches to solve the order preserving matching problem include prefix tables and preprocessing using indexing structures such as suffix trees [29, 28, 27]. Advances have been made in the design of algorithms for its exact version, i.e. when there is a match between strings X and Y ***iff*** $nr(X) = nr(Y)$. Results have been obtained in versions with preprocessing and indexing of the text [29], as well as in approximate versions with k errors [36]. It has also been shown that exact order preserving matching in permutations as a subsequence is a $NP-Complete$ problem, although some special cases have polynomial solutions [2, 38, 41]. Since its first solutions [44, 46], the exact order preserving matching problem has had polynomial solutions with practical implementations [8, 19, 18, 32, 40].

1.2. Definition of the Problem

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 Gawrychowski and Uznański [36]. 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 thesis, 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.

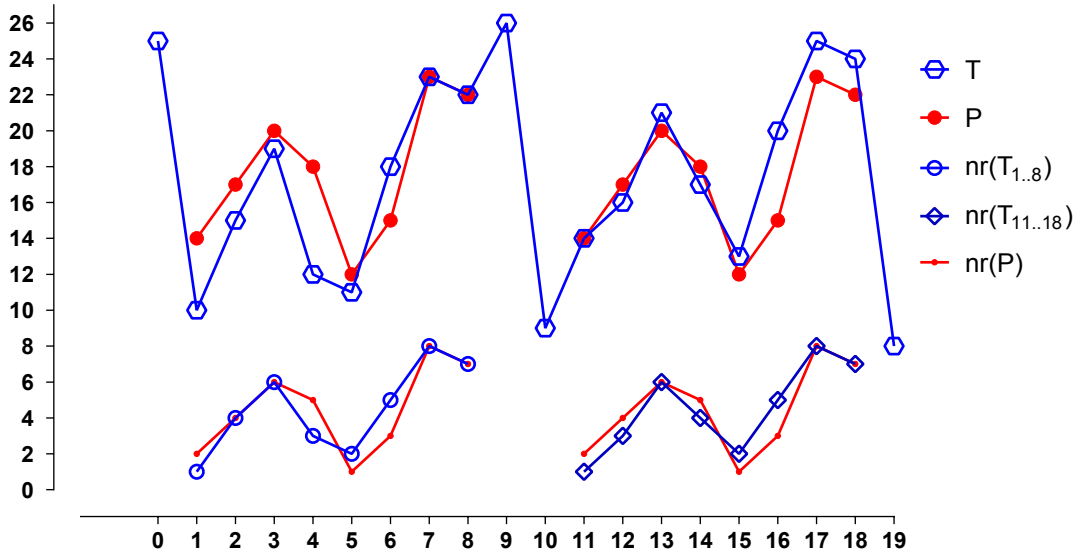

Figure 1-1.: Order preserving matching under $\delta\gamma$ approximation example.
In this example we can see the text T, the pattern P in two different positions. We also show below the natural representations of the pattern, and the natural representation of the two search windows similar to the pattern.

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. **1-1** 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. **1-1**).

1.3. Objectives

Define and solve an approximate variant of the order preserving matching problem under the δ and γ approximation distances.

1. Formally define the order preserving matching problem under the δ and γ distances.

2. Design at least two polynomial algorithms that solve the order preserving matching problem under the δ and γ distances.

3. Design and develop an experimental scheme that allows to analyze the efficiency of the proposed algorithms.

1.4. Document structure

In Chapter 2 we explain the fundamental concepts and definitions of the the area of study. We define the string matching problem, the approximate matching problem under δ and γ distances and the order preserving matching problem. Due to the fact that two of the four proposed algorithms are based on data structures, we include in this section the concepts, definitions and operations of those structures: the segment tree data structure, and the Fenwick tree also known as Binary Indexed Tree data structure (BIT from now on).

In Chapter 3 we construct the fundamental notation, definitions to formally define the new problem, the **Order preserving matching problem under $\delta\gamma$ approximation**. We define the subproblems and terminology needed to fully and formally define the new problem.

In Chapter 4 we show and explain the four algorithms that solve the new problem. We show the pseudo code of each one of them and calculate the theoretical upper bound complexity for each one of them.

In Chapter 5 we describe the experiment performed to compare the four algorithms. The experiments were done with two main objectives: To revise the correctness of all the algorithms and to compare the time complexity.

In Chapter 6 we show two applications with real data in two very different fields. One in music and other in finances. The music applications show how the metric can be applied to find similar sections of a melody inside the melody itself. We choose the very famous and well known Imperial March from the Star Wars movie franchise. We search a portion of the Imperial March in the complete melody and found the most similar portions, and those retrievals coincide with the portions that are considered similar for several professional musicians consulted. For the application in finance, we try to find similar changes of the stock prices of Facebook. Specifically, we took as the pattern the changes of a given 21-day interval period and searched for its matches as intervals in the stock prices of Facebook in history. We found that are very similar intervals for relatively low values of the similarity parameters (δ and γ). Finally in Chapter 7 we describe what could be the next steps in developments in this area, and questions that remain open.

2. Preliminaries

In this chapter we explain the fundamental concepts and definitions needed to formally define and solve the $\delta\gamma$-order preserving matching problem ($\delta\gamma$–**OPMP**). We start by giving the fundamentals on stringology in Section 2.1, then we explain and define the $\delta\gamma$ matching problem ($\delta\gamma$–**MP**) in Section 2.2 and the order preserving matching problem (**OPMP**) in Section 2.3. Furthermore, in Section 2.4 we describe the data structures that will be used in Chapter 4 to solve the $\delta\gamma$–**OPMP**.

2.1. String fundamentals

A string is a sequence of zero or more symbols from an alphabet Σ; the string with zero symbols is denoted by ϵ. The cardinality of alphabet Σ, denoted by $|\Sigma|$, is the number of characters in Σ. The set of all strings over the alphabet Σ is denoted by Σ^*. Throughout the thesis, we consider the numeric alphabet Σ_σ which is assumed to be an interval of integers from 1 to σ, i.e. $\Sigma_\sigma = \{1, 2, ..., \sigma\}$ where $|\Sigma| = \sigma$. A *text* $T = T_{0...n-1}$ is a string of length n defined over Σ_σ. T_i is used to denote the *i-th element* of T, $T_{i...j}$ is used as a notation for the *substring* $T_iT_{i+1} \cdots T_j$ of T, where $0 \leq i \leq j \leq n-1$. Similarly, a *pattern* $P = P_{0...m-1}$ is a string of length m defined over Σ_σ. For easy notation, we use T^i to denote the length-m substring of T starting at position i; thus $T^i = T_{i...i+m-1}$. Next, we present the definition of $\delta\gamma$-match and order-preserving match for the *string comparison problem*.

2.2. $\delta\gamma$–matching problem ($\delta\gamma$–MP)

Pattern matching under δ and/or γ has been studied for its application to real problems where similar pattern occurrences need to be encountered; that is, where errors are allowed under certain restrictions. This pattern matching is done over strings with integer alphabets since with them you can get symbol-to-symbol differences and similarity measures based on them. The parameter δ sets the maximum allowable difference between each character in the pattern and the corresponding character in the text search window. On the other hand, the γ parameter represents the maximum amount allowed in the sum of these differences. Now we are going to formally define what is a $\delta\gamma$–**match** and based on that definition we are going to define the $\delta\gamma$–matching problem ($\delta\gamma$–MP):

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
T_i	4	3	9	**4**	7	5	**3**	5	2	3	6	3	**8**	5	1	**4**	2	9
j				*0*	*1*	*2*	*3*	*4*				*0*	*1*	*2*	*3*	*4*		
P_j				**3**	7	5	**1**	5				3	**7**	5	1	**5**		
$\lvert T_i - P_j \rvert$				**1**	0	0	**2**	0				0	**1**	0	0	**1**		
$\sum_{j=0}^{m-1} \lvert T_{i+j} - P_j \rvert$						**3**								**2**				

Table 2-1.: $\delta\gamma$–Matching example.

Definition 1 ($\delta\gamma$–match) *Let $X = X_{0\ldots m-1}$ and $Y = Y_{0\ldots m-1}$ 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 $\delta\gamma$–match, denoted as $X \overset{\delta\gamma}{=} Y$, **iff** $\text{máx}_{j=0}^{m-1} |X_j - Y_j| \leq \delta$ and $\sum_{j=0}^{m-1} |X_j - Y_j| \leq \gamma$.*

Example 1 *There is a $\delta\gamma$–match, for $\delta = 2$ and $\gamma = 7$, between the strings $X = \langle 1,3,\ 1,3,6,\ 3,3,4,\ 1,2\rangle$ and $Y = \langle 2,2,\ 1,3,4,\ 3,4,5,\ 2,2\rangle$ defined over Σ_6 as $|X - Y| = \langle 1,1,\ 0,0,2,\ 0,1,1,\ 1,0\rangle$. Note that the maximum difference between corresponding characters is 2 and takes place at the fifth position. Similarly, the sum of all differences is 7.*

Problem 1 ($\delta\gamma$–matching problem ($\delta\gamma$–MP)) *Let $P = P_{0\ldots m-1}$ be a pattern string and $T = T_{0\ldots n-1}$ be a text string, both defined over Σ_σ. Also, let δ, γ be two given numbers ($\delta, \gamma \in \mathbb{N}$). The $\delta\gamma$–matching problem is to calculate the set of all indices i, $0 \leq i \leq n - m$, satisfying the condition $P \overset{\delta\gamma}{=} T^i$. From now on $\delta\gamma$–**MP**.*

In Table **2-1** we can see how the text $T = \langle 4,3,\ 9,4,\ 7,5,\ 3,5,\ 2,3,\ 6,3,\ 8,5,\ 1,4,\ 2,9\rangle$ has two occurrences of pattern $P = \langle 3,7,5,1,5\rangle$ in positions 3 and 11 (with $\delta = 2$ and $\gamma = 3$). These occurrences do not exceeds the limit given by $\delta = 2$, since the difference symbol to symbol (penultimate row of the table) is at most of 2, and also satisfy the limit given by $\gamma = 3$ since the sum of these differences for each occurrence is less than and equal to 3 (3 in the first and 2 in the second) in the last row of the table.

2.3. Order preserving pattern matching (OPMP)

Like the matching problem under parameters $\delta\gamma$, the *OPMP* can be seen as the task of finding a pattern P within a text T so that certain conditions are met. In this problem we also work with integer ordered alphabets. And consists of finding all the substrings (or positions i within the text T) that have the same relative order and length as the given pattern P. Formally (by taking the definition given in [17]), the problem can be defined from the order-isomorphism of strings: Given two strings u and v of the same length n over an ordered alphabet Σ, these strings are said to be isomorphic, written $u \approx v$ if and only if it is true:$u_i \leq u_j \iff v_i \leq v_j\ \forall_{0\leq i,j\leq n-1}$. Another form to determine order-isomorphism in strings is through the natural representation of a string presented in Chapter 1. For a string $T_{0\ldots n-1}$ the ranking of symbol T_i, denoted as $rank_T(i)$, is $1 + |\{T_j < T_i : 0 \leq j, i < n \wedge i \neq j\}|$

$+ |\{T_j = T_i : j < i\}|$. With definition of the *rank* of a character in a string, we can define the natural representation of a string, denoted as *nr(T)* and is the concatenation of the ranks in the strings, i.e. $nr(T) = rank_T(0)rank_T(1)\cdots rank_T(n-1)$. Now we present the definition of **order-preserving match** based no the natural representation of two strings of the same length, and based on that definition we formulate the **Order preserving matching problem**.

Definition 2 (order-preserving match) *Let $X = X_{0\ldots m-1}$ and $Y = Y_{0\ldots m-1}$ be two equal-length strings defined over Σ_σ. Strings X and Y are said to be a order-preserving match, denoted as $X \leftrightsquigarrow Y$, iff $nr(X) = nr(Y)$.*

Example 2 *Given integer strings $X = \langle 10, 15, 19, 12, 11, 18, 23, 22\rangle$ and $Y = \langle 12, 18, 22, 15, 13, 20, 30, 23\rangle$, $X \leftrightsquigarrow Y$ as $nr(X) = nr(Y) = \langle 1, 4, 6, 3, 2, 5, 8, 7\rangle$.*

Problem 2 (Order preserving–matching problem (OPMP)) *Let $P = P_{0\ldots m-1}$ be a pattern string and $T = T_{0\ldots n-1}$ be a text string, both defined over Σ_σ. Also, let δ, γ be two given numbers ($\delta, \gamma \in \mathbb{N}$). The OPMP is to calculate the set of all indices i, $0 \leq i \leq n - m$, satisfying the condition $P \leftrightsquigarrow T^i$. From now on **OPMP**.*

In the *OPMP* we want to locate all the substrings in the text T that are order-isomorphic with the pattern P. For example, the text $T = \langle 5, 7, 11, 10, 12, 15, 16, 9, 11, 10, 14, 17, 12\rangle$ in Figure **2-1**, the pattern $P = \langle 2, 4, 3, 6, 7\rangle$ (in blue) has two order-preserving matches. These occurrences are in the positions 1 and 7. The pattern is shown below the T text from the positions where the matching occurs. In this problem the *accuracy* refers to the pattern having the same *form* as the substring in T where the match is, although the symbols could be different from position to position in the window to be considered.

2.4. Data structures implemented and used

In this section we present two data structures we use in the algorithms we will present in later chapters. The first is the segment tree data structure, and the second is the Fenwick tree data structure. We will explain their operations we are going to use, and their complexities, mainly, the one we use in our algorithms. We also present both pseudo-codes of our implementations. The real C++ code can be seen in the Appendix C, D and E.

2.4.1. Segment Tree - Data Structure

The segment tree data structure is a powerful data structure with applications in many areas like in computational geometry [10, 1] and graph theory. The segment tree data structure uses the divide and conquer approach to answer queries in ranges of an underlying array

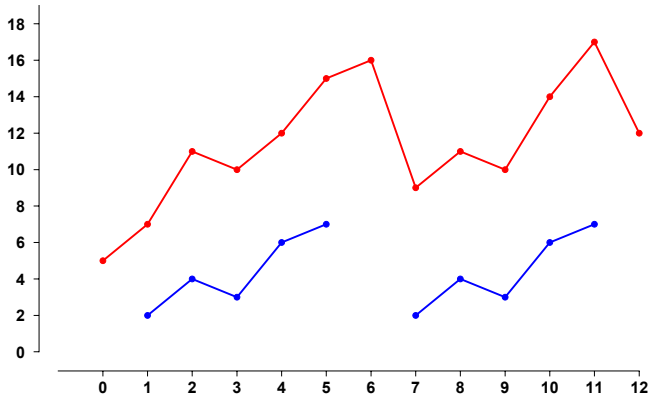

Figure 2-1.: Exact order preserving matching example
Exact order preserving matching for the text $T = \langle 5, 7, 11, 10, 12, 15, 16, 9, 11, 10, 14, 17, 12 \rangle$ and the pattern $P = \langle 2, 4, 3, 6, 7 \rangle$.

A. Every node in a Segment Tree is assigned a range and will contain the answer to the query for that specific range. Let node X have assigned the range $[a, b]$, so it's in charge of $A_a, A_{a+1}, \ldots, A_b$. If $a = b$, then X is a leaf; otherwise X has two children: $leftChild(X)$ in charge of the left half of $[a, b]$ and $rightChild(X)$ in charge of the right half (each child has a half of the interval $[a, b]$).

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 $O(\lg |A|)$. Figure **2-2** shows the segment tree for the array $A = \langle 90, 64, 65, 70, 66, 63, 70, 67 \rangle$. For example, the interval of the right child of the root in Figure **2-2** is labeled with 5, which means that the lowest value in the interval $[4, 7]$ is in that position (position 5), i.e. $A_5 = 63$, is the lowest element in the sub array $A_{4...7} = \langle 66, 63, 70, 67 \rangle$. Figure **2-3** shows the segment tree after updating A_4 to ∞ (in the implementation a big number can be used). Note that to update an element of A only the nodes from the root to the leaf containing the element can be changed (at most $O(\lg n)$ nodes). In the Figures **2-2** and **2-3** The squares below each leaf show the elements of the array A with their index bellow them. Every node of the segment tree is represented with a circle, with its range underneath that circle, and the index of the minimum value in the range inside the circle itself.

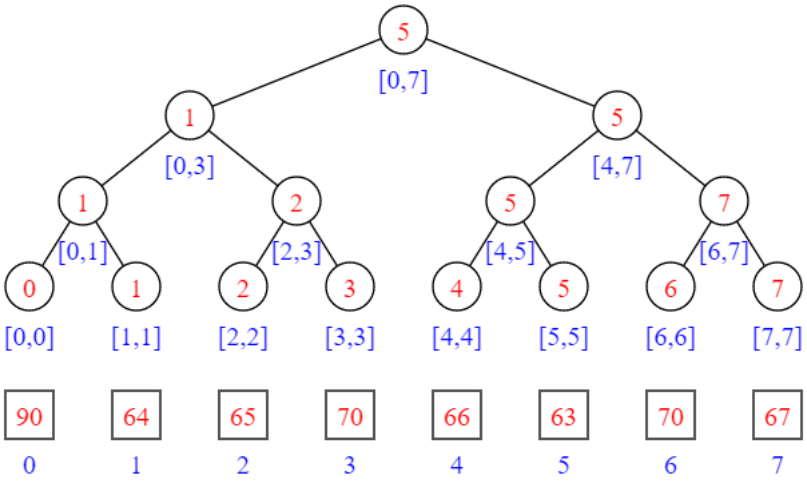

Figure 2-2.: Segment tree example.

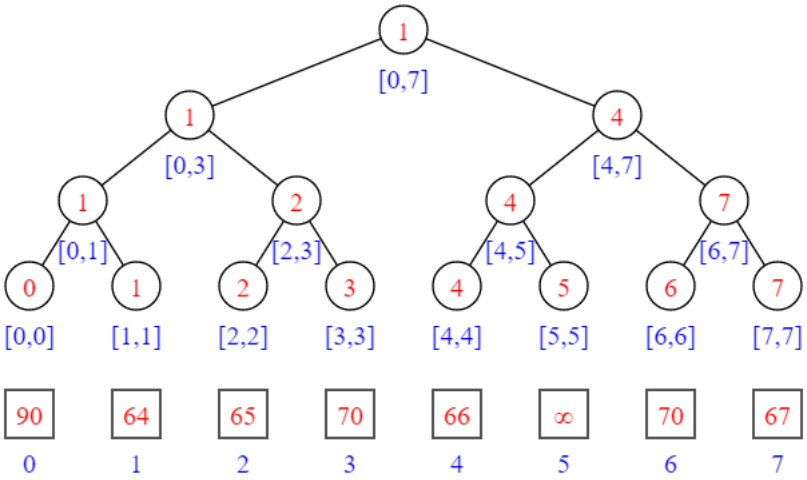

Figure 2-3.: Segment tree example after update.

Implementation

The data class *SegTreeNode* will be to stored a segment tree node. The fields stored will be:

- a: Start of the range the node is in charge, inclusive.
- b: End of the range the node is in charge, inclusive.
- *index*: Index of the minimum element of A in the range $[a, b]$. In case of ties, the left most is chosen.
- $leftChild$: Pointer to the left child of the node, or null if its a leaf.
- *rightChild*: Pointer to the right child of the node, or null if its a leaf.

In the pseudo-code we use the notation $newSegTreeNode(a, b, index, leftChild, rightChild)$ to instantiate a new $SegTreeNode$ with the given values for the fields. We will need the following procedures/methods:

- $buildSegTree(A, a, b)$: Builds a segment tree with the sub array $A_a, A_{a+1}, \ldots, A_b$ and returns the root node. The complexity is $O(n)$.
- $updateSegTree(node, i, x)$: Sets A_i to x. The complexity is $O(\lg n)$.
- $querySegTree(node, i, j)$: Returns the index of the minimum value among A_i, A_{i+1} , ... , A_j. If there are several minimum values, the leftmost (smallest index) is chosen. The complexity is $O(\lg n)$.

Function : $buildSegTree(A, a, b)$

Input: **integer array:** $A = A_{0...n-1}$, **integer:** a, b
Output: **SegTreeNode:** Root node of the Segment Tree
1. **Define:** $leftChild$**,** $rightChild$ **as SegTreeNode**
2. **if** $a = b$ **then return new** $SegTreeNode(a, a, a, null, null)$
3. $leftChild \leftarrow buildSegTree(A, a, \lfloor (a + b)/2 \rfloor$
4. $rightChild \leftarrow buildSegTree(A, \lfloor (a + b)/2 \rfloor + 1, b)$
5. **if** $A_{leftChild.index} <= A_{rightChild.index}$ **then**
6. **return new** $SegTreeNode(a, b, leftChild.index, leftChild, rightChild)$
7. **return new** $SegTreeNode(a, b, rightChild.index, leftChild, rightChild)$

Figure 2-4.: Segment tree construction pseudo code.

In Figure **2-4** we see how to recursively build a segment tree given an array A. The function $buildSegTree(A, 0, |A| - 1)$ will return the root of the segment tree. In line 2 we see how to build a segment tree of a subarray with a single element, we just need a leaf node. If there is more than one element in $[a, b]$ the node has two children (lines 1, 3 and 4) and each takes half the subarray, and then we get the index of the minimum element in $[a, b]$ by using the indexes of the minimums in the children (lines 5, 6 and 7). The $\leq$ sign in line 5 ensures that in case of a tie the left most will be chosen. The segment tree is built in $O(|A|)$ time and space, since there is $2 \times |A| - 1$ nodes and each node is built in constant time.

In Figure **2-5** we see how to change a value of A_i to x. First we go to the leaf node in charge of the range $[i, i]$ and assign x to A_i (lines 1 to 7). Then, as we backtrack we update the

Procedure : $updateSegTree(node, i, x)$

Input: **SegTreeNode:** node, **integer:** i, x
1. **if** $node.a = node.b$ **then**
2. $\quad A_i = \text{x}$
3. **else**
4. $\quad$ **if** $i \leq node.leftChild.b$ **then**
5. $\quad\quad$ updateSegTree$(node.leftChild, i, x)$
6. $\quad$ **else**
7. $\quad\quad$ updateSegTree$(node.rightChild, i, x)$
8. $\quad$ **if** $A_{leftChild.index} \leq A_{rightChild.index}$ **then**
9. $\quad\quad$ $node.index \leftarrow node.leftChild.index$
10. $\quad$ **else**
11. $\quad\quad$ $node.index \leftarrow node.rightChild.index$

Figure 2-5.: Segment tree update procedure pseudo code

index of every visited node with the indexes of its children (lines 8 to 11). This procedure works in $O(\lg n)$ since the height of the tree is $O(\lg n)$.

Function : $querySegTree(node, i, j)$

Input: **SegTreeNode:** *node*, **integer:** i, j
1. **Define:** $leftAns$, $rightAns$ **as integer**
2. **if** $node.a = i$ **and** $node.b = j$ **then**
3. $\quad$ **return** $node.index$
4. **if** $j \leq node.leftChild.b$ **then**
5. $\quad$ **return querySegTree(**$node.leftChild, i, j$**)**
6. **if** $i \geq node.rightChild.a$ **then**
7. $\quad$ **return querySegTree(**$node.rightChild, i, j$**)**
8. $leftAns \leftarrow$ querySegTree$(node.leftChild, i, node.leftChild.b)$
9. $rightAns \leftarrow$ querySegTree$(node.rightChild, node.rightChild.a, j)$
10. **if** $A_{leftAns} \leq A_{rightAns}$ **return** $leftAns$
11. **return** $rightAns$

Figure 2-6.: Segment tree query function pseudo code.

In Figure **2-6** we see how to get the index of the minimum value in the range $[i, j]$, it must hold that $node.a \leq i \leq j \leq node.b$. The base case is when the searched range equals the range of *node*, the answer is $node.index$ (lines 2 and 3). In lines 4 and 5 we see the case when the range searched is completely inside the left child, and similarly in lines 6 and 7 the case where the range is completely inside the right child. The only case left to consider is when a part of the range is in the left child and another part in the right child. In this case we recurse on both children (lines 8 and 9) and then pick the best answer out of the two (lines 10 and 11). Note that the $\leq$ on line 10 ensures that in case of a tie, the leftmost index will

be chosen. The complexity of this function is $O(\lg n)$ since on at most two nodes of each level the recursion will split, which means that at most 4 nodes of each level will be visited.

Optimization

Instead of using a data class, the whole segment tree can be stored in just a 1-indexed integer array T of size $2 * |A| - 1$. T_i will store the *index* of node i. The leaf node in charge of the range $[i, i]$ will be $i + |A|$, so $T_{i+|A|} = i$ for $0 \leq i < |A|$. This means that leaves are the nodes $|A|$ through $2 * |A| - 1$. There is no need to keep pointers or the range of each node, if we build T with this simple rule: The parent of i is $\lfloor i/2 \rfloor$. From this rule we know that the children of node i are $2 * i$ and $2 * i + 1$.

With this segment tree implementation we achieve the same complexities for building, querying and updating operations. But the execution time of the $segtreeBA$ algorithm that will be presented in Section 4.3 is halved by this optimization. Both implementations can be found in the Appendix C and Appendix D.

2.4.2. Fenwick tree (Binary indexed tree - BIT)

The Binary indexed tree (BIT) or Fenwick tree, proposed by Peter M. Fenwick in 1994 [33], is a data structure that can be used to maintain and query cumulative frequencies. In this section we explain the main ideas and operations of the BIT data structure that will be used later in this document. We also describe the complexities associated with the BIT operations we will use.

The BIT data structure keeps an abstraction of an array A with positions indexed from 1 to n. Initially the BIT is assumed to be full of zeros.

BIT - Operations

Here we present the two BIT operations of interest:

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

The BIT is in memory an array we are going to call $tree$, with position from 1 to n so that: $tree_i = A_{i-lbit(i)+1} + A_{i-lbit(i)+2} + \ldots + A_i$, where $lbit(x) = bitwiseAND(x, -x)$. Here $bitwiseAND(a, b)$ is the bit to bit logical *and* in the binary representation of two integers, and $-x$ is the two's complement of x. So $lbit(x)$ is a function that returns the value of the least significant bit of x in its binary representation (e.g. $lbit(10) = 2$ since since $6 = 1010_2$

Tree	5	8	1	13	5	8	1	24	1	6	4	16	4	4	6	57
Array	5	3	1	4	5	3	1	2	1	5	4	6	4	0	6	7
Indexes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Binary indexes	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000

Figure 2-7.: Binary indexed tree graphic example.

Function : $sumUpTo(tree, i)$

Input: **integer array:** $tree = tree_{1...n}$, **integer:** i
Output: $A_1 + A_2 + \ldots + A_i$
1. **Define:** $sum \leftarrow 0$ **as integer**
2. **while** $i > 0$ **do**
3. $\quad sum \leftarrow sum + tree[i]$
4. $\quad i \leftarrow i - bitwiseAND(i, -i)$
5. **return** sum

Figure 2-8.: Binary indexed tree function $sumUpTo(tree, i)$.

(here the subindex 2 means the base in which the number is written;10 in base 2), the least significant bit is 10_2 and has a value of 4. See Figure **2-7**.
With the tree array we can do $sumUpTo$ as shown in the function in Figure **2-8**. The complexity is in $O(\lg n)$, since by subtracting $bitwiseAND(i, -i)$ from i we get the binary representation of i without the least significant bit, so we remove one bit on each iteration. So the number of iterations is the number of bits 1 in i. As an example: $sumUpTo(tree, 11)$ will add the values of $tree_{11}$, $tree_{10}$ and $tree_8$ and return them.

Procedure : $addAt(bit, i, x)$

Input: **integer array:** $tree = tree_{1...n}$, **integer:** i, x
1. **while** $i \leq n$ **do**
2. $\quad tree_i \leftarrow tree_i + x$
3. $\quad i \leftarrow i + bitwiseAND(i, -i)$

Figure 2-9.: Binary indexed tree function: $addAt(bit, i, x)$.

In Figure **2-9** we see how to increment the value of A_i by x, we update all the elements of $tree$ that contain A_i in their sum (Line 2). The indexes of $tree$ to update are obtained starting with i and then incrementing i by its least significant bit on each iteration (Line 3).

What line 3 does is assign to i the next smallest index of $tree$ which sum contains the sum of $tree_i$ (e.g. if i starts at 5, the positions of $tree$ updated will be: 5, 6, 8 and 16, in that order, see Figure **2-7**). The complexity is $O(\lg n)$ since after each iteration the least significant bit of i becomes bigger.

3. Definition of $\delta\gamma$–order preserving matching problem ($\delta\gamma$–OPMP)

In this thesis it was proposed to combine the paradigms of patterns searching known as $\delta\gamma$–***matching*** and ***order preserving matching***. These two problems share some aspects, and can be formulated on the same basis:

- The search for a pattern P of length m is made on a text T of length n.
- The output of an algorithm is a series of positions within the text T: the positions where there is a match.
- Both the text and the pattern can be considered as consisting of symbols of an integer alphabet that could well be a subset of the natural numbers. This would make possible its application finance, text searches and musical information retrieval, since in all these fields, the abstraction of reality can be made by mapping the changes of values in time to a finite set of integers.
- Each of these two problems has different applications and advantages over the other depending on how a given practical problem is addressed.
- They also have a number of solutions that work well for each case.

Based on these similarities it was possible to formulate a new string pattern matching problem in integer sequences which combines the advantages of the approximate pattern matching based on δ and γ parameters with those of the order preserving matching problem. The result is a tool that would at least have the same applications possibilities as the order preserving matching applications, but with the flexibility to allow approximate matching under parameters δ and γ. In Section 3.1 we will explain the definition of the new problem: $\delta\gamma$–OPMP and in Section 3.2 we will give several examples to clarify the problem.

3.1. The proposal of order preserving matching under $\delta\gamma$ approximation

Given the possibility of combining the two mentioned problems ($\delta\gamma$–Matching and OPM) in a new variant of approximate pattern matching for integer alphabet, in this thesis we

formally formulate and design the definition and solutions of this new variation of the well known string matching problem. This problem, its solutions and applications will have the advantages of both approaches. In this variant the pattern to be searched has a relative order similar to that found in the text, and this similarity will be given by the δ and γ parameters. The application of the restrictions is done taking into account the differences in the ranking of each symbol in the pattern and the ranking of each symbol in the text window. This new problem is interesting algorithmically since it required the application of different techniques and data structures to design its solution, since the union of these two approaches has not been considered. Now 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 3 ($\delta\gamma$–order-preserving match) *Let $X = X_{0\ldots m-1}$ and $Y = Y_{0\ldots m-1}$ 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** $nr(X) \overset{\delta\gamma}{=} nr(Y)$.*

Example 3 *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 $nr(X) \overset{\delta\gamma}{=} nr(Y)$.*

Problem 3 ($\delta\gamma$–order-preserving matching ($\delta\gamma$–OPMP)) *Let $P = P_{0\ldots m-1}$ be a pattern string and $T = T_{0\ldots n-1}$ be a text string, both defined over Σ_σ. Also, let 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 $P \overset{\delta\gamma}{\leftrightsquigarrow} T^i$. From now on $\delta\gamma$–**OPMP**.*

3.2. Examples of $\delta\gamma$–OPMP

For the sake of clarity as to how the two variations of the pattern matching problem were combined ($\delta\gamma$–Matching and Order-preserving matching), see in Figure **3-1** in conjunction with Table **3-1** an example of text T and pattern P, and a $\delta\gamma$ match. In this example, at position 2 of the text T, there is an order preserving match under parameters $\delta = 2$ and $\gamma = 8$ of the pattern P. These restriction apply to the natural representation of the pattern, and the natural representation of T^2, i.e. $nr(P)$ and $nr(T^2)$. T^2, as we said before, is the the length– m search window starting at position 2 of the text T. The $\delta\gamma$ restriction are fulfilled since, as shown in the penultimate row of Table **3-1**, the maximum difference in the P and T rankings in the search window is at most 2 (δ) and the sum of the differences is 8 (γ in the last row). In Figure **3-2** we show the example given in Chapter 1 with given values in Table **3-2** for clarity.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
T_i	36	40	35	45	27	37	23	21	39	24	41	31	22	48	40	35
j			*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	
$nr(T^2)$			7	**12**	5	**8**	3	1	**9**	4	**11**	6	2	**13**	**10**	
P_j			30	41	27	40	22	21	34	22	45	27	21	44	42	
$nr(P)$			7	**10**	5	**9**	3	1	**8**	4	**13**	6	2	**12**	**11**	
$\lvert nr(T^2)_j - nr(P)_j \rvert$			0	**2**	0	**1**	0	0	**1**	0	**2**	0	0	**1**	**1**	
$\delta = max(\lvert nr(T^2)_j - nr(P)_j \rvert)$	**2**															
$\gamma = \sum_{j=0}^{12} \lvert nr(T^2)_j - nr(P)_j \rvert$	**8**															

Table 3-1.: Order preserving matching problem under δ and γ

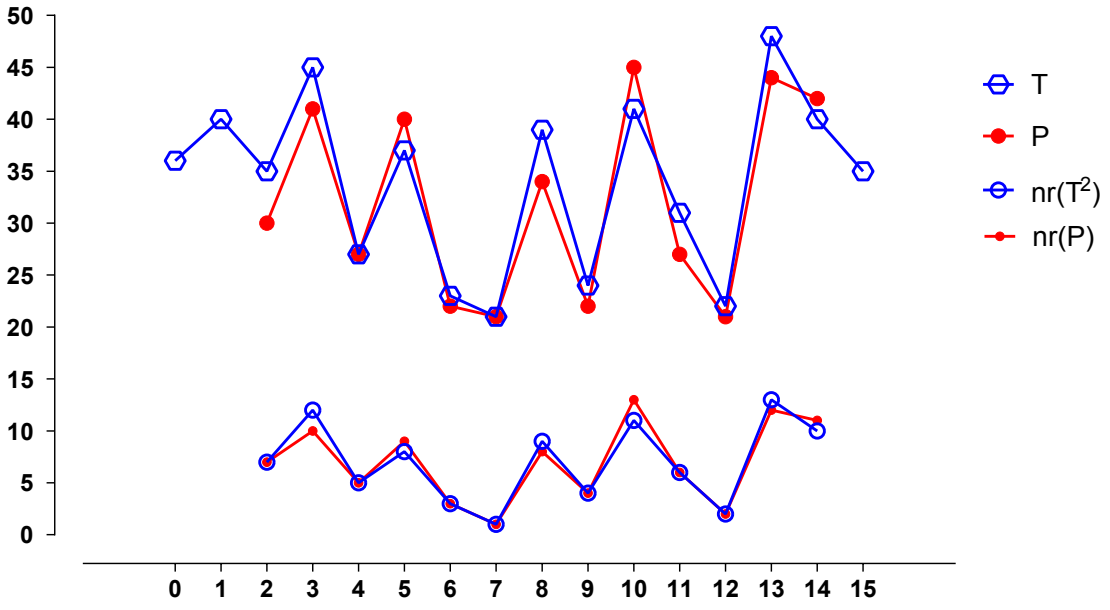

Figure 3-1.: Order preserving matching under δ and γ approximation example.

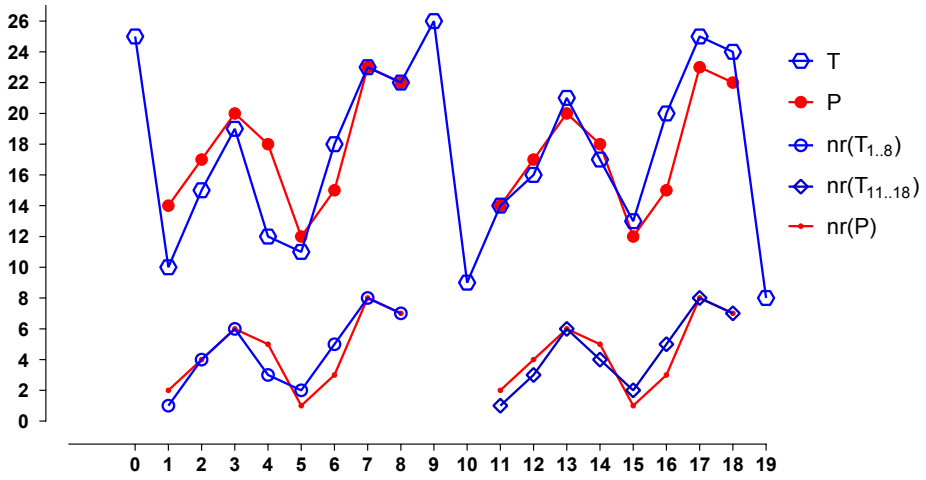

Figure 3-2.: $\delta\gamma$–OPMP example from introduction explained.

Example of $\delta\gamma$–order-preserving matching. In this example, $\delta = 2$, $\gamma = 6$ and integer strings $T = \langle 9, 10, 15, 19, 12, 11, 18, 23, 22, 26, 7, 14, 16, 21, 17, 13, 20, 25, 24, 8\rangle$ and $P = \langle 14, 17, 20, 18, 12, 15, 23, 22\rangle$ are defined over Σ_{26}. The X-axis and the Y-axis respectively correspond to the positions and values and rankings of both the pattern P and the substrings in T where there is two approximate order preserving matches with $\delta = 2$ and $\gamma = 6$ in positions 1 and 11. The figure in the lower side shows the similarity between the natural representation of the pattern and the natural representation of the substrings $T_{1...8}$ and $T_{11...18}$.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
T_i	**9**	**10**	**15**	**19**	**12**	**11**	**18**	**23**	**22**	**26**
$nr(T^1)$		1	4	6	3	2	5	8	7	
j		*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	
P_j		**14**	**17**	**20**	**18**	**12**	**15**	**23**	22	
$nr(P)$		2	4	6	5	1	3	8	7	
$\lvert nr(T^1)_j - nr(P)_j\rvert$		1	0	0	2	1	2	0	0	
$\delta = max(\lvert nr(T^1)_j - nr(P)_j\rvert)$						**2**				
$\gamma = \sum_{j=0}^{7} \lvert nr(T^1)_j - nr(P)_j\rvert$						**6**				
i	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
T_i	**7**	**14**	**16**	**21**	**17**	**13**	**20**	**25**	**24**	**8**
$nr(T^1)$		1	3	6	4	2	5	8	7	
j		*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	
P_j		**14**	**17**	**20**	**18**	**12**	**15**	**23**	**22**	
$nr(P)$		2	4	6	5	1	3	8	7	
$\lvert nr(T^1)_j - nr(P)_j\rvert$		1	1	0	1	1	2	0	0	
$\delta = max(\lvert nr(T^1)_j - nr(P)_j\rvert)$						**2**				
$\gamma = \sum_{j=0}^{7} \lvert nr(T^1)_j - nr(P)_j\rvert$						**6**				

Table 3-2.: $\delta\gamma$–OPMP example from introduction explained.

4. Proposed algorithms to solve the $\delta\gamma$–OPMP

In this chapter, we present four algorithms that solve the $\delta\gamma$–Order preserving matching problem ($\delta\gamma$–OPMP). All our algorithms work in a similar way to many string search algorithms (examples of pattern search approaches with sliding window can be found in [16]). There is always a sliding window to check if there is a match, from positions 0 until position $n - m$ of the text. A sliding window is a length–m substring of the text that *moves* in every iteration of the search algorithm by one or more positions. We assume that the size of the pattern is always much smaller than the size of the text ($m << n$).

The first algorithm, we call $naiveA$, is the naive algorithm with a complexity of $\Theta(nm \lg m)$. The naive algorithm tries all possible positions, and for each one of them verifies if there is a match in $\Theta(m \lg m + m)$ time. The complete description on this algorithm is presented in Section 4.1. The second algorithm, we call $updateBA$, is based on a linear update and verification, $\Theta(m)$, of the sliding window. The complexity of the second algorithm $\Theta(nm)$ and is presented in Section 4.2.

The third algorithm is based on the segment tree data structure, we call it $segtreeBA$. For each window, the algorithm iterates ranks from 1 to m checking if the $\delta\gamma$ restrictions hold. The complexity of the third algorithm is $O(nm \lg n)$. The last algorithm we present is based on the Fenwick tree data structure, also known as binary indexed tree or BIT (BIT from now on). We call the last algorithm: $bitBA$. It uses a BIT to calculate rankings in the sliding window and compare them to the rankings in the natural representation of the pattern. The complexity of the BIT-Based algorithm is $O(nm \lg n)$. The data structure based algorithms, both have a complexity of $\Omega(n \lg n)$, i.e. a better lower bound than the first two algorithms. the $segtreeBA$ is presented in in Section 4.3 and $bitBA$ in Section 4.4. The C++ code of all algorithms can be found in Appendix A ($naiveA$), Appendix B ($updateBA$), Appendix C ($segtreeBA$ implemented with pointers and classes), Appendix D ($segtreeBA$ implemented with arrays) and Appendix E ($bitBA$).

The difference between the third and fourth algorithm, besides the data structure used, is the order in which the search window is analyzed, and the way the individual ranks in the search window are calculated. In the segment tree–based solution the positions in the

sliding window are analyzed in ascending order of the their rank and not in the order they appear: first the smallest element in the sliding window, element with rank 1, then the second smallest element in the sliding window, element with rank 2, and so on, until the biggest element in the sliding window, with rank m. In the BIT–based Solution, the sliding window T^i is analyzed in the order that the symbols appear: first T^i_0, then T^i_1, and so on until T^i_m.

We now present some operations common to several algorithms:

- Random Access: All the random accesses to get the i–th symbol or element in a string, list or array; denoted in the algorithm with sub index (e.g. X_i) are considered to have $O(1)$.

- $isAMatch(Y, X, \delta, \gamma)$: Returns true iff $X \overset{\delta\gamma}{\rightsquigarrow} Y$. X and Y both of size m. When X and Y are original strings, the complexity of this function is $O(m \lg m + m)$. The term $m \lg m$ is due to the calculation of the natural representation of each string. The term m is due to the rank by rank comparison to check if the parameters δ and γ hold. If X and Y are the natural representation of two strings, i.e. each a permutation of the numbers between 1 and n, the complexity of this function is $O(m)$, because the only work it does is checking for every position if the restrictions δ and γ holds.

4.1. Naive algorithm ($naiveA$)

The first algorithm to solve the $\delta\gamma$–OPMP we present, is the naive algorithm (see Figure **4-1**). This algorithm iterates over all possible candidates $T^i, 0 \leq i \leq n - m$, and for each T^i decides in $\Theta(m \lg m + m)$ time if there is an $\delta\gamma$–order preserving match. The complexity can be calculated as follows. The cost of creating $P^{nr} = nr(P)$ in line 1 is $\Theta(m \lg m)$. The algorithm iterates over all $n-m+1$ possible candidates to a match. And for each one of them, it creates $T^{nr} = nr(T^i)$ in $\Theta(m \lg m)$ time and verifies if there is a match in $\Theta(m)$ time. The total complexity of the algorithm is then $\Theta((n-m+1)(m \lg m+m)+m \lg m) = \Theta(nm \lg m)$.

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

Input: $P = P_{0\ldots m-1}, T = T_{0\ldots n-1}, \delta, \gamma$
Output: $\{i \in \{0, \ldots, n-m\} : T^i \overset{\delta\gamma}{\rightsquigarrow} P\}$
1. **Create as Array:** $P^{nr} \leftarrow nr(P)$
2. **for** $i = 0 \rightarrow n - m$ **do**
3. $\quad T^{nr} \leftarrow nr(T^i)$
4. $\quad$ **if** $isAMatch(T^{nr}, P^{nr}, \delta, \gamma)$ **then report** i

Figure 4-1.: Naive algorithm: $naiveA$.

4.2. Update based algorithm ($updateBA$)

The second algorithm we present for the $\delta\gamma$–OPMP (see Figure **4-2**) has two phases: preprocessing and searching. In the preprocessing phase, we find the natural representation of the pattern and the natural representation of the first text window (i.e. T^0), which takes $\Theta(m \lg m)$. In the searching phase, we consider every candidate substring from T^0 to T^{n-m} to decide in $\Theta(m)$ time if it is a $\delta\gamma$–order preserving match under the given values. In particular, in order to achieve such complexity, we update some data structures to compute the natural representation of each text substring as the window slides. When we process the window $T^i = T_{i\ldots i+m-1}$, we update in $\Theta(m)$ the data structures needed to process the next search window. The total complexity is given by the cost of the preprocessing phase, plus the cost of iterating over all candidate positions, and for each one of them updating the natural representation of the search window and checking if there is a match. Then, the complexity of the proposed update based algorithm is $\Theta((n-m+1)m + m \lg m) = \Theta(nm)$.

The data structures that we use to be able to process each text window in $\Theta(m)$ time complexity are:

- An integer array P^{nr} with the natural representation of the pattern P. Its construction cost is $\Theta(m \lg m)$. The cost of the random access operations is $O(1)$. A native array in many programming languages will suffice.
- A list T^{nr} with the natural representation of each sliding window. Its construction cost is $\Theta(m \lg m)$. In each iteration of the main loop of the algorithm, the list will be updated in $\Theta(m)$ time. To get the desired complexity of the algorithm, the T^{nr} list must have a complexity of $O(1)$ (amortized or not) for the random access operation, adding elements at the end, and removing elements at the beginning. Such data structure could be, for example, a *deque* in the standard template library of C++ (see [23] for more on the C++ deque).

Operations in the list T^{nr}:

- $add(x)$: Adds an integer to the end. It has a complexity of $O(1)$.
- $removeFirst()$: It removes the first element. It has a complexity of $O(1)$.

The algorithm works as follows. In the preprocessing phase it creates the list T^{nr}: a list of size m with the natural representation of T^0, it cost $\Theta(m \lg m)$. It also creates the list P^{nr}, of size m, with the natural representation of the pattern P, with cost $\Theta(m \lg m)$. This phase has a time complexity of $\Theta(m \lg m)$. In the search phase, there is a verification of $\delta\gamma$ order preserving match at position i in the main loop. The verification phase is a very simple function with complexity $\Theta(m)$, which takes T^{nr} and P^{nr}, and checks if there is a $\delta\gamma$–order

preserving match (line 5). The update of the sliding window is done by removing the first element of T^{nr}, and then appending the new rank (rank of element T_{i+m} in the window), this will change some ranks in T^{nr} (lines 6 to 14 in Figure **4-2**). Finally, the last search window is checked for a $\delta\gamma$–OPM (line 15).

Algorithm 2: $\delta\gamma$–OPMP **updateBA**

Input: $P = P_{0\ldots m-1}, T = T_{0\ldots n-1}, \delta, \gamma$
Output: $\{i \in \{0,\ldots,n-m\} : T^i \overset{\delta\gamma}{\leftrightsquigarrow} P\}$
1. **Create as List:** $T^{nr} \leftarrow nr(T^0)$
2. **Create as Array:** $P^{nr} \leftarrow nr(P)$
3. **Define:** $rankToDelete$**,** $valueToAdd$**,** $rankToAdd$ **as integers**
4. **for** $i = 0 \rightarrow n-m-1$ **do**
5. **if** $isAMatch(T^{nr}, P^{nr}, \delta, \gamma)$ **then report** i
6. $rankToDelete \leftarrow T^{nr}_0$
7. $valueToAdd \leftarrow T_{i+m}$
8. $T^{nr}.removeFirst()$
9. $rankToAdd \leftarrow 1$
10. **for** $j = 0 \rightarrow m-2$ **do**
11. **if** $T^{nr}_j > rankToDelete$ **then** $T^{nr}_j \leftarrow T^{nr}_j - 1$
12. **if** $T^{nr}_{i+j+1} <= valueToAdd$ **then** $rankToAdd \leftarrow rankToAdd + 1$
13. **else** $T^{nr}_j \leftarrow T^{nr}_j + 1$
14. $T^{nr}.add(rankToAdd)$
15. **if** $isAMatch(T^{nr}, P^{nr}, \delta, \gamma)$ **then report** $n-m$

Figure 4-2.: Update based algorithm: *updateBA*.

4.3. Segment tree based algorithm ($segtreeBA$)

This solution, $settreeBA$, in Figure **4-3**, based on segment tree (see [1] for more on segment tree data structure), first calculates the natural representation of the pattern P (line 1 in Figure **4-3**). 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 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 Figure **4-3**), the values of T in the interval $\left[i, i+m-1\right]$ must be changed so 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 operations of the segment tree described in Chapter 2 to this solution. Here the segment tree is built on the node–type variable $minIndex$:

- $buildSegTree(T, 0, n-1)$: 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 $O(\lg n)$.

Algorithm 3: $\delta\gamma$–OPMP **segtreeBA**

Input: $P = P_{0...m-1}, T = T_{0...n-1}, \delta, \gamma$
Output: $\{i \in \{0, \ldots, n-m\} : T^i \overset{\delta\gamma}{\nsim} P\}$
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. $nChanges \leftarrow 0$

Figure 4-3.: Segment tree based algorithm: *segtreeBA*.

The complexity of *segtreeBA* can be computed as follows: In line 1 in Figure **4-3**, 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 1 to m using the segment tree. Finding the position of each rank in the window costs $O(\lg n)$. After each rank position finding, the algorithm checks if the $\delta\gamma$ restrictions

holds for the current window (lines 10 to 12). If so, it continue 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 cost $O(m \lg n)$ (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 $\Omega(\lg n)$. It is important to notice that this algorithm has a better lower bound than the first two algorithms (*naiveA* and *updateBA*). This because it can abort the analysis of a sliding window when it detects that either the δ or γ restrictions does not hold. The lower bound of this algorithm is then $\Omega(n \lg n)$, because in many cases it does not perform the m comparisons of 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 $\Omega(n \lg n)$.

4.4. Fenwick tree based algorithm (*bitBA*)

This algorithm, *bitBA*, showed in Figure **4-4**, uses a binary indexed tree (BIT or Fenwick tree) data structure (see [33] for more information on this data structure) to find the ranks in the sliding window. The BIT is a well known data structure that is mainly used to efficiently calculate prefix sums in an array of numbers. 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. 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 better lower bound in comparison to *naiveBA* and *updateBA*, it is: $\Omega(n \lg n)$ 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 Figure **4-4**).

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 Figure **4-4**). And the BIT must be updated so it does not consider the position i in the next search window (line 9 in Figure **4-4**).

Algorithm 4: $\delta\gamma$–OPMP **bitBA**

Input: $P = P_{0\ldots m-1}, T = T_{0\ldots n-1}, \delta, \gamma, \Sigma_\sigma$
Output: $\{i \in \{0,\ldots,n-m\} : T^i \overset{\delta\gamma}{\leftrightsquigarrow} P\}$
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** $i = 0 \rightarrow m-2$ **do**
5. $\quad addAt(bit, T_i^{nr}, 1)$
6. **for** $i = 0 \rightarrow n-m$ **do**
7. $\quad addAt(bit, T_{i+m-1}^{nr}, 1)$
8. $\quad isAMatch(i, bit, T^{nr}, P^{nr}, \delta, \gamma)$ **then report** i
9. $\quad addAt(bit, T_i^{nr}, -1)$

Figure 4-4.: BIT based algorithm: *bitBA*.

To understand better how *bitBA* works, consider for example consider the text $T = \langle 10, 5, 8, 12, 3, 9\rangle$ and the pattern $P = \langle 6, 2, 4\rangle$. In line 1 of Figure **4-4**, we create $T^{nr} = \langle 5, 2, 3, 6, 1, 4\rangle$ and $P^{nr} = \langle 3, 1, 2\rangle$ in line 2. The BIT is considered for this application as an encapsulation of an array with indices from 1 to n. In line 3 of the algorithm in Figure **4-4**, we create a BIT which is an encapsulation of the array: $bit = \langle 0, 1, 1, 0, 1, 0\rangle$. The 1 at positions 2, 3 and 5 (indexed from 1) are there because those three numbers are the first m (3 in this example) numbers in T^{nr}. To check the rank for example of $T_0 = 10$, in the natural representation of the first search window $T_{0\ldots 2}$, we must compute $sumUpTo(T_{i+j}^{nr}) = sumUpTo(T_{0+0}^{nr}) = sumUpTo(T_0^{nr}) = sumUpTo(5) = 3$. This can be seen in the *bit*, because the sum from position 1 to position 5 in *bit* is 3. In a similar fashion, we can compute the rank of T_2 in the first search window: $sumUpTo(T_{i+j}^{nr}) = sumUpTo(T_{0+2}^{nr}) = sumUpTo(T_2^{nr}) = sumUpTo(3) = 2$, and the rank of T_1 in the first search window: $sumUpTo(T_{i+j}^{nr}) = sumUpTo(T_{0+1}^{nr}) = sumUpTo(T_1^{nr}) = sumUpTo(2) = 1$. In other words, each call to $sumUpTo(T_k^{nr})$ counts the number of symbols lower or equal to T_k that are in the current window.

Function : $isAMatch(i, bit, T^{nr}, P^{nr}, \delta, \gamma)$

Input: $i, bit, T^{nr}, P^{nr}, \delta, \gamma$
Output: $true$ **if:** $T^i \overset{\delta\gamma}{\rightsquigarrow} P\}$, $false$ **otherwise**
1. **Define:** $gamma \leftarrow 0$ **as integer**
2. **for** $j = 0 \rightarrow m - 1$ **do**
3. $rank \leftarrow sumUpTo(bit, T^{nr}_{i+j})$
4. $delta \leftarrow |P^{nr}_j - rank|$
5. $gamma \leftarrow gamma + delta$
6. **if** $delta > \delta \vee gamma > \gamma)$ **then return** $false$
7. **return** $true$

Figure 4-5.: Function: $isAMatch(i, bit, T^{nr}, P^{nr}, \delta, \gamma)$.

So we can see that the natural representation of any search window can be calculated symbol by symbol using the operation of the BIT data structure, and in some cases it can ignore some rank calculations when the parameters δ and γ are relatively low. In those cases, i.e. when is very unlikely that in a given application a match occurs, the algorithm will very often break the search in a given window in line 6 of the function in Figure **4-5**.

5. Experiments

In this section, we describe the experimental setup we designed to evaluate the performance of the proposed algorithms. We compare the four algorithms. The algorithms to compare, *naiveA*, *updateBA*, *segtreeBA* and *bitBA* have similar theoretical complexities. The naive algorithm have a theoretical complexity of $\Theta(nm \lg m)$, the algorithm based on a $\Theta(m)$ update of the every search window in linear time, *updateBA* has a complexity of $\Theta(nm)$, and the algorithms based on data structures both have complexities $O(nm \lg n)$ for the worst case and $\Omega(n \lg n)$ lower bound. In Section 5.1 we present the experimental framework, while we describe the data generation in Section 5.2. Then, in Section 5.3, we discuss the results obtained. Finally in Section 5.4 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.

5.1. Experimental setup

In Section 5.1.1, we describe the hardware and software used for the experiments. Then, we show how we vary the input parameters in Section 5.1.2.

5.1.1. 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 @ 2.10GHz 2.69 GHz and installed RAM memory of 8GB. 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.

5.1.2. 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

	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	$[12, 240]$ $\Delta\sigma = 12$

Table 5-1.: Experimental values of n m, δ, γ and σ.

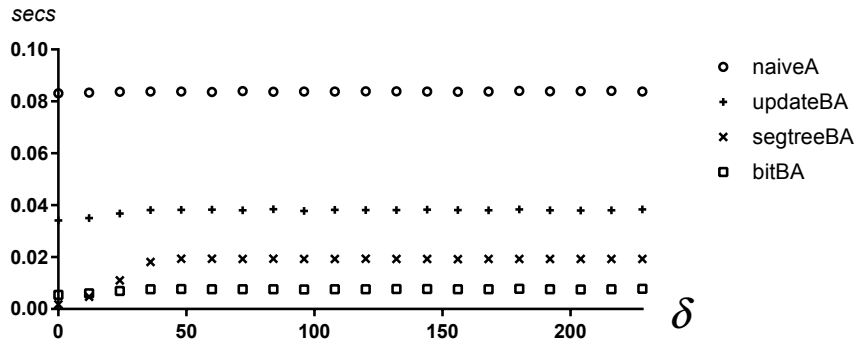

Figure 5-1.: Experiments varying the parameter δ.

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 **5-1**.

5.2. 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.

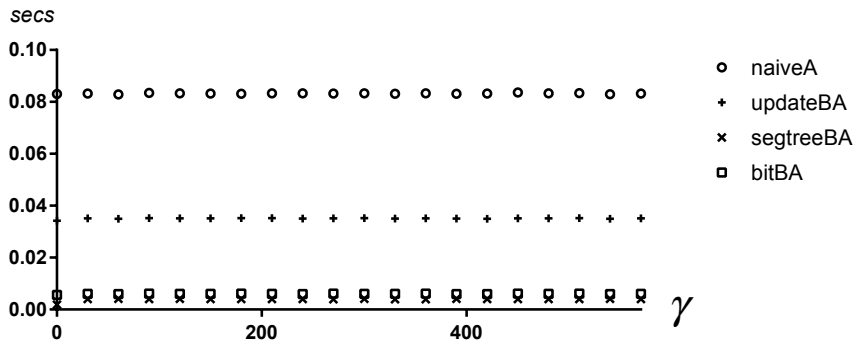

Figure 5-2.: Experiments varying the parameter γ.

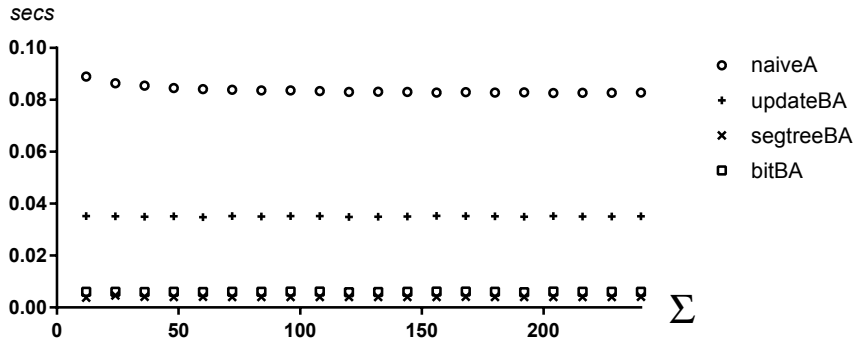

Figure 5-3.: Experiments varying alphabet size $|\Sigma|$.

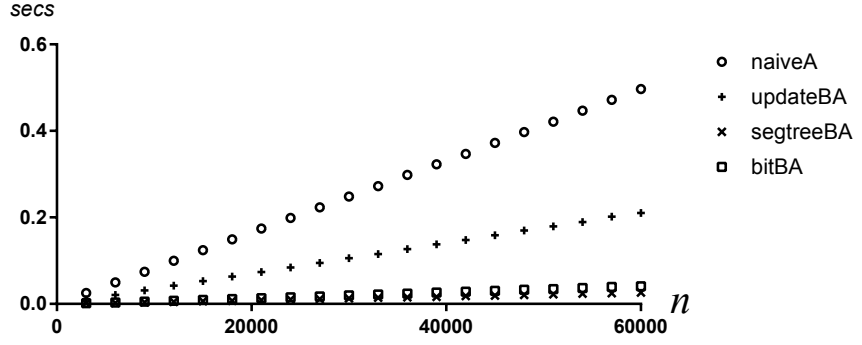

Figure 5-4.: Experiments varying text size n.

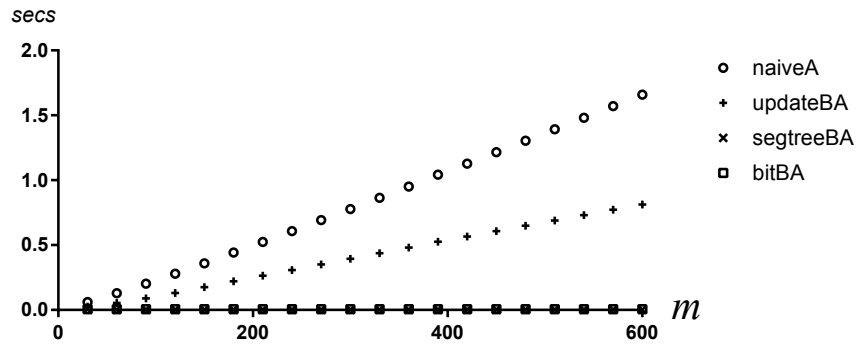

Figure 5-5.: Experiments varying pattern size m.

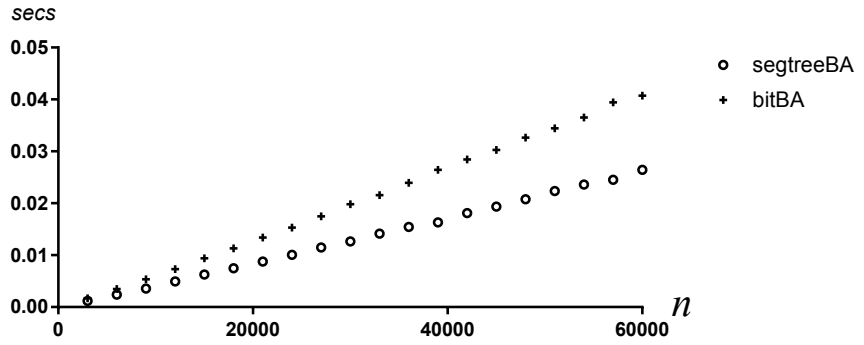

Figure 5-6.: Experiments varying text size n ($bitBA$ vs $segtreeBA$).

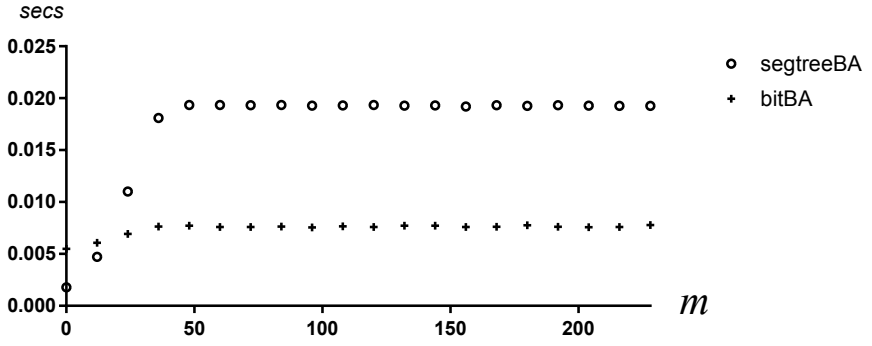

Figure 5-7.: Experiments varying pattern size m ($bitBA$ vs $segtreeBA$).

5.3. Experimental results and analysis

The first result to highlight is the fact that, in every experiment, is that the naive algorithm always have the worst performance, as expected. The results shown in Figures **5-2**, **5-3** and **5-1** 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 **5-4** and **5-5** 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 Figure **5-4**, m is a constant and n is a variable while in Figure **5-5**, n is a constant and m is a variable. 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, (Figures **5-4** and **5-5** show the behavior of all four algorithms while Figures **5-6** and **5-7** only show the best two algorithms on the same data), which are the main variables in the complexities, 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 algorithm 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. In Chapter 2 we show the classes based implementation, and for the experiment we use both. 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.

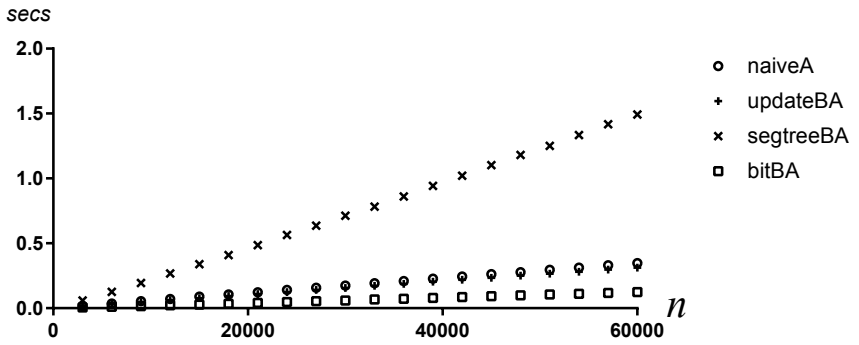

Figure 5-8.: Experiments varying pattern size n (Worst case $bitBA$ and $segtreeBA$).

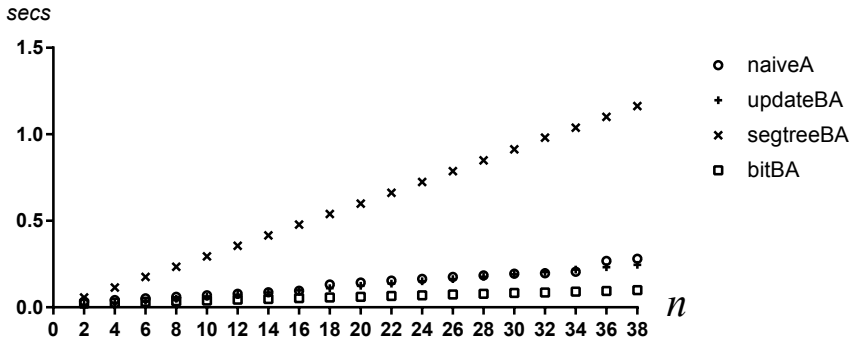

Figure 5-9.: Experiments varying pattern size m (Worst case $bitBA$ and $segtreeBA$).

5.4. 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 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 both P and T are strictly increasing or both 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 Figures **5-8** and **5-9**.

6. Applications

In this chapter we show a couple of applications of the defined problem, in music and finance. For music we show how $\delta\gamma$–**OPMP** can be applied to the searching of portions of similar melodies in a music piece. For finance we show how similar periods of time in the stock prices of the Facebook company can be found.

6.1. Application in music

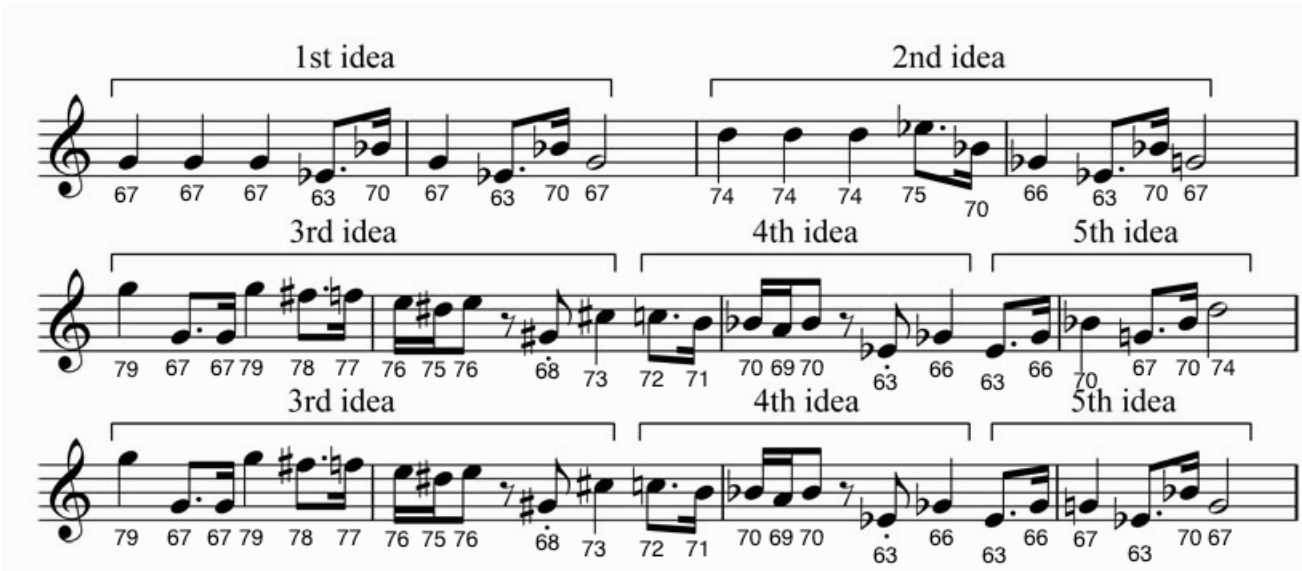

Figure 6-1.: Darth Vader's theme from Star Wars. by John Williams (Excerpt).
Here we show the melody of the Imperial March, we took the image of the score of the melody from [34] and added the MIDI numbers below.

For the music example, we choose the main theme from The Imperial March, soundtrack of the film series Star Wars [43] composed by John Williams [42] also known as the Darth Vader's theme because it represents him. This melody sounds every time this villain has a significant scene. Here we use an integer alphabet abstraction of a music piece, where each note of the melody is an integer. This abstraction of music takes into account only the pitch leaving out other aspects as silences, note duration, harmony, or instrumentation, but it gives a very good idea of the possible applications in music retrieval.

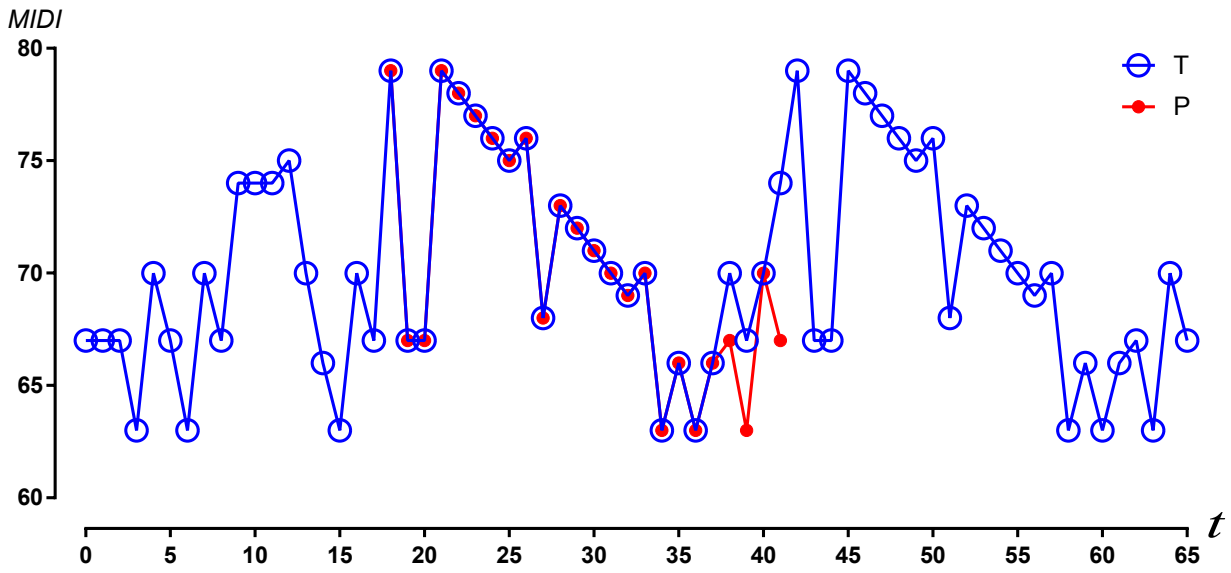

Figure 6-2.: T and P in the music application example.

The alphabet for music applications could be for example the given by the MIDI (Musical Instrument Digital Interface) technical standard [59, 7]. In the MIDI standard, the first note, 0 is a C note of the octave 0 (the lowest octave), note 1 is a $C\#$ of the same octave and so on. There could be up to note 127 which will be a G in the $10th$ octave.

In Figure **6-1** we show the excerpt of the Darth Vader's theme, in Figure **6-2** we show the melody in MIDI standard and the pattern to search. We can see in Figure **6-2** how similar the pattern to search and the text window where the $\delta\gamma$–OPM is found. In Figure **6-3** we show the natural representation of both the pattern P and the search window where the match is found. Here we show an example involving searching for substrings similar to a pattern that is also in the text. In the example, there was only one match found in all the melody. There we show the natural representation of the pattern and the search window that have the match: $T^{18} \overset{\delta\gamma}{\leftrightsquigarrow} P$. Here $\delta = 8$ and $\gamma = 32$. The music similarity of the match was verified by Gabriela Rojas, a professional musician from the National University of Colombia Conservatory. This gives an idea of possible applications in musical retrieval of approximate string matching. This can be useful for the advanced music students in order to help them with the theoretical analysis of the scores so they can look for melodic similarities or differences either in the same piece or comparing different pieces. Another application could also be the design of plagiarism detection software.

We draw an example of $\delta\gamma$–OPM with the same musical excerpt. For the example, we search the pattern $P = \langle 79, 67, 67, 79, 78, 77, 76, 75, 76, 68, 73, 72, 71, 70, 69, 70, 63, 66, 63, 66, 67, 63,$

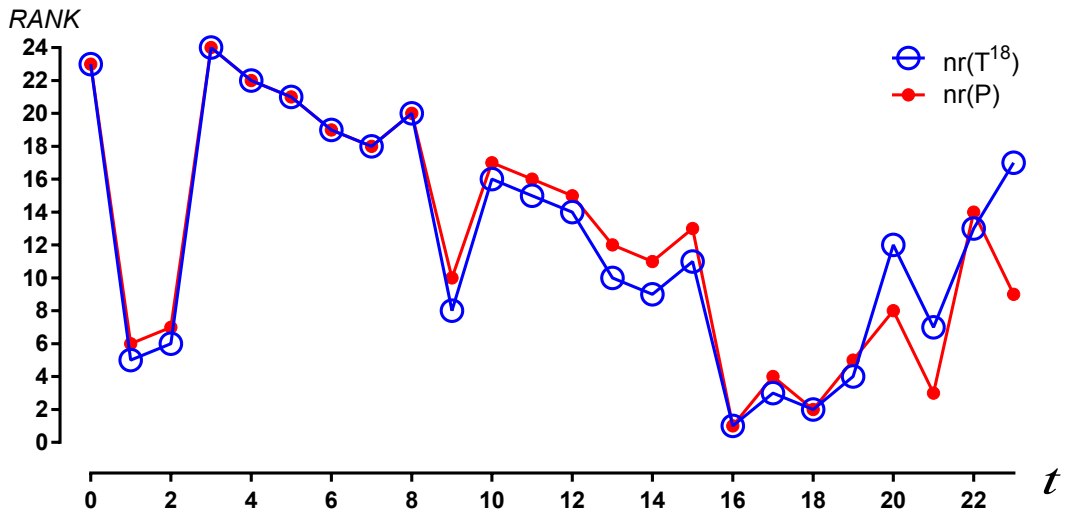

Figure 6-3.: P and search window natural representation in the music application example.

$70, 67\rangle$ in the given text for a $\delta = 8, \gamma = 32$ we found a match in position 18 of the text. This pattern as illustrated in the Figure **6-2** is also part of the main melody; the pattern appears at position 42, which for professional musicians, and even non professional musicians, sounds very similar to the found match.

6.2. Application in finance

For the finance application we choose to analyze the stock price of the Facebook company. We take the history of the stock price of Facebook from the beginning to April 2017. We take 21 days of recent Facebook stock price changes as the pattern. Specifically, the pattern to search is the 21 days period starting in February 28 2017 up to March 28 2017. Take into account that not all days the stock actions change, for that reason we choose 21 days which is approximately the amount of days the stock actions change in a month. The text to consider is the day to day stock price changes from May 18 2012 to March 31 2017 (The size of this text is 1225).

In Figure **6-4** we can see the representation of the pattern P. In Figure **6-5** we can see the representation of a portion of the text with the search window we found similar to the given pattern P. We can notice that the pattern in Figure **6-4** is similar in shape to the search windows highlighted with filled circles iin Figure **6-5**. It is important to see that the similarity holds despite the absolute values are very different.

In Figure **6-6** we can see the natural representation (or ranks) of both the pattern P and the natural representation of the search window that we found to be the most similar to

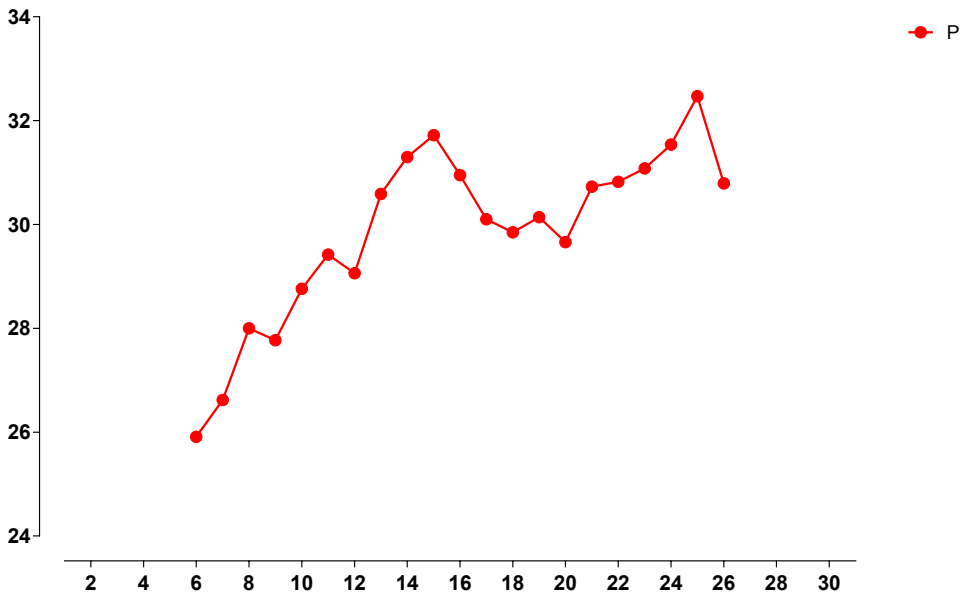

Figure 6-4.: P in the financial example.

the pattern. We can see that they both have a similar structure. Finally in Figure **6-7** we can see the portion of the text with the highlighted search window and the search pattern. In this last figure we omit the y and x axes labels because we want to show the similarity in shape of the pattern and the search window, not the similarity in absolute values which indeed is quite different. In fact the values in the pattern to search (first month of Facebook stock prices) are values lower than 34 and the search window found has values greater than 100.

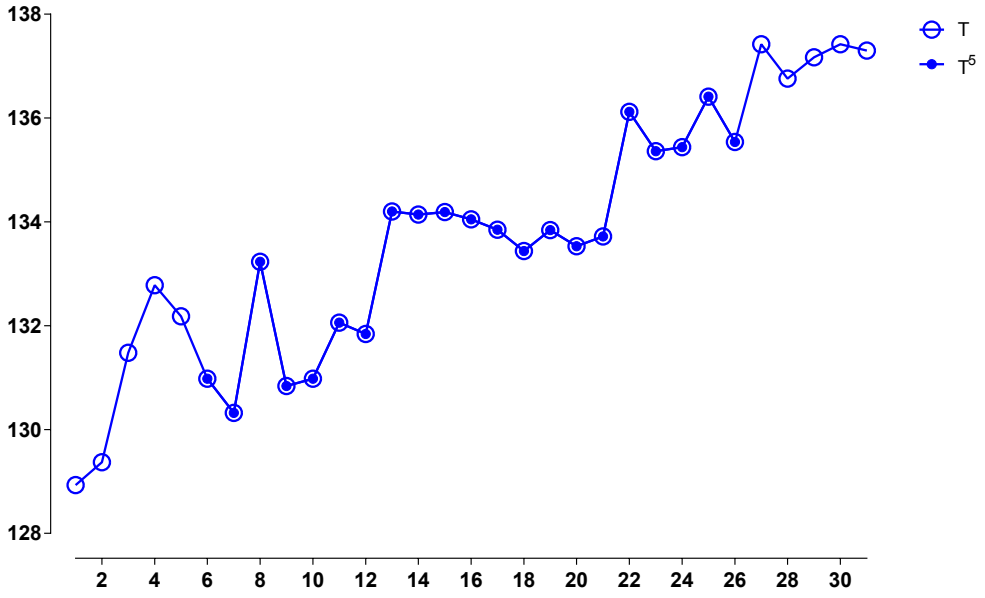

Figure 6-5.: T in the financial example.

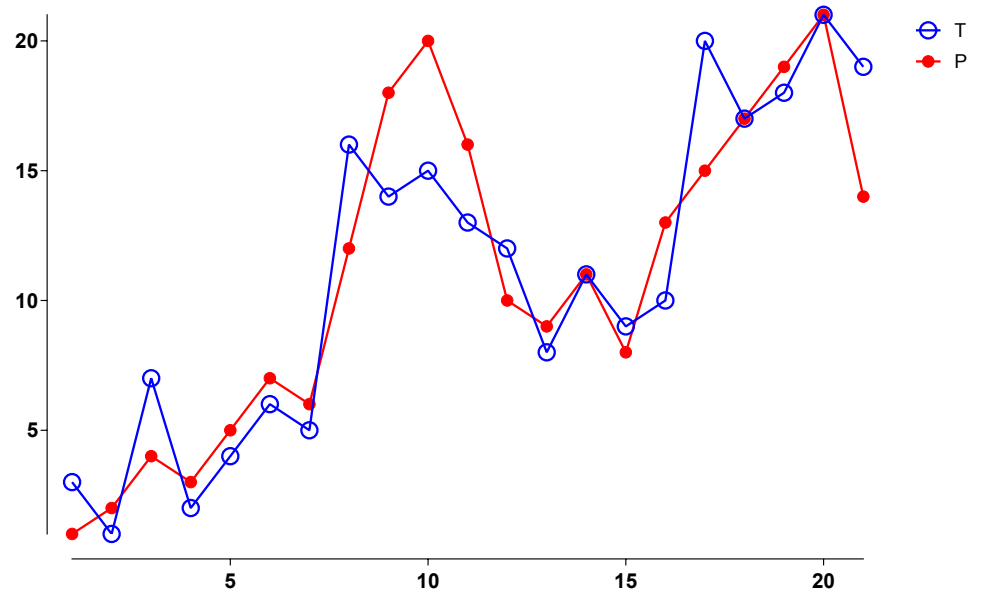

Figure 6-6.: Natural representations of T and P in the financial example.

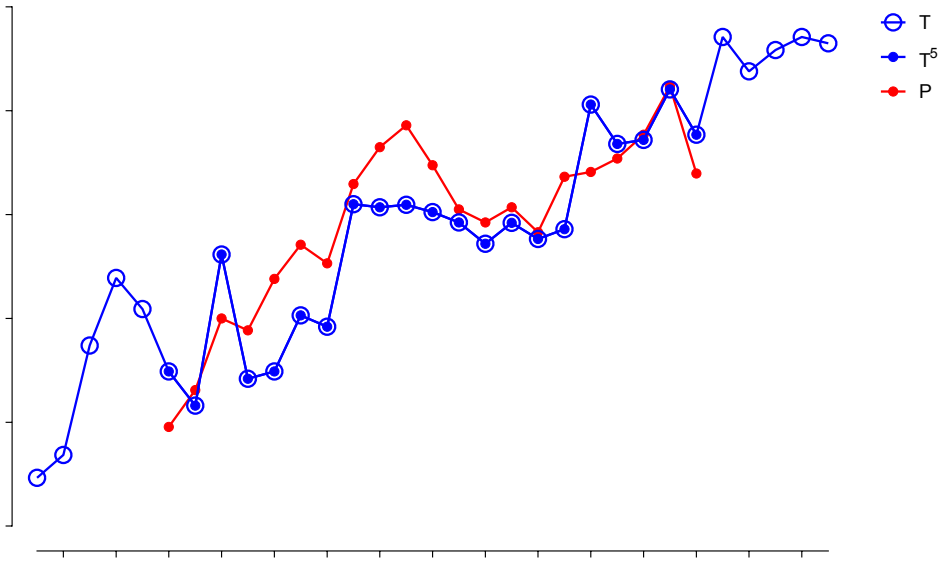

Figure 6-7.: T, P, and search window in the financial example of $\delta\gamma$–OPMP.

7. Conclusions and future work

We successfully define a new variant of the string matching problem, the $\delta\gamma$–**order preserving matching problem** ($\delta\gamma$–OPMP). This new variant gives 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 and implemented in C++ four algorithms that solve the $\delta\gamma$–**OPMP**. The naive algorithm ($naiveA$), the updated based algorithm ($updateBA$), the segment tree based algorithm ($segtreeBA$) and the Binary indexed tree based algorithm ($bitBA$). Their theoretical time complexity is $\Theta(nm \lg m)$ for the $naiveA$, $\Theta(nm)$ for the $updateBA$. The $segtreeBA$ and $bitBA$ both with complexities $O(nm \lg n)$ for their worst case and a $\Omega(n \lg n)$ lower bound.

We also designed an experimental setup to compare the designed algorithm and see how they behave in practice with randomly generated data. The experimental results show that in many cases, given the uniformly data generation, the data structure based 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 obtain.

We show experiment results on the worst cases of the $bitBA$ and $segtreeBA$. We conclude that the degradation in performance in the $segtreeBA$ algorithms is much more notorious than the degradation of $bitBA$. A question remains, and is if we can device an experimental setup where the better 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.

We show two applications with real data in music and finance. In music we use our findings to search for a portion of a melody in the melody itself; the melody we choose was the Imperial March of the Star Wars franchise. We show graphically how similar the portion to search and the match we found are similar. Those two portions of the melody are also very

similar according to professional musicians consulted. For the financial applications, we take a recent 21 day interval of stock actions of the Facebook company, and search for the most similar 21 day interval in all the history of the Facebook stock exchange history. We show how similar the changes despite the difference in their absolutes values.

More general experiments could be designed to test the efficiency of all the algorithm with real data, that could be financial data, music data or real data in other areas that are still yet to be determined. Finally, taking into account the way we generate the pseudo–random data for the experiments, i.e. all symbols with the same probability, we can conjecture that for specific applications more efficient algorithms could be designed based on the particularities of the chosen field (alphabet, language, etc).

Other aspect left to explore related to the applications showed are whether or not the advances in this thesis can be extended to more complex ideas. For example, in the finance application, the tools presented here could help to device or complement algorithms/techniques to make predictive analysis of stock price changes. In music, our contributions can be useful to design tools for the advanced music students in order to help them with the theoretical analysis of the scores so they can look for melodic similarities or differences either in the same piece or comparing different pieces. Also composers could see the $\delta\gamma$–OPMP, as a tool to check the perception they have about the similarity of musical ideas developed in different ways in one or several pieces of their own. For the musicologist this could be a way to track the development of one composer's musical ideas throughout their life and to analyze the way the composer evolves. Finally, another application could also be the support the design of the design of plagiarism detection software.

A. Appendix: C++ code of the naive algorithm ($naiveA$)

```
#include <bits/stdc++.h>
#include "Solver1.h"
#include "Utils.h"
using namespace std;

vector<int> Solver1::findAllDeltaGammaMatches(
        const vector<int>& T,
        const vector<int>& P,
        const int delta,
        const int gamma) {

    const int n = T.size();
    const int m = P.size();

    vector<int> ans, rP = getRankArray(P);

    for( int i = 0; i <= n-m; ++i )
    {
        vector<int> Ti = vector<int>(T.begin()+i,T.begin()+i+m);
        vector<int> rTi = getRankArray(Ti);
        if( isAMatch(rTi,rP,delta,gamma) )
            ans.push_back(i);
    }
    return ans;
}
```

B. Appendix: C++ code of the update based algorithm ($updateBA$)

```
#include <bits/stdc++.h>
#include "Solver2.h"
#include "Utils.h"
using namespace std;

vector<int> Solver2::findAllDeltaGammaMatches(
        const vector<int>& T,
        const vector<int>& P,
        const int delta,
        const int gamma)
{
    const int n = T.size();
    const int m = P.size();

    vector<int> ans;
    deque<int> Tnr = getRankDeque( vector<int>(T.begin(),T.begin()+m) );
    vector<int> Pnr = getRankArray( P );
    int rankToDelete, valueToAdd, rankToAdd;

    for( int i = 0; i < n-m; ++i )
    {
        if( isAMatchDeque(Tnr,Pnr,delta,gamma))
            ans.push_back(i);

        rankToDelete = Tnr[0];
        valueToAdd = T[i+m];
        Tnr.pop_front();

        rankToAdd = 1;
        for( int j = 0; j+1 < m; ++ j ) {
            if( Tnr[j] > rankToDelete ) Tnr[j]--;
            if ( T[i+j+1] <= valueToAdd ) rankToAdd++;
```

```
            else Tnr[j]++;
        }

        Tnr.push_back( rankToAdd );
    }

    if( isAMatchDeque(Tnr,Pnr,delta,gamma))
        ans.push_back(n-m);

    return ans;
}
```

C. Appendix: C++ code of the segment tree based algorithm ($segtreeBA$). Classes and pointers based version

---

```
#include <bits/stdc++.h>
#include "Utils.h"
#include "Solver3.h"
using namespace std;

#define L(x)    ((x)<<1) //left child of x
#define R(x)    (L(x)+1) //right child of x

void Solver3::updateFromChildren ( int node ) {
    if ( T[minIndex[L(node)]] <= T[minIndex[R(node)]] )
        minIndex[node] = minIndex[L(node)];
    else minIndex[node] = minIndex[R(node)];
}

void Solver3::buildSegtree ( int node, int fr, int to) {
    if ( fr == to ) minIndex[node] = fr;
    else {
        const int mid = (fr+to)>>1;
        buildSegtree(L(node),fr,mid);
        buildSegtree(R(node),mid+1,to);
        updateFromChildren ( node );
    }
}

int Solver3::queryMinIndex ( int a, int b, int node, int fr, int to ) {
    if ( a == fr && to == b ) return minIndex[node];
    const int mid = (fr+to)>>1;
    if ( b <= mid ) return queryMinIndex ( a, b, L(node), fr, mid );
    if ( a > mid ) return queryMinIndex ( a, b, R(node), mid+1, to );
    int iL = queryMinIndex ( a, mid, L(node), fr, mid );
```

```
    int iR = queryMinIndex ( mid+1, b, R(node), mid+1, to );
    return T[iL] <= T[iR] ? iL : iR;
}

void Solver3::update ( int idx, int newVal, int node, int fr, int to ) {
    if ( fr == to ) { // fr == to == idx
        T[idx] = newVal;
        return;
    }
    const int mid = (fr+to)>>1;
    if ( idx <= mid ) update ( idx, newVal, L(node), fr, mid );
    else update ( idx, newVal, R(node), mid+1, to );
    updateFromChildren(node);
}

#undef L
#undef R

vector<int> Solver3::findAllDeltaGammaMatches(
        const vector<int>& T,
        const vector<int>& P,
        const int delta,
        const int gamma)
{
    this->T = T;
    const int n = T.size();
    const int m = P.size();
    vector<int> rP = getRankArray(P);
    // seg tree size is 2 * ( n rounded up to a power of 2)
    minIndex.resize ( 2<<(32-__builtin_clz(n-1)) );

    // save changes made to T, to undo them later
    vector<int> changedIndex(m), oldValue(m);
    int nChanges = 0;

    vector<int> ans;

    buildSegtree(1, 0,n-1);
    for ( int startIdx = 0; startIdx+m <= n; ++startIdx ) {
        //cout << "matching T[" << startIdx << ".." << (startIdx + m - 1) << "]
            with P" << endl;
        int curDelta = 0, curGamma = 0, rank;
```

```
        for ( rank = 1; rank <= m; ++rank ) {
            const int idxT = queryMinIndex ( startIdx, startIdx+m-1, 1, 0, n-1 );
            const int idxP = idxT-startIdx;
            //cout << "comparing " << rank << " vs " << rP[idxP] << endl;
            curDelta = abs ( rank - rP[idxP] );
            curGamma += curDelta;
            if ( curDelta > delta || curGamma > gamma )
                break;

            //make T[idxT] = INT_MAX, so that it won't be the minimum again
            changedIndex[nChanges] = idxT;
            oldValue[nChanges++] = T[idxT];
            update ( idxT, INT_MAX, 1, 0, n-1 );
        }

        //undo changes
        for ( int c = 0; c < nChanges; ++c )
            update ( changedIndex[c], oldValue[c], 1, 0, n-1 );
        nChanges = 0;

        //found match, add it to answer
        if ( rank > m )
            ans.push_back ( startIdx );
    }
    return ans;
}
```

D. Appendix: C++ code of the segment tree based algorithm ($segtreeBA$). Array based version

---

```
#include <bits/stdc++.h>
#include "Solver5.h"
#include "Utils.h"
using namespace std;

inline int bestIndex( const vector<int>& T, int a, int b ) {
    return T[a] <= T[b] ? a : b;
}

void buildSegTree ( vector<int>& stree, const vector<int>& A ) {
    stree.resize ( A.size()*2 );
    for ( int i = 0; i < A.size(); ++i ) stree[A.size()+i] = i;
    for ( int i = A.size()-1; i; --i )
        stree[i] = bestIndex(A, stree[i<<1], stree[i<<1|1]);
}

void updateSegTree( vector<int>& stree, vector<int>& A, int i, int x ) {
    for (A[i] = x, i += A.size(); i >>= 1; )
        stree[i] = bestIndex(A, stree[i<<1], stree[i<<1|1] );
}

// minimum index in [l,r)
int querySegTree (const vector<int>& stree, const vector<int>& A, int l, int r )
    {
    int ansL = l, ansR = r-1;
    for ( l += A.size(), r += A.size(); l < r; l >>= 1, r >>= 1 ) {
        if ( l&1 ) ansL = bestIndex(A, ansL, stree[l++]);
        if ( r&1 ) ansR = bestIndex(A, stree[--r], ansR);
    }
    return bestIndex(A, ansL, ansR );
```

```
}

vector<int> Solver5::findAllDeltaGammaMatches(
        const vector<int>& _T,
        const vector<int>& P,
        const int delta,
        const int gamma)
{
    vector<int> T = _T;
    const int n = T.size();
    const int m = P.size();
    vector<int> rP = getRankArray(P);

    // build seg tree with n leaves and initialize it with T
    vector<int> minIndex;
    buildSegTree ( minIndex, T );

    // save changes made to T, to undo them later
    vector<int> changedIndex(m), oldValue(m);
    int nChanges = 0;

    vector<int> ans;
        for ( int startIdx = 0; startIdx+m <= n; ++startIdx ) {
        //cout << "matching T[" << startIdx << ".." << (startIdx + m - 1) << "]
            with P" << endl;
        int curDelta = 0, curGamma = 0, rank;
        for ( rank = 1; rank <= m; ++rank ) {
            const int idxT = querySegTree ( minIndex, T, startIdx, startIdx+m );
            const int idxP = idxT-startIdx;
            //cout << "comparing " << rank << " vs " << rP[idxP] << endl;
            curDelta = abs ( rank - rP[idxP] );
            curGamma += curDelta;
            if ( curDelta > delta || curGamma > gamma )
                break;

            //make T[idxT] = INT_MAX, so that it won't be the minimum again
            changedIndex[nChanges] = idxT;
            oldValue[nChanges++] = T[idxT];
            updateSegTree ( minIndex, T, idxT, INT_MAX );
        }
        //undo changes
        for ( int c = 0; c < nChanges; ++c )
```

```
            updateSegTree ( minIndex, T, changedIndex[c], oldValue[c] );
        nChanges = 0;
        //found match, add it to answer
        if ( rank > m )
            ans.push_back ( startIdx );
    }
    return ans;
}
#undef best
```

E. Appendix: C++ code of the binary indexed tree (BIT or Fenwick tree) based algorithm ($bitBA$)

```
#include <bits/stdc++.h>
#include "Utils.h"
#include "Solver4.h"
using namespace std;

inline void addAt ( vector<int>& tree, int i, int x ) {
    for ( ; i < tree.size(); i += (i&(-i)) )
        tree[i] += x;
}

inline int sumUpTo ( const vector<int>& tree, int i ) {
    int r = 0;
    for ( ; i ; i -= (i&(-i)) ) r += tree[i];
    return r;
}

//true if T[i...i+m-1] matches P
bool isAMatch ( int i, const vector<int>& bit, const vector<int>& Tnr,
        const vector<int>& Pnr, int delta, int gamma )
{
    int curGamma = 0;
    for ( unsigned j = 0; j < Pnr.size(); ++j )
    {
        const int rank = sumUpTo(bit, Tnr[i+j]);
        const int deltaJ = abs ( Pnr[j] - rank );
        curGamma += deltaJ;
        if ( deltaJ > delta || curGamma > gamma )
            return false;
    }
```

```
    return true;
}

vector<int> Solver4::findAllDeltaGammaMatches(
        const vector<int>& T,
        const vector<int>& P,
        const int delta,
        const int gamma)
{
    const int n = T.size(), m = P.size();
    vector<int> ans;

    vector<int> Tnr = getRankArray(T);
    vector<int> Pnr = getRankArray(P);
    vector<int> bit ( n+1, 0 );

    for ( int i = 0; i+1 < m && i < n; ++i )
        addAt(bit, Tnr[i],1);

    for ( int i = 0; i+m <= n; ++i ) {
        addAt(bit, Tnr[i+m-1],1);
        if ( isAMatch(i, bit, Tnr, Pnr, delta, gamma) )
            ans.push_back(i);
        addAt(bit, Tnr[i],-1);
    }

    return ans;
}
```

F. Appendix: C++ code of the experimental setup

F.1. Experimental setup to compare our algorithms solving $\delta\gamma$–OPMP

Here we show an example of the experimental setup used to compare our four algorithms. In particular, we show the experimental setup to compare the algorithm when the parameter n grows (see c++ function $run_experiment_ranging_n$ below), with the other four parameters fixed in a given value (δ, γ, the size of the pattern m and the size of the alphabet Σ_σ). Here we omit the implementation of some trivial utilitarian functions.

```
int global_experiment_counter = 0;
int files_generated_counter = 0;
int global_matches_counter_naive = 0;
int global_matches_counter_nm_solution = 0;
int global_matches_counter_nm_solution_DEBUG = 0;


const string SEPARATOR = "\t";
const int N_SOLVERS = 5;

int get_random_number( const int& min, const int& max );
template<class T> void write_to_file( const vector<T>& lines , const string&
    file_name );
vector<string> read_from_file( const string& file_name );
vector<int> to_int_vector( const vector<string>& v );
vector<string> to_string_vector( const vector<int>& v );
vector<int> get_ordered_permutation( const int& n);
vector<int> get_random_permutation( const int& n );
vector<int> get_random_array( const int& n, const int& min_value, const int&
    max_value );
set<int> get_random_set( const int& n, const int& min_value, const int&
    max_value );
```

```
vector<abstract_dgopm_solver*> get_solvers() {
        return vector<abstract_dgopm_solver*> {
                new dgopm_solver1(),
                new dgopm_solver2(),
                new dgopm_solver3(),
                new dgopm_solver4(),
                new dgopm_solver5()
        };
}
/**
  * Runs a single experiment with the given T, P, delta and gamma agains all the
  * solvers. And adds the time in seconds at the end of each times vector.
  * The return value is the number of matches found by the solvers. They all
     should
  * return the same, if they don't the program will be terminated and an error
     message
  * will be shown.
  */
int run_single_experiment (
                const vector<int>& T,
                const vector<int>& P,
                const int delta,
                const int gamma,
                vector<vector<double> >& times) {
        vector<abstract_dgopm_solver*> solvers = get_solvers();
        vector<int> previus_matches_found;
        for ( int solver_idx = 0; solver_idx < N_SOLVERS; ++solver_idx ) {
                const clock_t begin_time = clock();
                vector<int> matches_found =
                                solvers[solver_idx]->find_all_delta_gamma_matches
                                    (T,P,delta,gamma);
                times[solver_idx].push_back (( clock() - begin_time ) /
                    (double)CLOCKS_PER_SEC);
                if ( solver_idx && previus_matches_found != matches_found ) {
                        cout <<
                            "-------------------------------------------------" <<
                            endl;
                        cout << " ERROR ! MISMATCH, solutions " << solver_idx-1 <<
                            " ans ";
                        cout << solver_idx << " return different results" << endl;
                        cout << endl;
```

```
                    cout << endl;
                    cout <<
                        "-------------------------------------------------" <<
                        endl;
                    exit(0);
                }
                previus_matches_found = matches_found;
        }

        return previus_matches_found.size();
}

//! Run experiment with given values of m (size of the pattern P), delta, and
    gamma.
/*!
  \param min_n
  \param max_n
  \param step_n
  \param delta The given delta
  \param gamma The given gamma
  Run experiment with given values of m (size of the pattern P), delta, and
      gamma. The values
  of n starts at min_n and end in max_n. The value of n increases in steps of
      step_n. Each call to this
  function produces a file with three columns: value of n, average time to find
      all matches, and total amount of matches found.

*/
void run_experiment_ranging_n( int min_n, int max_n , int step_n , int m , int
    delta , int gamma ,
                            int min_real_value_T, int max_real_value_T, int
                                min_real_value_P, int max_real_value_P,
                            int experiment_count_per_instance )
{
    const bool LOCAL_DEB = false;
    ++files_generated_counter;
    string file_name = norm(tostring(files_generated_counter),3,'0')+" m
        "+tostring(m)+" delta "+tostring(delta)+" gamma "+tostring(gamma)+" n
        from "+tostring(min_n)+" to "+tostring(max_n)+" step "+tostring(step_n);
    vector<string> lines_to_file;
        lines_to_file.push_back ( get_columns_header("n") );
    int total_matches_per_instance = 0;
```

```
for( int n = min_n; n <= max_n; n += step_n )
{
    vector<vector<double> > times ( N_SOLVERS );

    int matches_accumulator = 0;
    cout << "For n = " << n << endl;
    for( int experiment = 0; experiment < experiment_count_per_instance; ++
        experiment )
    {
        cout << "Experiment " << experiment << endl;
        vector<int> T = get_random_array(n,min_real_value_T,max_real_value_T);
        vector<int> P = get_random_array(m,min_real_value_P,max_real_value_P);

        matches_accumulator += run_single_experiment(T,P,delta,gamma, times);
    }
    total_matches_per_instance += matches_accumulator;

    vector<double> times_median;
    for ( int i = 0; i < N_SOLVERS; ++i ) {
            sort ( times[i].begin(), times[i].end() );
            times_median.push_back ( times[i][times[i].size()/2] );
    }

    double average_matches = matches_accumulator /
        (double)experiment_count_per_instance;

    stringstream ssline;
    ssline << tostring(n);
    for ( int i = 0; i < N_SOLVERS; ++i )
            ssline << SEPARATOR + tostring(times_median[i]);
    ssline << SEPARATOR+tostring(average_matches);

    lines_to_file.push_back( ssline.str() );
    cout << lines_to_file.back() << endl;
}
if( LOCAL_DEB )
{
    cout << "In file \"" << file_name << "\" following " <<
        lines_to_file.size() << " lines to write: " << endl;
    for( int i = 0; i < lines_to_file.size(); ++ i )
        cout << lines_to_file[i] << endl;
```

```
    }
    file_name = file_name+" {"+tostring(total_matches_per_instance)+"total
        matches}.csv";
    cout << "\""<< file_name <<"\"" << endl;
    write_to_file(lines_to_file,file_name);
}
```

G. Appendix: C++ code of the utilitarian functions

```
#include <bits/stdc++.h>
using namespace std;
bool isAMatch(const vector<int>& ra, const vector<int>& rb , int maxDelta, int
    maxGamma )
{
    if( ra.size() != rb.size() )
    {
        cerr << "Error, illegal arguments in isAMatch()" << endl;
        return false;
    }
    int delta = 0, gamma = 0;
    for( int i = 0; i < ra.size(); ++ i )
        if( ra[i] != rb[i] )
        {
            delta = fabs(ra[i]-rb[i]);
            gamma += delta;
            if( delta > maxDelta || gamma > maxGamma )
                return false;
        }
    return true;
}
bool isAMatchDeque(const deque<int>& ra, const vector<int>& rb , int maxDelta,
    int maxGamma )
{
    if( ra.size() != rb.size() )
    {
        cerr << "Error, illegal arguments in isAMatchDeque()" << endl;
        return false;
    }
    int delta = 0, gamma = 0;
    for( int i = 0; i < ra.size(); ++ i )
        if( ra[i] != rb[i] )
```

```
        {
            delta = fabs(ra[i]-rb[i]);
            gamma += delta;
            if( delta > maxDelta || gamma > maxGamma )
                return false;
        }
    return true;
}
vector<int> getRankArray( const vector<int>& v )
{
    int n = v.size();
    vector<pair<int,int> > vPairs ( n );
    for( int i = 0; i < n ; ++ i )
        vPairs[i] = make_pair(v[i],i);

    sort( vPairs.begin(), vPairs.end());
    vector<int> rank(n,-1);

    for( int i = 0; i < n; ++ i )
        rank[ vPairs[i].second ] = i+1;
    return rank;
}
deque<int> getRankDeque( const vector<int>& v )
{
    int n = v.size();
    vector<pair<int,int> > vPairs(n);
    for( int i = 0; i < n ; ++ i )
        vPairs[i] = make_pair(v[i],i);

    sort( vPairs.begin(), vPairs.end());
    deque<int> rank(n,-1);

    for( int i = 0; i < n; ++ i )
        rank[ vPairs[i].second ] = i+1;
    return rank;
}
```

Bibliography

[1] *More Geometric Data Structures*, pages 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[2] Michael H. Albert, Robert E. L. Aldred, Mike D. Atkinson, and Derek A. Holton. *Algorithms for Pattern Involvement in Permutations*, pages 355–367. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[3] Amihood Amir, Yonatan Aumann, Piotr Indyk, Avivit Levy, and Ely Porat. Efficient computations of l_1 and l $_{\text{infinity}}$ rearrangement distances. *Theor. Comput. Sci.*, 410(43):4382–4390, 2009.

[4] Amihood Amir, Ohad Lipsky, Ely Porat, and Julia Umanski. Approximate matching in the l_1 metric. In *Combinatorial Pattern Matching, 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005, Proceedings*, pages 91–103, 2005.

[5] Alberto Apostolico and Zvi Galil, editors. *Pattern Matching Algorithms*. Oxford University Press, Oxford, UK, 1997.

[6] Yoan José Pinzón Ardila, Manolis Christodoulakis, Costas S. Iliopoulos, and Manal Mohamed. Efficient $(\delta\gamma)$-pattern-matching with don't cares *. 2007.

[7] MIDI Association. Midi official webpage. `https://www.midi.org/`. Accessed: 2016-04-15.

[8] Djamal Belazzougui, Adeline Pierrot, Mathieu Raffinot, and Stéphane Vialette. *Single and Multiple Consecutive Permutation Motif Search*, pages 66–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[9] Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. Pattern matching for permutations. *Inf. Process. Lett.*, 65(5):277–283, March 1998.

[10] P. Brass. *Advanced Data Structures*. Cambridge books online. Cambridge University Press, 2008.

[11] Marie-Louise Bruner and Martin Lackner. The computational landscape of permutation patterns. *CoRR*, abs/1301.0340, 2013.

[12] Emilios Cambouropoulos, Maxime Crochemore, Costas Iliopoulos, Laurent Mouchard, and Yoan Pinzon. Algorithms for computing approximate repetitions in musical sequences. *International Journal of Computer Mathematics*, 79(11):1135–1148, 2002.

[13] D. Cantone, S. Cristofaro, and S. Faro. Efficient Algorithms for the δ-Approximate String Matching Problem in Musical Sequences. In *Proc. of the Prague Stringology Conference*, 2004.

[14] Domenico Cantone, Salvatore Cristofaro, and Simone Faro. An efficient algorithm for alpha-approximate matching with *delta*-bounded gaps in musical sequences. In *Experimental and Efficient Algorithms, 4th InternationalWorkshop, WEA 2005, Santorini Island, Greece, May 10-13, 2005, Proceedings*, pages 428–439, 2005.

[15] Domenico Cantone, Salvatore Cristofaro, and Simone Faro. On tuning the ($\backslash\delta$, $\backslash\alpha$)-sequential-sampling algorithm for $\backslash\delta$-approximate matching with alpha-bounded gaps in musical sequences. In *ISMIR 2005, 6th International Conference on Music Information Retrieval, London, UK, 11-15 September 2005, Proceedings*, pages 454–459, 2005.

[16] Christian Charras and Thierry Lecroq. *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004.

[17] Tamanna Chhabra, M. Oğuzhan Kulekci, and Jorma Tarhio. Alternative algorithms for order-preserving matching. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2015*, pages 36–46, Czech Technical University in Prague, Czech Republic, 2015.

[18] Tamanna Chhabra and Jorma Tarhio. Order-preserving matching with filtration. In *Proceedings of the 13th International Symposium on Experimental Algorithms - Volume 8504*, pages 307–314, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[19] Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397 – 402, 2015.

[20] Peter Clifford, Raphaël Clifford, and Costas S. Iliopoulos. Faster algorithms for δ, γ-matching and related problems. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05)*, volume 3537 of *Lecture Notes in Computer Science*, pages 68–78. Springer, 2005.

[21] R. Clifford and C. Iliopoulos. Approximate string matching for music analysis. *Soft Computing*, 8(9):597–603, 2004.

[22] Richard Cole, Costas Iliopoulos, Thierry Lecroq, Wojciech Plandowski, and Wojciech Rytter. On special families of morphisms related to δ-matching and don't care symbols. *Information Processing Letters*, 85(5):227 – 233, 2003.

[23] Intel Corporation. Working draft, standard for programming language c++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3485.pdf`. Accessed: 2016-04-15.

[24] Tim Crawford, Costas S. Iliopoulos, and Rajeev Raman. String-Matching Techniques for Musical Similarity and Melodic Recognition. *Computing in Musicology*, 11:71–100, 1998.

[25] M. Crochemore, C. Iliopoulos, G. Navarro, Y. Pinzon, and A. Salinger. Bit-parallel (δ, γ)-matching suffix automata. *Journal of Discrete Algorithms (JDA)*, 3(2–4):198–214, 2005.

[26] M. Crochemore, C.S. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. Three Heuristics for δ-Matching δ-BM Algorithms. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, pages 178–189. Springer-Verlag London, UK, 2002.

[27] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. *Order-Preserving Incomplete Suffix Trees and Order-Preserving Indexes*, pages 84–95. Springer International Publishing, Cham, 2013.

[28] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving suffix trees and their algorithmic applications. *CoRR*, abs/1303.6872, 2013.

[29] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving indexing. *Theor. Comput. Sci.*, 638(C):122–135, July 2016.

[30] Maxime Crochemore, Costas S. Iliopoulos, Thierry Lecroq, Yoan Pinzon, Wojciech Plandowski, Wojciech Rytter, and Uniwersytet Warszawski. Occurrence and substring heuristics for δ-matching, 2003.

[31] Klim Efremenko and Ely Porat. Approximating general metric distances between a pattern and a text. *CoRR*, abs/0802.1427, 2008.

[32] Simone Faro and M. Oguzhan Külekci. Efficient algorithms for the order preserving pattern matching problem. *CoRR*, abs/1501.04001, 2015.

[33] Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24:327–336, 1994.

[34] Music Notes Film. Film music notes webpage. `http://www.filmmusicnotes.com/wp-content/uploads/2013/03/03-Main-Melody-ideas.jpg`. Accessed: 2016-04-15.

[35] K. Fredriksson and Sz. Grabowski. Efficient algorithms for (δ, γ, α) and (δ, $k_{\triangle}$, α)–matching. *International Journal of Foundations of Computer Science (IJFCS)*, 19(1):163–183, 2008.

[36] Paweł Gawrychowski and Przemysław Uznański. Order-preserving pattern matching with k mismatches. *Theoretical Computer Science*, 638:136 – 144, 2016. Pattern Matching, Text Data Structures and CompressionIssue in honor of the 60th birthday of Amihood Amir.

[37] Sylvain Guillemot and Dániel Marx. Finding small patterns in permutations in linear time. *CoRR*, abs/1307.3073, 2013.

[38] Sylvain Guillemot and Stéphane Vialette. *Pattern Matching for 321-Avoiding Permutations*, pages 1064–1073. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[39] Md. Mahbubul Hasan, A. S. M. Sohidull Islam, Mohammad Saifur Rahman, and M. Sohel Rahman. *Order Preserving Prefix Tables*, pages 111–116. Springer International Publishing, Cham, 2014.

[40] Md. Mahbubul Hasan, A.S.M.Shohidull Islam, Mohammad Saifur Rahman, and M.Sohel Rahman. Order preserving pattern matching revisited. *Pattern Recognition Letters*, 55(C):15–21, April 2015.

[41] Louis Ibarra. Finding pattern matchings for permutations. *Information Processing Letters*, 61(6):293 – 295, 1997.

[42] Inc IMDb. John Williams imdb profile. `http://www.imdb.com/name/nm0002354/`. Accessed: 2016-04-15.

[43] Inc IMDb. Star Wars: Episode V - The Empire Strikes Back (original title) imdb page. `http://www.imdb.com/title/tt0080684/`. Accessed: 2016-04-15.

[44] Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68 – 79, 2014. Advances in Stringology.

[45] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

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

[47] Inbok Lee, Raphaël Clifford, and Sung-Ryul Kim. Algorithms on extended (*delta*, *gamma*)-matching. In *Computational Science and Its Applications - ICCSA 2006, International Conference, Glasgow, UK, May 8-11, 2006, Proceedings, Part III*, pages 1137–1142, 2006.

[48] Inbok Lee, Juan Mendivelso, and Yoan J. Pinzón. *$\delta\gamma$ – Parameterized Matching*, pages 236–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[49] Ohad Lipsky. Efficient distance computations. Master's thesis, Bar-llan University, Israel, 2003.

[50] Ohad Lipsky and Ely Porat. Approximate matching in the l_∞ metric. *Inf. Process. Lett.*, 105(4):138–140, February 2008.

[51] Ohad Lipsky and Ely Porat. L1 pattern matching lower bound. *Inf. Process. Lett.*, 105(4):141–143, February 2008.

[52] Ohad Lipsky and Ely Porat. Approximate pattern matching with the L_1, L_2 and L_∞ metrics. *Algorithmica*, 60(2):335–348, 2011.

[53] László Lovász. *Combinatorial Problems and Exercises*. AMS Chelsea Pub., Providence, RI, 2007.

[54] Juan Mendivelso. Definition and solution of a new string searching variant termed $\delta\gamma$-parameterized matching. Master's thesis, National University of Colombia, Bogota, Colombia, 2010.

[55] Juan Mendivelso, Inbok Lee, and Yoan J. Pinzón. *Approximate Function Matching under δ- and γ- Distances*, pages 348–359. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[56] Juan Mendivelso, Camilo Pino, Luis F. Niño, and Yoan Pinzón. *Approximate Abelian Periods to Find Motifs in Biological Sequences*, pages 121–130. Springer International Publishing, Cham, 2015.

[57] Juan Mendivelso and Yoan Pinzón. 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.

[58] D. Rotem. Stack sortable permutations. *Discrete Math.*, 33(2):185–196, January 1981.

[59] Thomas Scarff. MIDI note numbers for different octaves. http://www.electronics.dit.ie/staff/tscarff/Music_technology/midi/midi_note_numbers_for_octaves.htm. Accessed: 2016-04-15.