A HEURISTIC ALGORITHM FOR BLOCKED MULTIPLE SEQUENCE ALIGNMENT

A HEURISTIC ALGORITHM FOR BLOCKED MULTIPLE SEQUENCE ALIGNMENT

By

PENG ZHAO, BSc in Computer Science

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2000) MCMASTER UNIVERSITY

(Computer Science) Hamilton, Ontario

TITLE: A Heuristic Algorithm for Blocked Multiple Sequence Alignment

AUTHOR: Peng Zhao

BSc (Beijing University of Aero. & Astro., China)

SUPERVISOR: Dr. Tao Jiang

NUMBER OF PAGES: x, 95

Abstract

Blocked multiple sequence alignment refers to the construction of multiple alignment by first aligning conserved regions into what we call "blocks" and then aligning the regions between successive blocks to form a final alignment. Instead of starting from low order pairwise alignments we propose a new way to form blocks by searching for closely related regions in all input sequences, allowing internal spaces in blocks as well as some degree of mismatch. We address the problem of semi-conserved patterns (patterns that do not appear in all input sequences) by introducing into the process two similarity thresholds that are adjusted dynamically according to the input. A method to control the number of blocks is also presented to deal with the situation when input sequences have so many similar regions that it becomes impractical to form blocks by trying every combination. BMA is an implementation of this approach, and our experimental results indicate that this approach is efficient, particularly on large numbers of long sequences with well-conserved regions.

Acknowledgments

I would like to thank Tao Jiang, my supervisor, for granting the opportunity, and for his invaluable supervision in guiding this work. His encouragement and support has been invaluable for me. I would also like to thank Sanzheng Qiao, Bill Smyth, and Stavros kolliopoulos for their careful reviewing of this work. Furthermore, I would like to extend my appreciation to Toby Gibson, Derek Lipiec, Julie Thompson, and Xian Zhang for the help they have given me along the way. This work is based on many excellent work done by other people. In particular, I would like to thank Mark Boguski, Walter Fitch, Ross Hardison, Hugo Martinez, Marcella McClure, Webb Miller, Balaji Raghavachari, Eric Sobel, Taha Vasi, and Zheng Zhang. Finally, I would like to give my warmest thanks to my wife Xiuhong Wu for her love, understanding, and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computational Biology, as the computational basis for molecular sequence analysis, has emerged and flourished in recent years. Many universities have started courses on computational biology, and more and more people are getting involved in it. In 1990, the U.S. Department of Energy and the National Institutes of Health started the 13-year Human Genome Project. The goals of the project are to identify all the genes in human DNA and to develop tools for data analysis.

Computational biology is emerging as an important field for the computer science community. It all started when people found that DNA and protein are the building blocks of life. At an abstract level, molecular sequences are long strings of characters over an alphabet of size four for DNA and twenty for protein. It is the sequence of these characters that decides the structure

and function of all living things. The last ten years has seen an explosive growth in the quantity of biological sequence information. The availability of this data has catalyzed a revolution in biological and medical research.

A fact that explains the importance of molecular sequence data and sequence comparison is that "high sequence similarity usually implies significant functional or structural similarity" [17, page 212]. Indeed, searching for sequence similarity by comparing many related sequences simultaneously has been used intensively in inferring the biological function of a sequence. Several computational techniques were developed to approach this problem. A scientist will turn to complex and expensive biological experiments after such theoretical hypothesis for further test.

One of the most important techniques involved in sequence comparison is constructing sequence alignment. A (global) *multiple alignment* of $k > 2$ sequences $S = \{S_1, S_2, \ldots, S_k\}$ is obtained by inserting chosen spaces into or at either end of each of the k sequences so that the resulting sequences have the same length. The sequences are aligned in such a way that each character or space in each sequence is contained in a unique column. An alignment displays a relationship among the sequences. People use different ways to construct sequence alignments to discover similarities between the sequences. An example is shown in Figure 1.1.

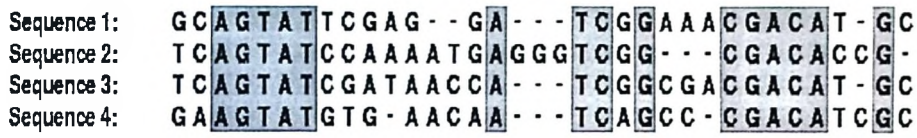

Figure 1.1: An example of multiple sequence alignment.

Multiple sequence alignment has been essentially utilized to identify biologically important conserved patterns among a set of related sequences, and to infer the evolutionary history of some species from the associated sequences. However, even extremely simple multiple sequence alignment problems are NP-complete, as shown by Wang and Jiang [37]. There have been many multiple alignment programs devised [3, 4, 13, 23, 28, 33, 35]; however, these existing methods are far from being satisfactory. Theoretical investigations can further progress towards practical solutions for precisely formulated subproblems.

We designed a heuristic algorithm for multiple sequence alignment. The intuition behind our method is very straightforward. Given $k > 2$ sequences, $S = \{S_1, S_2, \ldots, S_k\}$, a *region* (or *segment*) for each sequence S_i is obtained by selecting a substring from S_i. The substrings are then aligned. We call such alignment a *block* (or *motif*, we will use the terms block and motif interchangeably hereafter). The alignment of the regions is also referred to as *local multiple alignment*, as opposed to the global multiple alignment which

produces end-to-end alignment of all the sequences. The highlighted parts in figure 1.1 show the blocks we have found in the alignment. We observe that if the objective of alignment is to identify conserved regions, then instead of constructing an alignment first, we might start off by finding the regions directly. We can build a multiple alignment later on using the information of the regions.

Imagine that the blocks are embedded in a background of sequences, by aligning the gaps between successive blocks, a multiple alignment is obtained. In this thesis, we decompose the problem into three steps: (1) find similar substrings among input sequences, (2) build blocks using such similar substrings, which is a subset of all the possible combinations of the substrings, and then (3) chain the blocks to construct an alignment. In this process, a couple of technical issues may arise. In particular, what are the criteria to be used to decide similarity? How to construct blocks? Which blocks will be used to construct the final alignment, and how to construct the alignment? These issues will be addressed later in this thesis.

There is some previous work taking this approach [31, 33, 40, 41]; but none of them take into account of the problem of "repeats", which arises when there are so many copies of similar substrings from each sequence that generating all the combinations becomes impractical. Furthermore, they all

assume that there are no internal spaces inside a block, which is generally not the case, especially for DNA sequences. Due to the fact that "the extent of permissive mutations in structurally or functionally conserved molecules may be such that comparing two strings at a time reveals little of the critically conserved patterns or of the critical amino acids" [17, page 335], we try to avoid using pairwise alignments to construct blocks, as opposed to the approach taken by some previous work [5, 26, 27]. In our algorithm, we construct blocks using direct multiple alignment of similar substrings, which allows internal spaces and some degree of mismatch in blocks. Since there are cases when a block might not be "supported" by all the k sequences, we introduce *partial blocks* into the process. These issues will be covered in Chapter 3.

We have implemented the algorithm using C++ on a Unix platform. The program constructs a multiple sequence alignment from input sequences and reports the blocks that appear in the alignment. It also allows the user to specify some run-time parameters, such as length of a region, the maximum number of spaces allowed in a region, and gap penalties used in the alignment. Our experimental results indicate that this approach is efficient, particularly on a large number of long sequences with well-conserved regions. However, in some extreme cases, the program may run out of memory and thus fail to

produce an alignment. Some of them can be avoided by choosing different run-time parameters. Such extreme cases will be discussed later.

An outline of this thesis is as follows. First we will give a brief review in Chapter 2 on multiple sequence alignment and blocked sequence alignment. Then we describe the algorithm in Chapter 3, and implementation issues in Chapter 4. Some experimental results are reported in Chapter 5. Chapter 6 contains the conclusion and future work.

Chapter 2

The Blocked Alignment Problem

In this chapter, we first give a brief review on multiple sequence alignment, then describe in detail some previous work on blocked sequence alignment.

2.1 Multiple Sequence Alignment

During the process of evolution, a molecular sequence is duplicated and modified. Such modification includes insertion, deletion, and replacement. The result is that parts of the related sequences might be exactly or almost the same, while other parts might be dramatically different due to the modifications. The difference makes the conserved similarities even more significant, which in turn makes sequence comparison a very powerful tool in biology.

Although sequence similarity usually implies significant structural or functional similarity, the converse is not true. This is caused by the fact that many positions in a molecular sequence are noncritical, and can be mutated without destructive effect on the sequence. The consequence is that the best alignment between two related molecular sequences might be statistically indistinguishable from the best alignment of two random sequences. Multiple sequence alignment is a natural response to the problem. Conserved sequence features that cannot be detected by pairwise alignment might be revealed clearly by multiple sequence alignment. Multiple alignment is also used in the deduction of evolutionary history from related sequences.

Here we are trying to give a brief review on multiple sequence alignment methods that are related to our method. For a comprehensive survey, see McClure et al. [25] and Chan et al. [8]. Broadly speaking, each sequence alignment method can be put into one of the three categories: exhaustive methods, bounded-error approximation methods, and heuristic methods.

For a better understanding of the methods, let's first define a general optimization model for multiple sequence alignment. Given an objective function S, the *score* of a multiple alignment $\mathcal{A}$ is $S(\mathcal{A})$. To date, there is no objective function that has been as well accepted for multiple alignment as edit distance has been for two-string alignment. However, there are three

types of objective functions widely used in practise: *sum-of-pairs functions, consensus functions*, and *tree functions* [17, page 343]. An *optimal alignment* is the one which optimizes (either maximizes or minimizes) the objective function.

2.1.1 Exhaustive Methods

The exhaustive methods of multiple sequence alignment guarantee an optimal alignment. Many of them use dynamic programming. This approach refers to the simultaneous comparison of n sequences using a n-dimensional dynamic programming matrix. There are many papers published to improve this method by using different scoring schemes and improving the time complexity, especially for the case of three sequences [7, 14, 15, 23, 29]. However, since dynamic programming requires computation time proportional to N^n, where N is the average length of the n sequences, it becomes inefficient when applied to more than two sequences.

2.1.2 Approximation Methods

Approximation methods can efficiently find an alignment with a score that is guaranteed to be within a factor of $\rho(I)$ of the optimal alignment score for any input of size I, where $\rho(I)$ is a function of I. Some methods have been described [16, 19]. Although bounded-error approximation are not always the

most effective methods, they can provide guaranteed bounds in practice and, if combined with other methods, lead to very effective solutions. Furthermore, bounded-error methods with fixed error bound, which is independent of I, are often improvable to methods that have a provable trade-off between bounded-error and running time. Such methods are called *polynomial time approximation schemes*, some have been developed by Jiang et al. [19] and Bafna et al. [2].

2.1.3 Heuristic Methods

The heuristic methods try to find good alignments that are not necessarily optimal within reasonable time. There are many good methods available, for example, CLUSTAL W [35], DIALIGN [28], MSA [23], and DFALIGN [13]. Many of the previous work, including CLUSTAL W, MSA, and DFALIGN, begin by comparing all sequences in a pairwise fashion. Then they cluster the sequences into sub-alignments by using similarity measure (MSA) or a phylogenetic tree (CLUSTAL W and DFALIGN). Then the sub-alignments are aligned to each other progressively to produce a multiple alignment. This process is referred to as iterative pairwise alignment. The final alignment is dependent on the order in which the sequences are processed. When the relationships between the sequences are well understood, it works very well.

The second major approach commonly used in multiple alignment is called *repeated-motif method* [17]. This approach relies on first finding a substring or a small similar subsequence that is common to many of the input sequences. Once a first substring/subsequence of this type is found, the sequences containing it are shifted so that the occurrences of the pattern are aligned with each other. Then the problem of completing the multiple alignment of those sequences is divided into two smaller subproblems, one for substrings on each side of the aligned part, as shown by Posfai et al. [32]. There are some variants of this general approach, and we discuss some of them in section 2.3.

There are two other methods that are very different from the approaches mentioned above. One is Gibbs sampling method [22], and the other is hidden Markov models (HMM) method [21]. Both of them take a statistical approach to sequence analysis.

2.2 Local Sequence Alignment

Before we discuss *blocked sequence alignment*, let's take a quick look at local sequence alignment, which is basically a global alignment of chosen substrings/regions from each sequence.

Global alignment is often meaningful when the sequences are from the

same sequence family, in other words, when they are closely related. However, for sequences from different sequence families, local alignment is often the most appropriate type of alignment. The reason is that those sequences might only have a few similar regions, which are embedded in an overall background of dissimilarity. A global alignment would very likely fail to align those regions with each other.

There are many publications on identifying local sequence similarities. Some of them are based on string alignment or enumeration, while others use statistical methods. Typically, these algorithms are able to discover patterns of type ranging from simple strings to general regular expressions. For a comprehensive survey of several of these algorithms, see Brazma et al. [6]. The general form of pattern discovery has been shown to be an NP-hard task [24].

If combined with repeated-motif approach, local multiple alignment can be used effectively to construct global alignment, a method that we call "blocked multiple sequence alignment".

2.3 Blocked Multiple Sequence Alignment

We define a block from $n > 2$ sequences in Chapter 1: an alignment of n regions, one region from each sequence. Because we are interested in conserved sequence features, we only consider regions that are closely related in the construction of blocks. The idea of blocked multiple sequence alignment is we start by constructing blocks, then use these blocks to obtain a global alignment of the sequences. There have been many publications that turn this general outline of approach into concrete methods. Now, let's look at some in more detail, and based on these previous work, we will present our algorithm in the next chapter.

Method I: Sobel and Martinez (1986)

The strategy of Sobel and Martinez [33] is based on first finding common segments above a specified length and then "piecing" these together to maximize an alignment scoring function. Each group of common segments are exact repeats of a string that is common to all sequences. It is easy to see that there is at least one copy of matching segments in every sequence. A block is obtained by taking one copy from each sequence and then aligning them together. The best set of blocks making up the alignment is found by the classic longest path algorithm for directed acyclic graph.

This is the first practical method that we are aware of taking blocked approach to do multiple alignment, and much later work is based on this one. However, there are a couple of issues that require discussion. It is frequently of interest to align segments with some degree of mismatch, insertion, and deletion, especially for DNA sequences. Furthermore, there may be no segment common to all the sequences. Another problem arises when we have many repeats of a match in many sequences, such that generating all the blocks becomes computationally unrealistic, which is known as *combinatorial explosion* in literature. Sobel and Martinez attack this problem by putting restrictions on the length of common segments to help avoid the "explosion".

Method II: Waterman (1989); Waterman and Jones (1990)

Waterman [38] suggests constructing an alignment by consensus words. A consensus word is a k-letter string that occur in at least a preset percentage of sequences. Therefore there are 4^k such words in DNA and 20^k in proteins. Mismatch is allowed, and the degree is chosen by the user. With sequences arranged in rows, the method defines a *window* as the block from column j to column $j + W - 1$, where W is the window width. It proceeds by finding the frequency of occurrence for each word in each window, and calculates a score

$s(w_i)$ for word w_i based on its frequency. The goal of an optimal alignment is to find words w_i which satisfy

$$\max\{\sum_j s(w_{i_j}) : w_{i_1} < w_{i_2} < \ldots\},$$

where $w_{i_j} < w_{i_k}$ if the occurrences of w_{i_j} are to the left of the occurrences of w_{i_k} in all the sequences. The alignment is then obtained by aligning the corresponding letters of the "winning" words.

However, due to the large number of words, it is not possible to accomplish such optimization within reasonable time. The method proposes two algorithms to approximate the solution. The time complexity of this method is $O(NW^2nB)$ where N is the sequence length, n is the number of sequences, and B is a function of 4^k for DNA or 20^k for proteins. The statistical significance of a word depends on the window width, and it is not easy to decide whether $w_i < w_j$ when w_i and w_j appear in varying multiplicity and order within the given sequences.

Method III: Boguski et al. (1992); Miller et al. (1994); Miller (1993)

Miller et al. [27] propose constructing aligned sequence blocks from a set of pairwise alignments. From each pairwise alignment, a list of pairs of positions is obtained, one position from each sequence. The positions in each pair are

considered to be related. Given a family F of pairwise alignments among sequences $s_1, s_2, \ldots, s_n$ and a proposed column of positions, say position p_1 in s_1, p_2 in s_2, ..., p_n in s_n, they propose a method to decide whether to accept the proposed column into the alignment based on the evidence from F. The strongest requirement is that each pair of positions in the column is related according to the pairwise alignment between the corresponding two sequences [5], but has proved to be overly restrictive in many cases. The method of Miller et al. [27] accepts columns that support two independent chains of deductions confirming the relatedness of every pair of positions in the column. They call such columns *biconnected.*

A block is constructed by concatenating a run of consecutive biconnected columns, therefore there is no internal space inside a block. The result relies heavily on the quality of pairwise alignments. Also, it would save time and space by working directly with segments, instead of decomposing them into individual positions. Such an algorithm was developed for the maximum-connectivity case [26].

Method IV: Zhang et al. (1996)

Based on the previous work, Zhang et al. [40] propose to do local multiple alignment via subgraph enumeration. This method is also based on pairwise

alignments. First it constructs a single graph that subsumes all of the given pairwise alignments; after adding some special edges in the graph, it proceeds to enumerate all maximal cliques in that graph. In this way, it is able to construct a special type of block that does not necessarily contain a segment from every sequence, and a sequence might contribute more than one segment. After two more processes, *chaining* and *flattening* [40], a multiple alignment is obtained.

Since the number of maximal cliques can in theory be exponential in the number of vertices and edges, they apply the algorithm of Tsukiyama et al. [36] to achieve "polynomial delay" in the enumeration process, which does not really solve the problem. Another problem is that the process of chaining such incomplete blocks is NP-complete.

Method V: Parida et al. (1999)

Parida et al. [31] also use the idea of incomplete blocks, but attack the NP-completeness in a different way. In their method, they reduce the problem of selecting incomplete blocks to the well-known set cover problem, and use the result of Johnson [20] to achieve an approximation factor of $1 + ln|X|$, where $|X|$ is the size of set X. The objective function is to seek an alignment that minimizes the number of characters that do not match in at least K

sequences, where K is a preset constant. It does not allow internal spaces and there is no gap penalty in the final alignment.

Chapter 3

The Algorithm

This chapter gives a detailed description of our algorithm. First, we outline the approach by listing the issues we try to resolve in our algorithm. Then, after giving a schematic view of the algorithm, we elaborate on each step involved in the process of constructing sequence alignment. Some of the implementation issues are discussed in the next chapter.

3.1 Attacking The Problem

After inspecting some the previous work, we have designed a heuristic algorithm, trying to address the following issues:

- constructing the blocks using direct multiple alignment of similar substrings, instead of from low order pairwise alignments;

- solving the problem of finding related regions to form a block by searching directly in all the input sequences;
- allowing internal spaces and some degree of mismatch in a block;
- taking into account incomplete blocks, i.e., alignment of regions that are not common to all input sequences;
- when there are too many blocks, try to eliminate some unlikely scenarios to avoid combinatorial explosion;
- penalizing gaps in the final alignment.

The main idea of the algorithm is to first align similar regions into blocks, then align the regions between successive blocks to construct the final alignment.

3.2 A Schematic View of the Algorithm

Figure 3.1 shows a diagram of data flow in the algorithm. After obtaining input sequences, the algorithm proceeds sequentially through three steps, obtaining input only from the output of the previous step. Therefore, the alignment problem consists of three independent subproblems:

grouping Collect similar regions/substrings among input sequences.

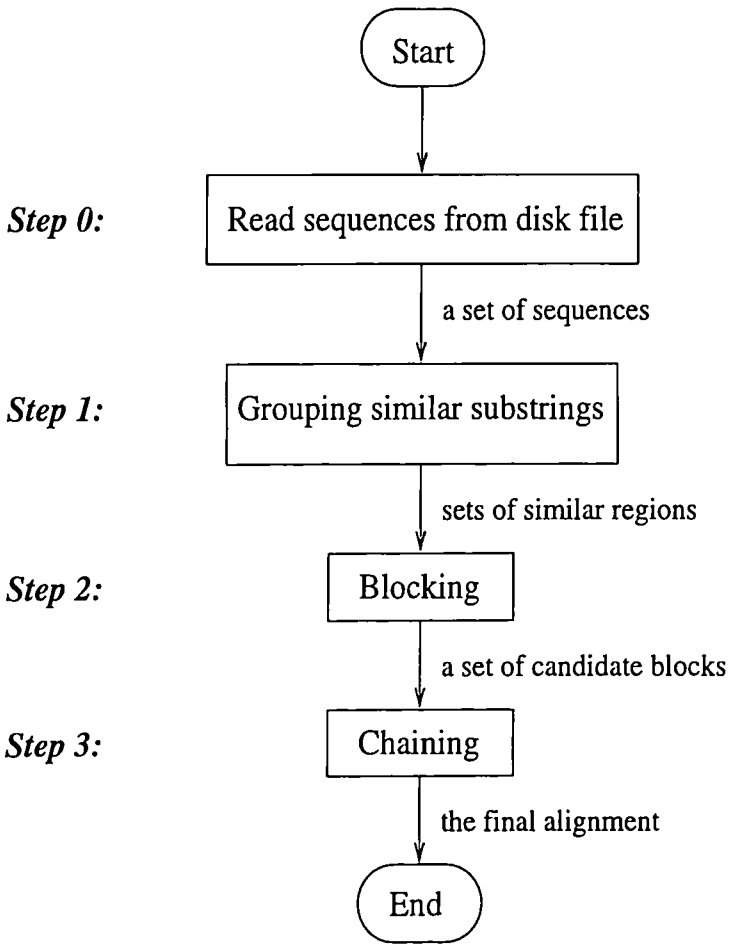

Figure 3.1: A schematic view of the algorithm

blocking Build blocks using those similar substrings and try to eliminate some unlikely combinations.

chaining Select blocks that will be used in the final alignment and chain them up to obtain the alignment.

In the following sections, we will discuss in detail about each of them.

3.3 Grouping Similar Substrings: Step 1

3.3.1 String Similarity

Edit distance is one of the ways that the relatedness of two strings has been formalized. An alternate, and often preferred, way of formalizing the relatedness of two strings is to measure their similarity rather than their distance. Let Σ be the alphabet used for sequences S_1 and S_2, let Σ' be Σ with the added character "␣" denoting a space. Then for any two characters $x, y \in \Sigma'$, $s(x, y)$ denotes the score obtained by aligning character x against character y. For a given alignment $\mathcal{A}$ of S_1 and S_2, let S_1' and S_2' denote the strings after the chosen insertion of spaces, and let l denote the length of the alignment. The *score* of the alignment is defined as

$$\sum_{i=1}^{l} s(S_1'(i), S_2'(i))$$

For example, let $\Sigma = \{a, c, g, t\}$ and let the pairwise scores be defined in the following matrix:

s	a	c	g	t	⊔
a	1	0	0	0	0
c		1	0	0	0
g			1	0	0
t				1	0
⊔					1

Then the alignment

g	a	g	⊔	t	c	t
g	a	c	c	t	c	⊔

has a score of $1 + 1 + 0 + 0 + 1 + 1 + 0 = 4$.

Given a pairwise scoring matrix over the alphabet Σ', the *similarity* of two strings S_1 and S_2 is defined as the score of the alignment $\mathcal{A}$ of S_1 and S_2 that maximizes the score. For strings "gagtct" and "gacctc", an optimal alignment can be obtained by dynamic programming [30]. The alignment is shown above. Therefore, the similarity of the strings is 4.

String similarity clearly depends on the specific scoring matrix involved. Numerous scoring matrices have been suggested for proteins and DNA, and no single scheme is right for all applications [17, page 226]. In our algorithm, we adopt the scoring matrix used by McClure et al. [25] for protein sequences, and the simple 0/1 matrix for DNA sequences. The user can change the score matrix at run time by feeding extra information into the program.

3.3.2 A Straightforward Algorithm to Identify Similar Regions

In our algorithm, two regions are considered to be similar if the similarity score is above a certain threshold. The objective of step 1 is to identify similar regions within the given sequences, under the assumption that these similar regions might be closely related and therefore should be aligned together. We will use these similar regions to construct blocks in the next step. A straightforward solution is outlined in figure 3.2.

```
1. begin
2.   select a subset of the input sequences;
3.   enumerate every substring of the selected sequences;
4.   for each substring s do
5.       use s to find in all input sequences every substring with
         a similarity score above a fixed threshold;
6.       if there is at least 1 substring from each sequence then
7.           output this family of similar substrings;
8. end;
```

Figure 3.2: Step 1: a straightforward algorithm.

If M is the total length of input sequences, then the total number of distinct substrings in these sequences is $O(M^2)$. For each substring of length ℓ, searching for similar substrings requires time $O(M\ell + M^2)$. Because ℓ is usually much smaller than M, the total time complexity is $O(M^4)$. This algorithm will complete within reasonable time. However, there are two

issues that require discussion. First, it is not proper to use a fixed threshold. Selecting the criteria to be used to decide similarity is a critical task in this step. This threshold varies with different inputs: it might be high when the regions are well conserved, and low when dissimilarity increases. Second, even in the same input, some of the sequences might have higher degree of similarity than the others. If we raise the threshold, we might lose some regions in the sequences that are less similar to the majority. On the other hand, if we lower the threshold, there might be too many regions that are assumed to be similar. Therefore, using single threshold to decide similarity is too restrictive in this case.

3.3.3 The Refined Algorithm

There are many ways to resolve the first issue mentioned above. We could only consider exact matching substrings, or we could use previous results of pattern discovery, which is discussed on page 11, or we could utilize pairwise alignments, a method that is used by Boguski et al. [5] and Miller et al. [27]. Some previous work addresses the second issue by using incomplete blocks, i.e., a local alignment that involves only part of the input sequences. In our method, we use two similarity thresholds and, after examining the input, adjust the thresholds dynamically within a limited range to try to adapt to

each specific input. The idea is as follows.

Given a substring α and a set of input sequences $S_1, S_2, \ldots, S_n$. First, we align α with every sequence S_i to find the best local alignment, in other words, to find the best matching substring in each sequence S_i. (For this reason, we call α a *center string.*) Denote the scores as $c_1, c_2, \ldots, c_n$. Then compute two temporary thresholds, $\tau_1 \geq \tau_2$, both are functions of $|\alpha|$, and divide the sequences into three sets:

$$G_1 = \{S_i | c_i \geq \tau_1\}, G_2 = \{S_i | \tau_2 \leq c_i < \tau_1\}, G_3 = \{S_i | c_i < \tau_2\}. \tag{3.1}$$

If G_3 is nonempty, we say that α does not score well and simply discard it and try the next center string. Otherwise, α is a candidate, and we continue to find in all sequences every substring that is "similar" to it. To decide this similarity, we compute another two thresholds, T_1 and T_2, for sequences in G_1 and G_2 respectively:

$$T_1 = \min\{c_i | S_i \in G_1\} \tag{3.2}$$

$$T_2 = \min\{c_i | S_i \in G_2\} \tag{3.3}$$

For each sequence S_i we use T_1 as the threshold for S_i if $S_i \in G_1$; otherwise, let T_2 be the threshold. Then we proceed to enumerate substrings of S_i that have similarity scores with α above the threshold. Those substrings are inserted into a set after padded with spaces that is calculated according

to the optimal alignment between each substring and α. In our algorithm, such optimal alignment is obtained by backtracking in a dynamic programming table, and of all the substrings ending at a specific position of an input sequence, only the substring with the maximum similarity score with α is inserted into the set. We will discuss further in section dynamic programming on page 30.

The algorithm is shown in figure 3.3. At the beginning, we need to select some sequences (line 5), and enumerate every substring whose length falls within a specific range specified by the user at run time (line 6). Then we use every substring as a center string to collect similar regions (lines 8–23). The following are some of the issues that we want to discuss further.

selecting and marking sequences

Due to extensive computations involved in later steps, it could require too much computation if we enumerate every substring of every sequence when the input is large. At line 5, we use a simple scheme to select sequences: when the input is not large, we select and mark all the sequences; we decrease the number of marked sequences proportionally according to the increase of the input size.

```
1.  procedure grouping(I);
2.  var
3.    A, R, M : set;
4.  begin
5.    select a subset of input sequences I;
6.    A := {substrings of the selected sequences};
7.    MAKENULL(R);
8.    for each center α ∈ A do
9.      find the best matching substring in each sequence;
10.     τ1 = func1(|α|);   τ2 = func2(|α|);
11.     compute G1, G2, G3 according to equation (3.1);
12.     if G3 is empty then
13.       compute thresholds T1 and T2 using equations (3.2, 3.3);
14.       MAKENULL(M);
15.       for each input sequence Si do
16.         if Si ∈ G1 then T := T1;
17.         else T := T2;
18.         for every substring β of Si do
19.           if similarity(α, β) > T then
20.             β' := β padded with chosen spaces;
21.             INSERT(β', M);
22.       INSERT(M, R);
23.   combine_similar_sets(R)
24.   output R;
25. end;
```

Figure 3.3: Step 1: the refined algorithm.

recording substrings in a prefix trie

At line 6, we want to enumerate in the marked sequences every substring whose length falls within a range specified by the user at run time. Because two identical substrings involve identical computations in later processes, we don't want to keep all the copies of a substring. In the algorithm, we avoid this case of two identical substrings by using a *trie* [1, pages 163–169] to store the substrings. A trie is a data structure that supports efficient **INSERT**, **DELETE**, and **FIND** operations for keys that can be represented by a unique string. The nodes of a trie correspond to the prefixes of words in the set. We add a special marker to some of the nodes to indicate the end of a word. An example of such a trie, representing the strings "he", "hi", "his", "hit", "it", and "is" is shown in figure 3.4.

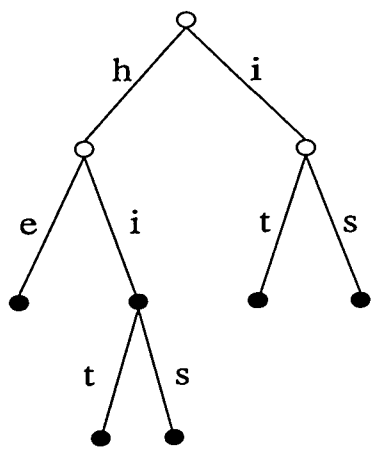

Figure 3.4: An example of prefix trie.

dynamic programming

At line 9, we use dynamic programming to find the best local alignment between a center string $\alpha = a_1 a_2 \cdots a_w$ and a sequence $B = b_1 b_2 \cdots b_v$, where at most K spaces is allowed inside each substring of B and no spaces are allowed in α, K is an integer specified by the user at run time. We will explain why no spaces are allowed in the center string later on. A dynamic programming table is shown in the following figure. Define C_{ij}^k = score of

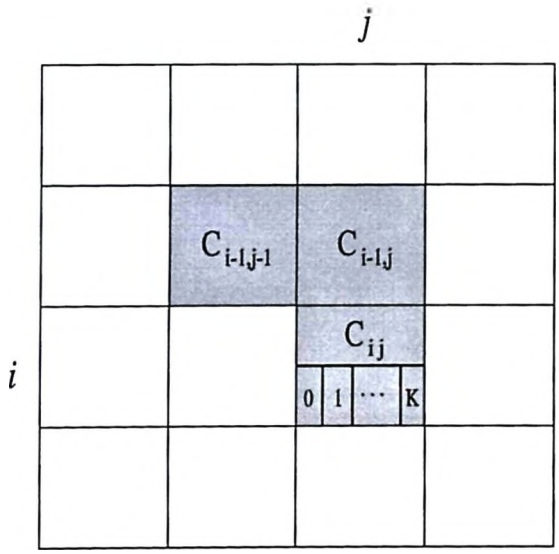

Figure 3.5: The dependencies of $C(i, j)$.

the best alignment between $a_1 a_2 \cdots a_i$ and all the substrings of B ending at position b_j with at most k spaces in each of the substrings of B, where $i \in [0, w], j \in [0, v], k \in [0, K]$. It follows that each cell in the table contains

$K+1$ values, $C_{ij}^0, C_{ij}^1, \ldots, C_{ij}^K$.

For each pair of characters x and y, $s(x,y)$ denotes the score obtained by aligning character x against character y. In our algorithm, it is defined in the form of a matrix that has a default value for each entry of the matrix. If x and y are closely related, then $s(x,y) = 1$; otherwise, $s(x,y) = 0$. The user can specify such relatedness between characters by feeding extra information into the process. We will come back to this later when we discuss the implementation issues.

Then we can compute each entry in the dynamic programming table as follows.

$$C_{ij}^k = \max \begin{cases} C_{i-1,j-1}^k + s(a_i, b_j) & (i > 0, j > 0) \\ C_{i-1,j}^{k-1} + s(a_i, \sqcup) & (i > 0, k > 0) \\ C_{ij}^{k-1} & (k > 0) \end{cases}$$

The first line in the equation corresponds to the case of aligning a_i against b_j. The second line corresponds to the case of aligning a_i against a space, and the substring ending at b_j is aligned with α's prefix, $a_1, a_2, ..., a_{i-1}$. The last line is added to guarantee that $C_{ij}^k = \max\{C_{ij}^u : u \leq k\}$.

The score of the best local alignment is

$$\max\left\{C_{wj}^K : 0 \leq j \leq v\right\}.$$

Similarly, from the last row of the table, we can obtain the best alignment

score between α and all the substrings ending at each position of the sequence. This information will be used at line 19 in figure 3.3. The padding at line 20 is computed by a traceback in this table starting from the corresponding column in the last row.

The reason that we don't allow any internal spaces in a center string is as follows. Suppose that $K = 1$, and the center string is "ACGT". Both "ATCG" and "ACTG" are similar to the center string, as shown in (1) and (2) of figure 3.6. Then we align the substrings and obtain a block as shown in (3). It turns out that we have two internal spaces in each substring, which violates the restriction of $K = 1$.

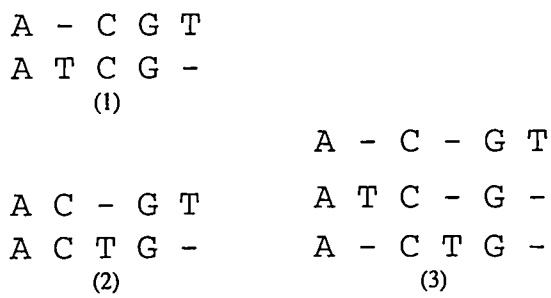

Figure 3.6: A violation of internal space restriction.

All strings stored in one set are similar to a common center string, but we do not further check the similarity between the strings. They are assumed to be similar to each other. Here is a tradeoff between time efficiency and the probability of grouping unrelated strings by mistake. For example, for a center string of length 8, the value of threshold $T = 6$, suppose both strings

A and B are similar to a center string. The probability of A and B having a similarity score of 4 is about 0.67, which is rather high. We could have added a procedure to check whether all strings related to the same center string are really "similar" to each other, but it will take a much longer time. Actually, the problem of having unrelated strings in one set becomes insignificant when the number of input sequences is large. When the number of input sequences increases, the number of strings in each set also increases. Even though two strings in a particular set might have a low similarity, the possibility of having other strings that relate these two strings to each other increases at the same time.

sequence grouping

At line 10, τ_1 and τ_2 are computed to divide the input sequences into three sets, where $\tau_1 \geq \tau_2$. We call G_1 a *good* set, which contains sequences with best local alignment score above threshold τ_1. G_2 consists of sequences with best local alignment score greater than τ_2. We call it a *reduced* set. G_3 contains sequences with score lower than τ_2, and we call it a *bad* set. As soon as we find a sequence in G_3, we will discard the substring α, assuming that α will not entail any useful blocks. Both func1($|\alpha|$) and func2($|\alpha|$) are functions of the length of the center string α. Their values increase when $|\alpha|$ increases, and decrease when $|\alpha|$ decreases. Currently, the function values are based on

experience and experimental results, in the form of a table that the user can easily adjust.

There are cases when some of the input sequences are closer to each other than the rest of the sequences, which make it improper to use single threshold to decide similarity. The idea is to divide sequences into two groups, good and reduced, and to further refine the thresholds for each of them.

eliminating redundant set

Another important operation in step 1 is to merge similar sets of substrings (line 23 in figure 3.3). Each set in $\mathcal{R}$ contains similar substrings and is associated with a center string α. Suppose $\mathcal{R}_1, \mathcal{R}_2 \in \mathcal{R}$, and $\mathcal{R}_1$ is associated with α_1, and $\mathcal{R}_2$ with α_2. It is easy to see that if $\alpha_1 = \alpha_2$ then $\mathcal{R}_1 = \mathcal{R}_2$. In the algorithm, this case has been taken care of by using a prefix trie to record only one copy of identical substrings.

On the other hand, if α_1 and α_2 are very similar then $\mathcal{R}_1$ and $\mathcal{R}_2$ might share a significant number of common elements; and if α_1 is a substring of α_2, $\mathcal{R}_1$ might contain corresponding substrings of those strings in $\mathcal{R}_2$. In the next step of the algorithm (figure 3.9 on page 38), we use the strings in each set $\mathcal{R}_i$ to construct blocks. Such redundant information would incur redundant computation. Therefore, we hope to eliminate such information as much as we can at step 1. Here we use a heuristic method to achieve it.

The algorithm is shown in figure 3.7.

```
1.  procedure combine_similar_sets(R);
2.  begin
3.    sort(R)
4.    for i := 1 to |R| do
5.      for j := i + 1 to |R| do
6.        if R_i and R_j consist of strings of the same length then
7.          S := R_i ∩ R_j;
8.          if (|S| ≥ |R_i| * c) or (|S| ≥ |R_j| * c) then
9.              R_j := R_i ∪ R_j;
10.             R := R − {R_i}
11.       else
12.         suppose that R_i contains shorter strings than R_j;
13.         S := {x|x ∈ R_i, ∃y ∈ R_j s.t. x is a substring of y};
14.         if (|S| ≥ |R_i| * c) then
15.             R := R − {R_i};
16.   return R;
17. end;
```

Figure 3.7: Step 1: subroutine combine_similar_sets.

The constant c at lines 8 and 14 of figure 3.7 is a real number that is less than 1. We should make sure that c is not too small. Otherwise, there will be a lot of elements in $\mathcal{R}$ being merged. In practice, we use $c = 0.9$, which works well in our tests.

The result of merging and deletion depends on the order in which the elements in $\mathcal{R}$ are inspected. Because the deletion is more likely to happen when comparing a set containing short strings with a set containing long strings, putting the "long" set in front of the "short" one can speed up the

process on average. This is done by sorting the sets in $\mathcal{R}$ at line 3, according to the length of strings that they contain.

3.4 Building Blocks: Step 2

The output from step 1 is a set $\mathcal{R}$, which contains sets of similar regions that might be padded with spaces. We want to align the regions into blocks, and use these blocks to construct an alignment in step 3.

Denote the i-th element of $\mathcal{R}$ as $\mathcal{R}_i = \{l_{i_1}, l_{i_2}, \ldots, l_{i_n}\}$, where l_{i_k} is a list of regions from the k-th sequence. Each region in the list is called a *repeat* of the corresponding center string. Figure 3.8 shows an example of $\mathcal{R}_i$. A

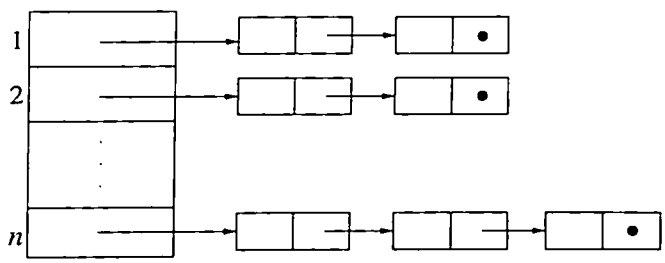

Figure 3.8: A set of similar regions.

block is formed by choosing one region from each list. Therefore, the set of blocks that could be generated from $\mathcal{R}_i$ is $\mathcal{B}_i = l_{i_1} \times l_{i_2} \times \cdots \times l_{i_n}$. The size of $\mathcal{B}_i$ is also referred to as the *volume* of $\mathcal{R}_i$. In practice, this number could increase exponentially in the length of the lists and it becomes impractical

to generate all the blocks. We call it the problem of repeats.

In our algorithm, we use a population control algorithm (page 40) to try to reduce the number of blocks generated from $\mathcal{R}$ below q, where q is an integer that is set at run-time. The process is outlined in figure 3.9. Let us assume that we have k sets of similar regions $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k$, and $v_1, v_2, \ldots, v_k$ represent the volume of each set respectively, such that $v_1 \leq v_2 \leq \ldots \leq v_k$. First, we compute quota of blocks to be generated, $q_1, q_2, \ldots, q_k$, and then, for each set, use the quota as a suggestion on how to form blocks. If $v_i \leq q_i$, then we simply generate all the blocks in set $\mathcal{R}_i$, by using every possible combination of the regions; otherwise, we use the quota q_i as a suggestion, and try to eliminate some unlikely scenarios.

3.4.1 Quota of Blocks

Integer q is the total number of blocks we want to generate, and each set has its quota of blocks to be generated. The algorithm tries to keep the number of blocks generated from each set below the quota for that set. The idea is to distribute q among all the sets, which is proportionally adjusted according to their volumes.

First, we compute a threshold

$$t = \frac{q}{\sigma|\mathcal{R}|}, \tag{3.4}$$

```
1.  procedure blocking(R, q);
2.  var
3.    B, S : set;
4.  begin
5.    MAKENULL(B);
6.    compute_quota(R, q);
7.    for each R_i ∈ R do
8.      if q_i ≤ volume(R_i) then
9.        S := generate_all_blocks(R_i);
10.     else
11.       S := generate_blocks(R_i, q_i);
12.     B := B ∪ S;
13.   merge_similar_blocks(B);
14.   output B;
15. end;
```

Figure 3.9: Step 2: building blocks.

where σ is an integer > 1, and $|\mathcal{R}|$ is the size of set $\mathcal{R}$. Threshold t acts like an average number blocks to be generated from each set. If a set's volume is greater than t, we call it a *large* set; otherwise, we call it a *small* set. All the sets in a small set are generated, while only a proportional number of blocks are generated from a large set. Denote $\mathcal{I} = \{i | v_i \leq t, i \leq |\mathcal{R}|\}$, where v_i is the volume of $\mathcal{R}_i$. Set $\mathcal{I}$ contains all the indices of small sets. The quota q_i for set $\mathcal{R}_i$ is computed as follows:

$$q_i = \begin{cases} v_i & i \in \mathcal{I}, \\ v_i \frac{q - \sum_{j \in \mathcal{I}} v_j}{\sum_{j \notin \mathcal{I}} v_j} & \text{otherwise.} \end{cases} \tag{3.5}$$

Threshold t is fine-tuned by a constant integer σ (equation 3.4). We find that of all the sets generated in step 1, usually there are a few sets that have extremely large volumes, while most of the others are much smaller. Therefore, removing σ from equation 3.4 could end up with generating blocks without any restriction for most of the sets, and putting too much restriction on the large sets, which are supposed to contain more information. In order to keep as much information as possible, we increase the quota for large sets by increasing the value of σ, and hence lowering threshold t. This relation is shown in figure 3.10.

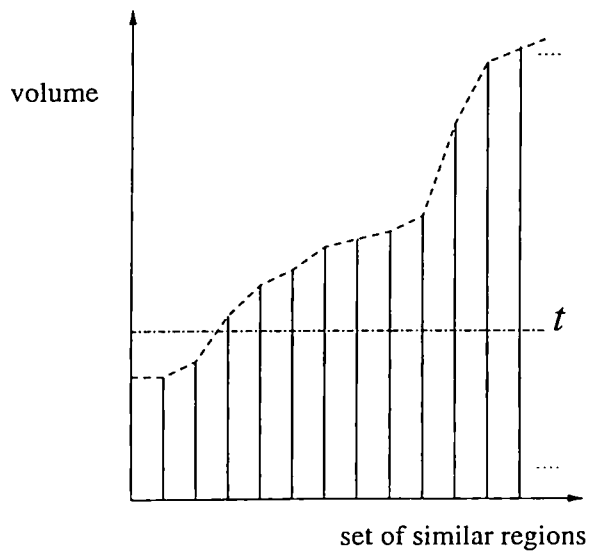

Figure 3.10: Lower threshold t to retain more information. In equation 3.5, the value of $q - \sum_{j \in \mathcal{I}} v_j$ increases with the decreasing of t. Therefore, the quota for each large set increases.

3.4.2 Population Control

A critical task in step 2 is to reduce the number of blocks. Given a set R of similar regions, which has a volume greater than its quota, the objective is to try to reduce the number of blocks below its quota while retaining as much information as possible.

In the algorithm, we use a method to eliminate some unlikely combinations. Suppose we have n input sequences, and a set R of similar regions, $R = \{l_1, l_2, \ldots, l_n\}$, where l_i is a list of regions from the i-th sequence for all $i \in [1, n]$. A *partial block* is formed by aligning $k \in [1, n]$ regions from k lists in R. The question is "Given a partial block, how to decide whether this is an unlikely scenario?"

The idea is that we first obtain an alignment $\mathcal{A}$ of the k involved sequences, and check whether the starting position of the regions are "close" enough in the alignment. For example, suppose $k = 2$, the partial block contains only two regions r_i and r_j from the i-th sequence and the j-th sequence. Considering alignment $\mathcal{A}$, let L be the length of the alignment, and region r_i starts at column x, and region r_j at column y. We define regions r_i and r_j to be *close* if $|x - y| < L/2$. In general, a partial block B is *valid* iff each pair of regions in B are close with respect to alignment $\mathcal{A}$. We can obtain this alignment $\mathcal{A}$ by simply padding the sequences with spaces at the end to

make them have a same length, or we can utilize other alignment programs to generate one for us. Because the criteria that we use to decide closeness is rather permissive, the result would not heavily rely on a specific alignment program. Procedure *generate_blocks* uses this idea to reduce the number of blocks as shown in figure 3.11.

```
1.  procedure generate_blocks(R, quota);
2.  var
3.    queue : PbQueue;
4.    done : boolean;
5.  begin
6.    MAKENULL(queue);
7.    for each list ℓ in R do
8.      INSERT(ℓ, queue);
9.    done := false;
10.   while not done do
11.     ℓ1 :=DELETEMAX(queue);
12.     ℓ2 :=DELETEMIN(queue);
13.     ℓ3 :=DELETEMIN(queue);
14.     if EMPTY(queue) then
15.       done := true;
16.     ℓ4 := partial_aln(ℓ1, ℓ2, ℓ3);
17.     INSERT(ℓ4, queue);
18.     if number of blocks in queue ≤ quota then
19.       done := true;
20.   output all blocks in queue;
21. end;
```

Figure 3.11: Step 2: subroutine to generate blocks

Subroutine *partial_aln* takes three lists of partial blocks as input, and

merge them into one, which contains a subset of all possible combinations of the three input lists. In this process, it uses a temporary partial alignment of the involved sequences to eliminate unlikely combinations. PbQueue is a priority queue of partial block lists, and operation **DELETEMAX/ DELETEMIN** deletes and returns the longest/shortest list in the queue. When there is only one list in *queue* or when the number of blocks to be generated has been reduced below *quota*, *partial_aln* will proceed to generate all blocks in *queue* and return.

In an earlier version, procedure *partial_aln* takes out the three longest lists in the queue. Later we find it runs faster on average if we delete the longest list and two shortest lists. The main reason is that using short lists enables us to discard some invalid combinations earlier in the process, rather than to keep them until they are found to be invalid after a lot of comparisons. This is a heuristic based on computational experiments.

Instead of taking out three lists, we could choose to take out only two lists from the queue and try to eliminate unlikely combinations; but then, a partial alignment might degenerate into a pairwise alignment, which we have been trying to avoid. Many programs can handle three sequences alignment well, so we use three in our algorithm.

Quota of blocks for each set provides a suggestion in the partial alignment.

Although it is not guaranteed to be able to reduce the number of blocks below quota, it helps to rule out most of the invalid alignments of the regions and, in most cases, keep the number of blocks reasonably low. However, in some extreme cases, the number of blocks could still be much higher than the program can handle. Such a case could occur if, in the temporary alignment $\mathcal{A}'$, a large number of similar regions are located in a "narrow" window, which has a width less than one half of the length of $\mathcal{A}'$. They would be considered to be close enough, and all the combinations are valid. The program will give a warning if this occurs, then the user can choose different run-time parameters to raise similarity thresholds or increase the length of similar regions to reduce the number of blocks.

3.4.3 Merging Blocks

Due to the extensive computation involved in the next step, it is desirable to have as few blocks as possible. At the end of step 2, procedure *merge_similar_blocks* is called. This procedure does two things. One is to delete extra copies of the same block. Although we have tried to eliminate redundant sets of regions, it is still possible to generate many copies of a same block; it also introduces redundant blocks when we use a substring of a center string as a new center string to form blocks. The other issue is to deal

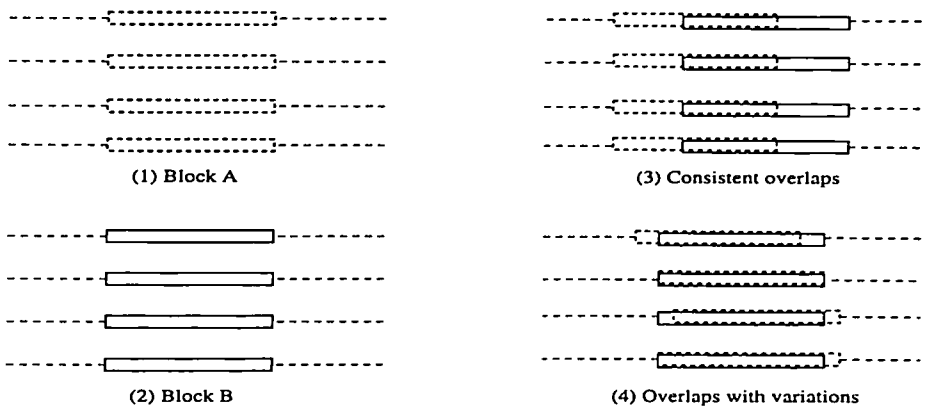

Figure 3.12: Overlapping blocks.

with similar copies of blocks. This is necessary because in our algorithm the regions are not generated using exact match. A well conserved motif might bring us many similar blocks; each one is a bit different from the true motif. In other words, if we find many such similar blocks within a small window, it could evidence the existence of one or more true motifs hiden in those blocks.

Similar blocks might overlap in a significant number of regions. In figure 3.12, a schematic representation of two blocks A and B is shown on the left, and two different ways they can overlap are shown on the right. A simple case is when two blocks overlap consistently in all sequences as shown in (3). We can easily construct a longer block that subsumes blocks A and B. It becomes more complex when the overlaps vary from one sequence to another (4). It is difficult to say which one is the right alignment. In our algorithm,

we merge some of the blocks, and let the chaining process decide which block is more important.

We define two blocks to be *siblings* if they overlap in every sequence no less than p percent of each region, where p is a constant integer less than 100, which has different values for DNA and protein sequences. If each region in block A is a substring of the corresponding region in block B, we say A is a *child* of B, and B a *parent* of A. The algorithm to merge blocks is outlined in figure 3.13. Subroutine *merge* at line 14 simply returns a block that aligns the regions covered by two siblings.

```
1.  procedure merge_similar_blocks(B);
2.  begin
3.    for i := 1 to |B| do
4.      for j := i + 1 to |B| do
5.        case get_relation(B_i, B_j) of
6.          'SAME':
7.            B := B - {B_j};
8.          'CHILD':
9.            B := B - {B_i};
10.           goto line 5;
11.         'PARENT':
12.           B := B - {B_j};
13.         'SIBLINGS':
14.           B_j := merge(B_i, B_j);
15.           B := B - {B_i};
16.           goto line 3;
17. end;
```

Figure 3.13: Step 2: procedure merge_similar_blocks

3.5 Chaining: Step 3

A block β from n sequences has n regions, one from each sequence; the length of the block is $|\beta|$. If β and β' are blocks from the same n sequences, then β *precedes* β' if the last sequence position of the i-th region in β strictly precedes the first sequence position of the i-th region in β', for all $i \in [1, n]$. Blocks $\beta_1, \beta_2, \ldots, \beta_m$ form a *chain* if β_j precedes β_{j+1} for all $j \in [1, m-1]$.

The input of step 3 is a set of candidate blocks, and the objective is to find the best chain of blocks, and use these blocks to construct a global alignment. A best chain of blocks can be a chain with the maximum number of blocks, or a chain that maximizes a score function. In our algorithm, we handle both cases. After obtaining a chain of blocks, constructing an alignment is rather straightforward. Each block in the chain is aligned already, only the regions between two consecutive blocks need to be aligned.

3.5.1 Near-Optimal Chains

A straightforward method of block chaining is a special case of classic optimal-path algorithm for directed acyclic graphs [10]. Given a set of blocks, we can construct a directed acyclic graph, or dag for short. Each block is associated with a vertex in the dag, and there is an edge from vertex i to vertex j if the block associated with the i-th vertex precedes the block associated with

the j-th vertex. An optimal path in the dag can be found by a breadth-first search on the graph. It runs in $O(B^2)$ time, where B is the number of blocks. For the special case of two sequences, there exist block-chaining algorithms that run in $O(B \log B)$ time or faster [9, 11, 12]. However, their methods do not generalize to $k > 2$ sequences. Zhang et al. [41] propose to build a highest-scoring chain by adopting the use of k-D trees. Although they do not give a theoretical analysis of their method's time complexity, they report the results of experiments indicating that it is far better than $O(B^2)$, but not as good as $O(B \log B)$.

Blocks in the optimal chain might not contain all the conserved regions. There could be other blocks that are even more informative but do not appear in the optimal chain. In our method, we use an algorithm [see 10] to obtain score of the optimal path, then continue to generate all near-optimal paths. In this way, the user has more choice about the blocks being used in the final alignment.

3.5.2 Scoring Scheme

Each block β has a score $\delta(\beta) > 0$, and there is a penalty $connect(\beta, \beta')$ for connecting β to a chain starting at β'. Define $score(\beta)$ to be the maximum

score over all block chains starting with β, which is,

$$score(\beta) = \max\{\delta(\beta) - connect(\beta, \beta') + score(\beta') : \beta \text{ precedes } \beta'\}$$

Therefore, a best chain is the one that maximizes $score(\beta)$ over all candidate blocks.

The weight of a block $\delta(\beta)$ is based on the length of the block ($|\beta|$). Furthermore, for blocks that are the results of merging similar blocks, we give them more weight than those that are not. The reason is that we assume similar blocks give a strong hint of the existence of a real motif.

$$\delta(\beta) = \begin{cases} wn|\beta| & \text{if } \beta \text{ a result of merging} \\ n|\beta| & \text{otherwise,} \end{cases} \tag{3.6}$$

where w is a constant real number that is greater than 1. This function has a time complexity of $O(1)$.

The penalty $connect(\beta, \beta')$ for connecting two blocks is computed as follows:

$$connect(\beta, \beta') = \sqrt{\sum_{1 \le i \le n} (d_i - \bar{d})^2}, \tag{3.7}$$

where d_i is the difference between the ending position of the i-th region in β and the starting position of the i-th region in β', and $\bar{d}$ is the average of d_i for all $i \in [1, n]$. The purpose is to penalize those blocks, in which the distance between corresponding regions varies drastically. This function can be computed in $O(n)$ time.

3.5.3 The Algorithm

In the corresponding weighted directed acyclic graph, each vertex is associated with a block β, the weight of the vertex is $\delta(\beta)$. There is an edge from vertex i to vertex j if block β_i precedes block β_j. After constructing such a graph, we add two more vertices: a *source* that has edges pointing to every other vertex, and a *sink* that is pointed to by all other vertices. Then an optimal chain in the original problem becomes an optimal path from *source* to *sink*.

The algorithm to compute near-optimal chains is shown in figure 3.14. First we compute $score(\beta)$ for each block β, and obtain the score OPT of an optimal path. Then put *source* in an empty chain, and let *source* be the current vertex cv. We check each vertex w that is connected to cv to see whether w could occur next in the near-optimal chain, i.e.,

$$\text{score of current chain} + score(w) - connect(cv, w) \geq \varphi OPT, \text{ where}$$

φ is a real number that can be set by the user. If cv=*sink* then a near-optimal chain has been found. This depth-first search continues until all near-optimal chains are found.

```
1.  procedure find_near_opt_path(B, φ);
2.  var
3.    G : graph;
4.    candidate : stack;
5.    chain : array of integer;
6.  begin
7.    G := construct_dag(B);
8.    OPT := score_of_optimal_path(G);
9.    MAKENULL(candidate);
10.   pos := 1;   cur_score := 0;   cv := source;
11.   chain[pos] := cv;
12.   for all edges (cv, w) in G do
13.     if score(w) − connect(cv, w) ≥ φOPT then
14.       PUSH({w, pos}, candidate);
15.   while not EMPTY(candidate) do
16.     POP({cv, pos}, candidate);
17.     pos := pos + 1;   chain[pos] := cv;
18.     cur_score := update_chain_score(pos);
19.     if cv = sink then
20.       output blocks chain[1] ... chain[pos];
21.     else
22.       for all edges (cv, w) in G do
23.         if cur_score + score(w) − connect(cv, w) ≥ φOPT
24.         then
25.           PUSH({w, pos}, candidate);
26. end;
```

Figure 3.14: Step 3: compute all near-optimal chains.

Chapter 4

Implementation

In this chapter, we first give a brief description about the programming language and platform we use to develop the program. Then the class design is elaborated to give an overview of the structure of the program. At the end of this chapter, we discuss time and space efficiency of our algorithm and some of the other issues involved in the implementation.

4.1 The Sequence Alignment Program

Based on the algorithms discussed in the previous chapter, we have developed a program BMA to do blocked multiple sequence alignment. The program is written in C++, and compiled using GNU C++ compiler (g++ version 2.8.1) in Sun Solaris. G++ is a freely available C++ compiler produced by the Free Software Foundation and is available for a wide range of computers. We do

not use any platform specific features of the C++ programming language, therefore, the program can be easily ported to other platforms, such as MS-Windows, Linux, and many other Unix environments, as long as there is a decent g++ compiler on that platform. With minor modifications, the program can be compiled using Borland C++ Builder 4.

4.2 Top-Level Design

4.2.1 Class Diagrams in UML

The most common object-oriented design representation is the diagrams of the Unified Modeling Language (UML), which is designed by Grady Booch, Ivar Jacobson, and Jim Rumbaugh. UML is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. It simplifies the complex process of software design, making a "blueprint" for construction. In this chapter, we use some of the notations of class diagram in UML to document our design.

UML class diagrams have two principle components: classes and relationships. Classes provide the set of classes and direct information about each class. Relationships provide other information, including the class hierarchy and how the various classes are related structurally.

A class is represented by a rectangle with three compartments separated

by horizontal lines. The top compartment contains name of the class, middle compartment holds structure (data fields) of the class, and the third compartment holds behavior (member functions) of the class. The icon at the start of each item indicates the visibility of the item: a plain icon indicates public, an icon with a lock indicates private, and an icon with a key indicates protected.

Lines are used in the diagram to express relationships. Two basic types of relationships can be defined, inheritance and containment. Inheritance is drawn as a line with a triangle between two classes where the triangle points to the superclass. Containment in general represents *has-a* relationship, which is drawn as a line with an arrow that points to the class being contained. Aggregation is a stronger form of containment between a whole and its parts. It is drawn as a line which contains a diamond placed next to the class that plays the role of the whole.

4.2.2 Basic Constructs

Our program consists of 44 classes. Some of the them are basic constructs used by higher level classes. For example, class **SeqError** is the base class of various error handlers used by other classes. These basic constructs are shown in figure 4.1 and figure 4.2.

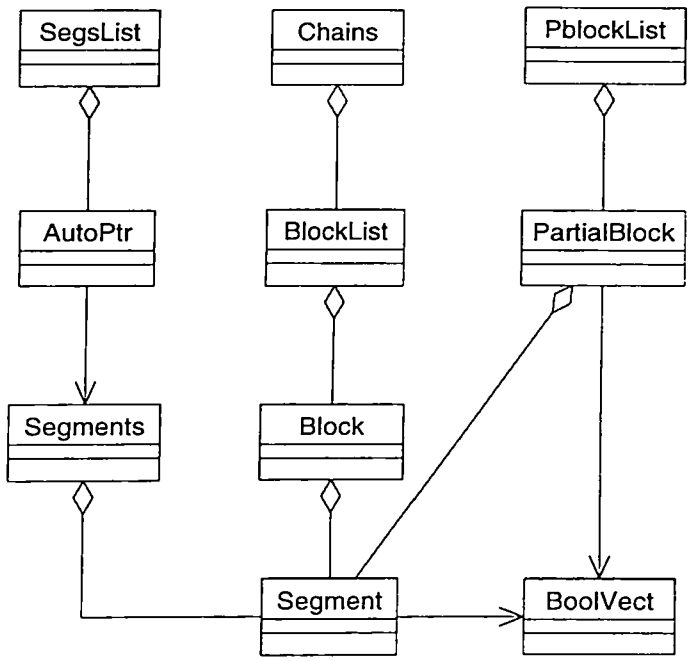

Figure 4.1: Class diagram for basic constructs.

Class **Segment** encodes a region defined in the algorithm, and the aggregation of **Segment** is called **Segments**, which corresponds to the set of similar regions in our algorithm. The underlying structure of **Segment** is a boolean vector, recording the patter of the substring, and an integer, indicating the start position of the substring. Class **Segments** is implemented as a vector of linked lists of **Segment** objects. We could have added to **Segment** another data field to indicate from which sequence this region

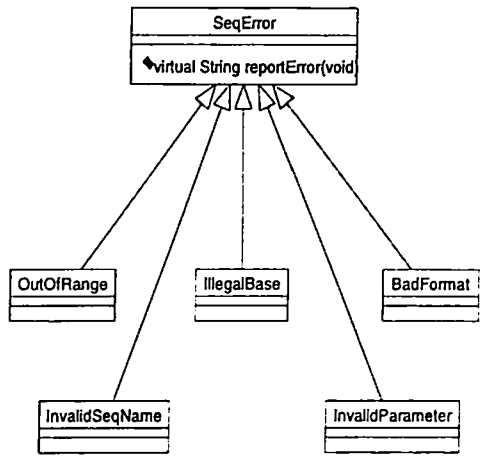

Figure 4.2: Class diagram for error handlers.

originates. However, we have a large number of **Segment** objects residing in memory, adding this data field would cause unnecessary memory overhead. Actually, this information can be retrieved by other methods. For example, in **Segments**, similar regions are recorded in linked lists, while every list corresponds to a specific input sequence. We will come back to the memory issue later, and we will see other cases when we add seemingly redundant data field to improve the running time.

Class **AutoPtr** comes into handy when we want to store a large object in memory. After the creation of a large object, copying or moving the object requires extra memory and CPU time. An alternative way is to keep a pointer of the object. However, keeping the pointer as a data member will not free

the object automatically from memory when the object holding the pointer is destroyed, since a pointer in C++ is not really a class. One solution is to wrap the pointer by a generic class, while this class may or may not own the pointer. If it has the ownership, the object referenced by the pointer will be destroyed when the wrapper's destructor is called. Sets of similar regions in the algorithm are represented by a linked list of such wrapper objects, each of which contains a pointer referring to a **Segments** object.

Both class **Block** and class **PartialBlock** contain a vector of **Segment** objects, and **PartialBlock** contains a boolean vector of size n, which is the number of input sequences. This boolean vector indicates which sequences are involved in the partial block. **SegsList**, **BlockList**, and **PblockList** store a set of objects in the form of linked lists (figure 4.1). Finally, class **Chains** holds all the near-optimal block chains generated in the program.

4.2.3 Top-Level Classes

Figure 4.3 shows the top-level classes in our design. The **AbstractMain** class does not really exist. We only use it in the diagram for the purpose of notational convenience. It abstract the *main* function of the program.

Class **Options** manages all the parameters of the program. Each parameter has a default value and can be set by the user at run-time. It also

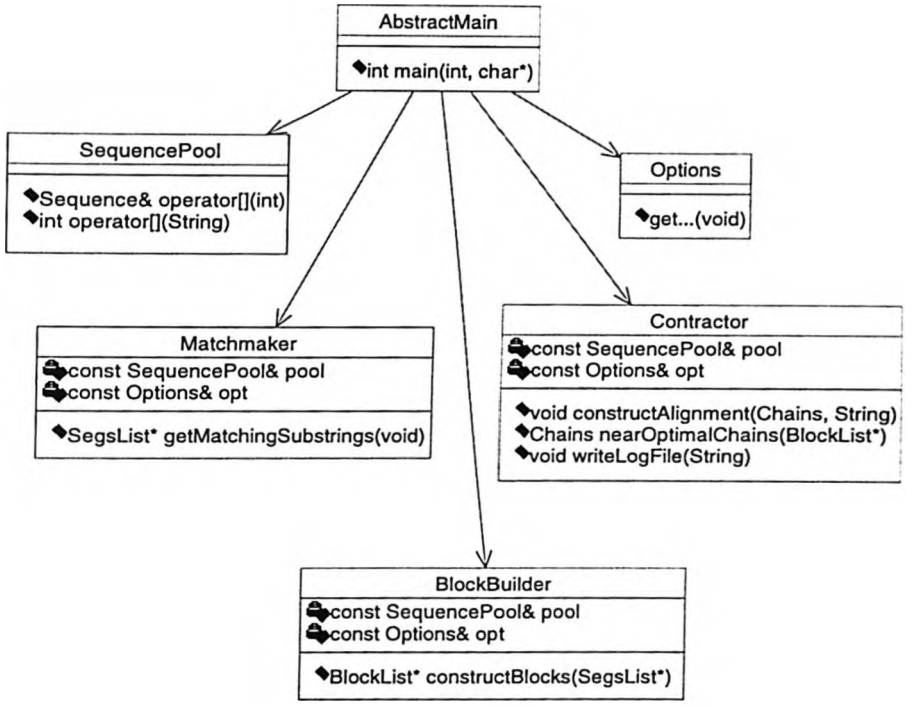

Figure 4.3: Class diagram of top-level classes.

provides an error checking mechanism to resolve conflict settings. The algorithm contains many threshold values that are calculated dynamically based on the parameters managed by this class.

The **SequencePool** object holds all the input sequences. It can index a sequence in the pool by the sequence name, and provides a string representation of the sequence. It also provides sequence input and output functions.

All the operations in step 1 of the algorithm are done by **Watchmaker**.

It collects similar regions and output the regions in a **SegsList** object, which **BlockBuilder** takes as input and generates a list of candidate blocks. The "contract" for the **Contractor** object is to find all the near-optimal chains and build alignments for each of them or only for the optimal chain, which depends on the user's choice. It also generates a log file reporting which blocks are found in the chains, and the score for each of them.

4.3 Second-Level Designs

In the previous section, we give an overview of the top-level classes. Now let's see different parts of the system.

4.3.1 SequencePool

SequencePool provides accessor/modifier methods by overloading the subscript operator [ ]. Given index of a sequence, it returns a **Sequence** object, which in turn provides functions to access or modify a sequence. In this small package, we have another class **Translator**. It translates between internal and external representations of a sequence. For example, DNA sequences are strings over an alphabet of four characters (A,C,G,T), which internally are mapped to integers (0,1,2,3).

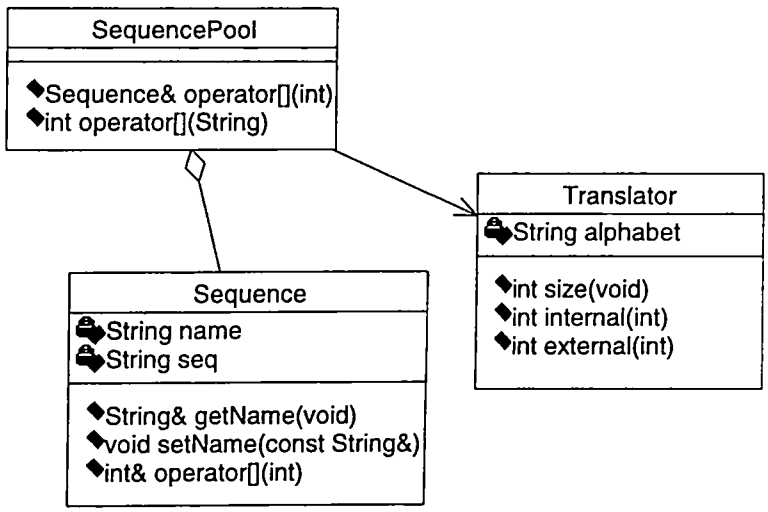

Figure 4.4: Class diagram for SequencePool.

4.3.2 Watchmaker

As We mentioned before, in step 2 we use a prefix trie to record center strings that has been used. A prefix trie holds the root of the trie, and the only public method is "bool search(key_type[])", which returns true if the word is in the trie; otherwise, inserts the word, and returns false. Basically, a trie node v holds labels of the edges connecting v to v's child nodes, and pointers pointing to the corresponding child nodes. Member function "bool wordEndWith(key_type e)" in **TrieNode** returns true if there is an end-of-word marker on the edge labeled with character "e". Otherwise, it returns false, and adds an edge with the marker. In this way, a word is inserted into

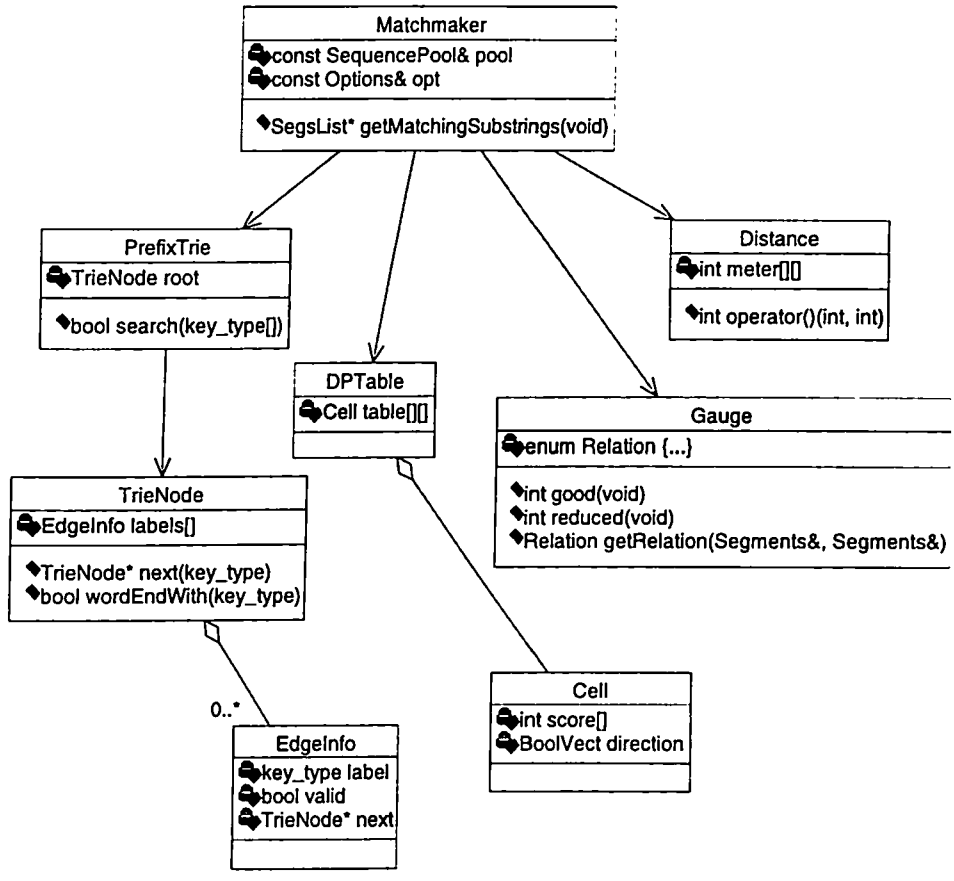

Figure 4.5: Class diagram for Watchmaker.

the trie.

When comparing protein sequences, people usually use various similarity scheme to specify the relatedness. In our program, the default scheme we use is (F,Y), (M,L,I,V), (A,G), (T,S), (Q,N), (K,R), and (E,D), which is also used by McClure et al. [25]. The user can customize the relatedness in a configuration file, which is read by a **Distance** object. We use this object all through the program as the only source to get the relatedness of two characters. For example, this object is used to compare similar substrings, and is also used in the prefix trie for comparing keys.

A dynamic programming table **DPTable** contains a 2-dimensional array of **Cell** objects. Each **Cell** object has all the information of $K + 1$ cases, as shown in figure 3.5. Since each entry depends on two other entries on the upper row, we can simply use a boolean vector to record the back-trace direction.

Class **Gauge** computes τ_1, τ_2 and performs sequence grouping. The values of τ_1 and τ_2 are functions of length of the center string. **Gauge** maintains a lookup table that has three sets of values. Each set corresponds to *high*, *normal*, or *low* similarity degree that can be set by the user. The default setting is *normal*. This parameter gives us some degree of flexibility to adjust to different types of input. For example, if we find there are too many

repeats in the input sequences, we can run the program again and set the degree to *high*. The lookup table is stored in a disk file, and can be modified easily. Perhaps the values can be fine-tuned further for each special type of input by feeding the program with a large amount of real data with known characteristics of that type of sequences. **Gauge** also decides the relationship between two **Segments** objects, and this information is used to merge similar sets of substrings in subroutine *combine_similar_sets* (figure 3.7).

4.3.3 BlockBuilder

BlockBuilder uses **PbQueue** and **Puritant** to form blocks and to keep the number of blocks under control. In class **Puritant**, function "getAln" returns a temporary alignment, which is used by **BlockBuilder** to decide the candidacy of a partial block in **PbQueue**.

AlignHelper is an abstract class. The purpose of this class is to provide an interface to use other alignment programs to generate temporary alignments. We use Clustal W in our program, which is implemented in the derived class **Cclustal**. Any other class that implements those virtual functions can be used in place of **Cclustal**. The functions include preparing sequences in a special input format that is used by the helper program, feeding the input into the program, and reading the alignment generated by

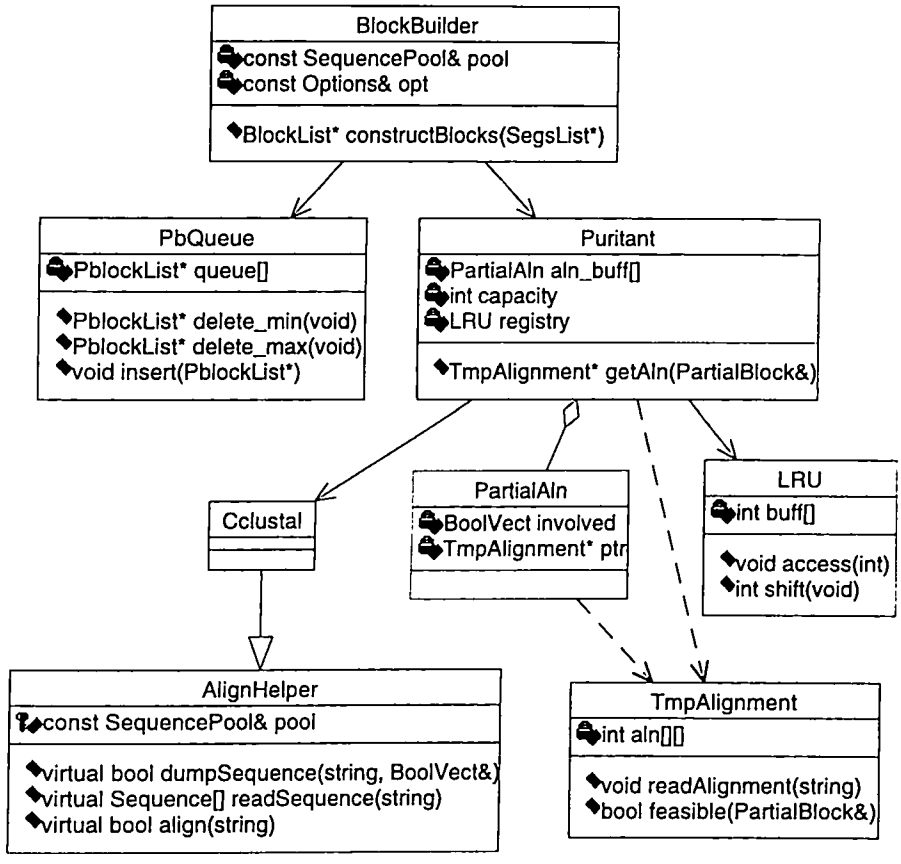

Figure 4.6: Class diagram for BlockBuilder.

the program into memory.

Given a partial block, **Puritant** generates a temporary alignment for those involved sequences and evaluates the validity of this partial block based on the alignment. **Puritant** stores all the alignments generated so far, some reside in memory, while others are saved in disk files. A registry records up to M temporary alignments being kept in the memory, M depends on the available memory on the computer. When a new alignment is needed, **Puritant** uses the helper program to generate one, and read this alignment into memory. If the registry is full, the least recently used alignment is swapped out, and this new one is put at the beginning of the LRU list.

4.3.4 Contractor

In the chaining step, the **Contractor** object first construct a **Cdag**, which represents a dag in the form of adjacency lists. Then it asks **Cdag** for the longest path or all near-optimal paths. For each block chain, **Contractor** can generate a final alignment consisting of the blocks. The unaligned regions between consecutive blocks are aligned by the helper program. **Contractor** also generate a log file to give detailed information about all the blocks that appear in the final alignment.

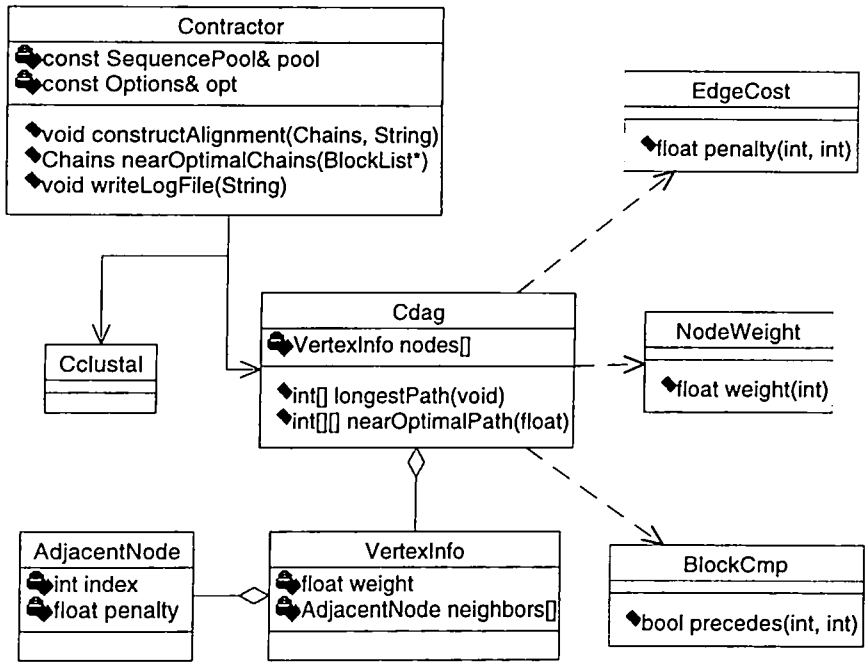

Figure 4.7: Class diagram for Contractor.

Cdag only records the index of each block. The detailed information of all blocks are stored in a vector outside of the Cdag object. Three function objects are passed to Cdag so that we can compute the score for each block.

4.4 Space Efficiency

There are two types of memory space used by the program. One is statically allocated memory. For example, we need some memory to store input sequences and blocks generated in the program. This memory space cannot

be used by other processes until the program terminates. The other type is dynamically allocated memory. Typically, a task first initiates a memory request and then, after obtaining the memory space, performs some computation. At the end of the task, such dynamically allocated space is freed and put back to free-store, and hence becomes available to other processes running at that time.

The size of statically allocated memory depends on the size of input and the number of blocks generated. For example, if integers and pointers are of 4 bytes in memory, 10000 blocks from 12 sequences occupies a memory space of about 2MB. In most cases, statically allocated memory is only a small fraction of all the memory space needed by BMA.

The major memory consumption happens in the process of population control (in step 2) and the construction of directed acyclic graph (in step 3). In step 2, we try to eliminate some unlikely alignments of similar regions by using partial alignments. Most of the sets are relatively small, and one or two partial alignments will reduce the number of blocks below quota. However, there are cases when the partial alignments don't help much. The consequence is that we have a large number of partial blocks generated, even though most of them will be eliminated in the later process. In our tests, such memory needs range mostly from 30MB to 150MB, but there are a few

that require as much as 300MB. It could happen that these partial blocks exhaust all available memory. If it happens, the program will exit, and give a warning that there are too many blocks.

In step 3, we need construct a graph to find block chains. In our program, we use adjacency lists to represent the graph. In an implementation under the same assumption about the size of integers and pointers, a graph with V nodes and E edges will require a space of $12V+8E$ bytes, in contrast to $8V+4V^2$ if we represent the graph in an adjacency matrix. We observe that on average $E = V^2/10$ in our test. Therefore, adjacency matrix representation requires about five times as much memory as adjacency lists representation. In our implementation, a graph with 10000 nodes represented as adjacency lists requires a memory space of about 120MB on average.

4.5 Time Efficiency

We run the program on our workstation, a Sun Ultra 2 Workstation running Solaris 2.x, using test data presented in the next chapter. The running time ranges from a few seconds to an hour. It highly depends on the input data. In this section, we give an analysis of the time complexity of our algorithm.

Given n input sequences of average length N, in step 1 we use dynamic programming to compare each center string α with every sequence,

allowing at most K spaces in each substring. In practice $|\alpha|$ is within a range of constant integers. Consider the algorithm outlined in figure 3.3 on page 28. The operations of line 5-6 require $O(nN)$ time. Within the for-loop of line 9–21, the most time consuming operation is the comparison between α and every sequence, which has a time complexity of $O(KnN|\alpha|)$. Computing τ_1 and τ_2 can be done in $O(1)$, and computing G_1, G_2, G_3, T_1, and T_2 in $O(n)$. Collecting similar substrings (line 20) has a time complexity of $O(nN)$. Therefore, the time complexity of the for-loop is $O(\sum_{\alpha}(KnN|\alpha| + n + nN)) = O(KnN\sum_{\alpha}|\alpha|)$, which is $O(Kn^2N^2)$. Subroutine *combine_similar_sets* inspects each pair of **Segments** objects, and merge them if they share a significant number of common regions. Let P be the maximum number of repeats of a center string in one sequence, and $|\mathcal{R}|$ be the number of **Segments** objects generated in step 1. From the algorithm outlined on page 35, it is easy to see its time complexity is $O(nP^2|\mathcal{R}|^2)$, and the total time complexity of step 1 is $O(Kn^2N^2 + nP^2|\mathcal{R}|^2)$.

The total length of input is $M = nN$. Because the length of center strings is within a constant range, the maximum number of similar sets is $O(M)$, which is also the maximum number of center strings. Hence $|\mathcal{R}| = O(M)$. Moreover, if both P and K are below a constant integer, then the time complexity will be $O(n^2N^2 + n^2N) = O(M^2)$.

In step 2, we use the algorithm shown on page 38 to generate blocks. The running time highly depends on the input sequences. The worst case is when none of the partial alignments helps to eliminate any blocks. Then for each set of substrings, we would have $O(n)$ partial alignments. We use each partial alignment to check the validity of at most V_{max} partial blocks, each taking $O(n^2)$ time, where V_{max} is the volume of the largest set of substrings. Therefore, the worst case time complexity of step 2 is $O(|\mathcal{R}|n(T'+n^2V_{max})) = O(nT'|\mathcal{R}| + n^3V_{max}|\mathcal{R}|)$, where T' is time complexity of the helper alignment program.

Let $|\mathcal{R}| = O(nN)$. Furthermore, suppose $V_{max} = 10000$. In this special case, the time complexity of step 2 will be $O(n^2NT'+10000n^4N)$. It depends on the time complexity of the helper alignment program. If that program runs too slow, the overall running time might be very long. However, if the number of blocks that could be generated from $\mathcal{R}$ is below q, which is the number of blocks the program can handle, then the helper program will not be called by BMA because no partial alignment is needed. In that case, the time complexity of step 2 is $O(|\mathcal{R}|V_{max})$.

We construct a dag in step 3 by inserting nodes into the graph. The calculation of each node's weight (equation 3.6) is in $O(1)$ time, and edge cost (equation 3.7) in $O(n)$. The precedence relationship between two blocks

can be determined in $O(n)$ time. Hence constructing the graph has a time complexity of $O(nB^2)$, where B is the number of blocks generated in step 2. The score of the optimal path and all near-optimal paths can be obtained in $O(B^2)$ time, and therefore the total time complexity of step 3 is $O(nB^2)$.

4.6 Other Issues in the Implementation

4.6.1 Improving the Performance

Besides the other things we do to improve the time efficiency, we run *gprof* to produce an execution profile of the program. To our surprise, we find that the program spends more than 70% of time in the function *getRange* of class **Segment**. This function returns the start position and the end position of a region in an input sequence.

The problem is that we use an integer to store the start position, and a boolean vector to record the pattern of the region — false for a space, and true for a character from the sequence. It is sufficient to record a region in this way; but each time when we want to know the end position of a region in the sequence, we have to go through the boolean vector to count how many real characters it has, and return this number plus index of the start position minus 1. After we add an extra data member in class **Segment** to record the end position, the time spent in function *getRange* is negligible comparing

with the time spent in other operations.

4.6.2 Fine-Tuning the Final Alignment

At the beginning, we use Clustal W version 1.7 to help generate temporary alignments. Later we find that the program does not penalize end spaces. The consequence is that if we use it to align the regions between consecutive blocks in step 3, it generates many spaces at both ends of the regions. However, Clustal V, which is an earlier version of the program, does penalize end spaces. After contacting the authors, we switched to Clustal W version 1.8. The source code is freely available, and after some small modifications in the source code, we can make it perform the penalization.

Chapter 5

Experimental Results

We run our program on some simulated data sets, as well as four real data sets proposed by McClure et al. [25]. In this chapter, we first show some of the experimental results of BMA on simulated data sets, and compare the results with those from Clustal W. Then we report the results of running BMA on the test cases proposed in [25].

5.1 Scoring for Motif

Here we adopt the scoring method that is used in [25]. Instead of using an independent scoring scheme to measure the global "goodness" of the alignments produced by the program, we score the method's ability to detect each motif in a data set. A score for a motif is the percentage of the number of sequences in the data set for which the motif is correctly identified. In the

case that a motif is found in more than one subset of the sequences that are not aligned together to produce a single multiple alignment of all the input sequences, the total percent correct match is a combined score of the aligned subsets, allowing full credit for motif identification in each subset as if the motifs were each aligned correctly throughout the set.

For example, given six input sequences, if a motif is correctly identified only in the first three sequences, then the score for this motif is 50 since $3/6 = 50\%$. If the motif is also found in the last two sequences ($2/6 = 33\%$), but the two subsets of conserved regions are not aligned together, then the combined score for this motif is 83 (50, 33).

5.2 Simulation Test

We use a simulator to generate 6 protein sequences of length within 10% variation of 2000. Ten motifs are inserted into the sequences. We run BMA to see how well it scores on each motif.

The similar regions of a motif are generated as follows. First we randomly choose a string, then modify the string according to a change rate. For example, for a string of length 8, if the change rate is 20%, then $8 \times 20\% = 1.6$ sites will be changed. Each similar region of the motif is generated by randomly changing 1 or 2 sites of the string based on the probability. Such changes

can be substitutions, insertions, or deletions; but in our simulation only substitutions are allowed in protein sequences. The background sequences are generated randomly, which ensures that it is unlikely to have any other motifs besides the ones that we insert intentionally. We use two change rates (10%, 20%) and generate 10 data sets for each. The length of a motif varies from 6 to 8.

We run both BMA and Clustal W on the simulated data. Tables 5.1–5.2 summarize the results with 10% change rate. Each column shows how a program scores for each motif in the particular data set. The last row shows the average score of the program for all the motifs in the data set. We can see that BMA correctly identifies all the motifs in each data set, while the overall average score of Clustal W is about 70. This shows BMA is very effective to find conserved regions in long sequences when the regions are well conserved.

We increase the change rate to 20%, and the results are shown in tables 5.3–5.4. BMA scores very well in data sets 1 and 2, almost all the motifs are identified correctly. However, in data sets 5 and 6, BMA misses some of the motifs completely and finds some in a small subset of the input sequences. This can be attributed to the simulator generating motifs based on probability. If many sites in a motif are changed, the regions forming the motif might not be similar any more. Comparing with the case of 10% change rate, the

Motif	Set 1		Set 2		Set 3		Set 4		Set 5	
	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL
I	100	67	100	67 (33 × 2)	100	67	100	33	100	100
II	100	83	100	33	100	100	100	50	100	0
III	100	100	100	0	100	67 (33 × 2)	100	0	100	0
IV	100	67	100	100	100	33	100	100	100	100 (50 × 2)
V	100	50	100	83 (50, 33)	100	100	100	100	100	100
VI	100	100	100	100	100	100	100	100	100	100
VII	100	83	100	33	100	33	100	33	100	0
VIII	100	100	100	33	100	0	100	100	100	100 (67, 33)
IX	100	67	100	100	100	100	100	83	100	0
X	100	100	100	100	100	83	100	100	100	67
avg score	100	82	100	65	100	68	100	70	100	57

Table 5.1: Simulation test — change rate 10% (part I)

Motif	Set 6		Set 7		Set 8		Set 9		Set 10	
	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL
I	100	83	100	33	100	100	100	33	100	100
II	100	0	100	50	100	100	100	83	100	0
III	100	0	100	0	100	67	100	100	100	100
IV	100	67 (33 × 2)	100	100	100	50	100	100	100	83 (50, 33)
V	100	50	100	83	100	67 (33 × 2)	100	85	100	83 (50, 33)
VI	100	83	100	83	100	100	100	67	100	100
VII	100	50	100	67	100	83	100	67	100	67
VIII	100	0	100	100	100	100	100	100	100	100
IX	100	67	100	83	100	50	100	100	100	50
X	100	100	100	100	100	100	100	83	100	83
avg score	100	50	100	70	100	82	100	82	100	85

Table 5.2: Simulation test — change rate 10% (part II)

overall average score of BMA drops from 100 to 82, while the score of Clustal W drops from 70 to 44.

Table 5.5 shows the run-time parameters that we choose for BMA: lower bound and upper bound specify the length of similar regions; similarity degree controls the thresholds that we use to divide the input sequences into groups and to evaluate similarity between two substrings. In the tests, we use the default value of $K = 0$, which is the number of spaces allowed in a block, $gap\_open = 10$ and $gap\_extension = 0.1$. The last two parameters specify gap open penalty and gap extension penalty that are used to align the regions between two blocks. For both change rates, we use the default settings for Clustal W. On our workstation, running BMA on these simulated data sets takes about 50 minutes each, and requires 40MB–60MB memory.

5.3 Real Data Test

We use four protein families as data sets to test the ability of BMA to reconstruct known biologically informative patterns: the hemoglobin family, the kinase family, the aspartic acid protease family, and the RH region of both the RNA-directed DNA polymerase and the Escherichia coli RH enzyme. The sequence length of the data sets ranges from 100 to 300.

Motif	Set 1		Set 2		Set 3		Set 4		Set 5	
	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL
I	83	0	100	67	100	83	0	33	100	67
II	100	100	100	0	33	100	33	67 (33 × 2)	100	33
III	100	33	100	0	100	33	100	0	67	50
IV	100	33	100	33	100	67 (33 × 2)	100	33	33	33
V	100	33	100	67	100	33	100	33	67	33
VI	100	67	100	33	100	100	100	33	0	0
VII	100	33	100	67	100	100	100	67	33	50
VIII	100	83	100	33	100	83 (50, 33)	100	0	0	33
IX	100	100 (67, 33)	100	0	50	67	100	67	100	50
X	100	83 (50, 33)	100	67	100	83 (50, 33)	33	50	100	100
avg score	98	57	100	37	88	75	77	38	60	45

Table 5.3: Simulation test — change rate 20% (part I)

Motif	Set 6		Set 7		Set 8		Set 9		Set 10	
	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL	BMA	CLUSTAL
I	100	100	100	33	100	67	100	100	100	100
II	100	0	67	83	0	0	0	33	33	67 (33 × 2)
III	0	33	100	83 (50, 33)	100	33	100	0	100	50
IV	0	0	100	83	100	67	100	67 (33 × 2)	67 (33 × 2)	0
V	50	33	100	83	100	33	83	33	100	33
VI	100	33	50	33	100	33	100	0	50	0
VII	100	33	67 (33 × 2)	0	67	0	100	0	100	0
VIII	33	33	67	67	100	33	100	50	100	33
IX	100	0	33	0	100	67	100	0	67	67
X	67	0	100	100	100	50	100	0	100	50
avg score	65	27	78	57	87	38	88	28	82	40

Table 5.4: Simulation test — change rate 20% (part II)

Change Rate (%)	Lower Bound	Upper Bound	Similarity Degree	GAP Open	GAP Extension	K
10	6	8	HIGH	10	0.1	0
20	6	8	NORM	10	0.1	0

Table 5.5: Run-time parameters for BMA

The hemoglobin family has often been used to illustrate the reconstructive ability of a new multiple alignment method. This data set includes α- and β-globins from mammals and birds, myoglobins from mammals, and hemoglobins from insects, plants, and bacteria. There are five motifs defining the globin family. We list the score of BMA on each of the motifs in table 5.6, we also list the results reported by McClure et al. [25] for comparison. BMA correctly identifies all five motifs as blocks appearing in the final alignment. It also finds two other blocks, one before motif I and one between motif II and motif III.

The eukaryotic kinase proteins constitute a large enzymatic family that regulates the most basic of cellular processes. This data set includes serine/threonine, tyrosine, and dual specificity kinases from mammals, birds, fungi, retroviruses, and herpes viruses. The result is listed in table 5.7. Of the eight motifs in this data set, BMA identifies six of them as six blocks. Motif III and motif VIII, which are motifs of length 1, are found by aligning regions between blocks. BMA also identifies two extra blocks, which are not

Program	Motif I (7 residues)	Motif II (5 residues)	Motif III (5 residues)	Motif IV (5 residues)	Motif V (3 residues)
AMULT	100	100	100	100	100
ASSEMBLE	100	92	100	100	100
CLUSTAL V	100	92	100	100	100
DFALIGN	100	100	100	100	100
GENALIGN	92 (67, 25)	100	100	83 (67, 17)	92 (67, 25)
MULTAL	100	92	100	100	100
MACAW	75	92	75	67	67
PIMA	100	92	100	100	100
PRALIGN	67	67 (33, 17 × 2)	75 (33, 25, 17)	67 (33, 17 × 2)	83 (67, 17)
BMA	92 (58, 33)	100	100	100 (83, 17)	100

Table 5.6: Scores for Programs Tested Using Globins

Program	Motif I (6 residues)	Motif II (1 residue)	Motif III (1 residue)	Motif IV (9 residues)	Motif V (3 residues)	Motif VI (3 residues)	Motif VII (8 residues)	Motif VIII (1 residue)
AMULT	100	83	92	100	100	100	100	100
ASSEMBLE	83	58 (33, 25)	83	100	100	100	100	100 (67, 33)
CLUSTAL V	100	92	92 (50, 42)	100	100	100	100	100 (58, 42)
DFALIGN	100	100	100	100	100	100	100	100
GENALIGN	100	75 (42, 33)	83	100	100	100	100 (50 × 2)	92 (67, 25)
MULTAL	100	75 (58, 17)	83 (50, 33)	100	100	100 (58, 42)	100	100
MACAW	67	0	75	100	100	83	100	0
PIMA	100	92	92	100	100	100	100	100
PRALIGN	100	83 (42 × 2)	50 (33, 17)	33	75 (42, 33)	75 (42, 33)	33	33
BMA	100	92	58	100	100	100	100	92

Table 5.7: Scores for Programs Tested Using Kinases

motifs according to McClure et al. [25], but which should be aligned together.

The aspartic acid protease data set includes pepsin from mammals, birds, and fungi and from representative members of the retroid family. The result is shown in table 5.8. In this data set, only two blocks are reported by BMA in the final alignment. These two blocks contain motif I and motif III. Motif II is found by aligning the regions between the two blocks.

Program	Motif I (3 residues)	Motif II (5 residues)	Motif III (3 residues)
AMULT	92	58	83
CLUSTAL V	100	75 (50, 25)	50 (25 × 2)
DFALIGN	100	100 (70, 30)	100
GENALIGN	92	67 (42, 25)	58 (25, 17 × 2)
MULTAL	83	58 (33, 25)	75 (50, 25)
MACAW	100	25	67
PIMA	100	42 (25, 17)	42 (25, 17)
PRALIGN	67 (33 × 2)	34 (17 × 2)	67 (25 × 2, 17)
BMA	100	83 (42 × 2)	83

Table 5.8: Scores for Programs Tested Using Proteases

The fourth data set includes sequences from *E. coli* and representative members of the retroid family, including retrovirusese, caulimoviruses, hepadnaviruses, retrotransposons, retroposons, and group II plasmids of filamentous ascomycete mitochondria. Table 5.9 compares the results of BMA and the programs tested in [25]. Two blocks are found by BMA in the final alignment, which contain Motif I and motif II. The other two motifs are found by aligning the regions after the second block.

Program	Motif I (3 residues)	Motif II (1 residue)	Motif III (3 residues)	Motif IV (5 residues)
AMULT	92	75 (58, 17)	67 (50, 17)	59 (25, 17 × 2)
CLUSTAL V	100	75	75 (58, 17)	75 (33, 25, 17)
DFALIGN	100	100	83	100
GENALIGN	100 (83, 17)	58	67 (33, 17 × 2)	75 (33, 25, 17)
MULTAL	92 (75, 17)	92 (58, 17 × 2)	75 (50, 25)	83
MACAW	58	42	58	17
PIMA	83	75	67 (33, 17 × 2)	92 (42, 33, 17)
PRALIGN	75	67 (33 × 2)	50 (33, 17)	17
BMA	83	92 (42, 33, 17)	75	67 (42, 25)

Table 5.9: Scores for Programs Tested Using RH

It is hard to say which method is the best. For example, program AMULT [3, 4] scores well in the first two tests, but in the last two tests it scores below average. In general, BMA is one of the best running on the aspartic acid protease data set, and scores above average in the other three data sets.

Finally, in table 5.10 we summarize the run-time parameters that we use in real data tests. In the tests, we use the default setting of $K = 0$, $gap\_open = 10$, and $gap\_extension = 0.1$. We also give the running time and memory size that are needed for each data set on our workstation. The column of total number of blocks indicates how many blocks are found by BMA in each data set, and blocks in final alignment shows how many blocks appear in the corresponding final alignment. From the table we can see that the hemoglobin family requires the largest amount of memory since it uses the option of *low* similarity degree, which generates more similar substrings.

Data Set	Lower Bound	Upper Bound	Similarity Degree	Total Number of Blocks	Blocks in Final alignment	Time (mm:ss)	Memory
Globins	8	9	*low*	2439	7	6:38	281MB
Kinases	7	10	*normal*	3869	8	4:06	45MB
Proteases	3	5	*normal*	1333	2	0:23	7MB
RH	3	7	*normal*	846	2	0:32	3MB

Table 5.10: Summary of Real Data Tests

Chapter 6

Conclusions, Future Work

We discuss three problems, *grouping*, *blocking*, and *chaining*, that arise in blocked multiple sequence alignment. For each problem, practical algorithms are presented.

To solve the problem of *grouping*, we propose a new way to search for closely related regions directly in all input sequence rather than to start from pairwise alignments. Internal spaces, as well as some degree of mismatch, are allowed in this process. We also present a method that takes into account incomplete blocks by dividing the sequences into groups and using different similarity thresholds for sequences in different groups. Recall that in the process of *blocking*, we might have a large number of similar regions, such that generating all the blocks is impractical. We present a method to control the number of blocks by using partial alignments to eliminate some unlikely

scenarios. The best block chain is found by using the classic optimal-path algorithm for directed acyclic graphs. We propose a scheme to calculate weight of nodes and edges in the graph. At the end of *chaining* all the near-optimal block chains are found, and we use those chains to construct final alignments.

BMA is an implementation of this approach. We have tested BMA on both simulated data and real data. Our experimental results indicate that this approach is efficient, particularly on large numbers of long sequences with well-conserved regions.

However, there are a couple of things closely related to extensions and performance improvements that need more consideration:

1. We find that for large inputs with long sequences, BMA spends most of the time in collecting similar regions. We are working on improving the time-efficiency of this process. If the length of center strings ranges from a to b ($a < b$), then a center string might be a substring of one or more of the other center strings. We should be able to use this information to speed up the collecting rather than using these strings independent of each other.

2. We are also working on alternative methods to reduce the number of blocks. A possible approach is to use partial chaining instead of partial

alignments. The idea is that we construct a DAG using the blocks generated so far and use the graph to decide the validity of a partial block by adding it to the graph. If the partial block could contribute to or be "compatible" with the highest scoring chain in the graph, we say it is valid.

3. Currently BMA has to go through all three steps each time it is executed. For example, if we want to find both the longest chain of blocks and the one that maximizes the score function, we have to run BMA twice, although the computations differ only in the third step. If we divide BMA into three programs, one for each step, it would be much easier to reuse the generated data. We could even use other methods to collect similar substrings and store them in a special format, which can be recognized by the program that constructs blocks. Similarly, the generated blocks can be fed into other block chaining programs, such as *chain* [41], to construct global alignments.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*, pages 163–169. Addison-Wesley, Reading, Massachusetts, March 1985.

[2] V. Bafna, E. Lawler, and P. Pevzner. Approximation algorithms for multiple sequence alignment. *Proc. 5th Symp. on Combinatorial Pattern Matching*, 807:43–53, 1994.

[3] G. J. Barton and M. J. E. Sternberg. Evaluation and improvements in automatic alignment of protein sequences. *Protein Eng.*, 1:89–94, 1987.

[4] G. J. Barton and M. J. E. Sternberg. A strategy for the rapid multiple alignment of protein sequences confidence levels from tertiary structure comparisons. *J. Mol. Biol.*, 198:327–337, 1987.

[5] M. Boguski, R. C. Hardison, S. Schwarts, and W. Miller. Analysis of conserved domains and sequence motifs in cellular regulatory proteins

and locus control regions using new software tools for multiple alignment and visualization. *The new Biologist*, 4:247–260, 1992.

[6] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. Technical report, Department of Informatics, University of Bergen, 1995.

[7] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM J. appl. Math.*, 48:1073–1082, 1988.

[8] S. C. Chan, A. K. C. Wong, and D. K. Y. Chiu. A survey of multiple sequence comparison methods. *Bull. Math. Biol.*, 54(4):563–598, 1992.

[9] K.-M. Chao and Webb Miller. Linear-space algorithms that build local alignments from fragments. *Algorithmica*, 1994.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[11] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming. i: linear cost functions. *J. Assoc. Comput. Mach.*, 39: 519–545, 1992.

[12] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic

programming. ii: convex and concave cost functions. *J. Assoc. Comput. Mach.*, 39:546–567, 1992.

[13] D. F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J. Mol. Evol.*, 25:351–360, 1987.

[14] M. L. Fredman. Algorithms for computing evolutionary similarity measures with length independent gap penalties. *Bull. Math. Biol.*, 46: 553–566, 1984.

[15] O. Gotoh. Alignment of three biological sequences with an efficient traceback procedure. *J. Theor. Biol.*, 121:327–337, 1986.

[16] Dan Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bull. Math. Biol.*, 55:141–154, 1993.

[17] Dan Gusfield. *Algorithms on strings, trees, and sequences: Computer Science and Computational Biology.* Cambridge University Press, first edition, 1997.

[18] Nicholas J. Higham. *Handbook of writing for the mathematical sciences.* SIAM, Philadelphia, Pennsylvania, 1993.

[19] Tao Jiang, Lusheng Wang, and E. L. Lawler. Approximation algorithms

for tree alignment with a given phylogeny. *Algorithmica*, 16:302–315, 1996.

[20] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 7:256–278, 1974.

[21] A. Krogh, M. Brown, I. Mian, K. Sjolander, and D. Haussler. Hidden markov models in computational biology: Applications to protein modelling. *J. Mol. Biol.*, 235:1501–1531, 1994.

[22] C. Lawrence, S. Altschul, M. Boguski, J. Liu, A. Neuwald, and J. Wooton. Detecting subtle sequence signals: a gibbs sampling strategy for multiple alignment. *Science*, 262:208–214, 1993.

[23] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu. A tool for multiple sequence alignment. *Proc. Natl. Acad. Sci. USA*, 86:4412–4415, 1989.

[24] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, pages 322–336, 1978.

[25] Marcella A. McClure, Taha K. Vasi, and Walter M. Fitch. Comparative analysis of multiple protein-sequence alignment methods. *Mol. Biol. Evol.*, 11(4):571–592, 1994.

[26] Webb Miller. Building multiple alignments from pairwise alignments. *CABIOS*, 9:169–176, 1993.

[27] Webb Miller, Mark Boguski, Balaji Raghavachari, Zheng Zhang, and Ross C. Hardison. Constructing aligned sequence blocks. *J. Computat. Biol.*, 1(1):51–64, 1994.

[28] Burkhard Morgenstern, Andereas Dress, and Thomas Werner. Multiple dna and protein sequence alignment based on segment-to-segment comparison. *Proc. Natl. Acad. Sci. USA*, 93:12098–12103, October 1996.

[29] M. Murata, S. Richardson, and J. L. Sussman. Simultaneous comparison of three proteinsequences. *Proc. Natl. Acad. Sci. USA*, 82:3073–3077, 1985.

[30] S. B. Needleman and C. D Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. mol. Biol.*, 48:444–453, 1970.

[31] Laxmi Parida, Aris Floratos, and Isidore Rigoutsos. An approximation algorithm for alignment of multiple sequences using motif discovery. *J. Combinatorial Optimization*, 20(7), April 22 1999.

[32] J. Posfai, A. Bhagwat, G. Posfai, and R. Roberts. Predictive motifs

derived from cytosine methyltranserases. *Nucl. Acids Res.*, 17:2421–2435, 1989.

[33] Eric Sobel and Hugo M. Martinez. A multiple sequence alignment program. *Nucl. Acids Res.*, 14(1):363–374, 1986.

[34] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, July 1997.

[35] Julie Thompson, Des Higgins, and Toby Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Necleic Acids Res.*, 22:4673–4680, 1994.

[36] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6: 505–517, 1977.

[37] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *J. Computat. Biol.*, 1(4):337–348, 1994.

[38] Michael S. Waterman. *Consensus patterns in sequences.* CRC Press, Boca Raton, Florida, 1989.

[39] Michael S. Waterman and Robert Jones. Consensus methods for dna

and protein sequence alignment. *Methods in Enzymology*, 183:221–237, 1990.

[40] Zheng Zhang, B. He, and Webb Miller. Local multiple alignment via subgraph enumeration. *Discreet appl. Math.*, 71:337–365, 1996.

[41] Zheng Zhang, Balaji Raghavachari, Ross C. Hardison, and Webb Miller. Chaining multiple-alignment blocks. *J. Computat. Biol*, 1(3):217–226, 1994.