8-23-2019 11:30 AM

A New Approach to Sequence Local Alignment: Normalization with Concave Functions

Qiang Zhou, *The University of Western Ontario*

Supervisor: Kaizhong Zhang, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

© Qiang Zhou 2019

Recommended Citation

Zhou, Qiang, "A New Approach to Sequence Local Alignment: Normalization with Concave Functions" (2019). *Electronic Thesis and Dissertation Repository*. 6470.
https://ir.lib.uwo.ca/etd/6470

Abstract

Sequence local alignment is to find the most similar segment pair from the two input sequences. The Smith-Waterman algorithm is one of the essential techniques in sequence local alignment, especially in computational molecular biology. This algorithm produces the optimal sequence local alignment, which is defined to be the segment pair with the highest similarity score as long as the similarity metric used is additive. However, the solution obtained by the Smith-Waterman algorithm may not be ideal in some cases. The segment pair produced by the algorithm may contains pieces of non-conserved regions between highly conserved regions as long as the whole segment pair has the highest similarity score.

In order to obtain consistently similar segments between two sequences, the concept of using segment lengths to normalize the corresponding local alignment similarity score was proposed. In this thesis, some existing algorithms for normalized sequence alignment will be discussed. We first generalized the concept of normalization with segment length to normalization with the normalized similarity metric. Then, we proposed a new algorithm to compute the optimal sequence local alignment with normalized similarity metric. Given a set of normalization (concave) functions, our algorithm can efficiently compute all the optimal sequence local alignments for every normalization function all together.

Keywords: Protein sequence, sequence local alignment, similarity metric, normalized similarity metric, red-black tree, dynamic programming, phylogenetic tree.

Lay Summary

Sequence comparison tools are widely used in many areas, such as the studies regarding DNA or protein sequences. The target is to find the most similar regions between any two sequences. Usually, an optimal similar region should consist of identical parts and some dissimilar fragments. Many existing applications may only focus on including identical regions as many as possible; however, the dissimilar fragments also need to be considered to measure the similarity of two subsequences. Our research provides a new approach to find the consistently similar region of input sequences, based on a normalized similarity metric.

A similarity metric will be applied to measure the similarity for each element pair from input sequences, and a score will be given. Typically, people measure the similarity of two subsequences by summing up the score of each element pair, and the solution should have the highest similarity score. However, besides the similarity score, our method also considered the segment scale to measure the similarity. A high score segment that contains a large poor fragment may not be ideal in our method. The solution found by our method is meaningful because there will not be any relatively large dissimilar fragments that exist in our solution.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Dr. Kaizhong Zhang for his guidance of my study and relative research, for his patience and immense knowledge. It is my great honor to work for Dr. Kaizhong Zhang, and he is the professor whom I respect the most in my life.

Secondly, I would like to thank my parents and my wife Yiwen Hao sincerely. I could not finish my studies without their support and understanding.

Last but not least, I thank the friends and lab mates who gave me valuable suggestions and help since I stepped in Computer Science.

Contents

Abstract **ii**

Lay Summary **iii**

Acknowlegements **iii**

List of Figures **vii**

List of Tables **x**

1 Introduction **1**

2 Background **5**
- 2.1 Sequence Global Alignment 5
- 2.2 Sequence Local Alignment 6
- 2.3 Smith-Waterman Algorithm 7
- 2.4 Similarity Metric 10
- 2.5 Distance Metric 12
- 2.6 Similarity and Distance Metric Normalization 13
- 2.7 Neighbor-joining and Phylogenetic Tree 15
- 2.8 Red-black Tree 17

3 Literature Review **19**
- 3.1 Normalized Editing Distance 19
- 3.2 An improvement of Normalized Editing Distance 23
- 3.3 Normalized Local Similarity Score 27
- 3.4 Similarity Metric and Normalized Similarity Metric 28
- 3.5 BLOSUM 35

4 A New Algorithm for Normalized Sequence Local Alignment **38**
- 4.1 A Simple Algorithm for finding Normalized Sequence Local Alignment 39
- 4.2 Main Idea of New Algorithm 40
- 4.3 Algorithm Design 42
  - 4.3.1 Strategy of Redundant Alignments Rejection 42
  - 4.3.2 Data Structure 46
  - 4.3.3 Strategy of Merging and Sorting Candidates 47
  - 4.3.4 Red-Black Tree 51

4.4 Process Stored Data by Applying Normalized Similarity Metric 54
4.5 Time and Space Complexity Analysis 55
4.6 Corresponding Algorithms 55

5 Experiment **59**
5.1 Protein Sequences 59
5.2 Datasets 60
5.3 Protein sequence local alignment 63
5.4 Build Phylogenetic Tree by Neighbor Joining Method 63
5.5 Experiment Result 66
5.6 Evaluation 67

6 Conclusion **70**

Bibliography **72**

Curriculum Vitae **73**

List of Figures

1.1 $X =$ "$ACAGTC$" and $Y = AGATCT$, there are many ways to align these two sequences, the optimal solution is with most matches and as fewer mismatches as possible. 2

2.1 A matrix H is created to split sequence alignment problem into smaller problems. Suppose we have sequences $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$, where $0 < i \leq n$ and $0 < j \leq m$. At any position (i, j) during dynamic programming, the alignment path can come from $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$. All the candidates will be extended to contain a_i and b_j, and the similarity scores or distances will be accumulated, the one with higher similarity degree will be kept for next dynamic programming steps. 6

2.2 Suppose the similarity scores for regions AB, BC and CD are 100, -120 and 110 respectively. If negative scores can be stored in H, at position C the similarity score will be -20, and 90 for position D; therefore, the optimal local alignment would be region AB, since it is the highest score. However, we know region CD is a better solution with a higher score. Smith-Waterman solved this problem by setting all negative scores to 0 (position C will store value 0), then the similarity score for CD can be obtained correctly. 8

2.3 For any two sequences $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$, suppose we know there are two alignment PR and QR ending at R with similarity score 30 and 20 respectively, and the alignment RS start from R has score 30, then alignment PS always provide higher additive score than QS. In this example, the cumulative score for PS is 60, but 50 for QS. Therefore, Storing the highest score alignment at each $H_{i,j}$ is enough for Smith-Waterman to find optimal alignment. 10

2.4 **Mosaic effect**. The local alignment which is found by the Smith-Waterman algorithm has a score 120, and the scale is 300×300. The whole alignment is composed of three segments, two with score 80 and scale 100×100, the other one with score -40 and scale 100×100 as well. For some applications, only the consistently similar regions are required, such as the two regions with score 80, so that the internal segments with score -40 is useless, but Smith-Waterman will not decompose them into three pieces. 11

2.5 **Shadow effect without overlapping**. There is another alignment with score 80, which is lower than 120, but its scale is much smaller than the algorithm result. That means the low score alignment may have much higher similarity degree than the ”optimal” one; however, Smith-Waterman only targets to find the alignment with the highest similarity score, so another biologically meaningful region will be discarded. 12

2.6 **Shadow effect with overlapping**. Smith-Waterman will take the alignment path which can obtain a higher similarity score 120, rather than a much shorter path with score 80. 13

2.7 This is an example of phylogenetic tree built by complete mtDNA sequences using frequency of k-mers [13]. 16

2.8 (a) is the original distance matrix obtained from sequences comparison; (b) is the Q values calculated by equation 2.3 based on (a), for example, $Q(a, b) = (4 - 2) \times 6 - (6 + 6 + 5) - (6 + 7 + 6) = -24$; (c) is the distance matrix after joining, and $d(u, c)$ and $d(u, d)$ are obtained by equations 2.4, 2.5 and 2.6. . . . 17

3.1 (a) shows the weight function of each pair. (b) shows the result of post-normailzation. The path with minimum weight has been obtained firstly with value 6, and $\hat{W}(P)$ is equal to 1.5 since $L(P)$ is 4; however, it is not the path with actual minimum normalized distance. The ideal path is shown in (c), with normalized distance 1.33 [14]. 20

3.2 The editing graph G_X, Y for the strings $X = aba$ and $Y = bab$. [2] 23

3.3 $a + c$ is the distance of X and Y, and b is the similarity. 32

3.4 Example of concave functions. L_1, L_2 and k are constants, where $L > 0$ and $k \geq 1$. 33

4.1 Any two alignments ending at the same position can have two different shapes based on their segment lengths. In 4.1a, neither of the alignment shapes fully contains the other one. Extending both alignments to C_1 and C_2 could give the opposite result, so either of them can be rejected at position C. In 4.1b, the subsequences of alignment Q fully contain the segments of P. If the similarity score of Q is lower, then no matter how the alignment extends, Q will not give better solution so that it can be discarded at position C. 44

4.2 The general shapes of alignments ending at position (i, j). Suppose there are three candidates O, P and Q ending at (i, j), then the segment lengths for P are x_o and y_o. Also, x_p and y_p for P, x_q, y_q for Q. 49

5.1 Human mitochondrial DNA gene map [5]. 61

5.2 An example of fasta file, the first line is file description, and the rest lines are the sequence of the particular protein which is inside the square bracket at the end of the description. 61

5.3 Rounded BLOSUM62 table, obtained from NCBI. 62

5.4 Two sequences of protein ATP6, one is from human and the other one is from cat. 63

5.5 By applying multiple concave functions, the solutions have different similarity degrees. The solution generated by the Smith-Waterman algorithm is identical with the original sequences. 64
5.6 For alignment of sequence X and Y, we can get L_1 and L_2 length, but if changing sequence Y to Z, new L_1 will be obtained corresponding to the new optimal alignment. 65
5.7 Phylogenetic tree of 20 mammals, the protein sequences were trimmed by the new approach. and the distances were used by neighbor joining are normalized. Similarity metric is *BLOSUM62*. 67
5.8 This Phylogenetic tree used the trimmed protein sequences which were generated by Smith-Waterman Algorithm, and the distances used by neighbor joining were not normalized. 68

List of Tables

3.1 The weight function which makes post-normalization fail triangular inequality. 21

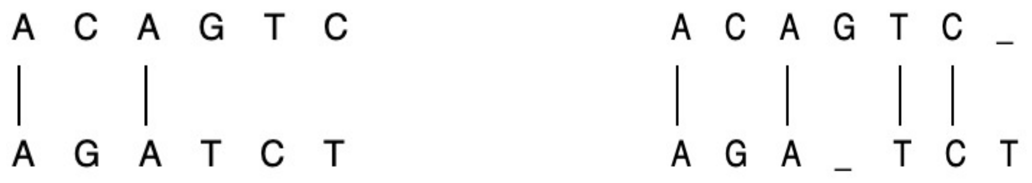
5.1 amino acids table. 60

Chapter 1

Introduction

Let Σ denote a finite alphabet with space, and Σ^* be the set of all finite-length string over Σ. For any two elements x and y from Σ, let $s(x, y)$ denotes the score of aligning x and y, if they are identical or similar, a positive $s(x, y)$ will be given for a **match**, which could be two identical nucleotides for DNA sequences alignment or both identical and very similar amino acids for protein sequence alignment; meanwhile, $s(x, y)$ could be negative for penalizing a mismatch (two irrelevant elements) or aligning one alphabet with space. Suppose we have strings $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$ with length m and n respectively. If spaces are inserted into both A and B to get sequences $A' = a'_1a'_2...a'_L$ and $B' = b'_1b'_2...b'_L$ with same length L, and additive similarity score $s(A', B') = \sum_{i=1}^{L} s(a'_i, b'_i)$ (such a similarity metric is called **additive similarity metric**), then sequence alignment is to find A' and B', which generate maximum $s(A', B')$. For example, in **Figure** 1.1, X and Y are two DNA sequences, where $X =$ "$ACAGTC$" and $Y = AGATCT$. The naive alignment shows in **Figure** 1.1a is to do nothing and align every two letters at the same position from X and Y, respectively. On the other hand, spaces can be inserted to gain more matches, like **Figure** 1.1b. 1.1b aligned the sequence better, or we say alignment shown in 1.1b has higher **similarity degree** (percentage of matches) than 1.1a. In this thesis, if an element is aligned to space, the pair is called an **indel**.

Sequence alignment is widely used in bioinformatics and biostatistics, in order to find the homology or common ancestor through DNA or protein sequences from different species or individuals. There are two kinds of sequence alignment for different purposes: global



(a) Naive alignment (b) Better alignment solution

Figure 1.1: X = "$ACAGTC$" and $Y = AGATCT$, there are many ways to align these two sequences, the optimal solution is with most matches and as fewer mismatches as possible.

alignment and local alignment. A global alignment attempts to align each residue in the input sequences, and the result can show how close the input sequences are. On the other hand, a local alignment focuses on finding high similarity degree regions of sequences; it is usually used in aligning protein sequences to find homology.

There are two kinds of metric can be employed to measure how good alignments are: similarity metric and distance metric. Both of them quantify matches and mismatches of pairs, then sum up the scores for all aligned pairs to obtain the total distance or similarity score, so that different alignments can be compared. The similarity score for any single residue alignment can be positive or negative, but all scores are no less than 0 for distance metric.

In 1965, Soviet mathematician Vladimir Levenshtein proposed the first algorithm for sequence alignment all over the world, named Levenshtein Distance. It is a string metric and tries to minimize the single-character editing number to modify one string to the other [10].

The Needleman Wunsch algorithm was proposed in 1970. Specifically for protein sequences, by using a similarity metric for each possible amino acid pair alignment and indel, the optimal global alignment pattern can be obtained by a dynamic programming approach that splits one massive problem into smaller problems. For each small step, the highest score will be stored and wait for the next step until reaching the ends for both sequences [15]. It is one of the earliest applications using dynamic programming to compare biological sequences [1]. Moreover, later on in 1981, in order to find the homologous part between two molecular sequences, the Smith-Waterman algorithm has been developed to find optimal sequence local alignment. Comparing to the Needleman Wunsch algorithm, there is no negative score stored during dynamic programming; so whenever a positive score appears, it is the start point of a local alignment. After all, the position, which contains the highest score, is the endpoint of the optimal local alignment [17].

Nowadays, Smith-Waterman is a popular algorithm for finding sequence local alignment, and are widely adopted in many applications; However, similarity score was the only criterion in this algorithm to select alignment pattern, regardless of the path's length. The purpose of this algorithm is to find the highest score alignment which cannot be extended on both sides; therefore, if a poorly conserved segment is surrounding by well-aligned subsequences, the algorithm will consider them as one entire alignment, since the aggregate similarity score is the highest. For example, it has been discussed in [3] that when comparing long genome sequences, the output given by Smith-Waterman may not be ideal.

Notice that for any two sequences $A = a_1...a_n$ and $B = b_1...b_m$, if an alignment is found with segments $A' = a_g...a_i$ and $B' = b_h...b_j$, then we denote the length (element numbers) of A' as **x-length**, and the length of B' to be **y-length**. To solve the above problem, both $x - length$ and $y - length$ should also be considered to evaluate the alignment. For example, if there are two local aligned patterns X and Y from the same genomic sequences $A = a_1...a_n$ and $B = b_1...b_m$, the scores of X and Y are 100 and 80 respectively, the $x - length$ and $y - length$ of X are both 120, but alignment Y has both $x - length$ and $y - length$ 50. In this situation, pattern Y will be discarded by the Smith-Waterman algorithm, but it is obvious that elements are more consistently similar in pattern Y than in X. There might be small segments in pattern X, which are not biologically homologous. Due to these, sometimes, pattern B need also be considered as a meaningful alignment. In 2001, a new algorithm was introduced to normalize the similarity score in order to measure the degree of sequence similarity [3]. Then in 2016, normalized similarity metrics and normalized distance metrics have been well-defined in [20], and it has been proved in the paper that the normalized sequence local similarity proposed in [3] is not a similarity metric, because it does not satisfy condition 4 of the definition. Details will be presented in chapter 3.

The similarity normalization strategy is to find sequence local alignment with higher similarity degree than the solution provided by the Smith-Waterman algorithm, but for two fixed input sequences, the higher the output similarity degree is, the shorter the alignment length will be. Therefore, for different applications, different normalization functions need to be applied for the same two sequences due to the different similarity degree requirements. Since most existing algorithms for finding sequence local alignment are based on dynamic programming which consumes a significant amount of time, when applying multiple normalization functions, dynamic programming will be executed multiple times, it leads to massive time consumption.

In order to obtain proper local alignments, we extended the idea of Smith-Waterman to design a dynamic programming algorithm to find optimal sequence local alignment. We used the normalized similarity metric family which is constructed by Minkowski type distance metric to find optimal solutions for different similarity degree requirements. Also, our algorithm only needs to run once, no matter how many normalization functions are applied since all useful data are kept during each dynamic programming run.

In Chapter 2, the specific introduction of the Smith-Waterman algorithm will be given, also the definition of similarity and distance metric, as well as the normalized metric. We will then review some existing algorithms for normalization in Chapter 3. Furthermore, Chapter 4 will introduce our new algorithm specifically, and all core algorithms will be given. Eventually, a experiment result will be shown in Chapter 5; we constructed a phylogenetic tree based on our algorithm output.

Chapter 2

Background

Sequence global alignment is a very basic idea to align two sequences and measure the similarity degree. The idea was extended to find sequence local alignment, and one of the most famous methods is the Smith-Waterman Algorithm. Later on, people noticed that only similarity score is not enough to find consistently similar local alignment; therefore, segment lengths has been considered for normalizing similarity score, aiming to find alignments that are consistently similar. At the same time, distance and similarity metrics have been well defined. Furthermore, since our experiment is to build a phylogenetic tree by the neighbor-joining method, a brief introduction of phylogenetic tree will be given at the end of this chapter.

2.1 Sequence Global Alignment

Sequence global alignment is to find the best arrangement, which can most efficiently align two sequences (achieve the highest similarity score or smallest distance). Specifically, every residue from the compared sequences must be aligned with space or a residue from the other sequence. Suppose we have sequences $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$, where $0 < i \leq n$ and $0 < j \leq m$. In terms of finding optimal global alignment, dynamic programming split the problem into subproblems. A matrix shown in **Figure** 2.1 can be built to keep optimal global alignment scores for subsequences $A' = a_1a_2...a_i$ and $B' = b_1b_2...b_j$ at each position (i, j). Notice that at the very first row and column, all similarity scores are negative (or positive for using distance metric) except value 0 at $(0, 0)$, because whenever a residue is aligned to space, a **gap penalty** δ with negative value should be given for using similarity metric. When we add spaces at the beginning of one sequence, the gap penalties will accumulate from position $(0, 0)$ to $(0, m)$ and $(n, 0)$. Then at each position (i, j) during dynamic programming, the alignment candidates from $(i - 1, j)$, $(i, j - 1)$ and $(i - 1, j - 1)$ will be extended by one more residue alignment, and the highest similarity score will be stored for calculating the rest part of matrix H. Eventually, the one stored in position (n, m) in matrix H will be the score of optimal global alignment.

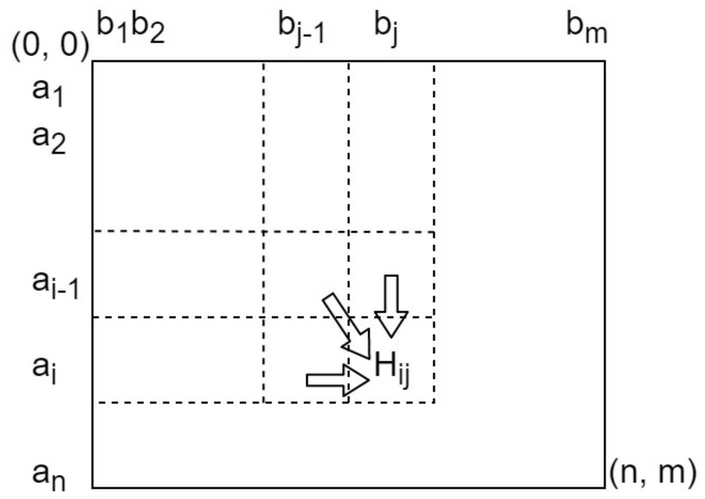


Figure 2.1: A matrix H is created to split sequence alignment problem into smaller problems. Suppose we have sequences $A = a_1 a_2 ... a_n$ and $B = b_1 b_2 ... b_m$, where $0 < i \leq n$ and $0 < j \leq m$. At any position (i, j) during dynamic programming, the alignment path can come from $(i-1, j)$, $(i, j - 1)$ and $(i - 1, j - 1)$. All the candidates will be extended to contain a_i and b_j, and the similarity scores or distances will be accumulated, the one with higher similarity degree will be kept for next dynamic programming steps.

2.2 Sequence Local Alignment

Sequence local alignment aims to find the most similar segment pairs from two input sequences. Different from global alignments, the *x-length* and *y-length* of a local alignment are not fixed, so an optimal solution can be different segment pair with different similarity degrees, in order to satisfy the distinguishing requirements for applications. For instance, the Smith-Waterman algorithm always finds the segment pair with the highest similarity score, for those applications which require the optimal local alignment to be the segment pair with maximum score, the Smith-Waterman algorithm is effectively. However, if the application needs the segment pairs which are strictly identical, the similarity score of optimal solution could be much smaller than the alignment found by the Smith-Waterman algorithm, which is more tolerant for mismatches (mismatches are accepted if they do not lower down similarity score below 0).

2.3 Smith-Waterman Algorithm

In the past half-century, along with more genes and proteins sequences of different species that have been decoded, people kept trying to develop new approaches to analyze the vast amount of sequence data. Before the Smith-Waterman algorithm was proposed, it was a tough task to find the homologous segments from long sequences, because along with the evolution, mutations and variations of DNA sequences always make too much noise along with the alignment. In 1981, based on global alignment approaches, the Smith-Waterman algorithm which is a dynamic programming algorithm was developed to find a pair of segments, from two given long sequences, such that there is no other pair of segments with greater similarity score (homology)" [17]. That means the optimal local alignment cannot be extended on both sides to achieve a higher similarity score. The Smith-Waterman algorithm also generates matrix H, but at each position (i, j) in H (we denote it as $H_{i,j}$), the data stored is the global alignment score, which is maximum among all possible subsequence pair $a_g...a_i$ and $b_h...b_j$, where $0 < g \leq i$ and $0 < h \leq j$. In this thesis, when we talk about alignment starting from position (g, h) and ending at (i, j) in H, it means the optimal global alignment for subsequence pair $a_g...a_i$ and $b_h...b_j$. Notice that in the algorithm of finding global alignment, $H_{i,j}$ can be negative, but the Smith-Waterman algorithm does not keep any negative scores in H. If a local alignment contains a series of mismatches which lower down the similarity score below 0, then the alignment before or after the mismatches can achieve higher similarity score than the whole piece. Meanwhile, if $g = i$ and $h = j$, then that means there is no similar segment pair exists between input sequences, such as DNA sequences $A =$ "GGG" and $B =$ "$TTTT$", that means the alignment which is found by the Smith-Waterman algorithm can be two empty segments with similarity score 0. Meanwhile, any alignments with negative scores will not be more optimal than score 0; therefore, negative scores will not be stored in H due to these two reasons. When negative scores are obtained during dynamic programming, the Smith-Waterman algorithm will record the scores as 0; then the similarity scores are correctly represented the similar regions without being effected by mismatches before them. An example is given in **Figure** 2.2.

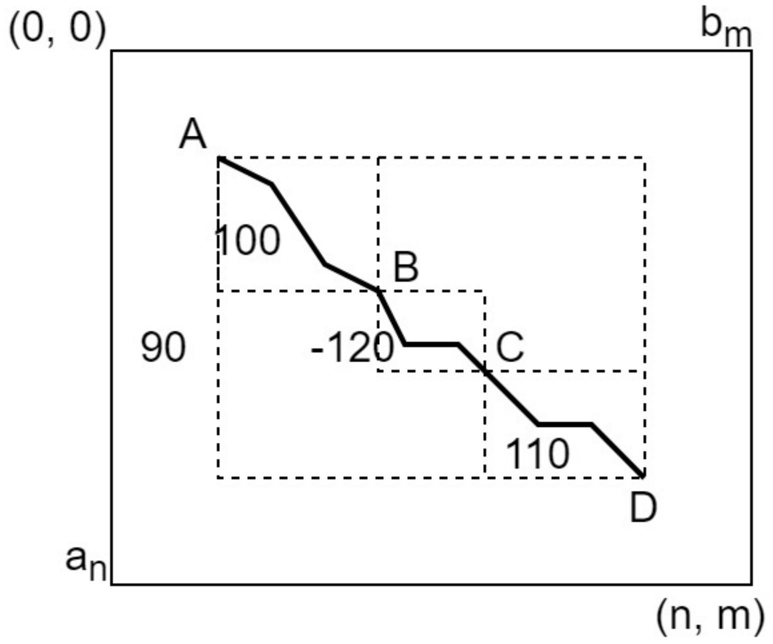


Figure 2.2: Suppose the similarity scores for regions AB, BC and CD are 100, -120 and 110 respectively. If negative scores can be stored in H, at position C the similarity score will be -20, and 90 for position D; therefore, the optimal local alignment would be region AB, since it is the highest score. However, we know region CD is a better solution with a higher score. Smith-Waterman solved this problem by setting all negative scores to 0 (position C will store value 0), then the similarity score for CD can be obtained correctly.

The Smith-Waterman algorithm firstly set up matrix H, and initialize all the values in the first row and column to 0. Then at any position (i, j) during dynamic programming, a gap penalty is added to the similarity score of the candidates from $H_{i-1,j}$ and $H_{i,j-1}$. And if the candidate is from $H_{i-1,j-1}$, the additive for similarity score will be $s(a_i, b_j)$. $H_{i,j}$ will take the maximal value among them and 0. The relationship is shown in Equation 2.1.

$$H_{i,j} = \max\begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ H_{i-1,j} + \delta, \\ H_{i,j-1} + \delta, \\ 0 \end{cases} \tag{2.1}$$

where δ is a gap penalty. The Smith-Waterman algorithm only need to store one alignment with the highest score at each $H_{i,j}$, because Smith-Waterman only measures the similarity score. As we know the score is cumulative, so that if more than one candidate are stored at any position (i, j), then the alignments with lower scores will not generate higher score alignment in the end. Moreover, an example can be found in Figure 2.3. The time complexity of Smith-Waterman is $O(mn)$ because, at each position (i, j), the comparison part consumes fixed time.

The significance of the Smith-Waterman algorithm is undoubted; however, in the following three scenarios, Smith-Waterman may provide inefficient alignment solutions due to its properties [3].

Firstly, the optimal solution can include poorly aligned segments, and it is called mosaic effect [3]. As long as the poorly aligned region cannot lower down the accumulative similarity score under 0 like shown in **Figure** 2.2, the whole piece will achieve the highest score. It is an optimal solution by the definition of Smith-Waterman, but it is not an ideal solution for all applications which may require more consistently similar alignment. The example is shown in **Figure** 2.4 [3].

Secondly, **Figure** 2.5 [3] shows the shadow effect without overlapping. There could be other alignments that have higher similarity degree, but the similarity score is lower than the solution obtained from Smith-Waterman locate in other regions. Such alignments are also biologically meaningful, due to the definition of Smith-Waterman, they will be ignored.

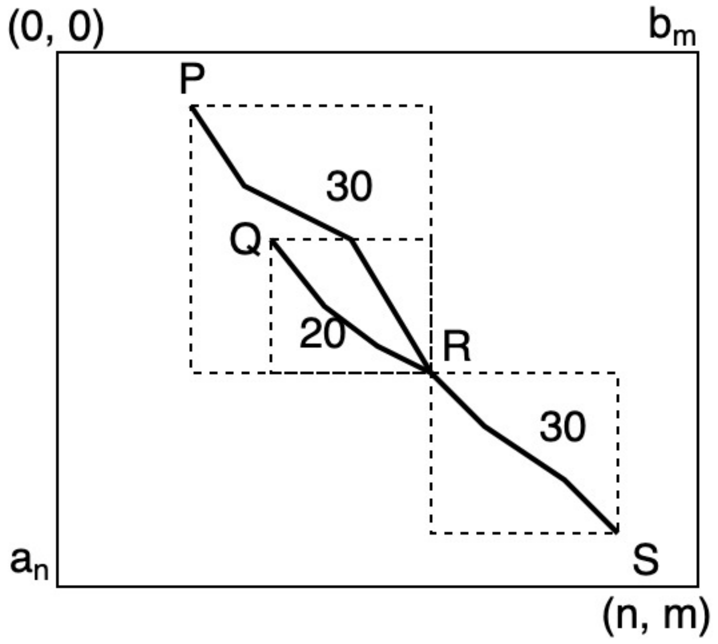


Figure 2.3: For any two sequences $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$, suppose we know there are two alignment PR and QR ending at R with similarity score 30 and 20 respectively, and the alignment RS start from R has score 30, then alignment PS always provide higher additive score than QS. In this example, the cumulative score for PS is 60, but 50 for QS. Therefore, Storing the highest score alignment at each $H_{i,j}$ is enough for Smith-Waterman to find optimal alignment.

The third problem is similar to the previous one, the alignment could take another path to an endpoint with a little low score, but much shorter, but Smith-Waterman was designed to take the path which can generate the highest similarity score. See **Figure** 2.6 [3] for an example.

Briefly, Even though Smith-Waterman is an efficient algorithm to find local alignment, it is not flexible for all applications to find corresponding ideal solutions, since the only criterion Smith-Waterman takes to measure alignments is similarity score.

2.4 Similarity Metric

Similarity metric is widely used in many areas, such as protein sequences comparison. The definition is given in [7] in 2007, and is stated below:

Definition Given a set X, a real valued function $s(a, b)$ on the Cartesian product $X \times X$ of X is

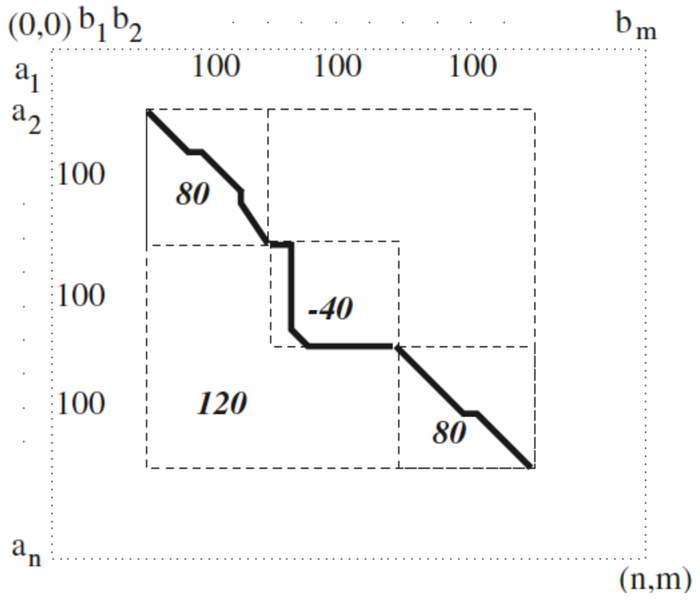


Figure 2.4: **Mosaic effect**. The local alignment which is found by the Smith-Waterman algorithm has a score 120, and the scale is 300×300. The whole alignment is composed of three segments, two with score 80 and scale 100×100, the other one with score -40 and scale 100×100 as well. For some applications, only the consistently similar regions are required, such as the two regions with score 80, so that the internal segments with score -40 is useless, but Smith-Waterman will not decompose them into three pieces.

a similarity metric if, $\forall a, b, c \in X$, it satisfies the following properties:

1. $s(a, b) = s(b, a)$
2. $s(a, a) \geq 0$
3. $s(a, a) \geq s(a, b)$
4. $s(a, b) + s(b, c) \leq s(b, b) + s(a, c)$
5. $s(a, a) = s(b, b) = s(a, b)$ if and only if $a = b$

Furthermore, we say a similarity metric $s(a, b)$ is a normalized similarity if

$$|s(a, b)| \leq 1. \tag{2.2}$$

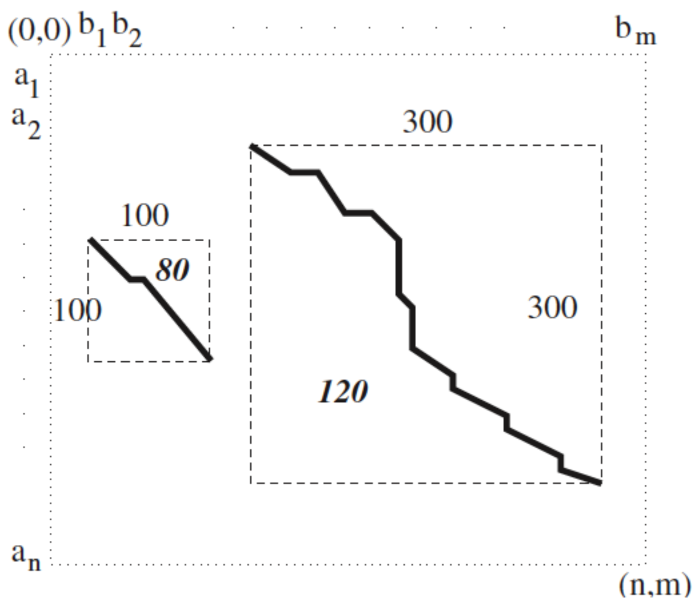


Figure 2.5: **Shadow effect without overlapping**. There is another alignment with score 80, which is lower than 120, but its scale is much smaller than the algorithm result. That means the low score alignment may have much higher similarity degree than the ”optimal” one; however, Smith-Waterman only targets to find the alignment with the highest similarity score, so another biologically meaningful region will be discarded.

2.5 Distance Metric

Let Σ be a set of finite characters including space, and (a, b) be a string pair of any finite length from Σ. Then an edit operation could be to substitute a with b, insertion or deletion. The penalty of each edit operation can be assigned by distance metric function λ, which has to satisfy the following conditions:

1. $\lambda(a, b) = \lambda(b, a)$,
2. $\lambda(a, b) \geq 0$,
3. $\lambda(a, c) \leq \lambda(a, b) + \lambda(b, c)$ (triangle inequality),
4. $\lambda(a, b) = 0$, if and only if $a = b$.

Also for a distance metric $\lambda(a, b)$ to be a normalized distance metric, $\lambda(a, b)$ must be less than or equal to 1.

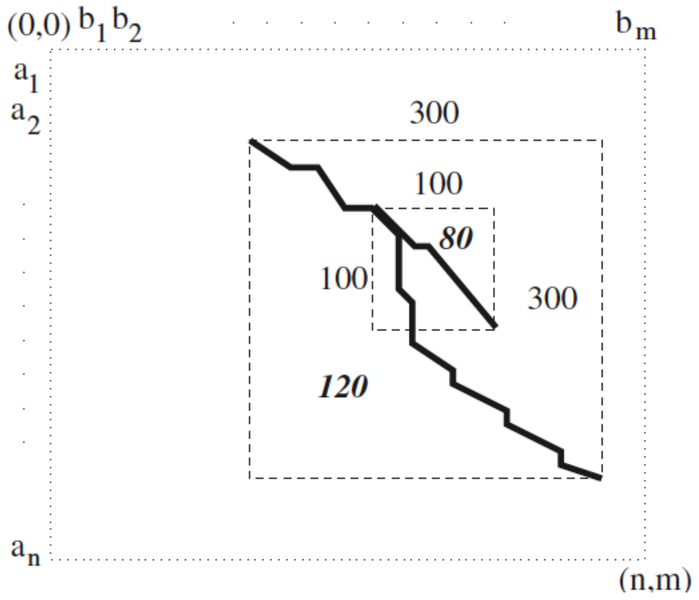


Figure 2.6: **Shadow effect with overlapping**. Smith-Waterman will take the alignment path which can obtain a higher similarity score 120, rather than a much shorter path with score 80.

From distance metric condition 4, it is obvious that if sequences X and Y are identical with infinite elements, the total distance for these two sequences will be 0 forever, by adding the distance of each element pair. In this case, if one base in X has been modified to another alphabet, then the distance of X and Y equal to the distance of the particular base pair. For example, if X_i was changed from "a" to "b", the distance of two infinite length string would be $\lambda(X_i, Y_j)$, which is $\lambda(b, a)$. Meanwhile, suppose we have another two strings M and N with length 1 for both, where M = "b" and N = "a". Then the distance for M and N will be the same as the distance of X and Y; however, over 99% base pair of X and Y are the same, and M has no relationship with N at all. To avoid this problem, the alignment length needs to be considered in order to normalize distance. The detail algorithms will be introduced in the following chapter.

2.6 Similarity and Distance Metric Normalization

Definition Distance metric $\lambda(a, b)$ is normalized distance metric if $\lambda(a, b) \leq 1$.

Definition Similarity metric $s(a, b)$ is normalized similarity metric if $|s(a, b)| \leq 1$.

The only measurement needed for global alignment is similarity score or distance because all the possible alignments must include every residue from both sequences, which means all possible alignments have the same length. Therefore, for any two fixed sequences, the optimal global alignment must be the one with the highest similarity score or smallest distance.

For finding local alignment, at each dynamic programming step, we cannot reject lower score alignment. Because at the specific point, candidates can have different starting points, so the alignment with a lower score may have much shorter lengths than another candidate with a higher similarity score. Also, since the local alignment length is flexible, then there could be a great number of similar segments combinations exist. Notice that for each found individual local alignment, it is the optimal global alignment among all candidates from the corresponding segment pair with fixed starting and ending points; however, its similarity degree may be less than another local aligned segment pair, so it has to be rejected. Briefly, both segment lengths and similarity score need to be considered, and it is much more complex than global alignment. Therefore, how to make rejection decision and find the most similarity segment pair become the most controversial issue for the local alignment problem. The very basic idea is to reject those alignments with lower similarity scores and including more residues (longer length) at the same time, but except those obvious inefficient candidates, there are still many lefts. In the past decades, people proposed many algorithms that attempted to effectively normalize similarity scores by alignment lengths for making rejection decisions, and chapter 3 will give details of different normalization approaches.

The normalized similarity and distance metrics we used may generate scores that are greater than 1. The method we used is considered as generalized normalization, the metrics functionally regulated the similarity score or distance, even though the normalized result may not satisfy the normalized metric definition.

2.7 Neighbor-joining and Phylogenetic Tree

According to the studies of different organisms, it is speculated that all creatures on the earth had one common ancestor, and the evolutionary relationships have been studied for many years. For different species, the relationships can be represented by a phylogenetic tree, which is an acyclic graph. Each leaf node of the tree stands for one particular organism, and the edges' lengths (or the weights) are the distances among species. **Figure** 2.7 is an example of phylogenetic tree, and it is observed that gibbon and orangutan, grey seal and harbor seal, opossum platypus are clustering as branches firstly, this situation can be interpreted as those three pairs of species are closely related respectively. A phylogenetic tree can be rooted or unrooted, where unrooted tree does not make any prediction for ancestor, it only shows the evolutionary distances. Since we constructed unrooted tree for experiments, all phylogenetic trees which are mentioned will be unrooted in the rest of this thesis.

Neighbor-joining is one of the most popular methods for constructing phylogenetic tree, and it was proposed in 1987 [16]. The main idea of this method is to iteratively join two particular taxa and gain a new graph that has less total distance than any other joining combination and reduce the taxa set by one. For example, in order to build a phylogenetic tree which includes n species, where $d(i, j)$ denotes the distance between any two taxa, the algorithm always try to find minimum Q in each iteration [11]:

$$Q(i, j) = (r-2)d(i, j) - \sum_{k=1}^{r} d(i, k) - \sum_{k=1}^{r} d(j, k) \tag{2.3}$$

where r is the current taxa number. After finding optimal joining combination, an internal node u will be created to join these two taxa (suppose they are f and g). Then:

$$\begin{aligned} d(f, u) &= \frac{1}{2} d(f, g) + \frac{1}{2(r-2)} [\sum_{k=1}^{r} d(f, k) - \sum_{k=1}^{r} d(g, k)] && (2.4) \\ d(u, g) &= d(f, g) - d(f, u) && (2.5) \end{aligned}$$

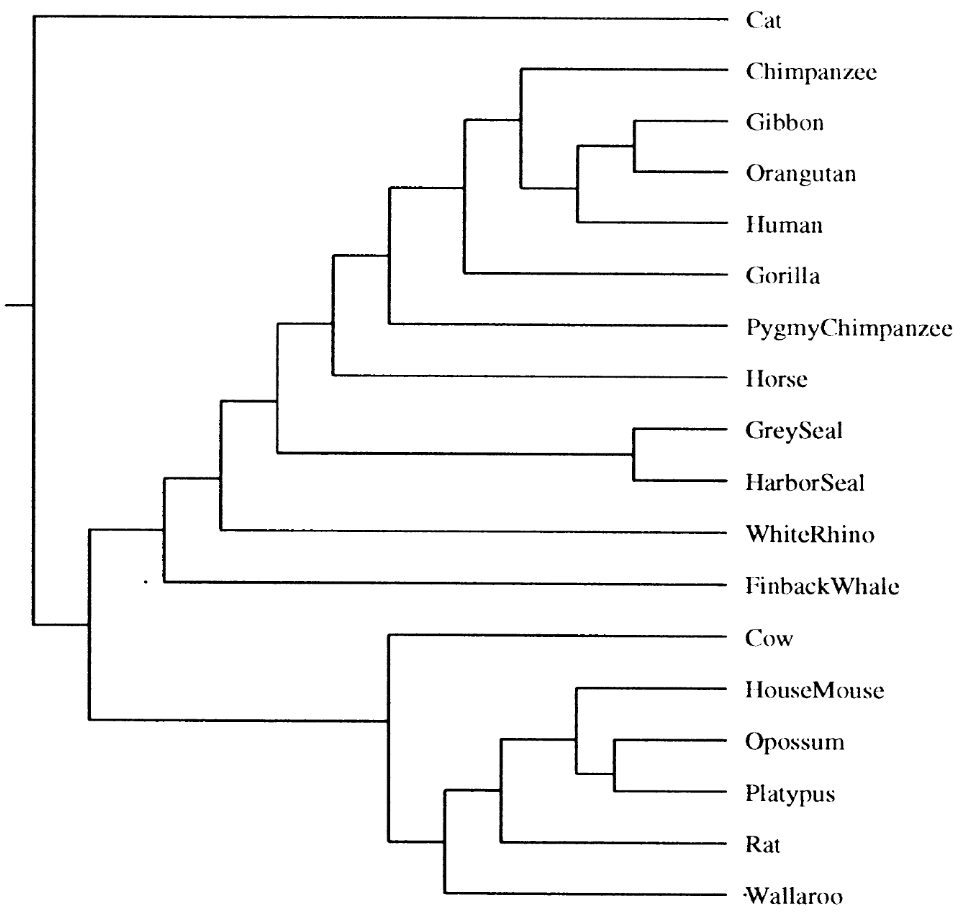
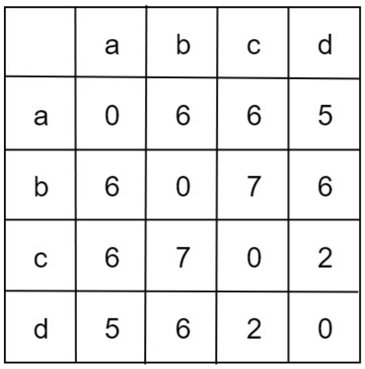


Figure 2.7: This is an example of phylogenetic tree built by complete mtDNA sequences using frequency of k-mers [13].

After joining, the distance from u to any other taxa k need be calculated for next iteration, and u will replace taxa f and g in the distance matrix.

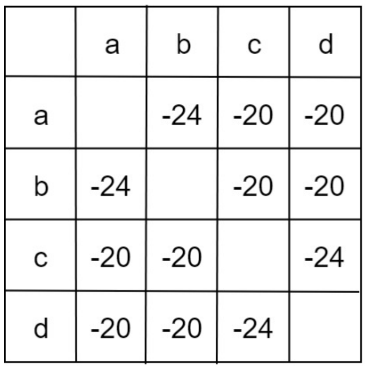
$$d(u,k) = \frac{1}{2}[d(f,k) - d(f,u)] + \frac{1}{2}[d(g,k) - d(g,u)] \tag{2.6}$$

For example, if we have four species a, b, c and d, with distances between any pair of them, then we have the distance matrix in **Figure** 2.8a. Firstly, $Q(i, j)$ will be calculated to generate **Figure** 2.8b; also since $Q(a, b)$ and $Q(c, d)$ have the same value, we can either join a, b or c, d. For this example, we join a, b, and they both connect to an internal node u. Then in the next iteration, the distances from u to other taxa will be considered, instead of all distances from a



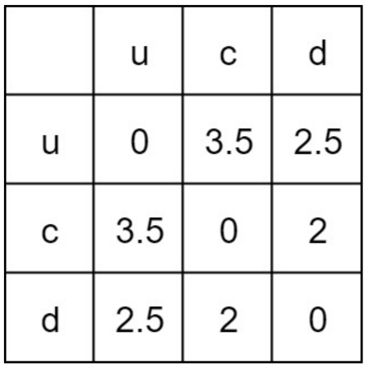
	a	b	c	d
a	0	6	6	5
b	6	0	7	6
c	6	7	0	2
d	5	6	2	0

(a) distance matrix



	a	b	c	d
a		-24	-20	-20
b	-24		-20	-20
c	-20	-20		-24
d	-20	-20	-24	

(b) Q values



	u	c	d
u	0	3.5	2.5
c	3.5	0	2
d	2.5	2	0

(c) maintained distance matrix for next iteration

Figure 2.8: (*a*) is the original distance matrix obtained from sequences comparison; (*b*) is the Q values calculated by equation 2.3 based on (*a*), for example, $Q(a, b) = (4 - 2) \times 6 - (6 + 6 + 5) - (6 + 7 + 6) = -24$; (*c*) is the distance matrix after joining, and $d(u, c)$ and $d(u, d)$ are obtained by equations 2.4, 2.5 and 2.6.

or *b*. After joining, the new distance matrix in **Figure** 2.8c needs to be generated for the next joining iteration.

2.8 Red-black Tree

A red-black tree is a binary search tree. Another attribute is added in each tree node to indicate the node color, in order to make the tree self-balancing. A red-black tree must have the following properties:

1. The color of root node must be black.
2. Each node must have color red or black.
3. Every leaf node must be black.
4. The children of a red node must be black.
5. All the paths from an arbitrary leaf node to the root must have same black-height (black-height is the number of black node in the path).

When a new node is inserted into the tree, a series of rotations must be taken to maintain the tree, in order to satisfy the above properties. Therefore, we get the following lemma [18], and the proof can be found on the page: [18, p. 309]:

Lemma 2.8.1 *A red-black tree with n internal nodes has height at most* $2\lg(n+1)$.

Chapter 3

Literature Review

3.1 Normalized Editing Distance

In the 1990s, many algorithms have been developed to fix the mosaic effect, such as the $X-$ *alignments* method, where X is a predetermined and fixed positive integer, the alignment will be considered, if the score does not drop more than X [21]. Later on, Zhang et al. tried to decompose the local alignment into sub-alignments [22]; however, if the highly aligned parts are split into different subsequences, they could be missed, since the segments may not win out in the corresponding subsequences. At the same time, normalizing distance or similarity score has been considered.

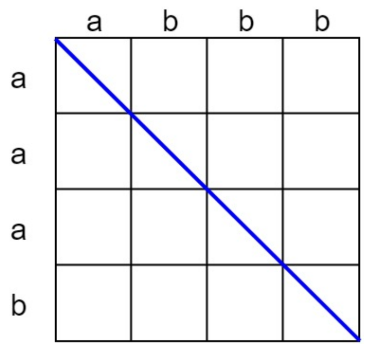
In 1993, one of the earliest normalization algorithms was published by Andres Marzal and Enrique Vidal [14]. In the paper, they provided a well-designed algorithm to find the normalized editing distance in $O(m \times n^2)$ time and $O(n^2)$ memory space, where m and n are the input sequence lengths, and $m \geq n$. Furthermore, they explained that normalized distance could not be calculated by firstly getting the minimum weight path, then use the length to normalize it.

Suppose the editing path $P = (i_0, j_0)...(i_m, j_m)$ with length $L(P) = m$, then the general formula to calculate normalized distance $\hat{W}(P)$ is:

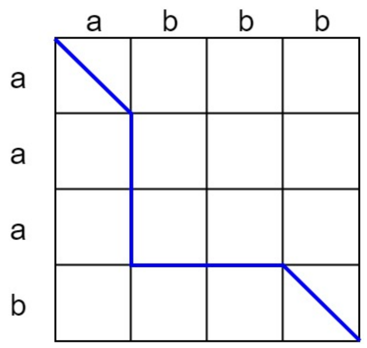
$$\hat{W}(P) = \frac{W(P)}{L(P)}, \tag{3.1}$$

W	a	b	space
a	0	3	2
b	3	0	2
space	2	2	

(a) Weighting function



(b) $\hat{W}(P) = \frac{6}{4} = 1.5$



(c) $\hat{W}(P) = \frac{8}{6} = 1.33$

Figure 3.1: (*a*) shows the weight function of each pair. (*b*) shows the result of post-normailzation. The path with minimum weight has been obtained firstly with value 6, and $\hat{W}(P)$ is equal to 1.5 since $L(P)$ is 4; however, it is not the path with actual minimum normalized distance. The ideal path is shown in (*c*), with normalized distance 1.33 [14].

where $W(P)$ is the weight of path P. Therefore, the normalized editing distance between string X and Y is:

$$d(X, Y) = min\{ \hat{W}(P) \mid P \text{ } is \text{ } an \text{ } editing \text{ } path \text{ } between \text{ } X \text{ } and \text{ } Y\} \tag{3.2}$$

The minimization step is to compare the normalized weights of paths with different lengths and find the minimum one, and it has been proved that the minimization step cannot be carried out before normalization. The following example can show the reason.

Given string $X = abbb$, $Y = aaab$, and the weight function is shown in **Figure** 3.1a. If the minimization step is carried out after the dynamic programming, indeed the resulting path is the one with minimum weight, but since the path lengths are not the same, the normalized weight may not be the minimum as well. The path shown in **Figure** 3.1c is the path with minimum normalized distance. Also, theoretically, the post-normalization method does not satisfy condition 3 (triangular inequality) of distance metric, which was given in chapter 2. For example, if the weight function in **Table** 3.1 is used, and $X = a$, $Y = ab$ and $Z = b$. The $\hat{W}(P)$ for X and Y is $\frac{1}{2}$, and $\frac{5}{2}$ for Y and Z; however, The $\hat{W}(P)$ for X and Z is $\frac{5}{1}$. Now we have $\frac{1}{2} + \frac{5}{2} \ngeq \frac{5}{1}$, it means $D(X, Y) + D(Y, Z) < D(X, Z)$ [14]. Actually, the $\hat{W}(P)$ for X and Z should be $\frac{6}{2}$, then the inequality hold again.

W	a	b	$space$
a	0	5	5
b	5	0	1
$space$	5	1	

Table 3.1: The weight function which makes post-normalization fail triangular inequality.

In order to get $d(X, Y)$, the naive way would be to list all the paths between X and Y, then calculate the normalized weight for each of them; however, it is too expensive to do so. Suppose $X = X_1X_2...X_n$, $Y = Y_1Y_2...Y_m$, and $n \geq m$, then the length of editing path between X and Y should be in the range $[n, m + n]$. Hence, during dynamic programming, at any position (X_i, Y_j), where $i \leq j$ there will be $i + 1$ paths with different lengths are calculated and saved. For each length, the calculation is shown in the following **Theorem** [14]:

Theorem 3.1.1 *$\forall i, j, 1 \leq i \leq |X|, 1 \leq j \leq |Y|, \forall k, max(i, j) \leq k \leq i + j$, let $\gamma(a \rightarrow b)$ be the editing operation weight function to transfer a to b.*

$$D(i, j, k) = min\{D(i-1, j, k-1)+\gamma(X_i \rightarrow \lambda), D(i, j-1, k-1)+\gamma(\lambda \rightarrow Y_j), D(i-1, j-1, k-1)+\gamma(X_i \rightarrow Y_j)\}$$

$$D(i, j, k) = \infty, \forall k \leq max(i, j), \forall k \geq i + j$$

From **Theorem** 3.1.1, it is known that the time complexity of calculating $D(i, j, k)$ is $O(1)$. So that the time complexity of getting all lengths path at position (X_i, Y_j) is $O(N)$, which is $O(m)$ since $m \leq n$ here. Eventually, to finish the whole dynamic programming, the time complexity would be $O(nm^2)$. The detail is shown in **Algorithm** 1 [14].

Algorithm 1 Normalized editing distance

```
    Input: $X = X_1X_2...X_n$, $Y = Y_1Y_2...Y_m$, weight function $\gamma(a \to b)$;
    Output: $d(X, Y)$
 1: $int\ i, j, k$
 2: $D[|X|, |Y|, |X| + |Y| + 1]$ is a 3D array
 3: $D[0, 0, 0] = 0$
 4: $D[0, 0, 1] = \infty$
 5: for $j = 1 \to |Y|$ do
 6:     $D[0, j, j - 1] = \infty$
 7:     $D[0, j, j] = D[0, j - 1, j - 1] + \gamma(\lambda \to Y_j)$
 8:     $D[0, j, j + 1] = \infty$
 9: end for
10: for $i = 1 \to |X|$ do
11:     $D[i, 0, i - 1] = \infty$
12:     $D[i, 0, i] = D[i - 1, 0, i - 1] + \gamma(X_i \to \lambda)$
13:     $D[i, 0, i + 1] = \infty$
14:     for $j = 1 \to |Y|$ do
15:         $D[i, j, max(i, j) - 1] = \infty$
16:         for $k = max(i, j)$ to $i + j$ do
17:             $D[i, j, k] = min(D[i - 1, j, k - 1] + \gamma(X_I \to \lambda), D[i, j - 1, k - 1] + \gamma(y_j \to \lambda), D[i - 1, j - 1, k - 1] + \gamma(X_i \to Y_j))$
18:         end for
19:         $D[i, j, i + j + 1] = \infty$
20:     end for
21: end for
```

3.2 An improvement of Normalized Editing Distance

As mentioned, the time complexity of the previous algorithm is $O(mn^2)$, if m and n are the lengths of sequence X and Y, respectively, and $m \geq n$. In 1999, Abdullah N. Arslan and Omer Egecioglu [2] proposed an improved algorithm that requires $O(mn \log n)$ time, if the cost of the same type of editing operation is uniform. Instead of calculating normalized weight along with the dynamic programming, this algorithm only solves ordinary editing distance problem at most $\log n$ times.

A graph could be used to describe the ordinary problem, just like 3.2. An editing path of string X and Y is from vertices $(0, 0)$ to (m, n), and can go to three directions corresponding to three different operations which are deletion (if goings horizontally), insertion (if going vertically) and substitution (if going diagonally). Also, the cost function is $\gamma = (\gamma_I, \gamma_D, \gamma_M, \gamma_N)$, which γ_I is the cost for insertion, γ_D for deletion, γ_M for matching substitution and γ_N for non-matching substitution. The assumption is that all of them are constant.

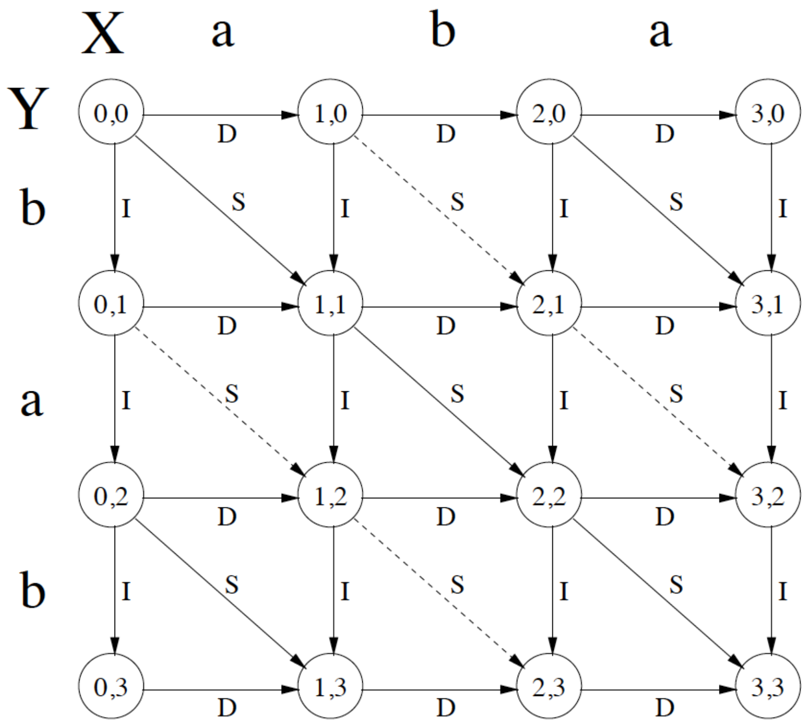


Figure 3.2: The editing graph G_X, Y for the strings $X = aba$ and $Y = bab$. [2]

Let $W_\gamma(p)$ denote the weight of editing path p, and $h(p)$ be the horizontal move number of path p, $v(p)$ be the vertical move number, $d_N(p)$ is the non-matching diagonal move number and $d_M(p)$ be the matching diagonal move number. Then:

$$W_\gamma(p) = \gamma_D h(p) + \gamma_I v(p) + \gamma_M d_M(p) + \gamma_N d_N(p), \tag{3.3}$$

The length of path p is:

$$L(p) = h(p) + v(p) + d_M(p) + d_N(p), \tag{3.4}$$

We also have:

$$m = h(p) + d_M(p) + d_N(p) \tag{3.5}$$

$$n = v(p) + d_M(p) + d_N(p) \tag{3.6}$$

Therefore, $W_\gamma(p)$ and $L(p)$ can be transformed to:

$$W_\gamma(p) = \gamma_D + n\gamma_I + (\gamma_M - \gamma_I - \gamma_D)d_M(p) + (\gamma_N - \gamma_I - \gamma_D)d_N(p) \tag{3.7}$$

$$L(p) = m + n - d_M(p) - d_N(p) \tag{3.8}$$

Recall the normalized editing distance is:

$$NED_{x,y,\gamma} = \min_{p \in P} \frac{W_\gamma(p)}{L(p)}, \tag{3.9}$$

So it can be transformed to:

$$NED_{x,y,\gamma} = \min_{p \in P} \frac{\gamma_D + n\gamma_I + (\gamma_M - \gamma_I - \gamma_D)d_M(p) + (\gamma_N - \gamma_I - \gamma_D)d_N(p)}{m + n - d_M(p) - d_N(p)}. \tag{3.10}$$

From the above **Equation** 3.10, we can see finding NED becomes to optimize the ratio of two linear functions. In order to solve this problem, Dinkelbach's algorithm [9] are used. The basic idea of this algorithm is fractional programming. The optimal solution of equation 3.10 can be achieved by solving **Equation** 3.11, where $\lambda \in R$. λ^* will be the optimal solution of **Equation** 3.10, if and only if the optimal vaule of $f(\lambda^*)$ is zero.

$$f(\lambda) = \min[W_\gamma(p) - \lambda L(p)]. \tag{3.11}$$

Noticed that, if we substitute $W_\gamma(p)$ and $L(p)$ with **Equation** 3.7 and 3.8, then 3.11 can be simplified to a new editing path weight function with new weights, and the variables are $d_M(p)$ and $d_N(p)$. Instead of calculating NED through the dynamic programming, we just need to try different λ by finite times to find the optimal value; therefore, the time consuming of this algorithm should be $O(kmn)$, where k is the number of trials. As we know, $d_M(p)+d_N(p)$ cannot be greater than the length of the shorter sequence, which is n here; therefore, there should be $n \times n$ combinations of $d_M(p)$ and $d_N(p)$, which corresponding to n^2 candidates of λ. To make it simple, that n^2 values will be calculated in advance. Then the median candidate (let's say $\hat{\lambda}$) value can be picked for first trial, after finding the minimum weight editing path, if $f(\hat{\lambda}))$ is greater than 0, that means the λ^* should be greater than $\hat{\lambda}$), then all the candidates smaller than $\hat{\lambda}$) can be ignored, vice versa, until it reaches zero. In each iteration, the range will be divided into half, then the worst case would be $\log n^2$ iterations. In other words, k is $O(\log n)$. Combining the time of building possible λ set, the time complexity of the whole algorithm is $O(n^2 + mn \log n)$, which is $O(mn \log n)$. The main steps of this algorithm is shown in **Algorithm** 2 [2].

Algorithm 2 NED algorithm for uniform weights

```
Input: $X = X_1X_2...X_n$, $Y = Y_1Y_2...Y_m$, weight function
Output: $\lambda^*$
1: if $m = n = 0$ then
2:     return $-1$
3: end if
4: if $m = 0$ then
5:     reuturn $\gamma_I$
6: end if
7: if $n = 0$ then
8:     return $\gamma_D$
9: end if
10: Generate the set $Q$ of $\lambda$
11: while true do
12:     Find the median $\lambda_{med}$ of $Q$
13:     Solve $E\ D_{X,Y,\gamma}(\lambda_{med})$
14:     if the minimum path weight is 0 then
15:         return $\lambda_{med}$
16:     else
17:         if the minimum path weight is smaller than 0 then
18:             remove the values equal and larger than$\lambda_{med}$
19:         else
20:             remove the values equal and smaller than$\lambda_{med}$
21:         end if
22:     end if
23: end while
```

3.3 Normalized Local Similarity Score

In order to find similar segments between DNA sequences of different species, local alignment algorithm must be used. In 2000, [3] extended the idea of uniform weight normalized editing distance algorithm [2], which was just discussed above, to normalize the similarity score. Suppose we have string X and Y, I and J are the substrings for X and Y respectively. The basic idea is to find the maximum value of $s(I, J)/(|I| + |J|)$, where $|I| + |J| \geq T$ and T is the threshold for alignment length. However, those alignments with high similarity degree but very short length will be obtained by this formula, the result is biologically meaningless. To fix this problem, they add a constant number L to the denominator. Then *fractional programming* will be used to find the optimal alignment.

As mentioned in the last section, an alignment can be considered as a graph, and the similarity score can be calculated by three values: number of matches, mismatches and indels (insert or deletion). Suppose the score for a match is 1, δ for a mismatch and μ for indel. Therefore, vector (x, y, z) can be used to represent a local alignment with x matches, y mismatches and z indels. Then the similarity score is:

$$SCORE(x, y, z) = x - \delta y - \mu z \tag{3.12}$$

The best alignment between substrings $a_i...a_k$ and $b_j...b_k$ would be the vector with maximum score among all the alignment vectors between these two strings. Furthermore, the optimal local alignment of strings a and b is to seek for two substrings $a_i...a_k$ and $b_j...b_l$, with the highest similarity score, let LA^* be the score of optimal local alignment, then:

$$LA^*(a, b) = max\{SCORE(x, y, z) \mid (x, y, z)\ is\ any\ alignment\ vector\ of\ a_i...a_k\ and\ b_j...b_l\} \tag{3.13}$$

The length of an alignment between $a_i...a_k$ and $b_j...b_l$ is $(k - i + 1) + (l - j + 1) + L =$

$2x + 2y + z + L$, where L is the constant to control the optimal alignment length:

$$LENGTH(x, y, z) = 2x + 2y + z + L \tag{3.14}$$

Therefore, the optimal normalized similarity score should be:

$$\begin{aligned} NLA^*(a, b) &= max\{\frac{SCORE(x, y, z)}{LENGTH(x, y, z)}\} && (3.15) \\ &= maximize \frac{x - \delta y - \mu z}{2x + 2y + z + L} && (3.16) \end{aligned}$$

Recall the *parametric method*, $NLA*(a, b)$ can be transferred to another ordinary similarity problem:

$$LA(\lambda)(a, b) = maximize\ x - \delta y - \mu z - \lambda(2x + 2y + z + L) \tag{3.17}$$

Then the Smith-Waterman and Dinkelbach's algorithm will do the rest work until find the optimal local alignment, just like the algorithm introduced in section 3.2. Later on, the author proved that the time complexity could be better if using Megiddo's algorithm [2].

3.4 Similarity Metric and Normalized Similarity Metric

Recall the definition of similarity metric was given in [7] is:

Definition Given a set X, a real valued function $s(x, y)$ on the Cartesian product $X \times X$ of X is a similarity metric if, $\forall x, y, z \in X$, it satisfies the following properties:

1. $s(x, y) = s(y, x)$
2. $s(x, x) \geq 0$
3. $s(x, x) \geq s(x, y)$

4. $s(x, y) + s(y, z) \leq s(y, y) + s(x, z)$

5. $s(x, x) = s(y, y) = s(x, y)$ if and only if $x = y$

Although similarity measure is used in many fields such as protein sequence comparison, there was no formal concept had been proposed until 2007 [7].

The first three conditions are intuitive, and condition four is similar to triangle inequality of distance metric. It states that the similarity between two elements through the third one is less than the actual similarity of these two elements plus the similarity of the third one comparing to itself. In addition, in order to understand condition 5, suppose we have $s(x, y)$ that satisfies the first 4 conditions of similarity metric, and $s(x, x) = s(y, y) = s(x, y)$. For any z, by condition 4, we have $s(x, y) + s(y, z) \leq s(x, z) + s(y, y)$. Since $s(x, y) = s(y, y)$, we can get $s(y, z) \leq s(x, z)$. On the other hand, we also can have $s(y, x) + s(x, z) \leq s(y, z) + s(x, x)$, which can be simplified to $s(x, z) \leq s(y, z)$. Therefore, we have $s(x, z) = s(y, z)$, which means x can be treated as y.

The definition of normalized similarity metric is:

Definition A similarity metric $s(x, y)$ is a **normalized similarity metric** if $|s(x, y)| \leq 1$.

In [8], the authors also pointed out the method, which had been introduced in the previous section ($\frac{s(a,b)}{|a|+|b|+L}$, where L is a constant and $L > 0$), is not a similarity metric. It can be easily proved by a counter example. Suppose we have sequences $X =$ "*abc*", $Y =$ "*abcde*" and $Z =$ "*cde*", for any character pair i and j, we have scoring function $s(i, i) = s(j, j) = 2$, $s(i, j) = -1$ and 0 for any indels. We can see $s(i, j)$ is a similarity metric, $s(X, Y) = s(Y, Z) = 6$, $s(Y, Y) = 12$ and $s(X, Z) = 2$. We also have $|X| = |Z| = 3$, $|Y| = 5$, and suppose $L = 1$. If we used $\frac{s(a,b)}{|a|+|b|+L}$ to normalize the scores, then we have $\frac{s(X,Y)}{|X|+|Y|+L} + \frac{s(Y,Z)}{|Y|+|Z|+L} = \frac{6}{9} + \frac{6}{9}$, and $\frac{s(X,Z)}{|X|+|Z|+L} + \frac{s(Y,Y)}{|Y|+|Y|+L} = \frac{2}{7} + \frac{10}{11}$. It is clearly that $\frac{s(X,Y)}{|X|+|Y|+L} + \frac{s(Y,Z)}{|Y|+|Z|+L}$ is larger than $\frac{s(X,Z)}{|X|+|Z|+L} + \frac{s(Y,Y)}{|Y|+|Y|+L}$, so condition 4 does not hold under this situation.

Based on the distance and similarity metric definitions, there are some important lemmas and theorems, which are highly relevant to our new algorithm, were proposed in [20]. The detail will be given in the following, with the definition of concave function. The proofs can be found in [20].

Definition A function f is concave over an interval $[a, b]$ if for every $x_1, x_2 \in [a, b]$ and $0 \leq \lambda \leq 1$,

$$\lambda f(x_1) + (1 - \lambda) f(x_2) \leq f(\lambda x_1 + (1 - \lambda) x_2) \tag{3.18}$$

Lemma 3.4.1 *if a function f is concave over interval $(-\infty, \infty)$, then for any a, $b \geq 0$ and $c \geq 0$,*

$$f(a) + f(a + b + c) \leq f(a + b) + f(a + c). \tag{3.19}$$

Lemma 3.4.2 *Let f be a non-negative concave function on domain $[0, \infty)$, and $0 \leq x \leq y$, $b \geq 0$, then $\frac{x}{f(x+b)} \leq \frac{y}{f(y+b)}$.*

The paper [20] also defined Minkowski type similarity and distance metric for $p \geq 1$.

Theorem 3.4.3 *If $s(x, y)$ is similarity metric, and $p \geq 1$, then:*

$$d(x, y) = \sqrt[p]{(s(x, x) - s(x, y))^p + (s(y, y) - s(x, y))^p} \tag{3.20}$$

is a distance metric.

Lemma 3.4.4 *If $s(x, y)$ and $d(x, y)$ are similarity and distance metric respectively, also satisfying normalization condition, and $f(x)$ is a monotone increasing concave function on $[0, \infty)$, also $f(x) > 0$. Then:*

$$\frac{s(x, y) + s(y, z) - s(y, y)}{f(d(x, y) + s(s, y) + d(y, z) + s(y, z) - s(y, y))} \leq \frac{s(x, z)}{f(d(x, z) + s(x, z))} \tag{3.21}$$

Theorem 3.4.5 *If $s(x, y)$ and $d(x, y)$ are similarity and distance metric respectively, also satisfying normalization condition, and $f(x)$ is a monotone increasing concave function on $[0, \infty)$, also $f(x) > 0$. Then:*

$$\bar{s}(x,y) = \frac{s(x,y)}{f(d(x,y)+s(x,y))} \tag{3.22}$$

is a similarity metric.

In order to find sequence local alignment, a proper similarity metric is required. Smith-Waterman applied additive similarity metric, so it may not provide ideal solution in some scenarios. The metric shown in **Theorem** 3.4.5 considers segment length when finding optimal solution, and it invokes concave function to control the similarity degree of solution; therefore, we consider the following similarity metric which satisfies **Theorem** 3.4.3 and 3.4.5 is proper for finding normalized sequence local alignment.

$$\bar{s}(x,y) = \frac{s(x,y)}{f(\sqrt[p]{(s(x,x)-s(x,y))^p + (s(y,y)-s(x,y))^p} + s(x,y))}, \tag{3.23}$$

Where $f(x)$ is a monotone increasing concave function on $[0,\infty)$, also $f(x) > 0$.

Being different from other normalization functions, **Equation** 3.23 quantifies both similar and dissimilar regions, rather than directly normalize the similarity score by the summation of two segment lengths. It can be easily understood by using set theory. We treat $s(x,y)$ as the common part of sequences x and y, it is shape b in Figure 3.3. Also, we can see shape a and c are the non-common parts for sequences x and y respectively, so $s(x,x) - s(x,y)$ represent shape a and $s(y,y) - s(x,y)$ for shape c, then $d(x,y)$ can be interpreted as $a + c$. Therefore, we use $a + b + c$ to represent the union of sequences x and y, which is $d(x,y) + s(x,y)$. In addition, different value of k can be chosen to calculate $d(x,y)$ for specific applications.

In addition, to normalize the similarity score properly and smoothly, a concave function must be applied to control the normalization strength, otherwise only length one and identical segments will be obtained, which are biologically meaningless. Due to the property of concave function, the normalized strength will be high when the alignment length is short, then keep reducing smoothly along with the alignment, which means we encourage relatively long alignments which are more biologically meaningful, but not too long as the output of Smith-Waterman.

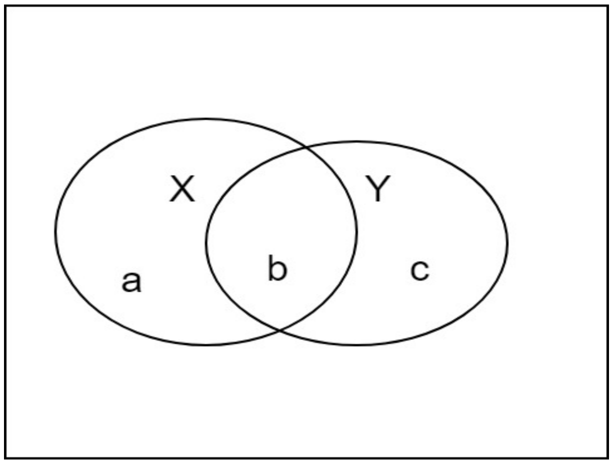


Figure 3.3: $a + c$ is the distance of X and Y, and b is the similarity.

Figure 3.4 shows an example of concave functions, where $f(x) = x + L_1$ is a special case of a concave function. When alignment is short, L_1 will be relatively large, so it effectively avoids obtaining very short alignments. However, notice that this function does not regulate variable x, so when the alignment gets longer, normalization strength still increases too fast to encourage longer alignment. On the other hand, $g(x) = x^{1/k} + L_2$ works better, because its normalization strength is relative with the alignment length. Longer the alignment length is, slower the strength increases. Furthermore, if a concave function converges too fast, the denominator of normalization function will be like a constant; therefore, the solution will be close to, or even the same as the solution of Smith-Waterman Algorithm.

Let X and Y be two sequences, then there are a great number of arrangement patterns for them with different similarity scores. Let $A(X, Y)$ denotes the additive similarity score for an alignment, then among all $A(X, Y)$, the highest one is the score of optimal global alignment for X and Y, and denoted as $s(X, Y)$. Specifically,

$$s(X, Y) = \max_{all\ A} A(X, Y). \tag{3.24}$$

For each particular global alignment with score $A(X, Y)$, there is corresponding normalized similarity score $\overline{A}(X, Y)$ exist:

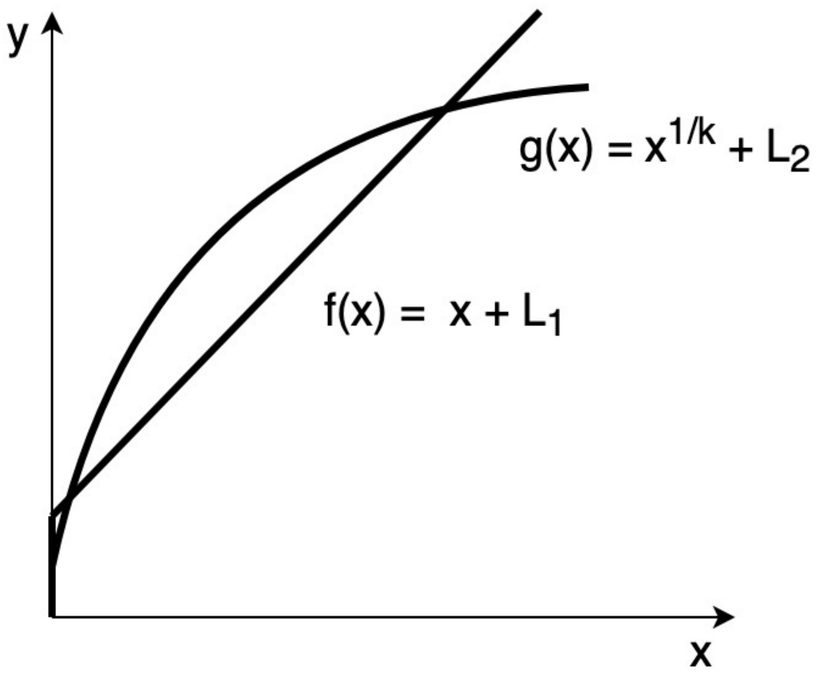


Figure 3.4: Example of concave functions. L_1, L_2 and k are constants, where $L > 0$ and $k \geq 1$.

$$\overline{A}(X,Y) = \frac{A(X,X)}{f(\sqrt[p]{(s(X,X) - A(X,Y))^p + (s(Y,Y) - A(X,Y))^p} + A(X,Y))}, \tag{3.25}$$

where $s(X,X)$ and $s(Y,Y)$ are the optimal global alignment scores for aligning X and Y to themselves. Then we have:

$$\overline{s}(X,Y) = \max_{allA} \overline{A}(X,Y). \tag{3.26}$$

Therefore, the normalized similarity metric shown in 3.23 has following lemma:

Lemma 3.4.6 *For any finite sequences X and Y, let $s(X,Y)$ denotes the additive similarity score of optimal global alignment, and $\overline{s}(X,Y)$ denotes the normalized similarity score of optimal alignment by applying normalized similarity metric showing in **Equation** 3.23, then the alignment generating $s(X,Y)$ will always generate $\overline{s}(X,Y)$ as well.*

Proof Suppose there are two global alignment patterns with similarity score $s_1(X,Y)$ and

$s_2(X, Y)$ respectively, where $s_1(X, Y) \leq s_2(X, Y)$, then by **Lemma** 3.4.2, we have:

$$\frac{s_1(X, Y)}{f(b + s_1(X, Y))} \leq \frac{s_2(X, Y)}{f(b + s_2(X, Y))}, \tag{3.27}$$

where $b \geq 0$. If $0 < c \leq b$, then

$$\frac{s_2(X, Y)}{f(b + s_2(X, Y))} \leq \frac{s_2(X, Y)}{f(c + s_2(X, Y))}, \tag{3.28}$$

since concave function $f(x)$ is monotone increasing. So

$$\frac{s_1(X, Y)}{f(b + s_1(X, Y))} \leq \frac{s_2(X, Y)}{f(c + s_2(X, Y))}. \tag{3.29}$$

Also since $s_1(X, X)$ and $s_2(Y, Y)$ are fixed, then:

$$\sqrt[p]{(s_1(X, X) - s_1(X, Y))^p + (s_1(Y, Y) - s_1(x, y))^p} \leq \sqrt[p]{(s_2(X, X) - s(X, Y))^p + (s(Y, Y) - s(X, Y))^p} \tag{3.30}$$

Let $b = \sqrt[p]{(s_1(X, X) - s_1(X, Y))^p + (s_1(Y, Y) - s_1(x, y))^p}$, $c = \sqrt[p]{(s_2(X, X) - s(X, Y))^p + (s(Y, Y) - s(X, Y))^p}$, then from **Equation** 3.29 we have

$$\overline{s_1}(X, Y) \leq \overline{s_2}(X, Y). \tag{3.31}$$

Therefore, higher $s(x, y)$ generate higher $\overline{s}(x, y)$. ∎

In **Lemma** 3.4.6, $s(x, y)$ does not have to be additive similarity metric, but the algorithm which we will propose in next chapter applied additive similarity metric, because the idea is extended from Smith-Waterman algorithm, which invokes additive metric.

3.5 BLOSUM

The protein sequence alignments are usually involved in order to study gene and protein function. No matter for global, local, or multiple sequence alignments, a scoring scheme must be invoked to measure the similarity degree. Before 1992, there are several scoring schemes have been proposed, and the most popular one is the mutation data matrices which were proposed by Dayhoff in 1968 [6]. In his model, the amino acid substitution rates are generated from protein sequences, which are aligned, and the similarity degree is above 85%. However, most tasks need to identify those distantly related segments by inferred from Dayhoff's model which is derived from high similarity protein sequences. Therefore, BLOSUM was proposed to use different protein sequence alignments groups which have particular lower similarity degree within specified sequence blocks.

Until now, BLOSUM metric series is still one of the most common methods to measure the similarity of any amino acid pairs. The matrices are proposed by Steven Henikoff and Jorja G. Henikoff in 1992 [12], and the content of BLOSUM matrices are the frequencies of corresponding amino acids are substituted by other amino acids, just like Dayhoff's model.

Firstly, from a group of related proteins, a set of blocks will be found, and a system called PROTOMAT finishes this process. Each block is the most common region for the particular protein family. For example, if we have 5 protein sequences from the same family, and the block length is 3 amino acids, then the block size is 5×3. For each column, all the matches and mismatches for the corresponding amino acid pair will be counted. For example, if there are 4 A, and only one B in the first column, then AA appears $3 + 2 + 1 = 6$ times, 4 for AB or BA, and 0 times for BB. The calculation for each column is summed up for a observed frequency table.

After counting, the probability of each amino acid pair will be calculated based on the table. For the same example, the 5×3 block can generate $3 \times 5 \times (5 - 1)/2 = 30$ pairs, and since AA appears 6 times, then the observed probability of (A, A) is $Pr(A, A) = 6/30 = 0.2$, and $P_(A, B) = Pr(B, A) = 4/30 = 0.13$.

$$Pr(i, j) = \frac{f_{ij}}{\sum_{i=2}^{20} \sum_{j=1}^{i} f_{ij}}. \tag{3.32}$$

Then, the expected probability that i appears in any pair is:

$$Pr(i) = Pr(i, i) + \sum_{j \neq i} Pr(i, j)/2. \tag{3.33}$$

In the example, the expected probability of A appears in a pair is $0.2 + 0.13/2 = 0.265$, and $0.13/2 = 0.065$ for B. And then, the expected probability for any pair (i, j) is:

$$\begin{aligned} e_{ij} &= Pr(i) \times Pr(i),\ if\ i = j & (3.34)\\ &= Pr(i) \times Pr(j) + Pr(j) \times Pr(i),\ if\ i \neq j. & (3.35) \end{aligned}$$

Eventually, the odds ratio can be got by:

$$s_{ij} = \log_2(Pr(i, j)/e_i j). \tag{3.36}$$

If $s_{ij} > 0$, it means the pair appears more than expected, it will be multiplied by scaling factor 2. Generally, the result will be rounded to the nearest integer, and it is how BLOSUM matrices are generated.

Furthermore, in order to reduce the frequency contribution from those protein sequences which are too closely related, such sequences will be clustered within blocks and contributed as one single sequence. There are different standards for clustering, such as BLOSUM62, it means that if sequences are identical more than 62% of their aligned positions, they will be clustered together. Without rounding, any BLOSUM-N matrix with $N \geq 55$ is a similarity metric [20].

Recall it is proved in [20] that:

$$\overline{s}(x, y) = \frac{s(x, y)}{f(\sqrt[p]{(s(x, x) - s(x, y))^p + (s(y, y) - s(x, y))^p} + s(x, y))}, \tag{3.37}$$

is a similarity metric. Therefore, we used this method to normalize the similarity score, where $s(x, y)$ is based on BLOSUM62. The reason we used the concave function is its slope function is monotone decreasing, so that along with the editing path get longer, the normalization strength gets weaker, which means we can control the output alignments length by adjusting the concave function. The specific experiment result with different concave functions will be shown in chapter 5.

Chapter 4

A New Algorithm for Normalized Sequence Local Alignment

Sequence local alignment is to find similar segments from input sequences, but ”similar” can be defined in different ways. For example, Smith-Waterman defines the optimal local alignment to be the one with the highest similarity score. Even though it is widely used in bioinformatics studies, it is not suitable for those applications which require consistently similar local alignment. In order to obtain segments with high similarity degree, normalization functions need to be invoked to take both similarity score and alignment length into account. Therefore, multiple normalization functions need to be applied on the same sequences for different similarity degree requirements, and it will consume a significant amount of time.

To solve these issues, we invoked the normalized similarity metric family [20] which are constructed by Minkowski type distance metric to satisfy different requirements for similarity degree. Then, a dynamic programming algorithm has been designed to find sequence local alignment. As introduced in chapter 2, algorithms which target finding sequence alignment will generate a matrix H. Our algorithm also produces matrix H, but the data stored in each $H_{i,j}$ includes all alignment candidates which pass through (i, j) and have the possibility to generate optimal solution. Therefore, no matter how many normalization functions from our similarity metric family are applied, the corresponding optimal solution can be obtained by iterating each $H_{i,j}$ and applying normalization functions to process all alignment candidates.

We define the optimal local alignment to be the segment pair with the highest normalized similarity score. Specifically, for any two sequences $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$, we look for subsequence pair $A' = a_g...a_i$ and $B' = b_h..b_j$, where $0 < g \leq i \leq n$ and $0 < h \leq j \leq m$, that has the highest normalized similarity score among all possible subsequence pairs of A and B. Recall the similarity metric we used is:

$$\overline{s}(x, y) = \frac{s(x, y)}{f(\sqrt[p]{(s(x, x) - s(x, y))^p + (s(y, y) - s(x, y))^p} + s(x, y))}, \tag{4.1}$$

where $s(x, y)$ is additive similarity metric, f is any non-negative concave functions monotone increase on $[0, \infty)$, p is a constant for Minkowski distance, which is always greater than 1. This metric takes both similar and dissimilar regions into account to normalize similarity score.

4.1 A Simple Algorithm for finding Normalized Sequence Local Alignment

A simple algorithm to find normalized sequence local alignment can be obtained by extending the algorithm for finding global alignment.

As introduced in chapter 2, for any given sequences $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$, similarity metric $s(x, y)$, the algorithm for finding sequence global alignment generates a matrix H during dynamic programming, each position $H_{i,j}$ of H contains the optimal global alignment score for subsequence pair starting from $(1, 1)$ and ending at (i, j), where $0 < i \leq n$ and $0 < j \leq m$. Then, not only the optimal global alignment for A and B is obtained, but also we can have all optimal global alignment starting from $(1, 1)$ to every possible (i, j). Then for any (g, h), where $0 < g \leq i \leq n$ and $0 < h \leq j \leq m$, if the algorithm of finding global alignment is applied on subsequences $a_g...a_n$ and $b_h...b_m$, we obtain the optimal global alignments from the particular (g, h) to all possible (i, j). Since there are $m \times n$ (g, h) positions exist, then if we repeat the algorithm on every (g, h), we have optimal global alignments for all subsequence pair $a_g...a_i$ and $b_h...b_j$ from A and B. Then by iterating all obtained alignments, optimal local alignment of A and B with additive similarity metric will be the one with the highest score.

Furthermore, recall **Lemma** 3.4.6 in chapter 3, for any two sequences A and B, the optimal global alignment with the highest additive similarity score also has the highest normalized score. Therefore, it is unnecessary to apply the normalized similarity metric during finding global alignments for $a_g...a_n$ and $b_h...b_m$, because they will generate identical solutions.

As long as we have optimal global alignments for all possible subsequence pairs, normalization function can be applied to each of them to find the highest normalized score alignment, which is the optimal solution. Instead of storing all possible alignments at each position, normalization function can be applied at each (g, h) to find the alignment with the highest normalized score, then only one alignment needs to be stored at each (g, h). This method can reduce memory consumption, but if more normalization functions need to be applied after, the algorithm needs to be executed multiple times, so in order to reduce time consumption, we do not consider applying normalization functions during dynamic programming.

The algorithm for finding sequence global alignment, and the algorithm to obtain matrix containing all alignments for all possible subsequence pair are given in **Algorithm** 3 and 4, respectively in section 4.6. Notice that the algorithm for finding sequence global alignment is dynamic programming based, so the time complexity is $O(mn)$. Also, since there are $m \times n$ positions in H, the algorithm of global alignment needs to be executed $m \times n$ times, so the time complexity for the local alignment algorithm is $O(m^2n^2)$. Meanwhile, matrix H needs to store additive score, and segment lengths for each alignment candidate, the space complexity for the algorithm is also $O(m^2n^2)$.

4.2 Main Idea of New Algorithm

The above simple algorithm is to find optimal global alignments starting from each (g, h). Symmetrically, if we reverse sequences A and B to get $A' = a_na_{n-1}...a_1$ and $B' = b_mb_{m-1}...b_1$, after applying the above algorithm to A' and B', matrix H will store all alignments starting from each (g, h) to all possible (i, j), notice that $0 < i \leq g \leq n$ and $0 < j \leq h \leq m$ here in A' and B', so it is identical with generating all possible alignment ending at each (g, h) for A and B. Therefore, we modified the above algorithm, to start dynamic programming from position $(1, 1)$, and store alignments ending at each (i, j) which have the possibility to generate the highest normalized score along with dynamic programming after (i, j).

In chapter 2, we discussed the Smith-Waterman algorithm only stores the highest similarity score and discard other candidates at each $H_{i,j}$ to reduce both time and space consumption and it is sufficient for applying additive similarity metric. Such a method does not work correctly in our algorithm, because of segment length matters in calculating normalized similarity score after dynamic programming. Some low additive score alignments at (i, j) may be contained by a more extended alignment that passes through (i, j) and generates a high normalized score. The naive method to store nonredundant alignments at each $H_{i,j}$ is to keep optimal alignments starting from all possible positions and ending at (i, j), it can guarantee the optimal solution will always be found; however, it costs too much time and space. Besides the nonredundant alignments, there are many redundant candidates exist, which can be determined that they cannot generate a higher normalized score, not only at the position they are determined but also with any extension. Therefore, we designed an algorithm to remove as many redundant alignments as possible at each $H_{i,j}$, the specific reason and strategy will be introduced in the next section.

As long as we save all necessary information at each position, after dynamic programming, we can either iterate each position to find the optimal solution for the corresponding normalization function or check any specific position (i, j) to see the best alignment ending at (i, j). Besides, when finding optimal local alignment, not only the one with the highest normalized score will be found, but also other alignments with relatively high normalized scores can be found and listed as well.

4.3 Algorithm Design

4.3.1 Strategy of Redundant Alignments Rejection

For all candidates ending at the same position (i, j), redundant alignments mean they cannot provide higher normalized similarity score than another existing alignment, not only ending at (i, j), but also with extension. Then discarding redundant alignments will not affect the solution.

When comparing two alignments ending at (i, j), if the subsequences of one alignment do not fully contain the subsequences of the other alignment, then either of them has the possibility to produce higher normalized score along with the extension, so both of them need to be kept. On the other hand, if one alignment covered the other one, and suppose the inner one has a higher similarity score, then the outer alignment can be discarded for sure. So we have the following lemma:

Lemma 4.3.1 *For any two sequences $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$, let $0 < p \leq r < i \leq n$ and $0 < q \leq s < j \leq m$, let L_1 be the optimal global alignment for subsequence $a_p...a_i$ and $b_q...b_j$, and L_2 for $a_r...a_i$ and $b_s...b_j$. By applying normalized similarity metric 4.1, if the additive similarity score of L_1 is less or equal to the additive similarity score of L_2, then L_1 can be discarded.*

Proof To prove L_1 can be discarded at (i, j), it is necessary to show either both L_1 and L_2 ending at (i, j) without extension, or they merge at (i, j) and extend to further position, L_1 will not generate higher normalized similarity score.

Let $A_1 = a_p...a_i, B_1 = b_q...b_j$ denote the subsequences of L_1, also $A_2 = a_r...a_i$ and $B_2 = b_s...b_j$ denote the subsequences of L_2. Then the additive similarity scores of L_1 and L_2 are $s(A_1, B_1)$ and $s(A_2, B_2)$. Also, let $A_3 = a_i...a_u, B_3 = b_j...b_v$, suppose L_3 is optimal alignment starting from (i, j) and ending at (u, v) with similarity score $s(A_3, B_3)$, where $i \leq u \leq n$ and $j \leq v \leq m$, then L_1' and L_2' will be alignments which both contain L_3 but are extended from L_1

and L_2 separately. Notice that when $i = u$ and $j = v$, there will be no extension for L_1 and L_2, for this situation, we consider they both end at (i, j), then $s(A_3, B_3) = 0, L'_1 = L_1, L'_2 = L_2$. Let $A'_1 = a_p...a_u, B'_1 = b_q...b_v$ denote the subsequences L'_1, also $A'_2 = a_r...a_u$ and $B'_2 = b_r...b_v$ for L'_2. So the normalized similarity score of L'_1 is:

$$\overline{s}(A'_1, B'_1) = \frac{s(A'_1, B'_1)}{f(\sqrt[p]{(s(A'_1, A'_1) - s(A'_1, B'_1))^p + (s(B'_1, B'_1) - s(A'_1, B'_1))^p} + s(A'_1, B'_1))}, \tag{4.2}$$

and the normalized similarity score of L'_2 is:

$$\overline{s}(A'_2, B'_2) = \frac{s(A'_2, B'_2)}{f(\sqrt[p]{(s(A'_2, A'_2) - s(A'_2, B'_2))^p + (s(B'_2, B'_2) - s(A'_2, B'_2))^p} + s(A'_2, B'_2))}. \tag{4.3}$$

Notice that $s(A'_1, B'_1) = s(A_1, B_1) + s(A_3, B_3)$ and $s(A'_2, B'_2) = s(A_2, B_2) + s(A_3, B_3)$, so $s(A'_1, B'_1) \leq s(A'_2, B'_2)$. Let d'_1 and d'_2 denote $\sqrt[p]{(s(A'_1, A'_1) - s(A'_1, B'_1))^p + (s(B'_1, B'_1) - s(A'_1, B'_1))^p}$ and $\sqrt[p]{(s(A'_2, A'_2) - s(A'_2, B'_2))^p + (s(B'_2, B'_2) - s(A'_2, B'_2))^p}$ respectively. Then we have:

$$\frac{s(A'_1, B'_1)}{f(d'_1 + s(A'_1, B'_1))} \leq \frac{s(A'_2, B'_2)}{f(d'_1 + s(A'_2, B'_2))}, \tag{4.4}$$

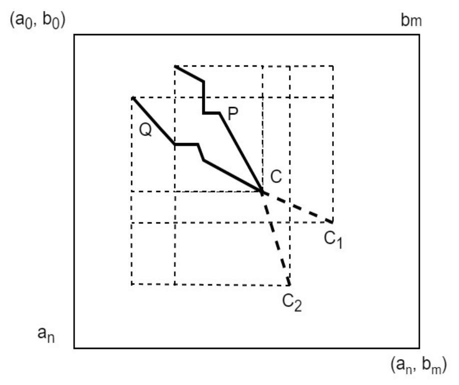
Also, since $A'_2 \leq A'_1$ and $B'_2 \leq B'_1$, $s(A'_1, B'_1) \leq s(A'_2, B'_2)$, then $d'_2 \leq d'_1$. So we can have:

$$\frac{s(A'_2, B'_2)}{f(d'_1 + s(A'_2, B'_2))} \leq \frac{s(A'_2, B'_2)}{f(d'_2 + s(A'_2, B'_2))}, \tag{4.5}$$

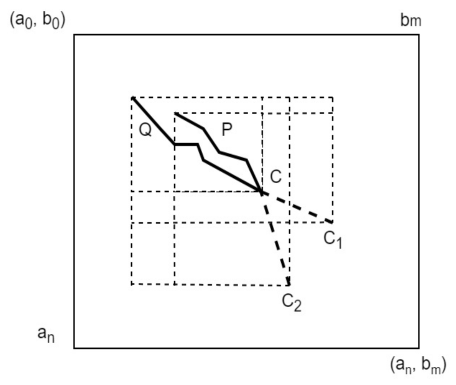
Then from 4.4 and 4.5,

$$\frac{s(A'_1, B'_1)}{f(d'_1 + s(A'_1, B'_1))} \leq \frac{s(A'_2, B'_2)}{f(d'_2 + s(A'_2, B'_2))}. \tag{4.6}$$

Therefore, L_1 cannot provide higher normalized similarity score, either ending at (i, j) or with extension, it can be discarded at (i, j). ∎



(a) Partially overlapping



(b) Fully covered

Figure 4.1: Any two alignments ending at the same position can have two different shapes based on their segment lengths. In 4.1a, neither of the alignment shapes fully contains the other one. Extending both alignments to C_1 and C_2 could give the opposite result, so either of them can be rejected at position C. In 4.1b, the subsequences of alignment Q fully contain the segments of P. If the similarity score of Q is lower, then no matter how the alignment extends, Q will not give better solution so that it can be discarded at position C.

If only consider two alignments which end at the same position (i, j), it is easy to make rejection decision by comparing the normalized similarity scores; however, along with dynamic programming, the two alignments can extend to further position, the alignment with lower normalized similarity score at (i, j) has the possibility to provide better alignment in the end. It depends on the shape of these two alignments at (i, j), and how they extend.

Figure 4.1 shows an example. Sequences $A = a_0a_1...a_n$ and $B = b_0b_1...b_m$, suppose we have two alignments P and Q ending at C, let x_p and y_p be the subsequence lengths of alignment P from A and B, also x_q and y_q for alignment Q. In Figure 4.1a, $x_p > x_q$ but $y_p < y_q$. Along with dynamic programming, both P and Q can extend to position C_1 or C_2. Recall the Minkowski distance in Equation 4.1 is:

$$d = \sqrt[p]{(s(a,a) - s(a,b))^p + (s(b,b) - s(a,b))^p}, \tag{4.7}$$

Where a and b stand for two subsequences. The comparison result of Minkowski distance of alignments P and Q can be opposite, based on the alignment extension direction. If the alignment extends to C_1, then the Minkowski distance of Q will increase much faster, which may cause P to be the one with the higher normalized score; however, if the extension is to C_2, the result may be different. Suppose the subsequence pair for P is A_p and B_p, also A_q and B_q for alignment Q. Then we can have the following specific example. If $s(A_p, A_p) = 60, s(B_p, B_p) = 30, s(A_q, A_q) = 20$ and $s(B_q, B_q) = 70$, also let $s(A_p, B_p) = s(A_q, B_q) = 100$ for convenient. We denote the subsequence pair from C to C_1 to be A_{c1} and B_{c1}, also A_{c2} and B_{c2} to be the subsequences from C to C_2. Let $S(A_{c1}, A_{c1}) = 10, S(B_{c1}, B_{c1}) = 30, S(A_{c2}, A_{c2}) = 30, S(B_{c2}, B_{c2}) = 10$ and $s(A_{c1}, B_{c1}) = s(A_{c2}, B_{c2}) = 5$. Since all the additive similarity scores are the same in this example for convenient, the dominant part in the normalization function 4.1 is the Minkowski distance. Firstly, if both alignments extend to C_1, then the Minkowski distance of P is:

$$d_p = \sqrt[p]{(60 + 10 - 20 - 5)^p + (30 + 30 - 20 - 5)^p} \tag{4.8}$$

Also the Minkowski distance of Q is:

$$d_q = \sqrt[p]{(20 + 10 - 20 - 5)^p + (70 + 30 - 20 - 5)^p} \tag{4.9}$$

Let $p = 2$, then $d_p = 57$, $d_q = 75$. Based on the normalized similarity metric shown in **Equation** 4.1, alignment P generate higher normalized score than Q, if the extension is to C_1, so P need to be saved at position C. However, if both P and Q extend to C_2, then the Minkowski distances for both P and Q are:

$$d_p = \sqrt[p]{(60 + 30 - 20 - 5)^p + (30 + 10 - 20 - 5)^p} \tag{4.10}$$

$$d_q = \sqrt[p]{(20 + 30 - 20 - 5)^p + (70 + 10 - 20 - 5)^p} \tag{4.11}$$

When $p = 2$, we have $d_p = 67$ and $d_q = 60$. The extension which is from Q will generate a higher normalized similarity score if both P and Q extend to C_2. In this situation, Q should also be kept at C. Therefore, based on these two scenario, if two alignments ending at same position, and the shape of one alignment does not fully cover the other one, neither of them can be eliminated, since at the current position, it is unknown that which one can provide better solution with extension, regardless of their current additive similarity score.

On the other hand, if the shape of Q fully cover P, like Figure 4.1b, and if the similarity score of P is no less than the score of Q, then regardless of how the alignment extends, the shape of Q will cover P, that means P always can produce higher normalized similarity score, so Q can be discarded at position C.

After realizing redundant candidates exist, we need to design how alignments are stored at each $H_{i,j}$, before attempting to remove them.

4.3.2 Data Structure

Recall the normalized similarity metric shown in **Equation** 4.1. In order to calculate the normalized score for alignment with corresponding subsequences A' and B' at any arbitrary (i, j), we need to know $s(A', B')$, $S(A', A')$ and $B(B', B')$. Besides the additive similarity score $s(A', B')$, segments A' and B' are required to calculate $S(A', A')$ and $B(B', B')$, so either the starting positions of A' and B' or their subsequence lengths need to be stored, in order to retrieve entire segments at (i, j). Besides, when sorting alignments at each position, segment lengths need to be compared; therefore, for each alignment, we firstly store the additive similarity score, and segment length in our algorithm. The starting position of any segment can be obtained by subtracting segment length from the ending position (i, j).

At each position (i, j), not one alignment but all nonredundant alignments need to be stored. They are generated by maintaining the stored candidates at $(i-1, j), (i-1, j-1)$ and $(i, j-1)$. In addition, if $s(a_i, b_j)$ is positive, then a new alignment with score $s(a_i, b_j)$ and both subsequence lengths one will be created and stored in $H_{i,j}$ as well. Consequentially, each position during dynamic programming will have alignments candidates coming from three directions; therefore, we used a list in each position to store nonredundant alignment candidates, in order to keep dynamic programming efficient, and easy to be maintained to the same structure at each position.

Furthermore, we need to decide the storing order of the candidate list. Briefly, candidates will be stored by descending order of additive similarity score, and same score alignments will be stored by ascending order of $x - length$. Then after removing redundancies, every position $H_{(i,j)}$ in matrix H should store a list of nonredundant alignments ending at (i, j), with the above order. The reason is when comparing candidates to remove redundancies, we attempt to reduce comparison times. Also, notice that there will not be any two alignments are stored in a list with same additive score and $x - length$ (or $y - length$), because if they have same score and same $x - length$ (or $y - length$), then the one with longer $y - length$ (or $x - length$) is redundant and will not be added into the list. From Lemma 4.3.1, it is known that in order to reject an alignment, there must be another alignment exist with higher additive similarity score and shorter segment length; therefore, it is relatively hard to reject high score alignments which can be used to reject lower score candidates. Due to this, at each position $H_{i,j}$, the candidates are stored by descending order of additive similarity score. Meanwhile, there could be some candidates with the same additive similarity score. For this scenario, as long as the particular alignment shape does not fully cover another alignment with the same score, it cannot be rejected by other candidates with same additive score, so shorter the alignment length is, less likely they will be rejected, in other words they are more likely to be the one to reject other alignments.

Next, we need to consider how to sort and remove all redundant candidates at each $H_{i,j}$ during dynamic programming, the method will be given in the following section.

4.3.3 Strategy of Merging and Sorting Candidates

At each position (i, j), we assume there is no redundant candidates exist in $(i-1, j)$, $(i-1, j-1)$ and $(i, j-1)$, and the alignment lists from them will be sorted by descending order of additive similarity score, for those alignments with the same score, they will be sorted by ascending order of $x - length$. Therefore, we need to remove all redundancy at (i, j), and store the left candidates as the same order at (i, j) to keep the assumption.

Firstly, we need to maintain the candidates from three directions by extending their subsequences and updating the additive similarity score. The process for maintaining candidates from each specific direction is shown below:

- From $(i-1, j)$: $x - length$ will be extended by 1, the other one remains the same, one indel penalty is added to the additive similarity score. If the updated additive score is not greater than 0, the alignment is removed;
- From $(i-1, j-1)$: both $x - length$ and $y - length$ will be extended by 1, score $s(a_i, b_j)$ will be added to the additive similarity score. If the updated additive score is not greater than 0, the alignment is removed; or if $s(a_i, b_j) > 0$, a new alignment with score $s(a_i, b_j)$ and both $x - length$ and $y - length$ one will be created;
- From $(i, j-1)$: $y - length$ will be extended by 1, the other one remains the same, one indel penalty is added to the additive similarity score. If the updated additive score is not greater than 0, the alignment is removed;

Even though there is no redundancy exists in the candidate lists of $(i-1, j), (i-1, j-1)$ and $(i, j-1)$, some of them may become redundant after maintaining and merging at (i, j). Figure 4.2 illustrates an example of shape for alignments ending at position (i, j). As discussed, we know by comparing candidates O and Q, it is impossible to reject either one regardless of their similarity scores, it is the same for comparing P and Q. However, if the sore of P is no less than O, then O can be rejected by P. Therefore, in order to make sure all redundant candidates will be removed, the naive method is to compare each pair of all candidates. Suppose we have k candidates ending at (i, j), then totally $\frac{k(k-1)}{2}$ comparisons have to be taken. To reduce the time consumption, we found a better method for comparison.

After maintaining the three lists from $(i, j-1), (i-1, j-1)$ and $(i-1, j)$, the first thing we do is merge them into a single list. During merging, we can remove the redundant alignments within the same additive similarity score groups. Notice that it may not be necessary to do so

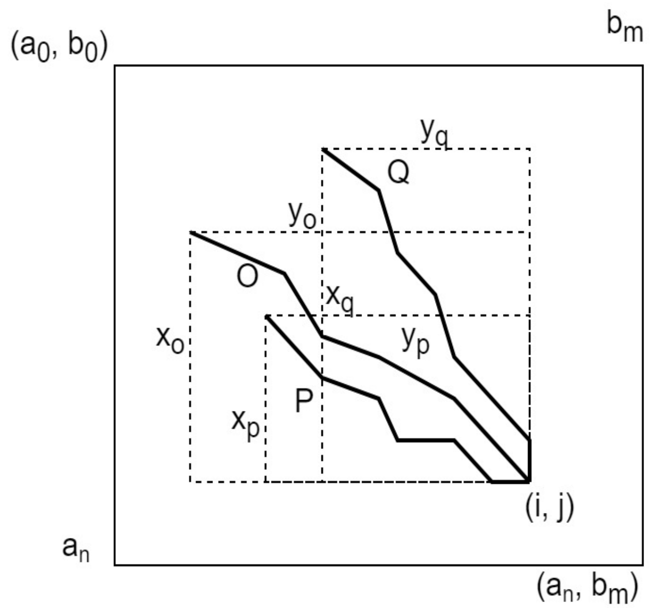


Figure 4.2: The general shapes of alignments ending at position (i, j). Suppose there are three candidates O, P and Q ending at (i, j), then the segment lengths for P are x_o and y_o. Also, x_p and y_p for P, x_q, y_q for Q.

during merging, but since it costs fixed time, so we do that for convenience. Suppose candidates O, P, and Q in Figure 4.2 have the same score and coming from three directions, then we find the one with minimum $x - length$, which is x_p. We knew that if one alignment is rejected, then there must be another one that has shorter $x - length$ and $y - length$ with no less similarity score. Therefore, among all the alignments with the same score, P will not be rejected for sure, because x_p is the minimum $x - length$. Then we can compare P with alignment O which has the second shortest $x - length$, in order to make rejection decision for O. Notice that we only need to compare $y - length$ here, if y_o is less than y_p, then candidate O can be kept temporarily. For this particular example, we can see the shape of P is covered by O, so O is discarded by Lemma 4.3.1. For another scenario, if y_o is less than y_p, then O is kept, and O will be used to compare with the candidates with third shortest $x - length$ and so on. In our algorithm, for comparing same score alignments, as long as the same score candidates are checked by the ascending order of $x - length$, then we only compare the next candidate with the previously kept one, because the previous kept candidates must have the shortest $y - length$ among all kept candidates with same additive score otherwise it will be discarded.

Specifically, we will have three sorted lists before merging. Let P_1, P_2, and P_3 be three alignment pointers which point to the first elements of three lists individually with corresponding additive scores and subsequence lengths showing below:

- Alignment P_1: **additive score** S_1, **x-length** x_1 and **y-length** y_1;
- Alignment P_2: **additive score** S_2, **x-length** x_2 and **y-length** y_2;
- Alignment P_3: **additive score** S_3, **x-length** x_3 and **y-length** y_3;

Here we only illustrate the situation when three of them have the same additive similarity score S, since it is the most complicated case. Suppose $S_1 = S_2 = S_3 = S$, also let M be the merged list. Then we will have the following scenarios during merging same score alignments:

- if $x_1 < x_2$ and $x_1 < x_3$, mark the alignment which pointed by P_1 as P_temp, move P_1 to the next element in the corresponding list;
- if $x_1 = x_2$ and $x_1 < x_3$, then if $y_1 \leq y_2$, mark the alignment which pointed by P_1 as P_{temp}, move P_1 and P_2 to the next element in the corresponding lists;
- if $x_1 = x_2$ and $s_1 = x_2$, then if $y_1 \leq y_2$ and $y_1 \leq y_3$, mark the alignment which pointed by P_1 as P_{temp}, move P_1, P_2, and P_3 to the next element in the corresponding lists;

Then, if P_{temp} is the first alignment stored in M with score S, or it is not the first one but its $y - length$ is shorter than the $y - length$ of the last element in M, then it will be directly added into M; otherwise, it means the previous element should have both shorter $x - length$ and $y - length$ than P_{temp}, but with the same score, then P_{temp} is redundant. If any pointer points to lower score alignment after moving, it will not be processed until its additive score is highest among all three pointers.

The comparison will keep going until every element from three lists is stored in M or discarded. After merging, we have a list of alignments ending at (i, j) which does not contain any redundancy within each same score group, and the list is sorted by similarity score in descending order; for those paths with the same score, they are sorted by $x-length$ in ascending order.

The merging step can guarantee that there is no redundant alignment within the candidates with the same similarity scores; however, we did not check if any higher score alignments can be used to reject lower score alignments during merging. For this situation, the simplest method is to compare all alignments with all relatively higher score alignments, but it will consume too much time. According to Lemma 4.3.1, in order to reject an alignment, we just need to find another alignment with a higher score and shorter $x-length$ and $y-length$. Hence we considered invoking a self-balancing tree to find if such alignments exist to reject lower score candidates. An augmenting red-black tree is designed to sort the merged list, and it will be discussed in the next section.

4.3.4 Red-Black Tree

Our problem is, for each candidate in the merged list, we need to search if alignment exists somewhere previously in the list, has a higher additive score and both shorter $x-length, y-length$. If such candidates exist, the current one is redundant and can be discarded; otherwise it needs to be kept at current position in matrix H. The simplest method is to compare current candidate with each one before it in the list, it can guarantee all redundant alignments will be found; however, the time complexity for this method is quadratic. Considering we only need to know if such a better candidate exists to reject an alignment, it is unnecessary to find the exact candidate. Typically, a self- balancing tree can be used to shorten the searching time. Due to **Lemma** 2.8.1 in chapter 2, the height of the red-black tree is $O(logn)$, where n is the tree size so that the time complexity for both searching and insertion is $O(logn)$. Hence, we considered invoking a red-black tree to solve this problem.

For each node in the tree, we decide to store the data of one candidate from the merged list at $H_{i,j}$, including $x-length$ and $y-length$. Recall that all redundant alignments within each score group are removed, then candidates who are already in the tree will not have less score than the current one, so it is unnecessary to store the additive score in tree node. The key of each node will be the $x - length$ of the corresponding candidate. Also, all nodes in the left subtree always have shorter $x - length$ than the root, and nodes in its right subtree always have greater $x - length$. Notice that there is no node in the tree that has the same $x - length$; the reason will be given.

In the red-black tree, each node only contains one key, but we need to consider both $x - length$ and $y - length$ when comparing two candidates, and we do not want to test each node in the subtree. Therefore, we augmented the red-black tree by adding one more attribute for each tree node, to determine the minimum $y - length$ among all the inserted alignments in the subtree rooted at the current node. It is called $minY$. Because when we compare the current candidate with a tree node, if its $x - length$ is greater than node key, then if its left child's $minY$ is less than the candidate's $y - length$, that means there must be an alignment exist in the tree have both shorter $x - length$ and $y - length$, so the candidate can be regarded without further testing.

Due to the pattern of merged list, any alignment in the list cannot be used to reject any other candidates before them, but may reject alignments after them, so we insert node from the left-hand side of merged list, that is the one with the highest additive score and shortest $x - length$ within the same score group. By this insertion order, any existing nodes (alignments) in the tree will not be rejected and removed from the tree. This is the reason why alignments with the same scores are sorted by ascending order of $x - length$; any tree node deletion will cost much extra time.

When an alignment candidate P with **x-length** x_p, **y-length** x_p and **similarity score** S_p is ready for comparison, firstly it will be compared with the root node R. Suppose the alignment stored in R has **x-length** x_r, **y-length** y_r and **similarity score** S_r. Then the comparison will have the following scenarios:

1. if $x_p < x_r$, then let R be the left child of R;

2. if $x_p = x_r$ and $y_p < y_r$, then change the alignment stored in R to alignment P;

3. if $x_p = x_r$ and $y_p \geq y_r$, then P can be discarded;

4. if $x_p > x_r$ and y_p is less than the $minY$ of R's left child, then let R be the right child of R;

5. if $x_p > x_r$ and y_p is greater than or equal to the $minY$ of R's left child, P can be discarded.

For cases 1 and 4, the loop will keep going until a leaf node is reached, or case 2, 3 or 5 happens.

Firstly if $x_p < x_r$, it means the alignment stored in R cannot judge alignment P; therefore, move to the left child of R and keep comparison. On the other hand, if $x_p = x_r$, then if $y_p < y_r$, the alignment stored in the root will be changed to P. This modification does not mean the previous alignment is discarded, it helps to keep the tree size as small as possible. For example,if $S_r = 10$, $x_r = 10$ and $y_r = 10$ for the alignment stored in R, $S_p = 9$, $x_p = 10$ and $y_p = 8$. If we do not replace the root alignment to P, then P will be inserted somewhere on the right side of R. At this time, suppose the next alignment T will be compared with the tree has similarity score 8, $x - length$ 10 but $y - length$ 9. By comparing with the root, T cannot be removed; then the comparison keeps going until reach the node contains P, which has a higher score than T, but the smaller shape, at this time, T can be discarded. On the other hand, suppose the alignment stored in root is changed to P. When comparing T, the decision can be made at the first comparison. Therefore, the replacing step can avoid inserting an extra node so that the comparison time is shortened. Also, there are no extra operations needed to maintain the tree, since the structure does not change, and the only modification is the data stored in R, which is fixed time consumption. Next, if $x_p = x_r$ and $y_p \geq y_r$, then it means P can be removed by R. The third case of comparison is $x_p > x_r$, then if y_p is less than the $minY$ of R's left child, move to the right child of R and keep comparison; otherwise, that means there must be another alignment that exists on the left side of root, has less $x - length$ and $y - length$ than P, then it can be discarded.

If there is no such alignment exists in the tree to reject the new candidate, it will be stored in the list of $H_{i,j}$ and inserted at the correct position in the augmenting red-black tree as a new node, the tree structure need to be checked, and certain operations will be taken if needed. Algorithm 5 is the insertion function, and the time complexity is $O(\log n)$, where n here is the size of candidate list. The new attribute *minY* actually will not affect the time consumption, because updating *minY* cost fixed time, and the rest part is the same as a regular red-black tree. The specific maintenance steps can be found in Algorithm 6, 7, and 8.

After trying to insert all elements from the merged list, all successfully inserted alignment are not redundant at (i, j), they will be stored in the list of $H_{i,j}$.

4.4 Process Stored Data by Applying Normalized Similarity Metric

After dynamic programming, a matrix H, which stores all the necessary data at each position (i, j), has been obtained. Let $S[i, j]$ denotes the list of all stored alignments which start from the different position but ending at (i, j). For different applications, any number of normalized similarity metric can be applied to find the corresponding optimal normalized local alignment; this can be done by iterating each element of H and testing all the candidates by the corresponding metric. Meanwhile, during iteration, a list can be generated, which contains a certain number of local alignments sorted by their normalized similarity scores.

Furthermore, because the alignment data at each point is kept, then it is possible to directly access the data in a particular position (i, j) and find the optimal local alignment ending at this point, without rerunning dynamic programming.

Therefore, our algorithm not only can find an optimal solution based on corresponding normalized similarity metric but also can find other biologically meaningful alignments that are not as good as the optimal solution but with relatively high normalized scores.

4.5 Time and Space Complexity Analysis

The theoretical time complexity of the whole program is $O(m^2n^2 \times \log(mn))$ since, at any position (i, j), the worst case is that there are $i \times j$ candidates need to be processed, then the maximum red-black tree height would be $\log(ij)$. However, our algorithm always remove redundant alignments at each position, so the maximum candidates number processed in each particular position have never been this huge, the actual complexity of this program is much smaller than $O(m^2n^2 \times \log(mn))$, it is close to $O(mn \log(k))$, where k is a constant which is much smaller than m or n. Since the program need to store the matrix H, the space complexity is $O(m^2n^2)$ theoretically in the worst case, but it never happened, so the actual space complexity is close to $O(mn)$.

4.6 Corresponding Algorithms

Algorithm 3 FindingGlobalAlignment

```
   Input: sequences $A = a_1...a_n$ and $B = b_1...b_m$, similarity metric $s(x, y)$, indel penalty $\delta$;
   Output: score matrix $H$
 1: initialize matrix $H$ with size $(m + 1) \times (n + 1)$
 2: set $H_{0,0} = 0$
 3: for $i$ is from 1 to $n$ do
 4:     set $H_{i,0} = H_{i-1,0} + \delta$
 5: end for
 6: for $j$ is from 1 to $m$ do
 7:     set $H_{0,j} = H_{0,j-1} + \delta$
 8: end for
 9: for $i$ is from 1 to $n$ do
10:     for $j$ is from 1 to $m$ do
11:         let $H_{i,j}$ be the maximum of $H_{i-1,j} + \delta$, $H_{i-1,j-1} + s(a_i, b_j)$ and $H_{i,j-1} + \delta$
12:     end for
13: end for
14: return $H$
```

Algorithm 4 GettingAllOptimalAlignmentCandidates

```
Input: sequences $A = a_1...a_n$ and $B = b_1...b_m$, similarity metric $s(x, y)$, indel penalty $\delta$;
Output: matrix $H$, stores all alignments starting from each possible position
1: initialize matrix $H$ with size $m \times n$
2: for $g$ is from $n - 1$ to 1 do
3:     for $h$ is from $m - 1$ to 1 do
4:         FindingGlobalAlignment($A_g = a_g...a_n, B_h = b_h...b_m, s(a, b), \delta$)
5:         store all optimal global alignments starting from $(g, h)$ to $H_{i,j}$
6:     end for
7: end for
8: return $H$
```

Algorithm 5 Augmented red-black tree insertion

```
Input: root node $R$ and new node $N$;
Output: true for successful insertion, false otherwise
1: while root node is not empty do
2:     if $N.x - length < N.x - length$ then
3:         set $N$ to $N$'s left child
4:     else
5:         if $N.x - length = R.x - length$ then
6:             if $N.y - length < R.y - length$ then
7:                 replace the path stored in $R$ to the new path in $N$
8:                 return true
9:             end if
10:            return false
11:        else
12:            if $N.y - length < R.y - length$ then
13:                if $N.y - length < R.minY$ or $R$'s left child is a leaf then
14:                    set $R$ to $R$'s right child
15:                else
16:                    return false
17:                end if
18:            else
19:                return false
20:            end if
21:        end if
22:    end if
23: end while
24: \\ $R$ is now the position to insert new node
25: add the new path
26: maintenance($R$)
```

Algorithm 6 Maintenance

```
Input: the inserted node $R$;
Output: maintain the tree structure
while $R$'s is not root and its parent's color is red do
    if $R$'s parent is the left child of $R$'s grandparent then
        let $right$ be the right child of $R$'s grandparent
        if $right$'s color is red then
            set $R$'s parent's color to black
            set $right$'s color to black
            set $R$'s grandparent's color to red
            set $R$ to $R$'s grandparent
        else
            if $R$ is right child then
                set $R$ to its parent
                leftRotate($R$)
                set $R$'s parent's color to black
                set $R$'s grandparent's color to red
                rightRotate($R$' grandparent)
            else
                set $R$'s parent's color to black
                set $R$'s grandparent's color to red
                rightRotate($R$'s grandparent)
            end if
        end if
    else
        let $left$ be the left child of $R$'s grandparent
        if $left$ is red then
            set $R$'s parent's color to black
            set $left$'s color to black
            set $R$'s grandparent's color to red
            set $R$ to its grandparent
        else
            if $R$ is left child then
                set $R$ to its parent
                rightRotate($R$)
                set $R$'s parent's color to black
                set $R$'s grandparent's color to red
                leftRotate($R$'s grandparent)
            else
                set $R$'s parent's color to black
                set $R$'s grandparent's color to red
                leftRotate($R$'s grandparent)
            end if
        end if
    end if
end while
set root's color to black
return true
```

Algorithm 7 LeftRotate

```
   Input: node N;
   Output: adjust local tree structure
1: let temp be the right child of N
2: let temp's left child be the right child of N
3: if N is root then
4:     let temp be the new root
5: else
6:     let temp be the child of N's parent, instead of N
7: end if
8: let N be the left child of temp
9: N.renewMinY()
```

Algorithm 8 RightRotate

```
   Input: node N;
   Output: adjust local tree structure
1: let temp be the left child of N
2: let temp's right child be the left child of N
3: if N is root then
4:     let temp be the new root
5: else
6:     let temp be the child of N's parent, instead of N
7: end if
8: let N be the right child of temp
9: N.renewMinY()
```

Chapter 5

Experiment

We used the above approach to find normalized local alignments of different protein sequences pairs, during the experiment several concave functions were tested. Finally, a phylogenetic tree was built over 20 mammals.

5.1 Protein Sequences

Proteins are large biomolecules and perform a massive number of functions within organisms, and consist of at least one amino acid chain, which is assembled by the different number of amino acids. So far, it has been studied that there are 20 standard amino acids coding from genes; therefore, protein sequences are much more variable than DNA sequences. The 20 amino acids are represented by 20 alphabetic letters so that a protein sequence can be treated as a finite string over a $20 - letter$ alphabet, shows in Table 5.1.

Let Σ be the corresponding $20 - letter$ alphabet, and A, B be two finite strings over Σ, where $A = a_1a_2...a_n$ and $B = b_1b_2...b_n$. The goal is to find subsequences $a_g...a_i$ and $b_k...b_j$ within A, B that have the highest similarity degree, where $0 < g < i \leq n$, $0 < k < j \leq m$. Notice that there are two more additional amino acids are found in some species to interpret stop codons; also BLOSUM matrices consist 24 letters, which the last 3 alphabetic letters are used to identify unknown residues during chemical analysis, and the $*$ letter is for matching a space.

amino acids	alphabetic letter	amono acids	alphabetic letter
Alanine	A	Leucine	L
Arginine	R	Lysine	K
Asparagine	N	Methionine	M
Aspartic acid	D	Phenylalanine	F
Cysteine	C	Proline	P
Glutamine	Q	Serine	S
Glutamic acid	E	Threonine	T
Glycine	G	Tryptophan	W
Histidine	H	Tyrosine	Y
Isoleucine	I	Valine	V

Table 5.1: amino acids table.

5.2 Datasets

The protein sequences we used for experiments are encoded by animal's mitochondrial DNA. It is known that in most cases, the inheritance of an animal's mtDNA is uniparental from mother [5], so mtDNA is highly conserved. This property is a potent tool to study the evolutionary relationships between organisms. The size of animal mtDNA is around 16k base pairs and contains the same 37 genes: 13 for proteins, 22 for tRNAs, and 2 for rRNAs [4]. An example of mtDNA is given in Figure 5.1, and it is a human mtDNA. By the sequence order of mtDNA, the 13 proteins are NADH dehydrogenase subunit 1 (NADH1), NADH dehydrogenase subunit 2 (NADH2), cytochrome c oxidase subunit I (COX1), cytochrome c oxidase subunit II (COX2), ATP synthase F0 subunit 8 (ATP8), ATP synthase F0 subunit 6(APT6), cytochrome c oxidase subunit III (COX3), NADH dehydrogenase subunit 3 (NADH3), NADH dehydrogenase subunit 4L (ND4L), NADH dehydrogenase subunit 5 (NADH5), NADH dehydrogenase subunit 6 (NADH6) and cytochrome b (CYTB).

We only use the 13 proteins portion for experiment, rather than the whole mtDNA sequence. The 20 species included in the experiment are blue whale (*balaenoptera musculus*), cat (*felis catus*), chimpanzee (*Pan troglodytes*), cow (*bos taurus*), finback whale (*balaenoptera physalus*), gibbon (*hylobates agilis*), gorilla (*gorilla gorilla*), grey seal (*halichoerus grypus*), harbo seal

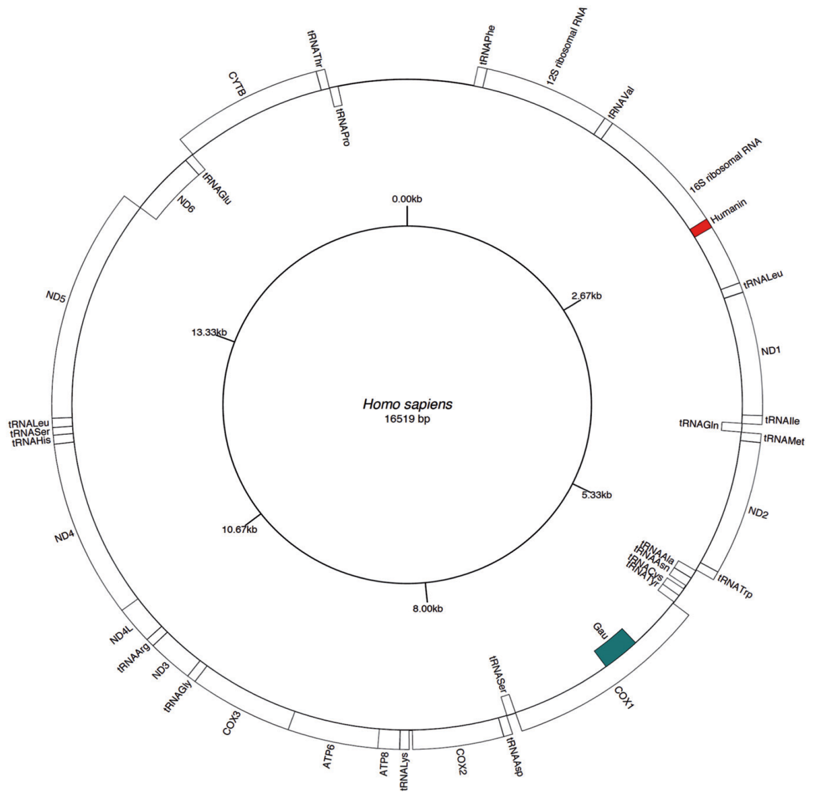


Figure 5.1: Human mitochondrial DNA gene map [5].

(*phoca vitulina*), human (*homo sapiens*), horse (*equus caballus*), house mouse (*mus musculus*), opossum (*micoureus demerarae*), bornean orangutan (*pongo pygmaeus*), platypus (*ornithorhynchus anatinus*), pygmy chimpanzee (*pan paniscus*), rat (*rattus norvegicus*), sumatran orangutan (*pongo abelii*), wallaroo (*osphranter robustus*) and white rhino(*ceratotherium simum*). All the above 20×13 protein sequences are obtained from GenBank in fasta file format. An example is given in Figure 5.2.

```
>NP_008212.1 NADH dehydrogenase subunit 1 (mitochondrion) [Gorilla gorilla]
MSMANLLLLIVPILIAMAFLMLTERKILGYMQLRKGPNVVGPYGLLQPFADAMKLFTKEPLKPSTSTITL
YITAPTLALTIALLLWTPLPMPNPLVNLNLGLLFILATSSLAVYSILWSGWASNSNYALIGALRAVAQTI
SYEVTLAIILLSTLLMNGSFNLSTLIMTQEHLWLLLPTWPLAMMWFISTLAETNRTPFDLAEGESELVSG
FNIEYAAGPLALFFMAEYMNIIMMNTLTTMIFLGTTYNAHSPELYTVCFITKTLLLTSLFLWIRTAYPRF
RYDQLMHLLWKNFLPLTLALLMWYISMPTTISSIPPQT
```

Figure 5.2: An example of fasta file, the first line is file description, and the rest lines are the sequence of the particular protein which is inside the square bracket at the end of the description.

Besides the protein sequences, we employed *BLOSUM62* for similarity metric, showed in Figure 5.3. All the values in the table were rounded to the nearest integers, and it still satisfies the similarity metric definition.

```
#  Matrix made by matblas from blosum62.iij
#  * column uses minimum score
#  BLOSUM Clustered Scoring Matrix in 1/2 Bit Units
#  Blocks Database = /data/blocks_5.0/blocks.dat
#  Cluster Percentage: >= 62
#  Entropy =   0.6979, Expected =  -0.5209
   A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  B  Z  X  *
A  4 -1 -2 -2  0 -1 -1  0 -2 -1 -1 -1 -1 -2 -1  1  0 -3 -2  0 -2 -1  0 -4
R -1  5  0 -2 -3  1  0 -2  0 -3 -2  2 -1 -3 -2 -1 -1 -3 -2 -3 -1  0 -1 -4
N -2  0  6  1 -3  0  0  0  1 -3 -3  0 -2 -3 -2  1  0 -4 -2 -3  3  0 -1 -4
D -2 -2  1  6 -3  0  2 -1 -1 -3 -4 -1 -3 -3 -1  0 -1 -4 -3 -3  4  1 -1 -4
C  0 -3 -3 -3  9 -3 -4 -3 -3 -1 -1 -3 -1 -2 -3 -1 -1 -2 -2 -1 -3 -3 -2 -4
Q -1  1  0  0 -3  5  2 -2  0 -3 -2  1  0 -3 -1  0 -1 -2 -1 -2  0  3 -1 -4
E -1  0  0  2 -4  2  5 -2  0 -3 -3  1 -2 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
G  0 -2  0 -1 -3 -2 -2  6 -2 -4 -4 -2 -3 -3 -2  0 -2 -2 -3 -3 -1 -2 -1 -4
H -2  0  1 -1 -3  0  0 -2  8 -3 -3 -1 -2 -1 -2 -1 -2 -2  2 -3  0  0 -1 -4
I -1 -3 -3 -3 -1 -3 -3 -4 -3  4  2 -3  1  0 -3 -2 -1 -3 -1  3 -3 -3 -1 -4
L -1 -2 -3 -4 -1 -2 -3 -4 -3  2  4 -2  2  0 -3 -2 -1 -2 -1  1 -4 -3 -1 -4
K -1  2  0 -1 -3  1  1 -2 -1 -3 -2  5 -1 -3 -1  0 -1 -3 -2 -2  0  1 -1 -4
M -1 -1 -2 -3 -1  0 -2 -3 -2  1  2 -1  5  0 -2 -1 -1 -1 -1  1 -3 -1 -1 -4
F -2 -3 -3 -3 -2 -3 -3 -3 -1  0  0 -3  0  6 -4 -2 -2  1  3 -1 -3 -3 -1 -4
P -1 -2 -2 -1 -3 -1 -1 -2 -2 -3 -3 -1 -2 -4  7 -1 -1 -4 -3 -2 -2 -1 -2 -4
S  1 -1  1  0 -1  0  0  0 -1 -2 -2  0 -1 -2 -1  4  1 -3 -2 -2  0  0  0 -4
T  0 -1  0 -1 -1 -1 -1 -2 -2 -1 -1 -1 -1 -2 -1  1  5 -2 -2  0 -1 -1  0 -4
W -3 -3 -4 -4 -2 -2 -3 -2 -2 -3 -2 -3 -1  1 -4 -3 -2 11  2 -3 -4 -3 -2 -4
Y -2 -2 -2 -3 -2 -1 -2 -3  2 -1 -1 -2 -1  3 -3 -2 -2  2  7 -1 -3 -2 -1 -4
V  0 -3 -3 -3 -1 -2 -2 -3 -3  3  1 -2  1 -1 -2 -2  0 -3 -1  4 -3 -2 -1 -4
B -2 -1  3  4 -3  0  1 -1  0 -3 -4  0 -3 -3 -2  0 -1 -4 -3 -3  4  1 -1 -4
Z -1  0  0  1 -3  3  4 -2  0 -3 -3  1 -1 -3 -1  0 -1 -3 -2 -2  1  4 -1 -4
X  0 -1 -1 -1 -2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -2  0  0 -2 -1 -1 -1 -1 -1 -4
* -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4 -4  1
```

Figure 5.3: Rounded BLOSUM62 table, obtained from NCBI.

5.3 Protein sequence local alignment

In this section, an experiment of aligning protein sequences is given. The two input sequences are both for protein ATP6, and are from human and cat respectively, showing in **Figure** 5.4.

Human	MNENLFASFIAPTILGLPAAVLIILFPPLLIPTSKYLINNRLITTQQWLIKLTSKQMMTMHNTKGRTWSLMLVSLIIFIATTNLLGLLPHSFTPTTQ LSMNLAMAIPLWAGTVIMGFRSKIKNALAHFLPQGTPTPLIPMLVIIETISLLIQPMALAVRLTANITAGHLLMHLIGSATLAMSTINLPSTLIIFT ILILLTILEIAVALIQAYVFTLLVSLYLHDNT
Cat	MNENLFASFTTPTMMGLPIVILIIMFPSILFPSPNRLINNRLVSLQQWLVQLTSKQMLAIHNHKGQTWALMLMSLILFIGSTNLLGLLPHSFTPT TQLSMNLGMAIPLWAGTVITGFRHKTKASLAHFLPQGTPVPLIPMLVVIETISLFIQPMALAVRLTANITAGHLLMHLIGGAALALMNISTSIAL ITFTILILLTILEFAVALIQAYVFTLLVSLYLHDNT

Figure 5.4: Two sequences of protein ATP6, one is from human and the other one is from cat.

For experiment, by using the normalized similarity metric:

$$\overline{s}(x, y) = \frac{s(x, y)}{f(d(x, y) + s(x, y))}. \tag{5.1}$$

For similarity metric $s(x, y)$, we applied BLOSUM62. We tested three different concave functions, the result is showed in **Figure** 5.5.

The result shows that similarity degree decreased, along with the decreasing of the power of x in concave functions. On the other hand, by applying the Smith-Waterman algorithm with BLOSUM62, the optimal local alignment is identical with the input sequences; therefore, it is ineffective in this situation.

5.4 Build Phylogenetic Tree by Neighbor Joining Method

A phylogenetic tree is a graph that consists of different species as terminal vertices, and the edge weights between each node are the actual distance among aligned sequences. There are also internal nodes which are used to lower the total distance of the graph and represent the similarity degree among species.

$y = x^{0.9}$	human	LAHFLPQGTPTPLIPMLVIIETISLLIQPMALAVRLTANITAGHLLMHLIG
	cat	LAHFLPQGTPVPLIPMLVVIETISLFIQPMALAVRLTANITAGHLLMHLIG
$y = x^{0.85}$	human	TNLLGLLPHSFTPTTQLSMNLAMAIPLWAGTVIMGFRSKIKNALAHFLPQGTPTPLIPMLVIIETISLLIQPMALAV RLTANITAGHLLMHLIG
	cat	TNLLGLLPHSFTPTTQLSMNLGMAIPLWAGTVITGFRHKTKASLAHFLPQGTPVPLIPMLVVIETISLFIQPMALA VRLTANITAGHLLMHLIG
$y = x^{0.75}$	human	LINNRLITTQQWLIKLTSKQMMTMHNTKGRTWSLMLVSLIIFIATTNLLGLLPHSFTPTTQLSMNLAMAIPLWAG TVIMGFRSKIKNALAHFLPQGTPTPLIPMLVIIETISLLIQPMALAVRLTANITAGHLLMHLIGSATLAMSTINLPST LIIFTILILLTILEIAVALIQAYVFTLLVSLYLHDN
	cat	LINNRLVSLQQWLVQLTSKQMLAIHNHKGQTWALMLMSLILFIGSTNLLGLLPHSFTPTTQLSMNLGMAIPLWAG TVITGFRHKTKASLAHFLPQGTPVPLIPMLVVIETISLFIQPMALAVRLTANITAGHLLMHLIGGAALALMNISTSIAL ITFTILILLTILEFAVALIQAYVFTLLVSLYLHDN
Smith-Waterman	human	MNENLFASFIAPTILGLPAAVLIILFPPLLIPTSKYLINNRLITTQQWLIKLTSKQMMTMHNTKGRTWSLMLVSLIIFI ATTNLLGLLPHSFTPTTQLSMNLAMAIPLWAGTVIMGFRSKIKNALAHFLPQGTPTPLIPMLVIIETISLLIQPMAL AVRLTANITAGHLLMHLIGSATLAMSTINLPSTLIIFTILILLTILEIAVALIQAYVFTLLVSLYLHDNT
	cat	MNENLFASFTTPTMMGLPIVILIIMFPSILFPSPNRLINNRLVSLQQWLVQLTSKQMLAIHNHKGQTWALMLMSL ILFIGSTNLLGLLPHSFTPTTQLSMNLGMAIPLWAGTVITGFRHKTKASLAHFLPQGTPVPLIPMLVVIETISLFIQ PMALAVRLTANITAGHLLMHLIGGAALALMNISTSIALITFTILILLTILEFAVALIQAYVFTLLVSLYLHDNT

Figure 5.5: By applying multiple concave functions, the solutions have different similarity degrees. The solution generated by the Smith-Waterman algorithm is identical with the original sequences.

There are many methods to build a phylogenetic tree such as the maximum-likelihood method [19] and by k-mers [13]. For our experiment, since the algorithm is to find optimal local alignment, and the normalized similarity score can be transferred to distance, then we used the neighbor-joining method, which is one of the most efficient methods to build a phylogenetic tree. In the beginning, we have a starlike graph that only contains one internal vertex, and it connects to all terminal nodes which represent each species. Moreover, a new internal vertex will be created to cluster any two species that can obtain the minimum total distance, and such grouping is the optimal solution in the current joining step. This will be looped $n - 1$ times, where n is the species number [16]. We used the approach from [11] to do neighbor join, and it has been proved that the method gives exactly the same phylogenetic tree as the original method in [16].

There are two constraints for using neighbor-joining by our local alignments result. Firstly, for each individual protein, when aligning with other sequences, the optimal local alignments may correspond to different regions. Even though we can transfer the similarity scores to distances, they cannot be compared with each other, since each distance value represents different segment pairs. To solve this problem, we tried to find a fixed subsequence from each protein sequence. For each protein sequence, after finding all the local alignments by aligning with other proteins, we trimmed it based on the longest alignment length among all those alignments. So the subsequence within the range will be kept. For example, suppose X, Y and Z are three protein sequences, and for sequence X, we will get individual best local alignments from X, Y, and X, Z, which related to different subsequences of X (L_1 in Figure 5.6). Then we compare all the alignments which contain X, and the longest L_1 will be used to trim X. In other words, we use local alignment results to find a subsequence of each protein; these subsequences will be used to represent corresponding protein sequences for finding global distances.

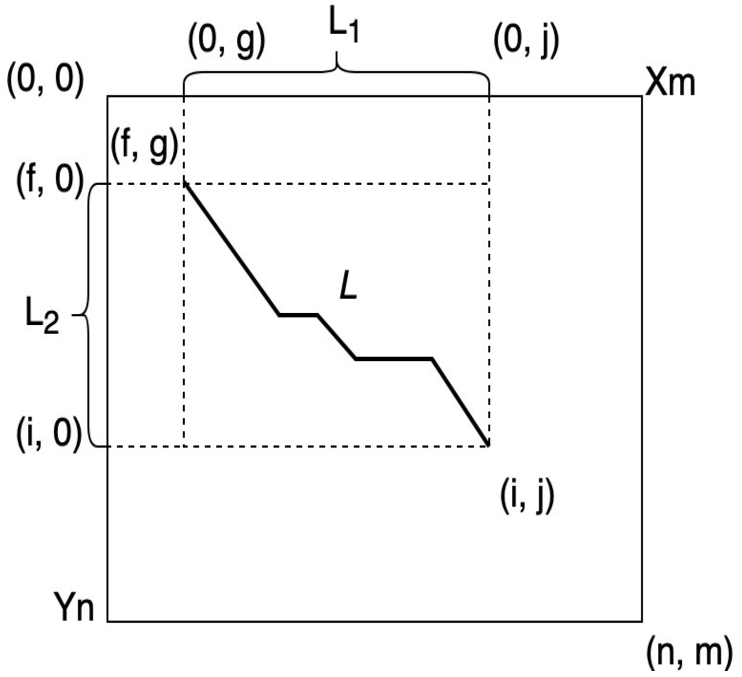


Figure 5.6: For alignment of sequence X and Y, we can get L_1 and L_2 length, but if changing sequence Y to Z, new L_1 will be obtained corresponding to the new optimal alignment.

Moreover, the similarity scores need to be transferred to distance values, then sum up the total distances for all 13 proteins. Notice that for each protein, all species have similar sequence lengths; nevertheless, for different proteins, the sequence lengths are diverse. For example, normally, the sequence lengths of protein NADH5 are over 600 amino acids, but protein ATP8 only contains less than 80 amino acids. Therefore, for each pair of species such as human and cat, if we sum up the distances of all 13 proteins, longer protein sequences will distribute more to the total distance of these two species. Therefore, we attempted to normalize the distances to interpret the actual distance of any two species for each protein. It has been proved in [20] that **Equation** 5.2 is a normalized distance metric, where $f(x)$ is a concave function, and $d_{L_2}(x,y) = \sqrt[2]{(s(x,x) - s(x,y))^2 + (s(y,y) - s(x,y))^2}$. Just like the normalized similarity metric which we used to find out optimal alignment, this distance metric interprets how different the two sequences are.

$$\overline{d}(x,y) = \frac{d_{L_2}(x,y)}{f(d_{L_2}(x,y) + s(x,y))} \tag{5.2}$$

At the beginning of our experiment, when calculating $d_{L_2}(x,y)$ in Equation 5.2, we used the trimmed sequences to generate $s(x,x)$ and $s(y,y)$, but the experiment result was not ideal. Then we considered that the parts had been discarded after trimming should also take into account, so we tried to use the original sequences to generate $s(x,x)$ and $s(y,y)$, the output phylogenetic tree was improved, and it is shown in **Figure** 5.7.

5.5 Experiment Result

Figure 5.7 is the phylogenetic tree, which was built following the above steps, and the similarity metric used is *BLOSUM62*. We also tried *BLOSUM80*, and the tree is identical to using *BLOSUM62*. Meanwhile, we constructed a contrasting phylogenetic tree, which invoked the Smith-Waterman algorithm to find local alignments. The tree is shown in Figure 5.8.

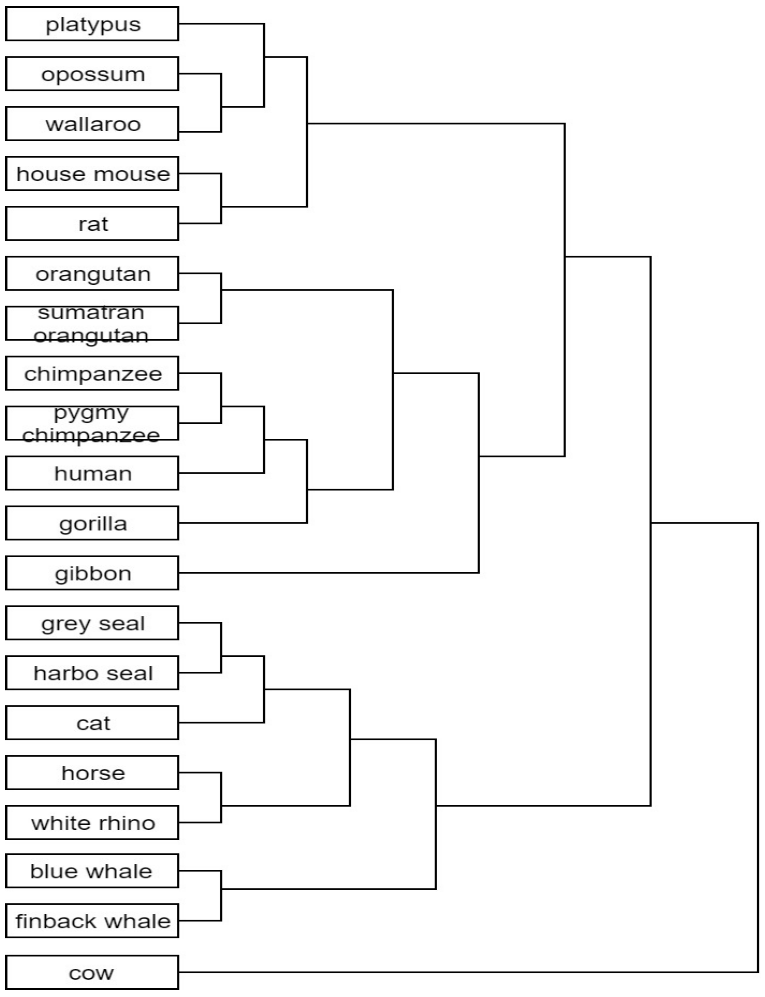


Figure 5.7: Phylogenetic tree of 20 mammals, the protein sequences were trimmed by the new approach. and the distances were used by neighbor joining are normalized. Similarity metric is *BLOSUM62*.

5.6 Evaluation

Firstly, we tracked the dynamic programming part of the experiment. If the length of protein sequences is around 230 amino acids, at each position (i, j), there will be around 30 candidates that have been stored, which means our algorithm efficiently reduced the size of the candidate list stored in each $H_{i,j}$. Furthermore, the protein sequences we used for experiments have high similarity degree, so if the sequences used are not closely related, then the average list size will be smaller.

Moreover, from the phylogenetic tree 5.7, we can see Primates, Ferungulates, and Rodents

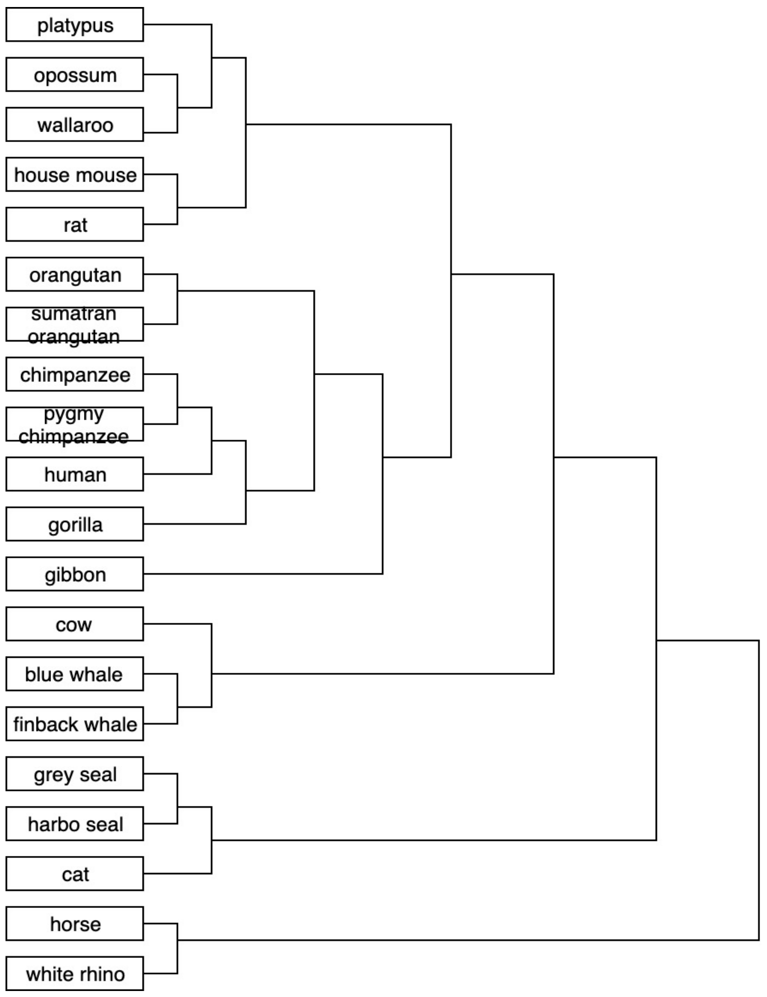


Figure 5.8: This Phylogenetic tree used the trimmed protein sequences which were generated by Smith-Waterman Algorithm, and the distances used by neighbor joining were not normalized.

are separately clustered correctly. The only flaw we got is cow, finback whale and, blue whale are all belong to Artiodactyla order biologically, but in the tree, cow is not closely related to those two species. The reason was unknown when finishing this thesis; a future study will be taken to figure out the reason. By contrast, the tree built by Smith-Waterman cannot cluster Ferungulates.

Meanwhile, we tried three different concave function for normalization: $f_1(x) = x^{0.95}$, $f_2(x) = x^{0.75}$ and $f_3(x) = x^{0.5}$; furthermore, the local alignments obtained by $f_3(x)$ are identical with the outputs from the Smith-Waterman algorithm. By contrast, the normalization strength of $f_1(x)$

is too much, so the local alignments are short, and the corresponding segments are highly identical. This function cannot provide ideal solutions even for aligning mitochondrial protein sequences, so it may give worse results for aligning dissimilar sequences. $f_2(x)$ works fine, because the optimal solutions given by this function are not so long as the output of Smith-Waterman, and not too short to distinguish the distances between different species.

Chapter 6

Conclusion

In this thesis, we have studied the topic of finding sequence local alignment by applying normalized similarity metric. Firstly we discussed the Smith-Waterman algorithm, which is one of the earliest methods for finding sequence local alignment. Then, we analyzed this algorithm may not be able to provide meaningful results in some scenarios because it only returns the alignment with the highest additive similarity score. Then reviewed a few approaches for normalizing similarity score or distance, in order to get the alignment with the required similarity degree.

Then, our new algorithm has been introduced. It is designed to find the optimal local alignment with the highest normalized similarity score. Firstly, it invokes a Minkowski distance typed normalized similarity metric family to find different optimal local alignments with distinct similarity degree, in order to satisfy the different requirements for biological applications. Meanwhile, after executing the algorithm, all necessary data is stored as a matrix, so that dynamic programming can be avoided when other normalization functions from the metric family need to be applied. It can be achieved by iterating the alignment candidates stored in the generated matrix. The theoretical time complexity of our algorithm is $O(m^2n^2)$, but since all redundant candidates are removed at each position of the matrix, the actual time complexity is much lower.

In the end, the experiment result was shown and discussed. A phylogenetic tree which includes 20 animal species, was built, based on the data obtained from our new algorithm.

Almost all species were grouped correctly by their biological order. Meanwhile, another tree was built by using the same raw data as the previous experiment but processed by the Smith-Waterman algorithm. The experiment result shows that our algorithm successfully find sequence local alignment with high similarity degree, based on normalized similarity metric.

In the future study, we attempt to discover if any other redundant candidates are stored in matrix H. Also, we will try to find a better trimming method for protein sequences, in order to produce a more accurate phylonegetic tree.

Bibliography

[1] Needleman–Wunsch algorithm. Needleman–wunsch algorithm — Wikipedia, the free encyclopedia, 2018. [Online; accessed 27-December-2018].

[2] Abdullah N Arslan and Omer Egecioglu. An efficient uniform-cost normalized edit distance algorithm. In *6th International Symposium on String Processing and Information Retrieval. 5th International Workshop on Groupware (Cat. No. PR00268)*, pages 8–15. IEEE, 1999.

[3] Abdullah N Arslan, Ömer Eğecioğlu, and Pavel A Pevzner. A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics*, 17(4):327–337, 2001.

[4] Jeffrey L Boore. Animal mitochondrial genomes. *Nucleic acids research*, 27(8):1767–1780, 1999.

[5] Charlotte Capt, Marco Passamonti, and Sophie Breton. The human mitochondrial genome may code for more than 13 proteins. *Mitochondrial DNA Part A*, 27(5):3098–3101, 2016.

[6] MA Chang, MO Dayhoff, RV Eck, and MR Sochard. Atlas of protein sequence and structure. 1965.

[7] Shihyen Chen, Bin Ma, and Kaizhong Zhang. The normalized similarity metric and its applications. In *2007 IEEE International Conference on Bioinformatics and Biomedicine (BIBM 2007)*, pages 172–180. IEEE, 2007.

[8] Shihyen Chen, Bin Ma, and Kaizhong Zhang. On the similarity metric and the distance metric. *Theoretical Computer Science*, 410(24-25):2365–2376, 2009.

[9] Werner Dinkelbach. On nonlinear fractional programming. *Management science*, 13(7):492–498, 1967.

[10] Levenshtein distance. Levenshtein distance — Wikipedia, the free encyclopedia, 2019. [Online; accessed 29-April-2019].

[11] Olivier Gascuel and Mike Steel. Neighbor-joining revealed. *Molecular biology and evolution*, 23(11):1997–2000, 2006.

[12] Steven Henikoff and Jorja G Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, 89(22):10915–10919, 1992.

[13] Ming Li, Xin Chen, Xin Li, Bin Ma, and Paul Vitányi. The similarity metric. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 863–872. Society for Industrial and Applied Mathematics, 2003.

[14] Andres Marzal and Enrique Vidal. Computation of normalized edit distance and applications. *IEEE transactions on pattern analysis and machine intelligence*, 15(9):926–932, 1993.

[15] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[16] Naruya Saitou and Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular biology and evolution*, 4(4):406–425, 1987.

[17] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[18] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms, Third Edition 3rd*. The MIT Press ©2009, 2009.

[19] Ziheng Yang. Maximum likelihood phylogenetic estimation from dna sequences with variable rates over sites: approximate methods. *Journal of Molecular evolution*, 39(3):306–314, 1994.

[20] Kaizhong Zhang. Similarity metric induced metrics with application in machine learning and bioinformatics. In *2016 IEEE 15th International Conference on Cognitive Informatics & Cognitive Computing (ICCI* CC)*, pages 283–287. IEEE, 2016.

[21] Zheng Zhang, Piotr Berman, and Webb Miller. Alignments without low-scoring regions. *Journal of Computational Biology*, 5(2):197–210, 1998.

[22] Zheng Zhang, Piotr Berman, Thomas Wiehe, and Webb Miller. Post-processing long pairwise alignments. *Bioinformatics*, 15(12):1012–1019, 1999.

Curriculum Vitae

Name:	Qiang Zhou
Post-Secondary Education and Degrees:	The University of Western Ontario London, Ontario, Canada 2008 - 2013 B.S.
	University of Western Ontario London, ON 2017 - 2019 M.S.
Honours and Awards:	Dean's Honor List 2016 - 2017
Related Work Experience:	Teaching Assistant The University of Western Ontario 2017 - 2019