ABSTRACT

Efficiency of Pair-Wise and Multiple Alignment Algorithms in Computational Biology

By

Man H. Vu

May 2010

DNA sequence alignment algorithms have revolutionized the way scientists study classification of species as well as genetic mutation and diseases. Due to the lengthy nature of genome sequences, which can be 2-3 billion base pairs, it is unrealistic to manually compare two such sequences. In this paper, we present various existing state-of-the-art alignment algorithms that have been applied to this problem, in particular, the N-Tuple, dynamical programming, and dot-matrix methods. The efficiency of each method to the DNA sequence alignment problem will be summarized to provide insights to the next-generation sequence alignment technology.

Efficiency of Pair-Wise and Multiple Alignment Algorithms in Computational Biology

A THESIS

Presented to the Department of Mathematics and Statistics

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Applied Mathematics

Committee Members:

Jen-Mei Chang, Ph.D
William Ziemer, Ph.D
Darin Goldstein, Ph.D

College Designee:

Robert Mena, Ph.D.

By Man H. Vu

B.S. Mathematical Science, 2008, University of California, Santa Barbara

May 2010

WE, THE UNDERSIGNED MEMBERS OF THE COMMITTEE,

HAVE APPROVED THIS THESIS

Efficiency of Pair-Wise and Multiple Alignment Algorithms in Computational Biology

By

Man H. Vu

COMMITTEE MEMBERS

Jen-Mei Chang, Ph.D	Mathematics and Statistics
William Ziemer, Ph.D	Mathematics and Statistics
Darin Goldstein, Ph.D	Computer Science and Computer Engineering

ACCEPTED AND APPROVED ON BEHALF OF THE UNIVERSITY

Robert Mena, Ph.D.
Department Chair, Department of Mathematics and Statistics

California State University, Long Beach

May 2010

ACKNOWLEDGEMENTS

To thank Dr. Jen-Mei Chang, thank you for your guidance and support throughout my stay at CSULB. Your enthusiasm for research has given me the opportunity to experience hands on research and put me one step closer towards my goals. Without you, my experience at CSULB would not have been the same. To Dr. William Ziemer, Dr. Darin Goldstein, and Dr. Claudia Rangel-Escareno thank you for investing your time and effort to help me complete this thesis. Lastly, thank you to my classmates at CSULB. I had a wonderful time laughing, studying, and dining with all of you. You all truely made my experience at CSULB mean so much more than just a master's degree.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iii

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

1. INTRODUCTION 1

2. THE WORD METHOD 5

Method 8
Computational Efficiency 11

3. DYNAMICAL PROGRAMMING 12

Global Alignment 14
Local Alignment 17
Parallel Implementations 22

4. DOT-MATRIX METHOD 28

Method 29
Summary 35

5. IMPLEMENTATIONS AND RESULTS 37

N-Tuple 39
Global Alignment 44
Dot-Matrix 46
Summary 48

6. CONCLUSION 49

APPENDIX 52

A. MATLAB CODES 53

BIBLIOGRAPHY 66

LIST OF TABLES

TABLE Page

1 The five sequences used to simulate a database search 38

2 The maximal N of Q versus each reference sequence 40

3 Results of the N-Tuple Method with various N's 41

4 Results of the GAM algorithm..... 44

5 A list of all the chosen fragments..... 47

LIST OF FIGURES

FIGURE		Page
1	(a) S_2 is held fixed while Q is shifted. (b) Dot matrix of S_2 and Q ...	10
2	The weighted similarity matrix of X_2 and Y_2 using GAM	17
3	The weighted similarity matrix of X_3 and Y_3 using LAM	19
4	Comparing GAM and LAM	21
5	A $U_{9\times 19}$ matrix with the first column and row shaded in gray	22
6	Matrix schematics of the Wozniak approach	24
7	Matrix schematics of the Rognes and Seeberg approach	25
8	Matrix schematics of the Striped Smith-Waterman	26
9	The family of dot-matrices for Example 4.3	31
10	A tree structure constructed from a consistent family	34
11	The dot plots of Q versus all reference sequences.....	42
12	A zoomed dot plot of Q versus all the reference sequences	43
13	A segment of the original five sequences and their global alignments..	45
14	A segment of the original five sequences and the aligned fragments ...	47

CHAPTER 1

INTRODUCTION

Deoxyribonucleic acid (DNA) is regarded as genetic fingerprints. Not only does DNA serve as an individual marker, it also controls physical characteristics of a living specimen. One DNA sequence contains various combinations of nucleotides: Adenine (A), Guanine (G), Thymine (T), and Cytosine (C). These sequences can range up to 2-3 billion base pairs [1]. With the advent of modern computing machines, DNA sequence alignment analysis have become more feasible and have proven useful in situations like a crime scene investigation. With certain pieces of evidence, such as a blood stain or a strand of hair, a DNA sequence can be recovered. This sequence will be used to search in a database and identify the person who was involved in the crime. In other applications, DNA sequence alignment analysis is helping to further the study of genetic mutations in age-related diseases, such as cancer, and revolutionized the study of phylogeny [2]. However, most applications of DNA sequence alignment analysis requires searching through an entire database for a match. The lengthy nature of DNA sequences combined with the doubling of GenBank/EMBL/DDBJ every 15 months produced a demand for a better, faster and a more robust DNA alignment method [3].

DNA research dated back to the late 1800's. Today's DNA sequence alignment analysis was made possible by the development of the Sanger Method in 1977, which allows the extraction of a DNA sequence from a DNA sample [4]. Since then, the multiplying collection of DNA sequences called for a centralized database which led to the development of GenBank in 1982 [1]. The vast database required efficient search methods which led to today's DNA sequence alignment analysis. Modern DNA sequence alignment softwares such as FASTA and BLAST made it possible to efficiently align and search through a database [1]. These softwares rely on a combination of DNA sequencing methods such as the N-Tuple

method, the Dynamical Programming method, and the Dot-Matrix method, which will be discussed in Chapters 2, 3, and 4, respectively.

In this paper, several key definitions will be used. We start by formally defining a DNA sequence in Definition 1.1.

Definition 1.1. A *DNA sequence*, $\{x_i\}_{i=1}^m$, of length m is such that $X = \{x_1, x_2, x_3, ..., x_m\}$ with $x_i \in \{A, G, T, C\}$ for $i = 1, 2, 3..., m$.

Each sequence is comprised of element(s) or residue(s) as defined in Definition 1.2.

Definition 1.2. Let $\{x_i\}_{i=1}^m$ be a DNA sequence, then any $x_k \in \{x_i\}_{i=1}^m$ for any $k \in \{1, 2, 3, ..., m\}$ is called an *element* or a *residue*.

In a database search, a *reference* sequence is one of many sequences in the database while a *query* sequence is the sequence that is being compared to a reference sequence. A DNA sequence can be broken up into different segments as defined in Definition 1.3.

Definition 1.3. Let $\{x_i\}_{i=1}^m$ be a DNA sequence. A *segment*, $\{x_j\}_{j=k}^n$, of $\{x_i\}_{i=1}^m$ is a subsequence such that $[k, n] \subset [i, m]$ where $k \in \{1, 2, ..., m\}$, $k < n$, and $i, j, k, m \in \mathbb{N}$.

Note that a segment is a subsequence in which the elements of the segment is in the same order as that of the main sequence. For example, if we take any sequence {A, **T**, **T**, **C**, **G**, **G**, C, A }, one possible segment is {T,T,C,G,G}. Since we are interested in comparing sequences, we must start by comparing each individual element of the sequences. In the comparison process, we categorize two elements as a match or a mismatch according to Definition 1.4.

Definition 1.4. Let $\{x_i\}_{i=1}^m$ and $\{y_j\}_{j=1}^n$ be two DNA sequences. A *match* occurs when $x_i = y_j$ for some $i = 1, 2, ..., m$ and $j = 1, 2, ..., n$. Otherwise the two elements are considered to be a *mismatch*, that is $x_i \neq y_j$.

For the remainder of this paper, we will discuss the three DNA alignment techniques and give an assessment of their computational efficiency. First we will discuss the most basic technique, the N-tuple method, also known as the Word method. Second, the Dynamical Programming method will be presented along with three parallel methods. Third, a specific use of the dot-matrix will be discussed in regards to multiple alignments. We will test these three methods using five different strands of the Influenza A virus. A discussion section will follow to give an overall comparison of the methods presented. In the final chapter, we will wrap up all the main ideas that are discussed in this paper.

CHAPTER 2

THE WORD METHOD

The N-Tuple method, also known as the Word method, searches for the best alignment between two sequences that satisfies the threshold N, where N represents the number of consecutive matches [5]. For example, given two sequences $Q = \{ACTCGGT\}$, short for query, and $S_1 = \{CT\boldsymbol{TCGG}C\}$ and $N = 4$. If Q and S_1 were to be aligned from their initial position, then we can deduce that these two sequences are highly related since the four consecutive matches, namely T, C, G and G, satisfy the given threshold criterion. However, the threshold will not always be satisfied when we align the sequences one directly on top of the other. To demonstrate, we take $S_2 = \{\boldsymbol{AC}ATCGG\}$ and align it directly on top of Q. We see that the number of consecutive matches is two, namely A and C, which does not satisfy the prescribed N. Intuitively, we can see that Q and S_2 are a match since the segment $TCGG$ appears in both sequences. Our alignment failed because an *insertion* of the element A in the third position of S_2 caused the remaining elements to shift one place to the right. Similarly, a *deletion* of an element will cause all the subsequent elements to shift one place to the left.

To find the best alignment position, we hold one sequence fixed while shifting the other sequence along the lengths of the fixed sequence. This process behaves like a filter, where at each position of the sequence, we get a score that describes the similarity between the two sequences. This will allow us to see the best number of consecutive matches at every position. The position in which they best match might not be right on top of each other, as previously seen when aligning Q and S_2. We can represent this convolution with a *similarity matrix* defined in Equation (2.1),

$$S_{i,j} = \begin{cases} 1, & x_i = y_j \\ 0, & x_i \neq y_j, \end{cases} \tag{2.1}$$

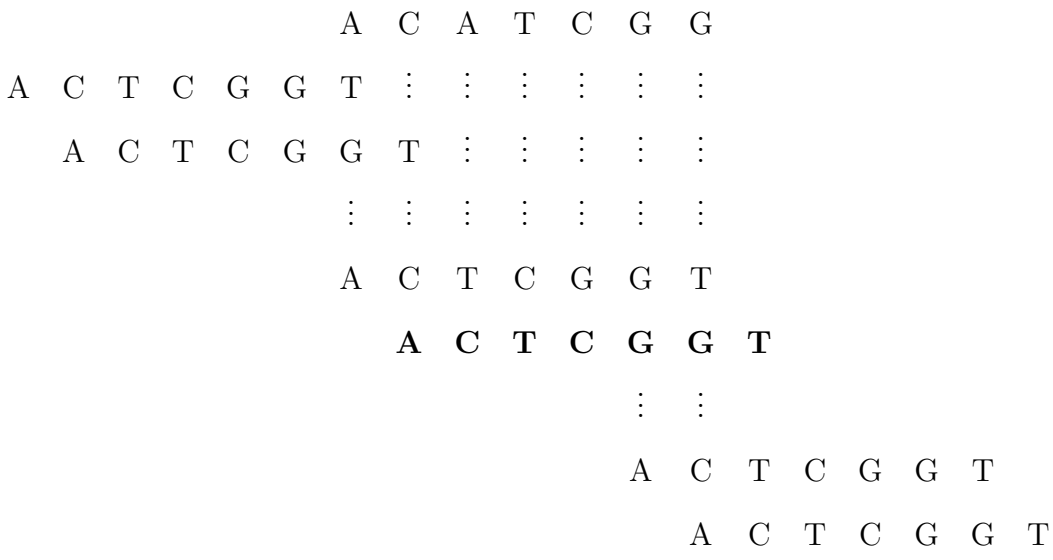
where $\{x_i\}_{i=1}^m$ and $\{y_i\}_{i=1}^n$ are two given sequences. The similarity matrix allows us to see whether an insertion or deletion has occurred with a string of consecutive ones off the main diagonal. Whereas, a direct alignment will be signaled by a string of consecutive ones along the main diagonal.

Method

In Figure 1(a) we illustrate the idea of convolving one sequence with another. In this case, S_2 is held fixed while Q is being shifted for the entire length of S_2. For discussion purposes, the main diagonal will be the diagonal zero, each diagonal above the main diagonal will be numbered in increments of one, and below the main diagonal will be numbered in decrements of negative one. In the first position, we can only compare the last element, T, of Q, with the first element, A, of S_2. The comparison resulted in a mismatch which is denoted with a zero in diagonal negative six in Figure 1(b). Similarly, in the second position we compare the last two elements of Q with the first two elements of S_2. The result is seen in diagonal negative five. The direct alignment is seen in the main diagonal with only the first two elements, A and C, of each sequence matching. Recall the insertion of the element A in the third position of Q caused the remaining elements to shift to the right. We see that an insertion has occurred since the best alignment, with four consecutive matches, lies on diagonal one, off the main diagonal. In short, given N, we can use the similarity matrix to find any diagonal in which N is satisfied. If N is satisfied, we can conclude that the two sequences are related.

When comparing extremely lengthy sequences, which is often the case, N will be set at a high value in order to say that the sequences are related. If, given two sequences of 1000 elements, then an N value of 10 cannot be used alone to indicate a strong relationship. On the other hand, setting N as an extremely high value will cause method to fail. Take the previous sequence $Q = A\boldsymbol{CT}C\boldsymbol{GG}T$ and compare it with the sequence $S_4 = C\boldsymbol{CT}T\boldsymbol{GG}A$. With $N = 4$, this method concludes that the sequences are not related because there are no four consecutive matches. However, we see that four out of the seven elements match at the exact location, namely the segments CT and GG, but not consecutively. Therefore, N

should be selected empirically and relative to the length of the sequence in question, Q. Additionally, prior experimentation should give the users a better idea on how to choose an appropriate N. There are situations like this where one would want to classify these two sequences. Knuth [6] suggested to not only set a threshold N, but also set the number of discontinuation. The discontinuation separates two matched segments with a number of mismatches in between. Going back to the example with Q and S_4, the discontinuation occurs after the segment CT then followed by the segment GG. With a discontinuation count of one, we see that $N = 4$ is satisfied since CT gives two consecutive matches followed by one discontinuation and two consecutive matches, GG. Because N is satisfied, we conclude that the sequences are related. Therefore, with a prescribed discontinuation condition and an N value, the method can execute more accurately.



(a)

	A	C	A	T	C	G	G
A	1	0	1	0	0	0	0
C	0	1	0	0	1	0	0
T	0	0	0	1	0	0	0
C	0	1	0	0	1	0	0
G	0	0	0	0	0	1	1
G	0	0	0	0	0	1	1
T	0	0	0	1	0	0	0

(b)

FIGURE 1: (a) $S_2 = \{ACATCGG\}$ held fixed while $Q = \{ACTCGGT\}$ is shifted. The optimal alignment is shown in bold with four consecutive matches. (b) Similarity matrix of Q and S_2 with Q listed along the side and S_2 listed along the top. The first column and row where the sequences are listed are only for the ease of comparison of two sequences and are not a part of the similarity matrix.

Computational Efficiency

The computer implementation of the N-tuple method is much less intricate than many other DNA sequencing methods such as the Dynamical Programming approach and the dot-matrix approach to be discussed. Computationally, this method executes on the order of $O(mn)$ where m and n are the lengths of the two sequences. To quickly execute this method, the entire similarity matrix will not need to be stored, we only need to check whether or not N is satisfied. Therefore, mn operations will rarely be reached. On the other hand, this method generally lacks accuracy because the alignment results from *only* one diagonal. The inaccuracy can be seen in Figure 1(b) when aligning S_2 and Q. Both sequences contain the segment AC in the first and second element. However, the resulting optimal alignment does not align AC because the most consecutive matches are on diagonal one. This method is best used as a filtration method, filtering all the database according to the set number N. The sequences that remains after the filtration can be aligned with a more accurate method, for example, the Dynamical Programming method to be discussed.

CHAPTER 3

DYNAMICAL PROGRAMMING

The Dynamical Programming approach to DNA sequence analysis was motivated by the idea of finding the optimal alignment path between two sequences. In the N-Tuple method, the optimal path lies along any one diagonal of the similarity matrix. Unlike the N-Tuple method, the Dynamical Programming method will allow an optimal path that does not have to lie entirely along any one diagonal. First, we will discuss the construction of a scoring function that will be necessary to trace the optimal path. Once the scoring function is established, we will discuss the optimal alignment path with respect to two dynamical programming approaches. They are the global alignment algorithm and the local alignment algorithm.

Given any two sequences $X = \{x_i\}_{i=1}^{n}$ and $Y = \{y_j\}_{j=1}^{m}$, we can represent the alignment with a matrix, $A_{2\times c}$, where c is defined as $max(n, m) \leq c \leq n + m$ [7].

Example 3.1. let $X_1 = \{AATGCT\}$ and $Y_1 = \{AACT\}$. It follows that $n = 6$, $m = 4$, and $c = 6$. Here are two possible alignments

$$A_1(X_1, Y_1) = \begin{matrix} \text{A} & \text{A} & \text{T} & \text{G} & \text{C} & \text{T} \\ \text{A} & \text{A} & \text{\_} & \text{\_} & \text{C} & \text{T} \end{matrix}$$

$$A_{1'}(X_1, Y_1) = \begin{matrix} \text{A} & \text{A} & \text{T} & \text{G} & \text{C} & \text{T} \\ \text{A} & \text{A} & \text{C} & \text{\_} & \text{\_} & \text{T} \end{matrix}$$

From Example 3.1, we can see that there are many ways to align two sequences. However, we seek the optimal alignment with respect to a scoring function. The scoring function acts as a guide, assigning a positive weight to a match and a negative weight to a mismatch, insertion, and deletion. The alignment with the highest score is the optimal choice. That is, for any $x_i \in X$ there exists as a corresponding $y_i \in Y$ such that their weighted score is defined by $\sigma(x_i, y_i)$. Note that x_i or y_i can be gaps. As seen in the alignment A_1 of

Example 3.1, x_3 is aligned with a gap that occurs on y_3 of the alignment A_1. The total score for the alignment can be found by summing up all the weights along $i = 1, 2, ..., c$. Note that it is up to the user to define a specific numerical weight for each match, mismatch, and insertion and deletion. In this paper, we will use the scoring function as given in Equation (3.1).

$$\sigma(x_i, y_i) = \begin{cases} +1, & \text{for a match} \\ 0, & \text{for a mismatch} \\ -1, & \text{for a gap} \end{cases} \tag{3.1}$$

It follows that the score of the alignment A_1 is two and the score of $A_{1'}$ is one.

Sequence alignment via dynamical programming is categorized into the global alignment method (GAM) and the local alignment method (LAM). Both of these algorithms require an overall computational complexity of $O(mn)$, where m and n are the lengths of the respective sequences. The Needleman-Wunsch [8] global alignment method is mainly used in cases where two sequences are of similar lengths. If one sequence is significantly shorter than the other, the GAM will add gaps into the shorter sequence so that both will result in the same length. A further explanation of the Needleman-Wunsch global alignment algorithm is presented in the following section.

<u>Global Alignment</u>

Given two sequences $X = \{x_i\}_{i=1}^{n}$ and $Y = \{y_j\}_{j=1}^{m}$ we want to construct a similarity matrix $G_{(n+1)\times(m+1)}$. The extra row and column are needed because we are also taking into consideration an alignment with a gap. Thus, when calculating G, each sequence will hold a gap before the actual sequence begins. This notion will be further explored in the following paragraphs. To distinguish from the N-Tuple Method, the similarity matrix pertaining to the Dynamical Programming

methods will be referred to as the weighted similarity matrix. Also, for any matrix, M, $m_{i,j}$ will be used to represent the element in the ith row and the jth column while $M_{i,j}$ will be used to indicate the position (i, j) of the matrix.

Unlike the similarity matrix in the N-Tuple method, this weighted similarity matrix, G, will contain the weighted value of each $G_{i,j}$ using the scoring function defined in Equation (3.1). The first values in the first row and the first column are given by the equations

$$g_{1,j} = \sum_{k=1}^{j} \sigma(-, y_k) \tag{3.2}$$

$$g_{i,1} = \sum_{k=1}^{i} \sigma(x_k, -) \tag{3.3}$$

where the hyphen $(-)$ represents a gap. The summation of penalties along the first column and the first row is due to the continuation of a gap. In other words, as we move down the first column or across the first row from $G_{1,1}$, the sequence is adding on an insertion or deletion. The more consecutive insertions or deletions we have the less likely that it will be the optimal path. Therefore, the summation allows the negative weights to accumulate as we move to each extremes. The remaining elements of G are determined using Equation (3.4).

$$g_{i,j} = max \begin{cases} g_{i-1,j-1} + \sigma(x_{i-1}, y_{j-1}) \\ g_{i-1,j} + \sigma(x_{i-1}, -) \\ g_{i,j-1} + \sigma(-, y_{j-1}) \end{cases} \tag{3.4}$$

where $i = 2, 3, ..., n$ and $j = 2, 3, ..., m$.

Once the weighted similarity matrix is calculated, we wish to find the optimal alignment. For any $G_{i,j}$, the optimal direction is dependent upon the values $g_{i-1,j-1}, g_{i-1,j}$, and $g_{i,j-1}$. Starting from $G_{i,j}$, there are three possible movements to the next alignment position: up, left and diagonal. An upward

movement from $G_{i,j}$ to $G_{i-1,j}$ will signal x_{i-1} to be aligned with a gap, this means y_{i-1} is a gap. Similarly, a left movement from $G_{i,j}$ to $G_{i,j-1}$ is equivalent to aligning y_{j-1} with a gap, which means x_{j-1} must be a gap. Finally, a diagonal movement from $G_{i,j}$ to $G_{i-1,j-1}$ translates to an alignment between x_{i-1} and y_{j-1}. In the case where there is a tie, the diagonal movement is favorable because moving off the current diagonal will allow more gaps. To ensure the whole length of both sequences are aligned, we must start at $G_{n+1,m+1}$ and end at $G_{1,1}$. Note that the two aligned sequences will hold the same number of elements counting the gaps and that the shorter of the two sequence will hold more gap elements. We use Example 3.2 to familiarize ourselves with the concepts previously discussed.

Example 3.2. In this example, we will use the Needleman-Wunsch global alignment method to align $X_2 = \{AACTC\}$ and $Y_2 = \{AATGCT\}$. Each value in the weighted similarity matrix, except for the first row and first column, is calculated using Equation (3.4). For example, with Equation (3.4), $g_{2,2}$ will be equal to the maximum of $g_{1,1} + \sigma(x_1, y_1)$, $g_{2,1} + \sigma(-, y_1)$, and $g_{1,2} - 1 + \sigma(x_1, -)$. $g_{1,1}, g_{2,1}$ and $g_{1,2}$ are given as 0, -1, and -1 respectively. Using the Equation (3.1), $\sigma(x_1, y_1)$ is equal to one, since x_1 and y_1 are a match and anything paired with a gap has a score of negative one. Then $g_{2,2}$ is equal to the maximum of one, negative two and negative two. This yields a one for $g_{2,2}$. Similarly, $g_{2,3}$ is the maximum of $g_{1,2} + 1, g_{2,2} - 1$, and $g_{1,3} - 1$ which is equivalent to the maximum of zero, zero, and negative three. Thus, $g_{1,3}$ will be equal to zero. Using the same process, $g_{2,4}$ is the maximum of $g_{1,3} + 0, g_{2,3} - 1$, and $g_{1,4} - 1$. Here, $g_{1,3}$ is added to zero instead of one because x_1 and y_3 does not match. The resulting value of $g_{2,4}$ is negative one. The remaining values of the G is shown in Figure 2.

Next, we describe how to obtain the final alignment. Starting from the $G_{6,7}$ position, we move in the left direction since the maximum of $G_{5,6}, G_{5,7}$ and $G_{6,6}$ is

$G =$

	_	A	A	T	G	C	T
_	**0**	-1	-2	-3	-4	-5	-6
A	-1	**1**	0	-1	-2	-3	-4
A	-2	0	**2**	1	0	-1	-2
C	-3	-1	1	**2**	1	1	0
T	-4	-2	0	2	**2**	1	2
C	-5	-3	-1	1	2	**3**	**2**

$$A_2(X_2, Y_2) = \begin{matrix} A & A & C & T & C & \_ \\ A & A & T & G & C & T \end{matrix}$$

FIGURE 2: The weighted similarity matrix between $X_2 = \{AACTC\}$ and $Y_2 = \{AATGCT\}$ in Example 3.4 with the optimal path shown in bold. The resulting alignment is presented as A_2.

$G_{6,6}$ which is a three. Through the same process but now starting from $G_{6,6}$, we find that the maximum will lead us to a tie between the left and the diagonal direction. To avoid additional gaps, we pick the diagonal direction. Since the length of X_2 is less than the length of Y_2, the resulting alignment will add a gap in X_2 so that X_2 and Y_2 will be of the same length. The gap in X_2 at x_6 occurs from a left movement which we previously saw going from $G_{6,7}$ to $G_{6,6}$. The resulting alignment is shown in bold in Figure 2.

Local Alignment

Like the Needleman-Wunsch global alignment, the Smith-Waterman local alignment uses the same scoring function, $\sigma(x_i, y_i)$, and the weighted similarity matrix idea [9]. The main use of this method is to align a short sequence to a much longer sequence. However, instead of accumulating the gap penalties along the first

row and the first column of the weighted similarity matrix, these values will be set to zero. The remaining entries of the weighted similarity matrix can be calculated using Equation (3.5) [9]. First, we will familiarize ourselves with sequence alignment using the LAM, then we will further explore why the zero reset is used.

$$l_{i,j} = max \begin{cases} l_{i-1,j-1} + \sigma(x_{i-1}, y_{j-1}) \\ l_{i-1,j} + \sigma(x_{i-1}, -) \\ l_{i,j-1} + \sigma(-, y_{j-1}) \\ 0 \end{cases} \tag{3.5}$$

where $i = 2, 3, ..., n$ and $j = 2, 3, ..., m$.

Once the weighted similarity is calculated, the alignment path can be found by starting from the maximum value of L that occurs on the last row or column. The movement from one cell to the next is same as discussed in the GAM section. To ensure the entire alignment of the shorter sequence onto the longer sequence, the alignment path must start in the last row or column and end in the first row or column, respectively. Note that the maximum $l_{i,j}$ will occur on last row or column depending on whether the user places the shorter sequence on the row or the column.

Example 3.3. Using the sequence $X_3 = \{TGTT\}$ and $Y_3 = \{AATGCTTCTG\}$, the calculated weighted similarity matrix using the LAM is as follows in Figure 3. Figure 3 shows an alignment using the LAM method. The calculation of each $l_{i,j}$ is much like the ones calculated in Example 3.2, but instead we will use Equation (3.5). For example, to calculate $l_{2,2}$, we must find the maximum between zero, $l_{2,1} - 1$, $l_{1,2} - 1$, and $l_{1,1} - \sigma(x_1, y_1)$. This is equivalent to finding the maximum between zero and negative one, which will yield $l_{2,2} = 0$. The alignment path begins at the maximum value $l_{5,7}$ and ends at $l_{1,3}$. Notice that the maximums occur on the last row because the shorter sequence is listed along the column. If

$L =$

	_	A	A	T	G	C	T	T	C	T	G
_	0	0	**0**	0	0	0	0	0	0	0	0
T	0	0	0	**1**	0	0	1	1	0	1	0
G	0	0	0	0	**2**	1	0	1	1	0	2
T	0	0	0	1	1	**2**	2	1	1	2	1
T	0	0	0	1	1	1	**3**	3	2	2	2

$$A_3(X_3, Y_3) = \begin{array}{cccccccccc} \text{-} & \text{-} & \text{T} & \text{G} & \text{T} & \text{T} & \text{-} & \text{-} & \text{-} & \text{-} \\ \text{A} & \text{A} & \text{T} & \text{G} & \text{C} & \text{T} & \text{T} & \text{C} & \text{T} & \text{G} \end{array}$$

FIGURE 3: The weighted similarity matrix between two sequences, $X_3 = \{TGTT\}$ and $Y_3 = \{AATGCTTCTG\}$, from Example 3.3 with the optimal patch shown in bold. The resulting alignment is presented as A_3.

the shorter sequence is listed along the rows then we can expect the maximums to occur on the last column. All movements in the alignment path is in the diagonal direction marked in bold and follows the exact procedure as the GAM. The resulting alignment is given with A_3.

Unlike the GAM, The LAM allows the score to reset to zero if the score is negative. We must keep in mind that when using the LAM, two sequences that we wish to align typically differ greatly in length. The inclusion of the zero in Equation (3.5) will allow negative scores to reset to zero in order to keep the best possible alignments positive. If we allow large negative values to accumulate in the first row and first column, possible alignments that occur later on the longer sequence will be impossible to detect. We will demonstrate this in Figure 4, using both LAM and GAM. In Figure 4, two sequences of different lengths are being compared. The sequence $\{\mathbf{AGCC}TGTTGT\mathbf{AGCCT}\}$ was created with the

intention of having the other sequence, $\{AGCCT\}$, match at the end. Additionally, a lesser match was placed at the beginning.

In Figure 4(a) we see that the LAM was able to detect the optimal alignment which occurred towards the right end of the weighted similarity matrix. However, when looking at Figure 4(b), the GAM was unable to detect the optimal alignment so it settled for the second best alignment. The GAM was unable to see the optimal alignment that occurred towards the right end of the weighted similarity matrix because of the large negative values that accumulated. Therefore, we conclude that when comparing sequences that differ greatly in length the LAM will be superior to the GAM.

	_	A	G	C	C	G	G	T	T	G	T	A	G	C	C	T
_	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0
A	0	1	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0
G	0	0	2	1	0	1	1	0	0	1	0	0	**2**	1	0	0
C	0	0	1	3	2	1	1	1	0	0	1	0	1	**3**	2	1
C	0	0	0	2	4	3	2	1	1	0	0	1	0	2	**4**	3
T	0	0	0	1	3	4	3	3	2	1	1	0	1	1	3	**5**

(a) LAM

	_	A	G	C	C	T	G	T	T	G	T	A	G	C	C	T
_	**0**	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15
A	-1	**1**	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13
G	-2	0	**2**	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11
C	-3	-1	1	**3**	2	1	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
C	-4	-2	0	2	**4**	3	2	1	0	-1	-2	-3	-4	-5	-6	-7
T	-5	-3	-1	1	3	**4**	3	3	2	1	0	-1	-2	-3	-4	-5

(b) GAM

FIGURE 4: (a) The weighted similarity matrix in the LAM. (b) The weighted similarity matrix in the GAM. Optimal alignments are shown in bold. For the same two sequences, The LAM picked up an optimal alignment with a score of five while the GAM picked an optimal alignment with a score of four.

Parallel Implementations

As stated before, the GAM and the LAM carries a computational complexity of $O(mn)$. When dealing with sequences in the billions, computational complexity grows exceedingly large. To reduce the computational time, parallel approaches were developed. These parallel algorithms aim at distributing the calculation of the weighted similarity matrix over many processors. Three parallel approaches will be discussed in this section.

Let the weighted similarity matrix in the LAM or GAM be denoted by $U_{m+1\times n+1}$. Further, let $V_{m\times n}$ be the submatrix of U which excludes $u_{1,n+1}$ and $u_{m+1,1}$ as shown in Figure 5. This is equivalent to setting $v_{i,j}$ equal to $u_{i+1,j+1}$, where $i = 1, 2, 3, ..., m, j = 1, 2, 3, ..., n$. For simplicity, sometimes V will be used instead of U.

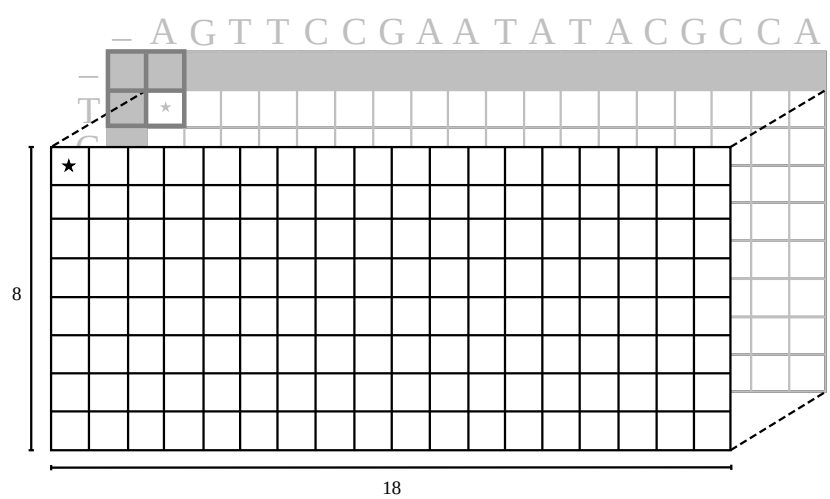


FIGURE 5: A $U_{9\times 19}$ matrix with the first column and row shaded in gray. The unshaded portion represents the matrix $V_{8\times 18}$. The data dependency for the value $\star$ is shown with bold lining.

Wozniak Approach

Recall the construction of the weighted similarity matrix from the previous section. From Figure 5, we notice that once $u_{2,2}$ is found, two other values, $u_{2,3}$ and $u_{3,2}$ can be calculated simultaneously. Note that $u_{2,2}$ is equivalent to $v_{1,1}$. Wozniak [10] used this data dependency pattern to distribute the computation amongst different processors and assigned one processor to each diagonal.

As in the Figure 6, the first processor, P1, starts at the $v_{1,1}$ position. Once this value is found, the information will be passed to the second processor, P2. With this updated information, P2 can calculate $v_{1,2}$ and then $v_{2,1}$. Instead of waiting for P2 to complete the assigned diagonal, the value of $v_{1,2}$ can be passed to a third processor, P3, immediately after it is calculated. This allows the calculation of both, $v_{1,3}$ and $v_{2,1}$ to occur simultaneously on separate processors P3 and P2, respectively. Once one processor completes the assigned diagonal, it will be directed to the next diagonal that has not yet been calculated. It is important to note that when calculating values like $v_{3,3}$, which is on the diagonal assigned to P3, P3 is required to get information from two other processors, namely P2 and P1. Similarly, calculating the rest of the entries in V will require two data transfers from two other processors.

This implementation was tested on the Sun Ultra Sparc using 12 processors each running at 167MHz. The results yield a speedup of over two folds compared to the original method running on a single processor of the same machine [10]. Though the results showed an total time improvement, the dependency of two data transfers from two separate processors delayed the overall process, because data transfer also require time.

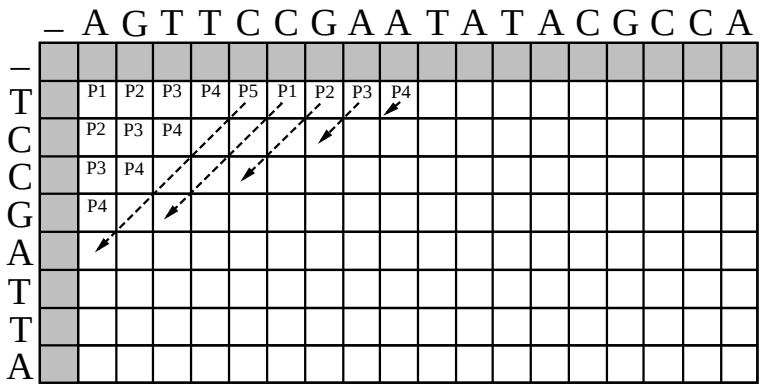


FIGURE 6: A matrix, $U_{9\times19}$ or $V_{8\times18}$, with P1-P5 representing the different processors. The arrows shows the subsequent pattern in which one computer will start right after the other in a diagonal fashion.

Rognes and Seeberg Approach

Another improvement to the previous method was presented by Rognes and Seeberg [11] using a similar approach. Each processor will be assigned a column instead of a diagonal. In Figure 7, P1 is assigned to the first column of V. Immediately after P1 calculates $v_{1,1}$, the information is passed to P2. This allows the simultaneous calculation of $v_{2,1}$ and $v_{1,2}$ by processors P1 and P2, respectively. The same pattern will continue for the remaining elements. Notice this method only requires one data transfer from another processor per matrix entry as compared to the two data transfers in the previous method. The decrease in data transfers allows this method to out perform the previous method. This algorithm ran on eight 8-bit Intel Pentium III processors running at 500Mhz and resulted in a six-fold speedup over the most up-to-date implementation running on the same hardware [11].

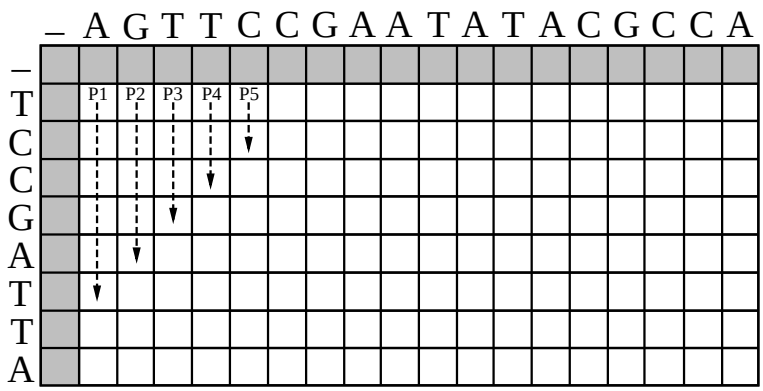


FIGURE 7: The downward arrows show the subsequent pattern in which processors, P1-P5, will start right after the other. P1 will calculate $v_{1,1}$ then pass the information to P2. This allows simultaneous calculation of $v_{2,1}$ and $v_{1,2}$ by P1 and P2, respectively. This pattern is continued with P3, P4, and P5.

Striped Smith-Waterman

The Striped Smith-Waterman [12] algorithm is the latest and fastest known implementation of the Dynamical Programming method. The algorithm requires splitting the processors evenly along any one row or column. Given a matrix, $V_{m\times n}$, with P_i processors, for $i = 1, 2, ..., k$, split along the rows. The resulting division assigns P_i to elements $v_{m,(\frac{n}{k})*(i-1)+1}$ to $v_{m,(\frac{n}{k})*i}$. Once the first row is finished, the processors will proceed to the second row. In effect, the matrix is being divided into evenly sized sub-matrices, with size $m \times \frac{n}{k}$, and are assigned a specific processor.

The advantage of this method is that it reduces the number of data transfers per entry calculation. However, recall that the calculation of $u_{i,j}$ requires $u_{i-1,j}$, $u_{i,j-1}$, and $u_{i-1,j-1}$. To make the division of processors along any one row or column possible, the algorithm forces the calculation of $u_{i,j}$ to be dependent upon $u_{i-1,j-1}$ and $u_{i-1,j}$ only. This will allow errors in the calculations with a two-third

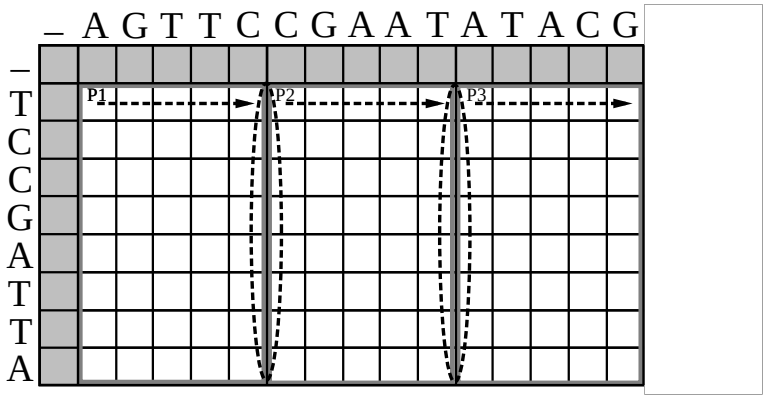


FIGURE 8: A matrix, $U_{9\times16}$ or $V_{8\times15}$, with P1-P3 representing the different processors. The arrows shows the how the striped algorithm [12] is processed. The gray bolded lining represents the different submatrices and the dotted encircling shows the error-prone areas.

chance of getting the calculation right. In the case that the calculation is wrong, several processors will be assigned to correct these errors only after $u_{i,j-1}$ is obtained. These processors will only check areas that are prone to errors, namely the entries $u_{m,(\frac{n}{k})*i}$, where $i = 2, 3, ..., k-1$. The checking process continues from $u_{m,(\frac{n}{k})*i}$ until a set threshold of consecutive correct values are encountered. Note that at each error-prone area, the probability of getting the value wrong is one-third. The probability of getting consecutive wrong values is decreased exponentially. Therefore, it is rarely the case that the correction process will have to correct the entire segment that the processors have been assigned.

Figure 8 shows the schematics of the weighted similarity matrix according to the Striped Smith-Waterman method [12]. The figure presents a $U_{9\times16}$, or $V_{8\times15}$, matrix with the first row of V split amongst three processors, P1-P3. On the first row of V, P1 calculates elements $v_{1,1}$ through $v_{1,5}$, P2 calculates elements $v_{1,6}$ through $v_{1,10}$, and P3 calculates elements $v_{1,11}$ through $v_{1,15}$. The elements $v_{1,6}$

and $v_{1,11}$ are calculated using only $v_{0,6}$ and $v_{0,11}$, or more properly, $u_{1,7}$ and $u_{1,12}$. These elements will be checked once $v_{1,5}$ and $v_{1,10}$ are available. If the elements $v_{1,6}$ and $v_{1,11}$ are correct, the checking process terminates and waits to check the next row. If $v_{1,6}$ and $v_{1,11}$ are incorrect, the wrong value will be replaced and several more values after that will be checked, i.e. $v_{1,7}$, $v_{1,8}$ and similarly $v_{1,12}$, $v_{1,13}$. This division allows P1-P3 to only calculate the assigned submatrix, which results in less data transfers. The only data transfers that are needed occur around the error-prone areas.

The decrease in amount the of data transfers resulted in a faster algorithm than the previous algorithms. The Striped method was implemented using 16 8-bit Intel Xeon Core 2 Duo processors running at 2.0 GHz. The algorithm showed two to eight times improvement over the most updated parallel methods [12].

CHAPTER 4

DOT-MATRIX METHOD

Biologically, DNA can undergo mutations as time progresses due to exposure to harmful chemicals or old age. Our goal is to find the segment(s) in which the mutation occurs. For example, take a patient diagnosed with cancer, if three sequences of pre-mutation and one sequence of post-mutation are given, we can align all four simultaneously in order to determine the similarities and the differences. In this case, the similarities will represent the normal genes and the differences will represent the cancerous genes. To execute such alignments, we will use a multiple sequence alignment method via similarity matrices, also known as dot-matrices. In this chapter, we will discuss a method by Vingron and Argos [13] which provides a method for filtering a series of similarity matrices.

Method

Given N sequences, we will denote the dot-matrix between s and t by $R_{s,t}$. To start the filtering process, a family of dot-matrices, defined in Definition 4.1, will be constructed.

Definition 4.1. Given N sequences, a *family* of dot-matrices contains all possible pairwise combination of similarity matrices and is given by,
$\Re_N = \{R_{1,2}, R_{1,3}, ..., R_{1,N}, R_{2,3}, R_{2,4}, ..., R_{2,N}, R_{N-1,N}\}$, containing a total of $\frac{N(N-1)}{2}$ elements.

This construction avoids the possibility of creating a dot-matrix of one sequence to itself, $R_{s,s}$. Additionally, if we have $R_{s,t}$, we can also avoid creating $R_{t,s}$ since $R_{t,s}$ is equal to the transpose of $R_{s,t}$. For example, if four sequences are given we can expect the family of dot-matrices to contain $R_{1,2}$, $R_{1,3}$, $R_{1,4}$, $R_{2,3}$, $R_{2,4}$, and $R_{3,4}$ for a total of six elements.

Using these dot-matrices, we can confirm the match at any $(R_{s,t})_{i,j}$ with a product of two other dot-matrices, such as $R_{s,u} \bullet R_{u,t}$.

Definition 4.2. Let $R_{s,u}$ and $R_{u,t}$ be dot-matrices of sequences s, t, and u each

with lengths l, m, and n. The *product* of two dot-matrices, $R_{s,u} \bullet R_{u,t}$, is given by $(R_{s,u} \bullet R_{u,t})_{i,j} = \bigvee_{k=1}^{m} ((R_{s,u})_{i,k} \wedge (R_{u,t})_{k,j})$ where $i = 1, 2, ..., l$ and $j = 1, 2, ..., m$. Here, $\wedge$ and $\vee$ represent the *and* and the *or* operator, respectively. All possible combinations using are as follows: $(1 \wedge 1) = 1$, $(1 \wedge 0) = 0$, $(0 \wedge 0) = 0$, $(1 \vee 1) = 1$, $(1 \vee 0) = 1$, and $(0 \vee 0) = 0$.

In particular, if $(R_{s,t})_{i,j} = (R_{s,u} \bullet R_{u,t})_{i,j}$ with $u \neq s, t$, then the match at the $(i, j)^{th}$ position is *confirmed*. Note that Definition 4.2 is much like the usual matrix multiplication but with multiplication replaced by an *and* ($\wedge$) operator and addition replaced by an *or* ($\vee$) operator.

Example 4.3. Let $S_1 = \{ATTCG\}$, $S_2 = \{ATCCG\}$, and $S_3 = \{ATTCA\}$, we want to find the corresponding family of dot-matrices which contain three elements, namely, $\Re_3 = \{R_{1,2}, R_{1,3}, R_{2,3}\}$. The family is shown in Figure 9. The product of $R_{1,2}$ and $R_{2,3}$ is then calculated using Definition 4.2. For example, take the second row of $R_{1,2}$ and multiply ($\wedge$) that by the second column of $R_{2,3}$ to yield $(R_{1,2} \bullet R_{2,3})_{2,2}$. That is, $(0\ 1\ 0\ 0\ 0) \wedge (0\ 1\ 0\ 0\ 0)^T$ will yield $(0 \wedge 0) \vee (1 \wedge 1) \vee (0 \wedge 0) \vee (0 \wedge 0) \vee (0 \wedge 0)$. After simplifying, we get $(0 \vee 1 \vee 0 \vee 0 \vee 0)$, which will result in $(R_{1,2} \bullet R_{2,3})_{2,2} = 1$. The resulting product is used to confirm the matches that exist in $R_{1,3}$ with that of $R_{1,2} \bullet R_{2,3}$. In this case, all matches are confirmed.

An *inconsistency* occurs when a match is present in $(R_{s,t})_{i,j}$ but is not present in $(R_{s,u} \bullet R_{u,t})_{i,j}$. In other words, the ith residue of s and t agree but it is different from that in u. On the other hand, if a match is present in both $(R_{s,t})_{i,j}$ and $(R_{s,u} \bullet R_{u,t})_{i,j}$ then $R_{s,t}$ is said to be *consistent*. To enforce consistency, we require the family of dot-matrices to satisfy Equation (4.1).

$$R_{s,t} \subseteq R_{s,u} \bullet R_{u,t} \quad \forall u \neq s, t \tag{4.1}$$

$R_{1,2} =$

	A	T	C	C	G
A	1	0	0	0	0
T	0	1	0	0	0
T	0	1	0	0	0
C	0	0	1	1	0
G	0	0	0	0	1

$R_{1,3} =$

	A	T	T	C	A
A	1	0	0	0	1
T	0	1	1	0	0
T	0	1	1	0	0
C	0	0	0	1	0
G	0	0	0	0	0

$R_{2,3} =$

	A	T	T	C	A
A	1	0	0	0	1
T	0	1	1	0	0
C	0	0	0	1	0
C	0	0	0	1	0
G	0	0	0	0	0

$R_{1,2} \bullet R_{2,3} =$

1	0	0	0	1
0	1	1	0	0
0	1	1	0	0
0	0	0	1	0
0	0	0	0	0

FIGURE 9: The family of dot-matrices for Example 4.3 and a matrix multiplication is presented.

Note that when calculating subsets, we are only concerned with the matches, the ones, not the zeros. That is, given (1 1 0 1) and (1 0 0 1), we can say that (1 0 0 1) is a subset of (1 1 0 1). Next, A consistent family of dot-matrices can be obtained through a filtering of all inconsistencies via Equation (4.2).

$$R_{s,t}^{(n+1)} = R_{s,t}^{(n)} \cap \bigcap_{\forall u \neq s,t} (R_{s,u}^{(n)} \bullet R_{u,t}^{(n)}) \tag{4.2}$$

Much like the notion of subsets, the intersection operates on the ones, not the

zeros. That is, if we intersect (1 1 0 1) with (1 0 0 1) then the solution will be (1 0 0 1).

The filtering process according to Equation (4.2) allows very little flexibility. That is, given five sequences, at the ith position, all five sequences must agree. Due to different mutations such as insertions and deletions, it is rare that all sequences will agree at the same residue. To fix this problem, the filtration process will follow Equation (4.3).

$$(R_{s,t}^{(n+1)})_{i,j} = 1 \Leftrightarrow C \leq (R_{s,t}^{(n)})_{i,j} + (\sum_{\forall u \neq s,t} R_{s,u}^{(n)} \bullet R_{u,t}^{(n)})_{i,j}, \tag{4.3}$$

where C is a threshold set by the user. For example, given five sequences, if three out of the five sequences match at the same residue then we can accept this position as match. Therefore, in this case, C should be set to three.

The iterative filtering process will update all $R_{s,t}$ in the family to ensure all inconsistencies are removed, this is demonstrated in Example 4.4. The iterative process will eventually converge to a consistent family. That is, $\Re_N^{(n)}$ will equal $\Re_N^{(n+1)}$ if and only if $\Re_N^{(n)}$ is consistent [13].

Example 4.4. Given four sequences, $\{S_1, S_2, S_3, S_4\}$. A family, $\Re_4^{(1)} = \{R_{1,2}^{(1)}, R_{1,3}^{(1)}, R_{1,4}^{(1)}, R_{2,3}^{(1)}, R_{2,4}^{(1)}, R_{3,4}^{(1)}\}$, is constructed. Following Equation (4.2), the next update will be $R_{1,2}^{(2)} = R_{1,2}^{(1)} \cap (\bigcap\{R_{1,3}^{(1)} \bullet R_{3,2}^{(1)}, R_{1,4}^{(1)} \bullet R_{4,2}^{(1)}\})$ and, similarly, $R_{1,3}^{(2)} = R_{1,3}^{(1)} \cap (\bigcap\{R_{1,2}^{(1)} \bullet R_{2,3}^{(1)}, R_{1,4}^{(1)} \bullet R_{4,3}^{(1)}\})$. The same process is conducted for the remaining elements to find $R_{1,4}^{(2)}, R_{2,3}^{(2)}, R_{2,4}^{(2)}$, and $R_{3,4}^{(2)}$. Once all $R_{s,t}^{(2)}$ are found, the new family will be denoted, $\Re_4^{(2)}$. The iterations will proceed until $\Re_4^{(n)} = \Re_4^{(n+1)}$.

Once a consistent family is established, we can recover an alignment of all the similarities using a method presented by Vingron and Argos in 1989 [14]. We start the alignment process by choosing a subset from the consistent family such that all sequences are used. That is, the subset must be ordered as in

Equation (4.4).

$$\{R_{1,2}^{(n)}, R_{2,3}^{(n)}, R_{3,4}^{(n)}, ..., R_{N-1,N}^{(n)}\} \tag{4.4}$$

Using the sequences from Example 4.4, the subset of the family as described in Equation (4.4) will be $\{R_{1,2}^{(n)}, R_{2,3}^{(n)}, R_{3,4}^{(n)}\}$, where n is the iteration in which consistency is reached.

Next, a tree-like organization will be created starting from all the matches in the first element, $R_{1,2}$, of the subset and ending with the last element, $R_{N-1,N}$, of the subset. That is, for a match occurring at $(R_{1,2})_{i,j}$. This corresponds to aligning the ith residue of Sequence 1 with the jth residue of Sequence 2. Subsequently, with a corresponding match at $(R_{2,3})_{j,k}$, the jth residue of Sequence 2 will be matched to the kth residue of Sequence 3. The indexes, i, j, k, and so on, will be stored in a vector which will be called a *fragment* and each element of the fragment will be called a *node*.

In Figure 10, using the dot-matrices $R_{1,2}$ and $R_{2,3}$, the positions of the matches with respect to Sequence 1 is listed, they are 1, 3, and 6. The matches occurring at 1, 3, and 6 on Sequence 1 corresponds to residue number 1, 5, and 6 on Sequence 2, respectively. Now using $R_{2,3}$ and residues 1, 5, and 6 of Sequence 2, this corresponds to residues 1, 5, and 6 on Sequence 3. From this tree we can list all possible fragments from left to right: (1, 1, 1), (3, 1, 1), (3, 5, 5), and (6, 6, 6).

Not all fragments will be used in the alignment process. To choose which fragments to use, we proceed from left to right and attach a higher score to each node that continues the same residue position; from node 1 in Sequence 1 to node 1 in Sequence 2. A lower score will be assigned to a node that changes to another residue position; node 3 of Sequence 1 to node 5 of Sequence 2. For example, if a score of one will be assigned to a continuation and zero will be assigned to a

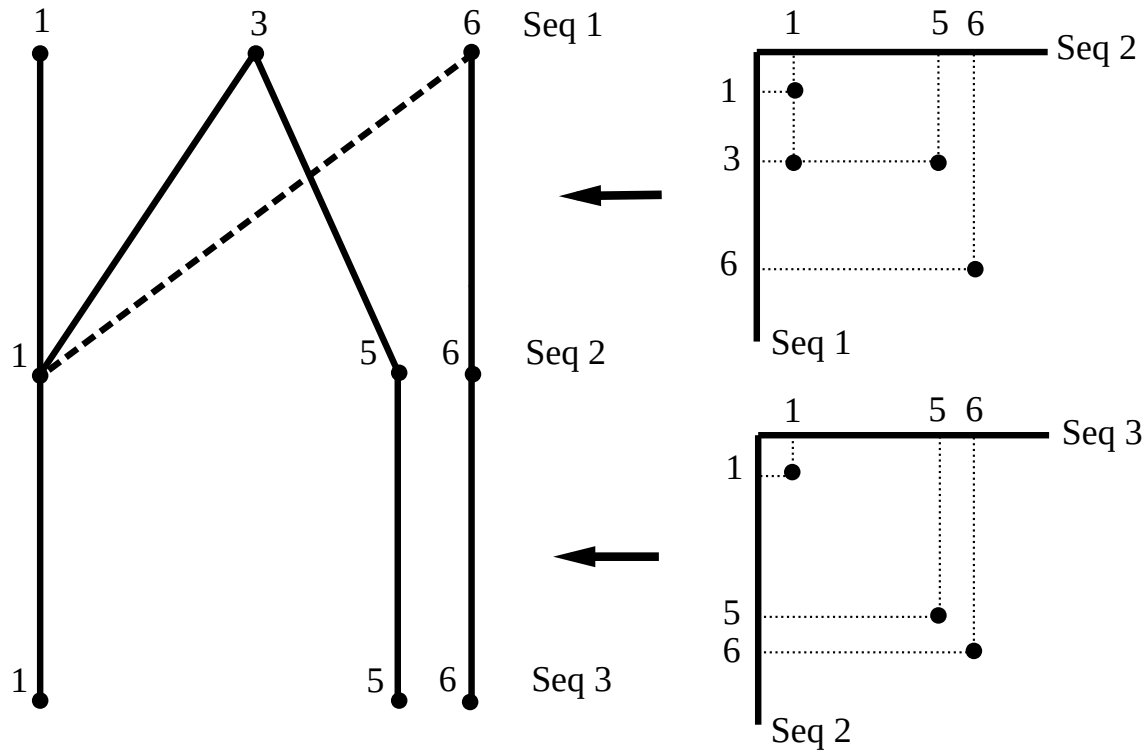


FIGURE 10: The dot matrices, $R_{1,2}$ and $R_{2,3}$, are given on the right. A tree is constructed according to Equation (4.4) starting from Sequence 1 and ending with Sequence 3. The first dot matrix gives rise to the first and second set of nodes on the tree labeled Seq 1 and Seq 2, respectively. The second matrix will give rise to the third set of nodes on the tree labeled Seq 3. The dotted line connecting node 6 of sequence 1 to node 1 of sequence 2 is not derived from the two dot matrices. This line is used to show the case that there is an extreme cross-over, the user can eliminate this path.

change, then (1, 1, 1) will have a score of two while (3, 5, 5) will have a score of 1. Once a residue position is used, it cannot be used again. For example, (1, 1, 1) and (3, 1, 1) cannot be chosen together because residue 1 in Sequence 1 is already aligned to residue 1 in Sequence 2. Using this process, the chosen fragments will be (1, 1, 1), (3, 5, 5), and (6, 6, 6). Note that in some cases, the links might cross

over like in the dotted link connecting node 6 and node 1 in Figure 10. In this case, we can set a limit to bound the distance between each element in the fragment, this limit is called a *window*.

The fragments chosen will signify all the similarities of one sequence to another. The remaining elements in between the fragments will be called an *intermediate sequence* as shown in Example 4.5. If our goal is to find the similarities and differences, then it is complete. However, if we wish to fully align the sequences along with the fragments, then the remaining intermediate sequences can be aligned using the GAM [14].

Example 4.5. Using $S_4 = \{AAGCG\}$, $S_5 = \{ATCTG\}$. If the chosen fragments are (1, 1) and (5, 5). Then the intermediate sequence belonging to S_4 is $\{AGC\}$ and, similarly, the intermediate sequence for S_5 is $\{TCT\}$.

Summary

In summary, this algorithm presented a way to simultaneously align multiple sequences to seek out the similarities and differences. The process begins with populating the family of dot-matrices using Equation (4.1). The elements of this family are filtered and updated using Equation (4.2). Once a consistent family is established, a subset of the consistent family is chosen according to Equation (4.4) so that all N sequences are used. Using this subset, we can start from the first element of the subset, then the second, then the third, etcetera. This process will create a tree structure showing all the possible fragments like in Figure 10. However, not all fragments will be used. We choose the proper fragments by attaching a higher score to nodes that lead to the same numbered nodes and a lower score to nodes that lead to a different numbered node. The fragments are chosen from left to right and once a node is used, it cannot be used again in another fragment. The resulting fragments yield all the aligned

similarities. If necessary, the remaining intermediate sequences will be aligned using the GAM. This algorithm executes with an overall computational complexity of $O(L^3N^4)$, where L is the maximum length of all N sequences [13].

CHAPTER 5

IMPLEMENTATIONS AND RESULTS

In this section, we will implement the N-Tuple method, the Dynamical Programming method, specifically the GAM, and the Dot-Matrix method. All methods are tested using MATLAB version 7.0 running on a 3.2 GHz dual-core Intel Pentium D processor 935 with 3.3 MB of RAM. The MATLAB codes for the corresponding methods are listed in Appendix A and are listed in the order in which they are used in this section. The testing will be done on five different

TABLE 1: This table gives the corresponding five sequences used to benchmark the MATLAB implementation of the N-Tuple, Global Alignment and Dot-Matrix methods

Sequence	Common Name/Location	Symbol	Length
H1N1	Mexico City	Q	2293
H1N1	Beijing	$R1$	2280
H5N1	Bird Flu	$R2$	2214
H3N2	Hong Kong Flu	$R3$	2284
H1N2	N/A	$R4$	2337

DNA sequences of the influenza A virus taken from GenBank [3]. All sequences that will be used here are listed on GenBank as segment 1 of 5 and are approximately the same lengths shown in Table 1. The first sequence is the H1N1 virus strand found in Mexico City. This strand will serve as the query sequence, Q, and will be used to match against four reference sequences, all of which are

different forms of the influenza A virus. The first reference sequence, $R1$, will be another H1N1 virus but extracted from an alternate location, Beijing. The second reference sequence, $R2$, is more commonly known as the bird flu, H5N1. The third reference sequence, $R3$, was responsible for the 1968 flu pandemic in Hong Kong, scientifically named H3N2 and commonly dubbed the Hong Kong Flu. The fourth reference sequence, $R4$, has no particular significance and is scientifically named H1N2. We avoided using the same H1N1 sequence as both the query sequence and the reference sequence because the result will be an obvious match. Since both H1N1 sequences that will be used are extracted from different locations, we can expect them to slightly differ from one another. Intuitively, when aligning Q with the reference sequences, we can expect the highest similarity to come from the alignment with $R1$ without any prior knowledge.

N-Tuple

The first implementation will be the N-Tuple method. Recall that N is the number of consecutive matches along any one diagonal. Before starting the N-Tuple algorithm, we will search for the highest N's when comparing Q with the references. Under normal circumstances, N will be empirically chosen using knowledge from prior experimentations. The list of the maximal N's is shown in Table 2. In searching for the maximal N, we see that our intuition is correct. That is, when comparing Q and $R1$ there is a high number of matches compared to the other reference sequences.

To begin the actual N-Tuple algorithm, we must note the results in Table 2 so that a proper N can be chosen. The N-Tuple algorithm will be executed with four different N values: 25, 50, 75, and 100. The results of each run is shown in Table 3 with time units in seconds. There are three points worth mentioning. First of all, as N increases the algorithm begins to filter out the sequences that do not

TABLE 2: A list of the maximal N when comparing the Query sequence, Q, versus each reference sequence

Sequences Compared	Maximal N
Q & $R1$	1167
Q & $R2$	33
Q & $R3$	57
Q & $R4$	81

satisfy N. When $N = 100$, only Q matches $R1$. In real cases, it is highly unlikely that the N-Tuple method will yield an exclusive match because the database will contain many more sequences than just four. Second of all, the algorithm spent approximately 99% of the time calculating the similarity matrix. However, as mentioned in Chapter 2.2, it is not necessary to calculate the entire similarity matrix. If we neglect the time spent calculating the similarity matrix, we see that the time spent in executing the method is only a fraction of a second. Lastly, as N increases in each case, the time it takes to execute the algorithm also increases. Recall that the algorithm will terminate once N is reached. If the number of matches is still below N the algorithm will continue to search until N is satisfied. Because of this, as N increases we can expect to see the execution time increase as well.

To further justify our results, we will examine the dot plots of Q versus each reference sequence, as shown in Figure 11. These figures are image representations of the similarity matrix with zero corresponding to white and one

TABLE 3: A list of results using the N-Tuple method, with N equals 25, 50, 75, and 100, the respective execution time which is the time spent calculating the similarity matrix and classification, and the total time in seconds

		Q & $R1$	Q & $R2$	Q & $R3$	Q & $R4$	Total Time
	Sim. Mat. Time	93.985	91.291	93.896	96.611	375.783
$N = 25$	Match(Y/N)	Y	Y	Y	Y	
	Execution Time	94.216	91.401	94.177	96.767	376.561
$N = 50$	Match(Y/N)	Y	N	Y	Y	
	Execution Time	94.365	91.606	94.255	96.795	377.021
$N = 75$	Match(Y/N)	Y	N	N	Y	
	Execution Time	94.371	91.619	94.561	96.801	377.352
$N = 100$	Match(Y/N)	Y	N	N	N	
	Execution Time	94.412	91.614	94.569	96.892	377.487

corresponding to black. In each dot plot, we can see a distinct diagonal line. The diagonal line shows the consecutive matches. If the diagonal line is more solid then the two sequences are more closely related. On the other hand, if the line is broken then the two sequences are less similar. It is difficult to determine whether or not the diagonal line is broken because of the length of the sequences. For this reason, we consider Figure 12 which is a zoomed version of Figure 11. Looking at Figure 12, we observe a seemingly solid line in Figure 12(a) and (d) and a more

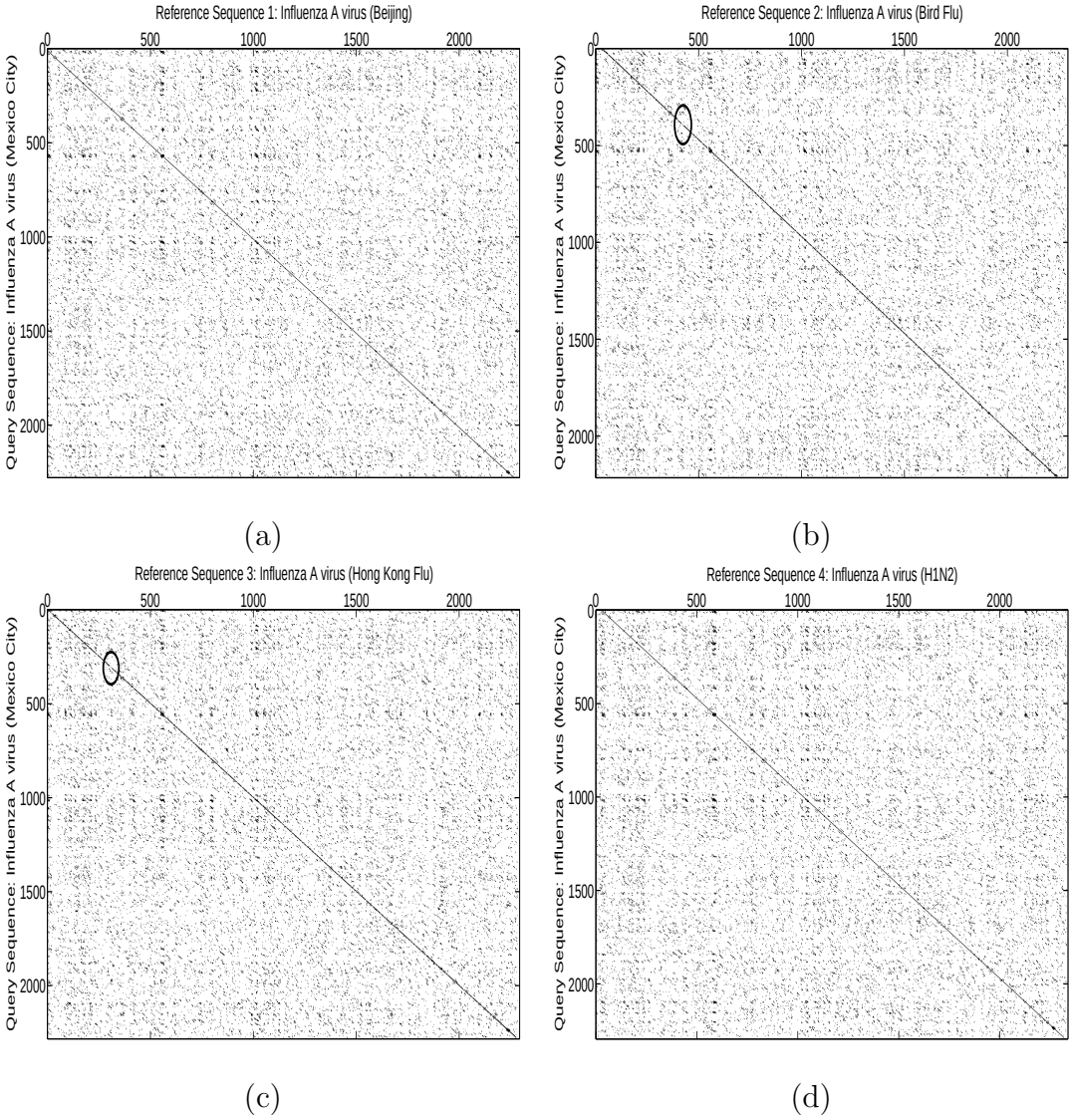


FIGURE 11: A dot plot of Q versus all the reference sequences in the database; (a) Q versus $R1$. (b) Q versus $R2$. (c) Q versus $R3$. (d) Q versus $R4$. The circled areas show some of the visible parts on the diagonal where there is a discontinuity.

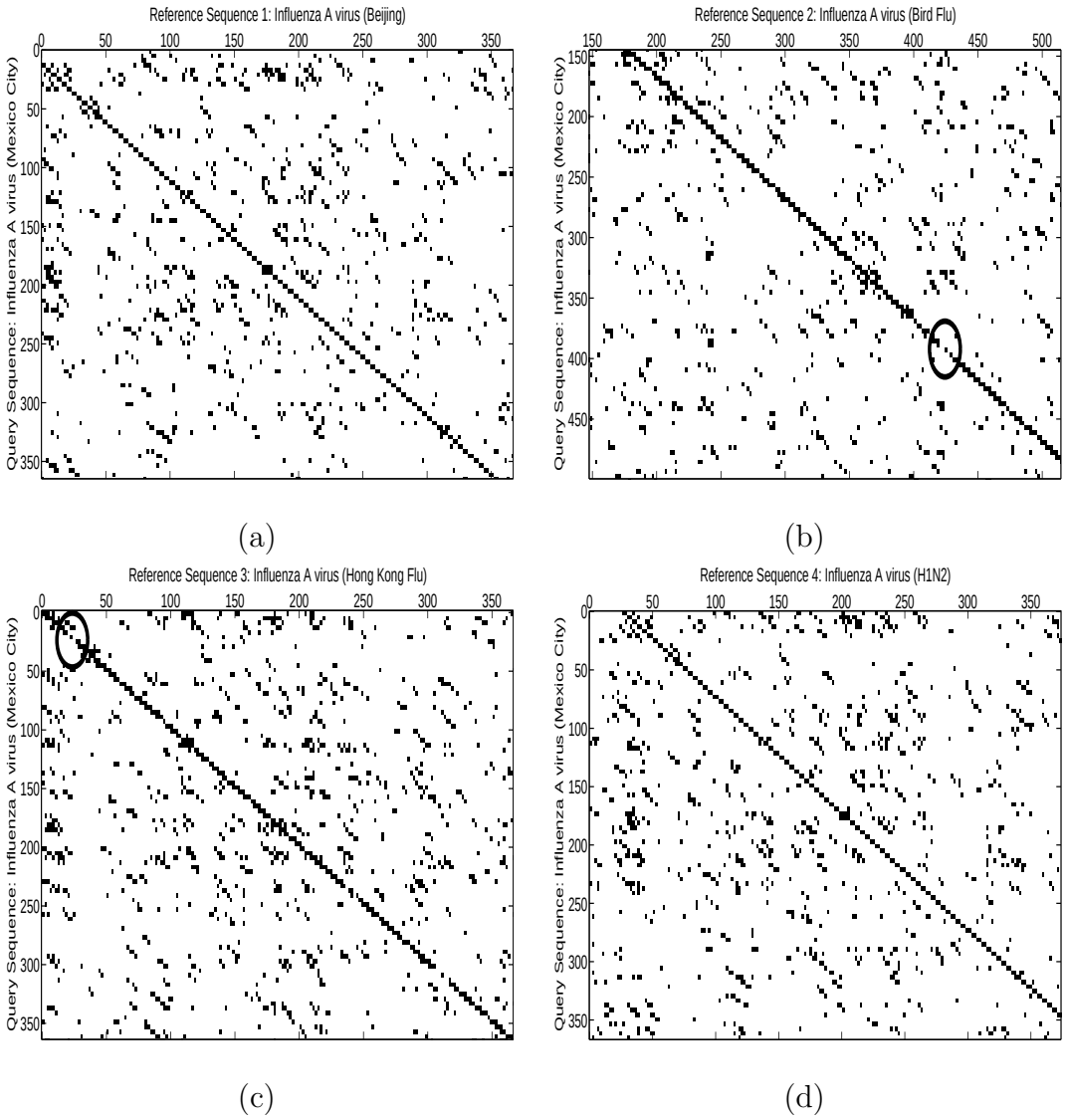


FIGURE 12: A zoomed in version of the dot plot in Figure 11. (a) Q versus $R1$. (b) Q versus $R2$. (c) Q versus $R3$. (d) Q versus $R4$. The circled areas show the parts in the diagonal where there is a discontinuity.

broken line in Figure 12(b) and (c). From these dot plots we suspect that Q is more closely related to $R1$ and $R4$ than to $R2$ and $R3$, which agrees with our results in Table 3.

Global Alignment

The second implementation will be the Dynamical Programming method. For this method, we will implement the GAM since we are dealing with sequences of relatively similar lengths. The results are noted in Table 4. The optimal alignment score coincides with the results we previously saw with the N-Tuple method in Table 3. That is, Q is most closely related to $R1$ and $R4$ than to $R2$ and $R3$. The resulting alignments are shown in Figure 13(b) with the original

TABLE 4: A list of results of the Needleman-Wunsch Global Alignment [8] documenting the optimal alignment score, $g_{m+1,n+1}$, and the overall execution time

Sequences Compared	Score	Execution Time (seconds)
Q vs. $R1$	2223	164.624
Q vs. $R2$	1791	153.590
Q vs. $R3$	1908	165.794
Q vs. $R4$	2127	181.641

sequences shown in Figure 13(a). Since the sequences are quite lengthy, we only show the first 100 elements in the alignments. The alignment with the highest score, Q with $R1$, is almost a direct match with the exception of some gaps in between elements 1 though 18. After the 18th element, the elements line up with a long string of consecutive matches. The alignment with the lowest score, Q with $R2$, is also very gappy in the beginning. Unlike the alignment of Q with $R1$, the

areas in which the elements are aligned show many mismatches. In every alignment, we can observe many more gaps inserted into the shorter of the two sequences, as discussed in Chapter 3.1.

```
Q:TATGGAGAGAATAAAAGAACTGAGAGATCTAATGTCGCAGTCCCGCACTCGCGAGATACTCACTAAGACCACTGTGGACCATATGGCCATAATCAAA
R1:CGTTCATATTCATATGGAGAGAATAAAAGAACTGAGAGATCTAATGTCGCAGTCCCGCACTCGCGAGATACTCACTAAGACCACTGTGGACCATATG
R2:ATGTCACAGTCCCGCACTCGCGAGATACTAACAAAAACCACTGTGGACCATATGGCCATAATCAAGAAATACACATCAGGAAGACAAGAGAAGAACC
R3:ATGGAAAGAATAAAAGAACTACGGAATCTGATGTCGCAGTCTCGCACTCGCGAGATACTGACAAAAACCACAGTGGACCATATGGCCATAATTAAGA
R4:AGCGAAAGCAGGTCAAATATATTCAATATGGAGAGAATAAAAGAACTAAGAGATCTAATGTCACAGTCCCGCACTCGCGAGATACTCACTAAGACCA
```

(a)

```
Q:T··A··TGG·------·AGAGAATAAAAGAACTGAGAGATCTAATGTCGCAGTCCCGCACTCGCGAGATACTCACTAAGACCACTGTGGACCATATG
R1:CGTTCATATTCATATGGAGAGAATAAAAGAACTGAGAGATCTAATGTCGCAGTCCCGCACTCGCGAGATACTCACTAAGACCACTGTGGACCATATG

Q:TATGGAGAGAATAAAAGAACTGAGAGATCTAATGTCGCAGTCCCGCACTCGCGAGATACTCACTAAGACCACTGTGGACCATATGGCCATAATCAAA
R2:A·TGTCA·CA·-----GT·---------CCCG··C----------ACTCGCGAGATACTAACAAAAACCACTGTGGACCATATGGCCATAATCAAG

Q:TATGGAGAGAATAAAAGAACTGAGAGA·TCTAATGTCGCAGTCCCGCACTCGCGAGATACTCACTAAGACCACTGTGGACCATATGGCCATAATCAA
R3:A·TGGAAAGAATAAAAGAACTACGG·AATCTGATGTCGCAGTCTCGCACTCGCGAGATACTGACAAAAACCACAGTGGACCATATGGCCATAATTAA

Q:TATGG·------------------------AGAGAATAAAAGAACTGAGAGATCTAATGTCGCAGTCCCGCACTCGCGAGATACTCACTAAGACCA
R4:AGCGAAAGCAGGTCAAATATATTCAATATGGAGAGAATAAAAGAACTAAGAGATCTAATGTCACAGTCCCGCACTCGCGAGATACTCACTAAGACCA
```

(b)

FIGURE 13: (a) The first 100 elements of the original five sequences. (b) The global alignment of each reference sequence with Q.

We will conclude implementation of the N-Tuple and GAM with two remarks. First, as discussed in Chapter 2.2, when using the N-Tuple method, we are not interested in the resulting alignment because it can yield many inaccuracies. Instead, we are only interested in using it to filter a vast database according to a specified N. The sequences that remain after the filtration is carefully aligned with a dynamical programming method. Consider this filtering process with $N = 75$. The N-Tuple algorithm will search all four reference sequences with a total time of 377.352 seconds. Then it will align the sequences

that are left, $R1$ and $R4$, with the GAM which will take an additional 346.265 seconds. The overall time for this process amounts to 723.617 seconds, which is approximately 12 minutes. Meanwhile, if we only use the GAM on all four comparisons, then the total time would be 665.649 seconds, this is approximately 11 minutes. This brings us to the second point; the filtering method is supposed to speed up the overall classification process but took 723.617 seconds while the GAM took 665.649 seconds. This is largely because we only have four sequences in the database. If we took a database of 100 sequences with an average N-Tuple time of 95 seconds per comparison, an average GAM time of 165 seconds, and 25 remaining sequences after the filtration. Then we will see that the total filtering process will take 13,625 seconds while searching only with the GAM takes 16,500 seconds. Therefore, we can conclude that in a large scale database search, which is often the case, it would be more beneficial to use the N-Tuple as a preprocessing followed by a dynamical method.

Dot-Matrix

Next, we will look at an implementation and result of the Dot-Matrix method. This algorithm was tested using the filtration method described in Equation (4.3) on the five sequences described before. The threshold, C, will be set to three and a score of one will be given to a node that leads to the same residue position. Otherwise the score will be 0. Additionally, a window of three will be set to limit the amount of crossing paths. In this implementation, if the distance between two residue exceeds the window then the total score is automatically -1. This will allow the algorithm to easily detect fragments that are not allowed. Some of the resulting fragments are listed in Table 5. Each element in the fragment will correspond to a certain residue on the corresponding sequence. For example, f_1 will align the 3rd residue in sequence Q with the 3rd residue in

TABLE 5: A list of all the fragments after the filtering process along with the score

	f_1	f_2	f_3	f_4	f_5
Q	3	6	8	14	17
$R1$	3	6	8	14	17
$R2$	4	6	8	16	19
$R3$	4	6	7	16	19
$R4$	4	6	7	16	19
Score	3	4	3	3	3

```
Q  : T A T G G A G A G A A T A A A A G A A C T G A G
R1 : C G T T C A T A T T C A T A T G G A G A G A A T
R2 : A T G T C A C A G T C C C G C A C T C G C G A G
R3 : A T G G A A A G A A T A A A A G A A C T A C G G
R4 : A G C G A A A G C A G G T C A A A T A T A T T C
```

(a)

```
Q  : T A - T G G A G A G A A T A - - A A A - - - G A A C T G A G
R1 : C G - T T C A T A T T C A T - - A T G - - - G A G A G A A T
R2 : A T G T C - A C A G T C C C G C A C T C G C G A G - - - - -
R3 : A T G G A - A - A G A A T A A A A G A A C - T A C G G - - -
R4 : A G C G A - A - A G C A G G T C A A A T A - T A T T C - - -
```

(b)

FIGURE 14: (a) The first 24 elements of the original sequences. (b) The sequences are shifted accordingly in order to align the fragments, underlined and bolded. The hyphens in this case represent a place holder instead of a gap.

sequence $R1$ with the 4th residue in sequence $R2$ and so on. Figure 14(b) shows all the alignment of the fragments. The remaining intermediate sequences can be aligned using GAM with a scoring profile [14].

Summary

In this chapter we have implemented and presented the results of each method discussed in this paper, namely the N-Tuple, a dynamical programming method, and the Dot-Matrix method. In a database search simulation, our results showed that using the N-Tuple method as a filter prior to using a dynamical method will yield a longer time than using the dynamical approach alone. However, our database only contained four reference sequences. In a large database search, we can expect the the filtering approach to reduce the search time. Additionally, the results showed that the N-Tuple method is faster than the Dynamical Programming method, more specifically the GAM, by almost two folds. Using $N = 100$, the N-Tuple was able to filter all the other sequences and yield an exclusive match, however, we cannot expect an exclusive match for a large database search. Therefore, it is necessary to use a dynamical programming method on the sequences that remain after the filtration. The result of using both alignment methods will increase speed and accuracy. The last implementation was using the dot-matrix to yield the fragments. Each one of these fragments were aligned to show areas of high similarities within five sequences. The Dot-Matrix method is not typically used in a database search, therefore it was not implemented in the same context as the N-Tuple and the Dynamical Programming method.

CHAPTER 6

CONCLUSION

To conclude, we will briefly reiterate all the main points we discussed in this paper. We first explored the N-Tuple method. This method creates a similarity matrix using the two sequences given. With this similarity matrix and a given threshold, N, a search is made along every single diagonal in order to find a consecutive number of matches that satisfies the threshold. Once N is satisfied, we can conclude that the two sequences are related. Next, we discussed the dynamical method. This method is categorized into two different algorithms, the GAM and the LAM. The GAM is used for sequences of relatively similar lengths and the LAM is used when the two sequences differ greatly in length. Both methods depend on the calculation of a weighted similarity matrix. Once the weighted similarity matrix is computed, the alignment path is chosen depending on the method used. These two dynamical programming methods allow an alignment path that will move from one diagonal to another where as the N-Tuple method only allows the alignment path to be on any one diagonal. Because of this, the Dynamical Method is much more accurate in detecting biological variations. However, the dynamical approach is more involved and takes almost twice the time as the N-Tuple to execute an alignment. The lack of execution speed led us to explore three different parallel computing schemes that enhanced the performance of the dynamical method, namely the methods proposed by Wozniak [10], Rognes and Seeberg [11], and Farrar [12]. The discussion of the dynamical methods was followed by a multiple alignment method using the dot-matrix method. Given n sequences, this method will create a family of $\binom{n}{2}$ dot-matrices. A filtration process will follow and remove all inconsistencies to create a consistent family of dot-plots. At the end, this method will yield highest scoring fragments which will be used in the alignment process. Since our goal was to find the segments of high similarity, then we are done. However, if one wishes to align the intermediate sequences in

between the fragments then this can be done using the GAM in conjunction with a scoring profile which is discussed in Vingron [14].

As a future study, we can focus on improving the multiple sequence alignment using the dot-matrices. Instead of using the filtration process discussed in Chapter 4.1, we can concatenate the family of dot-matrices into vectors. Using these vectors, we hope to be able to find a projection matrix that when applied to an incoming vector, it will yield all the matches between the vector and the family.

APPENDIX

APPENDIX A
MATLAB CODES

Code #1 This function creates a similarity matrix based on two input sequences, X and Y. The value at position (i, j) of the matrix, is based upon the comparison of the ith element of sequence X and the jth element of sequence Y. If there is a match between these two elements then the (i, j) position is one, otherwise it is zero.

```
function [SimMat]=Simi(X,Y)

for i=1:length(X)
    for j=1:length(Y)
        if X(i)==Y(j)
            SimMat(i,j)=1;
        else
            SimMat(i,j)=0;
        end
    end
end
```

Code #2 The purpose of this code is to find the maximum number of consecutive matches in any one diagonal of the similarity matrix.

```
function [MaxCount]=SearchCount(X,Y)

SimMat=Simi(X,Y); [a b]=size(SimMat); count2=0;
MaxDiag=b-1; MinDiag=-a+1;count=0;

for i=MinDiag:MaxDiag
%setting diagonals of SimMat into Diagonal
    Diagonal=diag(SimMat,i);
    %counting repeated ones
    for j=1:length(Diagonal)
        if Diagonal(j)==1
            count=count+1;
            if count2<count
                count2=count;
            end
        else
            count=0;
        end
    end
    MaxCount(a+i)=count2;
    count2=0;
    count=0;
end
```

Code #3 This is the implementation of the N-Tuple method. The user will input sequences, X and Y, and a threshold N. The algorithm will search through the diagonals of the similarity matrix to yield a match or no match depending on whether or not N is satisfied. The search will be terminated once N is satisfied or there are no more diagonals left.

```
function []=N_Tuple(N,X,Y)

SimMat=Simi(X,Y); [a b]=size(SimMat);
MaxDiag=b-N; MinDiag=-a+N; count=0;
for i=MinDiag:MaxDiag
%setting diagonals of SimMat into Diagonal
    Diagonal=diag(SimMat,i);
    %counting repeated ones
    for j=1:length(Diagonal)
        if Diagonal(j)==1
            count=count+1;
        else
            count=0;
        end
        if count>=N;
            break;
        end
    end
    if count>=N;
        break;
    end
end
if count>=N
    sprintf('Sequences are a match.')
else
    sprintf('Sequences are not a match.')
end
```

Code #4 This function creates the weighted similarity matrix of either the GAM or the LAM. The user will input sequences, X and Y, and a flag. If the flag is one, then the algorithm will calculate the weighted similarity matrix according to the LAM. If the flag is any other number, the algorithm will calculate the weighted similarity matrix according to the LAM.

```
function [G]=GlobaLocal(X,Y,flag)

n=length(X); m=length(Y);  Score=Simi(X,Y);
```

```
if flag==1
    G=zeros(n+1,m+1);
    Choice(4)=0;
    for i=2:n+1
        for j=2:m+1
            Choice(1)=G(i-1,j)-1;
            Choice(2)=G(i,j-1)-1;
            Choice(3)=G(i-1,j-1)+Score(i-1,j-1);
            G(i,j)=max(Choice);
        end
    end
else
    G=zeros(n,m);
    Column=-1*[1:n];
    Row=[0:-1:-m];
    G=horzcat(Column',G);
    G=vertcat(Row,G);
    for i=2:n+1
        for j=2:m+1
            Choice(1)=G(i-1,j)-1;
            Choice(2)=G(i,j-1)-1;
            Choice(3)=G(i-1,j-1)+Score(i-1,j-1);
            G(i,j)=max(Choice);
        end
    end
end
```

Code #5 This function aligns two sequences according to the GAM. This is used when two sequences are of similar lengths. The input will be two sequences and the output will be two aligned sequences. The resulting alignment may contain hyphens, which signals a gap.

```
function [newX, newY]=GlobalPath(X,Y)

G=GlobaLocal(X,Y,2); match=G(end,end);
[m, n]=size(G); row=m; col=n; gap='-';
Length=1; newY=''; newX='';

while (row~=1)&(col~=1)
    Choice=[G(row,col-1), G(row-1,col-1),G(row-1,col)];
    Index=find(Choice==max(Choice));
    if length(Index)==1
```

```
     %left movement
        if Index==1
            newX=strcat(gap,newX);
            newY=strcat(Y(col-1),newY);
            col=col-1;
        end
        %diagonal movement
        if Index==2
            newX=strcat(X(row-1),newX);
            newY=strcat(Y(col-1),newY);
            row=row-1; col=col-1;
        end
        %upward movement
        if Index==3
            newX=strcat(X(row-1),newX);
            newY=strcat(gap,newY);
            row=row-1;
        end
    else
        newX=strcat(X(row-1),newX);
        newY=strcat(Y(col-1),newY);
        row=row-1; col=col-1;
    end
end
```

Code #6 This function aligns two sequences according to the LAM. This is used when two sequences differ greatly in lengths. The input will be two sequences and the output will be two aligned sequences. The resulting alignment may contain hyphens, which signals a gap.

```
function [newX, newY]=LocalPath(X,Y)

M=GlobaLocal(X,Y,1);
[m, n]=size(M); gap='-'; row=m; dummy=M(m,:);
newY=''; newX=''; col=find(dummy==max(dummy));
%ensures single value
col=col(1); C=col;

%align only area of interest
while (row~=1)
    Choice=[M(row,col-1), M(row-1,col-1),M(row-1,col)];
    Index=find(Choice==max(Choice));
    if length(Index)==1
```

```
     %left movement
        switch Index
            case 1
                newX=strcat(gap,newX);
                newY=strcat(Y(col-1),newY);
                col=col-1;
            %diagonal movement
            case 2
                newX=strcat(X(row-1),newX);
                newY=strcat(Y(col-1),newY);
                row=row-1; col=col-1;
            %upward movement
            case 3
                newX=strcat(X(row-1),newX);
                newY=strcat(gap,newY);
                row=row-1;
        end
    else
        newX=strcat(X(row-1),newX);
        newY=strcat(Y(col-1),newY);
        row=row-1; col=col-1;
    end
end

%attach remaining of sequence
switch col&C
    case {col~=1 & C~=length(Y)}
        newY=strcat(Y(1:col),newY);
        newY=strcat(newY,Y(C:end));
        for i=1:col
            newX=strcat('*',newX);
        end
        for i=C:length(newY)-1
            newX=strcat(newX,'*');
        end
    case {col==1 & C~=length(Y)}
        newY=strcat(newY,Y(C:end));
        for i=C:length(newY)
            newX=strcat(newX,'*');
        end
    case {col~=1 & C==length(Y)}
        newY=strcat(Y(1:C,newY));
```

```
        for i=1:col
            newX=strcat('*',newX);
        end
end
```

Code #7 This function is used in the dot-matrix method in order to create a family of dot-matrices given n sequences. The input will be n sequences put into a matrix R. The number of rows of R will be n and the number of columns will depend on the maximum length of all the the sequences. For the shorter sequences, the user can fill remaining columns with arbitrary elements that does not match up with the other. That is, if there are three sequences the longest sequence will fill all the columns of R so no arbitrary elements will be needed. For the other two sequences, which are shorter than the maximum length, they will be filled with elements like "11111" and the other will be "22222" until the maximum length is reached. The arbitrary elements can be any element other than ones that already exist in the sequence, more specifically, A, T, C, or G. The output will be all possible dot-matrices stored in Fam. The Position matrix will contain as many rows as there are elements in the family and two columns. The two columns will represent the two sequences that are being compared via similarity matrix. The ith row will represent the ith element in the family.

```
function [Fam,Position]=Family(R)

[N m]=size(R);
k=1;

for i=1:N-1
%Simi of 1,2; 1,3 etc. no need for 1,1 so j=1.
    for j=i+1:N
        Fam(:,:,k)=Simi(R(i,:),R(j,:));
        Position(k,:)=[i j]';
        k=k+1;
    end
end
```

Code #8 This function executes a boolean multiplication of two matrices, given that their sizes are appropriate for matrix multiplication. Multiplication will be replaced with "and" and addition will be replaced with "or". For simplification, if the dot product between row i and column j is zero, then element (i, j) is zero, otherwise it is one.

```
function [Product]=BMult(A,B)
```

```
[c d]= size(A);
[e f]= size(B);

%creating the product matrix of size c x f.
for i=1:c
    for j=1:f
        if dot(A(i,:),B(:,j))==0
            Product(i,j)=0;
        else
            Product(i,j)=1;
        end
    end
end
```

Code #9 This function will search for two appropriate dot matrices based upon sequences s, t and i. In this case, i cannot be equal to s or t. The search process will look through the entire Position matrix to find a row that matches a specific row of Index. If the ith row matches the third row of Index during the jth iteration, then in the matrix multiplication process, the transpose of the jth element of Fam will be the matrix on the left. The result will yield two suitable matrices and their appropriate positioning, left or right, during the matrix multiplication process.

```
function [FirstMult,SecMult]=iSearch(Position,Fam,s,t,i)
[a b]=size(Position);
Index(1,:)=[s i];
Index(2,:)=[i t];
Index(3,:)=[i s];
Index(4,:)=[t i];

for j=1:a
    if Position(j,:)==Index(1,:)
        FirstMult=Fam(:,:,j);
    end
    if Position(j,:)==Index(2,:)
        SecMult=Fam(:,:,j);
    end
    if Position(j,:)==Index(3,:)
        FirstMult=Fam(:,:,j)';
    end
    if Position(j,:)==Index(4,:)
        SecMult=Fam(:,:,j)';
    end
end
```

Code #10 This function is the filtering process in the dot-matrix method specifically, according to Equation (4.3) in the Chapter 4.1. A threshold, Thresh, will be set by the user so that if Thresh amount of dot-matrices agree on the match of position (i, j) then position (i, j) will be kept as a match, one, instead of being reset to zero.

```
function [newFam]=FILSUM(Position,Fam,N,Thresh)

[x y z]=size(Fam);
max=100; summing=zeros(x,y); oldFam=Fam;
%counter for Prod
z=1;

for l=1:max
   %end of position matrix
   for k=1:(N*(N-1))/2
      %b=[s t]
      b=Position(k,:);
      for u=1:N
         %e.g 1,u * u,2 u cannot be 1 and 2
         if (u~=b(1))&(u~=b(2))
             [FirstMult SecMult]=iSearch(Position,oldFam,b(1),b(2),u);
             Prod(:,:,z)=BMult(FirstMult,SecMult);
             z=z+1;
         end
      end
      %Adding the current mat being filt to Prod
      Prod(:,:,z)=Fam(:,:,k);
      for i=1:z
         summing=summing+Prod(:,:,i);
      end
      z=1;
      for i=1:x
         for j=1:y
            if summing(i,j)>=Thresh
                newFam(i,j,k)=1;
            else
                newFam(i,j,k)=0;
            end
         end
      end
      tally(:,:,k)=summing;
      summing=zeros(x,y);
```

```
   end
   %convergence criteria
   if (newFam-oldFam)==0
      break;
   end
   %update oldFam and newFam
   oldFam=newFam;
end
```

Code #11 This function requires the input of the matrix R. R will contain all sequences that are being compared in rows. If the sequences are of different lengths then arbitrary elements can be added like described in Code #7. The algorithm will create a family of dot-matrices using Code #7 and the filtering process proceed according to Equation (4.3). The output will be a matrix of the chosen fragments and their respective scores.

```
function [chosenfrag]=FragSearch(R)

N=5;
[Fam,Position]=Family(R);
[newFam]=FILSUM(Position,Fam,N,3);

%creates index for back to back fam 12,23,34,45,..,N(N-1)
for i=1:N-1
    Arrange(i,1)=i;
    Arrange(i,2)=i+1;
end
k=1;
%looks for back to back fam
for i=1:N-1
    %end of position matrix
    for j=1:(N*(N-1))/2
        if norm(Arrange(i,:)-Position(j,:))==0
            FamMem(k)=j;
            k=k+1;
        end
    end
end

%Picks out back to back eles from newFam
for i=1:length(FamMem)
    FamArrange(:,:,i)=newFam(:,:,FamMem(i));
end
```

```
frag=[];
[a, b, c]=size(FamArrange);

%find all tree paths, i.e fragments
for i=1:a
    %find nonzero eles
    Ind1=find(FamArrange(i,:,1));
    if length(Ind1)>=1
        for j=1:length(Ind1)
            Ind2=find(FamArrange(i,:,2));
            for k=1:length(Ind2)
                Ind3=find(FamArrange(i,:,3));
                for l=1:length(Ind3)
                    Ind4=find(FamArrange(i,:,4));
                    if (length(Ind1)&length(Ind2)...
                        &length(Ind3)&length(Ind1))~=0
                        for m=1:length(Ind4)
                            temp(m,:)=[i,Ind1(j),...
                            Ind2(k),Ind3(l),Ind4(m)];
                        end
                        frag=[frag;temp];
                    end
                end
            end
        end
    end
end
%creats a 6th column to store score
[a,b]=size(frag); vect=zeros(a,1);
frag=horzcat(frag,vect); score=0; window=3;

%scoring all fragments
for i=1:a
    %check window size
    V=[abs(frag(i,1)-frag(i,2)),abs(frag(i,1)-frag(i,3)),...
        abs(frag(i,1)-frag(i,4)),abs(frag(i,1)-frag(i,5)),...
        abs(frag(i,1)-frag(i,2)),abs(frag(i,2)-frag(i,3)),...
        abs(frag(i,2)-frag(i,4)),abs(frag(i,2)-frag(i,5)),...
        abs(frag(i,3)-frag(i,4)),abs(frag(i,3)-frag(i,5)),...
        abs(frag(i,4)-frag(i,5))];
    for k=1:length(V)
```

```
        if V(k)>window
            score=-1;
            break;
        end
    end

    if score~=-1;
        for j=1:b-1
            compare1=frag(i,j);
            compare2=frag(i,j+1);
            %score +1 if elements are not branching
            if compare1==compare2
                score=score+1;
            end
        end
    end
    frag(i,b+1)=score;
    score=0;
end

i=1;
%deletes any row in fragment with a score of -1
while i<=a
    if frag(i,b+1)==-1
        frag(i,:)=[];
        i=i-1;
        a=a-1;
    end
    i=i+1;
end

[a b]=size(frag);
frag=sortrows(frag);temp=[]; chosenfrag=[]; maxscore=0; k=1;

%find all fragments with the highest score, 4
for i=1:a
    if frag(i,b)==4
        optimalfrag(k,:)=frag(i,:);
        k=k+1;
    end
end
```

```
for i=1:a-1
    compare1=frag(i,1);
    compare2=frag(i+1,1);
    %to account for the first row in fragment
    if (compare1~=compare2) & (i==1)
        chosenfrag=[chosenfrag;frag(1,:)];
    end
    %Compares scores for each branch with the same node
    if compare1==compare2
        compscore1=frag(i,b);
        compscore2=frag(i+1,b);
        if ((compscore1-compscore2)>=0) &...
            ((compscore1-compscore2)>maxscore)
            maxscore=compscore1-compscore2;
            temp=frag(i,:);
        end
    else
        chosenfrag=[chosenfrag;temp];
        maxscore=0;
    end
end

[a,b]=size(chosenfrag);
% replace row of chosenfrag with row optimal frag if
% the first node is the same
for i=1:a
    c=find(optimalfrag(:,1)==chosenfrag(i,1));
    if length(c)>0
        chosenfrag(i,:)=optimalfrag(c,:)
    end
end
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. Hutchison, "DNA sequencing: Bench to bedside and beyond," *Nucleic Acids Research*, vol. 35, no. 18, pp. 6227–6237, 2007.

[2] J. de Magalhaes, C. Finch, and G. Janssens, "Next-generation sequencing in aging research: Emerging applications, problems, pitfalls and possible solutions," *Ageing Research Reviews*, 2009.

[3] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, B. Rapp, and D. Wheeler, "Genbank," *Nucleic Acids Research*, vol. 33, pp. 34–38, 2005.

[4] E. Chan, "Advances in sequencing technology," *Mutation Research*, vol. 573, pp. 13–40, 2005.

[5] P. Pevzner, *Computational Molecular Biology: An Algorithmic Approach.* MIT Press, 2000.

[6] D. Knuth, *The Art of Computer Programming.* Addison-Wesley, 1973.

[7] C. Rangel-Escareno, J.-M. Chang, M. Vu, Q. Wu, and H. Wadhar, "A two-base encoded DNA sequence alignment problem in computational biology," claremont math-in-industry workshop, Harvey Mudd College, Claremont, California, 2009. Available at http://www.csulb.edu/∼jchang9/files/SequencingProbFinalReport09.pdf.

[8] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, pp. 443–453, 1970.

[9] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[10] A. Wozniak, "Using video-oriented instructions to speed up sequence comparison," *Oxford Journals: Bioinformatics*, vol. 13, no. 2, pp. 145–150, 1997.

[11] T. Rognes and E. Seeberg, "Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors," *Oxford Journals: Bioinformatics*, vol. 16, no. 8, pp. 699–706, 2000.

[12] M. Farrar, "Striped smith-waterman speeds database searches six times over other simd implementations," *Oxford Journals: Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[13] M. Vingron and P. Argos, "Motif recognition and alignment for many sequences by comparison of dot-matrices," *Journal of Molecular Biology*, vol. 218, pp. 33–43, 1991.

[14] M. Vingron and P. Argos, "A fast and sensitive multiple sequence alignment algorithm," *Computer Applications in the Biosciences*, vol. 5, no. 2, pp. 115–121, 1989.