AN ANCHOR-BASED MODEL FOR GLOBAL MULTIPLE ALIGNMENT OF WHOLE GENOME SEQUENCES

YUE MA

An Anchor-based Model for Global Multiple Alignment of Whole Genome Sequences

by

© Yue Ma

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

March 2005

St. John's Newfoundland

Abstract

With the benefit of advanced biotechnology, large numbers of whole genome sequences have been compiled. Aligning whole genome sequences is a fundamentally different problem than aligning short sequences. Recently, intensive research activities have been devoted to this problem. We propose an anchor-based model for global multiple alignment of whole genome sequences. The model includes three main phases. Firstly, an enhanced suffix array method is employed to find anchors. Next, an exact chaining algorithm, which is based on the dynamic programming technique and the longest common subsequence idea, calculates an anchor-chain for the weighted anchors. Lastly, a progressive multiple alignment method is used to close the gaps between the anchors. The proposed chaining procedure is based on evolutionary theory and can align whole genome sequences not only for close homologs, but also distant species. Combined with the exact suffix array approach, this model can compute partially accurate solutions and generate a high-quality alignment result in terms of computation and biology.

Acknowledgments

I would like to give many thanks to my supervisor, Dr. Caoan Wang, for his guidance and financial support of my study. I appreciate his suggestion, patience and kindness. His encouragement inspired me during the whole research period.

I would like to thank all the members of our computer science department. Thanks to Dr. Jianbo Qian for many friendly conversations. Thanks also to Dr. Wolfgang Banzhaf, Ms. Elaine Boone and Dr. Todd Wareham for their continuing kindness and help.

I would like to thank many of my colleagues and friends. Thanks to biologist GuangXu Liu and his colleagues from the Evolutionary Genetics Laboratory for many helpful consultations. Thanks to Ms. Sarah Morrissey from biology department for productive dialogues. Thanks to all my friends who shared many happy moments with me.

Finally, I would like to express my gratitude to my family. Their unconditional love accompanied me through the toughest periods of this program.

Contents

Abstract ii

Acknowledgments iii

Contents iv

List of Figures vii

1 Introduction 1

1.1 Biological Background....1
1.1.1 DNA and Protein....1
1.1.2 Gene and Genome....2
1.1.3 Evolutionary Theory....3
1.2 Bioinformatics....4
1.2.1 What is bioinformatics?....4
1.2.2 Why bioinformatics?....4
1.2.3 What is the goal of bioinformatics?....5
1.3 Basic problem: Sequence Alignment....5
1.3.1 Standard Sequence Alignment....5
1.3.2 Genome Sequence Alignment....6
1.4 Our Contributions....8

2 Related Theories and Techniques 10

2.1 Complexity Issue: P, NP and NP-Completeness....10
2.2 Dynamic Programming....11
2.3 Longest Common Subsequence....12
2.3.1 Solving the Longest Common Subsequence Problem for Two Sequences....13
2.3.2 Solving the Longest Common Subsequence Problem for Multiple Sequences....18
2.4 Suffix Array....19

2.5 Progressive Global Multiple Sequence Alignment.....20

3 A Literature Review for Recent Progresses in Anchor-based Genome Sequence Alignment 22

MUMmer.....22
PipMaker and MultiPipMaker.....24
GLASS.....25
WABA.....26
LSH-ALL-PAIRS.....27
CHAOS + DLALIGN.....28
MGA.....31
EMAGEN.....34
MAUVE.....36
LAGAN and Multi-LAGAN.....38
AVID and MAVID.....41

4 Our Chaining Algorithm 45

4.1 Our Ideas and Their Origins.....45
4.2 Computational Complexity.....47
4.2.1 Definition of the Problem.....47
4.2.2 The Multiple Heaviest Common Subsequence Problem is NP-Complete.....48
4.2.2.1 The Restriction Technique of Proving NP-Completeness.....48
4.2.2.2 The Complexity of the Longest Common Subsequence Problem.....49
4.2.2.3 Prove the NP-Completeness for *MHCS* problem.....51
4.3 Algorithm Description.....52
4.3.1 The Algorithm for 3 Sequences and Its Complexity Analysis.....53
4.3.2 The Algorithm for k Sequences and Its Complexity Analysis.....56
4.4 Implement and Results.....58

5 The Whole Procedure of Our Model 60

5.1 Our Ideas and their Origins.....60
5.2 Phase 1: Find Multi-MUMs as Anchors.....62
5.3 Phase 2: Find the Multiple Heaviest Common Subsequence

as Anchor-chain to Align Anchors.....68
5.4 Phase 3: Close Gaps and Get Detailed Alignment.....71
5.5 Time Complexity Analysis.....71

6 Conclusions and Future Work **73**

6.1 Conclusions.....73
6.2 Future Work.....74

Bibliography **76**

Appendix A CLUSTAL W: a tool for progressive global multiple alignment **84**

Appendix B Source Code **86**

List of Figures

1 The *b* and *l* tables computed by LCS-Length (X, Y).....16

2 The template sequence T and the sequence S_i.....51

3 The enhanced suffix array for four sequences S_1, S_2, S_3, S_4.....66

4 The *multi-MUM index sequences* of input sequences.....68

5 The alignment of the anchor-computing *multi-MUM index sequences*.....69

6 The alignment of all the *multi-MUM index sequence*.....70

7 The alignment of all the anchors.....70

8 The alignment result.....71

Chapter 1

Introduction

1.1 Biological Background

1.1.1 DNA and Protein

DNA (Deoxyribonucleic Acid) is a very large chemical molecule made up of linear, unbranched chains of subunits called nucleotides. According to the chemical structure, there are four types of the bases: Adenine (A), Cytosine (C), Thymine (T) and Guanine (G) [28]. Nucleotides are linked together by chemical bonding to form the long DNA polymer. In living cells, DNA is double-stranded, forming a double helix structure. The two strands in the double helix are complementary to each other through the pairing of the bases, where A pairs with T and C pairs with G [41].

Proteins are nitrogenous organic compounds that are essential constituents of living cells. Proteins, which are formed by the polymerization of amino acids, are coded by the segments of DNA. All of the proteins in living things are made of only 20 kinds of amino

acids [53].

1.1.2 Gene and Genome

A gene can be defined as a segment of DNA on a chromosome. Each gene carries some information for making certain proteins, which can determine the physical appearance of an organism, certain behavioral characteristics, how well it combats specific diseases, and other characteristics. It is a unit of heredity [41].

A genome used to be defined as the entire complement of the genetic material in a chromosome set. It is the entire genetic complement of a prokaryote, virus, mitochondrion, chloroplast or the haploid nuclear genetic complement of a eukaryotic species [47]. Now an increasing number of biologists simply define the genome as the sum of all DNA in an organism, including genes. The particular order of the four chemical bases A, T, C, G, as they repeat millions and even billions of times, is what makes species different. The genome of each organism is unique.

Genomes are complex but interesting materials. The genome of each person is unique. For humans, differences in just 0.1% of the genome will cause the different hair colors, builds, etc. The human genome shares 98.4% identity to that of chimpanzees [29].

1.1.3 Evolutionary Theory

Living things are fundamentally similar in their basic anatomical structures and chemical compositions. They all begin as single cells that reproduce themselves by similar division processes. All plants and animals receive their specific characteristics from their parents by inheriting particular combinations of genes. Despite the great diversity of life, the simple language of DNA is the same for all living things. The anatomical and chemical similarities between the living things imply that they either share a common ancestry or came into existence as a result of similar natural processes [39]. This is where the idea of evolution comes from.

After ancient Greek philosophers such as Anaximander supposed that the development of life is from non-life, Charles Darwin presented his theory of "natural selection" in which species accumulate minor advantageous genetic mutations. Suppose a member of a species developed a functional advantage, its offspring would inherit that advantage and pass it on to their offspring. The inferior (disadvantaged) members of the same species would gradually die out, leaving only the superior (advantaged) members of the species. Natural selection is the preservation of a functional advantage that enables a species to compete better in the wild. It eliminates inferior species traits gradually over time [39]. Even though some refutations of Darwin's theory have been presented, the basic idea commendably explains many evolutionary phenomena and holds a significant

position in molecular biology, biochemistry and genetics.

1.2 Bioinformatics

1.2.1 What is bioinformatics?

As more and more computational problems are arising from biology, bioinformatics (or computational molecular biology) is an emerging field combining Computer Science and Molecular Biology. Bioinformatics uses computational technology to deal with biological problems.

1.2.2 Why bioinformatics?

In the Human Genome Project (HGP) that was completed in 2003, one of the key research areas was bioinformatics. Without bioinformatics, people would have no idea how to analyze and draw meaning from the large amounts of data and information gleaned from the HGP. Since the human genome consists of approximately three billion base pairs [42], and it is difficult to imagine carrying on without computational support.

Advanced biotechnology brings us more and more important biological data, and computer science is all about automated problem solving. With its ability to analyze a problem, identify a suitable formula and design an efficient algorithm, computer science is continually called upon to solve the complex problems of biology.

1.2.3 What is the goal of bioinformatics?

The final goal of this interdisciplinary field is to design algorithmic solutions, which work efficiently on computers and are biologically correct. However, some of those solutions are quite far from this goal. Most efficient algorithmic solutions are not precisely biologically correct, while correct solutions do not always work very efficiently. Thus far, people are still looking for a trade-off.

1.3 Basic Problem: Sequence Alignment

1.3.1 Standard Sequence Alignment

Sequence alignment is the procedure of comparing sequences. The procedure involves searching for individual characters or character patterns that are in the same order in the sequences [41]. Identical or similar characters are aligned in the same column. At the same time, in a mismatch, nonidentical or different characters can be put in the same column. Also, a gap can be inserted in the sequences. Nonidentical characters and gaps are placed in order to bring as many identical or similar characters as possible into each column.

Sequence alignment can be used to discover functional, structure and evolutionary information between biological sequences [27]. If the sequences are relatively similar, even in some parts, they may have a similar biochemical structure and function. Similar

sequences from different organisms may belong to a common ancestor sequence; these sequences are then defined as being homologous [41].

There are two types of sequence alignment: global alignment and local alignment [53]. Global alignment attempts to align entire sequences by trying to align as many characters as possible until the ends. Global alignment is suitable for aligning similar sequences about the same length. Local alignment focuses on aligning the blocks that have the highest density of matches in the sequences, which leads to some subalignments between the sequences. Local alignment is suitable for aligning sequences that have some similar parts but which are dissimilar in others. Those sequences can have different lengths but have some conserved regions [28]. If only two sequences are aligned, it is called pairwise sequence alignment; otherwise, it is multiple sequence alignment [53].

1.3.2 Genome Sequence Alignment

People are always concerned about evolutionary changes in organisms. With the benefits of advanced biotechnology, more and more available whole genome sequences have been detected. People are no longer satisfied with aligning only short DNA and protein sequences; they now want to use different techniques to align genome sequences.

Whole genome alignment can be used for many purposes. It can be used to detect the conserved gene blocks, to find orthologous regions between sequences, to compare

evolutionary strains, and to analyze syntenic chromosomal regions [17].

Sequence alignment techniques have been developed considerably in recent decades. However, the standard sequence alignment methods cannot be used for whole genome alignment directly. Because the standard sequence alignment methods can only observe point mutation, insertion and deletion, the time and space complexity of existing algorithms are too high for large-scale sequences. The technique of whole genome alignment is slightly different from sequence alignment, but is based on it. The objective for whole genome alignment focuses on extracting the conserved gene blocks, which are the anchors for the alignment, and finding an optimal anchor chain for large transposition, insertion, and deletion in the genomes [13].

The inputs of whole genome alignment programs are usually assumed to be relatively conserved genome sequences. Many available whole genome alignment software systems have been developed recently [13]. Most of them can only align two genome sequences, which is defined as pairwise genome sequence alignment. However, many existing pairwise programs have been improved to deal with multiple genomes and several multiple genome alignment methods have been recently proposed. This technique has recently attracted more attention.

1.4 Our Contributions

To compare whole genome sequences, biologists increasingly need alignment methods that are both efficient enough to handle large numbers of long sequences, and accurate enough to correctly align the conserved biological features of distant species present in the sequences. So far, most programs work efficiently in aligning small numbers of closely related genome sequences. Very few genome alignment programs can align distant homologs and they usually cannot work efficiently for large numbers of genome sequences.

We present an anchor-based model for aligning multiple whole genome sequences. The model includes three main phases: finding anchors, finding an anchor-chain and closing gaps to get a detailed alignment. In the first phase, we employ an enhanced suffix array method to find anchors. In the second phase, we propose a chaining algorithm based on the dynamic programming technique and longest common subsequence idea to calculate an anchor-chain for the anchors, to which we append biologically meaningful weights. We refer to the problem of finding the anchor-chain as the problem of finding the *multiple heaviest common subsequence (MHCS)*. Then, we analyze the computational complexity of the *MHCS* problem and present methods to solve its conditional cases. In order to make up for the lack of methods for aligning distantly related genome sequences, we propose a novel strategy with biological reasons: the genome sequences from close

homologs are first selected for assembly, and then distantly related genome sequences are appended to the anchor alignment iteratively. In the last phase, we use the progressive multiple alignment method to close the gaps between the anchors.

Our chaining algorithm involving evolutionary theory finds a biologically more correct anchor-chain for the whole aligning process. Experiments show that this approach obtains meaningful results according to the appended weight. Our chaining procedure generates a more accurate and convincing anchor alignment in terms of computation and biology. It helps the model to assemble flexible genome sequences (i.e. more genome sequences at any evolutionary distance). Combined with the exact suffix array approach in the first phase, this model leads to a high-quality alignment result.

Chapter 2

Related Theories and Techniques

2.1 Complexity Issue: P, NP and NP-Completeness

The class P includes the problems which can be solved in time $O(n^k)$ for some constant k, where n is the size of the input. Simply speaking, by a deterministic Turing machine, these problems can be solved in polynomial time [25].

Given a "certificate" of a solution to one problem, if the certificate can be verified by a Turing machine in polynomial time in the size of the input of the problem, we say that the problem belongs to the class NP.

NP-Complete problems are the hardest problem in NP. Formally, a language L is defined to be NP-Complete if $L \in NP$ and for all other languages $L' \in NP$, L' can be transformed from L in polynomial time [16].

Any problem in P is in NP. Because if a problem is in P, it definitely can be solved in polynomial time without a certificate, that is, $P \subseteq NP$. However, whether or not P is a

proper subset of NP is still a famous open problem. If one NP-Complete problem has a polynomial algorithm solution, every problem in NP can be solved in polynomial time. Until now, no polynomial-time algorithm has ever been discovered for any NP-Complete problem, that is, any problem belonging to NP-Completeness class has not been solved in polynomial time [15].

2.2 Dynamic Programming

Dynamic programming (DP) is a commonly used method for solving multi-stage decision problems and it is typically applied to optimization problems. DP is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.

The development of a DP algorithm normally can be divided into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1-3 form the basis of a dynamic programming solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required. When step 4 is performed, additional information needs to be maintained sometime during the

computation in step 3 to ease the construction of an optimal solution [16].

DP is known to be an efficient algorithm technique for solving certain combinatorial problems. It is the basis of comparing biological sequences [53]. Examples include the Needleman-Wunsch and Smith-Waterman algorithms. Hence, DP is said to be the most fundamental technique in bioinformatics.

Exact DP algorithm not only gives an optimal solution for pairwise sequence alignment, but also provides an optimal global alignment of multiple sequences [36]. Because the number of computational steps and the amount of memory required grow exponentially, the number of sequences to be aligned is limited [26].

2.3 Longest Common Subsequence

The Longest Common Subsequence (LCS) problem has been studied for a long time and it deals with many problems in the Computational Biology field, especially for assembling biological sequences.

A subsequence of a given sequence is just the given sequence with zero or more elements left out [16]. Basically, given a sequence, a sequence is a subsequence of the given sequence if there exists a strictly increasing sequence of indices of the given sequence, and all the characters of both sequences that have those indices are the same. For example, $X = \langle A, C, T, A \rangle$ is a subsequence of $Y = \langle G, A, G, C, A, T, A \rangle$ with

corresponding index sequence $\langle 2, 4, 6, 7 \rangle$.

For two sequences X and Y, a common subsequence of X and Y is defined as a subsequence of both X and Y. For example, if $X = \langle A, T, C, G, T, A, A, C \rangle$ and $Y = \langle T, C, G, A, C \rangle$, then the sequence $\langle T, C, G \rangle$ is a common subsequence of both X and Y. But the sequence $\langle T, C, G \rangle$ is not a longest common subsequence of X and Y because there is another common subsequence $\langle T, C, G, A, C \rangle$ and its length is five, which is greater than the length of $\langle T, C, G \rangle$. Since there is no common subsequence with a length of six or greater, the common subsequence $\langle T, C, G, A, C \rangle$ is a longest common subsequence of X and Y.

2.3.1 Solving the Longest Common Subsequence Problem for Two Sequences

In the traditional Longest Common Subsequence problem, the input is two sequences and the output is a common subsequence with maximum length [16]. The brute-force approach to solve this problem is to check all the subsequences of one sequence and to see if each subsequence is a subsequence of the other sequence. The procedure ends when the longest subsequence is found, which is corresponding to a subset of the indices of the first sequence. If one sequence includes n characters, it has 2^n subsequences.

The problem has been extensively investigated [49] and many approximation

algorithms have already been proposed [4]. However, as an exact algorithm, dynamic programming technique can compute an accurate solution in reasonable time.

According to the book of Cormen [16], there are four basic steps. The first step is to characterize a longest common subsequence. An optimal substructure property has been proposed for the problem.

Given a sequence $X = \langle x_1, x_2, ..., x_m \rangle$, the ith prefix of X, for $i = 0,1,...,m$, is defined as $X_i = \langle x_1, x_2, ..., x_i \rangle$. For example, if $X = \langle A,T,C,G,C,A,T \rangle$, then $X_5 = \langle A,T,C,G,C \rangle$ and X_0 is the empty sequence.

The optimal substructure property of LCS is known as: Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be the two sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

The second step is a recursive solution. The recursive solution is to establish a recurrence for the value of an optimal solution. $l[i, j]$ is defined to be the length of an LCS of sequence X_i and Y_j.

The optimal substructure has already been discovered [16], which is the recursive formula:

$$l[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0, \\ l[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j, \\ \max(l[i,j-1], l[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j. \end{cases}$$

The third step is to compute the length of an LCS. A dynamic programming method can be used to compute the solution of the $\Theta(mn)$ distinct subproblems.

In the dynamic programming table, the $l[i,j]$ values are stored in the entries that are computed in row-major order. In order to simplify the construction of the optimal subproblem solution, there is a table $b[1...m,1...n]$, and $b[i,j]$ points to the table entry according to the choice of the optimal subproblem solution when computing $l[i,j]$. In the end, b and l tables are both returned and the length of an LCS of X and Y is in $l[m,n]$.

```
    LCS-Length (X, Y)
1   m ← length[X]
2   n ← length[Y]
3   for i ← 1 to m
4       do l[i,0] ← 0
5   for j ← 0 to n
6       do l[0,j] ← 0
7   for i ← 1 to m
8       do for j ← 1 to n
9              do if x_i = y_j
10                 then l[i,j] ← l[i-1,j-1]+1
```

```
11                    b[i, j] ← "^ "
12                else if l[i − 1, j] ≥ l[i, j − 1]
13                    then l[i, j] ← l[i − 1, j]
14                         b[i, j] ← "↑"
15                    else l[i, j] ← l[i, j − 1]
16                         b[i, j] ← "←"
17  return l and b    [16]
```

For example, if we have two sequences X = <A, T, C, G, T, A, A, C> and Y = <T, C, G, .A, C>, the b and l tables computed by LCS-Length (X, Y) is:

i	j	0	1	2	3	4	5
		y_j	T	C	G	A	C
0	x_i	0	0	0	0	0	0
1	A	0	0↑	0↑	0↑	1^	1←
2	T	0	1^	1←	1←	1←	1←
3	C	0	1↑	2^	2←	2←	2^
4	G	0	1↑	2↑	3^	3←	3←
5	T	0	1^	2↑	3↑	3↑	3↑
6	A	0	1↑	2↑	3↑	4^	4←
7	A	0	1↑	2↑	3↑	4^	4←
8	C	0	1↑	2^	3↑	4↑	5^

Figure 1: the b and l tables computed by LCS-Length (X, Y).

The entry square $[i, j]$ contains the value of $l[i, j]$ and the appropriate arrow for the value of $b[i, j]$. For $i, j > 0$, entry $l[i, j]$ depends on whether $x_i = y_j$ and the values in entries $l[i-1, j], l[i, j-1]$ and $l[i-1, j-1]$, which are computed before $l[i, j]$. The entry 5 in $l(8,5)$ is the length of the longest common subsquence $\langle T, C, G, A, C \rangle$.

The running time of each entry takes $\Theta(1)$ time to compute. Hence, the time complexity of this procedure is $\Theta(mn)$.

The fourth step is to construct an LCS. In this step, the b table is used to construct an LCS of $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$. Beginning at $b[m, n]$, we can trace through the table following the arrows. The symbol "^ "in entry $b[i, j]$ implies that $x_i = y_j$ is the element of the LCS. After that, the procedure reverses the order of the LCS, and prints it out. As the example above illustrates, we follow the $b[i, j]$ arrows from the lower right-hand corner, find each "^ " on the way for which $x_i = y_j$ is one of the members of the LCS.

This procedure can be described as:

PRINT – LCS (b, X, i, j)

1 **if** $i = 0$ **or** $j = 0$
2 **then return**
3 **if** $b[i, j] =$ "^ "
4 **then** PRINT-LCS $(b, X, i-1, j-1)$
5 print x_i
6 **else if** $b[i, j] =$ "↑"

7 **then** PRINT-LCS $(b, X, i-1, j)$

8 **else** PRINT-LCS $(b, X, i-1, j)$

Following this procedure, the LCS $\langle T, C, G, A, C \rangle$ will be printed. Because in each step of the recursion, at least one of i and j has to be determined, this step takes $\Theta(m+n)$ time.

2.3.2 Solving the Longest Common Subsequence Problem for Multiple Sequences

To solve the longest common subsequence problem for multiple sequences, the traditional case is extended. For the three sequences case, in the first step, the optimal substructure property is obtained. Let $A = \langle a_1, a_2, ..., a_m \rangle$, $B = \langle b_1, b_2, ..., b_n \rangle$, $C = \langle c_1, c_2, ..., c_p \rangle$ be the three sequences, and let $Z = \langle z_1, z_2, ..., z_r \rangle$ be any LCS of A, B and C.

1. If $a_m = b_n = c_p$, then $z_r = a_m = b_n = c_p$ and Z_{r-1} is an LCS of A_{m-1}, B_{n-1} and C_{p-1}.
2. If $a_m \neq b_n \neq c_p$, then $z_r \neq a_m$ implies that Z is an LCS of A_{m-1}, B and C.
3. If $a_m \neq b_n \neq c_p$, then $z_r \neq b_n$ implies that Z is an LCS of A, B_{n-1} and C.
4. If $a_m \neq b_n \neq c_p$, then $z_r \neq c_p$ implies that Z is an LCS of A, B and C_{p-1}

The second step is a recursive solution. Still, the recursive solution is to establish a recurrence for the value of an optimal solution. $l[i, j, k]$ is defined to be the length of an LCS of sequence.

The recursive formula is revised from the optimal substructure.

$$l[i,j,k] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \text{ or } k=0, \\ l[i-1,j-1,k-1]+1 & \text{if } i,j,k>0 \text{ and } a_i=b_j=c_k, \\ \max(l[i-1,j,k],l[i,j,k-1],l(i,j-1,k]) & \text{if } i,j,k>0 \text{ and } a_i \neq b_j \text{ or } b_j \neq c_k \end{cases}$$

The third and the fourth steps compute the length of and construct the LCS. According to the two sequences case and the recursive formula above, the algorithm can be straightforwardly extended.

2.4 Suffix Array

Suffix Array is a lexicographical sorted array of all the suffixes of a string [38]. It is a simpler and more compact alternative to the suffix tree method [41] in numerous applications. The main advantage of suffix array over suffix tree is that, in practice, it uses three to five times less space [38]. It is much more space-efficient and has competitive performance. Suffix array provides an efficient data structure to search for a query in a very long text, to find repeats in a string, and matches among multiple sequences. It works well for indexing and analyzing long genome sequences.

Many algorithms have been proposed to construct suffix arrays. Some of them first built a suffix tree then converted to suffix array. This idea takes linear time but the extra space requirement is very high. Fortunately, some direct linear time construction

algorithms [12] [32] are very simple and works desirably well in linear time and lower space requirement.

An enhanced suffix array, which combines with the information of longest common prefixes can require much less space than all the algorithms based on the bottom-up traversal of the suffix trees [1].

2.5 Progressive Global Multiple Sequence Alignment

The problem of finding the multiple sequence alignment with Sum-of-Pairs score was proved to be NP-complete [59]. No optimal algorithm exists for solving the multiple sequence alignment problem in polynomial time unless $P = NP$. The progressive global alignment method is the most commonly used heuristic today for aligning biological sequences. It is rapid, requires low memory space and offers good performance on relatively well-conserved, homologous sequences [28].

This method has three basic steps: first, compute the alignment scores (or distance) between all pairs of sequences; next, build a guide tree that reflects the similarities between sequences, using pairwise alignment distances; then, align the sequences following the guide tree. Corresponding to each node in the tree, the algorithm aligns the two sequences or alignments associated with its two daughter nodes. The process is repeated beginning from the tree leaves, which are the sequences, and ending with the

tree root.

Chapter 3

A Literature Review of Recent Progresses in Anchor-based Genome Sequence Alignment

MUMmer (Maximal Unique Match(mer))

MUMmer[19] is a pairwise anchor-based alignment program and it can detect every difference between two microbial genomes. The program can assemble two different versions of genome sequence: two drafts or a drafted and a complete genome. The anchors of this program are MUMs.

The system is packaged with three typical anchor-base alignment procedures: firstly, construct a suffix tree to find anchors; then, sort and extract the Longest Increasing Subsequence as an optimal chain; lastly, import the Smith-Waterman alignment for aligning all the regions between the anchors.

In MUMmer, a MUM is a maximal unique match, which is a subsequence that

occurs only once in both sequences and is not contained in a longer subsequence. Using the suffix tree method, MUMs can be computed in $O(n)$ time and space, where n is the length of both input sequences and the symbol appended. After the MUM decompositions have been sorted, the longest possible set of MUMs that occurs in the same order in both genomes can be extracted. The set is the Longest Increasing Subsequence, which is the anchor-chain for alignment. If there are m MUMs, this can be done in $O(m \log m)$ time. However, MUMer actually uses a simpler $O(m^2)$ time dynamic programming algorithm. At last, the gaps between the anchors are closed with a standard dynamic programming algorithm. The length of gap has a certain limit, and the default is 5,000 bp. Gaps longer than this limit are unaligned. This step takes $O(ll')$ time and $O(\min(l,l'))$ space for one gap that consists of two sequences of length l and l'.

The MUMmer program is a major solution for pairwise alignment of sufficiently similar whole genome sequences. MUMmer 1.0 was used to detect numerous large-scale inversions in bacterial genomes, leading to a new model of chromosome inversions. MUMmer 2.1 was used to align human chromosomes and detected numerous large-scale ancient segmental duplications in human genomes. MUMmer 3.0 has a completely rewritten core suffix tree library and is used for numerous applications [56].

However, the major drawback of MUMmer is that it can only align no more than two whole genome sequences.

The program is an open source package that is available at:

ftp://ftp.tigr.org/pub/software/MUMmer/

PipMaker (Percentage Identity Plot MAKER) and MultiPipMaker

PipMaker [52] is a web server that was designed to align two long DNA sequences to identify the conserved segments and produce informative, high-resolution displays of the resulting alignments. Now it is used for comparing genome sequences for two related species, although the information types depend on the level of conservation and the separation of the species. PipMaker can not only align those input sequences, but also summarize them with a percentage identity plot (PIP). It supports analysis of draft sequence to single reference sequence, but not a draft-to-draft comparison. The PipMaker program uses the *k-mers* (i.e. strings of length k) as the anchors.

In the anchor finding and chaining steps, PipMaker has two options. The program identifies only the anchors that appear in the same relative order in sequences when the invoking option "chaining" is selected. When the selected option is "single coverage", PipMaker avoids duplicate anchors by allowing only the highest scoring set of the alignments. In gap-closing step, the program uses the greedy algorithm.

The advanced version of PipMaker is the Advanced PipMaker program, and the

multiple input version is developed as MultiPipMaker [50]. They are also available at the same website. The MultiPipMaker program compares sequences pairwisely between the reference sequence and each of the secondary sequence that is computed by the *blastz* program [51]. MultiPipMaker is processed by an iterative refinement procedure from *ReAligner* [3] involving much more flexible alignment scores.

The main limitation of the PipMaker family was that they were only available on server; hence, the inputs were restricted. Last year, a beta version of PipMaker was developed. The server is located at: http://bio.cse.psu.edu/pipmaker.

GLASS (Global Alignment SyStem)

GLASS [5] was developed for the processing step of the gene prediction tool ROSETTA. It is also an anchor-based alignment tool for pairwise genome sequence alignment. GLASS is designed for aligning eukaryotic sequence model that contains long, weakly conserved introns and short, strongly conserved exons.

In the first step, GLASS searches all pairs of exact matching *k-mers* of two input sequences. Then, for a given pair of matching *k-mers*, a dynamic programming algorithm is applied to twelve nucleotides to the left of both *k-mers* and yields a score. DP does the same work to the right of both *k-mers* and yields the other score. It adds the two scores together to represent the score of the given pair of matching *k-mers*. In the second step,

DP computes the highest scoring sequence of *k-mers* that occurs in the same order in both sequences. Any matching *k-mer* will be removed if its score is below a given threshold or it inconsistently overlaps. The resulting *k-mers* serve as anchors in the alignment. Afterward, all the steps above are applied to the unaligned regions between the anchors recursively, with a decreasing value of k, namely 15, 12, 9, 8, 7, 6, 5. In the last step, all remaining gaps are aligned by a standard dynamic programming method.

However, GLASS gives a partial alignment of two sequences and its space requirement is very large. This program is not suitable for prokaryotes and may leave some unaligned regions in the input sequences.

GLASS is available at: http://crossspecies.lcs.mit.edu/.

WABA (Wobble Aware Bulk Aligner)

WABA [34] is the first alignment tool that accounts for divergence in the wobble position of coding regions. The anchors of this program are two 8-mers that ignore wobble bases in the 1 kb region. WABA works well to uncover exons. The key feature of WABA is that it is sensitive to the wobble base, which is the third base in a codon, treats it differently from other bases, because the mutations in this base are often silent in the sense that they do not change the corresponding amino acid. WABA was developed for separately aligning 229 different sequences from two closely related nematodes of the genus

Caenorhabditis: the *C. briggsae* and the *C. elegans*.

In the procedure of WABA, the two input sequences first break into short overlapping sequence fragments. Then, the homologies between those fragments and the other sequence are found. Two 8-mers, which ignore wobble bases in the 1 kb region, are found as anchors. This search is implemented in a modified gapped BLAST-like style. Then, homologous regions are aligned in an extended window using a pairwise hidden Markov model [23]. However, if any two of these local alignments overlap by at least 15 bp and are identical in overlapping regions, they are merged into one larger alignment. In WABA, high scoring pairs are not required to match exactly but may contain a mismatch every three bases. This is because the homologous regions in two related DNA sequences are most likely protein coding regions, so most point mutations occur in the third bases of a codon.

WABA was designed specially for certain sequences and is limited only to pairwise alignment. It is impractical for larger genome alignment.

LSH-ALL-PAIRS (Locality-Sensitive Hashing in All PAIRS)

LSH-ALL-PAIRS [11] is designed for finding ungapped alignments in genome sequences. The anchors of this program are gap-free fragments. The locality-sensitive hashing

method, which is an efficient randomized search technique, is used to look for those anchors and continue with the exact matching later. The exact matching requires selecting a minimum anchor length, which balances sensitivity and weak similarity against efficiency on long sequences, in order to reduce bias caused by random chance. This algorithm can find similar sequences in long anchors with frequent substitutions. It runs iteratively to reduce the risk of missing true positives with the random search. The overlapping part will be fixed into longer and ungapped local alignment.

LSH-ALL-PAIRS can only work for pairwise alignment and not yet for multiple genome sequences. Some other drawbacks are mentioned by Jeremy [11]: if the gaps between the segments are too small, they are likely to be missed in the initial random search; the long gapped similarities may be missed if their ungapped anchors do not score significantly; moreover, the initial anchor search is scored by a mismatch count not by a general score function.

CHAOS (CHAins Of Scores) + DIALIGN (DIagonal ALIGNment)

Almost all the available alignment programs before 1996 were developed to focus on relatively short sequences. The era of large-scale alignment algorithms began in 1996 with the versatile alignment program, DIALIGN [40] [13]. The DLALIGN method can

deal with both pairwise and multiple alignment. The first version can only use single bases for comparison, but the new version can use gap-free whole segments. The anchors of this program are fragments of equal length that form diagonals in a dot-matrix comparison. Quality scores will be assigned to those fragments based on the probability of their random occurrence and look for a collinear collection of non-overlapping fragments with maximum total score.

In the pairwise alignment case, DIALIGN is trying to find, through a modified dynamic programming scheme, the fragments that have the maximum sum of scores over all the optimal alignments. In the multiple alignment case, DIALIGN employs a greedy algorithm. It first creates all the pairwise alignments. Then the fragments contained in those pairwise alignments are sorted according to their scores and the degree of overlap with each other. After that, they are integrated into a growing multiple alignment. The result shows that the fragments are collinear with the alignment, and the non-colinear parts will be discarded. In the case that no additional fragment can be combined, gaps that are without gap penalty will be imported to arrange the selected segment pairs.

However, DIALIGN cannot handle very large and complicated genome sequences, because it takes too much time and requires too much memory. One way of speeding-up DIALIGN without compromising on alignment quality is to use the anchored-alignment procedure. The program called CHAOS [8] [10] is developed for rapid identification of

chains of local pair-wise sequence similarities. In the first step of whole genome alignment, it calculates local alignments as anchors.

Every genome alignment tool has to solve the chaining problem somehow [2]. CHAOS chains together pairs of similar regions that are anchors, one from each input sequence. An anchor can be chained to the other only if the indices of one are higher than the other and the distance and gap criteria are near to each other. The final score of the chain is the total number of the matching basic pairs in it. After computing the maximal chains, CHAOS scores each chain by using match and mismatch penalties for the base in each anchor, and throws away chains below a certain threshold.

After CHAOS identifies a collection of local alignments for the pair of input sequences, an algorithm based on the longest increasing subsequence is used to find the highest scoring chain as the anchor-chain in the next step. This step takes $O(n \log n)$ time, where n is the number of local alignments.

For pairwise alignment, the chain can be directly used into DIALIGN alignment. For multiple alignment, in the first step, CHAOS is applied to all possible pairs of input sequences to get a list of similarities which can be considered as candidates for anchor points. Then, the greedy algorithm, which DIALIGN uses to find consistent sets of local pairwise alignment during the multiple alignment calculation, is employed to solve the problem in the case that the similarities contradict each other. Each of the candidate

anchors is sorted by the quality score that is associated with them. Starting from the one with the highest score, those anchors are accepted as final anchor points if they do not contradict with others. So the set of pairwise anchor points has been found to fit into one multiple alignment in the greedy procedure of DIALIGN.

Anchor points created by CHAOS speed-up DIALIGN by one to two orders of magnitude without reducing the alignment quality. CHAOS+DIALGN can align large genome sequences very fast and sensitively.

The drawback of this program comes from the nature of the greedy algorithm. In the program, once a fragment has been put into the alignment, it is fixed and cannot be removed. This problem always happens in methods that involve greedy algorithms. The results may be misaligned, especially where the sequence contains a repeating part. Recently, Some new strategies, e.g. the sequence clustering algorithm-BAG [14], have been proposed. They can be used to deal with this problem.

The website to run the CHAOS+DIALGN program is:

http://dialign.gobics.de/chaos-dialign-submission

MGA (Multiple Genome Aligner)

MGA [30] use an anchor-based method to produce a global multiple alignment for closely related whole genomes. The anchors of this program are multiMEMs, which is the

maximal multiple exact matches.

In the first phase of this method, all the multiMEMs whose lengths exceed a given threshold are detected. The anchor (multiMEM) is a small sequence that occurs in all genomes sequences and cannot simultaneously be extended to the left or right maximality in each genome. Those multiMEMs are computed in there steps. First, the program constructs a virtual suffix tree of all the genome sequences and different separator symbols that do not occur in any of the genomes. This step takes $O(n)$ time and space, where n is the length of the sequence that comes from combining all the genome sequences and those separator symbols. Next, for every node of the suffix tree, a set of all the positions in the sequence that made from the genome sequence and symbols is computed. Then the set is divided into pairwise disjoint and possible empty position sets. If all position sets are not empty, a maximum exact match has been found to occur in each of the genome sequences at certain positions. Because the right maximality has been ensured during the incremental computing, the maximum exact match is ensured to be a multiMEM by comparing to the left characters to check if it is left maximal. The first step takes $O(kn+r)$ time to compute all multiMEMs, where k is the number of genomes, n is their total length and r is the number of right maximal multiple exact matches. The later version of MGA changed the suffix tree method to enhanced suffix arrays, which makes the method more efficient.

In the second phase, MGA computes the anchors consisting of the longest non-overlapping sequence of multiMEMs that occur in the same order in each genome. Every multiMEM is viewed as a *k*-dimensional cube in the Euclidean space with associated weight. In order to find the best non-overlapping sequence, the maximum weight chain has to be found. The problem has been studied before, and can be solved by constructing a weighted acyclic directed graph. A maximum weight chain of cubes corresponds to a path with maximum weight from the starting vertex to the stopping vertex in the acyclic graph. Because there are $O(m^2)$ edges in the graph, this phase needs $O(km^2)$ time to computer the chain, where m is the number of multiMEMs. This makes the time of the algorithm run up to quadratic. Later, an algorithm based on *kd-trees* is used, utilizing the genomic nature of the input data. The running time of this case cannot be precisely analyzed because of the nature of *kd-tree* algorithm, but it was proved to be practical.

In the third phase, MGA uses the progressive multiple alignment tool CLUSTAL W to close the gaps between the anchors and computes the alignment result.

In practice, MGA works well for aligning similar bacterial and double-stranded DNA viral genomes. The drawback of MGA is that it requires all the anchors existing in all the input sequences. MGA cannot find enough anchors for many short single-stranded RNA viral genomes and the search for relatively short anchors exhausts its memory.

MGA is available at: http://bibiserv.techfak.uni-bielefeld.de/mga/download.html.

EMAGEN (Efficient Multiple Alignment algorithm for whole GENomes)

EMAGEN [20] is also an anchor-based multiple whole genome alignment program. It first finds the anchors among multiple genomes in linear time and it works especially well on prokaryotic genomes. The anchors of this program are MUMs: maximum unique matchs.

The first phase is to find the anchors. EMAGEN uses a suffix array algorithm to find MUMs among the multiple whole genome data group. It creates a generalized suffix array for the concatenated string *S* of input data and different separator symbols. At the same time, it put three more arrays in the data structure. One is *lcp*, which is the longest common prefix of the suffix array. Another is *ps*, which is the proceeding symbol of the *S* sequences in the suffix array. The other is *so*, which is the order of the sequence within which the suffix *S* sequences begins in the suffix array. Then the program looks for the *MUM-Intervals* and outputs the MUMs. An interval in the input sequences can be called a *MUM-Interval* if the *so* parameters are pairwise distinct and the *ps* parameters are not the same value. So, the *lcp-string* of a *MUM-Interval* can be output as a MUM.

EMAGEN uses an efficient method to find all the *MUM-Intervals*. It first scans the

suffix array to locate a maximum interval, then checks if it is a *MUM-Interval*. All the *MUM-Intervals* can be found by checking the suffix array once, hence, all MUMs can be found in linear time. However, during our research, we have doubts with this method.

In the second phase, graph theory has been employed to choose the optimal chain. EMAGEN constructs a MUM diagram according to the MUMs that have been found in the first step, and a MUN graph is defined accordingly. This step takes $O(km^2)$ time, where k is the number of input sequences and m is the number of MUMs. After that, the program finds a maximum independent set of the MUM graph as alignment chains: LIS-MUMs, which is the longest increasing subsequence of the MUM graph. The longest increasing subsequence is the largest subset of the MUMs which appears in ascending order in each MUM sequence. The MUMs in LIS-MUMs do not cover each other. This step takes $O(m+e)$ time, where e is the number of edges in the complement graph of the MUM graph.

EMAGEN includes a special method for aligning the coding regions among multiple prokaryotic genomes, which constructs concatenated amino acid sequences to represent genomes instead of the original nucleotide sequences. The maximum sets of conserved regions from these long amino acid sequences are found as anchors for alignment. These anchors are actually short amino acid subsequences, which are mapped back to nucleotide sequences positions.

In the third phase, the gaps between LIS-MUMs are aligned by CLUSTAL W [54]. The program sets a threshold as the maximum length of the gaps that should be aligned.

MAUVE

Mauve [18] is a new tool for multiple whole genome alignment. It is the first alignment system that integrates analysis of large-scale evolutionary events with traditional multiple sequence alignment. It performs better than other systems for comparing genomes with significant rearrangements. Mauve also falls into the category of anchor-based alignment tools. The anchors of this program are Multi-MUMs of some minimum length. However, unlike other systems, the input genomes of Mauve's selection method do not necessarily have to be collinear. Instead, Mauve identifies and aligns regions of local collinearity called locally collinear blocks (LCBs), which are the homologous regions of sequences shared by two or more input sequences, and do not contain any rearrangements of homologous sequence.

Firstly, Mauve uses a simple seed-and-extend hashing method to find multi-MUMs, which are the Multiple Maximal Unique Matches. Although the algorithm takes $O(G^2 n + Gn \log Gn)$ time, where G is the number of input sequences and n is the average genome length, it performs fast in practice. Mauve uses the information from the subset multi-MUMs as a distance metric to construct a phylogenetic guide tree using

Neighbor Joining method. Then, it tries to select a proper subset of the multi-MUMs as anchors, because some sets may contain spurious matches due to random sequence similarity. This can be done when determining the boundaries of locally collinear blocks. Given a minimum weight criterion, Mauve uses a greedy breakpoint elimination algorithm to remove low-weight collinear blocks of the set. Because this anchoring step may not be sensitive enough to detect the full region of homology within and surrounding the LCBs, the program uses the existing anchors as a guide to perform recursive-anchoring repeat. Mauve searches the regions outside of LCBs to extend the boundaries of existing LCBs and to identify new ones. It also searches the unanchored regions within LCBs for additional alignment anchors. Unlike other methods that perform a fixed number of recursives passed with a predetermined sequence of anchor sizes, in this program, the minimum anchor size is based on the sequences and recursive-anchoring will stop either when no additional anchors are found or the length of the intervening region is smaller than a certain border. After getting a complete set of alignment anchors, in the last phase, Mauve uses CLUSTAL W to calculate a global alignment over each LCB.

This program works well for nine enterobacteria. As the writer mentioned, a more sophisticated rearrangement scoring method may improve the system. The program is free available at: http://gel.ahabs.wisc.edu/mauve/.

LAGAN (Limited Area Global Alignment of Nucleotides) and Multi-LAGAN

As most methods work efficiently in aligning closely related genome sequences, LAGAN [9] system is tested on alignments between distant relatives such as human and fugu. LAGAN is an efficient and reliable pairwise aligner even for genomes from distantly related organisms, and Multi-LAGAN is a multiple aligner based on progressive alignment with LAGAN.

LAGAN aligns pairwise genome sequences in the three phases, which anchor-based alignment usually has. In the first step, the program is to compute the local alignment between two sequences and assigns a weight to each local alignment. It uses CHAOS [8][10] to find local homologies between two sequences. Besides CHAOS, any efficient local alignment method can also be used for this task. The details about CHAOS can be referred to in the description of CHAO+DIALIGN above. After CHAOS finds the local alignments, LAGAN orders them into a rough global map. The highest-scoring chain is the optimal rough global map, which can be computed using Sparse Dynamic Programming in $O(n \log n)$ time, where n is the total number of local alignments [24]. Then, LAGAN uses a recursive method similar to the one used in GLASS [5] to try to get a trade-off of speed and sensitivity. During the recursive-anchoring step, LAGAN uses CHAOS with some restrictive parameters to compute a rough global map based on the

resulting local alignments. CHAOS has been used recursively with more permissive sets of parameters in the regions between each anchor of the global map. One thing has to be mentioned here, that some recursive anchoring steps can be translated. After that, LAGAN uses dynamic programming to compute the final global alignment and it uses the rough global map to limit the search area. For every anchor in the rough global map, LAGAN limits the computation of Needleman-Wunsch algorithm in two corner rectangles and the diagonal areas of the DP table. Hence, the anchors in this program are more flexible and provide only approximate locations by which the alignment should pass. In practice, LAGAN uses a memory-efficient idea that performs the entire computation with memory proportional to the size of the largest rectangle. Besides, if the anchors are about evenly spaced and get a constant density, the time complexity of the program can be linear.

Multi-MLAGAN is a tool for multiple genome alignment. It includes a progressive alignment phase based on LAGAN and an optional iterative improvement phase. It first finds the rough global maps between each pair of sequences. During the progressive alignment, Multi-MLAGAN imports LAGAN to give global alignment of the two closest sequences according to their order in the given phylogenetic tree. Then, it finds the rough global maps of the produced alignments (which are between two or more sequences) to other produced alignments. Afterward, the program iterates the two steps above to

perform a global alignment in every step and repeats until it gets a multiple alignment of all sequences. Each step merges two sequences or alignments into a larger alignment and constructs a profile of all the sequences. This program uses a combination of scoring approach: sum-of-pairs for substitution and consensus for gaps, which is the most similar to the CLUSTAL W method. However, the difference is that CLUSTAL W heuristically weights per-sequence penalties to score gaps while Multi-MLAGAN uses appropriately scaled consensus. In the optional iterative anchor refinement phase, Multi-MLAGAN performs a limited-area idea that performs more work in the needed area and allows large-scale adjustment: each sequence will be removed iteratively and every region, which is in the removed sequence and improves the alignment score significantly, is an anchor. Then it aligns each sequence to the multiple alignment of the other sequences with LAGAN.

LAGAN and Multi-MLAGAN both take advantage of some existing efficient methods, combine them and improve them to align 12 genome sequences, some of which are not closely related. However, because Multi-LAGAN performs progressive pairwise alignments that are guided by a user-specified phylogenetic tree, it still has some drawback as other progressive alignment methods: it might focus on a local optimal alignment and cannot get the global optimal solution. Besides, because the method uses sum-of-pairs metric, which is known to be NP-Complete [59], to align alignments, it will

consume more time.

The alignment server is available at: http://lagan.stanford.edu/lagan_web/.

AVID and MAVID

AVID [6] is a global alignment program for large genomic regions up to the megabase range. The input of this program is two genome sequences and the output is a global alignment with some additional information, e.g., an overall score. At first, the input sequences can be processed with the RepeatMasker program [48]. But different from the original program, AVID keeps both the masked and unmasked sequences used into the alignment process. The "match", which is maximal but not necessarily unique, can be divided into two groups: those overlapping repeats (repeat matches) and those not overlapping ones (clean matches). Each is used in a different way. The program transforms the problem of finding maximal repeated substrings in one string to find all maximal matches between two sequences. It uses a generalized suffix tree data structure of two sequences to find those matches.

After the matches have been found, AVID begins to the recursive process of anchoring and aligning. The anchor set here is a collection of non-overlapping, non-crossing matches. The program uses a heuristic to remove matches that are less than half the length of the longest match from initial consideration and the shorter matches will be

reconsidered for anchoring later. This is done in a certain order: first, clean matches are sorted by length; then, repeat matches are sorted when there are no more clean matches. Those anchors are selected using a different version of Smith-Waterman algorithm [27], which are required to be non-overlapping. The gap scores zero, the mismatch scores infinity, and the match scores based on its length and the alignment score of the regions flanking the match (10 bp on each side). This anchor-selecting process is similar to the GLASS method [5].

Once the anchors have been selected, they will form part of the final global alignment as a set. The program will check each match to see whether it lies entirely between two sets of anchors. Once the maximal matches have been found, the smaller regions between the anchors will be realigned using the anchor selection step before. This recursion will terminate when either no remaining bases are aligned or no significant matches exist in the remaining sequences.

AVID can order and orient draft sequences by using comparisons a finished sequence. It works well but only can handle two sequences. However, when the aligned regions are short enough to perform an optimal alignment, AVID will use anchors only if the total length of the anchor set is $> 50\%$ of the sequence length; otherwise, it will use the standard Needleman-Wunsch algorithm [27] to align those regions. And if the sequences are short ($\leq 4kb$ each), AVID will align them by the Needleman-Wunsch algorithm and

return a trivial alignment, where both sequences are completely gapped.

AVID can be used online at http://math.berkeley.edu/avid/

In order to improve AVID to deal with a large number of genomic regions, MAVID [7] was proposed one year later for obtaining a global multiple alignment. In this method, the gene-base anchors constrain a progressive alignment to incorporate biological information into the alignment procedure. These anchors will be computed firstly according to gene prediction and their protein alignments, and then assemble into the program as input data.

The core in MAVID is a progressive ancestral alignment that incorporates preprocessed constraints. Given a phylogenetic tree, the program constructs the alignments of all the sequences in the tree by aligning alignments recursively from leaves to root, and associates them into the vertices. Sequences are aligned by the AVID program after the ancestral sequence calculation. The alignment result will glue two alignments together to produce a new multiple alignment in the vertex. This procedure terminates with a final pairwise alignment at the root node.

The gene matches and constraints are based on a homology map for the input sequences and MAVID identifies the order and orientation of matching gene runs between the sequences. Gaps are assigned a linear gap penalty but prefer an affine gap penalty [7].

The MAVID program is based on AVID and other existing models of multiple

genome sequence alignment. The drawback of MAVID comes from the greedy nature of the progressive method, and the iterative algorithm is less sophisticated than some other exiting methods. However, in practice, the approach can deal with larger multiple problem, divergent sequences, as well as incomplete unfinished sequences reasonably quickly.

The program is available at: http://baboon.math.berkeley.edu/mavid/

Chapter 4

Our Chaining Algorithm

4.1 Our Ideas and their Origins

The anchor-based alignment approach divides initial large alignment problems into smaller, more manageable ones and combines program speed and sensitivity [10], which is a good solution for whole genome sequence alignment tasks. The procedure of the anchor-based whole genome alignment can be divided into three phases [13]:

1) Computation of all the anchors;
2) Computation of an optimal anchor-chain of collinear non-overlapping anchors: the anchors that form the basis of the alignment;
3) Alignment of the regions between the anchors.

We propose a chaining algorithm as one part of our model in the second phase. The algorithm uses the dynamic programming technique and is based on the standard Longest Common Subsequence idea.

The quality of a whole genome alignment method is measured not only by the running efficiency, but also by the biological significance [10] [7]. Therefore, it is important to involve biological ideas to improve the alignment quality and practicality. We place a weight on every anchor in order to find a biologically more correct anchor-chain. We believe that this idea can help our alignment model obtain a more meaningful result. After some helpful talks with biologists, we determined that our weight tends to be related to the length of the anchor. This is based on biological evolutionary theory, which was summarized in Chapter 1. If the large-scale sequences are assumed to be whole genome sequences, every anchor can be considered a conserved nucleotide block. According to evolutionary theories such as natural selection, the longer the block is, the more important the evolutionary information and structure it might contain. The reason for this is that only very valuable nucleotide blocks can survive during those significant sequence changes that result from selective pressures. During evolution, there are likely certain important reasons to keep some nucleotide blocks that do not easily change. According to this idea, the longer the block is, the heavier the weight we put on it.

We refer to this chaining procedure as a problem of finding the *Multiple Heaviest Common Subsequence (MHCS)* or the *multiple maximum weight common subsequence (MMWCS)*, which is the common subsequence with maximum weight in multiple

weighted sequences.

4.2 Computational Complexity

4.2.1 Definition of the Problem

We now formally define this problem.

Given a finite sequence $S = \langle s_1, s_2, ..., s_m \rangle$, a subsequence S' of S is any sequence that consists of S with k terms deleted, for $k \in [0, m]$. Given a set $R = \{S_1, S_2, ..., S_r\}$ of sequences, a *Common Subsequence* is a sequence that is the subsequence of each sequence $S_1, S_2, ..., S_r$ in R. In the weight set $W = \{w_1, w_2, ..., w_t\}$, $w_1, w_2, ..., w_t$ are the real numbers associated with each character in those sequences.

Definition 4.2.1.1 *Multiple Maximum Weight Common Subsequence (MMWCS)* problem or *Multiple Heaviest Common Subsequence (MHCS)* problem:

Given a multiple sequence set $R = \{S_1, S_2, ..., S_r\}$ with a particular weight w assigned to every character of each sequence S, what is the common subsequence with the maximum weight, i.e., what is the *MHCS(R)*?

The decision version of the problem is as follow. Given R and an integer bound B, is

the weight of an *MHCS(R)* greater than *B*?

4.2.2 The *Multiple Heaviest Common Subsequence* Problem is NP-Complete.

THEOREM 1 (COMPLEXITY) The decision version of the *Multiple Heaviest Common Subsequence* problem belongs to NP-Complete.

To prove this theorem, we reduce the Longest Common Subsequence problem to it.

4.2.2.1 The Restriction Technique of Proving NP-Completeness

As we know, there are various techniques for proving NP-Completeness. We use the restriction technique to prove the *MHCS* problem.

The restriction technique is the most frequently used proof type for the NP-Complete problem [25]. Garey and Johnson mentioned in their book that an NP-completeness proof by restriction for a given problem $\Pi \in$ NP consists of showing that Π contains a known NP-complete problem Π' as a special case. The keystone of this proof is to place the specification of the additional restrictions on the instance of Π, so that the resulting

restricted problem will be identical to Π'. The restricted problem and the known NP-complete problem are not required to be exactly the same, but there must be a clear one-to-one correspondence between their instances that preserves "yes" and "no" answers.

The restriction proof technique is different from the standard NP-completeness proofs. Instead of trying to discover a way of transforming a known NP-complete problem to the target problem, the technique focuses on the target problem itself and tries to restrict the inessential aspects to show the NP-Completeness of the problem.

4.2.2.2 The Complexity of the Longest Common Subsequence problem

The Longest Common Subsequence (LCS) problem has been described in Chapter 2. Here, we only focus on the complexity issue of this problem. When an arbitrary number of sequences is considered, this problem is proved to be NP-Complete [37].

The yes/no version of the problem is: given an integer k and a listing of the sequences in $R = \{S_1, S_2, ..., S_p\}$, is $|\mathrm{LCS}(R)| \geq k$?, where $|\ |$ denotes the cardinality of the set. $\sum(R)$, which is defined to be the alphabet of R, is the finite set of values in terms of sequence $S_1, S_2, ..., S_p$.

The proof is done by the reduction of the vertex cover problem, which is one of the

six basic NP-Complete problems.

Given an undirected graph $G=(N,E)$ and an integer k, the vertex cover problem is to determine if there is an $N' \subseteq N$, for $|N'|=k$, such that for every $(x,y)\in E$, either $x\in N'$ or $y\in N'$ (possibly both). The edge of E is assumed to be $k;(x_1,y_1);(x_2,y_2);...;(x_r,y_r)$, which is encoded into a string of length n. An arbitrary order $\{v_1,v_2,...,v_t\}$ is assigned to N. Here, $r,t\le n$. $r+1$ sequences of length at most $2(t-1)$ has been constructed as shown in Figure 2. The first sequence is the template sequence T, which is the sequence $v_1,v_2,...,v_t$. A sequence S_i is constructed for each edge $e_i=(x_i,y_i)$ in E. Assume without loss of generality that $x_i=v_j$, $y_i=v_m$ and $j<m$. Then S_i is $v_1,v_2,v_3,...,v_{j-1},v_{j+1},...,v_m,...v_t,v_1,v_2,...,v_j,...,v_{m-1},v_{m+1},...,v_t$.

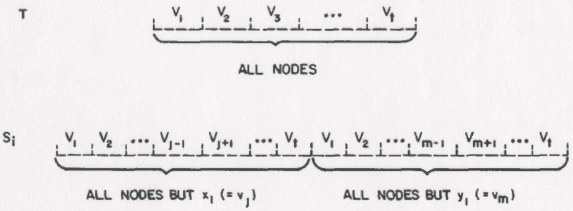

Figure 2: The template sequence T and the sequence S_i [37].

It has been proved that the graph G has a vertex cover of size k if and only if the

set $R=\{T,S_1,S_2,...,S_r\}$ has a common subsequence of size $t-k$. Hence, the minimal vertex cover of *G* has size *k* if and only if LCS(*R*) has size *t-k*. If the vertex cover problem has length *n*, the input for the LCS algorithm is of length $t+2r(t-1)\le O(n^2)$. So the construction can be done in polynomial time. Because of the polynomial reduction from the vertex cover problem, the LCS problem for $\Sigma(R)$ of arbitrary size is NP-Complete [37].

4.2.2.3 Proof of the NP-Completeness for the *MHCS* problem

Now we prove that the *MHCS* problem is NP-Complete.

The instance is: given a set of sequence $R=\{S_1,S_2,...,S_p\}$ and a weight set $W=\{w(x_1),w(x_2),...,w(x_t)\}$ for the alphabet of *R*, $\Sigma(R)$, whose size $|W|=|\Sigma(R)|$. Clearly $|\Sigma(R)|\le m_1+m_2+...+m_p$, where $m_i=|S_i|$. $MHCS(R)$ represents the weight of the heaviest common subsequence of *R*.

The yes/no version of the problem is: given an integer k, a listing of the sequences in R and a listing of the weights in *W*, is $MHCS(R)\ge k$?

Proof: First, it is easy to see that $MHCS\in NP$, since a nondeterministic algorithm need only guess a *k* and check in polynomial time whether the weight of the heaviest

common subsequence is larger than or equal to k, after the weights have been assigned to the alphabets.

Next, we use the restriction technique. We restrict the *MHCS* problem for $\sum(R)$ of arbitrary size by allowing only instance with weight 1 in $\sum(R)$. Then, the restricted *MHCS* problem becomes the LCS problem for $\sum(R)$ of arbitrary size. In other words, the LCS problem is a special case of the *MHCS* problem.

Therefore, the *Multiple Heaviest Common Subsequence (MHCS)* problem is NP-Complete.

4.3 Algorithm Description

We propose an algorithm for solving the *MHCS* problem with the idea of extending the dynamic programming technique of the standard longest common subsequence method. The *MHCS* problem has been proved to be NP-Complete, which means that no polynomial time algorithm exists for this problem unless P = NP. Moreover, with regard to the theory of parameterized complexity, an approach to attack intractable problems mainly developed by Doweny and Fellows [22] [21], the fixed alphabet longest common subsequence parameterized in the number of strings (FLCS) has recently been proved to be $W[1]$-hard [45]. Therefore, we can say that, in general, no exact polynomial-time algorithm can find an exact anchor-chain from arbitrary numbers of weighted sequences.

However, traditionally, for all the genome alignment programs, the number of the input sequences is forced to be limited to ignore the computational complexity. We limit the number of the input genome sequences, then, this algorithm can find the result in polynomial time. We describe two cases here: the case for three sequences and the case for k sequences, with a fixed integer k.

4.3.1 The Algorithm for 3 Sequences and Its Complexity Analysis

The input of the algorithm are three sequences $X = \langle x_1, x_2, ..., x_m \rangle$, $Y = \langle y_1, y_2, ..., y_n \rangle$, $Z = \langle z_1, z_2, ..., z_l \rangle$ and a weight set for all the characters of those sequences. x_i is a character in X; y_j is a character in Y; and z_t is a character in Z. $c[i, j, t]$ represents the weight cost of the *heaviest common subsequence*. A dynamic programming table $b[1..m, 1..n, 1..l]$ is maintained to simplify construction of the optimal solution. Therefore, the recursive formula is:

$$c[i,j,t] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \text{ or } t=0, \\ c[i-1,j-1,t-1]+w(x_i) & \text{if } i,j,t>0 \text{ and } x_i = y_j = z_t, \\ \max(c[i-1,j,t], c[i,j,t-1], c(i,j-1,t]) & \\ & \text{if } i,j,t>0 \text{ and } x_i \neq y_j \text{ or } y_j \neq z_t. \end{cases}$$

Clearly, the lengths of those three sequences are *length* [X] = m, *length* [Y] = n,

length [*Z*] = *l*.

Therefore, the procedure is:

MHCS-weight (X, Y, Z)

```
1   m ← length [X]
2   n ← length [Y]
3   l ← length [Z]
4   for j←1 to n
5       do for t←1 to l
6           do c[0, j, t] ←0
7   for i←1 to m
8       do for t←1 to l
9           do c[i, 0, t] ←0
10  for j←1 to n
11      do for i←1 to m
12          do c[i, j, 0] ←0
13  for i←1 to m
14      do for j←1 to n
15          do for t←1 to l
16                 do if  x_i = y_j = z_t
17                        then c[ i, j, t ] ←c[ i-1, j-1, t-1]+ w(x_i)
18                             b[i, j, t]←1
19                        else if c [i-1,j ,t] = max ( c[i-1, j, t], c[i, j-1, t], c[i, j, t-1] )
20                             then c[i, j, t] ←c[i-1, j, t]
21                                  b[i, j, t]←2
22                        else if c [i, j-1, t] = max ( c[i-1, j, t], c[i, j-1, t], c[i, j, t-1] )
```

```
23                    then c[i, j, t] ←c[i , j-1, t]
24                        b[i, j, t]←3
25                    else c[i, j, t] ←c[i , j , t-1],
26                        b[i, j, t]←4
27  return c and b
```

PRINT- *MHCS* (b, c, X, i, j, t)

```
1   if i = 0 or j = 0 or t = 0
2       then return
3   if b[i, j, t] = 1
4       then PRINT- MHCS (b, X, i-1, j-1, t-1), Print x_i
5   else if b[i, j, t] = 2
7       then PRINT- MHCS (b, c, X, i − 1, j, t)
8   else if b[i, j, t] = 3
9       then PRINT- MHCS (b, c, X, i, j − 1, t)
10   else PRINT- MHCS (b, c, X, i, j, t − 1)
```

The running time of the *MHCS-weight (X, Y, Z)* is $\Theta(mnl)$ and the running time of the PRINT- *MHCS* (b, c, X, i, j, t) is $\Theta(m+n+l)$. We can see that line 13 to line 27 in *MHCS-weight (X, Y, Z)* dominant the total running time.

4.3.2 The Algorithm for k Sequences and Its Complexity Analysis

Given k sequences in the sequence set $\alpha = \langle X_1, X_2, X_3, ..., X_k \rangle$, let T = $\langle$ c[i_1-1, i_2, ..., i_k], c[i_1, i_2-1, ..., i_k]...., c[i_1, i_2, ..., i_k-1] $\rangle$ and w is the corresponding weights.

Here is the procedure:

MHCS- weight $(X_1, X_2, X_3, ..., X_k)$

X_1.length $\leftarrow$ *length*$[X_1]$

X_2.length $\leftarrow$ *length*$[X_2]$

:

X_k.length $\leftarrow$ *length*$[X_k]$

for $i_2 \leftarrow 1$ **to** X_2.length

do for $i_3 \leftarrow 1$ **to** X_3.length

:

do for $i_k \leftarrow 1$ **to** X_k.length

do c[$0, i_2, i_3, ..., i_k$] $\leftarrow 0$

:

:

for $i_1 \leftarrow 1$ **to** X_1.length

do for $i_2 \leftarrow 1$ **to** X_2.length

:

do for $i_{k-1} \leftarrow 1$ **to** X_{k-1}.length

do c [$i_1, i_2, ..., i_{k-1}, 0$] $\leftarrow 0$

for $i_1 \leftarrow 1$ **to** X_1.length

do for $i_2 \leftarrow 1$ **to** X_2.length

do for $i_3 \leftarrow 1$ **to** X_3.length

:

do for $i_k \leftarrow 1$ **to** X_k.length

do if $X_{1\,i_1} = X_{2\,i_2} = \ldots = X_{k\,i_k}$

then c[i_1, i_2, ..., i_k] = c[i_1-1, i_2-1, ..., i_k-1] + w

b [i_1, i_2, ..., i_k] $\leftarrow 1$

else if c[i_1-1, i_2, ..., i_k] is max of T

then c[i_1, i_2, ..., i_k] $\leftarrow$ c[i_1-1, i_2, ..., i_k]

b [i_1, i_2, ..., i_k] $\leftarrow 2$

else if c[i_1, i_2-1, ..., i_k] is max of T

then c[i_1, i_2, ..., i_k] $\leftarrow$ c[i_1, i_2-1, ..., i_k]

b [i_1, i_2, ..., i_k] $\leftarrow 3$

:

else if c[i_1, i_2, ..., i_{k-1}-1, i_k] is max of T

then c[i_1, i_2, ..., i_k] $\leftarrow$ c[i_1, i_2, ..., i_{k-1}-1, i_k]

b[i_1, i_2, ..., i_k] $\leftarrow k$

else c[i_1, i_2, ..., i_k] $\leftarrow$ c[i_1, i_2, ..., i_{k-1}, i_k-1]

b[i_1, i_2, ..., i_k] $\leftarrow k+1$

return c and b

PRINT- *MHCS* (b, c, X_1, i_1, i_2, ..., i_k)

if i_1=0, i_2=0, ..., or i_k=0

then return

if b[i_1, i_2, ..., i_k] =1

then PRINT- *MHCS* (b, c, X_1, i_1-1, i_2-1, i_3-1, ..., i_k-1), Print $X_{1\,i_1}$

else if b[i_1, i_2, ..., i_k] = 2

then PRINT- *MHCS* (b, c, X_1, i_1 -1, i_2 , i_3 ,..., i_k)

else if b[i_1, i_2, ..., i_k] = 3

then PRINT- *MHCS* (b, c, X_1, i_1 , i_2 -1 , i_3 ,..., i_k)

:

else if b[i_1, i_2, ..., i_k] = k

then PRINT- *MHCS* (b, c, X_1, i_1 , i_2 , i_3 ,..., i_{k-1} -1, i_k)

else PRINT- *MHCS* (b, c, X_1, i_1 , i_2 , i_3 ,..., i_{k-1} , i_k -1)

The running time of the *MHCS- weight* $(X_1, X_2, X_3, ..., X_k)$ is Θ (X_1 .length · X_2 .length · ... · X_k .length) and the running time of the PRINT- *MHCS* (b, c, X_1, i_1, i_2,..., i_k) is Θ(X_1 .length + X_2 .length + ··· + X_k .length).

4.4 Implementation and Results

We use JAVA language to implement the *MHCS-weight (X, Y, Z)* and PRINT-*MHCS* (b, c, X, i, j, t). The program runs fast on our Intel Pentium III processor $1.20\,GH_z$, with $30\,GB^2$ hard drive.

If the input sequences are $X = \langle A, B, C, D \rangle$, $Y = \langle B, C, D, A \rangle$, $Z = \langle D, A, B, C \rangle$ and the user-defined weights are $A \leftarrow 1$, $B \leftarrow 1$, $C \leftarrow 1$, $D \leftarrow 1$, the result of the program is BC.

If we define the weights to be $A \leftarrow 5$, $B \leftarrow 1$, $C \leftarrow 1$, $D \leftarrow 1$, the result is A.

If we define the weights to be $A \leftarrow 5$, $B \leftarrow 5$, $C \leftarrow 1$, $D \leftarrow 1$, the result is BC.

If we define the weights to be $A \leftarrow 5$, $B \leftarrow 1$, $C \leftarrow 1$, $D \leftarrow 5$, the result is A.

If we define the weights to be $A \leftarrow 1$, $B \leftarrow 1$, $C \leftarrow 1$, $D \leftarrow 5$, the result is D.

The running results show that different weights assigned to different characters of the input sequences lead to different output solutions.

Chapter 5

The Whole Procedure of Our Model

5.1 Our Ideas and their Origins

After proposing our algorithm of finding the anchor-chain, we now describe the whole procedure of our anchor-based global multiple alignment model for whole genome sequences.

In the first phase, we use the enhanced suffix array method to find the conserved blocks among the input genome sequences. Because these conserved blocks are more likely to belong to the global alignment, they are used as anchors for assembling the multiple genome alignment.

In the second phase, we first weigh the anchors based on their lengths. Next, we use our chaining algorithm to find the *heaviest common subsequence* as the anchor-chain. Then, all the anchors are assembled based on this anchor-chain. In our model, we propose a novel alignment method to assemble the anchors. This method will make our model

more flexible for different input sequences and user requirements. After consulting with biologists who are currently using sequence alignment tools to help their evolutionary experiment, we realize that a tool for aligning the genome sequences of distantly related species and assembling large numbers of genome sequences are desired. Referring to our survey, most programs work efficiently in aligning closely related genome sequences and small number of input genomes (usually less than 15 sequences). Only very few genome alignment programs can align distant homologs and they usually cannot work efficiently for more than 12 sequences [9]. Therefore, we use a different aligning structure to assemble the anchors. For small numbers of closely related genome sequences, this model uses our chaining algorithm to find the anchor-chain and obtain an alignment from all anchors, which is the same as most alignment programs. However, when the inputs are many genome sequences from distantly related species, the model will use a new strategy: it asks users to choose the genome sequences that are from close homologs (i.e. from closely related species). Then, it uses the chaining algorithm to find an anchor-chain from these chosen sequences. Afterward, those unselected anchor sequences append to the anchor alignment iteratively based on the anchor-chain. This idea makes our model suitable for aligning not only closely related genome sequences but also distantly related ones, and it helps our model to align even large numbers of input genome sequences. Moreover, this method will lead to an evolutionary more correct and meaningful

anchor-chain. Because the inputs are genome sequences, every anchor found in the first phase consists of nucleotides. For closely related species, these nucleotide blocks are very likely to represent the same or similar traits that are beneficial to evolutionary research. However, for distantly related species, though the constituent nucleotides are the same, these blocks may not represent similar traits. In evolution, the anchors/nucleotide blocks from the closely related species may come from the same ancestor and be very meaningful, but those from the distantly related species may be just a result of unexpected mutation. If the anchor-chain is computed from all the anchor/nucleotide blocks from both closely related and distantly related genome sequences, this computing procedure will chain the anchors that have the same components together; however, this anchor-chain may only have structural meaning but not any evolutionary meaning. Hence, for genome sequences at any evolutionary distance, our strategy produces an evolutionary more correct anchor-chain that leads to a high-quality alignment result.

In the last phase, gaps between the anchors are further aligned by an existing progressive global multiple alignment tool to generate a detailed sequence alignment.

5.2 Phase 1: Find Multi-MUMs as Anchors

A MUM is defined a maximal unique match decomposition of two genomes in the program MUMmer [19]. It is a subsequence that occurs exactly once in both genomes,

and is not contained in any longer such sequence. The two character positions bounding an MUM must be mismatches [19]. Because of the assumption that input genome sequences are highly similar, a large number of MUMs are assured to be identified. The global alignment of two whole genome sequences can be built based on MUM alignment [19]. Our model aligns multiple whole genome sequences; therefore, we define the MUMs for multiple genomes as multi-MUMs. Aligning Multi-MUMs is the basic step for aligning multiple whole genomes.

Definition 5.2.1 A *multi-MUM* is a *maximal unique match* decomposition of multiple genomes. It occurs exactly only once in each sequence of a multiple sequence set and is not contained in any longer such sequence. The two characters bounding a multi-MUM must be mismatches in all the sequences.

In order to find the multi-MUMs from the input genome sequences, we use the enhanced suffix array algorithm [1], which is a suffix array enhanced with a table for longest common prefixes. We consider that the enhanced suffix array algorithm is better than the widely used suffix tree method because it requires much less space than the latter does. The enhanced suffix array method require not only less space but also much less time than other programs for genome analysis task [1].

Given k genome sequences: $S_1, S_2, ..., S_k$, a suffix array is built for the string $S = S_1\$_1 S_2\$_2 S_3\$_3 ... S_k\$_4$, which concatenates all the nucleotides of the genome sequences terminating with different separation symbols. This procedure takes $O(n)$ time, where n is the length of the string S [32].

Here are some basic notations and definitions for an enhanced suffix array:

Definition 5.2.2

sa: *sa* denotes a suffix array of S; $sa = sa[0...n-1]$.

sa[*i*]: *sa*[*i*] is the suffix array (*sa*) value in an entry of the suffix array.

S_i': S_i' denotes the *i*th suffix of S, which is $S[i...n-1]$.

lcp[*i*] : *lcp*[*i*] is the *longest common prefix* value of an entry *i*. *lcp*[*0*] equals 0; *lcp*[*i*] equals the length of the longest common prefix of *sa*[*i*] and *sa*[*i*−*1*] when $i > 0$.

ps[*i*]: *ps*[*i*] is the *proceeding symbol* of a suffix $S_{sa[i]}'$. So $ps[i] = S[_{sa[i]-1}]$. *ps*[*i*] will be undefined if *sa*[*i*] = 0.

so[*i*]: *so*[*i*] is the *sequence order (so)* value in an entry *i* of *S*; it is the order number of the sequence where the suffix $S_{sa[i]}'$ begins. If *sa*[*i*] begins from separation symbols, *so*[*i*] will be undefined.

lcp-Interval[*i*…*j*] : For a suffix array *sa*, *Interval*[*i*…*j*] is called a *lcp-Interval*[*i, j*] of *lcp-value l* if both *lcp*[*i*] and *lcp*[*j* + *1*] are smaller than *l*, and the smallest *lcp* value for entry $i + 1, ..., j$ is *l*. The *length* of the *lcp-Interval* is (*j* −*i*+*1*). The *lcp-string* of a

lcp-Interval[*i, j*] is the string $S[sa[i]...sa[i]+l-1]$.

Definition 5.2.3 A *lcp-Interval*[*i, j*] is a *MUM-Interval* if:

(1) the length of *lcp-Interval*[*i, j*] is *k*;

(2) *so*[*i*], . . . , *so*[*j*] are pairwise distinct;

(3) *ps*[*i*], . . . , *ps*[*j*] are not the same value.

From the definition of the *MUM-Interval*, the *longest common prefix string* of a *MUM-Interval*[*i, j*] occurs exactly only once in each input sequences and cannot be contained in a longer such sequence. So, this *lcp-string* is a Multi-MUM.

The brute force method to find Multi-MUMs is to scan the suffix array and check all the *Intervals* of length *k* to find *MUM-Intervals,* and then output the Multi-MUMs. We use a different method to speed up the process of finding all the *MUM-Intervals*. First, we scan the suffix array to locate all the *lcp-Intervals*. Next, for a *lcp-Interval* of length *k*, we check them with the requirements in Definition 5.1.3 to determine whether it is a *MUM-Interval* or not. Therefore, all the *MUM-Intervals* are found by scanning the suffix array only once, which indicates that multi-MUMs are found in linear time.

Here is an example:

Input sequences: S_1 = abeadc

S_2 = edbcaba

S_3 = cabed

S_4 = dcabea

S = abeadc$edbcaba*cabed#dcabea!

Suffix Array:

	sa		lcp		ps		so
0	11	aba*cabed#dcabea!	0		c		2
1	0	abeadc$edbcaba*cabed#dcabea!	2				1
2	23	abea!	4		c		4
3	16	abed# dcabea!	3		c		3
4	3	adc$edbcaba*cabed# dcabea!	1		e		1
5	13	a*cabed# dcabea!	1		b		2
6	26	a!	1		e		4
7	12	ba*cabed# dcabea!	0		a		2
8	9	bcaba*cabed# dcabea!	1		d		2
9	1	beadc$edbcaba*cabed#dcabea!	1		a		1
10	24	bea!	3		a		4
11	17	bed#dcabea!	2		a		3
12	10	caba*cabed#dcabea!	0		b		2
13	22	cabea!	3		d		4
14	15	cabed#dcabea!	4		*		3
15	5	c$edbcaba*cabed#dcabea!	1		d		1
16	8	dbcaba*cabed#dcabea!	0		e		2
17	21	dcabea!	1		#		4
18	4	dc$edbcaba*cabed#dcabea!	2		a		1
19	19	d#dcabea!	1		e		3
20	2	eadc$edbcaba*cabed#dcabea!	0		b		1
21	25	ea!	2		b		4
22	7	edbcaba*cabed#dcabea!	1		$		2
23	18	ed#dcabea!	2		b		3
24	6	$edbcaba*cabed#dcabea!	0		c		
25	14	*cabed#dcabea!	0		a		
26	20	#dcabea!	0		d		
27	27	!	0		a		

Figure 3: The enhanced suffix array for four sequences S_1, S_2, S_3, S_4.

From the enhanced suffix array for four sequences S_1, S_2, S_3, S_4, we find that *lcp-Interval*[*0,3*], *lcp-Interval*[*12,15*], *lcp-Interval*[*16,19*] and *lcp-Interval*[*20,23*] are *MUM-Intervals*. Therefore, we detect four multi-MUMs: (ab), (c), (d), (e).

After multi-MUMs have been identified, we label each of them with an integer from $\{1,2,3,...,m\}$, according to their positions in the first input sequence. Obviously, *m* is the number of the multi-MUMs and the integers are assigned as the indices of each of them. The indices are unique identifier of each multi-MUMs. In different input sequences, Multi-MUMs appear in different order according to their positions but the indices are always unique.

Therefore, each input sequence can be represented by the multi-MUMs and the gaps between them on a horizontal line. We use the corresponding index to represent each multi-MUM and ignore the gaps in this step. So, each input sequence can be transformed to a sequence consisting of the indices, which is defined as a *multi-MUM index sequence*.

Definition 5.2.4 A *multi-MUM index sequence* consists of the indexes of all the multi-MUMs. It is a permutation of $\{1,2,3,...,m\}$.

A *multi-MUM index sequence* for the input sequence S_i is denoted by I_i, every character of which is the index of the corresponding multi-MUM. In our example, the

four multi-MUMs are labeled as: 1=(ab), 2=(e), 3=(d), 4=(c). So the four input sequences can be transformed to four *multi-MUM index sequences*.

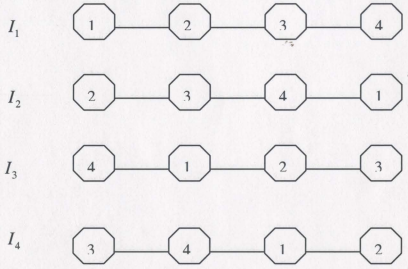

Figure 4: The *multi-MUM index sequences* of input sequences.

5.3 Phase 2: Find the *Multiple Heaviest Common Subsequence* as Anchor-chain to Align Anchors

The inputs of this phase are the *multi-MUM index sequences* that we got in phase 1. Based on the evolutionary relationship among the original genome sequences, certain numbers of *multi-MUM index sequences* are chosen to calculate the anchor-chain. A typical bioinformatics trade-off occurs here: large numbers of chosen sequences will result a more believable anchor-chain, but the procedure will consume more running time and space; small numbers of chosen sequences will use less running time and space but probably leads to a relative less accurate anchor-chain. This number choice depends on

the users' requirements. We weight the multi-MUMs in the chosen *multi-MUM index sequences* based on their length. Afterwards, our chaining algorithm is employed to find the *heaviest common subsequence* to be the anchor-chain. Different associated weights will result in different anchor-chains. The weights containing evolutionary information lead to a biologically more meaningful anchor-chain.

In the example here, we choose the first three sequences as the candidate sequences to compute the anchor-chain. "2" is weighed to the multi-MUM 1, "1" is weighed to the multi-MUM 2, "1" is weighed to the multi-MUM 3 and "1" is weighed to the multi-MUM 4. After running our program, we find the heaviest common subsequence of the first three sequences is the multi-MUM 1. Therefore, we choose the multi-MUM 1 to be the anchor-chain and assemble the three anchor-computing sequences according to the selected anchor-chain.

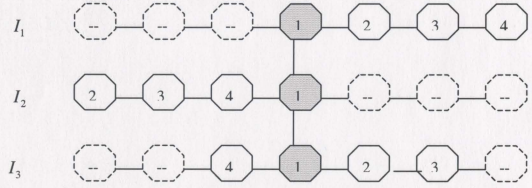

Figure 5: The alignment of the anchor-computing *multi-MUM index sequences*: Three anchor-computing *multi-MUM index sequences* I_1, I_2, I_3 are aligned according to the selected anchor-chain multi-MUM 1.

The *multi-MUM index sequences*, which are not selected to calculate the anchor-chain, are aligned according to this chain. That is: place the multi-MUMs, which have the same characters as the anchor-chain, to the anchor-chain column. Based on this procedure, an alignment for all the anchors is assembled.

In the example, I_4 is appended to the alignment based on the multi-MUM 1. Accordingly, an alignment of all the four *multi-MUM index sequences* is obtained.

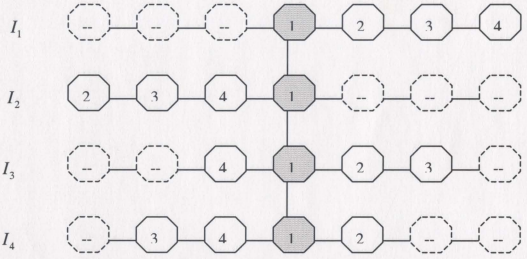

Figure 6: The alignment of all the *multi-MUM index sequences*: the fourth *multi-MUM index sequence* I_4 is appended to the alignment of the three anchor-computing *multi-MUM index sequences* according to the anchor-chain multi-MUM 1.

Therefore, the alignment of the anchors/Multi-MUMs is:

--	--	--	a	b	e	d	c
e	d	c	a	b	--	--	--
--	--	c	a	b	e	d	--
	d	c	a	b	e		

Figure 7: The alignment of all the anchors.

5.4 Phase 3: Close Gaps and Get Detailed Alignment

In this phase, the progressive global alignment method is used to align the gap regions between the anchors to generate detailed alignment.

The CLUSTAL W program [54] (Appendix A) can be used in this phase for detailed alignment. Because the target sequences are whole genomes, which are large-scale sequences, a threshold is set for the maximum length of the gaps to evaluate whether they should be align or not. If the length of a gap is out of the threshold, the gap will be ignored. For our example, the alignment result is:

--	--	--	--	a	b	e	a	d	c
e	d	b	c	a	b	--	a	--	--
--	--	--	c	a	b	e	--	d	--
--	d	--	c	a	b	e	a	--	--

Figure 8: The alignment result: the progressive global alignment method is used to align the characters in the gaps; together with the aligned anchors, the alignment result is obtained.

5.5 Time Complexity Analysis

In the first phase, a suffix array can be directly constructed in linear time [32]. The *lcp* array and the *ps* array can be obtained from the suffix array in linear time [33]. Hence, in the first phase, constructing a suffix array and computing all the multi-MUMs of input sequences requires linear time: $O(n)$, where n is the total length of all the input genome sequences.

With the finite automata algorithm [16], the anchor sequences can be transferred to *multi-MUM index sequence* in linear time.

In the second phase, the chaining algorithm for *k multi-MUM index sequences* works in $O(m^k)$ time, where m is the length of the *multi-MUM index sequence*. Then, it takes $O(k'm)$ time for the remaining k' sequences to be appended to the alignment.

The running time of the third phase depends on the threshold set by the user.

Chapter 6

Conclusions and Future work

6.1 Conclusions

We presented an anchor-based model for the global multiple alignment of whole genome sequences. Firstly, we introduced some background information on biology and bioinformatics. Then, we discussed several theories and techniques in computer science. Subsequently, we surveyed some existing anchor-based global alignment programs for two genome sequences or multiple genome sequences. We described the programs, some drawbacks, and their availability information. Next, we proposed a chaining algorithm. This algorithm is based on the dynamic programming technique and weighs each anchor by a proper weight that is based on evolutionary theory. Our algorithm finds the *heaviest common subsequence* among the weighted anchor sequences. Though we proved the *MHCS* problem is NP-complete, the algorithm works in polynomial time for limited sequence inputs. Our algorithm is presented in both the three sequences case and k

sequences case. We analyzed the running time of both cases. We implemented the case of three sequences by JAVA and verified that different associated weights lead to different results. Lastly, we described the whole procedure of our alignment method: first, we employed the enhanced suffix array method to find anchors; next, we used our chaining strategy to find the anchor-chain and to generate the alignment of the anchors; finally, we used the progressive multiple alignment tool CLUSTAL W to close the gaps. In the second phase of this procedure, in order to make up for the lack of methods for aligning distantly related genome sequences, we proposed a novel strategy: the genome sequences from close homologs are selected to assemble first, and then distantly related genome sequences are appended to the anchor alignment iteratively. This phase produces a more meaningful and accurate anchor alignment in terms of both computation and biology. It helps our model to assemble more genome sequences at any evolutionary distance. Combined with the exact suffix array approach in the first phase, this model leads to a high-quality alignment result.

6.2 Future work

Scientists usually evaluate sequence alignment programs by applying them to real-world data. For functional noncoding DNA, several benchmarking tools have been developed recently [46]. For protein alignment, some sets of benchmark sequences are available [55]

[35]. They have always been used as the standard to evaluate and compare the performance of multiple alignment programs. And for pairwise whole genome alignment, several benchmark data also have been compiled [5] [31].

As far as we know, there are still no generally accepted reference data to evaluate software programs for multiple alignment of genome sequences at any evolutionary distance. Because of this, we are lacking a standard for evaluating our method. We desire to test our method on a group of genome sequences in which several are from close homologs and others are from distant species.

Often, a model can be modified and improved. We will continue our research on improving and implementing this model to make it more efficient while retaining its accuracy.

Bibliography

[1] Abouelhoda M.I., Kurtz S., Ohlebusch E., "The enhanced suffix array and its application to genome analysis." Proceeding of the second workshop on algorithms in Bioinformatics (2002), Lecture Notes in Computer Science.

[2] Abouelhoda M.I., Ohlebusch E., "Multiple Genome Alignment: Chaining Algorithms Revisited", Proceeding of the Fourteenth Annual Symposium on Combinatorial Pattern Matching (2003): 1-16.

[3] Anson E.L., Myers E.W., "ReAligner: a program for refining DNA sequence multi-alignments." Journal of Computational Biology 4 (1997): 369-383.

[4] Apostolico A. and Guerra C. "The longest common subsequence problem revisited." Algorithmica 18(1) (1987): 1-11.

[5] Batzoglou S., Pachter L., Mesirov J.P., Berger B., Lander E.S. "Human and mouse gene structure: comparative analysis and application to exon prediction." Genome Research 10(7) (2000): 950-958.

[6] Bray N., Dubchak I., Pachter L. "AVID: A Global Alignment Program." Genome

Research 13 (2003): 97-102.

[7] Bray N., Pachter L. "MAVID: Constrained Ancestral Alignment of Multiple Sequences." Genome Research 14 (2004): 693-699.

[8] Brudno M., Chapman M., Gottgens B., Batzoglou S., Morgenstern B. "Fast and sensitive multiple alignment of large genomic sequences." BMC Bioinformatics (2003). Available from http://www.biomedcentral.com/1471-2105/4/66.

[9] Brudno M., *et al.* "LAGAN and Multi-LAGAN: Efficient Tools for Large-Scale Multiple Alignment of Genomic DNA." Genome Research 13 (2003): 721-731.

[10] Brudno M., Morgenstern B. "Fast and sensitive alignment of large genomic sequences." In Proceedings IEEE Computer Science Bioinformatics Conference (2002): 138-147.

[11] Buhler J. "Efficient large-scale sequence comparison by locality-sensitive hashing." Bioinformatics Vol.17 (2001): 419-428.

[12] Burkhard S., *et al.* "Fast lightweight suffix array construction and checking." CPM (2003), LNCS 2676: 55-69.

[13] Chain P., Kurtz S., Ohlebusch E., Slezak T., "An applications-focused review of comparative genomics tools: Capabilities, limitations and future challenges." Briefings in Bioinformatics Vol. 4 No.2 (2003): 105-123.

[14] Jeong-Hyeon Choi, *et. al*, "Multiple Genome Alignment by Clustering Pairwise

Matches", Proceeding of 2nd RECOMB Comparative Genomics Satellite Workshop (2004): 30-41.

[15] Stephen Cook "The P versus NP Problem." (2000) http://www.claymath.org

[16] Tomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms. The MIT Press, 2001.

[17] Couronne C., Poliakov A., Bray N., Ishkhanov T., Ryaboy D., Rubin E., Pachter L., Dubchak I., " Strategies and Tools for Whole-Genome Alignments." Genome Research 13 (2003): 73-80.

[18] Darling A., Mau B., Blattner F., Perna N. "Mauve: Multiple Alignment of Conserved Genomic Sequence With Rearrangements." Genome Research (2004).

[19] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White and Steven L. Salzberg. "Alignment of whole genomes." Nucleic Acids Research Vol.27, No.11 (1999): 2369-2376.

[20] Deogun J. S., Yang J., Ma F., "EMAGEN: An Efficient Approach to Multiple Whole Genome Alignment." APBC (2004).

[21] Downey R. G., Fellows M. R. "Fixed-Parameter Intractability (Extended Abstract)." In the Proceeding of the Seventh Annual Conference on Structure in Complexity Theory. (1992): 36-49.

[22] Downey R. G., Fellows M. R. Parameterized Complexity. Springer-Verlag, 1998.

[23] Durbin R., Eddy S., Krogh A. and Mitchison G, Biological Sequence Analysis. Cambridge University Press, Cambridge, 1998.

[24] Eppstein D., Galil Z., Giancarlo R., Italiano G.F. "Sparse dynamic programming I: Linear cost functions." JACM 39 (1992): 546-567.

[25] Michael R. Garey, David S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: W. H. Freeman and Company, 1979.

[26] Gupta S.K., Kececioglu J.D., Schaffer A.A. "Improving the Practical Time and Space Efficiency of the Shortest-Paths Approach to Sum-of-Pairs Multiple Sequence Alignment." J. Computational Biology 2(1995): 459-472.

[27] Gusfield D., Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.

[28] Des Higgins, Willie Taylor, Bioinformatics: Sequence, structure and databanks. The Oxford University Press, 2000.

[29] http://cse.stanford.edu/class/sophomore-college/projects-00/computers-and-the-hgp/bio.html

[30] Hohl M., Kurtz S., Ohlebusch E. "Efficient multiple genome alignment." Bioinformatics Vol.18 (2002): S312-S320.

[31] Niclas Jareborg, Ewan Birney, Richard Durbin. "Comparative Analysis of noncoding Regions of 77 Orthologous Mouse and Human Gene Pairs." Genome Research 9(1999):

815-824.

[32] Karkkainen J., Sander P. "Simple Linear Work Suffix array Construction." ICALP, LNCD 2719 (2003): 943-955.

[33] Kasai T., Lee G., Arimura H., Arikawa S., Park K. "Linear-Time Longest-Common-Prefix Computation in Suffix array and its Applications." CPM, LNCS 2089 (2001): 181-192.

[34] Kent W. J., Zahler A. M. "Conservation, regulation, synteny, and introns in large-scale *C. briggsae-C.elegans* genomic alignment." Genome Research Vol. 10(2000): 1115-1125.

[35] Lassmann T., Sonnhammer E.L.L. "Quality assessment of multiple alignment programs." FEBS Letters 529 (2002): 126-130.

[36] Lipman D.J., Altschul S.F., Kececioglu J.D. "A Tool for Multiple Sequence Alignment." Proc. Natl. Acad. Sci. USA 86 (1989): 4412-4415.

[37] David Maier "The Complexity of Some Problems on Subsequences and Supersequences." Journal of the Association for Computing Machinery (1978).

[38] Manber G. Myers G. "Suffix arrays: A new method for on-line string searches." SIAM J. Comput. 22(5) (1993): 935-948.

[39] Ernst Mayr What evolution is. New York: Basic Books 2001.

[40] Morgenstern B., Dress A., Werner T. "Multiple DNA and protein sequence alignment

based on segment-to-segment comparison." Proc. Natl Acad. Sci. USA, Vol. 93 (1996): 12098-12103.

[41] David W. Mount, Bioinformatics – Sequence and Genome Analysis. Cold Spring Harbor Laboratory Press, 2001.

[42] National Human Genome Research Institute: http://www.genome.gov/.

[43] Parra G., Agarwal P., Abril J. F., Wiehe T., Fickett J. W., Guigo R. "Comparative Gene Prediction in Human and Mouse." Genome Research 13 (2003): 108-117.

[44] M. Paterson and V. Dancik. "Longest common subsequences." In Mathematical Foundations of Computer Science, 19th International Symposium (MFCS), Vol.841 of LNCS (1994): 127.

[45] Pietrzak K. "On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems." Journal of Computer and System Sciences 67 (2003): 757-771.

[46] Daniel A. Pollard, Casey M. Bergman, Jens Stoye, Susan E. Celniker, Michael B. Eisen. "Benchmarking tools for the alignment of functional noncoding DNA." BMC Bioinformatics 5 (2004): 6.

[47] Primrose S. B., Twyman R. M. Principles of Genome Analysis and Genomics. (Third edition) Blackwell Publishing, 2003.

[48] RepeatMasker: AFA. Smit, P. Green. Available from

http://repeatmasker.genome.washington.edu/

[49] Rick C. "A new flexible algorithm for the longest common subsequence problem." In Proceedings of the 5th Combinatorial Pattern Matching, volume 937 of LNCS (1995): 340-351.

[50] Schwartz S. *et al.* "MultiPipMaker and supporting tools: alignments and analysis of multiple genomic DNA sequences." Nucleic Acid Research Vol.31, No.13 (2003): 3518-3524.

[51] Schwartz S., Kent W. J., Smit A., Zhang Z., Baertsch R., Hardison R. C., Haussler D., Miller W., "Human-mouse alignments with Blastz." Genome Research 13 (2003): 103-105.

[52] Schwartz S., Zhang Z., Fraszer K. et al. "PipMaker – a web server for aligning two genomic DNA sequences." Genome Research Vol. 10(2000): 577-586.

[53] Setubal, J. and Meidanis, J., Introduction to computational molecular biology. Boston, MA: PWS Publishing, 1997.

[54] Julie D. Thompson, Desmond G. Higgins, Toby J. Gibson. "CLUSTAL 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 (1994): 4673-4680.

[55] Thompson JD, Plewniak F, Poch O "BAliBASE: A benchmark alignment database

for the evaluation of multiple sequence alignment programs." Bioinformatics 15 (1999): 87-88.

[56] TIGR Genomes MUMmer page.

Available from http://www.tigr.org/software/mummer/

[57] Vincens P., Badel-Chagnon A., André C. and Hazout S. "D-ASSIRC: distributed program for finding sequence similarities in genomes." Bioinformatics 18 (3) (2001): 246-251.

[58] Vincens P., Buffat L., André C., Chevrolat J.P., Boisvieux J.F. and Hazout S. "A strategy for finding regions of similarity in complete genome sequences." Bioinformatics 14 (8) (1998): 715-725.

[59] Wang L., Jiang T. "On the complexity of Multiple sequence alignment." Journal of Computational Biology, Vol. 1(1994): 337-348.

Appendix A:

CLUSTAL W: a tool for progressive global multiple alignment

In the last step of our method, we use progressive global multiple alignment tool CLUSTAL W [54] to deal with the gaps between anchors. Here, we briefly describe CLUSTAL W.

The progressive alignment approach has been proposed to many tools. CLUSTAL W is a very classic and successful one. In the first step of this program, dynamic programming or heuristic algorithms compute the pairwise alignment cost. Dynamic programming gives more accurate scores, however, heuristic methods are faster. In CLUSTAL W, it allows to choose either dynamic programming or a heuristic method. In the second step, under a given distance matrix between sequences, CLUSTAL W builds a guide tree using the Neighbor-Joining algorithm. The third step is to consist of two alignments. CLUSTAL W uses profile alignment with position-specific gap penalties.

CLUSTAL W is a general-purpose progressive global alignment program for

biological sequences. It works well and is very commonly used.

CLUSTAL W is available at: http://www.ebi.ac.uk/clustalw/

Appendix B

Source Code

```
/* This code follows the idea in Chapter 4 and can find the heaviest common subsequence
in three sequences. */
import java.io.IOException;
import java.util.ArrayList;
import java.util.StringTokenizer;

/* Entrance*/
public class Entrance {
    public static void main(String[] args) throws IOException {

        int[] weightArray = null;
        ArrayList sequences = null;
        boolean weightsNeeded = true;
        boolean sequencesNeeded = true;
        while (true) {
            if (weightsNeeded) {
                byte[] inputs = new byte[0];
                inputs = new byte[1024];
                System.out.print("Please enter the WEIGHTs for A,B,C,D(seperated
by comma or space, optional):");
                System.in.read(inputs);
                String strWeights = new String(inputs);
                strWeights = strWeights.substring(0, strWeights.indexOf('\n'));

                ArrayList weightlist = new ArrayList();
```

```
                if (!"".equals(strWeights.trim())) {
                    for (StringTokenizer stringTokenizer = new
StringTokenizer(strWeights, ", "); stringTokenizer.hasMoreTokens();) {
                        String s = stringTokenizer.nextToken().trim();
                        try {
                            weightlist.add(new Integer(Integer.parseInt(s)));
                        } catch (NumberFormatException e) {
                        }
                    }
                }

                weightArray = new int[]{1, 1, 1, 1};
                int loop = weightlist.size() < 4 ? weightlist.size() : 4;
                if (!weightlist.isEmpty()) {
                    for (int i = 0; i < loop; i++) {
                        weightArray[i] = ((Integer) weightlist.get(i)).intValue();
                    }
                }
                System.out.println(new StringBuffer().append("The strWeights are: ")
                        .append(weightArray[0])
                        .append(", ")
                        .append(weightArray[1])
                        .append(", ")
                        .append(weightArray[2])
                        .append(", ")
                        .append(weightArray[3]).toString());
            } else {
                weightsNeeded = true;
            }

            if (sequencesNeeded) {
                char[] xyz = new char[]{'X', 'Y', 'Z'};
                sequences = new ArrayList();
                for (int i = 0; i < xyz.length; i++) {
                    System.out.print("Please enter sequence " + xyz[i] + ":");
                    byte[] inputs = new byte[1024];
                    System.in.read(inputs);
                    String sequence = new String(inputs);
                    sequence = sequence.substring(0, sequence.indexOf('\n'));
```

```
                    sequence = sequence.toUpperCase();
                    System.out.println("sequence = " + sequence);
                    sequences.add(sequence);
                }
            } else {
                sequencesNeeded = true;
            }

            Weighted weighted = new Weighted(weightArray[0], weightArray[1],
weightArray[2], weightArray[3]);
            int res = weighted.func((String) sequences.get(0), (String)
sequences.get(1), (String) sequences.get(2));
            if (res == 0) {
                System.out.println("\nThe result is: " + weighted.getResult());
                System.out.print("Another testing sequences?(Y/N)");
                byte[] inputs = new byte[1024];
                System.in.read(inputs);
                String answer = new String(inputs);
                answer = answer.substring(0, answer.indexOf('\n')).trim();
                if ("Y".equalsIgnoreCase(answer)) {
                    weightsNeeded = false;
                } else {
                    System.exit(0);
                }
            } else {
                System.out.println("\nERROR: Invlid strWeights populated. Please
select another weights for A,B,C,D.");
                sequencesNeeded = false;
            }
        }
    }
}

/*Weighted*/
public class Weighted {
    private int WA = 1;
    private int WB = 1;
    private int WC = 1;
    private int WD = 1;
```

```
private StringBuffer result = new StringBuffer();

public Weighted(int WA, int WB, int WC, int WD) {
    this.WA = WA;
    this.WB = WB;
    this.WC = WC;
    this.WD = WD;
}

public int func(String stringX, String stringY, String stringZ) {
    stringX = stringX.toUpperCase().trim();
    stringY = stringY.toUpperCase().trim();
    stringZ = stringZ.toUpperCase().trim();
    int l = stringX.length();
    int m = stringY.length();
    int n = stringZ.length();
    int[][][] matrixC = new int[l][m][n];
    int[][][] matrixD = new int[l][m][n];
    for (int i = 0; i < matrixC.length; i++) {
        int[][] ints = matrixC[i];
        for (int j = 0; j < ints.length; j++) {
            int[] anInt = ints[j];
            for (int k = 0; k < anInt.length; k++) {
                char xi = stringX.charAt(i);
                char yj = stringY.charAt(j);
                char zk = stringZ.charAt(k);
                boolean can = i * j * k != 0;
                if (xi == yj && yj == zk) {
                    int tmp = 0;
                    if (can) {
                        tmp = matrixC[i - 1][j - 1][k - 1];
                    }
                    matrixD[i][j][k] = 1;
                    switch (xi) {
                        case 'A':
                            matrixC[i][j][k] = tmp + WA;
                            break;
                        case 'B':
                            matrixC[i][j][k] = tmp + WB;
```

```
                        break;
                    case 'C':
                        matrixC[i][j][k] = tmp + WC;
                        break;
                    case 'D':
                        matrixC[i][j][k] = tmp + WD;
                        break;
                    default:
                }
            } else if (can && matrixC[i - 1][j][k] == max(matrixC[i - 1][j][k], matrixC[i][j - 1][k], matrixC[i][j][k - 1])) {
                matrixC[i][j][k] = matrixC[i - 1][j][k];
                matrixD[i][j][k] = 0;
            } else if (can && matrixC[i][j - 1][k] == max(matrixC[i - 1][j][k], matrixC[i][j - 1][k], matrixC[i][j][k - 1])) {
                matrixC[i][j][k] = matrixC[i][j - 1][k];
                matrixD[i][j][k] = 0;
            } else if (can) {
                matrixC[i][j][k] = matrixC[i][j][k - 1];
                matrixD[i][j][k] = 0;
            }
        }
    }
}

try {
    func2(matrixD, matrixC, stringX, l - 1, m - 1, n - 1);
} catch (Exception e) {
    return -1;
}
return 0;
}

private int max(int x, int y, int z) {
    int m = x > y ? x : y;
    return m > z ? m : z;
}

private void func2(int[][][] matrixD, int[][][] matrixC, String X, int i, int j, int k)
```

```
throws Exception {
        if (i < 0 || j < 0 || k < 0) {
            return;
        }

        try {
            if (matrixD[i][j][k] == 1) {
                func2(matrixD, matrixC, X, i - 1, j - 1, k - 1);
                result.append(X.charAt(i));
            } else if (matrixC[i - 1][j][k] == max(matrixC[i - 1][j][k], matrixC[i][j -
1][k], matrixC[i][j][k - 1])) {
                func2(matrixD, matrixC, X, i - 1, j, k);
            } else if (matrixC[i][j - 1][k] == max(matrixC[i - 1][j][k], matrixC[i][j -
1][k], matrixC[i][j][k - 1])) {
                func2(matrixD, matrixC, X, i, j - 1, k);
            } else {
                func2(matrixD, matrixC, X, i, j, k - 1);
            }
        } catch (Exception e) {
            throw e;
        }
    }

    public String getResult() {
        return result.toString();
    }
}
```