MULTIPLE SEQUENCE ALIGNMENT WITH EVOLUTIONARY TREES

By

KE LIU

Bachelor of Science

Central University of Economics

Beijing, China

1994

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 2001

MULTIPLE SEQUENCE ALIGNMENT

WITH EVOLUTIONARY TREES

Thesis Approved:

M. H. Samadzadeh

Thesis Adviser

R. E. Hedman

J Chandler

Timothy J. Peltier

Dean of the Graduate College

PREFACE

Computation of multiple sequence alignments is one of the major open problems in computational molecular biology. The purpose of this study was to provide a new method for calculation of multiple amino acid sequence alignments. The new method is a progressive method. Multiple alignments are built by the successive application of pairwise alignment algorithms. The new method assumes that the sequences are related with each other and that there exists some unknown evolutionary tree that corresponds to the multiple sequence alignment.

In this new method, the distance between two groups is defined as the distance between the two corresponding consensus sequences. Group to group alignment is also calculated using the alignment of the consensus sequences. One advantage of this new method is that it is simple and efficient, and in many cases generates reasonable results. Another advantage of this method is that the scoring can be done with reference to this evolutionary tree, even though the tree structure itself may remain unknown.

ACKNOWLEDGEMENTS

I would like to convey my sincere appreciation to my major advisor Dr. Mansur H. Samadzadeh for his intelligent supervision, constructive guidance, inspiration, and friendship. My sincere appreciation extends to my other committee members, Dr. G. E. Hedrick and Dr. John P. Chandler, for their guidance and assistance. My sincere appreciation also extends to Dr. Ulrich Melcher for all of his advice and guidance.

I would also like to express my special appreciation and gratitude to my husband, Kunhong Xiao, for his encouragement, support, and love. I would also like to extend many thanks to my parents, sisters, and all my friends for their support and encouragement.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Multiple sequence alignments, which are important tools in studying proteins, are processes that consist of taking a group of sequences and identifying structurally or functionally homologous amino acids. The information they provide is very useful in designing experiments to test and modify the function of specific proteins, in predicting the function and structure of proteins, and in identifying new members of protein families [Karlin and Altschul 90] [Barton and Russell 93]. This chapter introduces some basic biological concepts that are related to this study.

1.1 DNA and Protein Sequences

All information of life is encoded in the different types of biological sequences: DNA sequences and protein sequences [Voet and Voet 95].

DNA (deoxyribonucleic acid) is a class of nucleic acids, which is the hereditary molecule in all-cellular life forms. DNA directs its own replication during cell division and directs the transcription of complementary molecules of RNA (ribonucleic acid), which then direct the synthesis of protein sequences [Stryer 95]. DNA is actually a linear string of four different kinds of nucleotides: Adenine (A), cytosine (C), guanine (G), and thymine (T) [Lehninger et al. 93].

Proteins are the fundamental building blocks of life. They are either the molecular machines responsible for virtually all of the chemical transformations that cells are capable of, or much of the structure of a cell. A human contains more than 100,000 different proteins [Marks et al. 96]. Protein is also a linear string of 20 kinds of amino acids. These 20 kinds of amino acids are the basic structure units of proteins, and they are often designated by either a three-letter abbreviation or a one-letter symbol.

Table I below illustrates the abbreviation of the 20 kinds of amino acids [Voet and Voet 95]. In this thesis, the discussion is focused on protein sequences.

Table I. Three-Letter Abbreviations and Single-Letter Symbols of Amino Acids [Voet and Voet 95]

Alanine	Ala	A	Leucine	Leu	L
Arginine	Arg	R	Lysine	Lys	K
Asparagine	Asn	N	Methionine	Met	M
Aspartic acid	Asp	D	Phenylalanine	Phe	F
Cysteine	Cys	C	Proline	Pro	P
Glutamine	Gln	Q	Serine	Ser	S
Glutamic acid	Glu	E	Threonine	Thr	T
Glycine	Gly	G	Tryptophan	Trp	W
Histidine	His	H	Tyrosine	Tyr	Y
Isoleucine	Ile	I	Valine	Val	V

1.2 Sequence Alignment and Homology

A sequence alignment is a one-to-one matching of two or more sequences so that each character in one sequence is associated with a single character of the other sequence or with a null character '-' (gap), showing where these sequences are similar and where they differ.

Aligning two or more sequences for comparison is an essential step in determining if they are homologous. Homologous sequences are sequences descended by

evolution from the same sequence of a common ancestor. The sequences of the homologous proteins are identical at the time they originated. But proteins are continually undergoing the stochastic process of mutation [King and Wilson 75].

The simplest and most frequent mutation is the substitution of one or more amino acids by other amino acids. Insertion and deletion of one or more amino acids are also common. Consequently, closely related proteins generally differ only by replacements of one amino acid by another at a few positions in the sequence. Less frequently are differences in the total number of the amino acids, which are due to the deletion or insertion of residues within the sequences [Doolittle 81]. The more closely related they are evolutionarily, the more similar their proteins sequences are found to be. As less closely related species are compared, the differences among their proteins sequences increase [Alberts et al. 94].

1.3 Amino Acid Similarity Scoring Systems

Sequence alignment algorithms are limited by the underlying model of sequence similarity, which is typically based on a set of scoring rules [George et al. 90].

If we put the similarity scores that we give to 210 possible amino-acid pairs (190 pairs of different amino acids plus 20 pairs of identical amino acids) into a matrix, this is frequently called a scoring matrix [Barton 96]. In a similarity scoring matrix, higher values are assigned to more similar pairs and lower values to dissimilar pairs.

The simplest similarity matrix, which is a unitary matrix, assigns values of +1 for identities and 0 for non-identities. Over years, many different matrices have been proposed ([McLachlan 71] [Dayhoff et al. 78] [Feng et al. 85] [Risler et al. 88] [George

et al. 90] [Gonnet et al. 92] [Henikoff and Kenikoff 92] [Altschul 93]). Among such scoring systems, the widely used scoring matrices have been the Dayhoff mutation data matrix [Dayhoff et al. 78] and Blosum matrix [Henikoff and Henikoff 92].

1.3.1 Dayhoff Mutation Data Matrix

Possibly the most widely used scheme for scoring amino acid pairs is that developed by Dayhoff and co-workers [Dayhoff et al. 78] [Schwartz and Dayhoff 78] and its recent variations [Gonnet et al. 92]. From a study of observed residue replacements in 71 sets of closely related proteins, Dayhoff and co-workers constructed the PAM model of molecular evolution. PAM stands for Percent Accepted Mutations and were inferred from the types of changes observed in these closely related proteins [Dayhoff et al. 78].

They combined the information about the individual kinds of mutations and about the relative mutability of the amino acids into one distance-dependent "mutation probability matrix", M. Each element M_{ij} of M gives the probability of the amino acid in column j mutating to the amino acid in row i after a given evolutionary interval.

When used for the comparison of protein sequences, the mutation probability matrix is usually normalized by dividing each element M_{ij} by the relative frequency of exposure to mutation of the amino acid i. This operation results in the symmetrical ``relatedness odds matrix" with each element giving the probability of amino acid replacement per occurrence of $i$ per occurrence of $j$. The resulting matrix is the ``log-odds matrix", which is frequently referred to as ``Dayhoff's matrix" [Altschul 91].

Table II shows PAM-250 matrix of Dayhoff, which is known to be most useful to reproduce accurate protein sequence alignments [George et al. 90].

Table II. Log Odds Matrix for 250 PAMs [Dayhoff et al. 78]

C	12																					
S	0	2																				
T	-2	1	3																			
P	-3	1	0	6																		
A	-2	1	1	1	2																	
G	-3	1	0	-1	1	5																
N	-4	1	0	-1	0	0	2															
D	-5	0	0	-1	0	1	2	4														
E	-5	0	0	-1	0	0	1	3	4													
Q	-5	-1	-1	0	0	-1	1	2	2	4												
H	-3	-1	-1	0	-1	-2	2	1	1	3	6											
R	-4	0	-1	0	-2	-3	0	-1	-1	1	2	6										
K	-5	0	0	-1	-1	-2	1	0	0	1	0	3	5									
M	-5	-2	-1	-2	-1	-3	-2	-3	-2	-1	-2	0	0	6								
I	-2	-1	0	-2	-1	-3	-2	-2	-2	-2	-2	-2	-2	2	5							
L	-6	-3	-2	-3	-2	-4	-3	-4	-3	-2	-2	-3	-3	4	2	6						
V	-2	-1	0	-1	0	-1	-2	-2	-2	-2	-2	-2	-2	2	4	2	4					
F	-4	-3	-3	-5	-4	-5	-4	-6	-5	-5	-2	-4	-5	0	1	2	-1	9				
Y	0	-3	-3	-5	-3	-5	-2	-4	-4	-4	0	-4	-4	-2	-1	-1	-2	7	10			
W	-8	-2	-5	-6	-6	-7	-4	-7	-7	-5	-3	2	-3	-4	-5	-2	-6	0	0	17		
	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W		

Elements are shown multiplied by 10. The neutral score is zero. A score of -10 means that the pair would be expected to occur only one-tenth as frequently in related sequences as random chance would predict, and a score of +2 means that the pair would be expected to occur 1.6 times as frequently. The order of the amino acids has been arranged to illustrate the patterns in the mutation data.

1.3.2 BLOSUM Matrix

An alternative approach to estimating target frequencies and the corresponding log-odds matrices has been advanced by Henikoff and Henikoff [Henikoff and Henikoff 92]. They examine multiple alignments of distantly related protein regions directly, rather than extrapolate from closely related sequences, hence the term BLOSUM is from BLOcks SUbstitution Matrix. A number of tests suggest that the "BLOSUM" matrices

produced by this method generally are superior to the PAM matrices for detecting biological relationships [Pearson 95] [Henikoff and Henikoff 93].

Different levels of the BLOSUM matrix can be created by differentially weighting the degree of similarity between sequences. For example, a BLOSUM 62 matrix is calculated from protein blocks such that if two sequences are more than 62% identical, the contribution of these sequences is weighted to sum to one. In this way the contributions of multiple entries of closely related sequences is reduced [Pearson 95]. The BLOSUM 62 matrix is given in Table III.

Table III. Blosum 62 Substitution Matrix [Henikoff and Henikoff 92]

	A	C	D	E	F	G	H	I	K	L	M	N	P	Q	R	S	T	V	W	Y
A	4																			
C	0	9																		
D	-2	-3	6																	
E	-1	-4	2	5																
F	-2	-2	-3	-3	6															
G	0	-3	-1	-2	-3	6														
H	-2	-3	-1	0	-1	-2	8													
I	-1	-1	-3	-3	0	-4	-3	4												
K	-1	-3	-1	1	-3	-2	-1	-3	5											
L	-1	-1	-4	-3	0	-4	-3	2	-2	4										
M	-1	-1	-3	-2	0	-3	-2	1	-1	2	5									
N	-2	-3	1	0	-3	0	1	-3	0	-3	-2	6								
P	-1	-3	-1	-1	-4	-2	-2	-3	-1	-3	-2	-2	7							
Q	-1	-3	0	2	-3	-2	0	-3	1	-2	0	0	-1	5						
R	-1	-3	-2	0	-3	-2	0	-3	2	-2	-1	0	-2	1	5					
S	1	-1	0	0	-2	0	-1	-2	0	-2	-1	1	-1	0	-1	4				
T	0	-1	-1	-1	-2	-2	-2	-1	-1	-1	-1	0	-1	-1	-1	1	5			
V	0	-1	-3	-2	-1	-3	-3	3	-2	1	1	-3	-2	-2	-3	-2	0	4		
W	-3	-2	-4	-3	1	-2	-2	-3	-3	-2	-1	-4	-4	-2	-3	-3	-2	-3	11	
Y	-2	-2	-3	-2	3	-3	2	-1	-2	-1	-1	-2	-3	-1	-2	-2	-2	-1	2	7

BLOSUM 62 matrix is calculated from comparisons of sequences with no less than 62% divergence. Identities are assigned the most positive scores. Frequently observed substitutions also receive positive scores and seldom observed substitutions are given negative scores.

CHAPTER II

PAIRWISE SEQUENCE ALIGNMENT WITH DYNAMIC PROGRAMMING

2.1 Introduction to Pairwise Sequence Alignment

2.1.1 Definition of Pairwise Alignment

A pairwise alignment of two sequences S_1 and S_2 is produced when spaces (or gaps), '-', are inserted either into or at the ends of S_1 and S_2 to form the new sequences, S_1' and S_2', both of which must be of the same length. The two resulting sequences are then placed one above the other [Barton 96].

Let $S_1 = a_1a_2...a_m$ and $S_2 = b_1b_2...b_n$ be two sequences of lengths m and n, respectively. Then the alignment is

$a_1'a_2'...a_k'$ (S_1')
$b_1'b_2'..b_k'$ (S_2')

Ignoring gap characters, S_1' and S_2' are exactly sequences S_1 and S_2, respectively. Amino acid pairs of the alignments, (a_i, b_i), can end in one of three ways.

1) $(a, -)$ corresponds to the insertion of a into the first sequence, or the deletion of a from the second sequence.
2) (a, b) corresponds to an identity or match if $a = b$, or a substitution or mismatch if $a \neq b$.
3) $(-, b)$ corresponds to an insertion/deletion of b.

No alignment terms (-, -) are allowed, as there is no point in matching two deletions. Therefore, $\max(n, m) \leq k \leq n + m$ [Gusfield 97].

2.1.2 Value of an Alignment

There are two ways to quantify similarity of the alignment of two sequences, the similarity measures and the distance. In most cases, distance and similarity measures are interchangeable in the sense that a small distance means high similarity, and vice versa. Similarity measures are used in this chapter.

For a given alignment A of S_1 and S_2, let S_1' and S_2' denote the aligned sequences, and let k denote the (equal) length of the two sequences S_1' and S_2' in A. Let $s(a, b)$ denotes the value (or score) obtained by aligning character a against character b using a scoring matrix (scoring matrices were discussed briefly in Chapter I). The value of the alignment A is defined as

$$V = \sum_{i=1}^{K} s(a_i', b_i') .$$

Given a scoring matrix, the pairwise alignment problem is to find an optimal alignment maximizing the total alignment value V. The optimal alignment will emphasize matches (or similarities) while penalizing mismatches or inserted spaces.

2.1.3 Number of Alignments

From the definition for the sequence alignment discussed in Section 2.1.1, we know that there are many ways to align two sequences.

Theorem 2.1 [Waterman 95]

Define $f(m, n)$ = number of alignments of one sequence of m characters with another of n characters, for $m, n > 0$, then

$$f(m,n) = f(m-1,n) + f(m-1,n-1) + f(m,n-1)$$

with boundary conditions

$$f(m,0) = f(n,0) = 1$$

Proof. On the end of the alignment, if a_m is deleted, then there exist $f(m\text{-}1, n)$ alignments of the earlier part of the sequence. If a_m and b_n are aligned, $f(m\text{-}1, n\text{-}1)$ alignments result. If b_n is deleted, then $f(m, n\text{-}1)$ alignments result. ■

Waterman gave a solution for $f(n, n)$ as follows [Waterman 95].

$$f(n,n) \sim (1+\sqrt{2})^{2n+1} n^{-1/2} \ as \ n \to \infty$$

where $\sim$ is for limiting behavior

According to the above equation, two sequences of length 1000, for example, have

$$f(1000,1000) \approx (1+\sqrt{2})^{2001} 1000^{-1/2} = 10^{764.4...}$$

alignments. Obviously, one cannot examine all alignments to compute the optimal one.

2.2 Computing Optimal Alignment with Dynamic Programming

Dynamic programming is typically applied to optimization problems [Cormen et al. 92] [Bellman 57]. Dynamic programming solves problems by combining the solution to sub-problems, and it solves every sub-problem only once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time the sub-problem is encountered.

The first use of the dynamic programming approach for sequence alignment was reported by Needleman and Wunsch in 1970 [Needleman and Wunsch 70] and later in slightly modified forms by others [Sellers 74] [Sankoff and Kruskal 83].

2.2.1 Optimal Structure and Recursive Solution of Pairwise Alignment

Dynamic programming methods assume the principle of optimality, stating that each part of a globally optimal solution is itself an optimal solution to its corresponding partial problem. The optimal structure of the sequence pairwise alignment problem is defined in Theorem 2.2 below.

Theorem 2.2 [Gusfield 97]

Given $S_1 = a_1a_2\ldots a_m$ and $S_2 = b_1b_2\ldots b_n$, then alignment of S_1 and S_2 can end in one of three ways: $(a_m, -)$, (a_m, b_n), and $(-, b_n)$ (Section 2.1.1). Defined $V[i, j]$ ($1 \le i \le m$, $1 \le j \le n$) as the value of the optimal alignment of prefixes $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$. The optimal substructure of an optimal pairwise alignment can be stated as follows.

1) If the optimal alignment ends in (a_m, b_n), then $V[m, n] = V[m-1, n-1] + s(a_m, b_n)$.

2) If the optimal alignment ends in $(a_m, -)$, then $V[m, n] = V[m-1, n] + s(a_m, -)$.

3) If the optimal alignment ends in $(-, b_n)$, then $V[m, n] = V[m, n-1] + s(-, b_n)$.

Proof.

1) If the optimal alignment ends in (a_m, b_n), then the initial part of the alignment $a_1a_2\ldots a_{m-1}$ and $b_1b_2\ldots b_{n-1}$ must itself be optimal. Suppose for the purpose of contradiction there is an optimal alignment of $a_1a_2\ldots a_{m-1}$ and $b_1b_2\ldots b_{n-1}$ with similarity score $V'[m-1, n-1]$. Then aligning a_m and b_n to the initial part of the optimal alignment produces a

similarity score of S_1 and S_2, $V[m-1, n-1] + s(a_m, b_n)$, which is greater than $V[m, n]$. This is a contradiction to our assumption that $V[m, n]$ is an optimal similarity score.

2) If the optimal alignment ends in $(a_m, -)$, then the initial part of the alignment of $a_1a_2\ldots a_{m-1}$ and $b_1b_2\ldots b_n$ must also be optimal.

3) The case $(-, b_n)$ is identical in reasoning with the case $(a_m, -)$. ■

Based on Theorem 2.2, the recursive solution to the optimal alignment problem can be easily induced.

Theorem 2.3 [Waterman 95]

Given $S_1 = a_1a_2\ldots a_m$ and $S_2 = b_1b_2\ldots b_n$, define $V[i, j]$ as the value of the optimal alignment of prefixes $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$. Also set

$$\mathrm{V}[0,\ 0] = 0,\ V[0, j] = \sum_{k=1}^{j} s(-, b_k)\text{, and } V[i,0] = \sum_{k=1}^{i} s(a_k, -).$$

Then, for $1 \le i \le m$ and $1 \le j \le n$, the general recurrence is

$$V[i, j] = \max\{V[i-1, j-1] + s(a_i, b_j), V[i-1, j] + s(a_i, -), V[i, j-1] + s(-, b_j)\}$$ ■

2.2.2 Bottom-up Computation of Pairwise Alignment

Based on the recursive equation given by Theorem 2.3, we could easily code the recursive relations and base conditions for $V[i, j]$ as a recursive procedure. This top-down recursive approach requires exponential time complexity due to the massive number of redundant recursive calls to the procedure [Cormen et al. 92].

Typically, the bottom-up computation is organized with a dynamic programming table of size $(m+1)\times(n+1)$. The table holds the values of $V[i, j]$ for all choices of i and j, where $0 \le i \le m$ and $0 \le j \le n$. Sequence S_1 corresponds to the vertical axis of the table

while sequence S_2 corresponds to the horizontal axis. The values in row zero and column zero are filled in directly from the base conditions. After that, the remaining $m \times n$ sub-table is filled in one row at a time, in order of increasing i. Within each row, the cells are filled in order of increasing j. Figure 1 below shows the bottom-up computation algorithm to Theorem 2.3.

```
1  procedure S(a_1 a_2 ... a_m, b_1 b_2 ... b_n, m, n)
2      V[0,0] ← 0
3      for j ← 1 to n do
4          V[0, j] ← V[0, j − 1] + s(−, b_j)
5      for i ← 1 to m do
6          V[i, 0] ← V[i − 1, 0] + s(a_i, −)
7          for j ← 1 to n do
8              V[i, j] ← max{V[i−1, j−1]+s(a_i, b_j), V[i−1, j]+s(a_i, −), V[i, j−1]+s(−, b_j)}
9      return V[m, n]
```

Figure 1. Basic Dynamic Programming Method for Sequence Alignment [Waterman 95]

Table IV below shows the computation of the optimal similarity alignment between ABCNJRQCLCRPN and AJCJNRCKCRBP. The scoring matrix used is a unitary scoring matrix. Gaps are given the same penalty as a mismatch.

In this table, when computing the values for a specific cell (i, j), only cells $(i - 1, j - 1)$, $(i, j - 1)$, and $(i - 1, j)$ are examined. Hence, to fill in one cell takes a constant number of arithmetic operations and comparisons. There are $O(mn)$ cells in the table, so the dynamic programming algorithm for computing the alignment score between two sequences of length m and n can be computed in $O(mn)$ time [Waterman 95].

Table IV. A Sample Dynamic Programming Table for Two Sequences

		0	1	2	3	4	5	6	7	8	9	10	11	12	13
			A	B	C	N	J	R	Q	C	L	C	R	P	M
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1	1	1	1	1	1	1	1
2	J	0	1	1	1	1	2	2	2	2	2	2	2	2	2
3	C	0	1	1	2	2	2	2	2	3	3	3	3	3	3
4	J	0	1	1	2	2	3	3	3	3	3	3	3	3	3
5	N	0	1	1	2	3	3	3	3	3	3	3	3	3	3
6	R	0	1	1	2	3	3	4	4	4	4	4	4	4	4
7	C	0	1	1	2	3	3	4	4						
8	K														
9	C														
10	R														
11	B														
12	P														

The operation of successive summations began at the first row of the array and proceeded row-by-row towards the last row. The operation has been partially completed in row 7. The highlighted cell in this row is the site of the cell operation, which consists of a search along the enclosed cells indicated by borders for the largest value, which is 4. The sum of this value and the value of s(C, C), which is 1, is added to the cell from which the search began.

2.2.3 Traceback

The computation described above calculates an optimal similarity score without specifying the alignment or alignments that produce the score. The optimally matched alignments can be back-traced, guided by a direction matrix whose i, jth element is a pointer indicating the paths through which the maximum value of $V[i, j]$ is chosen [Smith et al. 81]. Pointers in each cell are set to:

$$\begin{cases} \text{upper left cell(“}\nwarrow\text{”) if } V[i,j] = V[i-1, j-1] + s(a_i, b_j) \\ \text{upper cell (“}\uparrow\text{”) if } V[i,j] = V[i-1, j] + s(a_i, -) \\ \text{left cell (“}\leftarrow\text{”) if } V[i,j] = V[i, j-1] + s(-, b_j) \end{cases}$$

The optimal alignment is recovered from the path by interpreting:

- "↖" as a match if $a_i = b_j$ and as a substitution if $a_i \neq b_j$
- "↑" as a deletion of a_i from S_1 (a space, -, inserted into S_2)
- "←" as an insertion of b_j into S_2 (a space, -, inserted into S_1)

For the example mentioned in Table IV, we can trace back paths from cell (12, 13) to cell (0, 0) (see Table V) and construct optimal alignments at the same time. One of the optimal alignments, whose pathway is illustrated in Table V, is as follows.

```
ABCNJ-RQCLCR-PM
| | | | | || |
AJC-JNR-CKCRBP-
```

where '|' shows the identities in the optimal alignment.

Table V. A Sample Traceback Table for Dynamic Programming

		A	B	C	N	J	R	Q	C	L	C	R	P	M
	0	←0	←0	←0	←0	←0	←0	←0	←0	←0	←0	←0	←0	←0
A	↑0	↖1	←1	←1	←1	←1	←1	←1	←1	←1	←1	←1	←1	←1
J	↑0	↑1	↖←1	↖←1	↖←1	↖2	←2	←2	←2	←2	←2	←2	←2	←2
C	↑0	↑1	↖←1	↖2	←2	←2	←2	↖←2	↖←2	↖3	←3	↖←3	←3	←3	←3
J	↑0	↑1	↖←1	↑2	↖←2	↖3	←3	←3	←3	←3	↖←3	↖←3	↖←3	↖←3	↖←3
N	↑0	↑1	↖←1	↑2	↖3	←3	↖←3	↖←3	↖←3	↖←3	↖←3	↖←3	↖←3	↖←3	↖←3
R	↑0	↑1	↖←1	↑2	↑3	↖←3	↖4	←4	←4	←4	←4	↖←4	←4	←4	←4
C	↑0	↑1	↖←1	↖2	↑3	↖←3	↑4	↖←4	↖5	←5	↖←5	←5	←5	←5	←5
K	↑0	↑1	↖←1	↑2	↑3	↖←3	↑4	↖←4	↑5	↖←5	↖←5	↖←5	↖←5	↖←5	↖←5
C	↑0	↑1	↖←1	↖2	↑3	↖←3	↑4	↖←4	↖5	↖←5	↖6	←6	←6	←6	←6
R	↑0	↑1	↖←1	↑2	↑3	↖←3	↖4	↖←4	↑5	↖←5	↑6	↖7	←7	←7	←7
B	↑0	↑1	↖2	←2	↑3	↖←3	↑4	↖←4	↑5	↖←5	↑6	↑7	↖←7	↖←7	↖←7
P	↑0	↑1	↑2	↖←2	↑3	↖←3	↑4	↖←4	↑5	↖←5	↑6	↑7	↖8	←8	←8

This is a completed table with traceback pointers for the example mentioned in Table IV. One of the pathways that could form the maximum match is illustrated.

2.3 Sequence Alignment Using Affine Gap Costs

So far, we have treated the gap symbol '-' as yet another character, denoting an individual insertion or deletion. Each gap is given a constant weight independent of the number of spaces in the gap. However, this view is not always adequate.

Frequently in sequence evolution, deletion or insertion of several adjacent letters are not the sum of single deletions or insertions but the result of one event [Stryer 95]. Let $g(k)$ be the indel weight for an indel of k bases, and g(1) be the gap penalty of one space. It is reasonable that $g(k) \leq kg(1)$ holds [Altschul and Erickson 86].

Affine gap penalty means that we charge a certain set-up cost for introducing a new gap, whereas extending an existing gap is less expensive [Fitch and Smith 83]. Affine gap penalty is of the form

$$g(k) = \alpha + \beta(k-1).$$

where α and β are two constants denote the gap open penalty and the gap extension penalty, respectively. Some examples of range of α and β in use for proteins are given in Table VI.

Table VI. Examples of Range of Affine Gap Penalties

Protein Scoring Matrix	Gap Open Penalty	Gap Extension Penalty
PAM 250	12-16	4-16
Blosum 62	9-12	3-12

Waterman et al. [Waterman et al. 76] have generalized the basic dynamic programming algorithm so that any set of gap penalty functions satisfying $g(k) \leq kg(1)$ can be used.

Theorem 2.4 [Waterman et al. 76]

Given $S_1 = a_1a_2\ldots a_m$ and $S_2 = b_1b_2\ldots b_n$, define $V[i, j]$ as the value of the optimal alignment of prefixes $a_1a_2\ldots a_i$ and $b_1b_2\ldots b_j$. Also set

$V[0, 0] = 0$, $V[0, j] = -g(j)$, and $V[i, 0] = -g(i)$.

Then

$$V[i,j] = \max\begin{Bmatrix} V[i-1, j-1] + s(a_i, b_j), \\ \max_{1\le k\le j}\{V[i, j-k] - g(k)\}, \\ \max_{1\le l\le i}\{V[i-l, j] - g(l)\} \end{Bmatrix}.$$

Proof.

The proof is the same as in Theorem 2.2, except here the alignments can end in deletion of up to i letters of the sequence $a_1\ldots a_i$. So, instead of $V[i-1, j] + s(a_i, -)$, which is the same as $V[i\text{-}1, j] - g(1)$, we have $\max_{1\le l\le i}\{V[i-l, j] - g(l)\}$. ∎

The main disadvantage of Theorem 2.4 is that it takes $\sum_{i,j}(i+j) = O(m^2n + n^2m)$, or $O(n^3)$ if $m = n$, number of steps to find the optimal alignment. Gotoh presented an algorithm that allows linear gap costs but runs in essentially mn steps [Gotoh 82].

Theorem 2.5 [Gotoh 82]

Let $g(k) = \alpha + \beta(k\text{-}1)$ for constants α and β. Set $E[0, 0] = F[0, 0] = V[0, 0] = 0$, $E[0, j] = F[0, j] = V[0, j] = -g(j)$, and $E[i, 0] = F[i, 0] = V[i, 0] = -g(i)$. Then if

$$E[i,j] = \max\{V[i, j-1] - \alpha, E[i, j-1] - \beta\},$$

and

$$F[i,j] = \max\{V[i-1, j] - \alpha, F[i-1, j] - \beta\},$$

then

$$V[i,j] = \max\{V[i-1, j-1] + s(a_i, b_j), E[i,j], F[i,j]\}.$$

Proof.

The identities that we need to establish are $E[i,j] = \max_{1 \le k \le j} \{V[i, j-k] - g(k)\}$ and $F[i,j] = \max_{1 \le l \le i} \{V[i-l, j] - g(l)\}$. The proof for $E[i,j]$ follows.

Assume $E[i,j^*] = \max_{1 \le k \le j^*} \{V[i,j^*] - g(k)\}$ *for* $0 \le j^* < j$, then

$$
\begin{aligned}
&\max_{1 \le k \le j} \{V[i, j-k] - g(k)\} \\
&= \max \{\max_{2 \le k \le j} \{V[i, j-k] - g(k)\}, V[i, j-1] - g(1)\} \\
&= \max \{\max_{1 \le k-1 \le j-1} \{V[i, (j-1)-(k-1)] - g(k-1)\} - \beta, V[i, j-1] - \alpha\} \\
&= \max \{V[i, j-1] - \alpha, E[i, j-1] - \beta\} = E[i,j]
\end{aligned}
$$

■

Examination of the recurrences in Theorem 2.5 shows that for any pair (i,j), each of the terms $V[i,j]$, $E[i,j]$, and $F[i,j]$ is evaluated by a constant number of references to the previously computed values, arithmetic operations, and comparisons. Hence $O(mn)$ time suffices to fill in all the $(n+1) \times (m+1)$ cells in the dynamic programming table [Altschul and Erickson 86].

All alignment algorithms discussed in this chapter are global alignments. Global denotes the feature that all letters of the two involved sequences must be accounted for in the alignment [Pearson and Lipman 88]. An alternative method is local alignment, also called approximate pattern matching, which places into correspondence a segment from each of the two sequences being compared [Barton 90] [Altschul et al. 90]. Related work in studying global and local pairwise alignment can be found in [Guo 00].

CHAPTER III

MULTIPLE SEQUENCE ALIGNMENT METHODS

3.1 Scoring a Multiple Alignment

A global multiple alignment of $k > 2$ sequences $S = \{S_1, S_2, \ldots, S_k\}$ is a natural generalization of pairwise alignment [Barton 96]. Spaces are inserted into or at either end of each of the k sequences so that the resulting sequences have the same length l. Then the sequences are arrayed in k rows of l columns each, so that each character and space of each sequence is in a unique column. The following is an example of multiple alignment [Gusfield 97].

```
VTISCTGSSSNIGAG-NHVKWYQQLPG
VTISCTGTSSNIGS--ITVNWYQQLPG
LRLSCSSSGFIFSS--YAMYWVRQAPG
LSLTCTVSGTSFDD--YYSTWVRQPPG
PEVTCVVVDVSHEDPQVKFNWYVDG--
ATLVCLISDFYPGA--VTVAWKADS--
AALGCLVKDYFPEP--VTVSWNSG---
VSLTCLVKGFYPSD--IAVEWESNG--
```

In the alignment above, eight fragments from immunoglobulin (antibody) protein sequences are displayed together. Their alignment highlights conserved residues (like 'C' at column 4 and 'W' at column 21), conserved regions (in particular, "Q.PG" at the end of the first four sequences), and more sophisticated patterns.

Although the notion of multiple alignment is easily extended from pairwise alignment, the score or goodness of a multiple alignment is not as easily generalized [Waterman 95].

The scoring system for a multiple alignment should take into account an important feature of multiple alignments: the fact that the sequences are not independent, but instead are related by a phylogenetic tree [Feng and Doolittle 87]. An idealized way to score a multiple alignment would therefore be to specify a complete model of molecular sequence evolution. Since there is not enough data to parameterize the complex evolutionary model, simplifying assumptions must be made to score a multiple alignment [Gusfield 97].

To date, there is no objective function that has been as well accepted for multiple alignments as similarity has been for pairwise alignment [Gusfield 97]. Different methods implement different scoring functions, and most of the algorithms may not implement any scoring functions [Waterman 95]. The goodness of those methods is judged by the biological meaning of the alignments that they produce, and so the biological insight of the evaluator is of critical importance.

Two of the most widely used multiple sequence alignment methods, the exact method and the progressive alignment method, are introduced in Section 3.2 and 3.3 below.

3.2 Exact Methods with the Sum-of-Pairs (SP) Score

The sum-of-pairs (SP) scoring method assumes that the individual columns of an alignment are statistically independent [Gusfield 97].The overall SP score for a multiple alignment can be defined as the sum of the scores for each column of the alignment.

Definition 3.1 [Waterman 95]

Given a multiple alignment M of length l, let $S(M)$ denote the overall SP score, $S(M_i)$ denote the SP score for column i, and $s(a, b)$ denote the similarity score of the amino acid a and b. Let M_i^k denote the amino acid at column i and row k of M. Assume gap cost $g(k) = kg(1)$ for a gap of length k, then

$$S(M) = \sum_{1 \le i \le l} S(M_i)$$

$$\text{where } S(M_i) = \sum_{k<h} s(M_i^k, M_i^h)$$ ■

The SP alignment problem is to compute a global multiple alignment with the maximum SP similarity score or with the minimum distance (or cost).

The SP score was first introduced by Carrillo and Lipman [Carrillo and Lipman 88], and was subsequently used by Bacon and Anderson [Bacon and Anderson 86], Murata [Murata et al. 85], Gotoh [Gotoh 86], and Gupta [Gupta et al. 95].

3.2.1 N-Dimensional Dynamic Programming Method

The SP problem can be solved exactly (optimally) via dynamic programming [Murata et al. 85]. However, $O(n^d)$ time and space is required to align d sequences of length at most n using the dynamic programming method. Hence this method is practical for only a small number of sequences.

What follow are dynamic programming recurrences for the case of three strings [Murata et al. 85]. Extension to larger number of strings is straightforward.

Theorem 3.1 [Murata et al. 85]

Let $S_1=a_1a_2\ldots a_{n1}$, $S_2=b_1b_2\ldots b_{n2}$, and $S_3=c_1c_2\ldots c_{n3}$ denote three strings of lengths n_1, n_2, and n_3, respectively. Let $s(a, b)$ denote the similarity score for amino acids a and b, and g(1) denote the gap penalty of one space. Let $V(n_1, n_2, n_3)$ be the optimal SP score for aligning S_1, S_2, and S_3 of lengths n_1, n_2, and n_3, then

$$V(i,j,k)=\max\begin{cases} V(i-1,j-1,k-1)+s(a_i,b_j)+s(a_i,c_k)+s(b_j,c_k); \\ V(i-1,j-1,k)+s(a_i,b_j)-2g(1); \\ V(i-1,j,k-1)+s(a_i,c_k)-2g(1); \\ V(i,j-1,k-1)+s(b_j,c_k)-2g(1); \\ V(i-1,j,k)-2g(1); \\ V(i,j-1,k)-2g(1); \\ V(i,j,k-1)-2g(1); \end{cases}$$

■

From the above theorem, $V(n_1, n_2, n_3)$ can be evaluated in $O(n_1n_2n_3)$ time [Murata et al.85].

3.2.2 MSA method

Carrillo and Lipman [Carrillo and Lipman 88] suggested a way to reduce some of the work needed in computing the optimal SP multiple alignment in practice. An extension of their idea was implemented in the program called MSA [Lipman et al. 89]. Additional refinements to MSA were reported by Gupta [Gupta et al. 95].

The MSA program implements a complex variation of Dijkstra's single-source shortest-paths algorithm [Dijkstra 59] [Gupta et al. 95]. MSA formulated the multiple

alignment problem as a single-source (s), single-destination (t) shortest-paths problem in the weighted, directed graph $G = (V, E)$ corresponding to a multiple alignment table. The number of vertices is the product of the sequence lengths. A path P starting at the source vertex $s = <0,\ldots,0>$ that ends at vertex $t = <n_1, \ldots, n_k>$ represents an alignment of $S_i[1\ldots n_i]$ for $1 \leq i \leq k$. There is a one-to-one correspondence between edges in the path and the columns in the alignment.

For the problem of aligning three sequences of n characters each, the directed graph has roughly n^3 nodes and $7n^3$ edges, and the shortest-path algorithm will explore each one of the edges [Cormen et al.92]. As Carrillo and Lipman stated, to solve the SP alignment problem, not all alignments has to be considered [Carrillo and Lipman 88].

The MSA program adopts the Carrillo and Lipman method [Carrillo and Lipman88] to calculate a lower bound L and an upper bound U to restrict the number of paths to be considered.

L, the lower bound on the cost of the multiple alignment of S, can be defined as the sum of distances of optimal pairwise alignments for each pair in S.

Let $S = S_1S_2\ldots S_n$ denote n sequences. Let $d(S_i, S_j)$ denote the distance for the optimal alignment of sequences S_i and S_j, then,

$$L = \sum_{i<j} d(S_i, S_j) \text{ where } 1 \leq i, j \leq n$$

This value of L is a lower bound because it assumes that each pair of sequences is aligned optimally, independent of the other sequences.

To get the value of the upper bound, U, one option is for the user to define one. If U is not provided by the user, the program computes a heuristic multiple sequence

alignment from pairwise alignments, and U is then induced from the cost of that heuristic alignment.

MSA searches only among alignments whose cost is $\leq U$. Theorem 3.2 is implemented by MSA to restrict the paths to be searched.

Theorem 3.2 [Carrillo and Lipman88]

Let $S = S_1S_2\ldots S_n$ denote n sequences, and $d(S_i, S_j)$ denote the minimum distance between sequences S_i and S_j. Let M be a multiple sequence alignment of minimum cost, and $c(M_{i,j})$ denote the cost of the i,jth sequences of the alignment M. Then

$$c(M_{i,j}) \leq d(S_i, S_j) + U - L \tag{1}$$

Proof:

$$\begin{aligned}
&c(M) \leq U \\
&\Rightarrow c(M) - L \leq U - L \\
&\Rightarrow \sum_{i<j} (c(M_{i,j}) - d(S_i, S_j)) \leq U - L \\
&\Rightarrow c(M_{i,j}) - d(S_i, S_j) \leq U - L \\
&\Rightarrow c(M_{i,j}) \leq d(S_i, S_j) + U - L
\end{aligned}$$

■

Inequality (1) above theorem restricts consideration to those paths whose projections onto the planes defined by all pairs S_i and S_j have cost at most $d(S_i, S_j) + U - L$.

The value of U is critical in the program because if U is too small, the program will fail to find any feasible solution, and, if U is too big, more time and space will be required for program execution.

It is reported that MSA can align six strings of lengths around 200 characters in a practical amount of time [Lipman et al. 89]. MSA performs better if one begins with an

extremely good value for U. However, it is not likely that MSA will make it practical to optimally align tens or hundreds of sequences [Lipman et al. 89].

3.3 Progressive Alignment Method

The most widely used approach to multiple sequence alignment is progressive alignment. A progressive alignment method utilizes pairwise alignment algorithm iteratively to achieve the multiple alignment of a set of protein sequences [Feng and Doolittle 96].

The progressive method is based on the concept of evolutionary trees [Stryer 95]. The dominative view of the evolution of life is that the high level history of life is ideally organized and displayed as a rooted, directed tree. In the progressive method, usually a guide tree is generated based on cluster analysis from pairwise alignment between all pairs of sequences. Guide tree is a rooted binary tree whose leaves represent sequences and whose interior nodes represent alignments. The root node represents a complete multiple alignment. [Feng and Doolittle 96].

Progressive alignment is heuristic: it does not directly optimize any global scoring function of alignment correctness. The most important heuristic of progressive alignment algorithm is to align the most similar pairs of sequences first. These are the most reliable alignments.

The advantage of progressive alignment is that it is fast and efficient, and in many cases the resulting alignments are reasonable. Many authors have proposed methods based on progressive alignment [Feng and Doolittle 87] [Taylor 88] [Higgins et al. 96] [Taylor 96]. The published methods vary in the way they choose the specified order to do

the alignment, and in the procedure used to align and score the sequences against existing alignments. Two of the most popular progressive methods, Feng-Doolittle algorithm and Clustal W, are introduced in the following sections.

3.3.1 Feng-Doolittle Algorithm

The Feng-Doolittle algorithm [Feng and Doolittle 87] was one of the first progressive alignment algorithms.

The Feng-Doolittle algorithm consists of the following three steps.

Step 1. Calculate a matrix of N(N-1)/2 distances between all pairs of N sequences by standard pairwise alignment, converting similarity scores to distances.

Distance of a pair of aligned sequences is converted from similarity scores. In Feng-Doolittle algorithm, distances are approximate values calculated by Poisson equation [Feng et al. 85]:

$$D = -\ \ln S \qquad (1)$$

where

$$S = \frac{S_{real}(i,j) - S_{rand}(i,j)}{S_{iden}(i,j) - S_{rand}(i,j)} \times 100 \qquad (2)$$

$S_{real}(i, j)$ is the observed similarity score for sequences being aligned. $S_{iden}(i, j)$ is the average of the score of aligning either sequence to itself. $S_{rand}(i, j)$ is the expected score for aligning two random sequences of the same length and residue composition. Random sequences are determined by a computer-intensive random shuffling or an approximate calculation provided by Feng and Doolittle [Feng and Doolittle 96]. Thus, S is a normalized percentage similarity; it is expected to roughly decay exponentially towards zero with increasing evolutionary distance.

Step 2. Construct a guide tree from the distance matrix using the clustering algorithm [Fitch and Margoliash 67].

The Feng-Doolittle algorithm constructs a guide tree based directly on the Fitch-Margoliash algorithm [Fitch and Margoliash 67], which is one of the fast clustering algorithms that build evolutionary trees from distance matrices. First, the smallest mutation distance score is identified and the two sequences involved are connected as sibling nodes. The parent node is the union of the two identified sequences. A new matrix is constructed that contains the average distances between members of the first pair (parent node) and the remaining members of the set. This procedure is continued until all scores have been incorporated. Figure 2 shows a guide tree built from a mutation distance table using the Fitch-Margoliash algorithm.

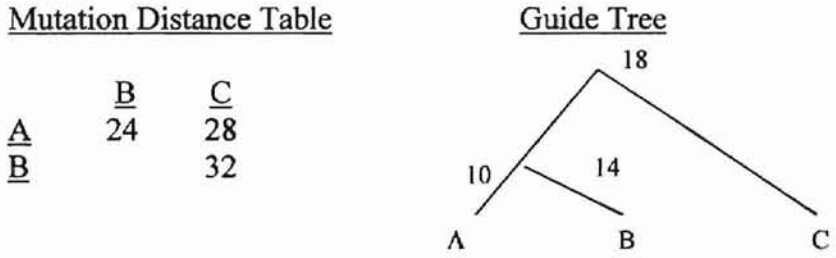

Figure 2. A Sample Phylogenetic Tree [Fitch and Margoliash 67]

Step 3. Starting from the first node added to the tree created in Step 2, align all child nodes in the order that they were added to the tree until all sequences have been aligned.

Sequence to sequence alignment is computed by the usual dynamic programming algorithm. A sequence is added to an existing alignment by aligning it to each sequence in the alignment in turn and the highest pairwise alignment score determines how the

sequence will be added. For aligning an alignment to an alignment, all sequence pairs between the two groups are tried and the best pairwise sequence alignment determines the final alignment.

After an alignment is completed, gap symbols are replaced with neutral elements (Xs). The key to this alignment is that new gaps can be incorporated into either sequence, but the earlier gaps are preserved. This is called the rule of "once a gap, always a gap" by Feng and Doolittle [Feng and Doolittle 87].

3.3.2 Clustal W

The first Clustal program was written by Higgins and Sharp in 1988 [Higgins and Sharp 88] [Higgins and Sharp 89]. Clustal V [Higgins et al. 91], which is a newer release, incorporated profile alignments and the facility to generate trees from the alignments by using the fast Neighbor-Joining method. Clustal W [Thompson et al. 94], the third generation, made some internal modifications to Clustal V, such as sequence weighting, the incorporation of position-specific gap penalties, and flexible weight matrix choices. All these various additional heuristics contribute to its accuracy.

The algorithm is as follows:

Step 1. Construct a distance matrix of all pairs by fast, approximate alignment algorithm [Wilbur and Lipman 83], followed by converting the similarity scores to evolutionary distances using the model of Kimura [Kimura 80].

The distances calculated are related to the degree of divergence (defined as 100% - percent identity) between the sequences. Clustal W uses Kimura's method [Kimura 80] to calculate protein distances.

Distance = $-\ln(1.0 - K - (K * K/5.0))$

where K is the degree of divergence (percent divergence / 100).

Step 2. Construct a guide tree using a neighbor-joining (NJ) clustering algorithm by Saitou and Nei [Saitou and Nei 87].

The neighbor-joining (NJ) method works on the distance matrix between all pairs of sequences to be analyzed. The NJ method works by constructing a tree T by steps, keeping a list L of active nodes in this tree [Saitou and Nei 87].

The NJ method calculates distance between two clusters as follows.

$$D_{i,j} = d_{i,j} - (r_i + r_j)$$

where

$$r_i = \frac{1}{|L| - 2} \sum_{k \in L} d_{i,k}$$ where $|L|$ denotes the size of the set L.

The NJ algorithm is given below in Figure 3.

Initialization:

Define T to be the set of leaf nodes, one for each given sequence, and let $L = T$.

Iteration:

Pick a pair i, j in L for which D_{ij} is minimal.

Define a new node k and set $d_{km} = (d_{im} + d_{jm} + d_{ij})/2$, for all m in L.

Add k to T with edges of lengths $d_{ik} = (d_{ij} + r_i - r_j)$ and $d_{jk} = d_{ij} - d_{jk}$, joining k to i and j, respectively.

Remove i and j from L, and add k.

Termination:

When L consists of two leaves i and j, add the remaining edge between i and j with length d_{ij}.

Figure 3. Neighbor-Joining Method [Saitou and Nei 87]

The principle of the NJ method is to find pairs of neighbors that minimize the total branch length at each stage of clustering of neighbors. A pair of neighbors is a pair of nodes connected through a single interior node in an unrooted, bifurcating tree.

The NJ method constructs a tree starting with a star-like tree, as given in Figure 4(a). In practice, some pairs of nodes are more closely related to each other than other pairs are. In the tree given in Figure 4(b), there is only one interior branch, XY, which connects the paired nodes 1 and 2 and the others that are connected by a single node Y. Any pair of nodes can take the position of 1 and 2 in the tree, and there are N(N-1)/2 ways of choosing them. Among these possible pairs of nodes, the one that gives the smallest sum of branch lengths is chosen. This pair is then regarded as a single node. This procedure is continued until all N-3 interior branches are found.

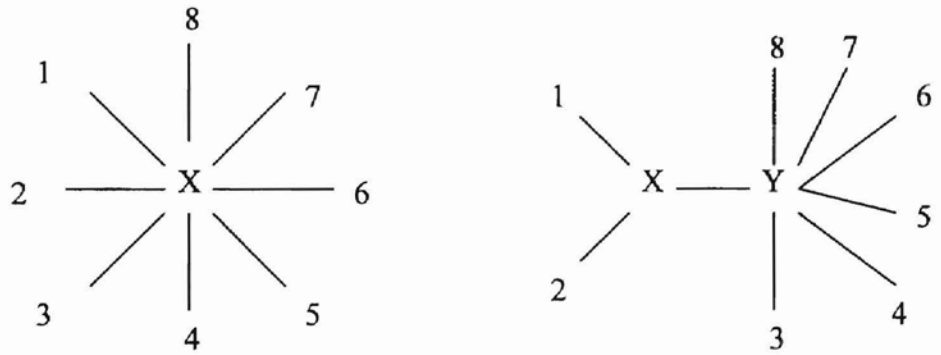

Figure 4. Star-like Tree and Neighbor-Joining in the NJ Method.
(a) A star-like tree with no hierarchical structure.
(b) A tree in which nodes 1 and 2 are clustered.

Step 3. Progressively align at nodes in order of decreasing similarity, using sequence-sequence, sequence-profile, and profile-profile alignment [Gribskov et al. 87].

After the tree is constructed, alignments are built progressively following the branch order in the tree. At each alignment stage, the algorithm of Myers and Miller [Myers and Miller 88] is used.

Sequence-alignment and alignment-alignment alignment are calculated using the profile alignment method, which is a simple extension of the profile method [Gribskov et al. 87]. In profile alignment, the alignment is treated as a single sequence, except that the score is calculated at aligned positions as the average weight matrix score of all the residues in one alignment versus all those in the other. Any gaps that are introduced are placed in all of the sequences of an alignment at the same position.

CHAPTER IV

THE NEW METHOD

The algorithm which implements the new method follows the general strategy of iteratively merging two multiple alignments of two subsets of sequences into a single multiple alignment of the union of those subsets. This algorithm builds an evolutionary tree from sequence data while simultaneously constructing an evolutionary informative multiple alignment.

The algorithm has two steps:

1) Construct a similarity matrix of all pairs.

2) Construct an evolutionary tree while simultaneously construct the multiple alignment.

4.1 Construct a Similarity Matrix

To construct the similarity matrix, we need to calculate a matrix of $n(n\text{-}1)/2$ similarity scores between all pairs of n sequences. Similarity scores of pairwise alignments are computed using Gotoh's algorithm [Gotoh 82], which was introduced in Chapter II.

Since the alignment score between two sequences of lengths not more than m can be computed in $O(m^2)$ time using a dynamic programming algorithm [Cormen et al. 92], the time complexity for constructing the similarity matrix of d sequences of lengths at most m is $O(d^2m^2)$ and the space required is $O(m^2)$ [Cormen et al. 92].

The example below is a set of sequences S. This example will be used through all the following sections to illustrate the new method.

S_1 : RPCVCPVLRQAAQQVLQRQIIQGPQQLRRLFAA
S_2 : RPCACPVLRQVVQQALQRQIIQGPQQLRRLFAA
S_3 : KPCLCPKQAAVKQAAHQQLYQGQLQGPKQVRRAFRLL
S_4 : KPCVCPRQLVLRQAAHLAQQLYQGQRQVRRAFVA
S_5 : KPCVCPRQLVLRQAAHQQLYQGQRQVRRLFAA

Figure 5. A Sample of a Set of Sequences

The similarity matrix for the set S is as given in Table VII below. PAM-250 matrix [Dayhoff et al. 78] (described in Chapter I) is used as the scoring system. Gap open penalty is 12 and gap extension penalty is 4.

Table VII. Similarity Matrix Calculated for the Example in Figure 5

	S_1	S_2	S_3	S_4	S_5
S_1	-	160	66	83	85
S_2	160	-	60	75	91
S_3	66	60	-	86	94
S_4	83	75	86	-	147
S_5	85	91	94	147	-

4.2 Construct an Evolutionary Tree

Same as the progressive alignment method, this new method is based on the concept of evolutionary trees [Stryer 95]. The evolutionary tree is a rooted binary tree. Let S be a set of n sequences. The evolutionary tree for S is a tree T with the following properties:

1) T contains n leaves, one for each sequence in S.

2) Each internal node of T presents a sequence alignment and has at l children.

3) The root contains the final multiple alignment.

The new algorithm builds the tree T based on cluster analysis of the s matrix. It begins with a set of n leaves and performs a sequence of n-1 operations to create the final evolutionary tree, and simultaneously to cons multiple sequence alignment.

The new algorithm is given below in Figure 6.

Initialization:

Define T to be the set of leaf nodes, one for each given sequence.

Iteration:

1) Pick a pair i, j from T for which $s(i, j)$ is maximal according to the s matrix.

2) Construct a new node A, which has i and j as its left child and rig respectively, and contains the alignment of i and j. Calculate the c string A_c for the alignment. Reconstruct the similarity matrix, setting s $s(A_c, m)$, for all m in T.

3) Add A to T.

Termination:

After n-1 loops, T will be a single full binary Tree. The root of T will con final multiple alignement.

Figure 6. The New Method: Constructing an Evolutionary Tree

For the first step, it takes $O(n^2)$ time to pick the maximum score from the similarity matrix for each iteration, so for n-1 iterations, it takes time $O(n^3)$.

Calculating the consensus string, re-constructing the similarity matrix, and computing multiple sequence alignment, which are the three critical points of the new algorithm given in Figure 6, are described in details in the following three subsections.

4.2.1 Calculating the Consensus Sequences

The consensus sequence is a consensus representation of the critical common features of a set of sequences [Barton and Anderson 86].

There is not known method to find the consensus sequence. In terms of multiple alignments, the problem of finding consensus sequence is based on definition of the consensus character.

Given a multiple alignment M of a set of sequences S, the consensus character of column i of M is the character that minimizes the summed distance to it from all the characters in column i.

Since the alphabet is finite, a consensus character for each column of M exists and can be found by enumeration. As one simple special case, if the pairwise scoring scheme scores a match with a zero and a mismatch or a space opposite a character with a one, then the consensus character in column i is the plurality character (i.e., the character occurring most often in column i). Note that the character can be a space.

A consensus character represents the critical characteristic of a column in a multiple alignment, but if the column is not highly conserved, the resulting consensus

character cannot faithfully reflect the characteristic of the column. A better solution would be to change the consensus character to the neutral character "X" [Melcher 00].

The consensus sequence S_M derived from alignment M is the concatenation of the consensus characters for each column of M. Note that S_M need not be from M and generally will not be [Barton and Anderson 86].

It is common in the computational biology to compute a multiple alignment M and then represent those sequences by the consensus sequence derived from M [Gusfield 97]. It is then natural to use the goodness of the consensus sequence as a way to evaluate the goodness of the multiple alignment M.

In the new algorithm, using consensus sequences as representatives of multiple alignments, the similarity score between two clusters can be computed by calculating the similarity score between two corresponding consensus sequences.

It takes $O(mn^2)$ time to compute a consensus sequence of a group of n sequences with length m.

4.2.2 Reconstructing the Similarity Matrix

Let T be the set of clusters of sequences defined in Figure 6. In each iteration, we first pick one pair of clusters, i and j, from T. Then a new node A, which contains the alignment of i and j, is created and added to T.

Accordingly, row i, row j, column i, and column j are removed from the similarity matrix (Section 4.1). A new row and a new column for A are added to the similarity matrix. Then we have to compute the similarity scores between A and all other nodes in the matrix.

Table VIII. Similarity Matrix After the First Iteration for the Example in Figure 5

	A_1	S_3	S_4	S_5
A_1	-	64	79	83
S_3	64	-	86	94
S_4	79	86	-	147
S_5	83	94	147	-

For our example in Figure 5, after the first iteration, Table VII will be updated to Table VIII. S_1 and S_2 will be removed from the table, and A_1, parent of S_1 and S_2, will be added to the table.

4.2.3 Constructing Multiple Sequence Alignment

Every internal node of tree T (Figure 6) contains the result of the pairwise alignment of its two children. The pairwise alignments are calculated as follows.

Sequence-to-sequence alignment is done using the standard pairwise alignment algorithm [Gotoh 82]. If any of the children contains an alignment, the consensus sequence for the alignment is calculated using the method illustrated in Section 4.2.1. Gap symbols in the resulting consensus sequences are replaced with neutral elements (Xs). So, sequence-alignment and alignment-alignment alignments can also be calculated using the standard pairwise alignment algorithm.

Given two child nodes $\{A, B\}$, which are both alignments of sequences, it takes five steps to construct the alignment of A and B.

1. Calculate the consensus sequences for the alignment of A and B. We refer to those two consensus sequences as a and b. Replace spaces in sequences a and b with neutral elements, e.g., Xs.

2. Calculate an optimal pairwise alignment of sequences a and b. The sequences in this alignment with the gaps are called a' and b'.

3. Insert all gaps from a' that are not already in a into all sequences in A.

4. Insert all gaps from b' that are not already in b into all sequences in B.

5. Merge A and B.

In the previous sections, we have discussed the details of the new algorithm given in Figure 6. For n sequences, the algorithm will take n-1 iterations to construct the evolutionary tree. For our example, the algorithm proceeds as shown in Figure 7.

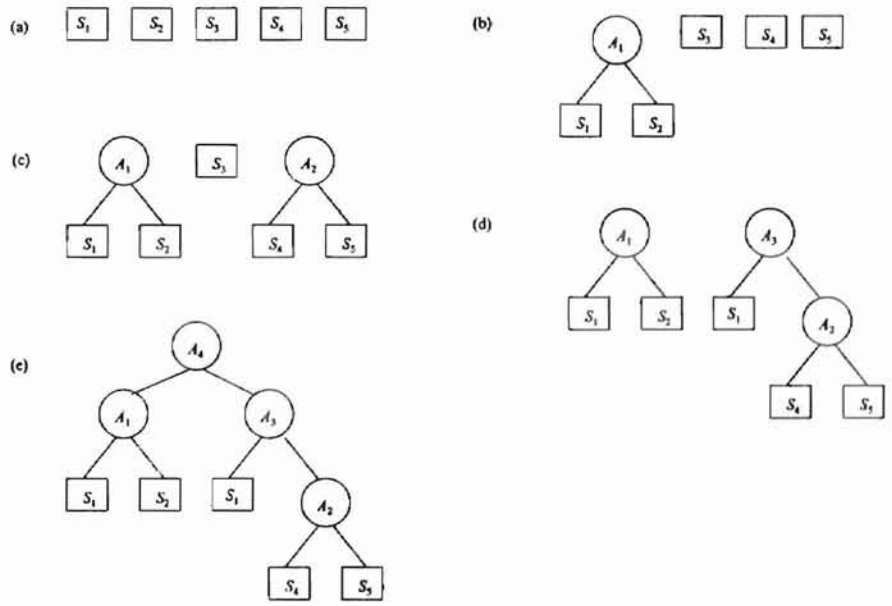

Figure 7. Process of Constructing an Evolutionary Tree for the Example in Figure 5

CHAPTER V

IMPLEMENTATION AND RESULTS

5.1 Implementation

The algorithm has been implemented in the C++ language. The source code listed in Appendix B has been compiled and tested under Microsoft Virtual C++ 6.0 on Windows 98, and under the Solaris 7 system.

To use this program under Unix, put all source files under one directory, and enter "make msa" after the command prompt. A new executable file named "msa" will be created. To run the program, simply enter "msa" after the prompt and follow the instructions.

To run this program through a web browser such as Netscape or Microsoft IE, use the following URL:

http://gradweb.cs.okstate.edu/~lke/msa/intro.html

You will be asked to provide your choices of gap open penalty, gap extension penalty, type of the scoring matrix, and the input sequences.

1. Input Format

Sequences can be in upper or lower cases. The only symbols recognized are 20 amino acid characters: A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y. All other symbols will be ignored.

The program recognizes and uses the FASTA format ([Pearson and Lipman 88]). The sequences are delimited by an angle bracket ">" in column 1. The text immediately after the ">" is used as a title. Every thing on the following line until the next ">" or the end of file is one sequence.

2. Gap Open Penalty

The default is 12. Reduce this to encourage gaps of all sizes, and increase it to discourage them.

3. Gap Extension Penalty

The default is 4. Reduce this to encourage longer gaps, and increase it to shorten them.

4. Scoring Matrix

The default is Dayhoff PAM 250 matrix [Dayhoff et al.78]. We also offer a Blosum 62 matrix [Henikoff and HeniKoff 92] and an identity matrix.

5.2 Results

To show a real example we took eight sequences of bacterial porin [Jeanteur et al. 91]. The lengths of the sequences range from 329 to 347. Figures 8 and 9 illustrate the eight protein sequences aligned by the Clustal W and this method, respectively.

It can be observed that both methods have correctly identified most of the matched units and discovered the overall relationships among the sequences. The alignment by this method is slightly more compact than by the Clustal W method. The former is 392 columns long and the latter is 394 columns long.

Figure 8. Alignment of Eight Protein Sequences by Clustal W. ('*' indicates a match tuple).

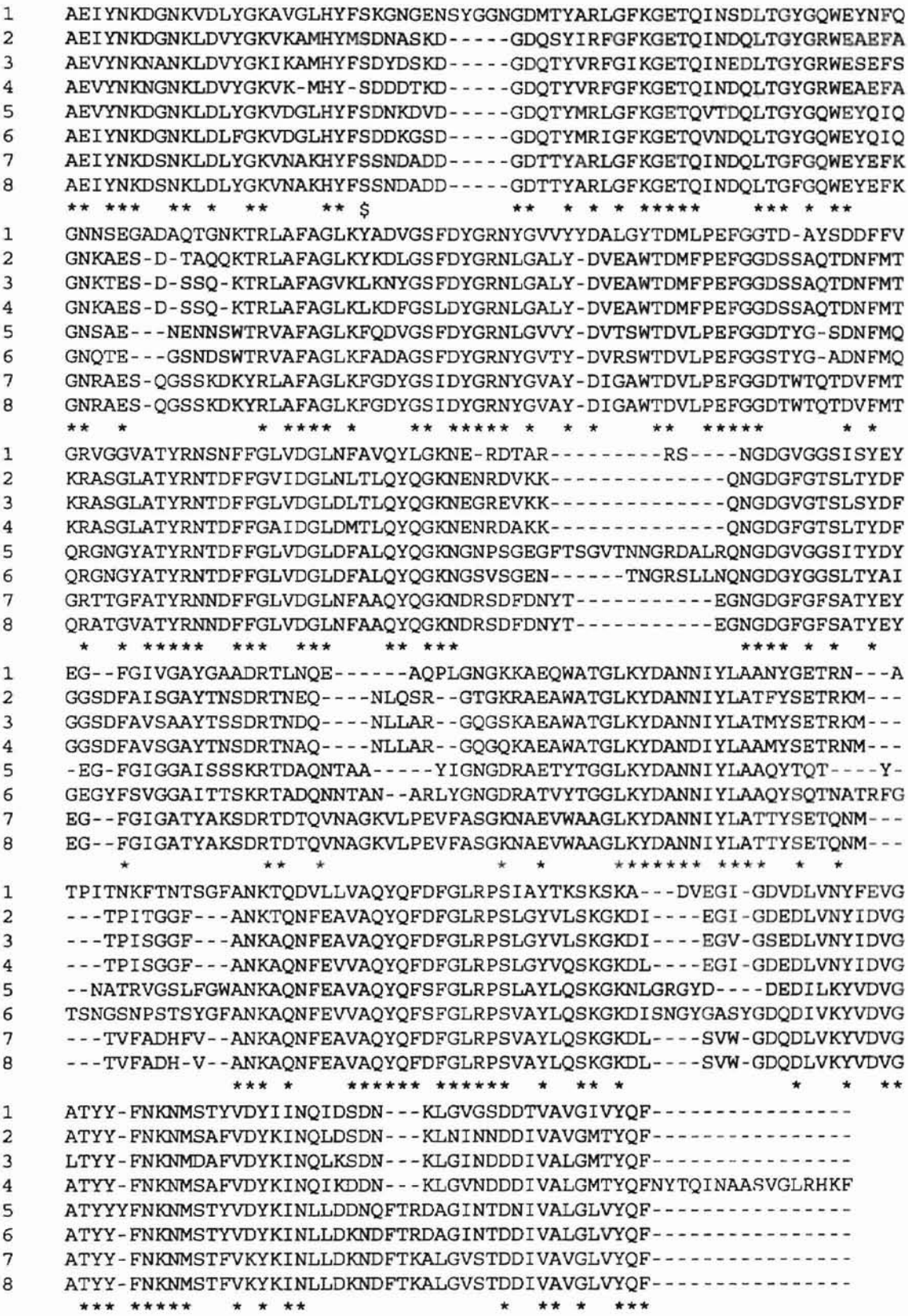

Figure 9. Alignment of Eight Protein Sequences by the New Method.

The score obtained with the new algorithm is very close to the score obtained from Clustal W. The result of the new algorithm is slightly better than that of Clustal W at some positions. As a specific example, the new algorithm found the conserved residue 'S' at column 24.

With the above example we do not intend to prove that the new method performs generally better than Clustal W, but we do want to show that the new method is usable and produces good results.

5.3 Discussion

In this study a new method was presented for multiple sequence alignment. Compared to other iterative algorithms, this method has two major properties.

First, this algorithm calculates the distance between two groups of sequences in an efficient and biologically meaningful way. The Feng-Doolittle's algorithm, which was introduced in Section 3.3.1, uses two original sequences, which have the minimum distance between two groups. The problem of the Feng-Doolittle's algorithm is that the original sequence cannot always present the major characteristics of a group, so the result may not be accurate for general cases. In the ClustalW algorithm, the group-group alignment is based on the average distance, and the new multiple alignment is created by aligning profiles. The disadvantage of Clustal W is that it needs a lot of memory space to store the profiles in order to compute the profile alignments.

In the new method, a consensus sequence is produced as the representative of a group of sequences. The distance between two groups is defined as the distance between the two corresponding consensus sequences. Since by definition the consensus sequence

minimizes the summed distance to it from all the sequences in the group, the consensus sequence is the center of the group. Therefore, the new algorithm solves the problem of Feng-Doolittle's algorithm because the consensus sequence, instead of an original sequence, more faithfully reflects the variations of a group of sequences. Moreover, since a group of sequences can be represented by a single sequence, the computation of the distance and alignment between two groups can be simplified to sequence-sequence distance and sequence-sequence alignment. Compared to Clustal W, it is more time and space efficient.

Second, this algorithm combines the steps of constructing the evolutionary tree and the computation of the multiple alignments. The progressive method calculates a guide tree first, then the multiple alignments are built based on the tree structure. In the new algorithm, multiple alignment can be done with reference to the evolutionary tree, even though the tree structure itself may remain unknown.

In conclusion, this algorithm is fast and efficient, and can produce reasonable and meaningful results. It also has the advantages of easy implementation and relatively simple data structures.

5.4 Future Work

In the new algorithm, each internal node of the evolutionary tree is represented by a consensus sequence, which inevitably leads to a loss of some information. Other objective functions may be added to keep more information about the alignment. Additional information such as predicted secondary structures or propensities of gap formation may be supplemented to the consensus sequence as well.

The implementation of the new algorithm uses Gotoh's algorithm to compute pairwise sequence alignments, which requires $O(mn)$ space to align two sequences of lengths m and n. The new algorithm also stores a similarity table of size $n*n$ to calculate the closely related pairs among n sequences. Further refinement may be made to reduce the space requirements.

REFERENCES

[Alberts et al. 94] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson, *Molecular Biology of the Cells*, Galdand Publishers, New York, NY, 1994.

[Altschul and Erickson 86] Stephen F. Altschul and Bruce W. Erickson, "Optimal Sequence Alignment Using Affine Gap Costs", *Bulletin of Mathematical Biology*, Vol. 48, No. 5/6, pp. 603-616, August 1986.

[Altschul 89] Stephen F. Altschul, "Gap Costs for Multiple Sequence Alignment", *Journal of Theoretical Biology*, Vol. 138, No. 3, pp. 297-309, June 1989.

[Altschul 91] Stephen. F. Altschul, "Amino Acid Substitution Matrices from an Information Theoretic Perspective", *Journal of Molecular Biology*, Vol. 219, No.3, pp. 555-565, June 1991.

[Altschul 93] Stephen F. Altschul, "A Protein Alignment Scoring System Sensitive at All Evolutionary Distances", *Journal of Molecular Evolution*, Vol. 36, No. 3, pp. 290-300, March 1993.

[Bacon and Anderson 86] David J. Bacon and Wayne F. Anderson, "Multiple Sequence Alignment", *Journal of Molecular Biology*, Vol. 191, No. 2, pp. 153-161, September 1986.

[Barton 90] Geoffrey J. Barton, "Protein Multiple Sequence Alignment and Flexible Pattern Matching", In *Methods in Enzymology, Volume 183: Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, Russell F. Doolittle (Ed.), Academic Press, pp. 403-428, San Diego, CA, 1990.

[Barton 96] Geoffrey J. Barton, "Protein Sequence Alignment and Database Scanning", In *Protein Structure Prediction: A Practical Approach*, Michael J. E. Steinberg (Ed.), Oxford University Press, New York, NY, 1996.

[Barton and Russell 93] Geoffrey J. Barton and Robert B. Russell, "Protein Structure Prediction", *Nature*, Vol. 361, No. 6412, pp. 505-506, February 1993.

[Bellman 57] Richard E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, NJ, 1957.

[Carrillo and Lipman 88] Humberto Carrillo and David Lipman, "The Multiple Sequence Alignment Problem in Biology", *SIAM Journal on Applied Mathematics*, Vol. 48, No. 5, pp. 1073-1082, October 1988.

[Cormen et al. 92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press and McGraw-Hill, Cambridge, MA, 1992.

[Dayhoff et al. 78] Margaret O. Dayhoff, R. M. Schwartz, and B. C. Orcutt, "A Model of Evolutionary Change in Proteins", In *Atlas of Protein Sequence and Structure, Volume 5, Supplement 3*, M. O. Dayhoff (ed.), National Biomedical Research Foundation, pp. 345 – 352, Washington D. C., 1978.

[Dijkstra 59] E. W. Dijkstra, "A Note on Two Problems in Connexion with Gaps", *Numerische Mathematik*, Vol. 1, No. 1, pp. 269-271, November 1959.

[Doolittle 81] Russell F. Doolittle, "Similar Amino Acid Sequence: Chance or Common Ancestry?", *Science*, Vol. 214, No. 4517, pp. 385-396, October 1981.

[Durbin et al. 98] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press, Cambridge, UK, 1998.

[Eddy 95] Sean R. Eddy, "Multiple Alignment Using Hidden Markov Models", *Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology*, pp. 114-120, Cambridge, UK, July 1995.

[Feng et al. 85] D. F. Feng, M. S. Johnson, and R. F. Doolittle, "Aligning Amino Acid Sequences: Comparison of Commonly Used Methods", *Journal of Molecular Evolution*, Vol. 21, No. 2, pp. 112-125, February 1985.

[Feng and Doolittle 87] Da-fei Feng and Russell F. Doolittle, "Progressive Sequence Alignment as a Prerequisite to Correct Phylogenetic Trees", *Journal of Molecular Evolution*, Vol. 25, No. 4, pp. 351-360, August 1987.

[Feng and Doolittle 96] Da-fei Fend and Russell F. Doolittle, "Progressive Alignment of Amino Acid Sequences and Construction of Phylogenetic Trees from Them", In *Methods in Enzymology, Volume 266: Computer Methods for Macromolecular Sequences Analysis*, Russell F. Doolittle (Ed.), Academic Press, pp. 368-382, San Diego, CA, 1996.

[Fitch and Margoliash 67] Walter M. Fitch and Emanuel Margoliash, "Construction of Phylogenic Trees", *Science*, Vol. 155, No. 3760, pp. 279-284, January 1967.

[Fitch and Smith 83] Walter M. Fitch and Temple F. Smith, "Optimal Sequence Alignments", *Proceedings of the National Academy of Sciences, USA*, Vol. 80, No. 5, pp. 1382-1386, March 1983.

[George et al. 90] David G. George, Winona C. Barker, and Lois T. Hunt, "Mutation Data Matrix and Its Uses", In *Methods in Enzymology, Volume 183: Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences*, Russell F. Doolittle (Ed.), Academic Press, pp. 333-351, San Diego, CA, 1990.

[Gilbert 91] Walter Gilbert, "Towards a Paradigm Shift in Biology", *Nature*, Vol. 349, No. 6305, p. 99, January 1991.

[Gonnet et al. 92] Gaston H. Gonnet, Mark A. Cohen, and Steven A. Benner, "Exhaustive Matching of the Entire Protein Sequence Database", *Science*, Vol. 256, No. 5062, pp. 1443-1445, June 1992.

[Gotoh 82] Osamu Gotoh, "An Improved Algorithm for Matching Biological Sequences", *Journal of Molecular Biology*, Vol. 162, No. 3, pp. 705-708, December 1982.

[Gotoh 86] Osamu Gotoh, "Alignment of Three Biological Sequences with an Efficient Traceback Procedure", *Journal of Theoretical Biology*, Vol. 121, No. 3, pp. 327-337, August 1986.

[Gribskov et al.87] M. Gribskov, A. McLachlan, and E. Eisenberg, "Profile Analysis Detection of Distantly Related Proteins", *Proceedings of the National Academy of Sciences, USA*, Vol. 88, No. 11, pp. 4355-4358, November 1987.

[Guo 00] Yanwen Guo, "Dynamic Programming and Its Application to Pairwise Biology Sequence Alignment", Master of Science Thesis, Computer Science Department, Oklahoma State University, May 2000.

[Gupta et al. 95] Sandeep K. Gupta, John D. Kececioglu, and Alejandro A. Schäffer, "Improving the Practical Space and Time Efficiency of the Shortest-Paths Approach to Sum-of-Pairs Multiple Sequence Alignment", *Journal of Computational Biology,* Vol. 2, No. 3, pp. 459-472, March 1995.

[Gusfield 97] Dan Gusfield, *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, New York, NY, 1997.

[Henikoff and Henikoff 92] Steven Henikoff and Jorja G. Henikoff, "Amino Acid Substitution Matrices from Protein Blocks", *Proceedings of the National Academy of Sciences, USA,* Vol. 89, No. 22, pp. 10915 – 10919, November 1992.

[Henikoff and Henikoff 93] Steven Henikoff and Jorja G. Henikoff, "Performance Evaluation of Amino Acid Substitution Matrices", *Proteins*, Vol. 17, No. 1, pp. 49-61, January 1993.

[Higgins and Sharp 88] Desmond G. Higgins and Paul M. Sharp, "CLUTAL: a Package for Performing Multiple Sequence Alignment on a Microcomputer", *Gene*, Vol. 73, No. 1, pp. 237-244, December 1988.

[Higgins and Sharp 89] Desmond G. Higgins and Paul M. Sharp, "Fast and Sensitive Multiple Sequence Alignment on a Microcomputer", *Computer Applications in the Biosciences: CABIOS*, Vol. 5, No. 1, pp. 151-153, December 1989.

[Higgins et al. 91] Desmond G. Higgins, A. J. Bleasby, and R. Fuchs, "CLUSTAL V: Improved Software for Multiple Sequence Alignment", *Computer Applications in the Biosciences: CABIOS*, Vol. 8, No. 1, pp. 189-191, December 1991.

[Higgins et al. 96] Desmond G. Higgins, Julie D. Thompson, and Toby J. Gibson, "Using CLUSTAL for Multiple Sequence Alignments", In *Methods in Enzymology, Volume 266: Computer Methods for Macromolecular Sequences Analysis*, Russell F. Doolittle (Ed.), Academic Press, pp. 383-401, San Diego, CA, 1996.

[Jeanteur et al. 91] D. Jeanteur, J. H. Lakey, and F. Pattus, "The Bacterial Porin Superfamily: Sequence Alignment and Structure Prediction", *Molecular Microbiology*, Vol. 5, No.4, pp. 2153-2164, October 1991.

[Karlin and Altschul 90] Samuel Karlin and Stephen F. Altschul, "Methods for Assessing Statistical Significance of Molecular Sequence Features by Using General Scoring Schemes", *Proceedings of the National Academy of Sciences, USA*, Vol. 87, No. 6, pp. 2264-2268, March 1990.

[Kimura 80] Motoo Kimura, "A Simple Method for Estimating Evolutionary Rates of Base Substitutions Through Comparative Studies of Nucleotide Sequences", *Journal of Molecular Biology*, Vol. 16, No. 2, pp. 111-120, December 1980.

[King and Wilson 75] M. C. King and A. C. Wilson, "Evolution at Two Levels in Humans and Chimpanzees", *Science*, Vol. 188, No. 4184, pp. 107-116, April 1975.

[Krogh et al. 94] Anders Krogh, Michael Brown, I. Saira Mian, Kimmen Sjolander, and David Haussler, "Hidden Markov Models in Computational Biology: Applications to Protein Modeling", *Journal of Molecular Biology*, Vol. 235, No. 5, pp. 1501-1531, February 1994.

[Lehninger et al. 93] Albert L. Lehninger, David L. Nelson, and Mechael M. Cox, *Principles of Biochemistry*, Worth Publishers, New York, NY, 1993.

[Lipman et al. 89] David J. Lipman, Stephen F. Altschul, and John D. Kececioglu, "A Tool for Multiple Sequence Alignment", *Proceedings of the National Academy of Sciences USA*, Vol. 86, No. 6, pp. 4412-4415, June 1989.

[Marks et al. 96] Dawn B. Marks, Allan D. Marks, and Colleen M. Smith, *Basic Medical Biochemistry*, Williams & Wilkins, Baltimore, MD, 1996.

[Mclachlan 71] A. D. Mclachlan, "Tests for Comparing Related Amino-Acid Sequences Cytochrome *c* and Cytochrome c_{551}", *Journal of Molecular Biology*, Vol. 61, No. 2, pp. 409-424, October 1971.

[Melcher 00] Ulrich Melcher, "The '30K' Superfamily of Viral Movement Proteins", *Journal of General Virology*, Vol. 81, No. 1, pp. 257-266, January 2000.

[Murata et al. 85] M. Murata, J. S. Richardson, and Joel L. Sussman, " Simultaneous Comparison of Three Protein Sequences", *Proceedings of the National Academy of Sciences, USA*, Vol. 82, No. 4, pp. 3073-3077, May 1985.

[Myers and Miller 88] Eugene W. Myers and Webb Miller, "Optimal Alignments in Linear-Space", *Computer Applications in the Biosciences: CABIOS*, Vol. 4, No. 1, pp.11- 17, August 1988.

[Needleman and Wunsch 70] 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,* Vol. 48, No. 3, pp. 443-453, March 1970.

[Pearson and Lipman 88] William R. Pearson and David J. Lipman, "Improved Tools for Biological Sequence Comparison", *Proceedings of the National Academy of Sciences, USA*, Vol. 85, No. 8, pp. 2444-2448, April 1988.

[Pearson 95] William R. Pearson, "Comparison of Methods for Searching Protein Sequence Databases", *Protein Science*, Vol. 4, No. 6, pp. 1145-1160, June 1995.

[Risler et al. 88] J. L. Risler, H. Delacroix, and A. Henaut, "Amino Acid Substitutions in Structurally Related Proteins: A Pattern Recognition Approach: Determination of a New and Efficient Scoring Matrix", *Journal of Molecular Biology*, Vol. 204, No. 4, pp. 1019-1029, December 1988.

[Saitou and Nei 87] Naruya Saitou and Masatoshi Nei, "The Neighbor-joining Method: A New Method for Reconstructing Phylogenetic Trees", *Molecular Biology and Evolution*, Vol. 4, No. 4, pp. 406-425, July 1987.

[Sankoff and Kruskal 83] D. Sankoff and J. B. Kruskal, Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison, Addison Wesley, MA, 1983.

[Schwartz and Dayhoff 78] R. M. Schwartz and M. O. Dayhoff, "Matrices for Detecting Distant Relationships", In *Atlas of Protein Sequence and Structure, Volume 5, Supplement 3*, M. O. Dayhoff (ed.), National Biomedical Research Foundation, pp. 353 – 358, Washington D. C., 1978.

[Sellers 74] Peter H. Sellers, "On the Theory of Computation of Evolutionary Distances", *SIAM Journal on Applied Mathematics,* Vol. 26, No. 4, pp. 787-793, June 1974.

[Smith et al. 81] T. F. Smith, M. S. Waterman, and W. M. Fitch, "Comparative Biosequence Metrics", *Journal of Molecular Evolution*, Vol. 18, No. 1, pp. 38-46, December 1981.

[Smith and Waterman 81] T. F. Smith and M. S. Waterman, "Identification of Common Molecular Subsequences", *Journal of Molecular Biology*, Vol. 147, No.1, pp. 195-197, March 1981.

[States and Boguski 92] David J. States and Mark S. Boguski, "Similarity and Homology", In *Sequence Analysis Primer*, Michael Gribskov and John Devereux (Eds.), Oxford University Press, New York, NY, 1992.

[Stryer 95] Lubert Stryer, *Biochemistry*, W. H. Freeman, New York, NY, 1995.

[Taylor 88] William R. Taylor, "A Flexible Method to Align Large Numbers of Biological Sequences", *Journal of Molecular Evolution*, Vol. 28, No. 1, pp. 161-169, December 1988.

[Taylor 96] William R. Taylor, "Hierarchical Method to Align Large Numbers of Biological Sequences", In *Methods in Enzymology, Volume 266: Computer Methods for Macromolecular Sequences Analysis*, Russell F. Doolittle (Ed.), Academic Press, pp. 456-474, San Diego, CA, 1996.

[Thompson et al. 94] Julie D. Thompson, Desmond G. Higgins, and Toby J. Gibson, "CLUSTUL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment through Sequence Weighting, Position-Specific Gap Penalties and Weight Matrix Choice", *Nucleic Acids Research*, Vol. 22, No. 22, pp. 4673-4680, November 1994.

[Voet and Voet 95] Danald Voet and Judith G. Voet, *Biochemistry*, John Wiley & Sons, New York, NY, 1995.

[Waterman et al. 76] M. S. Waterman, T. F. Smith, and W. A. Beyer, "Some Biological Sequence Metrics", *Advances in Mathematics*, Vol. 20, No. 3, pp. 367-387, June 1976.

[Waterman 95] Michael S. Waterman, *Introduction to Computational Biology: Maps Sequences and Genomes*, Chapmen and Hall, London, UK, 1995.

[Wilbur and Lipman 83] W. J. Wilbur and David J. Lipman, "Rapid Similarity Searches of Nucleic Acid and Protein Data Banks", *Proceedings of The National Academy of Sciences USA*, Vol. 80, pp. 726-730, February 1983.

APPENDICES

APPENDIX A

GLOSSARY

Alignment	A one-to-one matching of two sequences so that each character in one sequence is associated with a single character of the other sequence or with a null character (gap).
Amino Acid	Any of a class of 20 molecules that are combined to form proteins in living things, also called "residue".
Consensus Sequence	A single sequence that represents the content of a group of aligned sequences.
FASTA	A program that compares a protein sequence to another protein sequence or to a protein database, or a DNA sequence to another DNA sequence or a DNA library.
Gap	A space inserted into a sequence to enhance its alignment with another sequence. Gap are often represented by a '.' or '-' character.
Gap Extension Penalty	The length-dependent term, l, of a gap penalty of the form $w = g + lx$. Where w is the gap penalty, g is the length-independent term, and x the length of the gap.
Gap Opening Penalty	The length-independent term, g, of a gap penalty of the form $w = g + lx$. (see gap extension penalty).
Global Alignment	An optimal alignment that includes all characters from each sequence. Global alignment may miss short regions of high local similarity. Global alignments are most useful for closely related sequences of known homology.
Homology	Similarity attributable to descent from a common ancestor. Homology is an all or none relationship. In molecular biology, homology is often inferred from a high degree of sequence similarity.

Multiple Alignment	Simultaneous alignment of more than two sequences.
PAM	Percent accepted mutations.
Phylogenetic Tree	A hierarchical branching diagram based on morphological similarities where each branch represents a postulated clade or monophyletic group, a group comprised of all the sampled descendants of a single ancestral lineage.
Profile	A profile is a position specific scoring table that represents the information from a family of related sequences.
Progressive Alignment	A multiple alignment algorithm in which the sequences are first clustered and then added one by one, in order of decreasing similarity, to the growing multiple alignment.
Protein	A large molecule composed of one or more chains of amino acids in specific order. Examples are hormones, enzymes, and antibodies.
Scoring Matrix	In a dynamic programming alignment, the score matrix indicates the quality of the alignment ending at each possible pair of residues.
Sequence	The order of amino acids in a protein molecule.
Unitary Matrix	Also known as an identity matrix. A scoring system in which only identical characters received a positive similarity score.

APPENDIX B

CODE LISTINGS

The package for the new algorithm comes as seven C++ source files, five header files, and one makefile. The source code for the package is listed as indicated below.

File 1: makefile

```
install: msa

clean:
        rm *.o

OBJECTS = Common.o PairAlign.o Matrices.o GuideTree.o    \
          ScoreMat.o MultipleAlign.o main.o


HEADERS = Common.h PairAlign.h GuideTree.h    \
         MultipleAlign.h ScoreMat.h

CC      = g++
CFLAGS  = -c -g
LFLAGS  = -O -lm

msa : $(OBJECTS)
        $(CC) -o $@ $(OBJECTS) $(LFLAGS)

.c.o :
        $(CC) $(CFLAGS) $?
```

```
// File 2: main.cpp


#include <iostream>
#include <string>
#include <fstream>
#include <vector>
using namespace std;
#include <stdlib.h>
#include "PairAlign.h"
#include "MultipleAlign.h"

// Simple interface of the program.
// get parameters. open input file, output file and log file.

int main(char ** args)
{
      cout << "Enter input file name: " ;
      string inputFile;
      cin >> inputFile;
      ifstream inFile;
      inFile.open(inputFile.c_str(), ios::in);
      if (!inFile) {
            cerr << inputFile << " could not be opened." << endl;
            exit(1);
      }

      string matName[3] = {"PAM 250 Mutation Matrix",
            "BLOSUM 62 Substitution Matrix", "Unitary Matrix" };
      int mat = 0;
      int gapOpen = 12;
      int gapExt = 4;
      int change;

      do {
            cout << endl << "1. Change Scoring Matrix:        "
                        << matName[mat]<< endl
             << "2. Change Gap Open Penalty:      "<<gapOpen <<endl
             << "3. Change Gap Extension Penalty: "<< gapExt <<endl
             << "4. OK, Comput the Multiple Alignment Now " << endl
             << "5. Exit"  << endl << "Enter your choice: ";
            cin >> change;
            switch (change) {
            case 1:     int m;
                  cout << endl <<"Scoring Matrix Options: "<<endl
                  <<"1. PAM 250 Mutation Matrix" <<endl
                  <<"2. Blosum 62 Substitution Matrix" <<endl
                  <<"3. Unitary Matrix"<<endl
                  <<"Enter your choice: ";
                  cin >> m;
                  if (m != 1 && m != 2 && m != 3)
                        cout << "Your choice " << m
                        <<"is not a valid number" << endl;
                  else
                        mat = m - 1;
                  break;
             case 2:
```

```
                cout << "Enter gap opening penalty: ";
                cin >> gapOpen;
                break;
        case 3:
                cout <<"Enter gap extension penalty: ";
                cin >> gapExt;
                break;
        case 4: break;
        case 5: exit(1);
        default: cout << "Not a valid number. Try again.\n";
        }
    } while ( change != 4 );

    Matrices *sub;
    switch (mat) {
    case 0: sub = new PAM250(); break;
    case 1: sub = new Blosum62(); break;
    case 2: sub = new Unitary(); break;
    }

    string outputFile = inputFile + ".out";
    ofstream outFile;
    outFile.open(outputFile.c_str(), ios::out);
    if (!outFile) {
        cerr << outputFile << " could not be opened." << endl;
        exit(1);
    }

    string Log = inputFile + ".log";
    ofstream logFile;
    logFile.open(Log.c_str(), ios::out);
    if (!logFile) {
        cerr << Log << " could not be opened." << endl;
        exit(1);
    }

    multiAlign MA(inFile, outFile, logFile, sub, gapOpen, gapExt);

    inFile.close();
    outFile.close();
    logFile.close();
    cout << "The final result is printed to file "<< outputFile
        << endl << "and the log file is "<< Log << endl;
    return 0;
}
```

```
//File 3: Matrices.cpp


#include "PairAlign.h"

// This function maps the inputMatrix, which is ordered by
// residuesOrder, to a 26*26 score matrix, which is ordered
// by alphabet.
void Matrices::buildScore(int inputMatrix[][baseSize])
{
      for (int i = 0; i < baseSize; i++) {
        char a = residuesOrder[i];
        for (int j = 0; j<=i; j++) {
                  char b = residuesOrder[j];
                  score[a-'A'][b-'A'] = score[b-'A'][a-'A']
                        = inputMatrix[i][j];
            }
    }
}

// PAM250 Scoring Matrix
PAM250::PAM250() {

int M[baseSize][baseSize] = {
 /* A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  X */
 {  2  },
 { -2, 6  },
 {  0, 0, 2  },
 {  0,-1, 2, 4  },
 { -2,-4,-4,-5,12  },
 {  0, 1, 1, 2,-5, 4  },
 {  0,-1, 1, 3,-5, 2, 4  },
 {  1,-3, 0, 1,-3,-1, 0, 5  },
 { -1, 2, 2, 1,-3, 3, 1,-2, 6  },
 { -1,-2,-2,-2,-2,-2,-2,-3,-2, 5  },
 { -2,-3,-3,-4,-6,-2,-3,-4,-2, 2, 6  },
 { -1, 3, 1, 0,-5, 1, 0,-2, 0,-2,-3, 5  },
 { -1, 0,-2,-3,-5,-1,-2,-3,-2, 2, 4, 0, 6  },
 { -4,-4,-4,-6,-4,-5,-5,-5,-2, 1, 2,-5, 0, 9  },
 {  1, 0,-1,-1,-3, 0,-1,-1, 0,-2,-3,-1,-2,-5, 6  },
 {  1, 0, 1, 0, 0,-1, 0, 1,-1,-1,-3, 0,-2,-3, 1, 2  },
 {  1,-1, 0, 0,-2,-1, 0, 0,-1, 0,-2, 0,-1,-3, 0, 1, 3  },
 { -6, 2,-4,-7,-8,-5,-7,-7,-3,-5,-2,-3,-4, 0,-6,-2,-5,17  },
 { -3,-4,-2,-4, 0,-4,-4,-5, 0,-1,-1,-4,-2, 7,-5,-3,-3, 0,10  },
 {  0,-2,-2,-2,-2,-2,-2,-1,-2, 4, 2,-2, 2,-1,-1,-1, 0,-6,-2, 4 },
 {  0,-1, 0,-1,-3,-1,-1,-1,-1,-1,-1,-1,-1,-2,-1, 0, 0,-4,-2,-1,-1}
};
      buildScore(M);
}


// Blosum62 Mutation Matrix

Blosum62::Blosum62()
{
```

```
    int M[baseSize][baseSize] = {
   /* A  R  N  D  C  Q  E  G  H  I  L  K  M  F  P  S  T  W  Y  V  X  */
    { 4 },
    {-1, 5 },
    {-2, 0, 6 },
    {-2,-2, 1, 6 },
    { 0,-3,-3,-3, 9 },
    {-1, 1, 0, 0,-3, 5 },
    {-1, 0, 0, 2,-4, 2, 5 },
    { 0,-2, 0,-1,-3,-2,-2, 6 },
    {-2, 0, 1,-1,-3, 0, 0,-2, 8 },
    {-1,-3,-3,-3,-1,-3,-3,-4,-3, 4 },
    {-1,-2,-3,-4,-1,-2,-3,-4,-3, 2, 4 },
    {-1, 2, 0,-1,-3, 1, 1,-2,-1,-3,-2, 5 },
    {-1,-1,-2,-3,-1, 0,-2,-3,-2, 1, 2,-1, 5 },
    {-2,-3,-3,-3,-2,-3,-3,-3,-1, 0, 0,-3, 0, 6 },
    {-1,-2,-2,-1,-3,-1,-1,-2,-2,-3,-3,-1,-2,-4, 7 },
    { 1,-1, 1, 0,-1, 0, 0, 0,-1,-2,-2, 0,-1,-2,-1, 4 },
    { 0,-1, 0,-1,-1,-1,-1,-2,-2,-1,-1,-1,-1,-2,-1, 1, 5 },
    {-3,-3,-4,-4,-2,-2,-3,-2,-2,-3,-2,-3,-1, 1,-4,-3,-2,11 },
    {-2,-2,-2,-3,-2,-1,-2,-3, 2,-1,-1,-2,-1, 3,-3,-2,-2, 2, 7 },
    { 0,-3,-3,-3,-1,-2,-2,-3,-3, 3, 1,-2, 1,-1,-2,-2, 0,-3,-1, 4 },
    { 0,-1,-1,-1,-2,-1,-1,-1,-1,-1,-1,-1,-1,-1,-2, 0, 0,-2,-1,-1,-1}
};
      buildScore(M);
}

// Identity Scoring Matrix
Unitary::Unitary()
{
      int M[baseSize][baseSize] =
           /* A R N D C Q E G H I L K M F P S T W Y V X*/
  { /* A */ { 1                                                 },
    /* R */ { 0,1                                               },
    /* N */ { 0,0,1                                             },
    /* D */ { 0,0,0,1                                           },
    /* C */ { 0,0,0,0,1                                         },
    /* Q */ { 0,0,0,0,0,1                                       },
    /* E */ { 0,0,0,0,0,0,1                                     },
    /* G */ { 0,0,0,0,0,0,0,1                                   },
    /* H */ { 0,0,0,0,0,0,0,0,1                                 },
    /* I */ { 0,0,0,0,0,0,0,0,0,1                               },
    /* L */ { 0,0,0,0,0,0,0,0,0,0,1                             },
    /* K */ { 0,0,0,0,0,0,0,0,0,0,0,1                           },
    /* M */ { 0,0,0,0,0,0,0,0,0,0,0,0,1                         },
    /* F */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,1                       },
    /* P */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,1                     },
    /* S */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1                   },
    /* T */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1                 },
    /* W */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1               },
    /* Y */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1             },
    /* V */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1           },
    /* X */ { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1}
           /* A R N D C Q E G H I L K M F P S T W Y V X */
};
      buildScore(M);
}
```

```
// File 4: PairAlign.h


#ifndef __PAIRALIGN_H
#define __PAIRALIGN_H

#include "Common.h"

// The class of Matrices

class Matrices {
  public:
      void buildScore(int[][baseSize]);

    // a, b should both be upper case letters.
      int getScore(int a, int b) const { return score[a-'A'][b-'A';}

  protected:
      int score[26][26];
};


class PAM250 : public Matrices {
  public:
      PAM250();
};


class Blosum62 : public Matrices {
  public:
      Blosum62();
};


class Unitary : public Matrices {
  public:
      Unitary();
};


// Traceback Pointer

class tbPtr {
  public:
        tbPtr(int c1, int c2, int c3): k(c1), i(c2), j(c3) { }
        int k;
        int i;
        int j;                          // absolute coordinates
};


// Pairwise Alignment with affine gap costs

class pairAlign {
  public:
      pairAlign(Matrices *, int, int, const char *, const char *, bool);
      ~pairAlign(){ }
```

```
        // Return an array containing alignment of maximal score
        char **getAlign( ) { return A; }

        // Return the score of the best alignment
        int getScore() { return score; }

        int max (int x1, int x2){ return (x1 > x2 ? x1 : x2); }
        int max(int x1, int x2, int x3) { return max(x1, max(x2, x3)); }
        int max(int x1, int x2, int x3, int x4)
            { return max(max(x1, x2), max(x3, x4)); }

  private:
        void fastAlign(Matrices *,int,int, const char *, const char *);
        void freeTB();

        unsigned int m, n;          // lengths of input sequences s1 and s2
        int **V;                    // dynamic programming table
        tbPtr ***TB[3];             // the 3 m*n traceback pointer table
        char *A[2];                 // the final alignment
        int score;                  // contains V[m][n]
};


#endif
```

```
//File 5: PairAlign.cpp


#include <string>
using namespace std;
#include <stdlib.h>
#include <assert.h>
#include "PairAlign.h"


// takes a given scoring matrix, the gap open penalty d, gap extension
// penalty e, and the input sequences s1 and s2, calculate the dynamic
// programming table V and the traceback table TB.

pairAlign::pairAlign
(Matrices *sub, int d, int e, const char *s1, const char *s2, bool
traceback)
{
        if (!traceback) {
            fastAlign(sub, d, e, s1, s2);
            return;
        }

        m = strlen(s1);
      n = strlen(s2);
        int **E = allocTable(m, n);
      int **F = allocTable(m, n);
      V = allocTable(m, n);
        unsigned int i, j;
      for ( i = 0; i < 3; i++) {
            tbPtr **Tmp = (tbPtr **)calloc((m+1)*(n+1), sizeof(tbPtr
*));
            TB[i] = (tbPtr ***)calloc(m+1, sizeof(tbPtr **));
            for ( j = 0; j <= m; j++)
                  TB[i][j] = Tmp + j*(n+1);
      }

      // use Gotoh' algorithm, record to the traceback table as well

      E[0][0] = F[0][0] = V[0][0] = 0;
      for (i=1; i<=m; i++) {
            E[i][0] = F[i][0] = V[i][0] = -d - e * (i-1);
            TB[0][i][0] = new tbPtr(0, i-1, 0);
            TB[1][i][0] = new tbPtr(1, i-1, 0);
        }
        for (j=1; j<=n; j++) {
            E[0][j] = F[0][j] = V[0][j] = -d - e * (j-1);
            TB[0][0][j] = new tbPtr(0, 0, j-1);
            TB[2][0][j] = new tbPtr(2, 0, j-1);
        }

        int val;
        for (i=1; i<=m; i++)
            for (j=1; j<=n; j++) {
                  int s = sub->getScore(s1[i-1], s2[j-1]);

                  val = E[i][j] = max(V[i][j-1]-d, E[i][j-1]-e);
```

```
                    if (val != E[i][j-1]-e)
                          TB[2][i][j] = new tbPtr(0, i, j-1);
                    else
                          TB[2][i][j] = new tbPtr(2, i, j-1);

                    val = F[i][j] = max(V[i-1][j]-d, F[i-1][j]-e);
                    if (val != F[i-1][j]-e)
                          TB[1][i][j] = new tbPtr(0, i-1, j);
                    else
                          TB[1][i][j] = new tbPtr(1, i-1, j);

                    val = V[i][j] = max(V[i-1][j-1]+s, E[i][j], F[i][j]);

                    if (val == E[i][j])
                          TB[0][i][j] = new tbPtr(2, i, j);
                    else if ( val == F[i][j])
                          TB[0][i][j] = new tbPtr(1, i, j);
                    else
                          TB[0][i][j] = new tbPtr(0, i-1, j-1);
              }
        score = V[m][n];

      // array A will contain the alignment
        A[0] = allocString(m+n);
        A[1] = allocString(m+n);

      tbPtr *tb;
      i = m; j = n;
        unsigned int k = 0, x = 0;
      while ( i != 0 || j != 0 ) {
            tb = TB[k][i][j];
            if (i != tb->i || j != tb->j) {
                  A[0][x] = (i == tb->i) ? '-' : s1[i-1];
                  A[1][x] = (j == tb->j) ? '-' : s2[j-1];
                  x++;
            }
          k = tb->k;
            i = tb->i;
          j = tb->j;
        }

        reverseString(A[0]);
        reverseString(A[1]);

        freeTable(E, m, n);
        freeTable(F, m, n);
        freeTable(V, m, n);
        freeTB();
}


// free the memory allocated for matrix TB

void pairAlign::freeTB()
{
        for(int i = 0; i < 3; i++)
            for (int j = 0; j <= m; j++)
```

```
                for(int k = 0; k <= n; k++)
                    delete TB[i][j][k];
        for(int l = 0; l < 3; l++)
          free(TB[l][0]);
}


// align without traceback, compute the similarity score only

void pairAlign::fastAlign
(Matrices *sub, int d, int e, const char *s1, const char *s2)
{
        m = strlen(s1);
      n = strlen(s2);
        int **E = allocTable(m, n);
      int **F = allocTable(m, n);
      V = allocTable(m, n);

      // use Gotoh' algorithm, record to the traceback table as well

        unsigned int i, j;
      E[0][0] = F[0][0] = V[0][0] = 0;
      for (i=1; i<=m; i++) {
            E[i][0] = F[i][0] = V[i][0] = -d - e * (i-1);
        }
        for (j=1; j<=n; j++) {
            E[0][j] = F[0][j] = V[0][j] = -d - e * (j-1);
        }

        for (i=1; i<=m; i++)
            for (j=1; j<=n; j++) {
                  int s = sub->getScore(s1[i-1], s2[j-1]);
                  E[i][j] = max(V[i][j-1]-d, E[i][j-1]-e);
                  F[i][j] = max(V[i-1][j]-d, F[i-1][j]-e);
                  V[i][j] = max(V[i-1][j-1]+s, E[i][j], F[i][j]);
              }
        score = V[m][n];

        freeTable(E, m, n);
        freeTable(F, m, n);
        freeTable(V, m, n);
}
```

```
// File 6: GuideTree.h

#ifndef __GUIDETREE_H
#define __GUIDETREE_H

#include "Common.h"

struct treeNode {
      int Number;                        // Node Number
      vector<seqNode *> *alignment;  // the alignment it contains
      string *consensus;

      treeNode(int n, vector<seqNode *> *a, string &con)
            : Number(n), alignment(a)
            { consensus = new string(con); }
      treeNode(int n, seqNode * seq) : Number(n)
            { alignment = new vector<seqNode *>;
              alignment->push_back(seq);
              consensus = new string(*(seq->seq));
            }
      void prntNode(ostream & fout = cout);
};


class guideTree {
public:
      void initTree(vector<seqNode *> *);
      void addNode(treeNode *);
      treeNode *deleteNode(int i);
      treeNode *finalAlign();

private:
      vector<treeNode *> *treeList;
};


#endif
```

```
// File 7: GuideTree.cpp


#include <iostream>
#include <vector>
#include <string>
using namespace std;
#include <assert.h>
#include "GuideTree.h"
#include "Common.h"

// initialize tree T to be the set of all sequences

void guideTree::initTree(vector<seqNode *> *inputSeq)
{
      int num = inputSeq->size();
      treeList = new vector<treeNode *>;

      for (int i = 0; i < num; i++) {
            treeNode *node = new treeNode(i, (*inputSeq)[i]);
            treeList->push_back(node);
      }
}

// add a new node into the tree

void guideTree::addNode(treeNode * node)
{
      treeList->push_back(node);
}


// delete node #i and return the node, return null if #i not found

treeNode *guideTree::deleteNode(int i)
{
      treeNode * iNode;
      vector<treeNode *>::iterator iter= treeList->begin();
      for (; iter != treeList->end(); iter++) {
            if ( (*iter)->Number == i) {
                  iNode = *iter;
                  treeList->erase(iter);
                  return iNode;
            }
      }
      return NULL;
}


// return the final resulting alignment

treeNode *guideTree::finalAlign()
{
      assert(treeList->size() == 1);
      return treeList->front();
}
```

```
// print a node: number, sequence/alignment, consensus string

void treeNode::prntNode(ostream &fout)
{
      fout << "Node Number: " << Number << endl
            << "alignment: " << endl;
      int n = alignment->size();
      for (int i = 0; i < n; i++)
            fout << *((*alignment)[i]->seq) << endl;
      fout << "consensus string: "<<endl<<*consensus<<endl<<endl;
}
```

```
// File 8: ScoreMat.h

#ifndef __SCOREMAT_H
#define __SCOREMAT_H

class scoreMat {
public:
      scoreMat( Matrices * s, int d, int e) :
      sub(s), alpha(d), beta(e), maxScore(INT_MIN) { }
      bool initMat(vector<seqNode *> *);
      void updateMat(int i, int j, int z, string seq);
      void printTbl(ostream &fout = cout);
      int getMaxSeq1() { return maxSeq1; }
      int getMaxSeq2() { return maxSeq2; }

private:
      int **Matrix;           // similarity score matrix: n*n table
      string **lookupTbl;     // look up table mapping seq# and seqs
      int N;                  // size of Matrix/vector/lookupTbl
      int maxScore;           // maximum similarity score in Matrix
      int maxSeq1;            // sequence#1 which has the maxScore
      int maxSeq2;            // sequence#2 which has the maxScore
      Matrices *sub;          // mat
      int alpha;              // d
      int beta;               // e
};

#endif
```

```
// File 9: ScoreMat.cpp

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;
#include <stdlib.h>
#include <limits.h>
#include "PairAlign.h"
#include "ScoreMat.h"
#include "Common.h"


// Initial Matrix to be an N*N table of similarity scores
// between all pairs of N sequences
// lookupTbl is a reference table for Matrix

bool scoreMat::initMat(vector<seqNode *> *Seqs)
{
      N = Seqs->size();
      if (!N)
            return false;

      Matrix = allocTable(N-1, N-1);
      lookupTbl = (string **)malloc(sizeof(string *)*N);
      int i;
      for (i = 0; i < N; i++)
            lookupTbl[i] = new string(*((*Seqs)[i]->seq));

      for (i = 0; i < N; i++) {
            for ( int j = i+1; j < N; j++) {
            pairAlign *pair = new pairAlign
            (sub, alpha, beta, (*lookupTbl[i]).c_str(),
                (*lookupTbl[j]).c_str(), false);
                  int score = pair->getScore();
                  Matrix[i][j] = Matrix[j][i] = score;
                  if ( score >= maxScore ) {
                        maxScore = score;
                        maxSeq1 = i;
                        maxSeq2 = j;
                  }
            }
            Matrix[i][i] = INT_MIN;
      }

      return true;
}


// print the similarity matrix

void scoreMat::printTbl(ostream &fout)
{
      fout << endl;
      for ( int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
```

```
                if(Matrix[i][j] == INT_MIN) fout << setw(4) << "-";
                else fout << setw(4) << Matrix[i][j];
            }
            fout << endl;
        }
}


// delete child1 and child2 from Matrix and add parent to the matrix.
// seq is the consensus string of the parent

void scoreMat::updateMat
(int child1, int child2, int parent, string seq)
{
        lookupTbl[child1] = lookupTbl[child2] = NULL;
        lookupTbl[parent] = new string(seq);

        int i;

        for (i = 0; i < N; i++)
            Matrix[child1][i] = Matrix[child2][i]
                = Matrix[i][child1] = Matrix[i][child2] = INT_MIN;

        for (i = 0; i < N; i++) {
            if ( i != parent && lookupTbl[i] ) {
                pairAlign *pair = new pairAlign(sub, alpha, beta,
                    (*lookupTbl[i]).c_str(), seq.c_str(), flase);
                int score = pair->getScore();
                Matrix[i][parent] = Matrix[parent][i] = score;
            }
        }

        maxScore = INT_MIN;
        for (i = 0; i < N; i++)
            for(int j = i + 1; j < N; j++) {
                int score = Matrix[i][j];
                if ( score >= maxScore ) {
                    maxScore = score;
                    maxSeq1 = i;
                    maxSeq2 = j;
                }
            }
}
```

```
// File 10: MultipleAlign.h

#ifndef __MSA_H
#define __MSA_H

#include "GuideTree.h"

class multiAlign
{
public:
      multiAlign(istream & in, ostream & out, ostream & Log,
            Matrices * m, int d, int e)
            : mat(m), alpha(d), beta(e) { buildMSA(in, out, Log);}
      void buildMSA(istream & in, ostream & out, ostream & Log);

private:
      void getInput(istream & in, ostream & Log);
      void stripSeq(string & s);
      void prntResult(vector<seqNode *>*, ostream &, ostream &);
      vector<seqNode *> *getAlign(treeNode *, treeNode *, ostream &);
      void insertGaps(string &, const char*, vector<seqNode *>*);
      string &consensus(vector<seqNode *> *align);

      Matrices *mat;
      int alpha;
      int beta;
      vector<seqNode *> *inputSeqs;
};


#endif
```

```
// File 11: MultipleAlign.cpp

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;
#include <limits.h>
#include <stdlib.h>
#include <assert.h>
#include "PairAlign.h"
#include "ScoreMat.h"
#include "MultipleAlign.h"
#include "GuideTree.h"
#include "Common.h"


// the whole process of calculating the multiple sequence alignment

void multiAlign::buildMSA(istream & in, ostream & out, ostream & Log)
{
      getInput(in, Log);

      // step 1. build scoring matrix (n*n table)
      scoreMat *nnTbl = new scoreMat(mat, alpha, beta);
      assert(nnTbl);
      nnTbl->initMat(inputSeqs);

      //step 2. construct the tree and do alignment.
      // Initialize the tree
      guideTree *tree = new guideTree();
      tree->initTree(inputSeqs);

      // n-1 iteration
      int n = inputSeqs->size();
      for ( int i = 0; i < n - 1; i++) {

            //2.1 get the pair has maximum score
            nnTbl->printTbl(Log);
            int i = nnTbl->getMaxSeq1();
            int j = nnTbl->getMaxSeq2();
            Log << endl << "max score: ("
                  << i << ", " << j << ")" << endl << endl;

            //2.2 remove node i and j from the tree
            treeNode *iNode = tree->deleteNode(i);
            treeNode *jNode = tree->deleteNode(j);
            iNode->prntNode(Log);
            jNode->prntNode(Log);

            //2.3 construct a new node
            vector<seqNode *> *align = getAlign(iNode, jNode, Log);
            string conStr = consensus(align);
            treeNode *A = new treeNode(i, align, conStr);
            Log << endl << "new node :  ";
            A->prntNode(Log);
```

```
            //2.4 update scoring table
            nnTbl->updateMat(i, j, i, conStr);

            //2.5 add the new node to the tree
            tree->addNode(A);
      }

      // step 3. print the result
      prntResult(tree->finalAlign()->alignment, out, Log);
}


//read input with FASTA format.

void multiAlign::getInput(istream & in, ostream & Log)
{
      const int MaxLength = 255;
      char Line[MaxLength+1] = " ";
      inputSeqs = new vector<seqNode *>;

      while (in.getline(Line, MaxLength) && Line[0] != '>');
      if (Line[0] != '>') // empty or invalid format
            return;

      bool WasSeqLine = false;
      int seqNumber = 0;
      string seq = "";
      Log << "Input sequences: "<< endl;

      while ( in.getline(Line, MaxLength) ) {
            if (Line[0] == '>') {
                  if (WasSeqLine) {
                        stripSeq(seq);
                        if (!seq.empty()) {
                              seqNode *Node =
                                          new seqNode(seqNumber, seq);
                              inputSeqs->push_back(Node);
                              Log << seqNumber <<": " << seq << endl;
                              seqNumber++;
                        }
                  }
                  WasSeqLine = false;
                  seq = "";
            }

            else {
                  seq += Line;          // not a new sequence
                  WasSeqLine = true;
            }
      }//while getLine

      if (Line[0]!='>') {
            seq += Line;
            stripSeq(seq);
            if (!seq.empty()) {
                  seqNode *Node = new seqNode(seqNumber, seq);
                  inputSeqs->push_back(Node);
```

```
                    Log << seqNumber <<": " << seq << endl;
                    seqNumber++;
                }
        }
}


// print the final alignment to the output file and the log file

void multiAlign::prntResult(vector<seqNode *> *result, ostream & out,
ostream & cout)
{
        vector<seqNode *> *orderResult = new vector<seqNode *>;
        int i, n=result->size();
        for ( i = 0; i < n; i++ ) {
                int number = (*result)[i]->Num;
                vector<seqNode *>::iterator iter = orderResult->begin();
                for(; iter != orderResult->end(); iter++)
                        if( number < (*iter)->Num)      break;
                orderResult->insert(iter, (*result)[i]);
        }

        int length = strlen((*orderResult)[0]->seq->c_str());
        int start = 0, len;
        const int oneline = 60;
        while (length > 0) {
                len = length > oneline ? oneline : length;
                out << endl << "       ";
                cout << endl << "       ";
                for ( i = 1; i <= (len+1)/10; i++) {
                        cout << setw(10) << start + i*10;
                        out << setw(10) << start + i*10;
                }
                out << endl;
                cout << endl;
                for ( i = 0; i < n; i++) {
                        string current;
                        current.assign(*((*orderResult)[i]->seq), start, len);
                        out << setw(4) << i <<"  "<< current << endl;
                        cout << setw(4) << i <<"  "<< current << endl;
                }
                start += len;
                length -= oneline;
        }
}


// change string s to all capital, discard invalid letters

void multiAlign::stripSeq(string & s)
{
    bool valid[26];
    int i;
    for (i=0; i<26; i++)
            valid[i] = false;
    for (i=0; i<baseSize; i++)
            valid[residuesOrder[i]-'A'] = true;
```

```
        int m = s.size();
        int j = 0;
        for ( i = 0; i < m; i++) {
                char c = s[i];
                if (islower(c))
                        c = toupper(c);
                if (isupper(c) && valid[c-'A'])
                        s[j++] = c;
        }
        for(; j < m; j++)
        s[j] = '\0';
}


// compute the alignment of two nodes A and B using standard
// pairwise alignment

vector<seqNode *> *multiAlign::getAlign(treeNode *A, treeNode *B,
ostream & Log)
{
        string *Ac = new string(*(A->consensus));
        string *Bc = new string(*(B->consensus));

        pairAlign *pair = new pairAlign
                        (mat, alpha, beta, Ac->c_str(), Bc->c_str());
        char **result = pair->getAlign();
        Log << "pairalign result:" <<endl<<
                result[0] << endl << result[1] << endl << endl;

        insertGaps(*Ac, result[0], A->alignment);
        insertGaps(*Bc, result[1], B->alignment);

        A->alignment->insert(A->alignment->begin(),
                B->alignment->begin(), B->alignment->end());
        return A->alignment;
}


// insert new gaps in after which are not in string before into the
// same position of all sequences in vector seqs

void multiAlign::insertGaps
        (string &before, const char *after, vector<seqNode *> *seqs)
{
        int n = seqs->size();
        for (int i = 0; i < strlen(after); i++) {
                assert(i <= strlen(before.c_str()));
                if (before[i] != after[i]) {
                        assert(after[i] == '-' );
                        before.insert(i, "-");
                        for ( int j = 0; j < n; j++)
                                (*seqs)[j]->seq->insert(i, "-");
                }
        }
}
```

```
// calculate the consensus string of an alignment align

string & multiAlign::consensus(vector<seqNode *> *align)
{
      int numSeqs = align->size();
      int seqLength = strlen((*align)[0]->seq->c_str());
      char *conSeq = new char[seqLength+1];

      int charSum[26];
      int column;

      for ( column = 0; column < seqLength; column++) {

            int k;
            for ( k = 0; k < 26; k++)
                  charSum[k] = INT_MIN;
            for (k = 0; k < numSeqs; k++) {
                  char ch = (*align)[k]->seq->at(column);
                  if (ch == '-') ch = 'X';
                  charSum[ch-'A'] = 0;
            }

            for (int row1 = 0; row1 < numSeqs; row1++)
                  for (int row2 = row1+1; row2 < numSeqs; row2++) {
                        char a = (*align)[row1]->seq->at(column);
                        if ( a == '-' ) a = 'X';
                        char b = (*align)[row2]->seq->at(column);
                        if (b == '-' ) b = 'X';
                        int score = mat->getScore(a, b);
                        charSum[a-'A'] += score;
                        charSum[b-'A'] += score;
                  }

            int maxSum = INT_MIN;
            char conChar = '*';
            for (int c = 0; c < 26; c++) {
                  if (charSum[c] > maxSum) {
                        conChar = c + 'A';
                        maxSum = charSum[c];
                  }
            }
            if (maxSum < 0)
                  conChar = 'X';

            conSeq[column] = conChar;
      }
      conSeq[column] = '\0';
      string *conStr = new string(conSeq);
      return *conStr;  // its parent
}
```

```
// File 12: Common.h

#ifndef __COMMON_H
#define __COMMON_H

#include <string>
using namespace std;

const int baseSize = 21;    // 20 residues + 'X'
const char residuesOrder[26] = "ARNDCQEGHILKMFPSTWYVX";

struct seqNode {
      int Num;
      string *seq;

      seqNode(int n, string s) : Num(n)
      { seq = new string(s); }
};


char *allocString(int size);
char *reverseString(char *s);
int **allocTable(int, int);
int freeTable(int**, int, int);


#endif
```

```
// File 13: Common.cpp

#include <string>
using namespace std;
#include <stdlib.h>
#include <assert.h>
#include "Common.h"

// allocate and clear memory for a string of given size

char *allocString(int size)
{
      char *sptr = (char *)malloc(sizeof(char)*(size+1));
      for (int i = 0; i <= size; i++)
            sptr[i] = '\0';
      return sptr;
}


// return the reverse string of s

char *reverseString(char *s)
{
      int size = strlen(s);
      char c;
      for (int i = 0, j = size - 1; i <= size/2 - 1; i++, j--) {
            c = s[i];
            s[i] = s[j];
            s[j] = c;
      }
      return s;
}


// allocate a (m+1)*(n+1) table; row 0 and column 0 will not be used

int **allocTable(int m, int n)
{
      int *Tmp = (int *)malloc(sizeof(int)*(m+1)*(n+1));
      assert(Tmp);
      int **Table = (int **)malloc(sizeof(int*)*(m+1));
      assert(Table);
      for ( int i = 0; i <= m; i++)
            Table[i] = Tmp + i*(n+1);
      return Table;
}


// free the memory allocated for Table

int freeTable(int **Table, int m, int n)
{
    free(Table[0]);
    return EXIT_SUCCESS;
}
```

VITA

Ke Liu

Candidate for the Degree of

Master of Science

Thesis: MULTIPLE SEQUENCE ALIGNMENT WITH EVOLUTIONARY TREES

Major Field: Computer Science

Biographical:

Personal Data: Born in Hunan, China, January 28, 1973, daughter of Changsheng Liu and Xiaokun Yin.

Education: Received Bachelor of Science in Finance from Central University of Economics, Beijing, China in July 1994. Completed the requirements for the Master of Science degree in Computer Science at the Computer Science department at Oklahoma State University in December 2001.