Algorithms for Sequence Similarity Measures

by

Mustafa Mohamad

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
November 2010

Copyright © Mustafa Mohamad, 2010

Abstract

Given two sets of points A and B ($|A| = m$, $|B| = n$), we seek to find a minimum-weight many-to-many matching which seeks to match each point in A to at least one point in B and vice versa. Each matched pair (an edge) has a weight. The goal is to find the matching that minimizes the total weight. We study two kinds of problems depending on the edge weight used. The first edge weight is the Euclidean distance, d_1. The second is edge weight is the square of the Euclidean distance, d_2. There already exists an $O(k \log k)$ algorithm for d_1, where $k = m + n$. We provide an $O(mn)$ algorithm for the d_2 problem. We also solve the problem of finding the minimum-weight matching when the sets A and B are allowed to be translated on the real line. We present an $O(mnk \log k)$ algorithm for the d_1 problem and an $O(3^{mn})$ algorithm for the d_2. Furthermore, we also deal with the special case where A and B lie on a circle of a specific circumference. We present an $O(k^2 \log k)$ algorithm and $O(kmn)$ algorithm for solving the minimum-weight matching for the d_1, and d_2 weights respectively. Much like the problem on the real line, we extend this problem to allow the sets A and B to be rotated on the circle. We try to find the minimum-weight many-to-many matching when rotations are allowed. For d_1 we present an $O(k^2 mn \log k)$ algorithm and a $O(3^{mn})$ algorithm for d_2.

Acknowledgments

All praise and thanks are due to my lord and creator, Allah the most high and glorified, for his endless blessings, without which, I would not be able to produce this work.

A special thanks goes out to my supervisor, Dr. David Rappaport, who has been very helpful in guiding me through the process of working on this thesis. He provided helpful insights throughout, and was always available to discuss ideas and provide direction. In addition, he was very supportive in funding my expenses for attending conferences of interest.

I would also like to thank my dear wife, Dina Al-Yaseen, who was behind me a hundred percent when I decided to pursue my Master's at Queen's. She was always there during times of doubt, and was always happy with my achievements.

My utmost gratitude goes out to my wonderful parents who were behind me all the way with their emotional and financial support. Their hard work is what got me this far, and I continue to be indebted to them for all their favours.

Finally, I would like to to thank all of the Queen's staff and students who have made my experience at Queen's wonderful. What I love about Queen's is the humbleness and helpfulness of the entire staff, and the high spirit of the School of Computing. I am already looking forward to spending more time at Queen's School of Computing

pursuing my Ph.D.

Table of Contents

Abstract i

Acknowledgments ii

Table of Contents iv

List of Tables v

List of Figures vi

1 Introduction 1
- 1.1 Computational Biology 2
- 1.2 Computational Music Theory 3
- 1.3 Computer Vision 4
- 1.4 Natural Language Processing 5

Chapter 2:
Background 9
- 2.1 Graph Terminology 9
- 2.2 Minimum-weight matchings for point sets on a line or a circle 11

Chapter 3:
Algorithms for Computing Sequence Distance 18
- 3.1 The Static Distance 21
- 3.2 The Dynamic Distance 23
- 3.3 The Static Distance for Circular Sequences 34
- 3.4 The Dynamic Distance for Circular Sequences 38

Chapter 4:
Conclusion 41

Bibliography 43

List of Tables

3.1 Experimental Results of running Algorithm 3.3 30
3.2 Experimental Results of running Algorithm 3.3 on Compressed B datasets where $m = n = 10$ 31
3.3 Experimental Results of running Algorithm on Compressed B datasets of different cardinalities 3.3 33
3.4 Experimental Results of running Algorithm 3.3 different clustered configuration where $|A| = 15$ and $|B| = 11$ 34

List of Figures

1.1 Chronotonic Representation of the Rhythm 33242 4

2.1 A simple graph 10
2.2 A bipartite graph 10
2.3 Partitioning of the set $A \cup B$ 17

3.1 Dynamic Programming Algorithm chooses one of the three subproblems 21
3.2 The identity matching on the left is optimal for d_1 with $d_1(A, B) = 6$. However it is not optimal for d_2 where $d_2(A, B) = 12$. The matching on the right is optimal for d_2 with $d_2(A, B) = 10$. As can be seen the edge (a_2, b_2) is only optimal for d_2 if $\epsilon > \sqrt{2}$ or $\epsilon < -\sqrt{2}$ 24
3.3 Experimental Complexity in the domain of $m \times n$ and $m \times n \log(m \times n)$ 32
3.4 Arcs passing a point in opposite directions 35
3.5 Possible scenarios for arcs covering the circle in the many-to-many matching 36

Chapter 1

Introduction

Measuring the similarity between two sequences is a problem that arises in many fields including: computational biology [2], computational music theory [20],[21], computer vision [8], and natural language processing [4]. The term *distance* between two sequences is often used as a general term to capture the degree of similarity between two sequences: greater similarity corresponds to a smaller distance between two sequences. There is a variety of ways to measure the distance between two sequences depending on the specific field of study. We briefly outline some of these distance measures as they arise in different fields.

Before describing distance in different field. We would like to note that the term distance is used here as a similarity measure. It is not being used in the mathematical sense. A distance, d, in the mathematical sense is a metric on a set X, which satisfies the following properties:

1. $d(x, y) \geq 0$ (nonnegativity)
2. $d(x, y) = 0$ if and only if $x = y$ (identity of indiscernibles)
3. $d(x, y) = d(y, x)$ (symmetry)

4. $d(x, z) \leq d(x, y) + d(y, z)$ (triangle inequality)

We make no claims as to wheter the distances (similarity measures) discussed in this work are metrics.

1.1 Computational Biology

One of the goals of biologists is to understand the relationships between the genomes of different organisms. This is done through measuring the similarity between the DNA sequences that make up different genomes. A DNA sequence can be represented symbolically by a string over the alphabet [A,T,G,C], where the four letters represent the four nucleic acids, which are the building blocks of DNA sequences. The following are distance measures used to compare DNA sequences:

The Hamming distance: It refers to the number of positions in which two DNA sequences differ. This measure can be applied to any string comparison and is not restricted to DNA sequences.

The edit distance: It is related to the Hamming distance and it is the minimum cost of the *edit operations* that are required to change one sequence into the other. The potential edit operations that are allowed include: insertion, deletion, replacement and transposition. Each one of these operations is assigned a cost. Under the *unit cost model*, where the only operations that are allowed are insertion, and deletion and the cost of both operations is one [9].

1.2 Computational Music Theory

A fundamental problem in the field of computational music theory is computing the similarity between two rhythms. This has application in understanding ancestral relationships between different rhythms as well as real world scenarios such as copy right infringement detection. One way to represent a rhythm is through an *interval length vector* which is a vector of numbers where each number represents the duration between onsets (the beginning time of a note) in the rhythm. Another way to represent a rhythm is through a binary string. As an example, the vector [33242] and the binary string 10010010100010 represent the same rhythm. The following distance measures are used to compare rhythms of the same length:

The Euclidean interval vector distance: The Euclidean distance between the two interval length vectors.

The swap distance: In the binary representation of rhythm, a swap is defined as the interchange between a one and a zero that are adjacent to each other. The swap distance is the minimum number of swaps required to convert one rhythm into the other.

The chronotonic distance: An alternative method to represent a rhythm, proposed by Kjell Gustafson [10], is through a two dimensional graph, called the *chronotonic representation.* In this representation, the y-axis and the x-axis both represent the duration between the onsets of the rhythm. This makes it easier to see the relative duration of the intervals (see Figure 1.1).

In this representation, each rhythm can now be expressed as a piece-wise continuous function. Given two piece-wise continuous functions $f_1(x)$ and $f_2(x)$, the *Kolmogorov variational distance* [19] is given by:

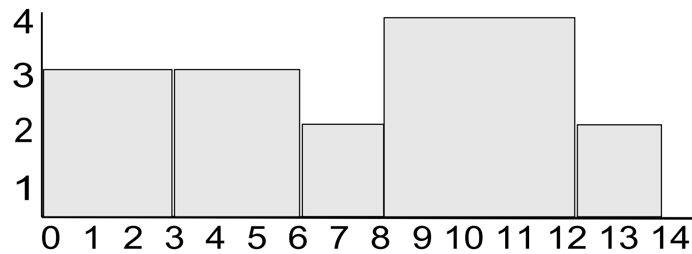


Figure 1.1: Chronotonic Representation of the Rhythm 33242

$$K = \int |f_1(x) - f_2(x)| \, dx$$

The term chronotonic distance refers to using the Kolmogorov variational distance on the chronotonic representation of rhythms.

1.3 Computer Vision

One of the fundamental problems studies in the field of computer vision is formulating algorithms that can distinguish between two different images. Naturally, this means there must be a way to measure how similar two images are to each other. Below we outline some of these similarity measures:

Earth Mover's Distance (EMD) : It is a measure of similarity between two probability distributions over a region, D. More intuitively, it can be thought of as the minimum amount of work that has to be done to move a set of scattered balls of possibly different weight into a set of holes that are scattered over the same region. Demirci et al. [8] compare two images by first representing each image as a directed graph that captures the relationships between the main features of the image. Next,

they embed the two graph representations into a normed vector space. This embedding creates a probability distribution over the normed vector space for each image. Next, the probability distributions of the images are compared using EMD.

Kolmogorov complexity based distance measures: Theoretically, the Kolmogorov Complexity, $K(x)$, is defined for a string, x, as the length of the shortest program required to produce x on a universal computer. $K(x)$ can be thought of as a compressed version of x because it can be used to produce x. Using this idea, data compression can be used to measure the distance between two different objects of interest. The objects of interest could be DNA sequences, strings of any sort, images or even videos. This idea is used by Campana and Keogh [5] to measure the similarity between two textures. Given two images to compare, they convert each image into a frame and create a two frame video file. This video file is then compressed using the MPEG-1 encoding. If the ratio of compression is high it indicates high resemblance between the two images.

1.4 Natural Language Processing

The field of Natural Language Processing studies the ability of computers to understand, modify, and generate natural human speech. A basic problem in the field, is enabling the computer to recognize words produced by humans. A word can be represented as a sequence of feature vectors. The main challenge is that the same word can be spoken differently by different individuals, and therefore have different sequence representations. One solution is to use the following distance measure proposed by Sakoe and Chiba [17]:

Warping Distance: Given any two sequences of feature vectors, $Q = q_1, q_2, ..., q_m$

and $G = g_1, g_2, ..., g_n$. A $m \times n$ matrix is formulated where each entry (i, j) in the matrix holds the quantity, $d(q_i, g_j) = (q_i - g_j)^2$. A *warping path*, W, in the matrix is defined as a contiguous path such that each element in Q is paired with at least one element in G and vice versa. W is defined as:

$$W = w_1, w_2, ..., w_k, ..., w_K \text{ where, } w_k = (i, j)_k \qquad max(m, n) \leq K < m + n + 1$$

W is contiguous if it satisfies the following three conditions:

- **Boundary Conditions:** $w_1 = (1, 1)$ and $w_K = (m, n)$ must be included in the path.
- **Continuity:** The cells in the path must all be adjacent horizontally, vertically or diagonally. In other words, there are no breaks in the path.
- **Monotonicity:** For $w_{k-1} = (s', t'), w_k = (s, t)$ then $s - s' \geq 0$ and $t - t' \geq 0$. In other words, the slope of the path is always non-negative.

The simplest form of the warping distance, $DW(Q, G)$, defined in [12] is then simply the cost of the path with smallest total distance.

$$DW(Q, G) = min\left\{ \sqrt{\sum_{k=1}^{K} w_k} / K \right.$$

Division by K is done to compensate for the fact that different warping paths may have different lengths. The path can be computed efficiently using Dynamic Programming. There is also a weighted form of this definition to allow some features to be weighted more heavily compared to others.

Another problem in the field of Natural Language processing is being able to measure how close two language models are to each other. One way to model a language is through an *n-gram model*. An n-gram model is probability distribution for predicting the next words in a sentence given the previous n-1 words. The details of how to construct n-gram models are not going to be discussed here but the book by Manning, and Schütze [14] can be referred to. The work of Singh [18] presents different similarity measures of n-gram probability distributions. Given two probability distributions of two different languages $p(x)$ and $q(x)$ their similarity, D, can be measured in numerous ways including:

Log Probability Measure:

$$D = \sum_x (\log p(x) - \log q(x))$$

Cross entropy

$$D = \sum_x (p(x) \times \log q(x))$$

Mutual cross entropy

$$D = \sum_x (p(x) \times \log q(x) + q(x) \times \log p(x))$$

These kinds of similarity measures fall into the class of distances that measure the similarity between distributions and are used in other fields as well.

As previously seen, the concept of similarity arises in numerous applications. In this thesis, we seek to measure similarity between two sequences of points in one dimensional space. The distance we used is the total weight of the *minimum-weight many-to-many matching* between the two sequences. In the next section, we present

some basic concepts from graph theory that aid in describing the problem as well as present previous research into problems that are very similar to our own.

Chapter 2

Background

In this chapter, we start by defining and explaining some basic concepts from Graph Theory and then briefly reviewing related literature.

2.1 Graph Terminology

A *graph* is a structure described by $G = (V, E)$ where V is a finite set of *nodes* or *vertices*, $\{v_1, v_2, ...\}$, and E is a set of elements that are pairs of V called *edges*, $\{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}\}$. For example the graph:

$$G = (\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_3\}, \{v_1, v_2\}, \{v_2, v_4\}, \{v_4, v_1\}\})$$

is shown in Figure 2.1.

A *walk* in a graph is a sequence of nodes $w = \{v_1, v_3, ..., v_k\}$, $k \geq 1$, such that $\{v_j, v_{j+1}\} \in E$. A walk without any repeated nodes except for the first node is called a *cycle*. In Figure 2.1 $c = \{v_1, v_2, v_4, v_1\}$ is a cycle of length 3. The length of the cycle

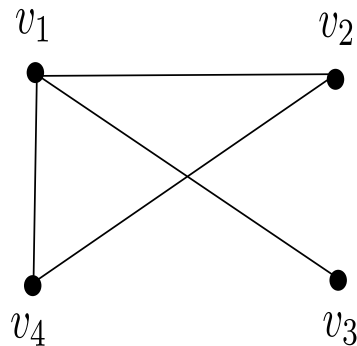


Figure 2.1: A simple graph

is defined as $|c| - 1$. A *bipartite graph* is a graph that does not contain any cycles of odd length. This allows for the nodes of the graph to be partitioned into two disjoint sets. Therefore, a bipartite graph is usually denoted by $B = (V, U, E)$ where V and U are the two disjoint sets of vertices, and each edge in E must have one vertex in V and the other in U. Figure 2.2 is an illustration of a bipartite graph.

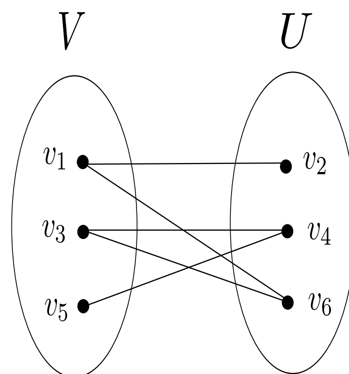


Figure 2.2: A bipartite graph

A *complete* bipartite graph, is one in which every $v \in V$ has an edge to all $u \in U$, and vice versa. The edges of a bipartite graph can also have weights assigned to them.

In this case we have a *weighted* bipartite graph.

A *matching* M, also called a *one-to-one matching*, of a bipartite graph $G = (V, U, E)$ is a subset of the edges, E, such that no two edges share a node. The *one-to-one matching problem* is concerned with finding a matching with the largest possible number of edges. A *many-to-one matching*, is a subset of the edges E, such that each $v \in V$ is paired with exactly one $u \in U$ and each $u \in U$ is paired with at least one $v \in V$. A *many-to-many* matching is a subset of E such that each $v \in V$ is paired to at least one $u \in U$ and each $u \in U$ is paired with at least one $v \in V$.

Given a complete weighted bipartite graph we define the problem of finding the *minimum-weight one-to-one matching* as finding a one-to-one matching such that the sum of weights of all edges in the matching is minimized. Similarly we define *minimum-weight many-to-one matching* and a *minimum-weight many-to-many matching*. These three problems will be referred to as *minimum-weight matchings*.

2.2 Minimum-weight matchings for point sets on a line or a circle

The problem of finding a minimum-weight one-to-one matching for arbitrarily weighted complete bipartite graphs (also called the *assignment problem*) can be solved using the Hungarian Algorithm due to Khun[13], in $O(|V|^3)$ operations where $|G| = 2.|V|$ [15].

The problems that will be explored are specific to point sets lying on the real line or on a circle. For point sets lying on a line the weight of each edge $[v_i, u_j]$ is simply the Euclidean distance between the two points, $w_{ij} = |v_i - u_j|$. The second

type is where the two point sets lie on a circle of circumference, C. The weight of each edge in this case, is the length of the shortest arc between the two points, $w_{ij} = min(u_j - v_i, v_i - u_j + C)$. In both cases, the point sets are in one dimensional space. The geometric structure of these two types of problems has led to algorithms that are more efficient than the Hungarian Algorithm as shall be elaborated upon.

In the remainder of the thesis, we will use the sets $A = \{a_1, a_2, ..., a_m\}$ and $B = \{b_1, b_2, ..., b_n\}$ to refer to the two sets of points which have to be matched. These two sets are represented as the two sets of nodes V and U in the complete bipartite graph.

Given two sets, A and B, on a real line such that $|A| + |B| = n$, A and B lie on the open interval $(0, X)$. The weight of an each edge (a_i, b_j) is simply the Euclidean distance, $|a_i - b_j|$. Karp and Li [11] define a height function, $H(x)$, as the difference between the number of points in A and B in the interval $(0, x]$.

$$H(x) = |A \cap (0, x]| - |B \cap (0, x]|, \qquad x \in (0, X)$$

They show that the cost of the minimum one-to-one matching equals:

$$\int_{x=0}^{X} |H(x)| \, dx$$

Furthermore, they show that the one-to-one matching that achieves this minimum weight is simply the *identity matching*, which matches a_i to b_i, for $i = 1 \cdots |A|$, when $|A| = |B|$.

If $|A| > |B|$, $|A| - |B| = e$, an optimal one-to-one matching between A and B seeks to choose the best elements of A to include in the one-to-one matching so that the sum of weights is minimized. For an arbitrary subset, E, such that $|E| = e$, the

authors define:

$$H^E(x) = |(A - E) \cap (0, x]| - |B \cap (0, x)|, \qquad x \in (0, X) \tag{2.1}$$

Therefore they seek to find E such that $\int_{x=0}^{X} |H^E(x)|\, dx$ is minimized. They calculate the profit of including an element $a \in A$ in the matching by the following function:

$$P(a) = \int_{x=a}^{X} (|H(x) - H(a) + 1| - |H(x) - H(a)|)\, dx$$

Therefore they reformulate $\int |H^E(x)|\, dx$ to be:

$$\int_{x=0}^{X} |H^E(x)|\, dx = \int_{x=0}^{X} |H(x)|\, dx - \sum_{a \in E} P(a)$$

Clearly, an optimal E must maximize $\sum_{a \in E} P(a)$. They also show that an optimal E must satisfy the property that $H(a_k) = k$ where a_k is the kth smallest element of E. The Algorithm for finding an E that satisfies the previous two properties is:

1. Sort A and B
2. Calculate $H(x)$ for each $x \in A \cup B$
3. Calculate $P(a)$ for each $a \in A$
4. For $k = 1, 2, ..., e$
   (a) Find the leftmost point a_k of height k that maximizes $P(a_k)$
   (b) Add a_k to E
5. return E

It is not hard to see that all the steps in the Algorithm except for the sorting step have a complexity of $O(n)$. We know the sorting step has a complexity of $O(n \log n)$.

Therefore the entire complexity of the Algorithm is $O(n \log n)$.

If A and B lie on a circle the weight of each edge, (a_i, b_j) is simply the shortest arc length between the two points. In the case where $|A| = |B|$, Karp and Li. [11] prove that that the cost of the optimal matching is given by:

$$\min_h \int_{x=0}^{X} |H(x) - h| \, dx, \quad \text{where } min(H(x)) \leq h \leq max(H(x))$$

In the case where $|A| = |B|$, the authors show that the h that minimizes the cost is simply the median value of $H(x)$. Letting $h^* =$ median of $H(x)$ they define $f(x) = H(x) - h^*$. They compute x^* such that $f(x^*) = 0$. If we were to imagine an edge, (a_i, b_j), to be an arc connecting the two points, a_i and b_j, the point x^* represents a point on the circle such that the edges of optimal matching do not pass over x^*. Once x^* is computed, the points of A are relabeled $\{a_1, ..., a_k\}$, where a_1 is the first point in the clockwise direction from x^* and a_k is the last. The points of B are relabeled in the same fashion. The optimal matching, Q, is then simply defined as $Q(a_i) = b_i$. Q could also be viewed as the identity matching of the set of points on the line produced by breaking the circle at x^*.

If A and B on the circle are such that $|A| > |B|$, the set E is defined as in equation 2.1. This time the function to be minimized is:

$$\min_h \min_E \int_{x=0}^{X} |H(x) - h| \, dx \tag{2.2}$$

The authors prove that the optimal value of equation 2.2 is:

$$\min_h \left[\int_{x=0}^{X} |H(x) - h| \, dx - \sum_{l=h+1}^{h+e} P(a_l^*) \right], \qquad \text{where } a_l^*$$

$$\text{is leftmost point in } A \text{ that maximizes } P(a) \text{ and } H(a_l^*) = l,$$

$$min(H(x)) \leq h \leq max(H(x))$$

In this case, h is not so easily determined as the case where $e = 0$. However, the algorithm still has running time of $O(n \log n)$, because the number of h values to test is bounded by $O(n)$. For each h value it has been already shown that it takes $O(n)$ time to find the optimal E set. Therefore the total complexity is dominated by the sorting step which is $O(n \log n)$

In a slightly modified version of the problem, we can consider each b_i as a sink that has a certain demand, d_i, and each a_j as a source that has a certain supply, s_j, where $\sum_{i=1}^{|B|} d_i \leq \sum_{j=1}^{|A|} s_j$. This means that one a_i may be assigned to more than one b_j and the opposite may also be true. Karp and Li, state that the same algorithms for finding the minimum-weight one-to-one matchings on a circle and a line can be extended to compute the optimal matchings in this case with $O(n^2)$ order of complexity. However, Aggarwal et al. [1] provide a more efficient algorithm that can solve the same problem in $O(n \log n)$ time. The main property that is used to come up with a more efficient algorithm is the fact that in the optimal matching on a circle or on a line, a point must be connected to its *left or right partner*. In the case of a circle, the left partner for a point a_i is defined as the first point b_j in the counter clockwise direction such that there is an equal number of a's and b's between a_i and b_j. The right partner has the same definition but in the clockwise direction.

In Werman et al. [23], the problem of finding a minimum-weight one-to-one matching on the circle is solved with an algorithmic complexity of $O(n \log n)$. The weight on the edges is the shortest Euclidean arc length as before. The technique used is very similar to Karp and Li's technique [11]. Using the height function, $H(x)$ a place on the circle where no optimal matching edges pass is found. By cutting the circle at such a gap, the problem is then transformed into finding the optimal matching one a line. This is easily computed by using the identity matching. In addition, A novel algorithm is proposed for finding the translation that would optimize the one-to-one matching on a line. That is, if we are allowed to translate each element of A (or B) by the same amount, we want to find the translation that would result in the minimum-weight one-to-one matching. The authors call an edge (a_i, b_j) a left edge if $a_i < b_j$, and a right edge if $a_i > b_j$, and a straight edge if $a_i = b_j$. They prove that the optimal translation ensures that, $\#(straight\ edges) \geq |\#(right\ edges) - \#(left\ edges)|$. Using this property the optimal translation is computed. This is done in $O(n \log n)$ time. Similarly, the problem of finding the optimal rotation that would lead to a minimum-weight one-to-one matching on the circle is solved in $O(n^2)$ operations.

Using the profit maximization scheme proposed by Karp and Li [11], Calannino et al. [6] present an $O(n \log n)$ algorithm for finding the minimum-weight many-to-one matching. For an optimal many-to-one matching, M, let $M^{-1}(b)$ represent the set of all $a \in A$ that are matched to b. The key observation made in [6] is captured in following lemma:

Lemma 2.2.1. *Let M be a minimum-weight many-to-one matching, let $b \in B$ be such that $M^{-1}(b)$ contains two or more elements. Then for each $a \in M^{-1}(b)$, b is a nearest neighbour of a.*

This observation is used to augment the profit function defined in [11] to formulate a $O(n \log n)$ algorithm.

An $O(n \log n)$ algorithm for solving the many-to-many matching problem is proposed by Calannino et al. [7], where $n = |A| + |B|$. Without loss of generality, the set A is assumed to have the left-most element in $A \cup B$. The algorithm partitions the set of sorted points into $P_0, P_1, P_2, ...$ subsets such that all points in P_i are less than all points in P_{i+1}. P_0 is a maximal subset of consecutive points in A, P_1 is a maximal subset of consecutive points in B, and so on (see Figure 2.3). The algorithm then computes $C(p_i)$ which is the cost of the optimal matching up to and including the point p_i. This is done using a dynamic programming approach that uses special properties of the structure of the optimal matching to efficiently compute $C(p_i)$. These special properties will be elaborated on further in the next section.

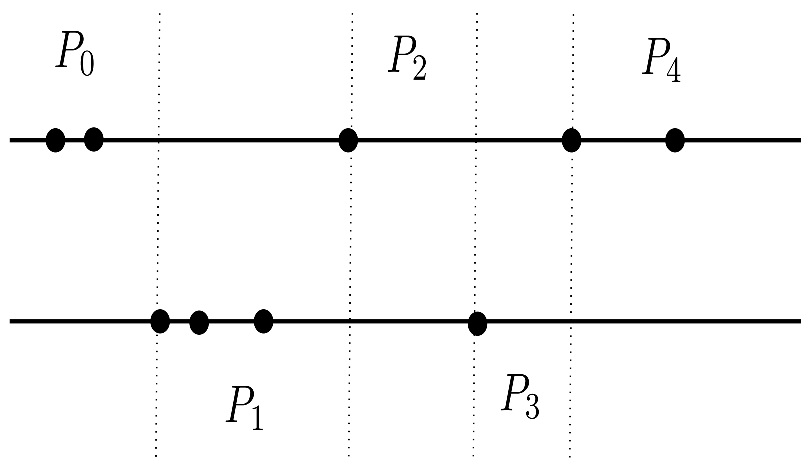


Figure 2.3: Partitioning of the set $A \cup B$

Chapter 3

Algorithms for Computing Sequence Distance

We have presented different applications of comparing two sequences. Our work is mainly motivated by the task of comparing rhythm patterns, or tuning systems. To be more specific, if we wanted to compare two different tuning systems, we can represent each scale in two different tuning systems by a sorted numerical sequence, and then determine the similarity of the sequences. It makes sense to use a sorted sequence since the notes that make up the scale are ordered according to their frequency. In this chapter, we formally describe these problems and present algorithms to solve them.

We represent a collection of musical pitches as two sets of non-negative integers, A and B. A is an ordered sequence of m values: $(a_1, a_2, ..., a_m)$. B is an ordered sequence of n values. We let $k = m + n$. The goal is to find the cost of the minimum-weight many-to-many matching between A and B. We call this cost, the *distance between A and B* as it serves as a similarity measure for A and B. It is assumed that

these sequences are already normalized to eliminate any possible scaling factors.

The actual weight that is used for each edge (a_i, b_j) in the matching can be different. In this work, we concern ourselves with two different weighting schemes. The first specifies the weight of an edge (a_i, b_j) as the Euclidean distance in one dimension, $|a_i - b_j|$. The second is simply the square of the Euclidean distance, $|a_i - b_j|^2$. These two measures and others were used due to the fact that the musical community is interested in comparing scales and rhythms using these two measures [22].

We will use d_1 to label the first measure and d_2 to label the second. $d_1(A, B)$ will represent the distance between A and B under the d_1 measure and $d_2(A, B)$ will represent the distance between A and B under the d_2 measure. We also use $d(A, B)$ as a general term to represent both $d_1(A, B)$ and $d_2(A, B)$. Suppose $D = d(A, B)$ then there exists an optimal (minimum-weight) many-to-many matching between A and B, $M^\dagger$ such that:

$$D = \sum_{(a_i,b_j)\in M^\dagger} d(a_i, b_j)$$

and

$$D \leq \sum_{(a_i,b_j)\in M} d(a_i, b_j)$$

for every many-to-many matching, M, of A and B. We call $d(A, B)$ the *static distance* between A and B.

A slightly modified version of this problem is one in which we are allowed to translate the sets A and B. Put more precisely, we define $A^s = (a_1+s, a_2+s, ..., a_m+s)$ and $B^t = (b_1+t, b_2+t, ..., b_n+t)$. There exist translations s and t that would minimize the total weight of the many-to-many matching between A and B. To make things

simpler, we could fix A and translate only B. We let the distance, D $= d(\bar{A}, \bar{B})$, represent the minimized weight of the matching under translation. We call $d(\bar{A}, \bar{B})$ the *dynamic distance* between A and B.

One characteristic of musical pitches is that pitches that are an octave apart belong to the same pitch class. For example, the note A440, with fundamental frequency 440 Hz, is recognized by human perception as an octave below the note with twice the fundamental frequency, A880. Similarly, it is perceived as an octave above the note with half the fundamental frequency, A220. All three notes are given the same name although they differ in absolute pitch [3]. Due to this characteristic, it is best to represent musical sequences as circular sequences. A circular sequence is defined as two sequences, A and B, that lie on a circle of a circumference C. This would mean the value x and x mod 3 laying on a circle of circumference three would occupy the same position on the circle. The distance between two points on the circle is the length of the shortest arc between the two points. Supposing $a_i \leq b_j$ then $d_1(a_i, b_j) = min(b_j - a_i, a_i - b_j + C)$ and $d_2 = d_1^2$. We use $\hat{A}, \hat{B}$ to represent the circular sequences derived from A and B and represent the distance between them as $d(\hat{A}, \hat{B})$. We call $d(\hat{A}, \hat{B})$ the *static distance for circular sequences.* We also extend the dynamic distance to the circle by allowing a rotation t *modulo* C to the set B. We call $d(\breve{A}, \breve{B})$ *the dynamic distance for circular sequences.*

In the following sections, we describe algorithms to compute the four different types of distances described above: $d(A, B)$, $d(\bar{A}, \bar{B})$, $d(\hat{A}, \hat{B})$, and $d(\breve{A}, \breve{B})$ under the d_1 and d_2 measures. Please note that the following two sections have been adapted from [16]. However, the proofs related to Algorithm 3.3 and the experimental results are the novel work of this thesis.

3.1 The Static Distance

In order to compute the many-to-many matching that minimizes $d(A, B)$ we use a *dynamic programming* approach. It stores the optimal solutions to each subproblem in a table, W, of dimension $m \times n$. The entry w_{ij} in table W stores the optimal matching, $d(A_i, B_j)$, of A_i and B_j, where $A_i = (a_1, a_2, ..., a_i)$ and $B_j = (b_1, b_2, ..., b_j)$. The basic idea of the algorithm is that for the problem of obtaining an optimal matching between A_i and B_j, the edge (a_i, b_j) must always be part of the matching. This is clearly true because: if $b_j > a_i$ then a_i is the closest point to b_j so it is the optimal point in A_i to be paired with b_j. If $a_i > b_j$ a similar argument applies. The edge (a_i, b_j) can be added to the optimal solution of three possible subproblems, namely: $(A_i, B_{j-1}), (A_{i-1}, B_j), (A_{i-1}, B_{j-1})$. The subproblem with the minimum distance is chosen and the edge (a_i, b_j) is added to it (see Figure 3.1). Refer to Algorithm 3.1 for the psuedocode.

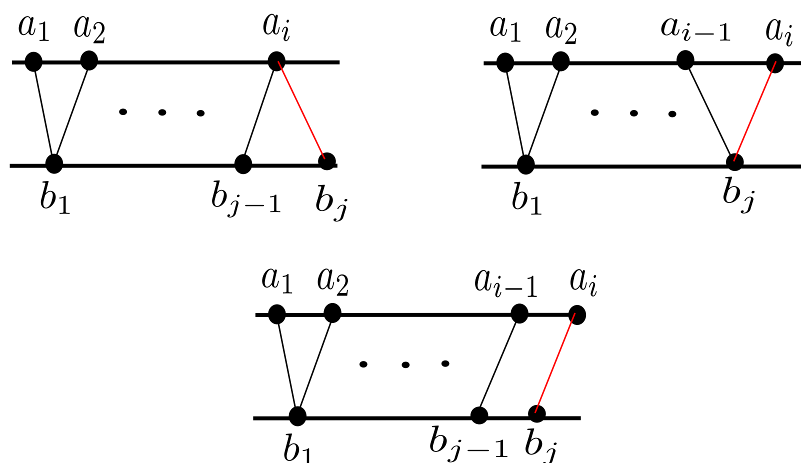


Figure 3.1: Dynamic Programming Algorithm chooses one of the three subproblems

In order to prove Algorithm 3.1 is correct we prove the following lemma:

Algorithm 3.1 Dynamic Programming Algorithm to compute the minimum cost many-to-many matching for the static distance using both the d1 and d2 measures

---

{(a_1, b_1) will always be part of the matching}
$W_{1,1} \leftarrow d(a_1, b_1)$
for $i = 2$ to m **do**
  $W_{i,1} \leftarrow W_{i-1,1} + d(a_i - b1)$
end for
for $j = 2$ to n **do**
  $W_(1, j) \leftarrow W_{1,j-1} + d(a1 - b_j)$
end for
for $i = 2$ to m **do**
  for $j = 2$ to n **do**
    $W^* \leftarrow \min(W_{i-1,j}, W_{i,j-1}, Wi - 1, j - 1)$
    $W_{i,j} \leftarrow W^* + d(a_i, b_j)$
  end for
end for
{W_{mn} stores the weight of the optimal many-to-many matching}
return W_{mn}

Lemma 3.1.1. *The optimal value of* W_{ij} *is given by* $W^* + d(a_i, b_j)$ *where* $W^* = min(W_{i-1,j}, W_{i,j-1}, W_{i-1,j-1})$

Proof. As stated above, clearly, the edge (a_i, b_j) must be part of the minimal many-to-many matching of A_i and B_j. The edge is connected to a minimum cost many-to-many matching of A_{i-1} and B_j or A_i and B_{j-1} or A_{i-1} and B_{j-1}. Since the best subproblem is chosen, W_{ij} must be optimal. □

Once table W is computed, the actual matching can be extracted from it in $O(mn)$ time. The idea is to traverse table W from the entry $W_{m,n}$ backwards until the entry $W_{1,1}$ is reached. At each step in the traversal there are three choices to make and the one with the minimum weight is chosen. Algorithm 3.2 describes this process in more detail.

Clearly the combined complexity of both algorithms is bounded by the size of table

Algorithm 3.2 Reconstruct the Minimum Cost Matching of A and B

{Initialization}
$M \leftarrow (a_1, b_1)$
$i \leftarrow n$
$j \leftarrow m$
while $i > 1$ and $j > 1$ **do**
  $M \leftarrow M \cup (a_i, b_j)$
  if $W_{i-1,j-1} + d(a_i, b_j) = W_{i,j}$ **then**
    $i \leftarrow i - 1$
    $j \leftarrow j - 1$
  else if $W_{i-1,j} + d(a_i, b_j) = W_{i,j}$ **then**
    $i \leftarrow i - 1$
  else
    $\{W_{i,j-1} + d(a_i, b_j) = W_{i,j}\}$
    $j \leftarrow j - 1$
  end if
end while

W, therefore the total complexity of finding the minimum-weight matching is $O(mn)$. As previously noted in the background section, the minimum-weight many-to-many matching using the d_1 measure can be computed more efficiently with complexity, $O(k \log k)$ [7]. The main property that allows for this efficient algorithm is the fact that the optimal way to match s consecutive points in two consecutive partitions is to use the identity matching where a_i is paired with b_i for $i = 1...s$. However, this property does not hold for the d_2 measure (see Figure 3.2). Therefore, it is not clear whether a more efficient algorithm exists for the d_2 measure.

3.2 The Dynamic Distance

In order to compute the dynamic distance, $d(\bar{A}, \bar{B})$, one needs to find the translation, t, of B, that would lead to the minimum-weight many-to-many matching. In case of

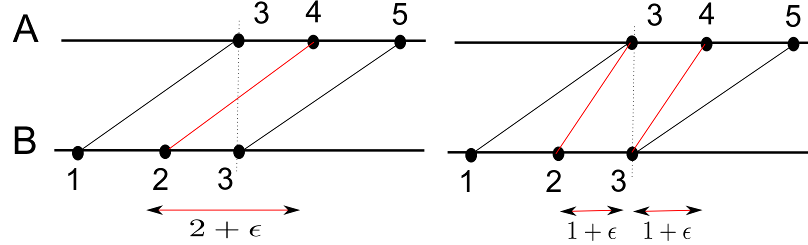


Figure 3.2: The identity matching on the left is optimal for d_1 with $d_1(A, B) = 6$. However it is not optimal for d_2 where $d_2(A, B) = 12$. The matching on the right is optimal for d_2 with $d_2(A, B) = 10$. As can be seen the edge (a_2, b_2) is only optimal for d_2 if $\epsilon > \sqrt{2}$ or $\epsilon < -\sqrt{2}$

d_1, the following lemma shows that only $O(mn)$ translations are sufficient to find the optimal matching.

Lemma 3.2.1. *Suppose D=$d_1(\bar{A}, \bar{B})$. Then there exists a translation, t, and a many-to-many matching M of A and B^t such that $d_1(A, B^t) = D$ and there is an element of A that is coincident with an element of B. That is there is $a_i \in A$ and $b_j \in B^t$ such that $a_i = b_j + t$*

Proof. Suppose D $= d_1(\bar{A}, \bar{B})$. If there happens to be a point from A that coincides with a point from B then we are done. Otherwise, we show that there is a translation $t_2 \neq t$ of B such that $d_1(A, B^{t_2}) \leq d_1(A, B^t)$ and there is an $a_i \in A$, $b_j \in B^{t_2}$ that coincide. Let M be a matching of A and B^t. We partition the edges of M into left edges and right edges. For $a \in A$, $b \in B$, an edge is a left edge if $a < b$ and a right edge if $a > b$. Since none of the points coincide, we know we don't have the case where $a = b$. Now, if the number of left edges is greater than the number of right edges we move the set B to the right until we encounter the first coincidence. We denote this new translation of B as t_2. Clearly, $d_1(A, B^{t_2}) < d_1(A, B^t)$. A symmetric argument applies if the number of right edges is larger than the number of left edges.

If the cardinality of the left edges and right edges is the same then we move B to the right or the left until we encounter the first coincidence. In this case $d_1(A, B^{t_2}) = d_1(A, B^t)$. □

The total complexity of the algorithm = number of translations × complexity of computing $d(A^t, B) = O(mnk \log k)$.

The same argument cannot be extended to the d_2 measure. To see this, suppose $A = (a_1 = 2, a_2 = 4, a_3 = 6)$, $B = (b_1 = 2, b_2 = 4)$. Aligning any element of A with any element of B results in a $d_2(A, B) = 4$. On the other hand, if we translate B by $t = 1$, we get $d_2(A, B^1) = 3$. In fact, this is still not optimal. The optimal value is $t = 1.33$, where $d2(A, B^{1.33}) = 2.667$. To the best of our knowledge, currently, there does not exist an "easy" way of computing the optimal translation, $t_{optimal}$, that would lead to computing $d_2(\bar{A}, \bar{B})$.

We have developed an algorithm that takes a finite number of steps to find $t_{optimal}$. Let M be the matching for $d_2(A, B)$. In table W in Algorithm 3.1, $W_{i,j} = \sum_{(a_i,b_j)\in M} (a_i - b_j)^2$. We now add the translation variable, t, to every entry of the table W. The modified entry is $W_{i,j} = \sum_{(a_i,b_j)\in M} (a_i - b_j - t)^2$.

Our approach is to iterate over the real line over discrete translations, t, where there is a change in table W. Recall that computing $W_{i,j}$ = the cost of edge (a_i, b_j), $d(a_i, b_j)$, added to one of the following three subproblems: $W_{i-1,j}, W_{i,j-1}, W_{i-1,j-1}$. Therefore a change in W happens when one of the non-chosen subproblems becomes a better choice than the currently chosen subproblem. Graphically, each subproblem is represented by a parabola, therefore it is easy to determine where a change might happen by computing the intersection of the parabola representing the current choice with the parabolas representing the two other choices. Using this idea we formulate

an algorithm for finding the optimal translation, $t_{optimal}$. We simply start with the set B all the way to the left of A and compute W. Next, we find all intersections between each chosen subproblem (i.e part of the matching) and the two non-chosen subproblems related to it. Out of all intersections, we pick the one with the smallest positive t value. We translate B by this t value and repeat the process until B is translated all the way to the right of A. We store each minimum-weight matching as B is being translated. Once B has been translated all the way to the right of A, we pick the smallest value out of all stored values (Refer to Algorithm 3.3).

The correctness of the algorithm is proved in two steps. First we prove that the algorithm terminates with an exponential upper bound and then we prove that it computes $d(\bar{A}, \bar{B})$.

We first define a table, F, that stores the choice of subproblem made for each entry W_{ij} in W. It can store the values 1 or 2 or 3 depending on which one of $W_{i-1,j-1}, W_{i-1,j}, W_{i,j-1}$ is the optimal subproblem for W_{ij}. We use the term state F to represent the current choices that are made by the algorithm.

Lemma 3.2.2. *Once an entry, f_{ij} in table F changes, it can only change back to its original value if a previous entry in F changes. A previous entry, $f_{i'j'}$, is one where $i' < i, j' \leq j$ or $j' < j, i' \leq i$.*

Proof. Each subproblem is simply a quadratic equation. A change in an entry f_{ij} means one quadratic equation is less than the currently chosen quadratic equation beyond a specific t value. If a previous entry does not change it means the competing subproblems that yield the choice for f_{ij} simply remain the same and therefore f_{ij} will not change back to its previous value. Only a change to a previous entry can potentially change one of the three competing subproblems and the entry f_{ij} could

Algorithm 3.3 Algorithm for computing $d_2(\bar{A}, \bar{B})$

```
{Initialization}
results ← an empty list
{Shift all of B to the left of A (i.e. b_n = a_1)}
t ← 0
limit ← a_m {Used to stop the loop when all of B is the right of A (i.e. b_1 = a_m)}
while t ≤ limit do
   Compute W for (A, B^t) using Algorithm 3.1
   E ← equation of parabola at W_{m,n}
   {Find the optimal distance for this specific matching}
   E' ← first derivative of E
   optimalT ← t where E' = 0
   optimalD ← E(optimalT)
   Add (optimalT, optimalD) to results
   for i from 2 to m do
      for j from 2 to n do
         intersects[i, j] ← positive intersections of current subproblem with the other
         two subproblems for W_{i,j}
      end for
   end for
   nexTrans ← min(intersects[i, j])
   B^t ← B^t + nextTrans
   t ← t + nextTrans
end while
index = min(first column of results)
bestT ← results[index, 2]
bestMatching ← result of running Algorithm 3.1 on (A, B^{bestT})
return bestMatching
```

potentially revert back to its previous choice. □

Theorem 3.2.3. *Algorithm 3.3 terminates and has an upper bound of* $O(3^{mn})$.

Proof. We can bound Algorithm 3.3 by the number of possible changes that can potentially occur in table, F. Since each entry in F can take on three different values, namely, {1,2,3}, we clearly have an upper bound of 3^{mn} possible different F tables or states. We must also show that the algorithm does not reach a state F more than once. We do this through by contradiction. Assume the algorithm reaches some state F then some other state, F' and finally go back to F again. This means there is at least one entry, f_{ij}, in F that changes in F' and then changes back to what it was to produce F again. By lemma 3.2.2, this can only happen if some previous entry, $f_{i'j'}$ changes. If f_{ij} is back to its original value then $f_{i'j'}$ is different. We apply the same argument to $f_{i'j'}$ as we did to f_{ij} which means a previous entry to $f_{i'j'}$ must change to allow for $f_{i'j}$ to revert back to its original value. Following this line of argument, it is clear that the entry f_{11} must also change. This is a contradiction because the edge (a_1, b_1) is always part of the matching and therefor f_{11} can never change.

□

Theorem 3.2.4. *Algorithm 3.3 computes* $d_2(\bar{A}, \bar{B})$

Proof. Assume there is a translation, t, of the set B for which $d_2(\bar{A}, \bar{B})$ is realized. Clearly, $a_1 - b_n \leq t \leq a_m - b_1$. Therefore t must correspond to a matching in which B falls within the above range. Algorithm 3.3 considers all possible matchings between A and B that occur as B is being translated within $(a_1 - b_n, a_m - b_1)$. It does so by recomputing the matching every time one of the subproblems becomes a better choice than any other subproblem anywhere in the table W. Therefore the matching that

corresponds to t must be found by Algorithm 3.3. For every matching considered, the algorithm finds the translation that optimizes it. This translation must be equal to t since t is optimal. □

3.2.1 Experimental Results for Algorithm 3.3

The upper bound that we provide for Algorithm 3.3 does not appear to be tight. We have not been able to construct an example that requires a super-polynomial number of steps. We ran various experiments to gain a better understanding of the true running time of the Algorithm. We know that for each translation that the Algorithm makes it takes $O(mn)$ time to compute the distance using the dynamic programming algorithm. What concerns us is the number of translations that Algorithm 3.3 makes before finding the optimal translation. Therefore, the number of translations determines whether the Algorithm has polynomial running time or not. Instead of comparing the number of translations to the total cardinality of the two sets $(m+n)$, we chose to compare it to the product of the cardinalities mn instead. We defined a ratio, R, by the following equation:

$$R = \frac{\text{number of translations}}{mn}$$

Clearly, if R does not super-polynomially increase in our experimental runs that would suggest that Algorithm 3.3 is polynomial in the size of the input.

The first experiment was to randomly generate the sets A and B with specific cardinalities, m and n. For the cardinalities chosen, A and B were randomly generated 5 times while running the algorithm on each newly generated pair. Out of the 5 runs, the run that caused the largest number of translations is reported. Table 3.1 captures

Table 3.1: Experimental Results of running Algorithm 3.3

m	n	Number of Translations	R	R_2	3^{mn}
5	5	63	2.52	0.78	8.5×10^{12}
8	10	247	3.09	0.70	1.47×10^{38}
15	11	589	3.57	0.70	5.31×10^{78}
16	20	1182	3.69	0.64	4.77×10^{152}
21	20	1556	3.70	0.61	2.46×10^{200}
25	28	2655	3.79	0.62	9.66×10^{333}
30	30	3520	3.91	0.62	2.57×10^{429}
30	20	2379	3.96	0.58	2.87×10^{286}
15	31	1758	3.78	0.57	7.27×10^{221}
40	45	7038	3.91	0.52	6.58×10^{858}
50	55	10990	4.00	0.54	1.21×10^{1312}
60	61	14769	4.01	0.51	1.84×10^{1746}
25	80	8151	4.08	0.49	1.74×10^{954}
90	100	37176	4.13	0.45	1.23×10^{4294}

the results of this experiment. It can be seen that the number of translations of the Algorithm is much less than the theoretical upper bound that we have provided. Furthermore, the ratio, R, has a maximum value of 4.13. Plotting mn versus the number of translations we can clearly see that there exists a linear relationship between the two quantities (Figure 3.3). Looking at table 3.1 we see that the value of R seems to be slowly increasing. Testing the conjecture that the number of translations is poly-logarithmic, we define a new ratio R_2 as:

$$R_2 = \frac{\text{number of translations}}{mn \times \log(mn)}$$

Referring to Table 3.1, it can be seen that R_2 decreases as m and n increase. This appears to indicate that the running time of Algorithm 3.3 cannot be larger than $mn \log(mn)$.

Figure 3.3 plots the number of translations versus both mn and $mn \log(mn)$. The

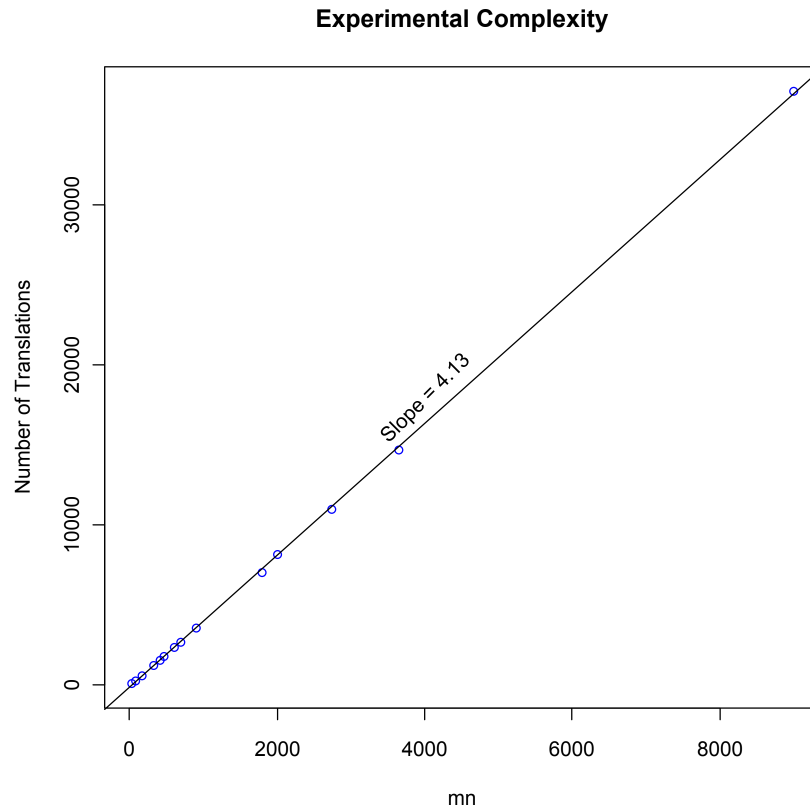
Table 3.2: Experimental Results of running Algorithm 3.3 on Compressed B datasets where $m = n = 10$

Divisor	Number of Translations	R	R_2
1	180	1.80	0.39
2	110	1.10	0.24
4	207	2.07	0.45
6	225	2.25	0.49
10	282	2.82	0.61
10^2	540	5.40	1.17
10^3	560	5.60	1.22
10^4	426	4.26	0.93
10^5	367	2.67	0.80
10^6	223	2.23	0.48
10^7	224	2.24	0.49
10^8	116	1.16	0.25
10^9	66	0.66	0.14

graph indicates a relationship that is almost linear in both cases. Again, this agrees with our analysis since neither R nor R_2 grow exponentially.

In our second experiment, A and B were two randomly generated sets of 10 points where A and B are the same. We compressed B by dividing the elements of B by an ever increasing factor and ran the Algorithm. Overall the number of translations seemed to increase as B was further compressed by dividing by a larger number. However, the trend reached a peak, and as B was compressed further the number of translations started to decrease. Table 3.2 summarizes our results.

Picking the largest number of translations in this dataset, we can see that the value of R seems to be slightly higher than the random data set of Table 3.1. Namely, 5.60 versus 4.13. We performed the same experiment with different cardinalities. The same pattern was noticed, R increased at first as B is compressed further. However, a peak R is reached and further compression of B leads to decreasing R. For each



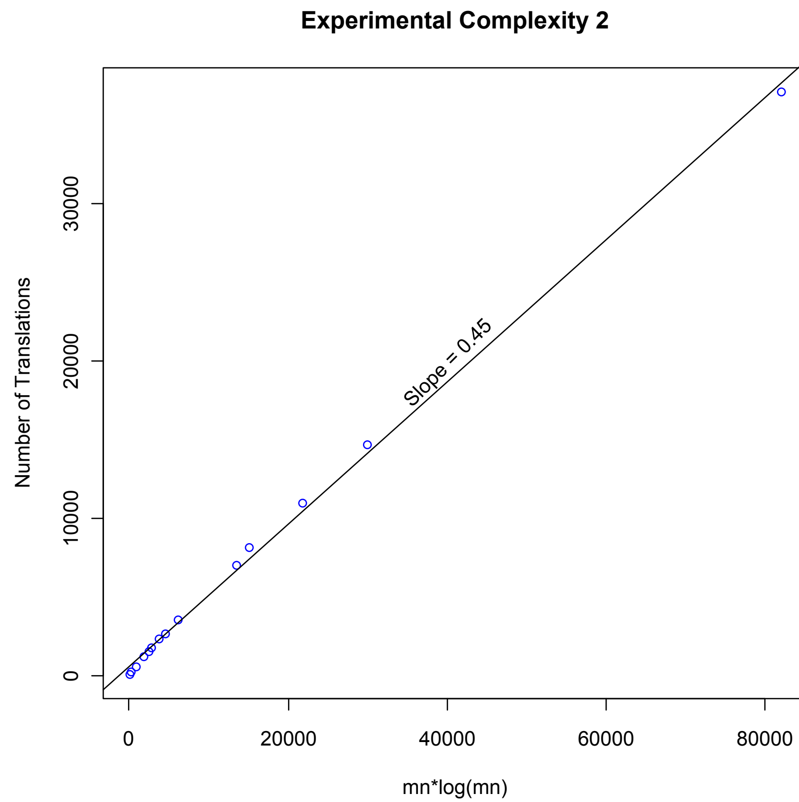


Figure 3.3: Experimental Complexity in the domain of $m \times n$ and $m \times n \log(m \times n)$

Table 3.3: Experimental Results of running Algorithm on Compressed B datasets of different cardinalities 3.3

$m = n$	$m \times n$	Number of Translations	R	R_2
15	225	1817	8.07	1.49
20	400	2680	6.70	1.18
25	625	4266	6.83	1.06
30	900	9402	10.45	1.54
35	1225	10822	8.83	1.24
40	1600	13741	8.59	1.16
45	2025	14761	7.29	0.96

cardinality, Table 3.3 shows the results of the compression that caused the largest number of translations and therefore the largest R. The R values are higher than all previous results, but we still do not notice an exponential increase of R as the cardinalities of the two sets increase. These results support the fact that the running time of Algorithm 3.3 appears to have a polynomial running time. Similar results were obtained for $A \neq B$.

Another possibility for a data set that might have not been well captured by generating points randomly, is to have configurations that contain clusters of contiguous points. We generated different cluster configurations for $m = 15$ and $n = 11$. In the results table a configuration of (3,2) indicates the set A is composed of 3 clusters of points separated by some distance, and likewise the set B is composed of 2 clusters. Each cluster is composed of a fixed number of points that fall within a specific range. The cluster points are generated randomly within the specified range for each cluster. For each cluster configuration, the experiment was run 10 times and the trial with the largest number of translations is shown in Table 3.4. It can be seen in Table 3.4 that the largest value for R is 4.11 which is lower than the largest R value in table 3.1.

Table 3.4: Experimental Results of running Algorithm 3.3 different clustered configuration where $|A| = 15$ and $|B| = 11$

Configuration	Number of Translations	R	R_2
(3,3)	615	3.73	0.73
(3,2)	676	4.10	0.80
(4,4)	616	3.73	0.73
(4,2)	644	3.90	0.77
(4,1)	678	4.11	0.80

We can conclude from these results that Algorithm 3.3 exhibits polynomial runtime behaviour.

3.3 The Static Distance for Circular Sequences

The main challenge when points are placed on a circle is that there is no longer a starting point and an ending point. Therefore, trying to apply the dynamic programming approach of Algorithm 3.1 directly does not work. In order to use Algorithm 3.1, the circle must be transformed into a line. This can be done by breaking the circle at a specific point. To show that this approach will work we must prove that there is at least one point on the circle that is not crossed by any edge of the optimal many-to-many matching. We visualize an edge as an arc that starts at a point a_i and ends at a point b_j. We first show that the optimal matching for circular sequences does not contain opposite pointing arcs as demonstrated in Fig. 3.4.

Lemma 3.3.1. *The minimum-weight many-to-many matching on the circle does not contain a pair of arcs that pass over a single point in opposite directions.*

Proof. The only two possibilities for a pair of arcs passing over a single point in opposite directions are demonstrated in Fig. 3.4. It is clear that we can improve both

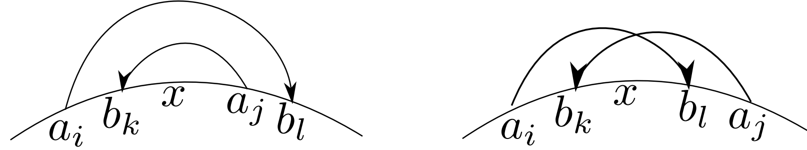


Figure 3.4: Arcs passing a point in opposite directions

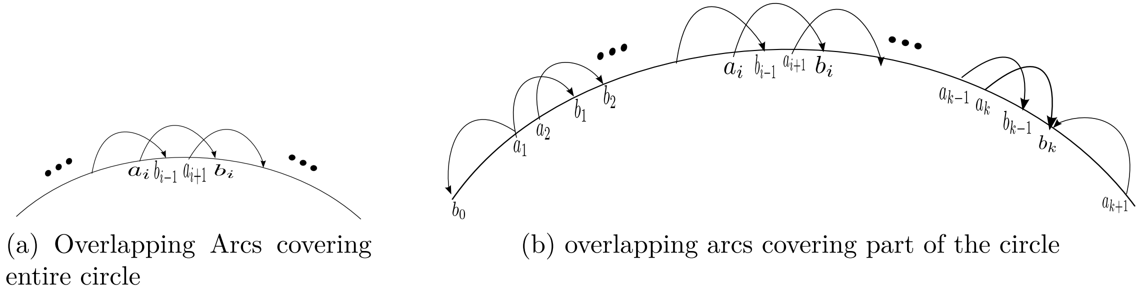
matchings by using the edges (a_i, b_k), (a_j, b_l) instead of (a_i, b_l),(a_j, b_k) to obtain a better matching in which the arcs do not pass over the point x in different directions.

□

Lemma 3.3.2. *In the minimal matching on the circle there exists at least one point such that no arcs pass over it. We call this point a gap in the matching.*

Proof. Assume that in the optimal matching of the circle every point in the circle has an arc passing over it (i.e. covered by an arc). There are two cases to consider:

Case 1: The circle is covered by arcs that do not overlap. Meaning an arc does not start within the interior of another arc. This means there has to be a $b_k \in B$ in between every two consecutive points in A, a_i, a_{i+1}. If there was no b_k in between a_i and a_{i+1} then the only way to cover the region of the circle in between a_i and a_{i+1} is to use two intersecting opposite arcs. By lemma 3.3.1 this cannot happen in an optimal matching. In addition, in order to cover the entire circle every a_i must have an arc to the b point directly to its right, b_R, and to the b point directly to its left, b_L. Similarly, each b_j must have an arc from the a to its right, a_R, and the a to its left, a_L. However, such a matching is not optimal since we can improve it by simply removing all arcs $(a_i$,$b_R)$ for $i = 1...m$. We can do this because the arcs (a_i, b_L) are sufficient to match all points in A to all points in B. This means the circle cannot be completely covered with arcs that have no overlap.



(a) Overlapping Arcs covering entire circle

(b) overlapping arcs covering part of the circle

Figure 3.5: Possible scenarios for arcs covering the circle in the many-to-many matching

Case 2: There is at least one arc that starts within the interior of another arc. We call these arcs overlapping arcs. Let (a_1, b_1),(a_2, b_2),...,(a_k, b_k) be a subset of consecutive overlapping arcs such that within each arc there is exactly one arc entering and one arc exiting. If this subset covers the entire circle as in Fig. 3.5a then we can improve the matching by replacing each (a_i, b_i) with $(a_{(i+1) \text{ mod k}}, b_i)$. This will lead to a lower cost matching with gaps introduced in this subset. If this subset does not cover the entire circle, b_k must be connected to some a_{k+1} and a_1 is connected to some b_0 as in Fig. 3.5b. This is the only other possibility because by lemma 3.3.1 we cannot have arcs that cross each other in opposite directions at any point in the circle. Clearly, In this case we can improve the matching by replacing each (a_i, b_i) with (a_{i+1}, b_i), for $i = 1..k-1$. It is also possible that $a_{k+1} = a_1$ and $b_0 = b_k$. In this case (a_{k+1}, b_k) and (a_1, b_0) are the same arc. The new matching is better and contains gaps within the subset. If we apply the same improvement to all such subsets we will guarantee that the new improved matching will have gaps. □

Clearly, the point where there is a gap in the matching, referred to as a *gap point* from this point onwards, is surrounded by two neighbouring points that belong to $A \cup B$. These two points are the closest neighbours of the gap point. These points

cannot be connected to one another because there is a gap point in between them. By this property and lemma 3.3.1 these two points basically form the boundaries of a *gap region*, which is a region that has no arcs passing over it. If we traverse the circle in a clockwise manner the right boundary point will represent the first point in the matching on a line and the left boundary point will represent the last. We use the term *good break point* to refer to the right boundary point of the gap region. By breaking the circle at all points in $A \cup B$ then we are guaranteed to find the good break point. Once we have done that we can simply find the optimal solution using the dynamic programming algorithm. See Algorithm 3.4.

Algorithm 3.4 Algorithm for computing $d(\hat{A}, \hat{B})$

$sets \leftarrow$ 'AB'
for i from 1 to 2 **do**
  $currSet \leftarrow sets[i]$
  for j from 1 to length($currSet$) **do**
    $breakPoint \leftarrow currSet[j]$
    $thisA \leftarrow A - breakPoint$
    $thisB \leftarrow B - breakPoint$
    run Algorithm 3.1 on $(thisA, thisB)$ and store result
  end for
end for
return best result of stored results

Clearly the complexity of the algorithm = the number of breakpoints $\times$ cost of computing $d(A, B)$. Therefore, the complexity is $O(k^2 \log k)$ for d_1 and $O(mnk)$ for d_2.

3.4 The Dynamic Distance for Circular Sequences

Computing the dynamic distance, $d(\breve{A}, \breve{B})$ for circular sequences is done through extending the algorithm proposed in the previous section. The idea is to make the $(m + n)$ breaks and then run the appropriate dynamic distance algorithm for each permutation produced by the breaks. In case of $d_1(\breve{A}, \breve{B})$, for each permutation, $O(mn)$ alignment operations are carried out. The alignment operations are done modulo the circumference of the circle, C. Refer to Algorithm 3.5.

Algorithm 3.5 Algorithm for computing $d_1(\breve{A}, \breve{B})$

sets ← 'AB' {The basic idea is to introduce breaks in A and align B with the points in A and then introduce breaks in B and align A with the points in B}
for k from 1 to 2 **do**
  {The set where break points will be introduced}
  $breakSet \leftarrow sets[k]$
  {The set that will be translated}
  $movingSet \leftarrow sets[3 - k]$
  for i from 1 to length(breakSet) **do**
    $x \leftarrow breakSet[i]$
    for j from 1 to length (movingSet) **do**
      align $movingSet[j]$ with x
      $results$ ← optimal matching for this permutation using Algorithm 3.1
    end for
  end for
end for
return min(results)

Theorem 3.4.1. *Algorithm 3.5 computes* $d_1(\breve{A}, \breve{B})$

Proof. Clearly, By lemma 3.2.1 the optimal matching on the circle is one in which an $a \in A$ coincides with a $b \in B$. We also know that the optimal matching on the circle must contain a good break point, x. This means the line transformation of the optimal solution is one where x is the first point on the line and some point

from the set not containing x aligns with one of the points of the set containing x. Clearly, Algorithm 3.5 will encounter this matching because it introduces breaks at all points and thus captures the good break point. In other words, the algorithm will realize the correct permutation of points. Once the correct permutation is realized, the algorithm will capture the alignment of points with the optimal matching because it iterates over all possible alignments. Finally, calling Algorithm 3.1 will compute the optimal matching for this alignment. □

The complexity of this algorithm is $O(k \times (mn) \times k \log k) = O(k^2 mn \log k)$. However, there is a slight efficiency that can be introduced. Notice that breaking the circle at a_i and aligning b_j with a_i will produce the exact same permutation as breaking the circle at b_j and aligning a_i with b_j. This means we have mn duplicate operations in our algorithm. Therefore, our improved running time would be $O((kmn - mn)k \log k)$.

In computing $d_2(\breve{A}, \breve{B})$ a similar approach is followed. For each permutation produced by a break, Algorithm 3.3 is used to compute the optimal matching for that specific permutation. We now present the algorithm and proof of correctness:

Algorithm 3.6 Algorithm for computing $d_2(\breve{A}, \breve{B})$

foreach $a \in A$
  break circle at a
  run Algorithm 3.3 using modulo arithmetic on the permutation produced by the break
endfor
foreach $b \in B$
  break circle at b
  run Algorithm 3.3 using modulo arithmetic on permutation produced by the break
endfor
output the best matching out of all the breaks

Theorem 3.4.2. *Algorithm 3.6 computes* $d_2(\breve{A}, \breve{B})$.

Proof. As previously shown we have a good break point, x. The point x is either part of the set A or part of the set B. In case $x \in A$, it is clear that the set A in the optimal solution can simply be mapped onto a line where x is the first point on the line. Therefore the good break determines the correct permutation of the set A on the line. In order to capture the optimal matching on the line we must determine the correct permutation of the set B on the real line. Since x is not connected to any point in the set B that is behind it, all data points of the set B must have a value larger than or equal to x on the real line. Furthermore, the set B can be mapped on the real line in $|B|$ different permutations. One of these permutations along with the permutation of A determined by the good break holds the optimal solution. Since Algorithm 3.6 translates the set B a total distance of the circumference of the circle, all possible permutations of B are captured. Therefore, Algorithm 3.6 will capture the correct permutation of the set A and the set B on the real line. Once the correct permutation is captured, Algorithm 3.6 uses Algorithm 3.3 to compute the best matching for this permutation . □

The complexity of this algorithm is bounded by the steps that invoke Algorithm 3.3. Therefore the complexity is $O(3^{mn})$.

Chapter 4

Conclusion

In this thesis we looked at previous results dealing with finding a minimum-weight one-to-one matching, many-to-one matching and many-to-many matching. These problems were all in the context of points lying on a line or a circle. We looked at the problem of finding a minimum-weight many-to-many matching with two different weighting schemes. The first weighting scheme was where the weight of each edge is simply the Euclidean distance, d_1. The second was where the weight is the square of the Euclidean distance, d_2. A lower bound for finding minimum-weight many-to-many matching had already been proposed by Colannino et al. [7] for the d_1 measure. We presented an $O(mn)$ algorithm for the d_2 version of the problem. It is an open question whether a more efficient algorithm exists.

We also looked at the problem of finding the minimum-weight many-to-many matching when translations are allowed, the dynamic distance. We provided a novel algorithm for computing the dynamic distance under the d_2 measure but were unable to theoretically prove a tight upper bound. This is the most major contribution of our work as it the algorithm proposed provided a way to discretize the problem of

finding the distance under the d_2 measure. The upper bound that we provided was exponential. However, our experimental results seem to suggest that our Algorithm seems to have a polynomial upper bound. A major open question is to verify our experimental results theoretically.

For points lying on a circle, we proved that the problem on the circle can be solved on the line by finding the correct place to break the circle. We showed that such a "good breakpoint" always exists. We conjecture that our algorithms are doing too much work by breaking the circle at every feasible breakpoint. In the previous literature reviewed, finding the minimum-weight one-to-one matching on the circle required finding the correct breakpoint. However this was done in one step as demonstrated by Werman et al. [23] and Karp and Li [11]. It is still an open question whether computing the good break point can be done more efficiently in case of the many-to-many matching.

Bibliography

[1] Alok Aggarwal, Amotz Bar-Noy, Samir Khuller, Dina Kravets, and Baruch Schieber. Efficient minimum cost matching and transportation using quadrangle inequality. *Journal of Algorithms*, 19(1):116–143, July 1995.

[2] Amir Ben-Dor, Richard M. Karp, Benno Schwikowski, and Ron Shamir. The restriction scaffold problem. *Journal of Computational Biology*, 10(2):385–398, 2003.

[3] Dave J. Benson. Music: a mathematical offering. *The Mathematical Intelligencer*, 30(1):76–77, 2008.

[4] Samuel R. Buss and Peter N. Yianilos. A bipartite matching approach to approximate string comparison and search. Technical report, NEC Research Institute, 1995.

[5] Bilson J. L. Campana and Eamonn J. Keogh. A Compression Based Distance Measure for Texture. 2010.

[6] Justin Colannino, Mirela Damian, Ferran Hurtado, John Iacono, Henk Meijer, Suneeta Ramaswami, and Godfried Toussaint. A $o(nlogn)$-time algorithm for

the restriction scaffold assignment problem. *Journal of Computational Biology*, 13(4):979–989, 2006.

[7] Justin Colannino, Mirela Damian, Ferran Hurtado, Stefan Langerman, Henk Meijer, Suneeta Ramaswami, Diane Souvaine, and Godfried Toussaint. Efficient many-to-many point matching in one dimension. *Graphs and Combinatorics*, 23:169–178, 2007.

[8] M. Fatih Demirci, Ali Shokoufandeh, Yakov Keselman, Lars Bretzner, and Sven Dickinson. Object recognition as many-to-many feature matching. *International Journal of Computer Vision*, 69(2):203–222, 2006.

[9] Harvey J. Greenberg, William E. Hart, and Guiseppe Lancia. Opportunities of combinatorial optimization in computational biology. *INFORMS Journal on Computing*, 16:211–231, 2004.

[10] Kjell Gustafson. The graphical representation of rhythm. *PROPH) Progress Reports from Oxford Phonetics*, pages 6–26, 1988.

[11] Richard M. Karp and Shuo-Yen R. Li. Two sepcial cases of the assignment problem. *Discrete Mathematics*, 13:129–142, 1975.

[12] Eamonn J. Keogh and Michael J. Pazzani. Derivative dynamic time warping. In *First SIAM international conference on data mining*. Citeseer, 2001.

[13] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics*, 2:83–97, 1955.

[14] Chris Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, 2000.

[15] Christos D. Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Dover Pubns, 1998.

[16] David Rappaport. Minimum many to many matchings for computing the distance between two sequences. Technical report, Queen's University, August 2009.

[17] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *Readings in speech recognition*, pages 159–165, 1990.

[18] Anil K. Singh. Study of some distance measures for language and encoding identification. In *Proceedings of the Workshop on Linguistic Distances*, pages 63–72. Association for Computational Linguistics, 2006.

[19] Godfried Toussaint. Sharper lower bounds for discrimination information in terms of variation (Corresp.). *Information Theory, IEEE Transactions on*, 21(1):99–100, 2002.

[20] Godfried Toussaint. A comparison of rhythmic similarity measures. In *Proceedings of the 5th International Conference on Music Information Retrieval*, pages 242–245, 2004.

[21] Godfried Toussaint. The geometry of musical rhythm. *Discrete and Computational Geometry*, pages 198–212, 2005.

[22] Dmitri Tymoczko. The geometry of musical chords. *Science*, 313(5783):72, 2006.

[23] Michael Werman, Shmuel Peleg, Robert Melter, and T. Y. Kong. Bipartite graph matching for points on a line or a circle. *Journal of Algorithms*, 7:277–284, 1986.