Multiple Sequence Alignment Augmented by Expert User Constraints

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Lingling Jin

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Sequence alignment has become one of the most common tasks in bioinformatics. Most of the existing sequence alignment methods use general scoring schemes. But these alignments are sometimes not completely relevant because they do not necessarily provide the desired information. It would be extremely difficult, if not impossible, to include any possible objective into an algorithm. Our goal is to allow a working biologist to augment a given alignment with additional information based on their knowledge and objectives.

In this thesis, we will formally define constraints and compatible constraint sets for an alignment which require some positions of the sequences to be aligned together. Using this approach, one can align some specific segments such as domains within protein sequences by inputting constraints (the positions of the segments on the sequences), and the algorithm will automatically find an optimal alignment in which the segments are aligned together.

A necessary prerequisite of calculating an alignment is that the constraints inputted be compatible with each other, and we will develop algorithms to check this condition for both pairwise and multiple sequence alignments. The algorithms are based on a depth-first search on a graph that is converted from the constraints and the alignment. We then develop algorithms to perform pairwise and multiple sequence alignments satisfying these compatible constraints.

Using straightforward dynamic programming for pairwise sequence alignment satisfying a compatible constraint set, an optimal alignment corresponds to a path going through the dynamic programming matrix, and as we are only using single-position constraints, a constraint can be represented as a point on the matrix, so a compatible constraint set is a set of points. We try to determine a new path, rather than the original path, that achieves the highest score which goes through all the compatible constraint set points. The path is a concatenation of sub-paths, so that only the scores in the sub-matrices need to be calculated. This means the time required to get the new path decreases as the number of constraints increases, and it also varies as the positions of the points change. It can be further reduced by using the information from the original alignment, which can offer a significant speed gain.

We then use exact and progressive algorithms to find multiple sequence alignments satisfying a compatible constraint set, which are extensions of pairwise sequence alignments. With exact algorithms for three sequences, where constraints are represented as lines, we discuss a method to force the optimal path to cross the constraint lines. And with progressive algorithms, we use a set of pairwise alignments satisfying compatible constraints to construct multiple sequence alignments progressively. Because they are more complex, we leave some extensions as future work.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Ian McQuillan for his guidance, encouragement, support and constant patience. Words cannot describe my thanks. I am lucky to be your student, you are an excellent supervisor and a good friend.

I would like to give very special thanks to our mom, Jan, a special person to me. I am grateful to have her support and encouragement when I was away from my family. She is my mom in Canada!

I want to express my thanks to my advisory committee: Dr. Tony Kusalik, Dr. Nate Osgood, and my external examiner Dr. Joe Angel for their valuable suggestions and comments.

I also want to thank all the fellow members in the Bioinformatics lab, especially Wayne Clarke, my friend Sheng Wang (Department of Microbiology & Immunology) and Dr. Hong Wang (The Department of Biochemistry) for helping me with a lot of discussions on the motivation of this thesis.

I would like to give thanks to my parents in China, especially my mom. Thank you for your love and support. I am proud to be your daughter.

I would also like to say "thank you" to my husband Shi Shi, an excellent visual designer, for helping draw the fantastic pictures in this thesis for me. He and our lovely baby April make my life full of love and happiness.

Contents

List of Tables

List of Figures

List of Abbreviations

DP	Dynamic Programming
SP	Sum of Pairs
MSA	Multiple Sequence Alignment
CCS	Compatible Constraint Set
DAG	Directed Acyclic Graph
DFS	Depth First Search
LCA	Lowest Common Ancestor
PCR	Polymerase Chain Reaction
NP	Nondeterministic Polynomial

Chapter 1

Introduction

1.1 The Power of Sequence Alignment

Throughout evolution, DNA sequences accumulate substitutions, insertions and deletions, which then cause differences in the RNA and proteins they produce. We can often discover a significant similarity between a new sequence and a sequence about which something is already known. When this occurs, we can sometimes infer information such as structure and/or function for the new sequence, such as the secondary structures of a protein sequence, or the regulatory function of a gene. To evaluate the similarity of two sequences, one typically begins by finding a plausible alignment between them. For this reason, techniques for aligning and comparing sequences have become the most common task in bioinformatics and an important avenue of bioinformatics research.

Sequence alignment is a method for sequence comparison, which allows biologists to reveal similarity between different sequences of DNA, RNA or proteins. The consequences of performing this type of analysis have been significant. One of the first success stories involved establishing a link between cancer-causing genes and normal growth genes [14]. In 1984, scientists used a simple computational technique to compare the newly discovered cancer-causing ν-sis oncogene with all (at the time) known sequenced genes. To their astonishment, the cancer-causing gene matched a normal gene involved in growth and development called platelet-derived growth factor (PDGF). After discovering this similarity, scientists became suspicious that cancer might be caused by a normal growth gene being stimulated at the wrong time, which causes uncontrolled cell growth and leads to cancer.

To perform a sequence alignment is to map the letters of one sequence onto other sequences. As with the example in Figure 1.1, aligned sequences of nucleotides are typically represented as rows within a matrix. Gaps are inserted between the characters using the gap symbol, "-", allowing similar characters to be aligned in successive columns. An alignment between two or more sequences can represent an explicit hypothesis regarding the evolutionary history of those sequences. As a result, comparisons of related sequences have facilitated many recent advances in understanding the information of genetic sequences.

A	T	-	G	T	T	A	T	-
A	T	C	G	G	-	A	-	C

Figure 1.1: An example of sequence alignment. Here the first, second, fourth, and seventh positions align correctly. The fifth position is a mismatch. The third, sixth, eighth and ninth positions are gaps.

1.2 Manual Refinement of Sequence Alignment

While the true evolutionary path followed by a sequence cannot be inferred with certainty, sequence alignment algorithms can be used to identify alignments with a low probability of similarity by chance. However, it is a common practice for expert users to manually adjust alignments after running some automatic alignment algorithms. Then the alignments can more accurately reflect a researcher's knowledge, the truth, or can be tailored towards their particular objectives. In this thesis, we are interested in studying efficient algorithms to manually refine alignments.

1.3 Layout of the Thesis

In the thesis, we will mainly address the problem of how to combine additional information from expert users with existing alignments.

In Chapter 1, we give a brief introduction. In Chapter 2, we provide the motivation and objectives of the work in the thesis. Chapter 3 gives preliminaries and background knowledge necessary to understand our contributions. Chapter 4 gives the formal definition of constraints (additional information from specialists) and algorithms to determine whether a set of constraints is consistent. Chapter 5 and Chapter 6 provide the methods/algorithms to perform pairwise and multiple sequence alignments satisfying a compatible constraint set. And Chapter 7 gives the conclusion of this thesis and a list of future works.

Chapter 2

Motivation and Objectives

In this chapter, we will talk about the motivations of manually refining alignments, and our main goals that we seek to accomplish in the thesis.

2.1 Necessity of Sequence Alignment Adjustments

Most sequence alignment algorithms are designed to find either optimal or close-to-optimal alignments according to some scoring scheme. However, a maximum scoring alignment can only hope to approximate the truth. From a biologist's point of view, an optimal alignment and score do not necessarily provide them with the desired information because these scores do not always reflect the significant biological events contributing to the evolution of each portion of the sequences. Although some algorithms can even guarantee "optimal alignments" in terms of achieving the highest possible score, they might not be tailored for the objective of the user. For example, even the best multiple sequence alignment methods only achieve $< 50\%$ accuracy per position in alignment of sequences with $< 20\%$ identity [32]. As a consequence, all alignments require inspection and interpretation, and often adjustment by hand, in order to produce an alignment that best represents the biological context of the sequences [36]. We will elaborate on this point in the next section.

Biologists can often use other information about the sequence or structure of a family of proteins to improve a multiple sequence alignment. Therefore, an important aspect of constructing accurate alignments is to allow a working biologist to augment a given alignment with additional information based on their knowledge and objectives.

For example, some sequences have specific segments which contribute to the function of the sequences, such as an exon in a DNA sequence or a domain in a protein sequence. Perhaps a particular biologist is only interested in the functional segments, but is less concerned with the other regions of the sequences. The primary task of this kind of alignment is to align the functional segments together, which may not be properly reflected with the scoring scheme. The straightforward global or local alignment cannot guarantee to align these parts together even by changing the scoring or gap penalty schemes. Moreover, it would be extremely difficult, if not impossible, to include any possible objective into such an algorithm. We will discuss four such objectives in

Section 2.2, which shows that they can be quite simple or quite complex based on the particular expert knowledge.

2.2 Four Cases Requiring Manual Adjustment

In this section, we will present some distinct objectives which would benefit from manual refinement.

2.2.1 Case 1

Figure 2.1: Pictured above is a hypothetical manual adjustment to unify several small gaps into one large gap.

A multiple sequence alignment carried out using an automatic alignment algorithm often results in an alignment with several small gaps in the sequences. From a biological point of view, insertions or deletions are more likely to result from a large gap rather than many small adjacent ones. This is because the frequency of gaps of many characters are expected almost as much as gaps of a single character [28]. This problem can be accommodated in an automatic alignment program by using a strategy with a separate gap-open and gap-extension penalty, such as one from [31]. In this scenario, we typically penalize more for a gap opening than extending a gap. This favours longer insertions and deletions than would otherwise be the case with single gaps. Therefore, if one wanted to unify several small gaps into a large one, one could increase the gap opening penalty versus the gap extension penalty. Disadvantages of this approach are that they can negatively impact other parts of the alignment, it is costly in terms of time to recompute entire alignments, and the best balance of parameters can be difficult to estimate. It is also not user-friendly to biologists who are not always experts on the details of the algorithms. Otherwise, one can manually and iteratively adjust the alignment in order to move the small gaps together to form a large one (Figure 2.1). This is often a desirable approach.

Along this same line, one might iteratively change the scoring matrix if one does not know the evolutionary distance between the sequences to be aligned. A *scoring matrix* is a matrix of score values that describes the rate at which one character in a sequence changes to other character states over time. They are usually seen in the context of amino acid or DNA sequence alignments, where the similarity between sequences depends on their time of divergence and the substitution rates as

represented in the matrix. However, if we are performing a multiple sequence alignment between sequences that do not have the same distance between them, one cannot pick an ideal single matrix. Although ClustalW [31] can choose different matrices as the alignment proceeds, depending on the estimated divergence of the sequences to be aligned at each stage, this estimated distance might not be ideal to calculate an accurate alignment. Furthermore, it is likely the default matrix used by algorithms is rarely changed by users.

2.2.2 Case 2

Some biologists desire to align several very related sequences. The resulting alignment sometimes contains gaps in the sequences, but with the gaps not being close in each of the sequences. For the alignment, one might want to focus attention on the subset of columns corresponding to key characters and core structural elements that can be aligned with confidence.

Biologists can produce high quality multiple sequence alignments manually using expert knowledge of sequence evolution such as:

- expected patterns of insertions and deletions that tend to alternate with blocks of conserved sequence;
- the phylogenetic relationships between sequences dictating regions which should and/or should not align in columns and in the patterns of gaps.

As the sequences are closely related, we might expect the gaps in them to be approximately the same length and so should be aligned in columns, so a manual adjustment is required to unify the gaps (Figure 2.2).

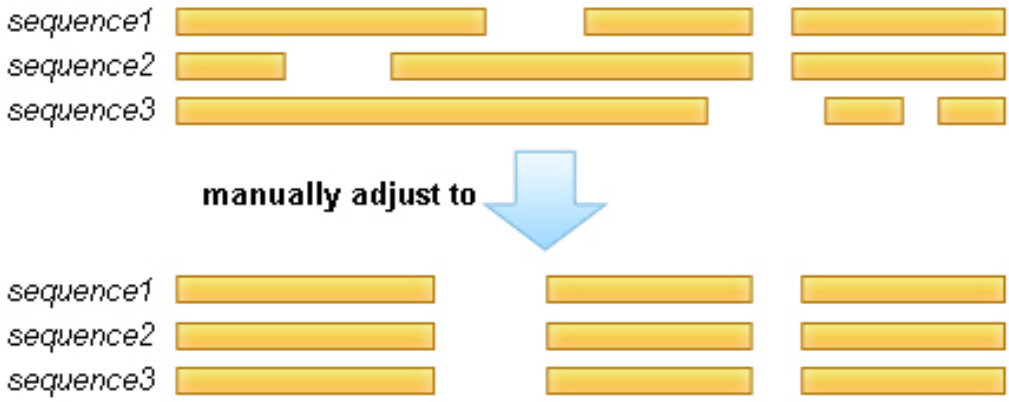


Figure 2.2: This picture demonstrates a manual adjustment in order to move the gaps in homologous sequences into the same columns.

In Figure 2.3, we show some sequences of *Brassica napus* from [7]. Figure 2.3 (a) is the multiple sequence alignment between them using ClustalW with the default parameters. Some parts of the alignment are likely not accurate and require adjustments. We zoom in one part of them in Figure 2.3 (b), in which one can find that the gaps are quite distributed. In order to make a high quality alignment which is biologically meaningful, we need to manually adjust it, in order to better reflect the relationships between the sequences. Figure 2.3 (c) shows an alignment after a manual adjustment. One can see that the characters are aligned better and the gaps are placed into the same or very close columns in each sequence.

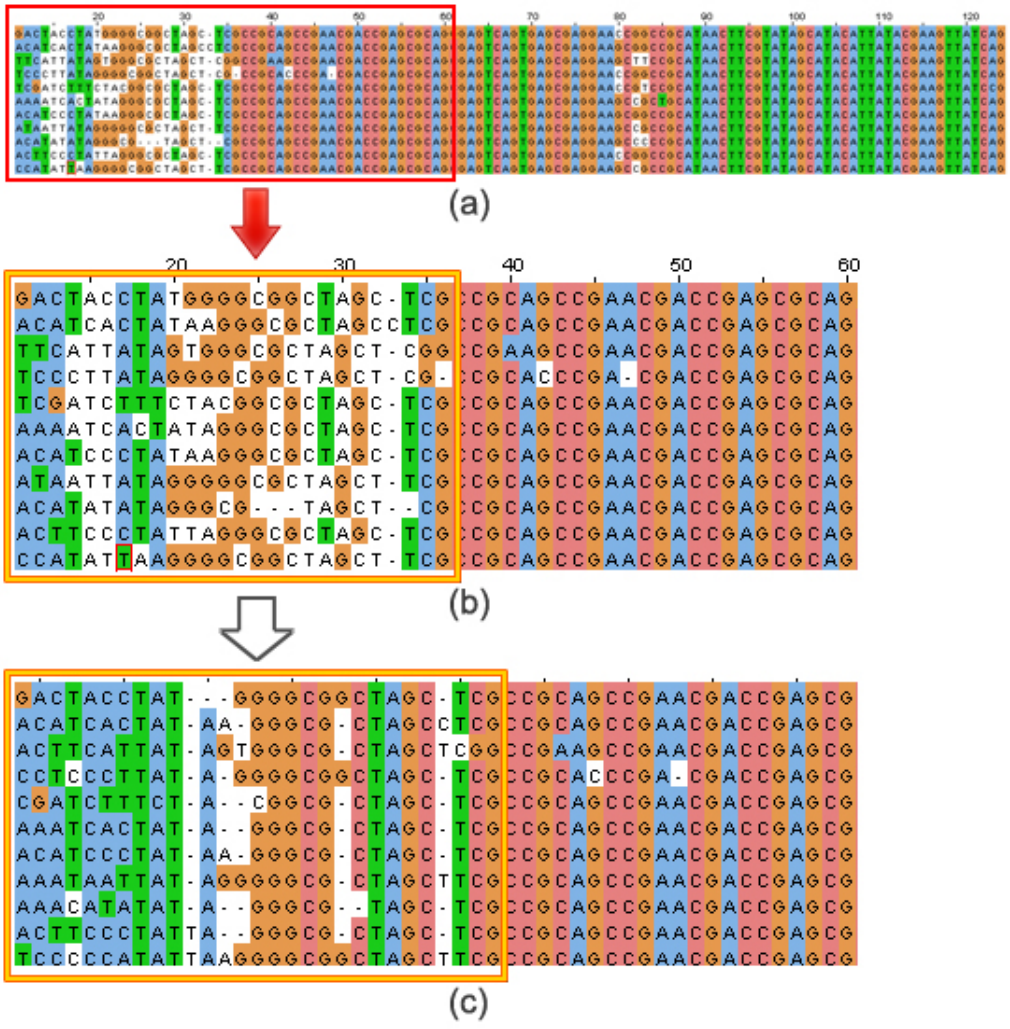


Figure 2.3: Picture (a) demonstrates a multiple alignment of selected *Brassica napus* sequences using ClustalW. Picture (b) is a zoom in of (a) in the region where the gaps in the sequences mostly occur. Picture (c) is the alignment after manual adjustment. One can see that the sequences are aligned better after adjustment. There is a separate color for each nucleotide of the consensus.

We compare the scores of the two segments marked with boxes in Figure 2.3 (b) and (c). Here, we use the *sum-of-pairs function* to score the multiple alignments, in which columns are scored by a "sum of pairs" (SP) function using a scoring matrix. The sum-of-pairs score for a column m_i is defined in [9] as:

$$S(m_i) = \sum_{k<l} s(m_{k,i}, m_{l,i}), \tag{2.1}$$

where k, l are two sequences, and scores $s(a, b)$ come from a scoring matrix. In this case, we gave 1 as the match score and -1 as the mismatch score. Gaps are handled by defining $s(a, -)$ and $s(-, a)$ to be the gap penalty, which is -1 in this case, and $s(-, -)$ to be 0.

Then we can achieve -107 for (b) and 620 for (c). It is hence very likely that a manual adjustment can make the alignment better in these regions.

2.2.3 Case 3

Some biologists would like to study the conservation of sequences which have common domains. To do this work, the common domains need to be aligned together in a multiple alignment, based on which the biologists can then study the relationships of the remaining parts of the sequences. Here is an example from [35] of eight sequences, which all contain a ring domain. The names of the sequences and the position of the ring domains are listed as Table 2.1.

Name of protein sequence	Position of ring domain
AT5G38895	169-209
AT3G02290	181-221
AT5G15790	182-222
AT5G41350	161-201
AT4G00335	139-179
AT4G23450	104-144
AT4G14220 (RHF1A)	46-87
AT5G22000	33-73

Table 2.1: The names of the eight sequences, and the positions of the ring domain.

For a biologist interested primarily in aligning these domains, one would expect a result of aligning domains such as shown in Figure 2.4. But from the alignment of the entire sequences in Figure 2.5, we can easily see that the domains in two sequences marked with boxes are not aligned with the domains in the other four sequences also marked with boxes using ClustalX [13] (in the example). That is because the position of the domains on AT4G14220 and AT5G22000 are quite distant from the domains on the other four sequences. To align the domains together, it requires

there be a very long gap in AT4G14220 and AT5G22000, which will likely violate the optimality of the entire alignment with a gap penalty. Therefore the algorithm instead aligns them as shown in Figure 2.5.

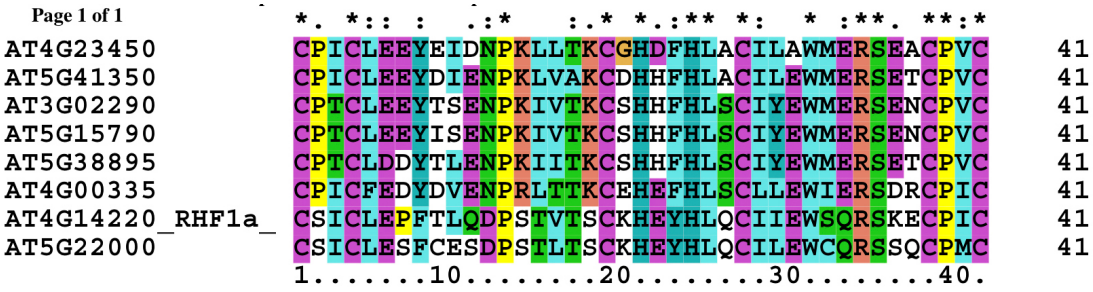


Figure 2.4: Pictured above is the alignment of the ring domain in eight sequences.

In this situation, a manual adjustment is required to move the unaligned domains in order to align them together. Then biologists can study the conservation of the other parts of the sequences.

2.2.4 Case 4

Next, we will provide a more specialized case which serves to illustrate the wide range of objectives for performing a multiple sequence alignment, and how that can change the manual refinement.

Polymerase chain reaction (PCR) is an experimental technique widely used in molecular biology to amplify specific regions of a DNA strand (the DNA target), which can be a gene, part of a gene, or a non-coding sequence. As PCR progresses, the DNA generated is itself used as a template for replication that sets in motion a chain reaction in which the DNA template is exponentially amplified. With PCR, it is possible to amplify a single or few copies of a DNA fragment across several orders of magnitude, generating millions or more copies of the DNA. A primer is a strand of nucleic acid that serves as a starting point for DNA replication, and is required for initiation of DNA synthesis. The selectivity of PCR stems from the use of primers that are complementary to the DNA region targeted for amplification under specific thermal cycling conditions [3].

Family specific PCR is a technique used when PCR primers amplify an entire gene family, which is a group of closely related genes that encode similar products and have descended from the same ancestral gene. If one wants to sequence introns which are conserved in the gene, designing family specific PCR primers that span these conserved regions is a process that currently requires a lot of human participation [7]. In this process, researchers can align small primers with the reference sequence at the correct positions at the boundaries of exons. The reason to align primers with a known reference sequence is because the study of several species from a gene family is of interest, where we expect variations between them and they are slightly different from the reference sequence.

As shown in Figure 2.6 (taken from [7]), we know that $exon2$ and $exon3$ are highly similar in

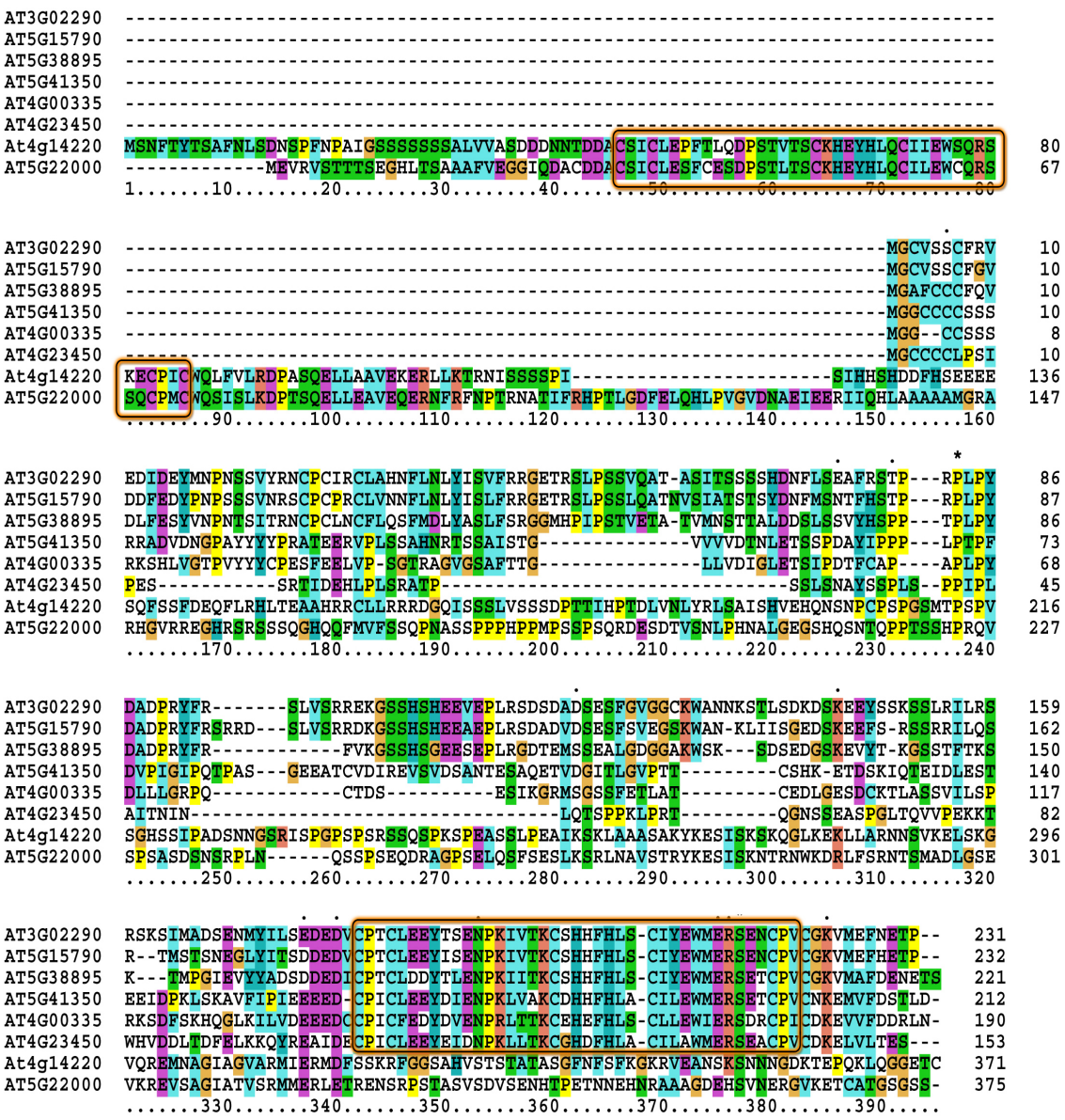


Figure 2.5: Pictured above is an alignment of eight sequences made by ClustalX, and the five different subgraphs show successive regions of the sequences. The unaligned ring domains are marked with boxes.

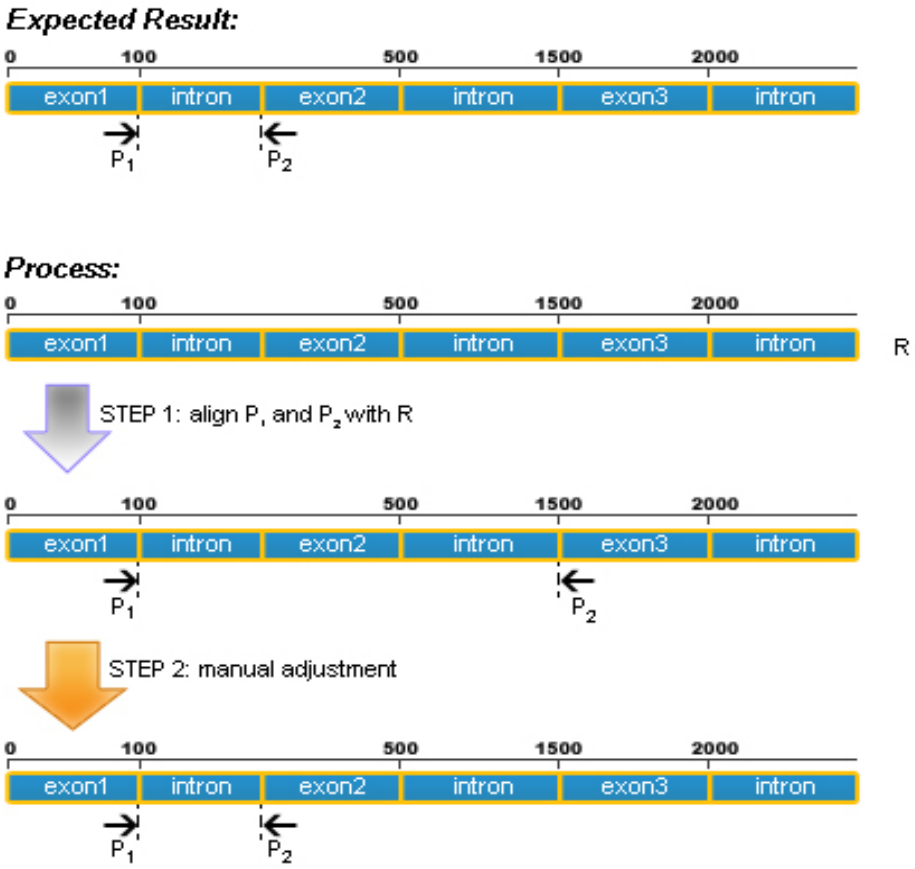


Figure 2.6: Pictured above is a description of a mistake which could occur when aligning small primers with a reference sequence, which requires manual adjustment to move the primer P_2 to its correct position. In step 1, a mistake occurs which aligns P_2 at an incorrect position in $exon3$ with a distance between P_1 and P_2 of more than 100 bp. In step 2, we manually adjust the alignment to place P_2 in $exon2$.

the reference sequence and the distance between them is far bigger than we can consider while designing primers, and the distance between the two primers P_1 and P_2 is already known (less than 100bp). Step 1 is to align the primers P_1 and P_2 with the reference sequence R. We expect that P_1 can be aligned with $exon1$ and P_2 can be aligned with $exon2$. However, there can be mistakes that occur in the alignment in this step, as in Figure 2.6.

As we know that $exon2$ and $exon3$ are similar, mistakes can easily occur which aligns the primer at an incorrect position in $exon3$ with a distance between the two primers of more than 100 bp as shown in step 1 of Figure 2.6. If the distance between P_1 and P_2 is known, we can manually adjust the alignment to place P_2 in $exon2$.

Therefore, a manual adjustment is required to move P_2 to its correct position in step 2 as shown in step 2 in Figure 2.6.

2.3 Our Objectives

The objectives of this thesis can be divided into three categories. The first is to formalize the notion of an alignment constraint, a compatible constraint set and an alignment satisfying a compatible constraint set. The second is to design algorithms to determine whether a set of constraints is compatible in a sequence alignment. This allows us to describe alignment constraints which are consistent with each other. The third goal is to provide algorithms to efficiently compute realignment or alignments based on expert users' manual adjustments. The algorithms should provide a method that gives a simple, flexible, fast and accurate solution to the problem of how to combine additional information to yield an optimal alignment satisfying constraints. We will study theoretically how this can be accomplished starting with different multiple sequence alignment techniques.

Furthermore, in principle, one can even make constraints automatically, not just based on users' input. For example, one can automatically add constraints based on databases of domains, exons, or some certain protein structures.

Chapter 3

Preliminaries

In this chapter, we will introduce some preliminaries, including definitions we will use in the thesis, a survey of sequence alignment algorithms, summaries of some classical algorithms and methods, and a survey of existing algorithms or packages which allow additional user constraints.

3.1 Introduction

We briefly discussed what sequence alignments look like in Chapter 2. Next, we will formally define an alignment.

3.1.1 Formal Definitions

In this section, we will introduce some formal definitions similar to those in [14] in order to mathematically define an alignment, which will be important for the rest of the thesis. The definitions are given in an abstract manner in order to apply to any type of sequences, such as DNA, RNA or protein, and to work precisely in a variety of situations. Furthermore, it is possible to define sequence alignment in terms of standard mathematical constructs. We will also give examples to clarify them.

Definition 1. *An* alphabet Σ *is an abstract and finite set of symbols. Let '$-$' be a new symbol not in Σ, and we let $\Sigma_- = \Sigma \cup \{-\}$. We call '$-$' the* gap symbol.

So, for example, we could consider an alphabet of amino acids, or deoxyribonucleotides, or ribonucleotides depending on whether we are considering proteins, DNA or RNA.

Definition 2. *A* string *is any finite sequence of characters over an alphabet. Let Σ be an alphabet and $s = s_1 s_2 \cdots s_n$ be a string, $s_i \in \Sigma$, $1 \leq i \leq n$. Let j, k satisfy $1 \leq j \leq k \leq n$. Then the* substring *of s which begins at the jth character, and ends at the kth character is $s(j,k) = s_j s_{j+1} \cdots s_k$. Moreover, $s(j) = s_j$, the jth character alone. The* length *of s is denoted by $|s|$, which is the number of characters in the string.*

Definition 3. *If Σ is an alphabet, then Σ^* is* the set of all strings *over the alphabet Σ, which includes the empty word, λ.*

Definition 4. *Let Σ and Γ be two alphabets. A* string homomorphism *h is a function from Σ^* to Γ^* such that $h(\lambda) = \lambda$ and $h(xa) = h(x)h(a)$ for $x \in \Sigma^*, a \in \Sigma$.*

The homomorphism in the following example maps a DNA string onto the corresponding RNA string.

Example 1. *Let $s = ATTCAATCG$. We can define a homorphism from $\{A, C, T, G\}^*$ to $\{A, C, U, G\}^*$ such that $h(A) = A$, $h(C) = C$, $h(T) = U$, and $h(G) = G$. Then $h(s) = AUUCAAUCG$.*

A string homomorphism is defined entirely in terms of how it acts on individual letters. Therefore, on a string $w = a_1a_2 \cdots a_n, a_i \in \Sigma, 1 \leq i \leq n$, then $h(w) = h(a_1)h(a_2) \cdots h(a_n)$.

Definition 5. *Let Σ be an alphabet. We define* a homomorphism *h_- from Σ_-^* to Σ^* defined by $h(a)$=a for all $a \in \Sigma$ and $h(-) = \lambda$. We also define* two homomorphisms *h_1 and h_2 from $(\Sigma_- \times \Sigma_-)^*$ to ${\Sigma_-}^*$, defined by $h_1((a, b)) = a, h_2((a, b)) = b$, for each $a, b \in {\Sigma_-}^*$.*

Now we have constructed enough definitions in order to define alignments in a way that corresponds to the more pictorial type of alignments with which most bioinformaticians are familiar.

Definition 6. *Let $s = s_1 \cdots s_n$, $t = t_1 \cdots t_m$, $s_i, t_j \in \Sigma$, $1 \leq i \leq n$, $1 \leq j \leq m$. An* alignment *of s and t is a string $x \in (\Sigma_- \times \Sigma_-)^*$, such that $h_-(h_1(x)) = s$, $h_-(h_2(x)) = t$, and there does not exist i, such that $h_1(x(i)) = h_2(x(i)) = -$. We say that a string in $x \in (\Sigma_- \times \Sigma_-)^*$ is an alignment over Σ, if it is an alignment of some strings s and t in Σ^*.*

In order to understand these definitions better, here is an example:

Example 2. *Let us start with two sequences: s=ATTCGA, t=GATAA. One alignment between the two sequences is $x = (-, G)(A, A)(T, T)(T, -)(C, -)(G, A)(A, A)$, as $h_-(h_2(x)) = t$, $h_-(h_1(x)) = s$, and $(-, -)$ is not a character of the alignment. An alignment such as x can be visualized as:*

$$\begin{array}{c} - \ A \ T \ T \ C \ G \ A \\ G \ A \ T - - \ A \ A. \end{array}$$

Then we have:

$h_1(x) = s' = -ATTCGA$, $h_2(x) = t' = GAT--AA$,
$h_-(s') = s = ATTCGA$, $h_-(t') = t = GATAA$.

It is also desirable to be able to align more than two sequences. We need to generalize the definitions for multiple alignments.

Definition 7. *For natural number k, we define* homomorphism *h_i, $1 \leq i \leq k$ from $\overbrace{(\Sigma_- \times \cdots \times \Sigma_-)}^{k}{}^*$ to ${\Sigma_-}^*$, defined by $h_i((a_1, \cdots, a_k)) = a_i$ for each i, $1 \leq i \leq k$.*

Definition 8. *Let*

$$\begin{aligned} s_1 &= s_{1_1} \cdots s_{1_{n_1}}, \\ s_2 &= s_{2_1} \cdots s_{2_{n_1}}, \\ &\cdots, \\ s_k &= s_{k_1} \cdots s_{k_{n_k}}, \end{aligned}$$

$s_{i_j} \in \Sigma, 1 \leq i \leq k, 1 \leq j \leq n_i$. *An* alignment of $s_1, \cdots, s_k$ *is a string* $x \in \overbrace{(\Sigma_- \times \cdots \times \Sigma_-)}^{k}{}^{*}$ *such that* $h_-(h_i(x)) = s_i$, *for each* $i, 1 \leq i \leq k$, *and there does not exist* j *such that* $h_i(x(j)) = -$, *for all* $i, 1 \leq i \leq k$. *We say that a string* $x \in \overbrace{(\Sigma_- \times \cdots \times \Sigma_-)}^{k}{}^{*}$ *is an alignment over* Σ, *if it is an alignment of some strings* $s_1, \cdots, s_k$ *in* Σ^*.

Example 3. *For the alignment* $x = (A, A, A)(C, C, G)(A, -, -)(G, -, -)(T, T, G)(A, C, C)(G, G, G)$, *we get the following visualization:*

$$\begin{aligned} &A\ C\ A\ G\ T\ A\ G \\ &A\ C - -\ T\ C\ G \\ &A\ G - -\ G\ C\ G. \end{aligned}$$

We also develop the idea of a "consensus sequence". The *consensus sequence* of a multiple alignment is informally, a "best" single sequence to represent the alignment.

Definition 9. *A* consensus y *of an alignment* $x \in \Sigma^*$ *is a string, such that* $y(i) = a$, *where* a *occurs at least as many times in* $x(i)$ *as any other letter.*

Example 4. *A consensus for*

$$\begin{aligned} &A\ C\ A\ G\ T\ A\ G \\ &A\ C - -\ T\ C\ G \\ &A\ G - -\ G\ C\ G \end{aligned}$$

is ACAGTCG. Notice that it is possible to have more than one consensus.

3.1.2 Evolutionary Mutations

Throughout evolution, biological sequences show complex patterns of similarity to one another [15]. When we perform a sequence comparison, we are looking for evidence that they have diverged from a common ancestor by a process of mutations which were introduced in one or both lineages in the time since they diverged from one another. The basic mutational processes that are considered are *substitutions*, which change characters in a sequence, and *insertions* and *deletions*, which add or remove characters. Insertions and deletions are together referred to as *gaps*.

As shown in Figure 1.1, columns that contain the same letter in both rows are called *matches*, while columns containing different letters are called *mismatches*. The columns of the alignment containing a *gap* could be the result of an insertion in one sequence or a deletion in the other.

3.1.3 The Scoring Scheme

There are many possible alignments of strings, but we would like to find the best possible one. A *scoring scheme* is commonly used which assigns a number to each alignment. A simple scheme frequently used is to have fixed scores associated with matches, mismatches and gaps, and then to assign the score of an alignment to be the sum of the appropriate terms for each aligned pair of characters, plus terms for the gaps. Informally, if sequences are homologous, then we expect identities and conservative substitutions to be more likely in alignments than we expect by chance, and so to contribute positive score terms; and non-conservative changes are expected to be observed less frequently in real alignments than we expect by chance, and so these contribute negative score terms [9]. The alignment shown in Figure 1.1 has four matches, one mismatches, and four gaps. The number of matches plus the number of mismatches plus the number of gaps is equal to the length of the alignment matrix and must be smaller than or equal to the sum of the lengths of the sequences.

Using an additive scoring scheme corresponds to an assumption that we can consider mutations at different sites in a sequence to have occurred independently (at least from the perspective of creating equal scores). A *gap penalty* is a negative score, and we call it penalty, because we add the negative scores. There are several ways to assign gap penalties. One technique is to use *constant gap penalties*. In this case, only one parameter, d, is added to the alignment score when the gap is first opened. This means that every gap, no matter what its size is, receives the same penalty. Another technique is to use *linear gap penalties*. Here we only have one parameter, d, which is a penalty per unit length of gap. This is almost always negative, so that the alignment with fewer gaps is favoured over the alignment with more gaps. Under a linear gap penalty, the overall penalty for one large gap is the same as for many small gaps. However, in biological sequences, it is more likely to have one gap of n characters from a single insertion or deletion event, than it is to have n gaps of single character [28]. An *affine gap penalty* is a technique which attempts to overcome this problem. Affine gap penalties are length dependent and use a gap opening penalty, o, and a gap extension penalty, e. A gap of length l is then given a penalty $o + (l - 1)e$. So that gaps are discouraged, o and e are almost always negative. Furthermore, because a few large gaps are better than many small gaps, e is almost always bigger than o to encourage gap extension rather than gap introduction (that is, the absolute value of e is smaller).

3.2 Dynamic Programming and Optimal Alignment

Finding the optimal alignment between two sequences can be a computationally complex task, as there are a huge number of possible alignments of even two sequences. Fortunately, a technique called *dynamic programming* makes sequence alignment more tractable. Some algorithms break a problem into smaller subproblems and use the solutions of the subproblems to construct solutions to large ones. During this process, the number of subproblems may become very large, and some algorithms solve the same subproblem repeatedly, needlessly increasing the running time. Dynamic programming organizes computations to avoid recomputing values that have already been determined, which can often save a great deal of time [15]. It provides a framework for understanding DNA sequence comparison algorithms.

The simplest dynamic programming alignment algorithms to understand are pairwise sequence alignment algorithms. Many dynamic programming algorithms are guaranteed to find the optimal scoring alignment or set of alignments. Using the scoring schemes we have discussed, better alignments will have higher scores, so we want to maximize the score to find the optimal alignment. In the following subsections we will introduce two algorithms. Although the details of the algorithms are slightly different across their application, they are both based on dynamic programming. They will also form the basis for new methods in the thesis.

3.2.1 Global Alignment: the Needleman-Wunsch Algorithm

The global alignment is the alignment of entire sequences. The first problem we consider is that of obtaining the optimal global alignment between two entire sequences. The dynamic programming algorithm for solving this problem is known in biological sequence analysis as the Needleman-Wunsch algorithm [21].

We use a parameter d for the score when a character is aligned with a gap, which is negative. We also need score terms for each aligned character pair, and for that purpose, we use a *substitution matrix* s, which gives a score associated with substituting one character in a sequence with other characters over time. Substitution matrices are usually seen in the context of amino acid or DNA sequence alignments, where the similarity between sequences depends on their time since divergence and the substitution rates as represented in the two-dimensional matrix. For amino acids, a substitution matrix assigns scores or frequencies to the alignment of each possible pair of amino acids, usually based on the similarity of the amino acids' chemical properties and/or the evolutionary probability of the mutation. While for nucleotide sequences, typically a much simpler substitution matrix is used, where only identical matches and mismatches are considered.

The idea is to build up an optimal alignment using previous solutions for optimal alignments of smaller subsequences. Let $x = x_1 \cdots x_m$, $y = y_1 \cdots y_n$, where x_i, y_i are individual characters.

We construct a two-dimensional dynamic programming matrix F indexed by i and j, one index for each sequence, where the value $F(i,j)$ is the score of the best alignment between the initial segment $x_1x_2\cdots x_i$ and the initial segment $y_1y_2\cdots y_j$. We can build $F(i,j)$ iteratively. We begin by initializing $F(0,0) = 0$. We then proceed to fill the matrix from top-left to bottom-right. If $F(i-1,j-1), F(i-1,j)$ and $F(i,j-1)$ are known, it is possible to calculate $F(i,j)$. There are three possible ways that the best score $F(i,j)$ of an alignment up to x_i, y_j could be obtained (supposing a linear gap penalty): x_i could be aligned to y_j, in which case $F(i,j) = F(i-1,j-1) + s(x_i,y_j)$; or x_i is aligned to a gap, in which case $F(i,j) = F(i-1,j) + d$; or y_j is aligned to a gap, in which case $F(i,j) = F(i,j-1) + d$ as in Figure 3.1. The best score up to (i,j) will be the largest of these three options, assuming that the three scores in each has been properly calculated, which can be proven inductively [9]. And we must deal with some boundary conditions. Along the top row, where $j = 0$, the values $F(i,j-1)$ and $F(i-1,j-1)$ are not defined so the values $F(i,0)$ must be handled specially. The values $F(i,0)$ represent alignments of a prefix of x to all gaps in y, so we can define $F(i,0) = id$. Likewise, down the left column we set $F(0,j) = jd$.

Therefore, we have the following recurrence:

$$F(i,j) = max \begin{cases} F(i-1,j) + d, & \text{for } i > 0, \\ F(i,j-1) + d, & \text{for } j > 0, \\ F(i-1,j-1) + s(x_i,y_j), & \text{for } i > 0 \text{ and } j > 0. \end{cases} \tag{3.1}$$

This equation is applied repeatedly to fill in the matrix of $F(i,j)$ values, calculating the value in the bottom-right corner of each square of four cells from one of the other three values (above-left, left, or above) as in Figure 3.1.

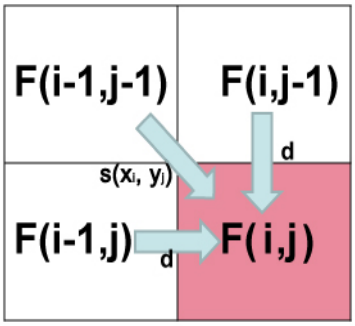


Figure 3.1: The dynamic programming computation scheme which calculates the value in the bottom-right corner of each square of four cells from one of the other three values (above-left, left, or above).

As we fill in the $F(i,j)$ values, we can also keep an arrow (or some entity, such as an integer representing them) in each cell, back to the cell from which its $F(i,j)$ was derived, as shown in the example of the full dynamic programming matrix in Figure 3.2. This arrow describes the last position of the best alignment of $x_1x_2\cdots x_i$ with $y_1y_2\cdots y_j$.

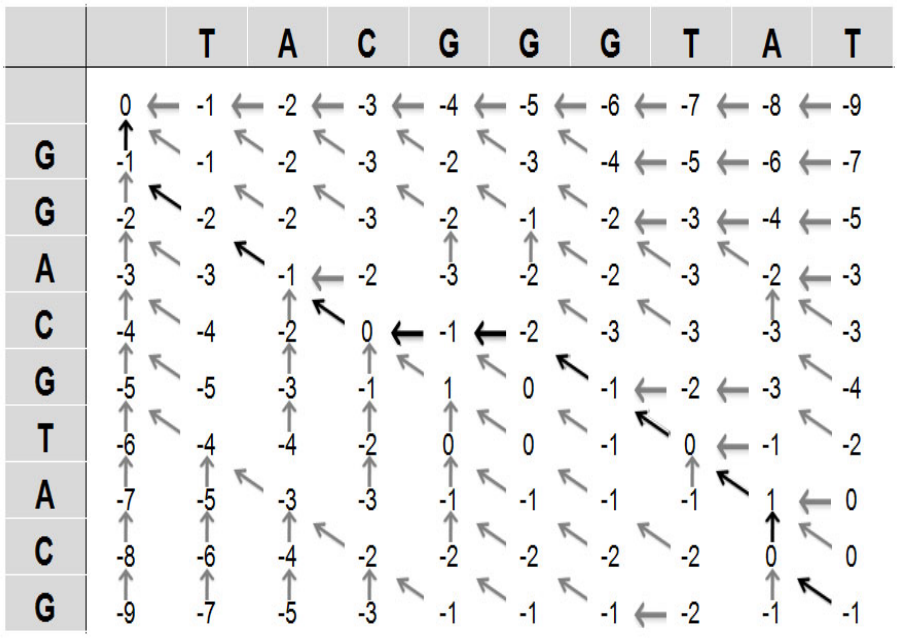


Figure 3.2: Pictured above is the dynamic programming matrix of the Needleman-Wunsch Algorithm.

The value in the final cell of the matrix, $F(n, m)$, is by induction the best score for an alignment of $x_1x_2\cdots x_n$ to $y_1y_2\cdots y_m$, which is what we want: the score of the best global alignment of x and y. To find the alignment itself, we must backtrack the dynamic programming matrix using some sequences of arrows starting at (n, m) to $(0, 0)$ to completely find a path of choices that led to this final value. Indeed, when taking the maximum value in Equation 3.1, we are determining the possibility for the last position of an optimal alignment of $x_1x_2\cdots x_i$ with $y_1y_2\cdots y_j$, and thus by continuing iteratively, we can determine one or every optimal alignment of x with y backtracking in this manner.

3.2.2 Local Alignment: the Smith-Waterman Algorithm

Thus far, we know how to find the best match between sequences from one end to the other. However, a common situation occurs when we are looking for the best alignment between subsequences of x and y (as opposed to entire sequences). This type of alignment would be desired, for example, when it is suspected that two protein sequences share a common domain, or when comparing extended sections of genomic DNA sequence. It is also usually the most sensitive way to detect similarity when comparing two very highly diverged sequences, even when they may have a shared evolutionary origin along their entire length [29]. The highest scoring alignment of subsequences of x and y is called the *best local alignment*, and the algorithm to find the best local alignment is called the *Smith-Waterman algorithm* [29].

The algorithm for finding an optimal local alignment is closely related to that described for global alignments, but there are two differences. First, in each cell in the table, an extra possibility is added to the equation, allowing $F(i, j)$ to take the value 0 if all other options have values less than 0. Taking the 0 option corresponds to starting a new alignment.

$$F(i,j) = max \begin{cases} 0, & \text{for } i \geq 0 \text{ or } j \geq 0 \ , \\ F(i-1,j) + d, & \text{for } i > 0 \ \ , \\ F(i,j-1) + d, & \text{for } j > 0 \ , \\ F(i-1,j-1) + s(x_i, y_j), & \text{for } i > 0 \text{ and } j > 0 \ . \end{cases} \tag{3.2}$$

The second change is that now an alignment can end anywhere in the matrix, so instead of taking the value in the bottom right corner, $F(n, m)$, for the best score, we look for the highest value, $\max_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}} F(i, j)$, over the whole matrix, and start the traceback from there. The traceback ends when we hit a cell with value 0, which corresponds to the start of the alignment. A comparison of the best global versus local alignment path is shown in Figure 3.3 [14].

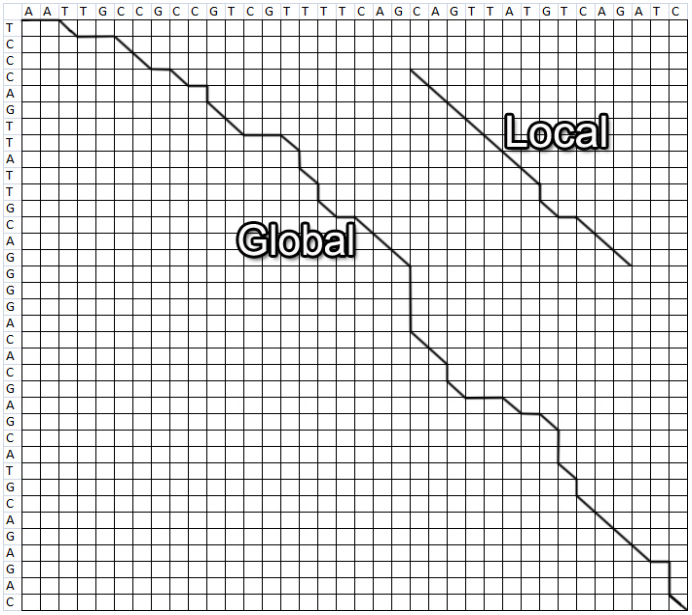


Figure 3.3: The lower line represents a potential global alignment between the two sequences on the axes and the upper line represents a potential local alignment.

3.3 Multiple Sequence Alignment

> "One amino acid sequence plays coy; a pair of homologous sequences whisper; many aligned sequences shout out loud." [17]

In nature, even a single sequence contains all the information necessary to dictate the structure of the protein. Through a pairwise sequence alignment, we can infer the similarity level between two

sequences, matches and mismatches among the alignment, gaps and conserved characters. However, such a comparison fails to give much insight on evolutionary relationships. Multiple alignments, a natural extension of two-sequence comparisons, are a powerful way to study biological sequences. In a multiple sequence alignment, homologous characters among a set of sequences are aligned together in columns. "*Homologous*" has come to mean corresponding or similar position, structure, function, or characteristics due to evolutionary relatedness. Sequences can be aligned to visualize the effect of evolution across the whole family. Ideally, a column of aligned characters all diverge from a common ancestor.

3.3.1 Multi-dimensional Dynamic Programming

From the Needleman-Wunsch and Smith-Waterman Algorithms we can see how to find the global or local optimal alignment of two sequences. It is possible to generalize pairwise dynamic programming alignment to alignments of n sequences. In Section 3.2, we calculated the optimal alignment of two sequences using a two-dimensional matrix. Similarly, the optimal alignment of three sequences can be calculated using a cube (or a three-dimensional matrix) as shown in Figure 3.4, where the sequences to be aligned are marked as sequence 1, sequence 2 and sequence 3. The choice of path which corresponds to the optimal alignment goes through the cube from the corner at position A to the opposite one at position B. The corner that corresponds to the first character of all sequences is called the *original corner* (position A in Figure 3.4), and the corner corresponding to the last character of all sequences is called the *end corner* (position B in Figure 3.4).

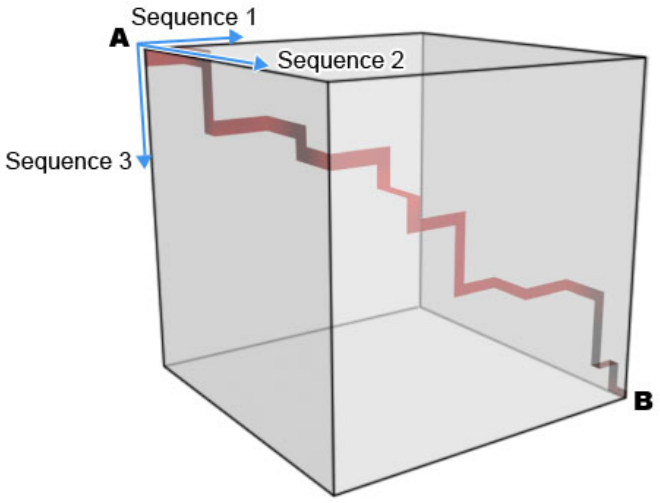


Figure 3.4: 3-dimensional dynamic programming to calculate the alignment of three sequences.

To calculate the optimal alignment (optimal path) of n sequences $s_1, s_2, \cdots, s_n$, we need an n-dimensional matrix. Consider the n-dimensional matrix $M(s_1, s_2, \cdots, s_n)$. Each position in M

may be thought of as the end corner of the sub-matrix $M(s_1(1,i_1),\cdots,s_n(1,i_n))$, where for each sequence s, $s(1,i)$ denotes the subsequence consisting of the first i characters of s as in Definition 2. The idea of the method is to iteratively find optimal paths for all these sub-matrices. We compute the maximum score needed to reach each position through a valid path. This process is repeated until we calculate the maximum score needed to reach the end corner of $M(s_1,s_2,\cdots,s_n)$. Together with the optimal score of each position of the matrix, we keep an arrow to trace back an optimal path.

Performing a multiple sequence alignment with dynamic programming involves calculation in a multi-dimensional matrix and it is computationally complex to produce. The search space increases exponentially with increasing n and is dependent on sequence length. That is, the time complexity is $O(|s_1| \times |s_2| \times \cdots \times |s_n|)$, where $s_1, s_2 \cdots s_n$ are the sequences. To find the global optimum for n sequences this way has been shown to be NP-complete where the number of sequences varies [34], unless we take n as constant in which case it can be performed in polynomial time. Nevertheless, the usefulness of these alignments in bioinformatics has led to the development of a variety of methods suitable for aligning three or more sequences that are more efficient but are not guaranteed to produce optimal results. The field of multiple sequence alignments has undergone drastic changes with the introduction of several new algorithms and new evaluation methods [22, 33, 11, 23].

3.3.2 Classification of Multiple Sequence Alignment Algorithms

Considering the most common strategies, it is convenient to classify existing algorithms into three main categories: exact algorithms, iterative alignments and progressive alignments [22]. The iterative and progressive algorithms are heuristic methods.

Exact Algorithms

Exact algorithms attempt to simultaneously align multiple sequences and find an optimal alignment given the scoring scheme. This would be especially useful when dealing with sets of extremely divergent sequences whose pairwise alignments are all likely to be incorrect [22]. This was achieved in the MSA program, an implementation of the Carrillo and Lipman algorithm [5]. The MSA program [18] optimizes the sum of all of the pairs of characters at each position in the alignment (the so-called sum-of-pairs score), and used a branch-and-bound technique to make small multiple alignments more practical in a reasonable amount of time. It was restricted to small sets of moderately similar sequences.

The exact algorithm has high memory and computational time requirements, and can only handle a small number of sequences. Hence it is still impractical for many multiple sequence alignment applications that require the simultaneous alignment of many sequences.

Iterative Algorithms

Iterative algorithms provide a heuristic technique to perform a multiple sequence alignment. It produces an alignment and refines it through a series of iterations until no more improvements can be made. Iteration has been successfully used and is a key optimization technique for multiple sequence alignment. Iterative algorithms do not provide any guarantees about finding optimal solutions but the alignments are reasonably robust and much less sensitive to the number of sequences to be aligned. They are based on the idea that the solution to a given problem can be computed by modifying an already existing sub-optimal solution. Each modification step is an iteration.

Iterative methods can be deterministic or stochastic, depending on the strategy used to improve the alignment. The simpler iterative strategies are deterministic. Traditional stochastic iterative methods include simulated annealing and genetic algorithms. Bioinformatics packages using iterative algorithms are PRRP[22], DIALIGN[20], MUSCLE[10], etc.

Progressive Algorithms

Probably the most commonly used approach to multiple sequence alignment is progressive alignment. The progressive algorithm provides an approximate solution, which can provide an alignment but cannot guarantee that the score is maximal. It is a fast and effective technique to multiply align a set of sequences with feasible time and space, and in many cases the resulting alignments are reasonable.

Generally, there are two steps to perform a progressive algorithm:

1. Build a guide tree.

   The most important heuristic of progressive alignment algorithms is to align the most similar pairs of sequences first, which form the most reliable initial alignments. Most algorithms build a "guide tree", which is determined by a clustering method such as the neighbour-joining method, and may be calculated using distances based on the number of identical two letter sub-sequences [31]. It is a binary tree whose leaves represent sequences and whose interior nodes represent alignments. The root node represents a complete multiple alignment. The nodes together on the same subtree are the most similar pairs.

2. Progressively align sequences following the order of the guide tree.

   The method depends on a progressive assembly of the multiple alignments where sequences or alignments are added one by one so that there are never more than two simultaneous alignments using dynamic programming. Initially, two sequences are chosen from the guide tree and aligned by standard pairwise alignment; this alignment is fixed. Then, a third sequence is chosen and aligned to the consensus sequence (or aligned to an alignment) of the first alignment, and this process is iterated until all sequences have been aligned.

This approach has the advantage of speed and simplicity combined with reasonable sensitivity. Although successful in a wide variety of cases, this method suffers from its "greediness". Errors made in the first alignments cannot be rectified later as the rest of the sequences are added in.

The most widely used multiple sequence alignment packages are based on an implementation of this algorithm. For example, ClustalW [31], which attempts to optimize the weighted sum-of-pairs with affine gap penalties, is a straightforward progressive alignment strategy where sequences are added to the multiple alignment according to the order indicated by a pre-computed dendrogram. In general, ClustalW performs better when the phylogenetic tree is relatively dense without any obvious outlier. T-Coffee [24] is an improvement to the progressive alignment algorithm where sequences are aligned in a progressive manner but using a strategy to minimize potential errors, especially in the early stages of the alignment.

3.4 Existing Algorithms or Packages for Alignment with Constraints

There are some existing algorithms or packages allowing users to add some additional manual information in order to refine alignments. One can attempt to modify alignments using additional constraints, based on different types of expert knowledge. But they have drawbacks in different respects. In this section, we will briefly introduce them.

In order to get more accurate alignments, some of the alignment algorithms tried to incorporate additional biological information. PROMALS [26] improves alignment quality by combining several advanced techniques, such as database searching for additional homologs, secondary structure prediction and probabilistic consistency of profile-to-profile comparisons. Further improvements to PROMALS alignment quality arise from using constraints on the regions that should be aligned. Such constraints can be defined by structural superposition or other additional expert knowledge. The resulting program, PROMALS3D [27], brings together sequence and structure-based alignment to generate multiple alignments consistent with both sequence and structural information. In PROMALS3D, for input sequences, they used similarity searches to retrieve homologs with available 3D structures. The structure-based alignments among these homologs help define high-quality constraints that are combined with sequence-based profile-to-profile alignments enhanced by predicted secondary structures. However, as these "high-quality constraints" are derived from available protein 3D structures, and the process of search, retrieval and defining constraints can be influenced by many other factors, it may have problems. The program does not allow expert interaction to correct any problems which may arise while running the program.

Jalview [6, 36] is a multiple alignment editor written in Java. One of its functions is editing alignments. Using Jalview, gaps can be inserted/deleted, insertion or deletion of gaps in groups of

sequences can be performed, and gapped columns can be removed using the mouse or keyboard. However, all these modifications on alignments are made in the form of editing an output figure, without changing results algorithmically based on the data used to generate the figure. It does not recompute the other parts of the alignment, even though the modifications could change the alignments' optimality.

CLC Sequence Viewer [4] gives options of identifying specific positions in sequences and alignments which should be forced to align to each other and realigning a selection of an alignment. It is possible to introduce an alignment *fixpoint* in a sequence or alignment. When aligning two sequences with alignment fixpoints, the fixpoint regions will then be forced to align with each other. One example would be three sequences A, B and C where sequences A and B have one copy of a domain while sequence C has two copies of the domain. Then you can force sequence A to align to the first copy and sequence B to align to the second copy of the domains in sequence C. This is a more flexible way than the use of constraints in PROMALS3D, while Jalview, it does algorithmically change the remaining parts of the alignment.

However, the package is a commercial product. Also, the entire alignment is recomputed and refreshed instead of only refreshing the sections that have changed, which is less effective computationally. This solution is not ideal, Teal and Rudnicky [30] found that a delay in response time caused users to change their strategies in how they used tools. Therefore, recomputing entire alignments can harm interaction as well.

Thus, the question remains as to how an alignment can be dynamically recomputed in a fluid and fast manner.

Chapter 4

Compatible Constraint Set

4.1 What is a Constraint?

In Chapter 2, we discussed some types of additional information that biologists might incorporate in various scenarios while running automatic alignment programs, in order to find an alignment better satisfying their objectives. In this chapter, we will introduce the definition of a constraint and the relationships between constraints in order to accommodate these and other scenarios.

Many types of additional information can be represented as a kind of constraint. Here, we will first describe constraints informally, with an example. The SEC homology 3 domain (SH3 domain) is a small protein domain of about 60 amino acid residues. It is usually found in proteins that interact with other proteins and mediate assembly of specific protein complexes, typically via binding to proline-rich peptides with their respective binding partner [25]. For those studying the interactions between proteins, a natural objective of the sequence alignment would be to align the SH3 domains together. This requirement can be captured as a constraint on the alignment of sequences that contain the domain.

As shown in Figure 4.1 (a), CRKL and GRB2 are two sequences where CRKL has one copy of the SH3 domain while GRB2 has two copies of the SH3 domains. A constraint on the alignment can be used to force CRKL to align to the second copy (Figure 4.1 (b)) or to the first copy (Figure 4.1 (c)) of SH3 domains in GRB2.

4.1.1 Definition of a Constraint

From the example in Figure 4.1, we developed some intuitive notions for constraints, which we will now define formally. Some definitions from Chapter 3 will be used within these new definitions.

Definition 10. *Let $s_1, s_2, \cdots, s_m, m > 1$ be strings. A* constraint *is a pair $\{(k,i),(l,j)\}$ between strings s_k and s_l, where $l \neq k$, $1 \leq k,l \leq m$, $1 \leq i \leq |s_k|$, $1 \leq j \leq |s_l|$. Here, k and l are called the* sequence numbers*, and i and j are called the* position numbers*.*

A constraint can also be visualized as in Figure 4.2. Essentially, it is a restriction enforcing that character of one sequence must align with one character on another. This is a general type

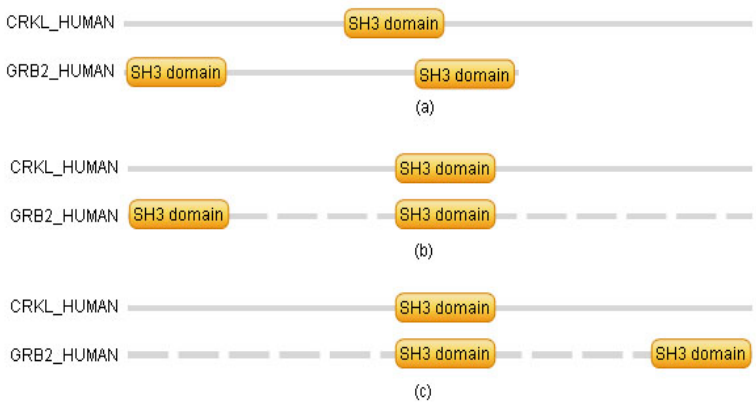


Figure 4.1: (a) two sequences containing SH3 domains: CRKL contains one copy and GRB2 contains two copies; (b) a constraint to align the SH3 domain in CRKL with the second copy of SH3 domain in GRB2; (c) a constraint to align the SH3 domain in CRKL with the first copy of SH3 domain in GRB2.

of constraint and most additional information can be abstracted to this form. But there are some other types of constraints on alignments that can also be treated computationally, but we will not study here. For example, some expert users might not know the exact positions in the sequences that should be forced to align, but would like to align a region in one sequence with a certain region in other sequences.

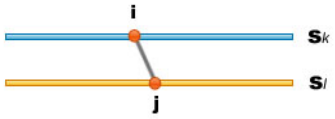


Figure 4.2: One can visualize a constraint between position i of sequence s_k and position j of sequence s_l with a line.

4.1.2 Definition of Compatible Constraints

Using the definition of a constraint, we can now define an alignment satisfying constraints, and the optimal alignment satisfying constraints.

Definition 11. *An* alignment *of m sequences $s_1, s_2, \cdots, s_m$ satisfying n constraints*

$$C_1 = \{(k_1, i_1), (l_1, j_1)\}, C_2 = \{(k_2, i_2), (l_2, j_2)\}, \cdots, C_n = \{(k_n, i_n), (l_n, j_n)\},$$

is any alignment x of $s_1, s_2, \cdots, s_m$, where for each $C_p, 1 \leq p \leq n$, character i_p of s_{k_p} is aligned with character j_p of s_{l_p}.

Definition 12. *Given a scoring scheme δ which assigns a number to each alignment, an* optimal alignment *of $s_1, s_2, \cdots, s_m$ satisfying constraints $C_1, C_2, \cdots, C_n$ is the highest-scoring alignment*

satisfying these constraints. That is, an alignment with score

$\max\{\delta(x) \mid x$ *is a multiple alignment of* $s_1, s_2, \cdots, s_m$ *which satisfies constraints* $C_1, C_2, \cdots, C_n\}$.

We will formalize the notion, and give the general definition of *compatible constraints* as follows.

Definition 13. *Given sequences* $s_1, s_2, \cdots, s_m$ *and constraints* $C_1, C_2, \cdots, C_n$ *between them,* the constraints are compatible *if there exists an alignment satisfying these constraints. If they are compatible, we say that* $\{C_1, C_2, \cdots, C_n\}$ *is a* compatible constraint set*, or CCS for short.*

By this definition, a set of constraints must be compatible in order for an alignment satisfying those constraints to be defined.

4.2 Compatible Constraint Set on Pairwise Sequence Alignment

We will first discuss constraints on pairwise alignments in this section due to their simplicity and importance for certain generalizations to multiple sequences. It is also necessary to introduce them separately because in the next section we will convert the constraints on multiple alignments to this form.

4.2.1 Analysis of Compatible Constraints on Pairwise Sequences

Compatible constraints represent a relationship between constraints. An alignment cannot satisfy constraints which are incompatible within the same aligning process. In order to compute alignments using constraints, we must first make sure the constraints do not conflict. For example, the two constraints $C_1 = \{(k, 10), (l, 20)\}$ and $C_2 = \{(k, 20), (l, 10)\}$ enforces aligning the $10th$ character of sequence s_k with the $20th$ character of sequence s_l, and aligning the $20th$ character of sequence s_k with the $10th$ character of sequence s_l. It is obvious that such an alignment satisfying both C_1 and C_2 does not exist.

In the following sections, we will describe algorithms which decide the compatibility of the constraints. For only two sequences, the situation is relatively simple. This will become a more complicated problem as we generalize the results of this section to multiple sequences.

Before introducing the algorithms, we need to talk about the kinds of constraints that are compatible on two sequences.

As visualized in Figure 4.2 shows, a constraint can be described as a line from a point on one sequence to a point on the other sequence. For example, a constraint $C = \{(k, i), (l, j)\}$ is a line from the ith position of sequence s_k to the jth position of sequence s_l. Then two constraints can be described as two lines. Figure 4.3 gives all relationships between two constraints, $C_1 =$

$\{(k, i_1), (l, j_1)\}, C_2 = \{(k, i_2), (l, j_2)\}$, on two sequences. We can easily find that Figure 4.3 (a) and (b) are compatible constraints (they can co-exist in an alignment), while Figure 4.3 (c) and (d) are conflicting. This is because in Figure 4.3 (c), if we are trying to force i_1 to align with j_1, and we know that $i_2 > i_1$, that would imply that j_2 would necessarily be bigger than j_1, which is not true.

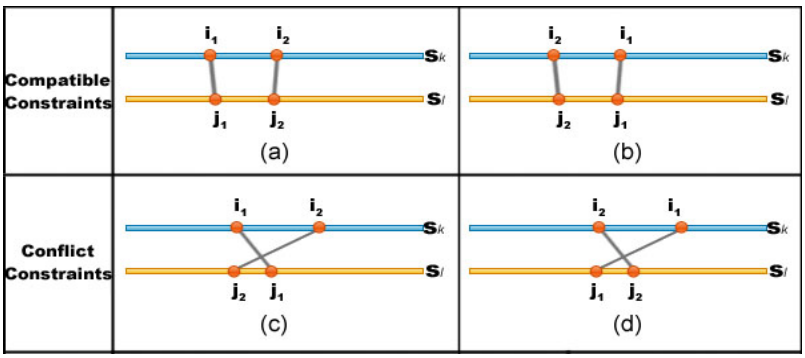


Figure 4.3: In this figure, we draw possible cases of the relationships between two constraints, $C_1 = \{(k, i_1), (l, j_1)\}, C_2 = \{(k, i_2), (l, j_2)\}$, on two sequences s_k and s_l.

Intuitively, we can accept Figure 4.3 (a) and (b) as compatible constraints, and the constraints of the form as depicted in Figure 4.3 (c) and (d) are not compatible, because the lines cross. Visually with only two sequences, two constraints are compatible, if and only if the two lines do not cross with each other or meet at the same point at one sequence.

There is a law to mathematically decide whether two constraints are compatible or not in the situation of only using two sequences.

Proposition 1. *Given two sequences s_k and s_l, and two constraints $C_1 = \{(k, i_1), (l, j_1)\}, C_2 = \{(k, i_2), (l, j_2)\}$ with $C_1 \neq C_2$, C_1 is compatible with C_2, if and only if at least one of the following occurs:*

$$i_2 < i_1 \text{ and } j_2 < j_1,$$

$$i_2 > i_1 \text{ and } j_2 > j_1.$$

Otherwise, C_1 and C_2 are incompatible on the two sequences.

Proof. Assume that C_1 and C_2 are compatible. Then there exists an alignment x, where the i_1th character of sequence s_k is aligned with the j_1th character of sequence s_l; and the i_2th character of sequence s_k is aligned with the j_2th character of sequence s_l. Therefore, let α be the position of x containing the i_1th character of sequence s_k and the j_1th character of sequence s_l; and let β be the position of x containing the i_2th character of sequence s_k and the j_2th character of sequence s_l. Since $C_1 \neq C_2$, we know that $\alpha \neq \beta$. If $\alpha > \beta$, then $i_2 < i_1$ and $j_2 < j_1$; if $\alpha < \beta$, then $i_2 > i_1$ and $j_2 > j_1$.

Conversely, assume that either $i_2 < i_1$ and $j_2 < j_1$ or $i_2 > i_1$ and $j_2 > j_1$. In the first case, we can create any alignment of $s_k(1)\cdots s_k(i_2-1)$ with $s_l(1)\cdots s_l(j_2-1)$, concatenated with $s_k(i_2)$ aligned with $s_l(j_2)$, concatenated with any alignment of $s_k(i_2+1)\cdots s_k(i_1-1)$ with $s_l(j_2{+}1)\cdots s_l(j_1{-}1)$, concatenated with $s_k(i_1)$ aligned with $s_l(j_1)$, concatenated with any alignment of $s_k(i_1+1)\cdots s_k(|s_k|)$ with $s_l(j_1+1)\cdots s_l(|s_l|)$. Such an alignment must exist. Similarly, in the second case, we can create any alignment of $s_k(1)\cdots s_k(i_1{-}1)$ with $s_l(1)\cdots s_l(j_1{-}1)$, concatenated with $s_k(i_1)$ aligned with $s_l(j_1)$, concatenated with any alignment of $s_k(i_1+1)\cdots s_k(i_2-1)$ with $s_l(j_1{+}1)\cdots s_l(j_2{-}1)$, concatenated with $s_k(i_2)$ aligned with $s_l(j_2)$, concatenated with any alignment of $s_k(i_2+1)\cdots s_k(|s_k|)$ with $s_l(j_2+1)\cdots s_l(|s_l|)$. Such an alignment also must exist. Hence C_1 and C_2 are compatible. □

Thus, for distinct constraints C_1 and C_2, C_1 is incompatible with C_2 if and only if one of the following happens:

$$i_2 \leq i_1 \text{ and } j_2 \geq j_1,$$

$$i_2 \geq i_1 \text{ and } j_2 \leq j_1.$$

This proposition provides a simple criterion, which can be calculated in constant time for a pair of constraints.

Now let us give the definition of a *pair CCS*.

Definition 14. *A* pair CCS *is a set of constraints, in which every pair of constraints is compatible.*

Knowing exactly when two constraints are compatible, we now can introduce a proposition to determine whether a given set of constraints is a compatible constraint set (CCS) for pairwise sequences, which we will then use within Algorithm 1.

Proposition 2. *Given two sequences s_k, s_l, and a constraint set $C_p = \{\{(k, i_p), (l, j_p)\} \mid 1 \leq p \leq n\}$ on them. The constraints are compatible, if and only if all pairs of the constraints in the set are compatible. That is, the set is a CCS if and only if it is a pair CCS.*

Proof. If the constraints $C_1, C_2, \cdots, C_n$ are compatible, then all pairs of them are compatible as well.

Conversely, assume all pairs of the constraints are compatible for the given constraint set $C_p = \{\{(k, i_p), (l, j_p)\} \mid 1 \leq p \leq n\}$. We will show by induction on m ($m \leq n$) that $C_1, C_2, \cdots, C_n$ must be compatible.

The base case occurs with $m = 2$ when there are only two constraints in the set. Because all pairs of constraints are compatible, then the two constraints in the set must be compatible.

Assume that $C_1, C_2, \cdots, C_m$ are compatible when $2 \leq m < n$. Then following the definition of *compatible constraints* (Definition 13), there must exist an alignment x, satisfying $C_1, C_2, \cdots, C_m$.

Let p be such that i_p is maximal in $i_1, \cdots, i_m$ with $i_p < i_{m+1}$, and let q be such that i_q is minimal in $i_1, \cdots, i_m$ with $i_q > i_{m+1}$, if they exist. We know that C_p and C_q are compatible with C_{m+1}. From Proposition 1, we know that ($i_{m+1} < i_p$ and $j_{m+1} < j_p$) or ($i_{m+1} > i_p$ and $j_{m+1} > j_p$), and we also know that ($i_{m+1} < i_q$ and $j_{m+1} < j_q$) or ($i_{m+1} > i_q$ and $j_{m+1} > j_q$). Such an alignment, satisfying the inequalities concurrently, must exist. So $C_1, C_2, \cdots, C_{m+1}$ are compatible. The case is similar if p does not exist ($m+1$ is the first constraint) and if q does not exist ($m+1$ is the last constraint).

Based on the above, when $m = n$, $C_1, C_2, \cdots, C_n$ are compatible. □

4.2.2 The First Algorithm to Determine a pair CCS

Before constraining an alignment, we must first determine whether the given constraints are compatible. In light of Proposition 2, only a pair CCS can be added to an alignment of two sequences. Here is the first algorithm to decide whether a given constraint set is a pair CCS or not. This is a prerequisite to computing alignments with constraints, and its correctness follows from Proposition 1 and Proposition 2, as it checks whether each pair of constraints is compatible.

Algorithm 1: *Determine pair CCS*

Input: two sequences s_k and s_l;
n constraints ($n \geq 1$): $C_p = \{\{(k, i_p), (l, j_p)\} \mid 1 \leq p \leq n\}$.

Process: Check compatibilities of all pairs of constraints.

```
1   for constraints p from 1 to n − 1
2       for constraints q from p + 1 to n
3           if (i_q − i_p) × (j_q − j_p) > 0 //when (i_q < i_p and j_q < j_p) or (i_q > i_p and j_q > j_p)
4                   // do nothing
5           else Return false;
6   Return true;
```

The complexity of Algorithm 1 is $O(n^2)$, which is high. As the number of constraints increases, the algorithm becomes slower. So we need a more efficient algorithm to determine if constraints are compatible.

4.2.3 Sort Constraints on Multiple Sequences

Since the constraints are given by users according to their needs, they may not be in order, from left to right according to the alignment. Before constraining an alignment, we can sort the constraints. Algorithm 2 will address this work. It sorts the constraints based on their positions in one sequence. This algorithm can be used for either two sequences or multiple sequences.

We assume that all the constraints are distinct. When sorting constraints involving each sequence s_q, they are sorted in increasing order of the position of s_q, and if there are two constraints involving the same position of s_q, we rank constraints lower depending on the other sequence of the constraint. For example, if we have constraints $C_1 = \{(k,i),(l,j)\}$, $C_2 = \{(k,i),(q,p)\}$, then we rank C_1 first if $l < q$, and C_2 first otherwise. If $l = q$, then we use j and p to order.

Then we have our Algorithm 2: *Sort* as follows.

Algorithm 2: *Sort*

Input	m sequences $s_1, s_2, \cdots, s_m$;	
	n constraints($n \geq 1$): $C_p = \{\{(k_p, i_p),(l_p, j_p)\} \mid 1 \leq p \leq n, 1 \leq k_p, l_p \leq m\}$.	
Process	Sort constraints based on position numbers.	
1	$r_{1\cdots m} = 0$;	//initialize m counters
2	for p from 1 to n	//for all constraints involving C_p
5	$r_{k_p}++$;	
5	$r_{l_p}++$;	
6	$Sorted_{k_p}[r_{k_p}] = C_p$;	
6	$Sorted_{l_p}[r_{l_p}] = C_p$;	
7	for q from 1 to m	//for all sequences
8	$Sorted_q = QuickSort(Sorted_q)$;	//call a standard quicksort algorithm
Output	return $Sorted_{1\cdots m}$;	

Quicksort is a well-known sorting algorithm developed by Hoare [12] that, on average, makes $O(n \log n)$ comparisons to sort n items. In Algorithm 2, the time taken is $O(n + k_1 \log k_1 + \cdots + k_m \log k_m)$, where n is the number of constraints, m is the number of sequences, and k_i is the number of constraints involving sequence s_i ($1 \leq i \leq m$). This will give $O(n \log n)$ complexity for constraints either in two sequences or m sequences, and it will be useful for the rest of this thesis.

4.2.4 Determine a pair CCS Using a Graph Algorithm

Here we present another algorithm to do the same task which is the one we will generalize to multiple sequences. Following the definition of constraints (Definition 10), we can use a pair to denote a constraint C between sequences s_k and s_l over m sequences $s_1, \cdots, s_m$: $C = \{(k,i),(l,j)\}$, where $l \neq k, 1 \leq k, l \leq m, 1 \leq i \leq |s_k|, i \leq j \leq |s_l|$. In this section, we only discuss constraints on two sequences s_k and s_l, and we have n constraints $C_1 = \{(k,i_1),(l,j_1)\}$, $C_2 = \{(k,i_2),(l,j_2)\}, \cdots, C_n = \{(k,i_n),(l,j_n)\}$.

Before introducing the algorithm, it is necessary to review some definitions from graph theory from [1] and [19].

Definition 15. *A* directed graph $G = (V, E)$ *is a finite set of vertices* V*, and a finite set of edges with* $E \subseteq V \times V = \{(v_1, v_2) \mid v_1, v_2 \in V, v_1$ *is a source and* v_2 *is a target of the edge* $(v_1, v_2)\}$.

Definition 16. *Let* $G = (V, E)$ *be a graph. A* path *in* G *is a sequence of vertices* $v_1, v_2, \cdots, v_n$*, such that* (v_1, v_2)*,* (v_2, v_3)*,*$\cdots$*,* (v_{n-1}, v_n) *are edges. This path is from vertex* v_1 *to vertex* v_n*, and passes through vertices* $v_2, v_3, \cdots, v_{n-1}$*, and ends at vertex* v_n*. The* length *of the path is the number of edges on the path, which is* $n - 1$*. As a special case, a single vertex* v *by itself denotes a path of length zero from* v *to* v.

Definition 17. *Any path of length at least one that begins and ends in the same vertex is a* cycle.

Definition 18. *A* directed acyclic graph*, or* DAG *for short, is a directed graph with no cycles.*

Our algorithm uses a graph algorithm to determine if a set of constraints on two sequences is compatible. It contains two steps: first, to convert the given constraints to a graph; second, to find if there is any cycle in the graph.

Step 1: Convert Constraints to a Graph

From the constraints, we are trying to discuss the compatibility in the form of a graph problem. First, we discuss how we can convert our constraints to a directed graph.

In the graph, we convert each constraint into one vertex, and then n constraints can be converted to n vertices, and we will denote each vertex by the constraint label. For example, a constraint $C = \{(k, i), (l, j)\}$ can be converted to a vertex with label C.

We connect the vertices with edges following certain rules. As vertices have an order decided by the position number in each sequence (the position of the constraint in sequence k and l), we first sort them with each sequence in ascending order using Algorithm 2. Then we connect the neighbouring vertices on the same sequence with one directed edge from the one with a smaller position number to the one with a larger position number. If there are two consecutive constraints involving the same position of the same sequence, then we add edges in both directions introducing a cycle. So if the number of constraints is n, then the number of vertices would be n, and the number of edges would be at most $4(n - 1)$, as we can have a different sorted order for both sequences. Here is an example to demonstrate the conversion.

Example 5. *Given two sequences* s_k *and* s_l*, and four constraints* $C_1 = \{(k, 5), (l, 3)\}$*,* $C_2 = \{(k, 20), (l, 10)\}$*,* $C_3 = \{(k, 50), (l, 66)\}$ *and* $C_4 = \{(k, 73), (l, 90)\}$*, the constraints can be visualized in Figure 4.4.*

After converting the constraints to a graph, we obtain the graph in Figure 4.5.

The following algorithm converts constraints $C_1, C_2, \cdots, C_n$ on sequences s_k and s_l to a graph using adjacency lists as the data structure [8]. The algorithm calls Algorithm 2 using QuickSort [12] to sort the constraints in ascending order according to sequence number.

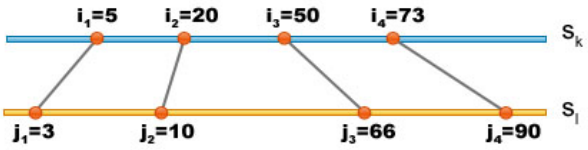


Figure 4.4: We show the visualization of four constraints.

Figure 4.5: Pictured above is the graph obtained from converting the constraints in Figure 4.4.

Algorithm 3: $Convert_{pairwise}$

Input	two sequences s_k and s_l;	
	n constraints ($n \geq 1$): $C_p = \{\{(k, i_p), (l, j_p)\} \mid 1 \leq p \leq n)\}$;	
Process	Sort constraints on each of the two sequences, then convert them to a directed graph.	
1	for p from 1 to n	//the first *for loop*: add all constraints
2	$v_p = CreateVertex(C[p])$;	
3	$Sorted_{k,l} = Sort(s_{k,l}, C_{1\cdots n})$;	
4	for i from 1 to $\lvert Sorted_k \rvert$	//the second *for loop*:
		//add all constraints involving sequence s_k
5	$CreateEdge(Sorted_k[i], Sorted_k[i+1])$;	
6	if $Sorted_k[i]$ and $Sorted_k[i+1]$ both involve the same position of s_k	
7	$CreateEdge(Sorted_k[i+1], Sorted_k[i])$;	
8	for i from 1 to $\lvert Sorted_l \rvert$	//the third *for loop*:
		//add all constraints involving sequence s_l
9	if edge does not exist	
10	$CreateEdge(Sorted_l[i], Sorted_l[i+1])$;	
11	if $Sorted_l[i]$ and $Sorted_l[i+1]$ both involve the same position of s_l, and edge does not exist	
12	$CreateEdge(Sorted_l[i+1], Sorted_l[i])$;	
Output	return $G = (V, E)$;	

The complexity to convert sorted constraints to a graph is $O(n)$, and combined with the sorting this takes $O(n \log n)$ time.

Step 2: Find a Cycle in the Graph

The reason we are trying to determine whether there exists a cycle in the graphs produced is because we found that there is a relationship between the compatibility of constraints and the existence of cycles in a graph, as given in Theorem 1.

Theorem 1. *Given two sequences s_k and s_l, and n constraints ($n \geq 1$): $C_p = \{\{(k, i_p), (l, j_p)\} \mid 1 \leq p \leq n\}$, and a graph $G = (V, E)$ created from the constraints via Algorithm 3. Then $C_1, C_2, \cdots, C_n$ are compatible constraints if and only if G has no cycles.*

Proof. Assume that $C_1, C_2, \cdots, C_n$ $(n \geq 1)$ are compatible constraints, and they are sorted by Algorithm 2. Then there exists an alignment x satisfying $C_1, C_2, \cdots, C_n$, which means that the i_pth character of sequence s_k is aligned with the j_pth character of sequence s_l, for every $1 \leq p \leq n$. Moreover, for both s_k and s_l, these positions must be disjoint for an alignment to exist. Thus neither *if statement* on line 6 nor line 11 are true. Therefore, let α_p be the position of x containing the i_pth character of sequence s_k and the j_pth character of sequence s_l, and let v_p be the vertex corresponding to this constraint. If we sort $\alpha_1, \alpha_2, \cdots, \alpha_n$, then the indices will be identical to the edges added in Algorithm 3 in the second *for loop*, and also in the third *for loop*. Moreover, there is not any repeated vertex, since the second coordinate of the pair with k as the first parameter strictly increases. Hence, there are no cycles.

Conversely, assume that the graph G is a DAG, where G was created from $C_1, C_2, \cdots, C_n$ $(n \geq 1)$, following Algorithm 3. Then neither *if statement* in line 6 nor line 11 can be true, since otherwise a cycle would be immediately introduced. Thus the constraints strictly increase according to each sequence. Then in Algorithm 3, the second and third for loops must sort in the same order as otherwise there are two vertices C_i, C_j, such that C_i appears before C_j in the first for loop, and C_j appears before C_i in the second for loop, which contradicts the fact that there are no cycles. Then if $r_1, r_2, \cdots, r_n$ is the sorted sequence positions in s_k and $t_1, t_2, \cdots, t_n$ is the sorted sequence positions in s_l, then we can create any alignment of $s_k(1) \cdots s_k(r_1 - 1)$ with $s_l(1) \cdots s_l(t_1 - 1)$, concatenated with $s_k(r_1)$ aligned with $s_l(t_1)$, concatenated with any alignment of $s_k(r_1 + 1) \cdots s_k(r_2 - 1)$ with $s_l(t_1 + 1) \cdots s_l(t_2 - 1)$, concatenated with $s_k(r_2)$ aligned with $s_l(t_2)$, $\cdots$, concatenated with any alignment of $s_k(r_{p-1}+1) \cdots s_k(r_p-1)$ with $s_l(t_{p-1}+1) \cdots s_l(t_p-1)$, concatenated with $s_k(r_p)$ aligned with $s_l(t_p)$, $\cdots$, concatenated with $s_k(r_n)$ aligned with $s_l(t_n)$, concatenated with any alignment of $s_k(r_n + 1) \cdots s_k(|s_k|)$ with $s_l(t_n + 1) \cdots s_l(|s_l|)$. Such an alignment must exist, so $C_1, C_2, \cdots, C_n$ are compatible. □

To detect if a graph has cycles, or if a graph is a DAG, we will use a depth-first search, $DFS(G, v)$ [8]. The strategy followed by DFS is, as its name implies, to search "deeper" in the graph whenever possible. In DFS, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to

explore edges leaving the vertex from which v was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered. The complexity of a depth-first search is $O(|V| + |E|)$, and for our algorithm, this gives $O(n)$ when we use the adjacency list as the data structure for graphs [1] in the current situation of pairwise alignment.

4.2.5 Complexity

For two sequences, the complexity of sorting constraints is $O(n \log n)$, the complexity of converting sorted constraints to a graph is $O(n)$, and the complexity to detect cycles in the graph is $O(n)$, so the entire complexity for detecting the compatibility of a given set of constraints is $O(n \log n)$. Moreover, this graph theoretic approach will be the most useful with multiple sequences in the next section.

4.3 Compatible Constraint Set on Multiple Sequence Alignment

In Section 4.2, we discussed compatible constraints on two sequences, and have defined *compatible constraints* and a *compatible constraint set (CCS)* in Definition 13 and a *pair CCS* in Definition 14. Then we gave Proposition 1 and Proposition 2 to provide a simple criteria to determine if a set of constraints is compatible or not. We also provided algorithms to determine if a constraint set is a compatible constraint set on two sequences. In this section, we will talk about the compatibility of constraints on multiple sequences.

4.3.1 Analysis of Compatible Constraints on Multiple Sequences

The characterization of Proposition 1 and Proposition 2 for two sequences does not hold any more for multiple sequences. Take the constraints in Figure 4.6 as an example. Although all pairs of constraints are compatible, the constraint set is incompatible. However, one needs to analyze all constraints simultaneously in order to establish this fact. Thus for multiple sequences, we need another characterization and algorithm to address this problem.

4.3.2 Determine a CCS on Multiple Sequences Using a Graph Algorithm

Here, we will generalize Algorithm 3 in Section 4.2.4 to multiple sequences, in order to determine if a set of constraints on multiple sequences is compatible. It contains three steps: we will first

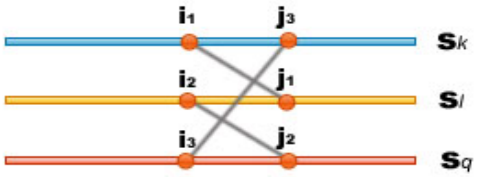


Figure 4.6: We provide an example of incompatible constraints on multiple sequences.

sort the constraints involving each sequence by location; then we will convert the constraints to a directed graph; third, we will check if there exists a cycle in the directed graph.

Step 1: Convert Constraints to a graph

In the graph, we convert each constraint into one vertex, then n constraints can be converted to n vertices. Then we sort them by each sequence in an ascending order using Algorithm 2, and connect the neighbouring vertices on the same sequence with one directed edge from the vertex with a smaller position number to the one with a larger position number, and also adding in a second reverse edge if we have two constraints meet at the same position of one sequence both going to a same sequence. Thus if the number of constraints is n, then we have n vertices, and the number of edges would be at most $4n$.

Algorithm 4: *Convert_multiple*

Input m sequences $s_1, s_2, \cdots s_m$;

n constraints ($n \geq 1$): $C_p = \{\{(k_p, i_p), (l_p, j_p)\} \mid 1 \leq p \leq n, 1 \leq k_p, l_p \leq m\}$;

Process: Sort constraints on multiple sequences, then convert them to a directed graph.

1 for p from 1 to n //the first *for loop*: add all constraints

2 $v_p = CreateVertex(C[p])$;

3 $Sorted_{1\cdots m} = Sort(s_{1\cdots m}, C_{1\cdots n})$;

4 for k from 1 to m //the second *for loop*: add all sequences

5 for i from 1 to $|Sorted_k| - 1$ //the third *for loop*:
//add all constraints involving each sequence

6 if edge does not exist

7 $CreateEdge(Sorted_k[i], Sorted_k[i+1])$;

8 if $Sorted_k[i]$ and $Sorted_k[i+1]$ both involve the same position of s_k,
and both get mapped to the same sequence,
and edge does not exist

9 $CreateEdge(Sorted_k[i+1], Sorted_k[i])$;

Output return $G = (V, E)$;

The complexity of the sorting algorithm, Algorithm 2, is $O(n \log n)$. So for Algorithm 4: *Convert_multiple*, using k_i to denote the number of constraints involving sequence s_i $(1 \leq i \leq m)$, it takes $O(n \log n + n + (k_1 + k_2 + \cdots + k_m)) \leq O(n \log n + n + 2n) = O(n \log n + 3n)$ time, which will give a complexity of $O(n \log n)$.

Step 2: Find a Cycle in the Graph

First, let us look at some examples.

Example 6. *Following Algorithm 4, the result of sorting constraints in Figure 4.6 could be represented as*

$$s_k : C_1 < C_3,$$
$$s_l : C_2 < C_1,$$
$$s_q : C_3 < C_2.$$

And Figure 4.7 shows the graph produced from the constraints, which are incompatible.

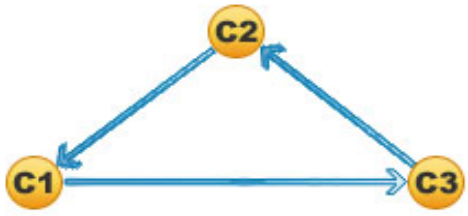


Figure 4.7: The graph obtained from the incompatible constraints in Figure 4.6.

It is obvious to see that there are cycles in the graph.

Example 7. *We provide another example of compatible constraints in Figure 4.8 which can be converted into Figure 4.9, with the result of sorting.*

$$s_k : C_1 < C_3,$$
$$s_l : C_2 < C_1,$$
$$s_q : C_2 < C_3.$$

As shown in Figure 4.9, the graph converted from the compatible constraints does not have any cycle.

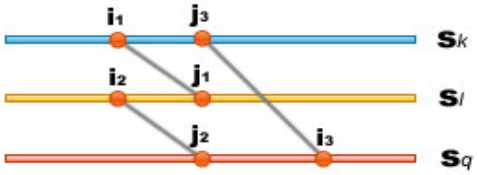


Figure 4.8: We provide an example of compatible constraints on multiple sequences.

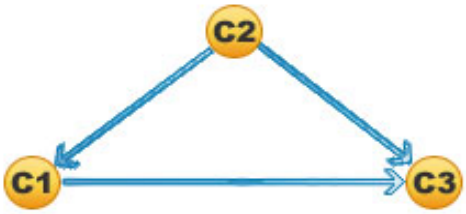


Figure 4.9: The graph converted from the compatible constraints in Figure 4.8.

From the two examples above, we can find that there is a relationship between the compatibility of constraints and the existence of cycles in a directed graph. That is, if there exists a cycle in the directed graph, the constraints are incompatible, otherwise, the constraints are compatible. Then we have a theorem as follows:

Theorem 2. *Given m sequences $s_1, s_2 \cdots s_m$, n constraints ($n \geq 1$): $C_p = \{\{(k_p, i_p), (l_p, j_p)\} \mid 1 \leq p \leq n, 1 \leq k, l \leq m\}$, and a graph $G = (V, E)$ created from the constraints via Algorithm 4. Then $C_1, C_2, \cdots, C_n$ are compatible constraints, if and only if G has no cycles.*

Before proving Theorem 2, we need to introduce a definition of *topological sort* of a DAG.

Definition 19. A topological sort *is a process of assigning a linear ordering to the vertices of a DAG, so that if there is an edge from vertex i to vertex j, then i appears before j in the linear ordering.*

From [16], we know that one important property of a DAG is the *topological sorting property*. It is always possible to write the vertices of a DAG in a list, $v_1, v_2, \cdots, v_n$, in such a way that if there is a path from v_i to v_j in the DAG, then in the list v_i precedes v_j (that is $i < j$).

For example, a, b, d, c and a, d, b, c are both topological sorts of the DAG given in Figure 4.10.

If we have an alignment and constraints as in Figure 4.11, then this corresponds to the graph in Figure 4.10.

Then, let us prove Theorem 2.

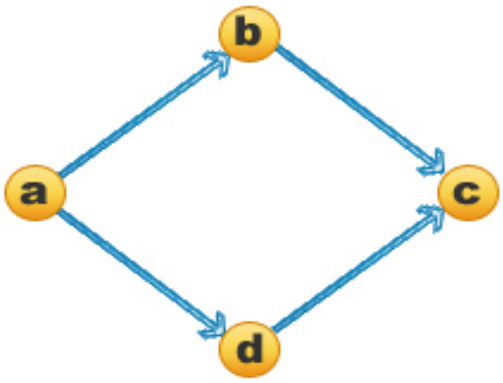


Figure 4.10: Pictured above is an example of a DAG.

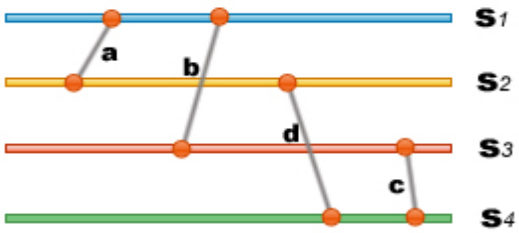


Figure 4.11: Pictured above is a sequence alignment restricted by four constraints a, b, c and d, which corresponds to the graph in Figure 4.10.

Proof. Assume that $C_1, C_2, \cdots, C_n (n \geq 1)$ are compatible constraints, and they are sorted by Algorithm 2 which sorts the constraints by each sequence based on their positions in that sequence. If there is more than one constraint with the same sequence number, and because the constraints are compatible, there cannot be two constraints from the same position of one sequence both mapped to the same sequence. Thus, no edges are introduced in line 9. Then there exists an alignment x satisfying $C_1, C_2, \cdots, C_n$, which means that the i_pth character of sequence s_{k_p} is aligned with the j_pth character of sequence s_{l_p}, for every $1 \leq p \leq n$. Therefore, let α_p be the position of x containing the i_pth character of s_{k_p} and the j_pth character of s_{l_p}, and let v_p be the vertex corresponding to this constraint. In Algorithm 4, when we create the edges, the third *for loop* adds edges from v_r to v_t, only if $\alpha_r \leq \alpha_t$, and if $\alpha_r = \alpha_t$, then only one edge is added from sequence with smaller sequence number to that with larger sequence number (going down the alignment). Thus, there are no repeat vertices on any path. Hence, there are no cycles.

Conversely, assume that G is a DAG, where G is created from $C_1, C_2, \cdots, C_n (n \geq 1)$ following Algorithm 4. Since there is always a topological sort of any DAG, then we can topologically sort G, and find an order of the vertices in the DAG, which does not contain a cycle (topological sort of any DAG can be easily accomplished using a depth-first search as in [1]). Thus an alignment

satisfying all $C_1, C_2, \cdots, C_n (n \geq 1)$ must exist, where the constraints appear in the same order as the topological sort. Therefore, the constraints are compatible. □

4.3.3 Complexity

In a similar fashion to the last section, we can also use a depth first search [8] to detect the existence of cycles in multiple sequences, which again gives a $O(|V| + |E|) = O(n)$ time complexity. Thus, it is not dependent on the number of sequences.

For m sequences and n constraints, the sorting of constraints, then converting the constraints into a graph, and then detecting cycles can also be accomplished in $O(n \log n)$ time, and is therefore independent of the number of sequences.

Chapter 5

Pairwise Sequence Alignment Satisfying A Compatible Constraint Set

In Chapter 4, we analyzed compatible constraint sets and provided algorithms to determine if a set of constraints is a CCS for both two sequences and multiple sequences. Given a CCS and an alignment, we would like to focus on establishing the correspondence between the alignment and the CCS. In this chapter, we will address this work for the case of using two sequences. Most of the examples are given using DNA sequences, but it also works with protein sequences if we use a different scoring matrix. The work in this chapter applies to both global and local alignment (except the speed-ups of Section 5.3 only apply to global alignment) with linear gap penalties.

5.1 The Representation of an Alignment and a CCS on a Dynamic Programming Matrix

As introduced in Section 3.2, the Needleman-Wunsch and Smith-Waterman algorithms can align two sequences based on dynamic programming, which calculates two matrices, called dynamic programming matrices. One matrix stores the highest scores of all sub-alignments, and the other records the arrows or paths, which are calculated from the scores. One can easily find an alignment by a path on a dynamic programming matrix.

We also examined CCS in Chapter 4, and determined that a CCS can be visualized as lines between sequences as shown in Figure 4.2. In this chapter, we will represent constraints differently. A constraint will instead be visualized as a point on the dynamic programming matrix, and a CCS as a set of points (or positions) on the dynamic programming matrix. This abstraction will be used for algorithmic design.

Let us look at an example. In Figure 5.1(a), we use points to represent constraints. The optimal alignment of the two sequences is represented with a path.

One can see that in the example of Figure 5.1(a), the path does not go through any point. In other words, the optimal alignment does not satisfy the given CCS. Naturally, we would like to find another path, which does traverse the CCS points and at the same time yields a highest score

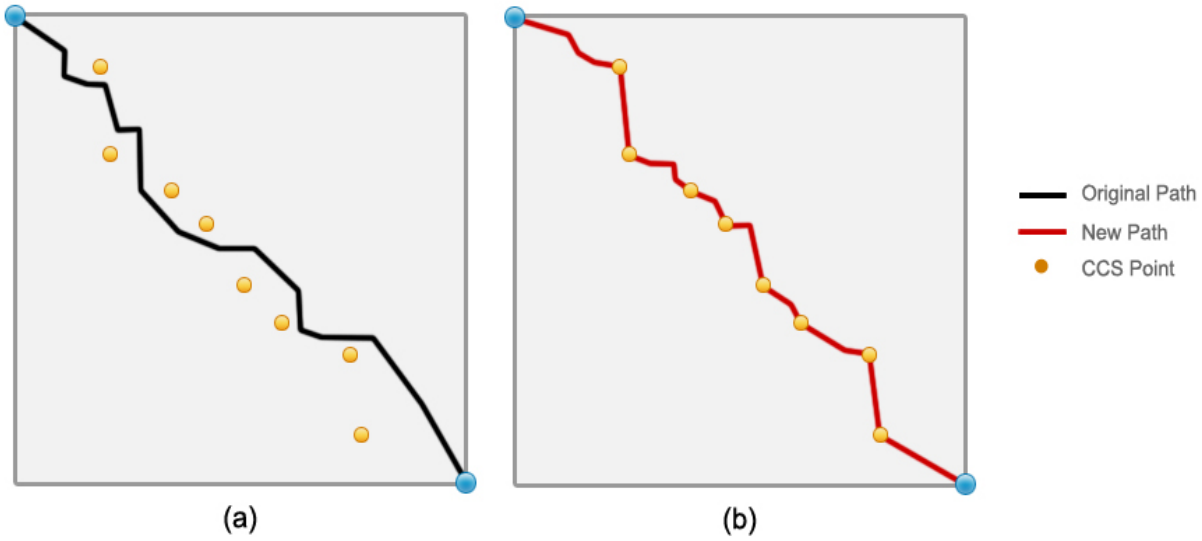


Figure 5.1: (a) We show the original optimal alignment path together with the CCS points; (b) The new optimal alignment satisfying all constraints goes through all the CCS points.

compared to other paths through all such points. A new path in Figure 5.1(b) gives the optimal alignment that satisfies the CCS, which we wish to calculate in this chapter.

5.2 A Basic Method of Calculation

The straightforward dynamic programming algorithm can find the mathematically optimal path, which achieves the highest score. However, this path cannot guarantee, and will likely not, traverse all the points, or guarantee that the alignment satisfies the CCS. This occurs because it does not have any criteria to restrict the path except using the highest score.

In this section, we will introduce a basic method to find a path which traverses all CCS points, and at the same time, is an optimal path which does go through the points. It works for both global and local alignments.

We mentioned that the constraints can be represented as points on a dynamic programming matrix, and therefore given a constraint, it is easy to convert it into a point. Let $s_k = x_1 \cdots x_m$, $s_l = y_1 \cdots y_n$ be two sequences, where x_i, y_j $(1 \leq i \leq m, 1 \leq j \leq n)$ are individual characters. The dynamic programming matrix is indexed by i and j respectively for each sequence. Then a constraint $C = \{(k, i), (l, j)\}$ is represented with the point (i, j) on the matrix.

One process of getting such an alignment includes the following four steps (Figure 5.2):

1. Convert constraints to points and fix them onto the dynamic programming matrix;

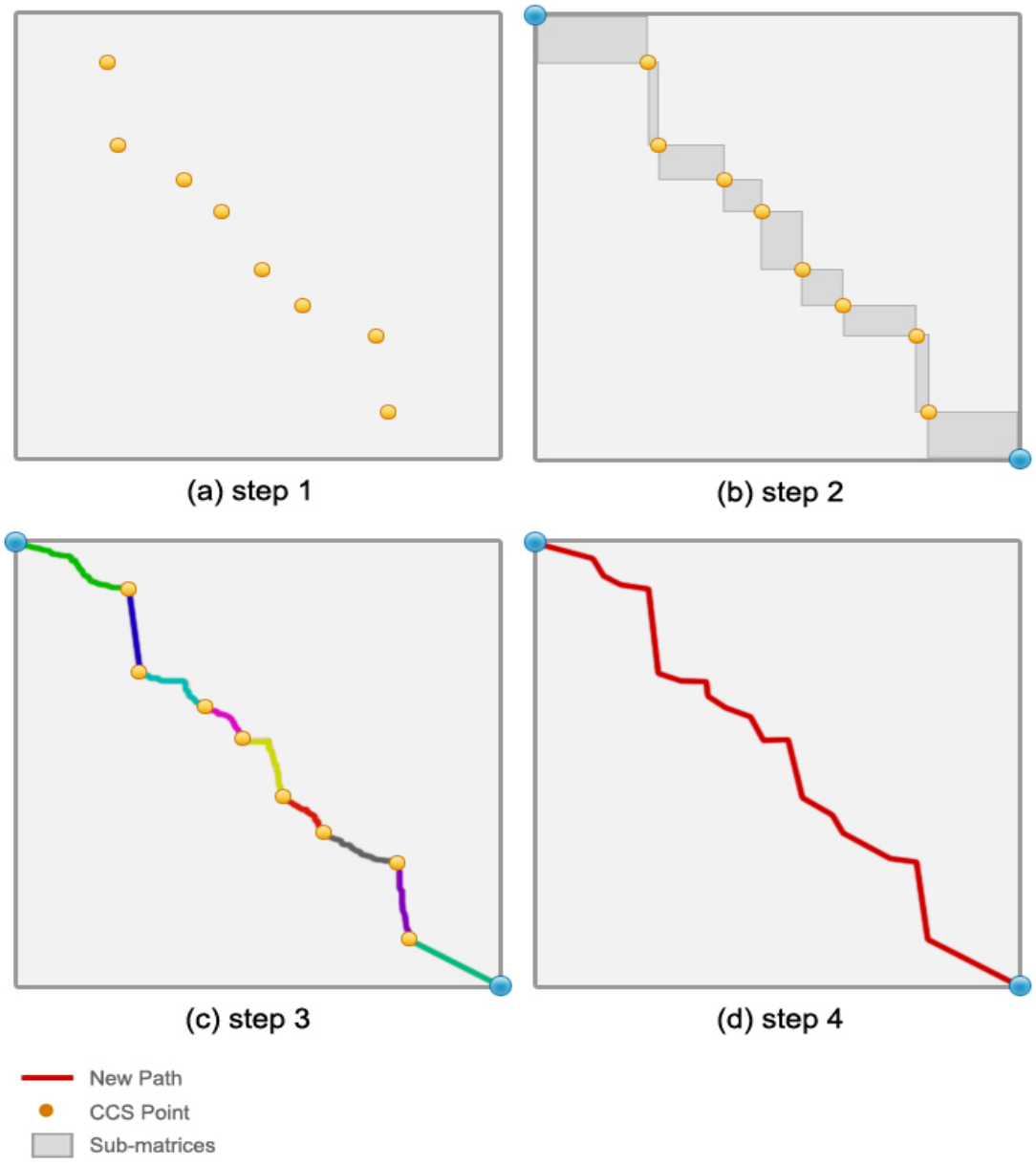


Figure 5.2: Four steps of getting a new path going through all the CCS points. (a) Convert constraints to points and fix them onto the dynamic programming matrix; (b) Divide the matrix into several sub-matrices regularized by the points; (c) For each sub-matrix, perform straightforward dynamic programming starting at the high score from the previous matrix to find the optimal sub-path in it; (d) Concatenate the sub-paths from (c), which goes through all the points, and get the new alignment from the new path.

2. Divide the matrix into several sub-matrices regularized by the points;

3. For each sub-matrix, starting from the high score of the previous sub-matrix as the initial value, perform straightforward dynamic programming to find the optimal sub-path in it. For global sequence alignment, initialize the first row and column to multiples of the gap penalty. For local sequence alignment, start at the high score from the previous sub-matrix, and in the first row and column, add the gap penalty until hitting 0, then do not go below;

4. Concatenate the sub-paths obtained in step 3, which goes through all the points, and get a new alignment from the new path.

Then, if an optimal alignment is first calculated, after which constraints are entered, we can use the same original dynamic programming matrices to calculate all of the sub-alignments.

The flow chart in Figure 5.3 summarizes the entire procedure.

5.3 Speed-ups Using the Original Alignment

The information from the original alignment can be used in calculating the new alignment, so that we do not have to recompute the entirety of all sub-matrices. This section works for global sequence alignment, but we have not extended it to local sequence alignment. First of all, let us introduce some definitions which are useful for the rest of this section.

Definition 20. *If we have a path α on the matrix, and two points a and b on α, then the path of α between a and b is called α* restricted to *a, b which is denoted by $\alpha|_{a,b}$.*

Then let us define the *offset* between a constraint point and the original path α.

Definition 21. *If there is a constraint point on a dynamic programming matrix, then the point can divide the entire matrix into four sub-matrices, which we denote as I, II, III, IV as shown in Figure 5.4. If the original path α passes into a sub-matrix, then the* offset *between α and the constraint point of sub-matrix I is denoted by d_I, and the* offset *between α and the constraint point of sub-matrix III is denoted by d_{III}. If the constraint point is beneath α, the offset is negative as shown in Figure 5.4 (a), otherwise, if the point is above α, then the offset is positive as in Figure 5.4 (b).*

In this section, if C is a constraint, then we will only recalculate sub-matrices I and III, as the new optimal alignment must go through I and III, but not II nor IV (the new optimal path must go through the constraint). Thus, if we are calculating I, we are only concerned with d_I, and if we are calculating III, then we are only concerned with d_{III}. Thus, if the direction I or III is clear, we will just speak of the offset. Hence, in the figures below, we denote the offset between the original path and point C_1 by d_1, the offset between the original path and C_2 by d_2, as the sub-matrix we are calculating is clear from the contexts.

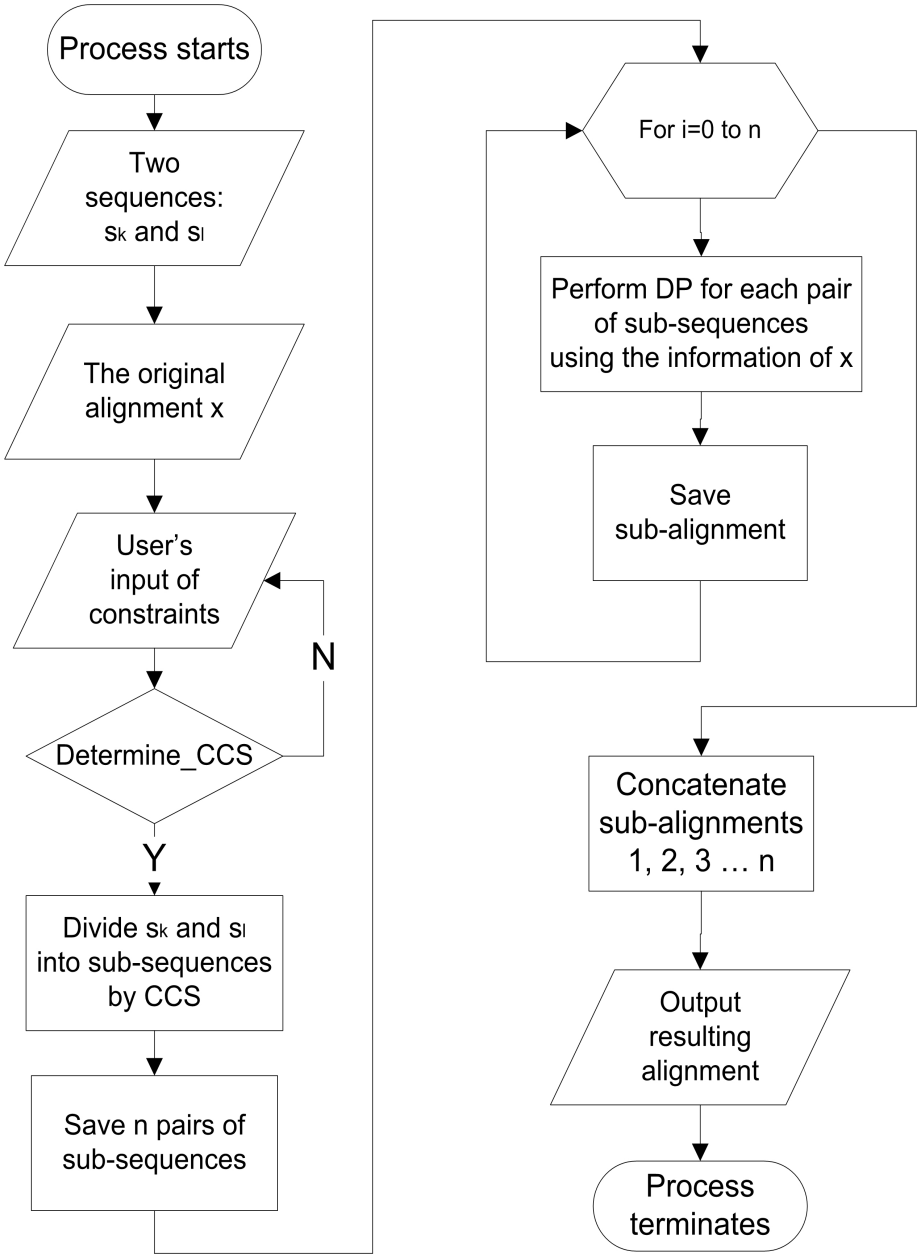


Figure 5.3: The flow chart of calculating the optimal pairwise alignment satisfying a CCS.

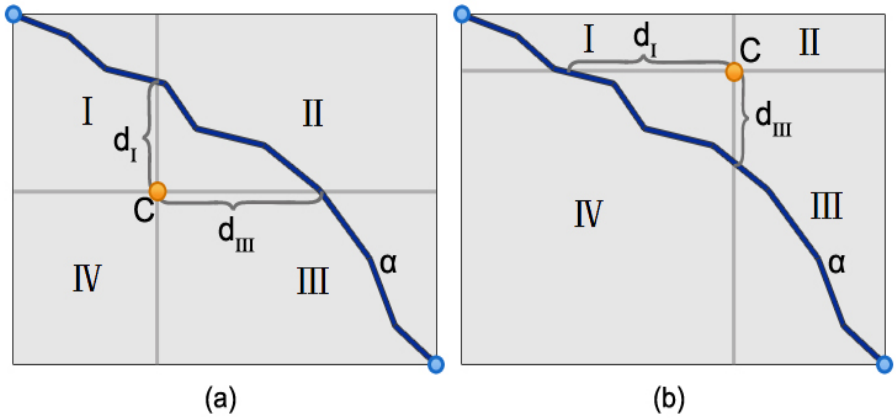


Figure 5.4: The figures above demonstrate the offset between a constraint point and the original path.

5.3.1 Some Parts can be Ignored

Let us examine the relationships between the original path α and the sub-matrices. All the possible cases are listed below, in which two consecutive constraints C_1 and C_2 form the two corners of the sub-matrix.

1. The original path α goes through the top-right part of the sub-matrix between constraints C_1 and C_2, as shown in Figure 5.5. In this case, we have

$$d_1 \leq 0,\ d_2 \leq 0.$$

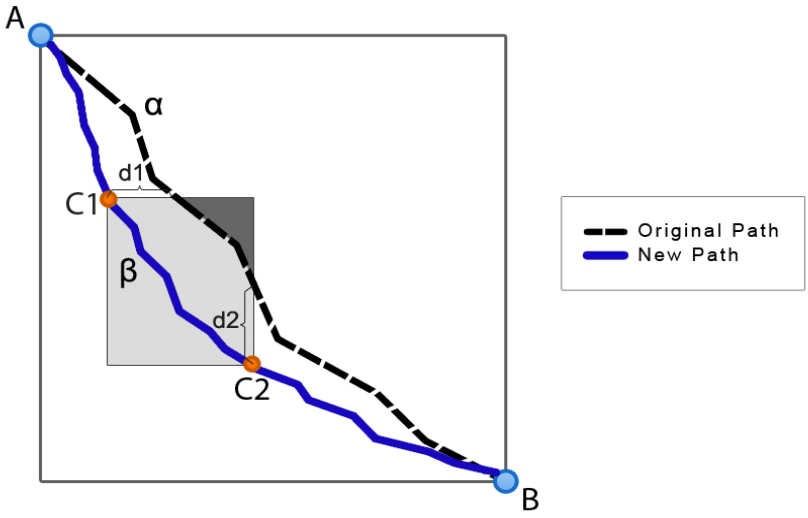


Figure 5.5: The original path (dotted line) goes through the top-right part of the sub-matrix between constraints C_1 and C_2.

2. The original path α goes through the bottom-left part of the sub-matrix between constraints C_1 and C_2, as shown in Figure 5.6. In this case, we have

$$d_1 \geq 0,\ d_2 \geq 0.$$

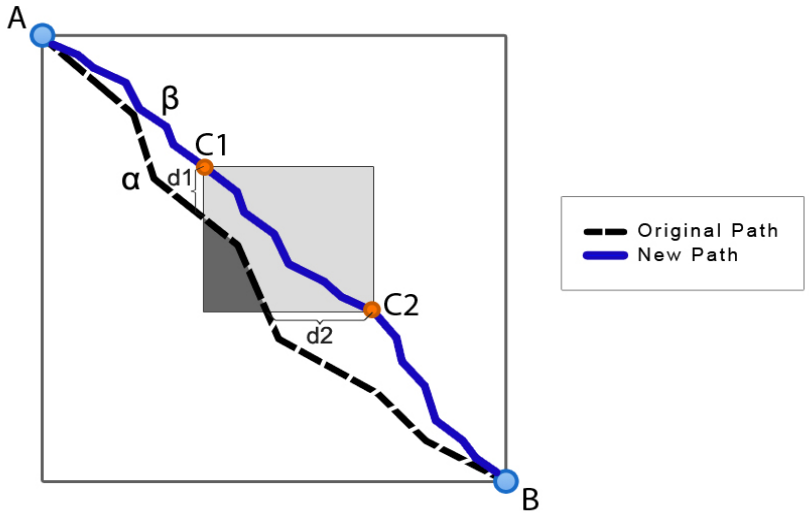


Figure 5.6: The original path (dotted line) goes through the bottom-left part of the sub-matrix between constraints C_1 and C_2.

3. The original path α crosses the new optimal path β between constraints C_1 and C_2, as shown in Figure 5.7.

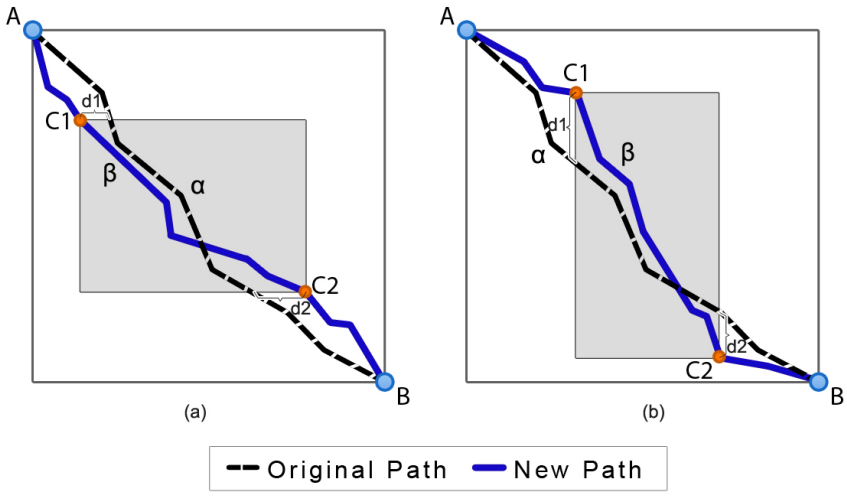


Figure 5.7: The original path (dotted line) crosses the new optimal path.

In (a), we have

$$d_1 < 0,\ d_2 > 0,$$

and in (b), we have

$$d_1 > 0,\ d_2 < 0.$$

Because the new optimal path crosses the original path, it is not obvious how we can use the information from the original path. In this case, we recalculate the whole sub-matrix to find the new optimal alignment.

4. The original path α does not go through the sub-matrix between C_1 and C_2 as shown in Figure 5.8. In this case, the offset between the constraint point and the original path is not defined. It is not obvious how we can use the information from the original path, so we need to recalculate the whole sub-matrix to find the new optimal alignment.

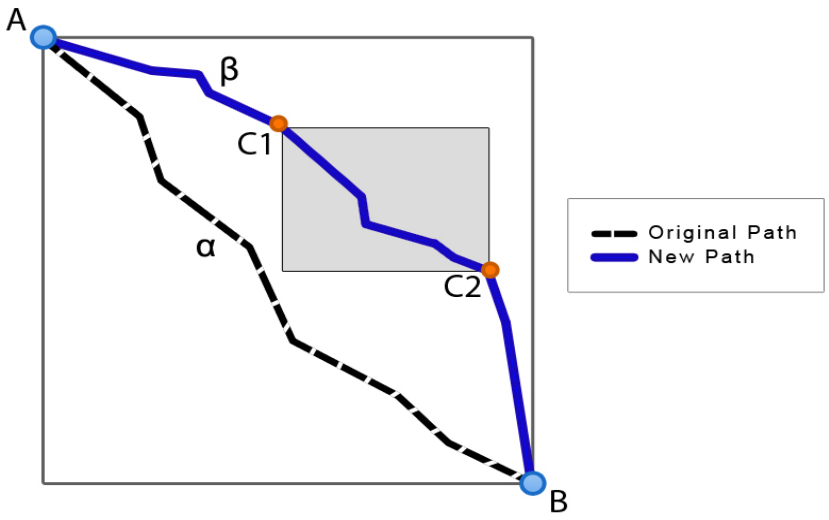


Figure 5.8: The original path (dotted line) does not go through the sub-matrix between constraints C_1 and C_2.

In all cases above, C_1 and C_2 can either be constraints or corners of the entire matrix.

From these cases, we found that if $d_1 \times d_2 \geq 0$, some parts of the score calculation and alignment construction in the sub-matrix can be ignored, such as in case 1 and 2. But if $d_1 \times d_2 < 0$, we have to recalculate all sub-matrices, as is required in case 3. For example, to find the new optimal alignment, we can ignore the part of the matrix shaded in dark grey in case 1 and case 2, each of which we call an *irrelevant part*, and recalculate the part shaded in light grey, because the new optimal path will not pass into the irrelevant parts. There is a special case when $d_1 = 0$ and $d_2 = 0$ which is covered by both class 1 and class 2. Here, the new optimal path overlaps the original path, so we do not need to recalculate it. This is because we use the same algorithm both in calculating

the original path and the new path, and if we recalculate the sub-matrix for the new path, though we use different initial values from the original values, the path would be the same.

Now let us prove that an irrelevant part can be safely ignored while we still find the optimal path. This speed gain might end up being significant as constraints might often be found in proximity to the optimal alignment. We only need to prove that if the new optimal path hits the original path, the optimal segment between the two points of intersection is on the original path.

Proposition 3. *Given two sequences s_k and s_l, and two points (i_1, j_1) and (i_2, j_2), where $C_1 = \{(k, i_1), (l, j_1)\}$ is a constraint or (i_1, j_1) is the upper left corner of the matrix, and $C_2 = \{(k, i_2), (l, j_2)\}$ is a constraint or (i_2, j_2) is the lower right corner of the matrix. If an original optimal path α crosses with the new path β from C_1 to C_2 at two points a at (i_a, j_a) and b at (i_b, j_b), then $\beta|_{C_1,a}$, followed by $\alpha|_{a,b}$, followed by $\beta|_{b,C_2}$ is optimal.*

Proof. Figure 5.9 shows a sub-matrix with two consecutive constraint points C_1 and C_2 as the corners. We use contradiction to prove that $\beta|_{C_1,a}$, followed by $\alpha|_{a,b}$, followed by $\beta|_{b,C_2}$ is optimal.

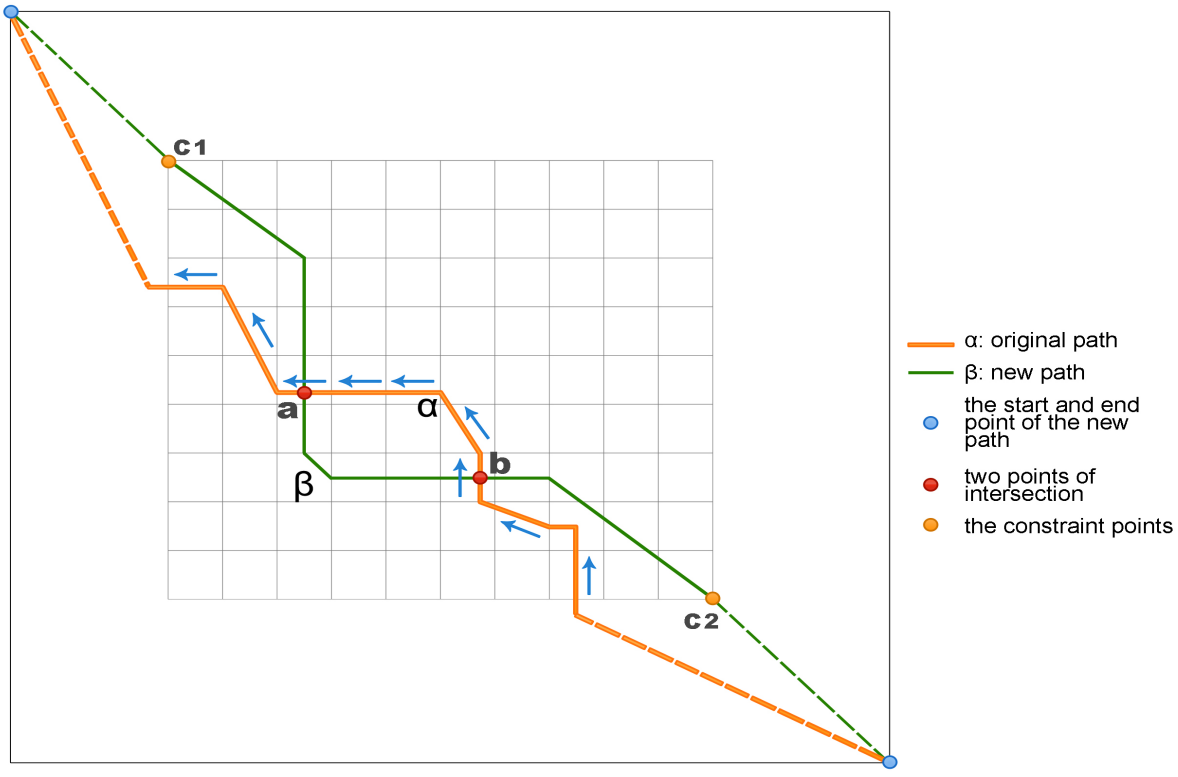


Figure 5.9: The picture above shows the original and the new path in a sub-matrix with two consecutive constraint points C_1 and C_2 as the corners. If they cross each other, then $\beta|_{C_1,a}$, followed by $\alpha|_{a,b}$, followed by $\beta|_{b,C_2}$ is optimal.

Assume that it is not optimal. Then the alignment with the path $\beta|_{C_1,C_2}$ has a higher score

than the alignment corresponding to the path $\beta|_{C_1,a}$, followed by $\alpha|_{a,b}$, followed by $\beta|_{b,C_2}$.

Let y be the alignment corresponding to the first path with $\beta|_{a,b}$, and let x be the alignment corresponding to the second path with $\alpha|_{a,b}$. Then the combined score of y is higher than the score of x. Thus we can substitute y for x between the position a at (i_a, j_a) and b at (i_b, j_b) in the alignment from C_1 to C_2. Therefore, this new alignment has a higher score than α, contradicting the original optimality of α.

Therefore, the alignment from C_1 to C_2 corresponding to $\beta|_{C_1,a}$, followed by $\alpha|_{a,b}$, followed by $\beta|_{b,C_2}$ is optimal.

□

5.3.2 Method of Calculation While Omitting the Irrelevant Parts

Now we can give the method of calculation while omitting the irrelevant parts for the cases of $d_1 \times d_2 \geq 0$ in Section 5.3.1.

1. $d_1 \leq 0$ and $d_2 \leq 0$.

   In this case, the original path α passes into the top right part of the sub-matrix between C_1 and C_2, and divides the sub-matrix into two parts, X and Y. We can omit the calculation in part Y, and only calculate part X as shown in Figure 5.10.

   First, we mark each position on the original path. Then we perform a straightforward dynamic programming in the sub-matrix between C_1 and C_2. Starting from C_1, we calculate row by row. In each row, we calculate from left to right until we hit a marked position in this row which is on the original path, then we stop calculating in this row and start in the next row in the same manner until we end at point C_2. The process of calculating is also shown in Figure 5.10.

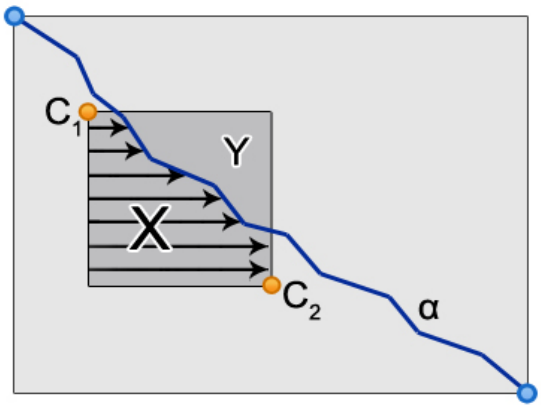


Figure 5.10: The method of calculation when $d_1 \leq 0$ and $d_2 \leq 0$.

2. $d_1 \geq 0$ and $d_2 \geq 0$.

   In this case, the original path α passes into the left bottom part of the sub-matrix between C_1 and C_2, and divides the sub-matrix into two parts, X and Y. We can omit the calculation in part Y, and only calculate part X as shown in Figure 5.11.

   First, we also mark each position on the original path, then perform a straightforward dynamic programming in the sub-matrix between C_1 and C_2. Starting from C_1, instead of calculating row by row, we calculate column by column in this case. In each column, we calculate from top to bottom until we hit a marked position in this column which is on the original path, then we stop calculating in this column and start in the next column in the same manner until we end at point C_2. The process of calculating is also shown in Figure 5.11. Indeed, it does not matter as to whether we calculate dynamic programming matrices in a row-by-row, or a column-by-column fashion.

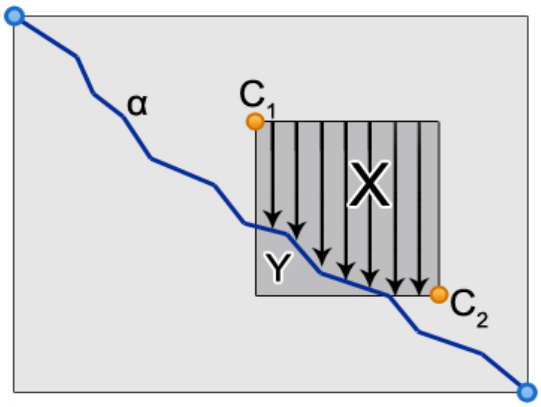


Figure 5.11: The method of calculation when $d_1 \geq 0$ and $d_2 \geq 0$.

We just discussed in detail the method of calculation while omitting the irrelevant part, and now let us look at a more general example to show this speed-up still guarantees optimality.

Example 8. *In Figure 5.12, the parts labelled A, B can be eliminated from any matrix calculation, as we are only interested in the alignment going through the constraints. But also, sections labelled C, D, E, F can be ignored. For example, we do not need to recalculate part D as introduced in class 1. One can use the original scores and calculation to find the optimal alignment from 2 to 3 without recalculating the values in D. Hence, when performing dynamic programming row-by-row, one can stop calculating at the position where the original path hits each row. Part E can be omitted from calculating as introduced in class 2, by performing dynamic programming in a column-by-column fashion.*

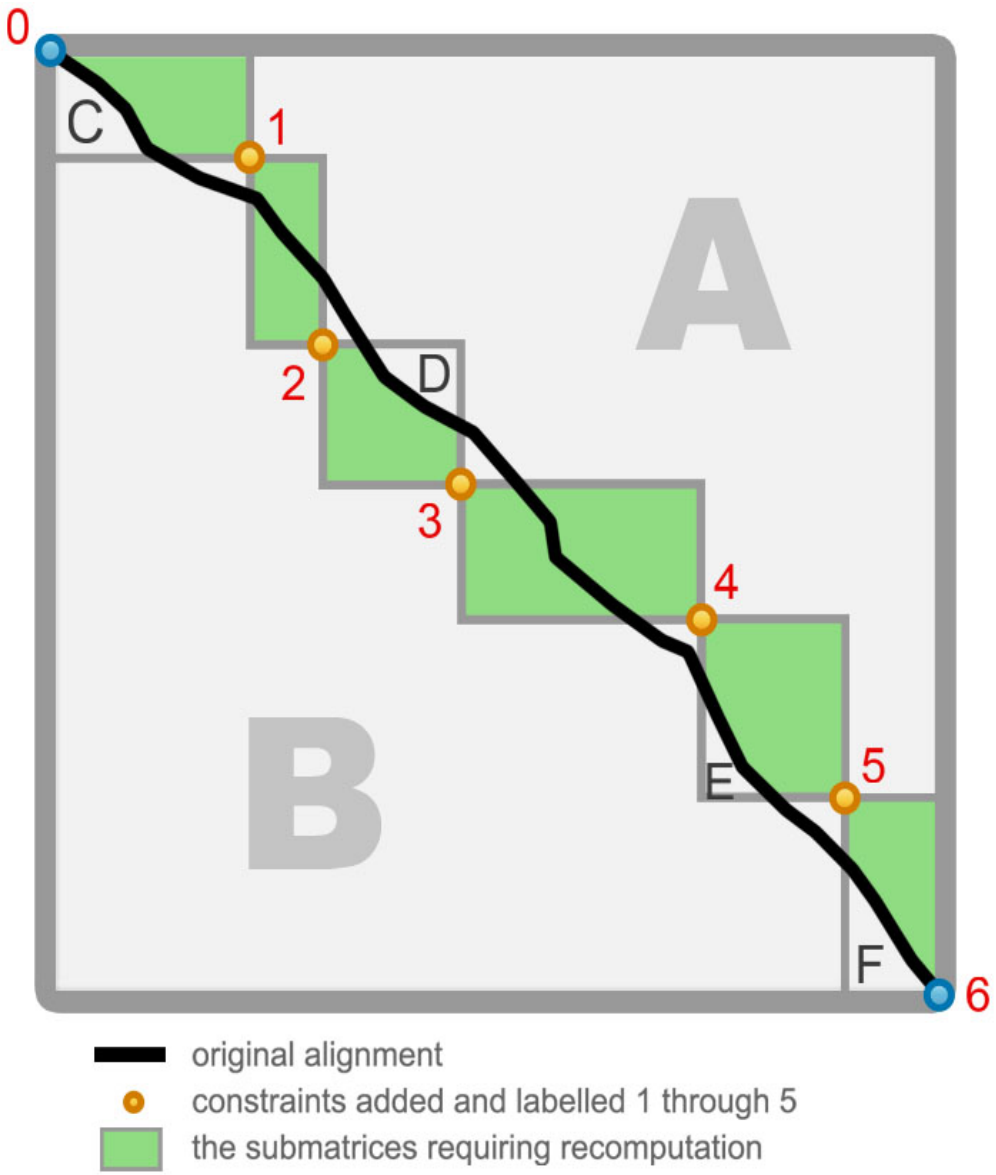


Figure 5.12: The information from the original alignment can be used in calculating the new alignment.

5.3.3 The Amount of Calculation

In this subsection, let us examine the amount of calculation that we need to do in the DP matrix, excluding the irrelevant parts.

Because the original path has an exponential number of possibilities (with three directions which are up, down and diagonal) and can appear anywhere in the matrices, it is complicated to consider all the possible cases or an average case, so we will calculate some special cases of interest. If we assume in a simplified condition, the original path is along the diagonal of the entire matrix (which means no gaps are inserted in the original alignment) as shown in Figure 5.13.

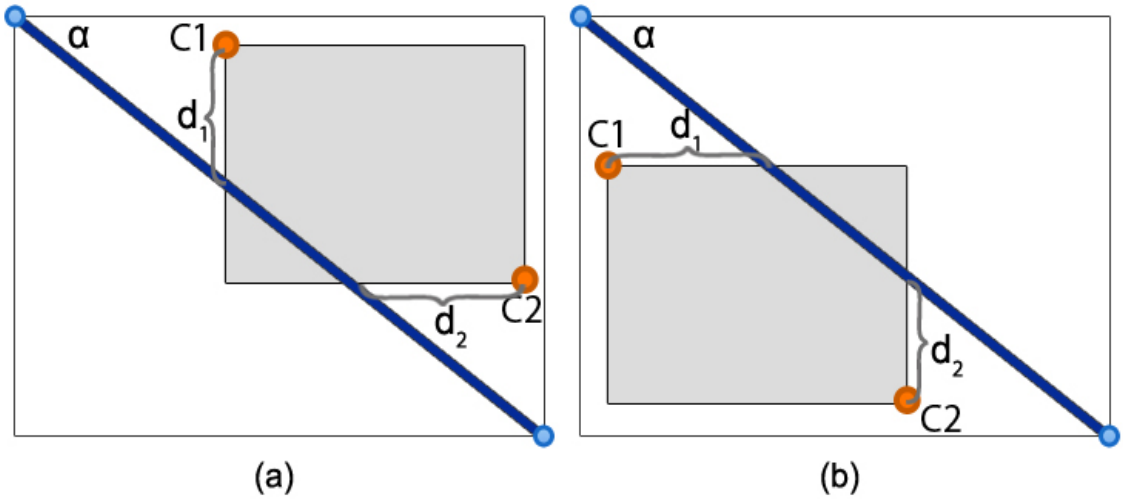


Figure 5.13: Pictured above is a simplified condition that the original path is a diagonal.

Under this assumption, the amount of calculation depends on the offset of the original path and the constraint points. Suppose the two sequences are s_k and s_l, the two consecutive constraint points are $C_1 = \{(k, i_1), (l, j_1)\}$ and $C_2 = \{(k, i_2), (l, j_2)\}$ which form the two diagonal corners of a sub-matrix, where $1 \leq i_1, i_2 \leq |s_k|$, $1 \leq j_1, j_2 \leq |s_l|$. For the cases that the original path does go through the sub-matrix in cases 1 to 3 in Section 5.3.1, suppose the offset between C_1 to the original path is d_1, and the offset between C_2 to the original path is d_2, we have the amount of calculation Q for a sub-matrix:

$$\mathrm{Q} \simeq (i_2 - i_1)(j_2 - j_1) - \frac{(i_2 - i_1 - d_1)(j_2 - j_1 - d_2)}{2}$$

One can easily see that, the closer that the constraint points are to the original path, the less we need to recalculate.

5.4 Complexity

The complexity of the straightforward dynamic programming algorithm for aligning sequences of lengths n_1 and n_2 respectively is superlinear with the sequence sizes; in this case it is $O(n_1 \times n_2)$ [14]. For large matrices, the sum of the time to perform alignment of a series of sub-matrices will be much less than the total time to do sequence alignment on the entire matrix.

In our problem, the time required to get the new alignment can be reduced by:

1. The number of constraints.

   We divide the dynamic programming matrix into several sub-matrices regularized by the CCS points in the process of calculation. We found that to get the alignment satisfying a CCS, only the scores in the shaded sub-matrices of Figure 5.2(b) need to be calculated, which means that the amount of calculation required to obtain the new alignment decreases as the number of constraints increases.

2. The position of the CCS points within the dynamic programming matrix.

   The specific amount depends on the position of the CCS points on the dynamic programming matrix. For the case of only one point on the matrix, the amount of calculation changes as the varying position of this point as represented in Figure 5.14. The amount of calculation increases as the grey becomes darker, so that the point in area 1 or 3 needs more calculation than the one in area 2 or 4.

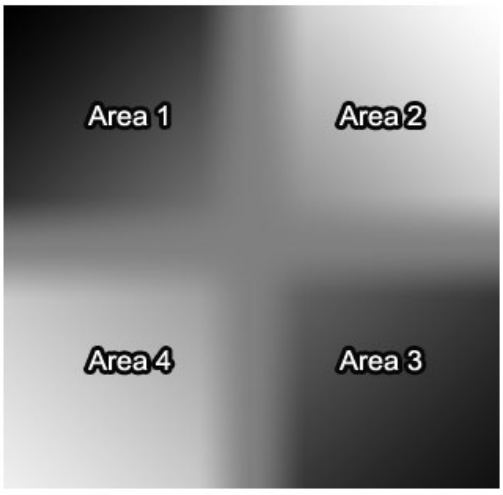


Figure 5.14: A visualization of the relationship of the amount of computation with the position of a single constraint point on the matrix. The darker the grey is, the more computation required.

Now suppose there are k points on the matrix. The new path should traverse all these points, and thus the best case is when the points are "close" to the top-right or bottom-left corners, and the worst case is when they are "close" to the top-left or bottom-right corners. An example of $m = 15, n = 15, k = 4$ is given in Figure 5.15, where m, n are the lengths of the two sequences, and k is the number of constraints.

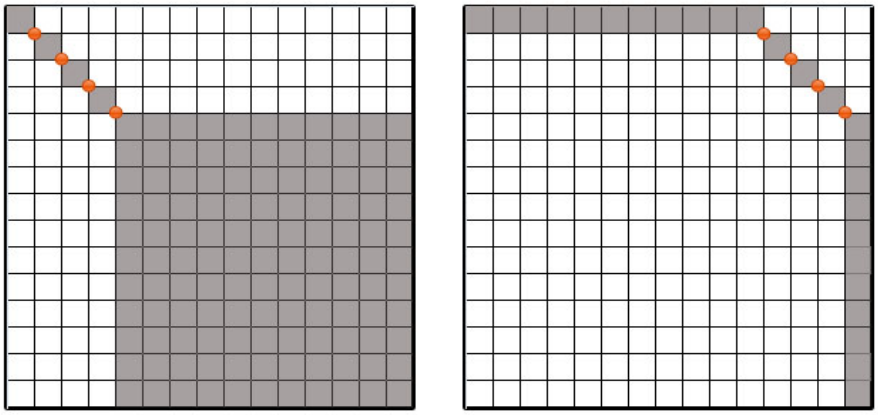


Figure 5.15: We provide two examples with two sequences of length 15, and 4 points on the matrix. (a) the worst case; (b) the best case.

Then, with k points on the matrix, the best case is

$$(n_1 - k) + (n_2 - k) + (k - 1) = n_1 + n_2 - (k + 1) \ ,$$

and the worst case is

$$k + (n_1 - k) \times (n_2 - k).$$

3. The position of the original alignment within the sub-matrices regularized by CCS points. We also noticed in Section 5.3, that the information from the original alignment is very useful in reducing the complexity for global alignment. The amount of calculation depends on the position of the original path within the sub-matrix. This could be a significant speed gain.

4. The offset between the CCS points and the original path, as introduced in the case of global alignment in Section 5.3.3.

In total, for two sequences s_k and s_l, if there exists a constant Z, such that for every two consecutive constraints $C_1 = \{(k, i_1), (l, j_1)\}$ and $C_2 = \{(k, i_2), (l, j_2)\}$ with the constraint points (i_1, j_1) and (i_2, j_2), either $(i_2 - i_1) \leq Z$ or $(j_2 - j_1) \leq Z$, then the time complexity to compute the alignment between s_k and s_l satisfying C_1 and C_2 is $O(|s_k| + |s_l|)$. This essentially means that the grey boxes in Figure 5.15 are all "close enough" to lines, that the entire time complexity is linear.

Chapter 6

Multiple Sequence Alignment Satisfying A Compatible Constraint Set

Multiple sequence alignments are a powerful way to study biological sequences, are widely used in the areas of DNA and protein sequence analysis, and are a natural extension of pairwise sequence alignments. In this chapter, we will extend the method of finding pairwise sequence alignments satisfying a CCS in Chapter 5 to multiple sequences.

We have discussed algorithms to detect if a set of constraints is a CCS or not for multiple sequences, using a graph algorithm in Chapter 4. Based on this graph formulation, we will now establish a correspondence between a CCS and multiple sequence alignments.

As mentioned in Section 3.3.2, three different techniques used to calculate multiple alignments are exact, iterative, and progressive algorithms. We will focus on exact and progressive algorithms in this chapter. This work applies to both global and local alignment with linear gap penalties.

6.1 Exact Algorithms for Alignments Satisfying a CCS with Three Sequences

Exact algorithms attempt to simultaneously align multiple sequences and find the optimal answer given a scoring scheme, which is especially useful when dealing with sets of divergent sequences.

In Section 3.3.1, we introduced a known method of calculating an optimal alignment of n sequences using an n-dimensional matrix. In this subsection, we will discuss how to calculate an alignment that satisfies a CCS using a multi-dimensional matrix. First, let us look at an example of aligning three sequences s_1, s_2, s_3 satisfying three compatible constraints C_1, C_2, C_3, as shown in Figure 6.1.

To calculate the optimal alignment in this example, we need a cube (or a 3D matrix) as shown in Figure 6.2. The constraints can be represented as lines inside the cube, which are called *constraint lines*. Indeed, as each constraint is only between two positions of the three sequences, they are lines in the 3-dimensional matrix rather than points. The three constraints are given by three lines $C_1'C_1''$, $C_2'C_2''$, $C_3'C_3''$ in the figure. To calculate the optimal alignment satisfying the constraints is

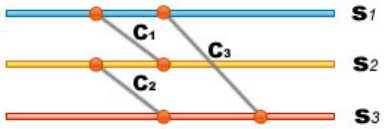


Figure 6.1: In this example, we have s_1, s_2, s_3 as sequences to be aligned, and C_1, C_2, C_3 as compatible constraints.

to calculate the optimal path from position A to position B while crossing all three constraint lines.

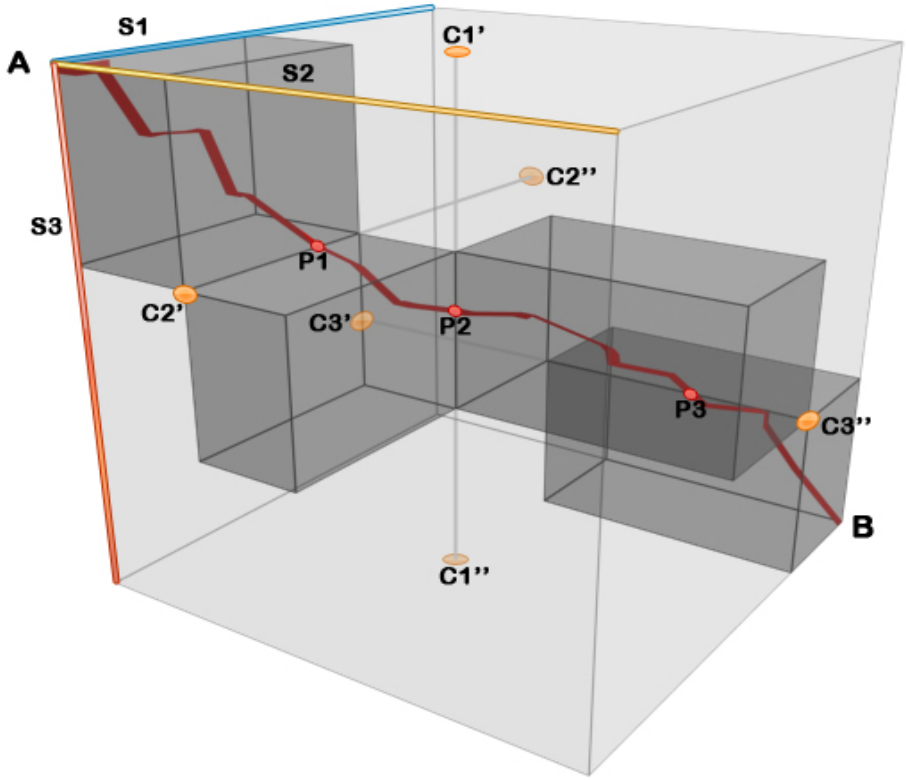


Figure 6.2: A 3-dimensional matrix is used to calculate the optimal global alignment of s_1, s_2, s_3 satisfying compatible constraints C_1, C_2, C_3. An optimal global path is from position A to position B, and the constraints can be represented as lines inside the cube. The optimal path must cross all lines in order to satisfy the constraints.

6.1.1 Method of Calculation

The basic idea in finding the optimal alignment satisfying the constraints is to force the path to cross the constraint lines and yield the highest score. We can implement this with the strategy of not assigning scores to the paths that do not cross the constraint lines, so that the highest scoring path would cross the lines automatically.

We will use global sequence alignment, although it works for local sequence alignment as well. First, we need to define a function $g(i, j, k)$, because it is useful for defining the recurrence.

Definition 22. *Let $g(i, j, k)$ be a function on three sequences s_1, s_2 and s_3 with X as a CCS, where $1 \leq i \leq |s_1|$, $1 \leq j \leq |s_2|$, $1 \leq k \leq |s_3|$. Then $g(i, j, k)$ is true if the constraints $\{(1, i), (2, j)\}$, $\{(2, j), (3, k)\}$, $\{(1, i), (3, k)\}$, together with X are compatible with s_1, s_2, s_3, and false otherwise.*

In other words, $g(i, j, k)$ is true if it is possible to have the characters $s_1(i)$, $s_2(j)$, $s_3(k)$ be aligned together and still satisfy the constraints. This is important, as we are only interested in calculating the values at each position (i, j, k) of the matrix if $g(i, j, k)$ is true. In the example of Figure 6.2, $g(i, j, k)$ is *true* when the point is in the shaded sub-cubes, and it is $false$ in other regions because the new optimal path that satisfies the constraints will not pass into those regions. A straightforward algorithm to test if $g(i, j, k)$ is true for each (i, j, k) is to test whether the three constraints of Definition 22 are compatible with the CCS. However, this is not efficient to do for each (i, j, k). We leave an efficient method of finding out the values of $g(i, j, k)$ in the entire matrix for future work.

The straightforward dynamic programming method calculates the alignment between three sequences using the Equation (6.1), where M is a 3-dimensional dynamic programming matrix, and s is the scoring matrix.

The base case is $M(0, 0, 0) = 0$. Then we can fill in items in the cube iteratively following the recurrence. We define $M(i, j, k)$ to be defined if $g(i, j, k)$ is true. Then $M(i, j, k) =$

$$
max \begin{cases}
M(i-1, j-1, k-1) + s(x_i, y_j, z_k), & \text{if } i, j, k > 0 \,\&\, g(i-1, j-1, k-1), & ① \\
M(i-1, j-1, k) + s(x_i, y_j, -), & \text{if } i, j > 0,\ k \geq 0 \,\&\, g(i-1, j-1, k), & ② \\
M(i-1, j, k-1) + s(x_i, -, z_k), & \text{if } i, k > 0,\ j \geq 0 \,\&\, g(i-1, j, k-1), & ③ \\
M(i-1, j, k) + s(x_i, -, -), & \text{if } i > 0,\ j, k \geq 0 \,\&\, g(i-1, j, k), & ④ \\
M(i, j-1, k-1) + s(-, y_j, z_k), & \text{if } j, k > 0,\ i \geq 0 \,\&\, g(i, j-1, k-1), & ⑤ \\
M(i, j-1, k) + s(-, y_j, -), & \text{if } j > 0,\ i, k \geq 0 \,\&\, g(i, j-1, k), & ⑥ \\
M(i, j, k-1) + s(-, -, z_k), & \text{if } k > 0,\ i, j \geq 0 \,\&\, g(i, j, k-1), & ⑦ \\
undefined, & \text{if none of the above apply.}
\end{cases}
\tag{6.1}
$$

We marked each option in Equation (6.1) with a number, which corresponds to that of Figure 6.3 to better understand the computation scheme.

The reason we have $g(i, j, k)$ in the recurrence is because we are not willing to assign scores to the paths that do not cross the constraint lines, and we know that with the constraints added, the regions that the new path might pass into will be defined by $g(i, j, k)$. Thus we have $g(i, j, k)$ in the *if condition* part of each case in the recurrence, in order to restrict the path to be only in the region where is defined $g(i, j, k)$. This means the path would cross the constraint lines

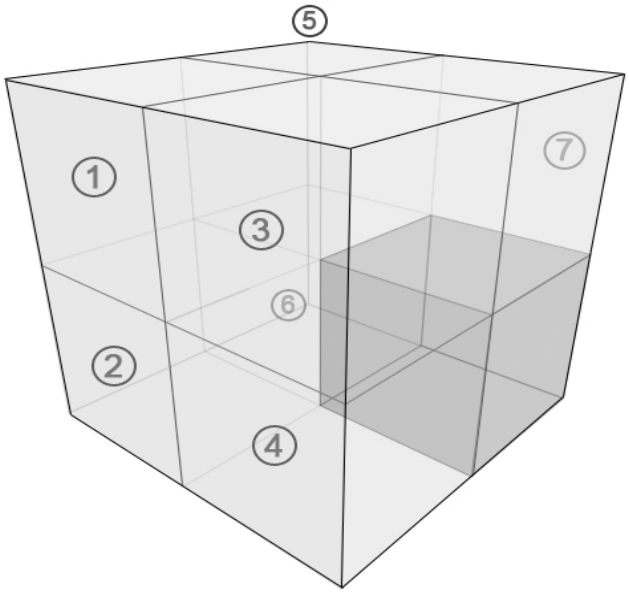


Figure 6.3: A 3-dimensional dynamic programming computation scheme which calculates the value using the other seven values beside it. The value of interest is the shaded box which assumes we have calculated the other boxes.

automatically. Having $g(i, j, k)$ in the recurrence is different from the usual recurrence, because for each calculation of $M(i, j, k)$, we can take any of the seven options in the usual recurrence; whereas with $g(i, j, k)$ in the *if condition*, we only take the options when their corresponding $g(i, j, k)$ is true, so that we may have less than seven options.

Intuitively, we should fill in the cube iteratively, and stop calculating every time we hit a character on the constraint lines, and start filling in the next sub-cube using the scores on the constraint line as the initial values.

In Figure 6.1, suppose the three constraints are $C_1 = \{(1, i_1), (2, j_1)\}$, $C_2 = \{(2, j_2), (3, k_2)\}$, $C_3 = \{(1, i_3), (3, k_3)\}$, and we can take a close look at the calculation of the first two sub-cubes in Figure 6.4. We will firstly hit C_2 while calculating the first sub-cube, and hit C_1 while calculating the second sub-cube.

After initialization, we start filling in the first sub-cube from the start point A following Equation (6.1). We will first hit j_2 on s_2 when we fill in the upward surface row by row, then we stop and go to the next surface beneath. We fill it in plane by plane as k increases until we hit k_2 on s_3, and the calculation for the first sub-cube stops.

The method of the calculation of the second sub-cube is similar to the first one. But we know that $g(i, j, k)$ is $false$ outside the shaded cubes, thus except for the constraint line $C_2'C_2''$ as the initial values of the second sub-cube, for the upward surface of the second sub-cube, scores coming from above are not defined, and therefore, instead of taking the 7 possibilities in Equation (6.1), we only need to find the maximum value of ②, ④, ⑥. For the leftward surface, we remove the scores

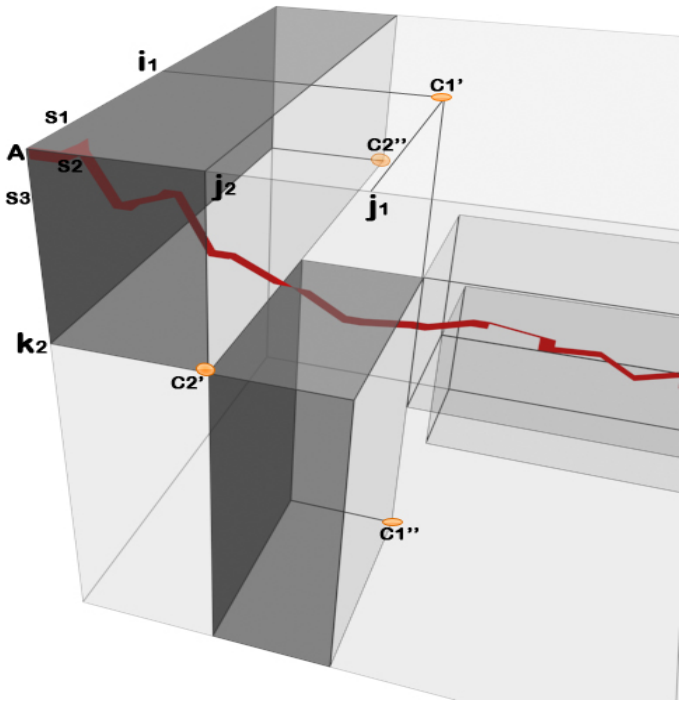


Figure 6.4: A close look at the 3-dimensional matrix to calculate the optimal alignment of S_1, S_2, S_3 satisfying compatible constraints C_1, C_2, C_3.

coming from the left, and only calculate the values of ③, ④, ⑦. For the front surface, we remove the scores coming from the front, and only calculate the values of ⑤, ⑥, ⑦. Having the values on the surfaces, we can now fill in the second sub-cube following Equation 6.1 until we hit i_1 on s_1 then j_1 on s_2.

Using the same method, all the sub-cubes can be filled in. As a result, the new optimal path will start at position A, end at position B and cross the constraint lines automatically.

In general, it seems difficult to give a precise algorithm that only calculates $M(i, j, k)$ exactly when $g(i, j, k)$ is defined. Thus, we leave an efficient algorithm using the recurrence of Equation (6.1) as future work.

6.1.2 Complexity

The complexity of calculating the multiple alignment between three sequences s_1, s_2 and s_3 using exact algorithms can be done in $O(|s_1| \times |s_2| \times |s_3|)$. After adding constraints to the alignment, the amount of the calculation of the new alignment varies as the number of constraints changes, but is still $O(|s_1| \times |s_2| \times |s_3|)$ in the worst case.

6.2 Progressive Algorithms for Multiple Sequence Alignment Satisfying a CCS

Progressive algorithms allow large alignments of distantly related sequences to be constructed quickly and simply. Some dramatic improvements have been made to the methodology with respect to speed, and capacity to deal with large numbers of sequences and accuracy.

With progressive algorithms, there are usually two steps: the first is to build a guide tree to guide the alignment, then the second step is to use a series of pairwise alignments to align larger and larger groups of sequences, following the branching order of the guide tree. In the following subsections, we will discuss the steps in detail.

6.2.1 Build a Guide Tree

The algorithm is based on building the full alignment progressively, using the branching order of a "guide tree" to guide the alignments. As the name suggests, this tree is meant to guide the clustering process. ClustalW includes steps to build a guide tree, which is relatively efficient. In this subsection, we will take an interest in building a tree as done in ClustalW [31].

In order to build a guide tree, there are two stages:

1. Aligning all pairs of sequences separately to calculate pairwise distances.

   There are many different ways of defining distance. In ClustalW, it calculates scores of each pairwise alignment using dynamic programming, and these scores represent distances between each pair of sequences. Then it can create an $n \times n$ matrix from the distances, where n is the number of sequences, as shown in the example in Figure 6.5 (a).

2. Making a guide tree from pairwise distances.

   A rooted binary tree (Figure 6.5 (c)) can be calculated from the distance matrix of stage 1 using a hierarchical clustering method such as the Neighbour-Joining method, or UPGMA in [31].

6.2.2 Progressively Align Multiple Sequences

The basic procedure at this stage is to use a series of pairwise alignments to progressively align groups of sequences, following the branching order in the guide tree. We proceed from the leaves of the rooted tree towards the root.

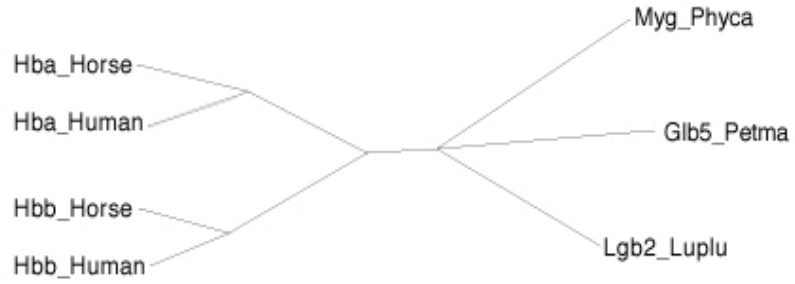
Each step consists of aligning two existing alignments or sequences.

1. Align two sequences.

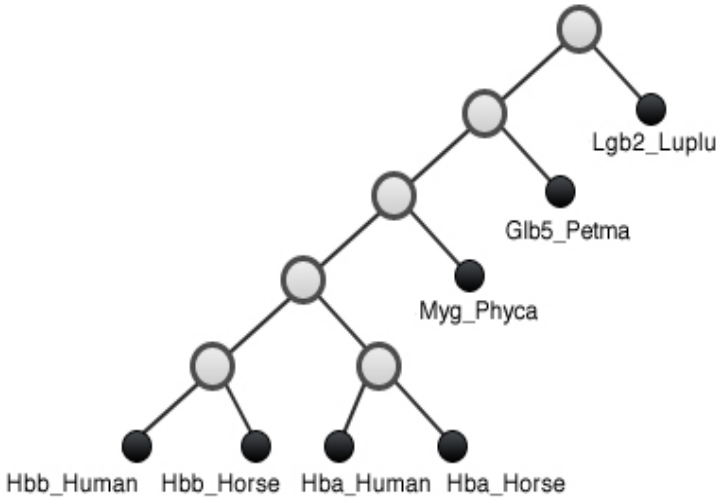
   When aligning two sequences with constraints between them, we follow the algorithms in

Hbb_Human	1	-					
Hbb_Horse	2	.17	-				
Hba_Human	3	.59	.60	-			
Hba_Horse	4	.59	.59	.13	-		
Myg_Phyca	5	.77	.77	.75	.75	-	
Glb5_Petma	6	.81	.82	.73	.74	.80	-
Lgb2_Luplu	7	.87	.86	.86	.88	.93	.90	-
		1	2	3	4	5	6	7

(a)



(b)



(c)

Figure 6.5: The procedure of building a guide tree adapted from [31]. (a) The distance matrix is calculated from all pairwise alignments. (b) The unrooted tree made from the distance matrix using the Neighbour-Joining method. (c) The rooted tree converted from the unrooted tree.

Chapter 5 to perform a pairwise sequence alignment satisfying CCS.

2. Align an alignment with a sequence.

   When aligning an alignment with a sequence, we use a consensus sequence (Definition 9) of an alignment. In our algorithm, we use the character that occurs at least as many times as any other characters at this position. Notice that it is possible to have more than one consensus. If there is more than one option, we pick one arbitrarily.

   Example 9. *A consensus for*

$$\begin{array}{c} A\ C\ A\ G\ T\ A\ G \\ A\ C - -\ T\ C\ G \\ A\ G - -\ G\ C\ G \\ A\ G\ A\ C\ T\ G\ C \end{array}$$

   could be any one of ACAGTCG, ACACTCG, AGAGTCG, AGACTCG.

   Therefore, we need to convert the constraints from the sequences to the consensus sequences, and then perform a pairwise alignment satisfying these constraints.

3. Align two alignments.

   When aligning two alignments, we can first find the consensus sequences of the two alignments separately, and convert the constraints from sequences to the consensus sequences, then perform a pairwise alignment satisfying these constraints using the algorithms introduced in Chapter 5.

Here is an example to show the idea of how to align the three sequences with a CCS in Figure 6.1.

Example 10. *The sequences to be aligned are s_1, s_2 and s_3, and the CCS consists of three constraints: C_1, C_2 and C_3, as shown in Figure 6.1. The guide tree calculated from the pairwise distances is shown in Figure 6.6, where s_1 and s_2 are constrained by C_1, s_2 and s_3 are constrained by C_2, and s_1 and s_3 are constrained by C_3. Then we enforce C_2 and C_3 with R', the consensus of s_1 and s_2, aligned with s_3.*

In this example, the procedure of performing the multiple sequence alignment satisfying CCS includes the following steps: first, build a guide tree using the distances calculated from dynamic programming of each pair of sequences; second, align two sequences s_1 and s_2 constrained by C_1; third find the consensus sequence R' of the alignment of s_1 and s_2, and convert C_2 from s_2 to R', convert C_3 from s_1 to R'; last, align the consensus sequence R' with s_3 constrained by C_2, C_3.

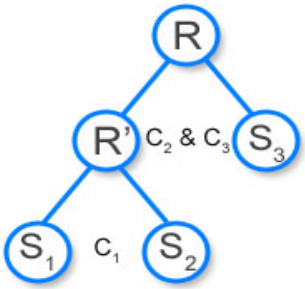


Figure 6.6: The guide tree of s_1, s_2 and s_3, with the constraints C_1, C_2, C_3.

6.2.3 The Amount of Calculation

If we store the results of every step when originally progressively aligning the sequences, and after adding constraints to the alignment, we only need to recalculate some of the pairwise alignments.

First let us look at a definition that is useful for the rest of this section. The *lowest common ancestor (LCA)* [2] is a concept in graph theory and computer science.

Definition 23. *Let T be a rooted tree with n nodes. The lowest common ancestor (LCA) is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).*

The LCA of v and w in a tree T is the shared ancestor of v and w that is located farthest from the root. As in Figure 6.7, the LCA of s_1 and s_2 is 3, the LCA of s_3 and s_4 is 2, and the LCA of s_2 and s_5 is 1.

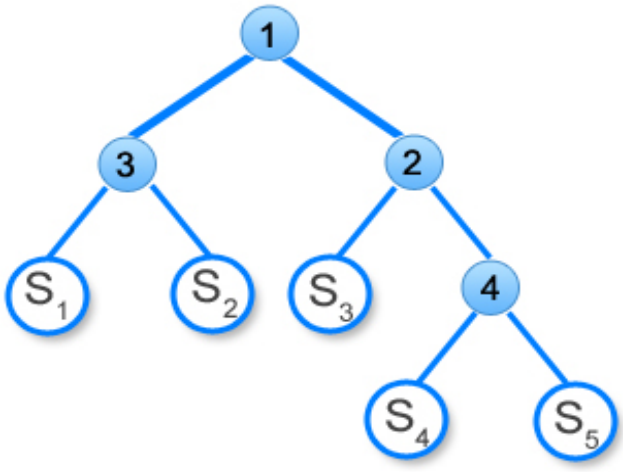


Figure 6.7: An example to demonstrate the concept of LCA.

After adding constraints to an alignment, we only need to recalculate starting at the LCA of the

two sequences that are constrained, upwards towards the root. For the example in Figure 6.7, if the constraint is between s_1 and s_5, all calculation under node ① remains the same, but the constraint between s_1 and s_5 involves calculating consensus ④, ② and ③, as before with no constraints between, and we must recalculate the pairwise alignment of consensus ③ and consensus ②, enforcing the constraint between s_1 and s_5 with appropriate positioning of the consensus.

Chapter 7

Conclusion and Future Directions

7.1 Summary of the Thesis

In the thesis, we introduced the background knowledge and the motivation of our work in the first three chapters. Then we tried to find optimal sequence alignments satisfying some constraints, which can be created based on expert user's knowledge, experience or needs.

We first formally defined a constraint, compatible constraints and a compatible constraint set (CCS) in Chapter 4. Then on the basis of the definitions, we designed algorithms using a depth-first search on a directed graph that is converted from the constraints and the alignment, in order to determine if a set of constraints is compatible or not. If the constraints are compatible, we can add them to the alignment to force the new alignment to satisfy them while yielding the highest score.

Having the compatible constraint set, in Chapter 5, we gave a procedure to perform a pairwise sequence alignment satisfying a CCS. The algorithm can be sped up depending on the number and position of the constraints, and in the case of global alignment, we can also omit some of the score calculations in the dynamic programming matrix using the information from the original alignment, which can also be a big speed gain.

In Chapter 6, we developed algorithms for multiple sequence alignment satisfying a CCS with three sequences based on exact algorithms, and then with multiple sequences based on progressive algorithms.

We also leave some open questions in the thesis for future research.

7.2 Future Work

In the future, there are several directions to extend the work of the thesis.

1. Extending the definition of a constraint, and allowing users to input constraints without exact positions, but with regions.

2. Based on point 1, we will develop new algorithms to determine the compatibility of the more general constraints.

3. An efficient algorithm to find which values of $g(i, j, k)$ are true, and calculate exactly $M(i, j, k)$ where $g(i, j, k)$ is true.

4. An exact algorithm for multiple sequence alignment with more than three sequences satisfying a CCS rather than only for three sequences.

5. For the progressive algorithm for multiple sequence alignment satisfying a CCS, it is possible to dynamically adjust the guide tree after adding constraints, to update the order of clustering sequences.

6. One can automatically find the underlying common segments in various databases, such as a database of known domains, and add them as constraints to an alignment.

7. Add other types of constraints that force the algorithm to not match certain positions.

References

[1] A.V. Aho, J.E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, USA, 1983.

[2] AV Aho, JE Hopcroft, and JD Ullman. On finding lowest common ancestors in trees. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 253–265. ACM New York, NY, USA, 1973.

[3] J.M.S. Bartlett and D. Stirling. A short history of the polymerase chain reaction. *DNA*, 226:3–6.

[4] CLC bio. CLC Sequence Viewer. www.clcbio.com.

[5] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, 48(5):1073–1082, 1988.

[6] M. Clamp, J. Cuff, S.M. Searle, and G.J. Barton. The jalview java alignment editor. *Bioinformatics*, 20(3):426–427, 2004.

[7] Wayne Clarke. Paper in preparation. 2008.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT press, USA, 2001.

[9] R. Durbin, S.R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, 1998.

[10] R.C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–1797, 2004.

[11] R.C. Edgar and S. Batzoglou. Multiple sequence alignment. *Current Opinion in Structural Biology*, 16(3):368–373, 2006.

[12] C.A.R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[13] F. Jeanmougin, J.D. Thompson, M. Gouy, D.G. Higgins, and T.J. Gibson. Multiple sequence alignment with Clustal X. *Trends in Biochemical Sciences*, 23(10):403–405, 1998.

[14] N.C. Jones and P.A. Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT Press Cambridge, Mass, 5 Cambridge Center, Cambridge, 2004.

[15] I. Korf, M. Yandell, and J. Bedell. *BLAST*. O'Reilly Media, Inc., Sebastopol, CA, USA, 2003.

[16] R.R. Korfhage. *Discrete Computational Structures*. Academic Press, Inc. Orlando, FL, USA, USA, 1983.

[17] A.M. Lesk. *Introduction to Bioinformatics*. Oxford University Press Oxford, New York, 2002.

[18] D.J. Lipman, S.F. Altschul, and J.D. Kececioglu. A tool for multiple sequence alignment. *Proceedings of the National Academy of Sciences*, 86(12):4412–4415, 1989.

[19] Miroslav Martinovic. CMSC 410: Advanced Algorithms, 2001. Department of Computer Science, The College of New Jersey.

[20] B. Morgenstern, K. Frech, A. Dress, and T. Werner. DIALIGN: finding local similarities by multiple sequence alignment. *Bioinformatics*, 14(3):290–294, 1998.

[21] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[22] C. Notredame. Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, 3(1):131–144, 2002.

[23] C. Notredame. Recent evolutions of multiple sequence alignment algorithms. *PLoS Computational Biology*, 3(8):e123, 2007.

[24] C. Notredame, D.G. Higgins, and J. Heringa. T-coffee: a novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology*, 302(1):205–217, 2000.

[25] T. Pawson and J. Schlessinger. SH2 and SH3 domains. *Current Biology*, 3:434–434, 1993.

[26] J. Pei and N.V. Grishin. PROMALS: towards accurate multiple sequence alignments of distantly related proteins. *Bioinformatics*, 23(7):802–808, 2007.

[27] J. Pei, B.H. Kim, and N.V. Grishin. PROMALS3D: a tool for multiple protein sequence and structure alignments. *Nucleic Acids Research*, 36(7):2295–2300, 2008.

[28] B. Qian and R.A. Goldstein. Distribution of indel lengths. *PROTEINS: Structure, Function, and Genetics*, 45(1):102–104, 2001.

[29] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[30] S.L. Teal and A.I. Rudnicky. A performance model of system delay and user strategy selection. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 295–305. ACM New York, NY, USA, 1992.

[31] J.D. Thompson, D.G. Higgins, T.J. Gibson, et al. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4673, 1994.

[32] J.D. Thompson, F. Plewniak, and O. Poch. A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research*, 27(13):2682–2690, 1999.

[33] I.M. Wallace, G. Blackshields, and D.G. Higgins. Multiple sequence alignments. *Current Opinion in Structural Biology*, 15(3):261–266, 2005.

[34] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *J. Comput. Biol*, 1(4):337–348, 1994.

[35] Sheng Wang. Paper in preparation. 2010.

[36] A.M. Waterhouse, J.B. Procter, D.M.A. Martin, M. Clamp, and G.J. Barton. Jalview Version 2–a multiple sequence alignment editor and analysis workbench. *Bioinformatics*, 25(9):1189–1911, 2009.