Divide-and-Conquer Sequence Alignment Within a Specified Band

Emir Basic

Examensarbete för 20 p, Institutionen för datavetenskap,
Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in computer science, 20 credit points,
Department of Computer Science,
Faculty of Science, Lund University

Divide-and-Conquer Sequence Alignment Within a Specified Band

Abstract

Sequence comparison is a fundamental problem in computational molecular biology that aims to discover similarity relationships between biological sequences. An *alignment* of two or more sequences is a scheme in which sequences are placed on top of each other and spaces are inserted into the sequences to make them of equal lengths. An *optimal* alignment maximizes the value of a *similarity score function*. Given two sequences $s = s_1 s_2 \dots s_m$ and $t = t_1 t_1 \dots t_n$, the standard *dynamic programming* (*DP*) algorithm computes their optimal alignment in time and space $\mathcal{O}(mn)$. Hirschberg's recursive *divide-and-conquer* ($D\&C$) algorithm [6] finds an optimal alignment of s and t in space $\mathcal{O}(m+n)$ and time $\mathcal{O}(mn)$. The *DBand* algorithm for aligning within a constant band of width $2d+1$ determines an optimal alignment in time and space $\mathcal{O}(dn)$. By combining the $D\&C$ and banding technique, we have derived a linear-space algorithm *FastDCA* that spends $\mathcal{O}(dn)$ time in finding an optimal alignment in the d-strip. We have also generalized this algorithm to compute an optimal alignment of k sequences within a specified band in time $\mathcal{O}\left(k^2 2^k\, n d^{k-1} \log_2 n\right)$ and space $\mathcal{O}\left(kn + d^{k-1}\right)$, where n is length of shortest sequence and d is band radius. Given k similar sequences, the algorithm performs considerably faster than $\mathcal{O}\left(k^2 2^k n^k\right)$ of the dynamic programming in k dimensions when $d \ll n$.

Söndra-och-härska metod för aligning av sekvenser inom band

Sammanfattning

Sekvensjämförelser är ett grundläggande problem inom beräkningsbiologi med syfte att upptäcka likheter mellan biologiska sekvenser. En *aligning* är en upplinjering av två eller flera sekvenser för att visa fram likheter mellan dem. En *optimal* aligning maximerar värdet av en *scoring-funktion*. Givet två sekvenser $s = s_1 s_2 \dots s_m$ och $t = t_1 t_1 \dots t_n$, en optimal aligning kan beräknas med *dynamisk programmering* i tid $\mathcal{O}(mn)$ och utrymme $\mathcal{O}(mn)$. Hirschbergs rekursiva, *söndra-och-härska* ($D\&C$) algoritm [6] beräknar en optimal aligning av s och t i tid $\mathcal{O}(mn)$ och utrymme $\mathcal{O}(m+n)$. *DBand*-algoritm beräknar en optimal aligning inom d-band i $\mathcal{O}(dn)$ tid och utrymme, för någon konstant d. Genom att kombinera $D\&C$- och bandtekniken härledde vi en algoritm för aligning inom specifierat band som tar $\mathcal{O}(dn)$ tid och $\mathcal{O}(p\,d)$ minne. Vi har också generaliserat algoritmen för att beräkna en multialigning av k sekvenser i $\mathcal{O}\left(k^2 2^k\, n d^{k-1} \log_2 n\right)$ tid och $\mathcal{O}\left(kn + d^{k-1}\right)$ utrymme, där d är bandets radius. För k relaterade sekvenser uppträder algoritmen betydligt snabbare än $\mathcal{O}\left(k^2 2^k n^k\right)$ av dynamisk programmering i k-dimensioner, för $d \ll n$.

Contents

List of Figures

Chapter 1

Introduction

One of the great problems of biology is determining relationships between organisms or sequences that share similarity. At the most basic level, morphology of organisms has been used to make very broad comparisons. Recently, with the advent of molecular data, comparisons have been made at the molecular level. The standard way to compare molecular sequences is based on alignments.

1.1 Basic terms in molecular biology

This section gives a very brief overview of some basic terms from molecular biology, such as: cells, DNA, proteins, genes, chromosomes, genomes etc. For in-depth knowledge, we refer to some introductory text in molecular biology such as [1].

Cells are building blocks in any living organism. Advanced organisms, known as *eukaryotes*, possess a nucleus. In the nucleus of each cell, inside tiny structures called *chromosomes* lie DNA (deoxyribonucleic acid) molecules. DNA is the hereditary material that carries information about how an organism has developed from a single cell. The totality of hereditary information is the *genome* of an organism. DNA has the structure of a double stranded helix. DNA sequences can be represented as strings over the alphabet of four nucleic bases (nucleotides): adenine (A), guanine (G), cytosine (C) and thymine (T). These bases can only form two different base pairs (*bp*): A-T and G-C. DNA molecules can be millions of *bp* long. The human genome contains roughly 3 billion *bp*. About 3% of the DNA in a chromosome represents the basic units of heredity, known as *genes*. The human genome is estimated as comprising more than 30,000 genes. *Proteins* are molecules that are responsible for development of any organism. Most of functions in a cell are accomplished by proteins. For example, many proteins act as enzymes and catalyze chemical reactions. The building blocks of protein sequences comprise 20 amino acids. Proteins are constructed by translating a DNA sequence into a sequence of amino acids. A strand of DNA contains triplets of nucleotides, *codons*, that are encoded into amino acids. A sequence of codons is called the *genetic code*. A typical protein sequence contains 100-5000 amino acids.

Any organism has the same DNA in each of its cells. An important property of a DNA molecules is its *replication*. Before a cell divides, the DNA is unwound into two strands by the RNA polymerase enzyme. During this process, errors or *mutations* frequently occur. A mutation is *a substitution* or *a replacement* if an amino acid gets replaced by another amino acid. When an amino acid is added to or lost from DNA, the mutation is *an insertion* or *a deletion*, respectively. Insertion and deletions are sometimes called *indels*.

Protein synthesis consists of two phases: *the transcription* and *translation*. During the tran-

scription phase, a DNA molecule is unwound into two strands, which act as templates to form the messenger RNA (mRNA). The mRNA is the complementary strand to the DNA. In the translation phase, the codons of mRNA are matched with the particular amino acids that are added to a linear molecule to form a protein. This process, known as the *central dogma* in molecular biology, is completed in *ribosomes*.

1.2 Sequence comparison

The primary aim of sequence comparison is to discover similarity relationships between molecular sequences, in terms of substitutions and indels. The similarity between two sequences gives a measure of how similar sequences are, in a sense that the higher value indicates greater similarity.

1.2.1 Motivation

Although all living organisms have a common origin, they become distant in the process of evolution due to mutations frequently occurring in DNA sequences. The *first fact of biological sequence analysis* states that high sequence similarity usually implies strong functional or structural similarity. Two closely related DNA sequences that share a common evolutionary origin are usually more similar than two unrelated sequences. For instance, the human and mouse genomes are 85% similar. Therefore, when a new gene is sequenced in a labaratory, any analysis starts with searching various data banks for similar sequences. With this in mind, sequence comparisons may help in recovering functional, structural or evolutionary relationships between various organisms. Comparison of protein sequences may reveal their functionality, as a protein functionality is defined by its structure, which is in turn determined by the sequence of amino acids. We now introduce the computational problem that will be discussed in this thesis.

1.2.2 Computational problems

Similarities between two sequences can be detected by aligning them one above the other and inserting spaces into both sequences to make them of equal length. Finding the true alignment that describes the evolution of one sequence into the other is a difficult biological task and out of the scope of this thesis. Instead we will use a *rule* to assign a numerical value or *score* to any possible alignment between the sequences. An *optimal alignment* between two sequences is an alignment that achieves the highest score. This maximum score is called the *similarity* between the sequences. The problem of aligning two sequences while maximizing the value of a score function is called the *sequence alignment problem*. It is appropriate to mention that this thesis deals only with the task of finding global alignments as opposed to, say, local alignments, which are alignments between substrings of sequences. Chapter 2 examines current and new methods for computing optimal alignments of two sequences.

Simultaneously aligning more than two sequences, the so-called *multiple sequences alignment* or *multi-alignment*, is a natural generalization of the two-sequence alignment. The problem of computing optimal multi-alignments under the well-known SP-score is studied in Chapter 3.

1.3 Thesis goals

Our primary objective is the theoretical study of the alignment problem and, in particular, underlying optimization techniques. We are interested in exploring and combining the three basic algorithmic techniques: dynamic programming, divide-and-conquer, and branch-and-bound.

1.4 Previous results

An excellent introductory textbook on algorithms in computational biology is [9]. Our reviews of standard algorithms (Chapter 2, Sections 2.2 – 2.4, and Chapter 3, Sections 3.2 – 3.3) are mainly based on Chapter 3 of this book. The sequence alignment problem can be solved in time and space $O(mn)$ using the well-known *dynamic programming* technique. Hirschberg [6] introduced a divide-and-conquer method that reduces the quadratic space complexity of the dynamic programming algorithm to linear space. Charter et al. [4] implemented $FastLSA$ algorithm that is, as they they claim, superior to Hirschberg's algorithm. Arvestad [2] generalized Hirschberg's algorithm to k-dimensions. Aligning within a constant band is a popular technique for speeding up the dynamic programming algorithm. The problem of aligning within a constant band has been studied by many researchers [4, 7, 8]. Carrillo and Lipman [3] presented a branch-and-bound heuristic for multiple sequence alignment based on pairwise projections of multiple alignments. An implementation of this method is the MSA program which can align up to 6 sequences of size 200. To align somewhat longer sequences, Stoye [10] devised an efficient divide-and-conquer algorithm (DCA) for multiple sequence alignment, which uses MSA as subprocedure.

1.5 Main contributions

The main contribution is a divide-and-conquer algorithm for aligning two sequences within a specified band, and its generalization to k dimensions.

Our algorithm *BandDCA*, a hybrid of Hirschberg's method and the band-aligning algorithm, computes an optimal alignment in a specified band using $\mathcal{O}(dn \log_2 n)$ time and $\mathcal{O}(m+n)$ space. Its k-dimensional generalization, the algorithm *BandDCMA* computes an optimal alignment of k sequences within a specified band, using the SP measure, in time $\mathcal{O}\left(k^2 2^k\, n d^{k-1} \log_2 n\right)$ and space $\mathcal{O}\left(kn + d^{k-1}\right)$, where n is the length of longest sequence and d is a constant specifying the band radius.

Chapter 2

Pairwise sequence alignment

In this chapter, we examine various algorithmic techniques for the alignment problem, which is a typical combinatorial problem with many possible solutions of which we seek to find the *optimal* one, that is, a solution that optimizes the value of a scoring function.

Chapter 2 is organized as follows: In Section 2.1, we give a mathematical formulation of the alignment problem. Section 2.2 examines the standard dynamic programming algorithm for computing an optimal alignment of two sequences in quadratic time and space. Section 2.3 discuss Hirschberg's recursive algorithm for finding an optimal alignment in linear space. Section 2.4 reviews a $O(dn)$-time algorithm for aligning similar sequences within a band of width $2d+1$. In Section 2.5, we apply Hirschberg's method to a specified band obtaining an algorithm that can recover an optimal alignment in linear space and in $\mathcal{O}(dn \log_2 n)$ time. In Section 2.6, we improve the linear-space algorithm to deliver an optimal alignment within a band in $\mathcal{O}(dn)$ time.

2.1 Similarity and alignments

In this section, we give a precise formulation of the alignment problem. We start out by formalizing the concepts of the similarity and optimal alignments between two sequences.

An example alignment between two sequences is shown in Figure 2.1. We see that an alignment column may contain two identical characters (a *match*), two distinct characters (a *mismatch*) or a character and a space (an *insertion* or a *deletion*).

s' :	G	T	A	C	T	A	–	G
t' :	–	–	C	C	T	A	C	G

Figure 2.1. *Example alignment between sequences* $s =$ GTACTAG *and* $t =$ CCTACG.

Definition 2.1 (Alignment) Let s and t be two sequences over an alphabet Σ. An *alignment* between s and t is a pair of extended sequences $s', t' \in \Sigma^+ = \Sigma \cup \{-\}$. The alignment $\alpha = (s',\ t')$ must satisfy:

1. $|s'| = |t'|$.
2. After removal of all spaces, we have $s' = s$ and $t' = t$.
3. For all i, either $s'[i]$ or $t'[i]$ is not a space.

There are many possible ways to define a score function that assigns a value to an alignment of two sequences. A simple function assigns a score to each alignment column depending on its contents. For example, a column containing a match receives score $+1$, a column with a mismatch is given score -1, and a column containing an indel gets score -2 (*indel*). In general, we can define a score function $w : \Sigma^+ \times \Sigma^+ \mapsto \mathbb{R}$ for a pair of characters. Our column scores can now be rewritten as: $w(x, y) = 1$ if $x = y$, $w(x, y) = -1$ if $x \neq y$ and $w(x, -) = w(-, x) = -2$.

Definition 2.2 (Alignment score) Let α be an alignment of s and t, and let w be a score function. Then, the score of α, denoted $score(\alpha)$, is equal to the sum of column scores:

$$score(\alpha) = \sum_{i=1}^{|\alpha|} w(s[i], t[i]). \tag{2.1}$$

Definition 2.3 (Optimal alignment score or similarity) The *optimal score* of an alignment of s and t, also called the *similarity* and denoted $sim(s, t)$ for sequences s and t, is the highest score of any alignment of s and t, i.e.

$$sim(s, t) = \max_{\alpha \in \mathcal{A}(s,t)} score(\alpha),$$

where $\mathcal{A}$ (s,t) is the set of all alignments between s and t.

Definition 2.4 (Optimal alignment) *An optimal alignment*, denoted $opt(s, t)$ for sequences s and t, is an alignment with the highest score.

Notice that the alignment score depends greatly on the choice of scoring function. With our choice, the score of the above example alignment is obtained as follows:

$$4 * (1) + 1 * (-1) + 3 * (-2) = -3,$$

since the alignment contains 4 columns with identical letters, one column with distinct letters and three columns with a space. We have chosen a scoring function that rewards matches and penalizes mismatches and spaces because we are interested in maximizing similarities between two sequences. Simple column scores work fine for aligning DNA sequences. For comparison of protein sequences, it is, however, appropriate to use a *score matrix* S, where $S(x, y)$ is the score for matching letter x against letter y. Spaces are handled separately because they tend to occur in bunches or *gaps* which is why the space penalty is often made dependent on gap length. We give now a precise definition of the similarity and alignment problem.

Problem 2.1 (Similarity problem) Given two sequences over an alphabet Σ and a score function w, find the maximum score of their alignment.

Problem 2.2 (Optimal alignment problem) Given two sequences over an alphabet Σ and a score function w, find their optimal alignment, i.e. an alignment that achieves the highest score.

Having clarified alignment scoring, we now turn our attention to computing optimal alignments.

2.2 Dynamic programming algorithm

There exists a exponentially large number of possible alignments of two sequences which makes a greedy algorithm that generates all possible alignments and picks the best one very inefficient. In the following, we review an efficient algorithm for computing optimal alignments that employs the *dynamic programming* technique. The general idea of dynamic programming is to save solutions for smaller instances of the problem in a table and use them later to solve the whole problem.

Let s and t be two sequences of length m and n, respectively, and let w be a score function. The dynamic programming algorithm builds up the optimal score of an alignment between s and t by computing the optimal scores of alignments between all prefixes of s and t. There are $n+1$ prefixes of s and $m+1$ prefixes of t including the empty strings which makes it appropriate to arrange the computations in an $(m+1) \times (n+1)$ matrix a, in which an element $a[i,j]$ contains the optimal score of an alignment between $s[1..i]$ and $t[1..j]$. We will call a the matrix of *prefix scores*.

The algorithm exploits the fact that an optimal alignment between $s[1..i]$ and $t[1..j]$, denoted $opt\,(s[1..i], t[1..j])$, can be obtained only in one of the following three ways:

- Construct $opt\,(s[1..i-1], t[1..j])$ and align a space with $t[j]$, or
- Construct $opt\,(s[1..i-1], t[1..j-1])$ and align $s[i]$ with $t[j]$, or
- Construct $opt\,(s[1..i], t[1..j-1])$ and align $s[i]$ with a space.

The three possibilities are exhaustive since an alignment column may not contain two spaces. This gives us immediately the recurrence relation for computing $a[i,j]$:

$$a[i,j] \;=\; \max \begin{cases} a[i-1,j] \;+\; w\,(s[i],-) & i>0, j\geq 0 \\ a[i-1,j-1] \;+\; w\,(s[i],t[j]) & i,j>0 \\ a[i,j-1] \;+\; w\,(-,t[j]) & i\geq 0, j>0, \end{cases} \tag{2.2}$$

where $w(x,y) = 1$ if $x = y$, $w(x,y) = -1$ if $x \neq y$ and $w(x,-) = w(-,x) = -2$. The first row and the first column of a are initialized with multiples of -2. This is because the only way to align $s[1..i]$ (or $t[1..j]$) with an empty string is to align each character of s (or t) with a space. The remaining elements are computed in the row or column fashion according to the above recurrence. On completion, $a[m,n]$ will contain the optimal score, $sim(s,t)$. In Figure 2.2 appears the complete pseudocode of the algorithm, which will be referred to as the *basic algorithm* in the rest of Chapter 2. It should be mentioned this algorithm, like all other algorithms in this thesis, depends on the *additive* property of the alignment score, that is, if the alignment is divided in two parts and each part is scored independently, then the sum of partial scores is equal to the score of the complete alignment.

An actual alignment can be reconstructed by *backtracking* in the matrix a. During the matrix computation, this procedure draws an arrow indicating how each (i,j) was obtained according to the recurrence in Equation (2.2). Once the complete matrix is filled, the procedure traces back through arrows from (m,n) to $(0,0)$. Each arrow represents an alignment column. A vertical arrow leaving (i,j) stands for a column in which $s[i]$ is matched with a space in t, a diagonal arrow corresponds to a column aligning $s[i]$ with $t[j]$, while a horizontal arrow means that $s[i]$ is matched with a space in t. Computing an optimal alignment is, thus, equivalent to finding a longest path in a *grid graph*. Figure 2.3 illustrates an example of a filled matrix with drawn optimal paths and corresponding alignments.

In Figure 2.4 is given a recursive algorithm that accepts sequences s and t and the matrix a, previously computed by basic algorithm, and returns an optimal alignment of s and t in the reverse order. In this code, instead of implementing arrows explicitly, a simple test is used

Algorithm *Similarity*

input: sequences s and t
output: similarity between s and t

$m := |s|$
$n := |t|$
for $i := 0$ **to** m **do**
  $a[i, 0] := i * w(s[i], -)$
for $j := 0$ **to** n **do**
  $a[0, j] := j * w(-, t[j])$
for $i := 1$ **to** m **do**
  for $j := 1$ **to** n **do**
    $a[i, j] := \max(a[i-1, j] + w(s[i], -)$
    $a[i-1, j-1] + w(s[i], t[j])$,
    $a[i, j-1] + w(-, t[j]))$
return $a[m, n]$

Figure 2.2. *Algorithm for computing optimal alignment score.*

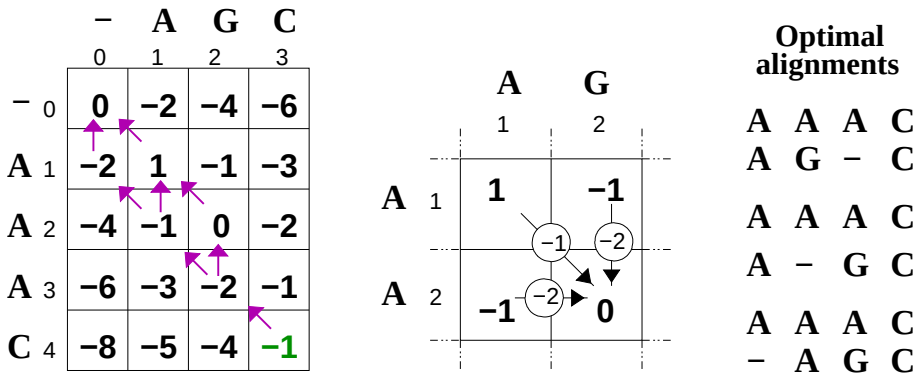

Figure 2.3. *Dynamic programming matrix a for sequences* AAAC *and* AGC*, optimal paths indicated with magenta arrows (left) and associated optimal alignments (right). The middle figure clarifies the evaluation of entry $a[2, 2]$: it is obtained from $a[1, 1]$, which is why we draw a magenta arrow going from (2,2) to (1,1) in the matrix a.*

to decide the next matrix element to visit. The call $Align(s, t, len, opt\text{-}align)$ returns only one optimal alignment in global variable *opt-align*.

Theorem 2.1 *An optimal alignment between two sequences s and t of length m and n, respectively, can be computed in time and space $\mathcal{O}(mn)$.*

Algorithm *Align*

input: sequences s, t and array a given by *Similarity*
output: optimal alignment in two-row array *opt-align*

$pos := 0$
if $i = 0$ `and` $j = 0$ **then**
  return $reverse(opt\text{-}align)$
else if $i > 0$ **and** $a[i,j] = a[i-1,j] \ + \ w(s[i], -)$ **then**
  $Align(i-1, j, pos)$
  $pos := pos + 1$
  $opt\text{-}align[1, pos] := s[i]$
  $opt\text{-}align[2, pos] := -$
else if $i > 0$ **and** $j > 0$ **and** $a[i,j] = a[i-1,j-1] \ + \ w(s[i], t[j])$ **then**
  $Align(i-1, j-1, pos)$
  $pos := pos + 1$
  $opt\text{-}align[1, pos] := s[i]$
  $opt\text{-}align[2, pos] := t[j]$
else// *has to be* $j > 0$ *and* $a[i,j] = a[i,j-1] \ + \ w(-, t[j])$
  $Align(i, j-1, pos)$
  $pos := pos + 1$
  $opt\text{-}align[1, pos] := -$
  $opt\text{-}align[2, pos] := t[j]$

Figure 2.4. *Algorithm for optimal alignment.*

2.3 Hirschberg's algorithm

In this section, we review Hirschberg's recursive algorithm [6] that finds an optimal alignment in $\mathcal{O}(m+n)$ space, at the cost of roughly doubling computation time. Linear-space algorithms are useful for aligning long sequences, in which case the space is often the limiting factor.

An immediate improvement in the computational time of the basic algorithm can be derived by noticing that each row in the dynamic programming matrix depends only on the preceding one. Figure 2.5 shows the modified version of the algorithm that computes in linear space the complete matrix leading to $sim(s,t)$. To recover an alignment associated with this highest score, Hirschberg employs a divide-and-conquer strategy, which consist of recursively breaking the problem into smaller independent subproblems, solving the subproblems directly and combining their solution to solve the whole problem.

The basic idea is to divide s into two parts and find an appropriate split of t such that the concatenation of the two subalignments, one aligning $s[1..i]$ and $t[1..j]$ and the other aligning the rest of s and t, produces an optimal alignment of the complete sequences s and t, i.e.

$$opt(s[1..i], t[1..j]) \ + \ opt(s[i+1..m], t[j+1..n]) = opt(s,t), \quad \text{for } j \in [1,n] \tag{2.3}$$

The alignment problem is, thus, reduced into two subproblems that involve aligning shorter sequences. The subproblems are in turn subdivided recursively until they are small enough to be aligned in the available memory using the standard procedure *Align*. Then the concatenation of optimal subalignments yields an optimal overall alignment.

Algorithm *BestScore*
input: sequences s and t
output: vector a

$m := |s|$
$n := |t|$
for $j := 0$ **to** n **do**
  $a[j] := j * w(-, t[j])$
for $i := 1$ **to** m **do**
  $old := a[0]$
  $a[0] := i * w(s[i], -)$
  for $j := 1$ **to** n **do**
    $temp := a[j]$
    $a[j] := \max(a[j] + w(s[i], -),$
      $old + w(s[i], t[j]),$
      $a[j-1] + w(-, t[j]))$
    $old := temp$
return $a[n]$

Figure 2.5. *Linear-space version of the algorithm Similarity.*

The main difficulty with this approach is to determine, for a fixed i, an optimal value of j that would allow the problem decomposition. This value j is determined as follows. The algorithm computes in linear space the scores for aligning $s[1..i]$ with an arbitrary prefix of t, and also the scores for aligning $s[i+1..m]$ with an arbitrary suffix of t. The former scores are saved in the matrix a of *prefix scores* and the latter in the matrix b of *suffix scores*. We saw earlier how to compute a. The matrix b is computed just like a but backward. We initialize the last row and column and compute backward until we reach $b[0,0]$. Summing the prefix and suffix scores gives the scores of j alignments in Equation 2.3 for $j \in [1,n]$. The best among these is, obviously, equal to $sim(s,t)$, so the choice of an optimal j is easy — pick j yielding the maximum total score.

We saw earlier that *BestScore* (Figure 2.5) computes the matrix a in linear space, a similar procedure is used for b. Figure 2.6. shows the complete pseudocode of the algorithm. In this code, i is set to be $\lfloor m/2 \rfloor$, as we want the subproblems to have roughly the same size. The call *BestScore*$(s[a..i], t[c..d],$ *pref-sim*$)$ returns in *pref-sim* the optimal scores for aligning $s[1..i]$ and $t[c..j]$ for $j \in [c-1, d]$. Similarly, *BestScoreRev*$(s[i+1..b], t[c..d],$ *suf-sim*$)$ returns in *suf-sim* the optimal scores for aligning $s[i+1..b]$ with $t[j+1..d]$ for $j \in [c-1, d]$. The call $DCA(s[1..m], t[1..n])$ returns an optimal alignment of s and t.

Complexity analysis

To analyze the storage requirement of the algorithm, we consider the space needed for the input sequences plus the space used in the recursion by *BestScore*. At the first recursion level, *BestScore* uses one buffer of size n to hold the optimal scores $a[i,0]$ through $a[i,j]$, and $a[i-1,j]$ through $a[i-1,m]$. This buffer may be reused in the recursion, hence the space complexity remains linear in m and n, that is, $\mathcal{O}(m+n)$.

Algorithm *DCA* (*D*ivide-and-*C*onquer *A*lignment)
input: sequences s and t, indices a, b, c, d
output: optimal alignment
description: *pref-align* and *suff-align* are two-row buffers holding prefix and suffix alignment. $(i, posmax)$ is optimal midpoint. *Align* returns optimal subalignments in buf.

```
// Base case: align s and t with standard procedure Align
if |s| * |t| < M then // M is available memory
    return Align(s[a..b], t[c..d], buf) // direct solution
else
    // General case: decompose, solve recursively and combine
    i := ⌊(a + b)/2⌋    // fix middle position in s

    // compute matrices of prefix and suffix scores
    BestScore (s[a..i], t[c..d], pref-scores)
    BestScoreRev (s[i + 1..b], t[c..d], suff-scores)

    // find optimal index posmax in t to allow decomposition
    posmax := c − 1
    vmax := pref-scores[c − 1] + suff-scores[c − 1]
    for j := c to d do
        score := pref-scores[j] + suff-scores[j]
        if score > vmax then
            vmax := score
            posmax := j

    // solve two subproblems recursively
    DCA (s[a..i], t[c..posmax], pref-align)
    DCA (s[i + 1..b], t[posmax + 1..d], suff-align)
    return Concat(pref-align, suff-align)
```

Figure 2.6. *Hirschberg's divide-and-conquer algorithm.*

The time complexity of the algorithm, denoted $T(m, n)$, satisfies the following recursion:

$$T(m, n) \leq mn \ + \ T(n/2, j) \ + \ T(n/2, m - j),$$

where mn is the total cost of the calls to *BestScore* and *BestScoreRev*, each taking $(m/2)n$ time, while $T(m/2, j)$ and $T(m/2, n-j)$ are the costs for the two recursive calls. In the following lemma, we use induction to prove that the time roughly doubles as a consequence of the recursion.

Lemma 2.1 *The time complexity of the BandDCA algorithm is as follows:* $T(m, n) \leq 2mn$.

Proof. For $m = 1$, no maximum computations occur, thus $T(1, n) \leq 2n$. For $m > 1$, we have

$$\begin{aligned} T(m,n) &\leq \frac{mn}{2} + \frac{mn}{2} + T(m/2, j) + T(m/2, n-j) \\ &\leq mn + nj + mn - nj \ = \ 2mn \end{aligned}$$

■

Theorem 2.2 *An optimal alignment between two sequences s and t of lengths m and n can be computed in time $\mathcal{O}(mn)$ and space $\mathcal{O}(m + n)$.*

2.4 *DBand* algorithm

This section examines a faster algorithm for finding high-scoring alignments in the case when sequences are fairly similar.

Let s and t be two very similar sequences of the common length n. From Section 2.2, we recall that alignments corresponds to paths in the dynamic programming matrix. An alignment of s and t that contains no spaces or, equivalently, in which all letters are aligned, has its corresponding path along the main diagonal. If this is not an optimal alignment, we may insert spaces into the sequences to form a better alignment. Since spaces are represented with horizontal (or vertical) moves in the matrix a, this will cause the associated path to wander away from the main diagonal. Hence, if sequences are similar, their alignment path is most likely contained within a narrow band around the main diagonal. The main idea is that we might be able compute the optimal score by filling only in a band portion of the matrix.

The algorithm *DBand* accepts two sequences s and t with $|s| = |t| = n$ and a parameter d as input and computes elements in the band of the horizontal width $2d + 1$. On completion, it returns in $a[n, n]$ the optimal score of an alignment confined to the band. Elements outside the d-strip is neither initialized nor used in maximum computations. The test for whether a position (i, j) lies inside the strip is completed with the following statement:

$$InsideStrip(i, j, d) \equiv (-d \leq j - i \leq d).$$

Each element $a[i, j]$ inside the strip is evaluated as with the standard dynamic programming except for the elements in the border, for which maximum computations may involve fewer than three terms. In the code, only positions $(i-1, j)$ and $(i, j-1)$ are tested with $InsideStrip$ because they may be outside the strip when (i, j) is in the border. As before, an actual alignment can be reconstructed using the backtracking algorithm *Align* in Figure 2.4.

Algorithm *DBand*
input: sequences s and t of equal length n, integer d
output: optimal alignment score in the d-strip

$n := |s|$
for $i := 0$ **to** d **do**
  $a[i, 0] := i * w\,(s[i], -)$
for $j := 0$ **to** d **do**
  $a[0, j] := j * w\,(-, t[j])$
for $i := 1$ **to** n **do**
  for $j := \max(0, i - d)$ **to** $\min(n, i + d)$ **do**
    // compute maximum among predecessors
    $a[i, j] := a[i - 1, j - 1] \; + \; w\,(s[i], t[j])$
    if $InsideStrip\ (i - 1, j, d)$ **then**
      $a[i, j] := a[i - 1, j] \; + \; w\,(s[i], -)$
    if $InsideStrip(i, j - 1, d)$ **then**
      $a[i, j] := a[i, j - 1] \; + \; w\,(-, t[j])$
return $a[n, n]$

Figure 2.7. *Algorithm for optimal score in d-strip.*

One last concern remains: Will the computed score in $a[n, n]$ be optimal even without the band constraints, that is, will we obtain the same score if we use the basic algorithm instead? This depends, of course, on the choice of parameter d and similarity degree between the sequences. The smaller value of d, the faster the algorithm will be but also the higher possibility of ending up with a suboptimal alignment, that is, an alignment whose score is close to the optimal score. There exist many methods for estimating d so an optimal alignment is likely contained in the d-strip, however none of them guarantees to return an optimal solution. We assume here that our d is well-suited for the input sequences and that the algorithm returns the optimal score.

It is straightforward to extend the algorithm to handle sequences of different lengths. Let two sequences s and t have lengths m and n, where $m \leq n$. The alignment definition implies that we must insert $n - m$ spaces into the shorter sequence s plus c additional space pairs into both sequences. Therefore, we need to modify the algorithm so it computes the band portion of the matrix between diagonals $-c$ and $c+n-m$, where $c = m-n+d$ and $d \geq n-m$. A *diagonal* k of a matrix consists of those elements (i, j) with $j - i = k$. Consequently, the d-strip criterion must be modified as follows:

$$InsideStrip(i, j, d) \equiv (-c \leq j - i \leq c + n - m).$$

The case when $m > n$ is handled analogously.

Complexity analysis

The computation time of *DBand* is proportional to the area of the d-strip, which is $dn+n-d^2 = \mathcal{O}(dn)$. The evaluation of each element $a[i, j]$ takes $\mathcal{O}(1)$ time, so the total computational time is, thus, $\mathcal{O}(dn)$. For $d \ll n$, this is a great speed-up of the basic algorithm.

The space requirement is also $\mathcal{O}(dn)$ since the backtracking procedure requires the complete d-strip in order to reconstruct the alignment itself. This space complexity may be reduced to linear by using Hirschberg's method, as we will show in the next section.

Theorem 2.3 *An optimal alignment in the d-strip between two sequences s and t can be computed in time and space $\mathcal{O}(dn)$, assuming that the sequences have the same length n.*

2.5 *BandDCA* algorithm: Aligning within a band in linear space

In this section, we derive a linear-space algorithm for finding alignments in a constant band by combining the divide-and-conquer- and banding technique. We begin by modifying *DBand* to compute $sim(s, t)$ in linear space. In Figure 2.8 appears a linear-space algorithm that accepts two sequences of different lengths and returns the best score. *BandScore* can be used as subprocedure in a divide-and-conquer algorithm that recovers an actual alignment.

BandDCA is a recursive algorithm that determines the middle pair of positions in an optimal alignment between two sequence and uses it to find the other positions in the alignment recursively. It does this by computing the d-strip of matrices a and b and selecting j from interval $[m/2 - d, m/2 + d]$ so it maximizes the total scores. This optimal j is the sufficient information to decompose the problem into two independent subproblems. The algorithm then makes two recursive calls subdividing the subproblems further until they are small enough to be solved directly. The optimal alignment is produced by appending these trivial alignments.

The pseudocode of the algorithm appears in Figure 2.9. The call $BandScore(s[a..i], t[c..d], ld, hd)$ returns the optimal prefix scores between $s[a..i]$ and $t[c..j]$ for $j \in [i + ld..i + hd]$. Analogously, the call $BandScoreRev(s[i+1..b], t[c..d], ld, hd)$ returns the optimal suffix scores between $s[i+1..b]$ and $t[j+1..d]$ for $j \in [i+ld..i+hd]$. The call $BandDCA(s[1..m], t[1..n], ld, hd)$ returns an optimal alignment of s and t confined to the band between diagonals ld and hd.

Algorithm *BandScore*
input: sequences s and t, integers ld, hd
output: vector a

$m := |s|$
$n := |t|$
for $j :=$ $\max(0, ld)$ **to** $\min(n - m, lh)$ **do**
  $a[j] := j * w\,(s[i], -)$
for $i := 0$ **to** m **do**
  $old := a[0]$
  $a[0] := i * w\,(s[i], -)$
  for $j :=$ $\max(0, i + ld)$ **to** $\min(n - m, i + lh)$ **do**
    $temp := a[j]$
    $a[j] := old + w\,(s[i], t[j])$
    if $InsideStrip(i - 1, j, ld, hd)$ **then**
      $a[j] :=$ $\max(a[j], a[j] + w\,(s[i], -))$
    if $InsideStrip(i, j - 1, ld, hd)$ **then**
      $a[j] :=$ $\max(a[j], a[j - 1] + w\,(-, t[j]))$
    $old := temp$
return $a[n]$

Figure 2.8. *Linear-space algorithm for computing the optimal score.*

Complexity analysis

We assume, for simplicity, that sequences s and t have the same size n and that $-ld = hd = d$. The time complexity of *BandDCA*, denoted $T(n, d)$, can be estimated with the following recursion:

$$T(n, d) \;\leq\; c\, dn + 2T(n/2, d),$$

where c is a constant, $\mathcal{O}(dn)$ is the time spent in finding the alignment midpoint and $2T(n/2, d)$ is the cost of the two recursive calls to itself. We observe that when the band is reduced to the main diagonal, i.e. when d is 1, the time complexity expression becomes $T(n) \leq c\, n + 2T(n/2)$. After unfolding, the recurrence has $\log_2 n$ terms:

$$\sum_{i=1}^{\log n} 2^i \frac{n}{2^i} = n \sum_{i=1}^{\log n} 1^i = n \log_2 n.$$

In the worst case, the algorithm recomputes all elements in the main diagonal at each recursive level, which causes the total slow-down in the algorithm by the factor $\log_2 n$. This claim is proved strictly in the following lemma.

Lemma 2.2 *The time complexity of the BandDCA algorithm is as follows:* $T(n, d) \leq c\, dn \log_2 n$.

Proof. The proof is developed by induction on n.

For $n = 2$, $T(2, d) \leq 2d$ is obviously true for $d \geq 1$.

Algorithm *BandDCA* (*Band*ed *D*ivide-and-*C*onquer *A*lignment)
input: sequences s, t, indices a, b, c, d, diagonals ld, hd
output: optimal alignment
description: *pref-align* and *suff-align* are two-row buffers holding prefix and suffix alignment. $(i, posmax)$ is optimal midpoint. *Align* returns part-alignments in *opt-align*.

```
// Base case: align s and t with standard procedure Align
if |s| * |t| < M then // M is available memory
    return Align(s[a..b], t[c..d], ld, hd, opt-align)
else
    // General case: decompose, solve recursively and combine
    i := ⌊(a + b)/2⌋    // fix middle position in s

    // compute matrices of prefix and suffix scores
    BandScore (s[a..i], t[c..d], ld, hd, pref-scores)
    BandScoreRev (s[i + 1..b], t[c..d], ld, hd, suff-scores)

    // find optimal index posmax in t to allow decomposition
    posmax := max(i + ld, c − 1)
    vmax := pref-scores[posmax] + suff-scores[posmax]
    for j := max(i + ld, c) to min(i + hd, d) do
        score := pref-scores[j] + suff-scores[j]
        if score > vmax then
            vmax := score
            posmax := j

    // solve two subproblems recursively
    DCA (s[a..i], t[c..posmax], ld, hd, pref-align)
    DCA (s[i + 1..b], t[posmax + 1..d], ld, hd, suff-align)
    // concatenate two sub-alignments
    return Concat(pref-align, suff-align)
```

Figure 2.9. *Linear-space algorithm for optimal alignment in d-strip.*

For $n > 2$, we have the following:

$$\begin{aligned} T(n,d) &\leq c\,dn + 2T(n/2,d) \\ &\leq c\,dn + c\,2d(n/2)\log_2(n/2) \\ &= c\,dn + c\,dn\log_2 n - c\,dn \\ &= c\,dn\log_2 n. \end{aligned}$$

■

To analyze the space complexity, we consider lengths of the sequences and two buffers of size $ld + hd + 1$ to hold the optimal scores computed with *BandScore* and *BandScoreRev*. These buffers are reused in the recursion so the space complexity of the algorithm is $\mathcal{O}(m + n)$.

Theorem 2.4 *A d-optimal alignment of two sequences s and t of length n can be computed in time $\mathcal{O}(dn\log_2 n)$ and space $\mathcal{O}(m + n)$, in worst case.*

2.6 *FastDCA*: A faster, space-efficient aligning algorithm

A drawback with the previous algorithm is the $\log_2 n$ factor appearing in the expression for the worst-time complexity. In this section, we show how to speed up the algorithm, at the cost of using a few extra linear buffers. The space requirement remains, however, linear in m and n.

The main idea is to divide s into p substrings of equal size and find a suitable split of t into p substrings such that a concatenation of p optimal subalignments between the corresponding substrings in s and t produces the total optimal alignment. The alignment problem is this way decomposed directly in p independent subproblems which, in turn, are subdivided recursively until they are small enough to be solved with the standard procedure. In this decomposition of problems, the subproblem k involves aligning $s[k\frac{m}{p}+1..(k+1)\frac{m}{p}]$ and $t[j_k+1..j_{k+1}]$, for $0 \leq k < p$. This dividing scheme, which is often used in parallel algorithms for sequence comparisons [5], can be seen as a generalization of Hirschberg's approach. For $p = 2$, the original Hirschberg's method is obtained.

The difficulty with this approach is in finding, for p fixed positions $\frac{m}{p}, \frac{2m}{p}, \ldots, m$ in s, a p-tuple of cut positions in t that would allow the problem decomposition. To do this, we make use of a matrix c of *total scores*, in which $c[i,j]$ contains the highest score of an alignment that *cuts* at (i,j). An alignment contains $cut(i,j)$ when the alignment can be divided into two subalignments, one aligning prefixes $s[1..i]$ and $t[1..j]$, and the other aligning suffixes $s[i+1..m]$ and $t[j+1..n]$. The matrix c is computed as the sum of the matrices a and b with their contents, where a and b are computed as before:

$$\begin{aligned} a[i,j] &= sim\,(s[1..i], t[1..j]) \\ b[i,j] &= sim\,(s[i+1..m], t[j+1..n]). \end{aligned}$$

In fact, we only compute p dividing rows in c corresponding to the total alignment scores for each possible p-tuple of cut positions in t and select the one that maximizes the score in each of p rows of c. In addition to using a different dividing scheme, we also must impose band constraints on the algorithm in order to restrict the search space for optimal alignments to the band between diagonals ld and hd. The procedure *BandScore* does exactly this. An outline of the algorithm for sequence s and t follows:

1. Define p dividing rows of the matrix c to be $\frac{m}{p}, \frac{2m}{p}, \ldots, m$, where $c = a + b$.
2. Using linear space, compute and save p dividing rows of c.
3. Find the maximum-valued element in each of p dividing rows in order to reveal optimal cut positions $j_1, j_2, \ldots, j_p$ in t. For dividing row i, this involves computing any $c[i,j]$ and selecting the one with the highest value:
$$c[i,j] \;=\; \max\{\, c[i,l] \;\; :\; l \in [i-ld, i+hd] \,\}\,.$$
4. Use optimal cut positions to subdivide the problem into p subproblems. For each subproblem, if the product of the sequences lengths is smaller than a threshold M, then we construct an optimal alignment directly. Otherwise, we make p recursive calls to further subdivide the subproblems.
5. Concatenate produced optimal subalignments to obtain the total optimal alignment.

The complete pseudocode of this recursive algorithm is given in Figure 2.10.

Complexity analysis

To analyze the running time, we assume, for simplicity, that s and t have the same size n and the subproblems are evenly subdivided at each recursion level. Also we set $d = -ld = hd$. The

Algorithm *FastDCA* (*Fast* Divide-and-*C*onquer *A*lignment)
input: sequences s, t, indices a, b, c, d, diagonals ld, hd, constant p
output: optimal alignment in *align*

```
align = ∅    // two-row buffer to hold alignment
bandwidth := ld + hd + 1
// Base case: align s and t with standard procedure Align
if |s| * |t| < M then // M is the available memory
    return BandAlign (s[a..b], t[c..d], ld, hd, buf)
else
    // General case: decompose, solve recursively and combine
    // compute band portions of matrices of prefix and
    // suffix scores, saving only p dividing rows
    PrefixScores (s[a..b], t[c..d], p, ld, hd, pref-scores)
    SuffixScores (s[a..b], t[c..d], p, ld, hd, suff-scores)
    // find optimal index jmax for each dividing row
    for k := 0 to p − 1 do
        jmax := c − 1
        maxscore := pref-scores[k, c − 1] + suff-scores[k, c − 1]
        i := ⌊(k + 1) * (a + b)/p⌋    // dividing row i
        for j := 1 to bandwidth do
            score := pref-scores[j] + suff-scores[j]
            if score > maxscore then
                maxscore := score
                jmax := j
        // solve subproblem k recursively
        FastDCA (s[a..i], t[c..jmax], p, ld, hd, subalign-k)
        align = Concat (align, subalign-k)    // acummulate alignment
        a := i + 1
        c := jmax + 1
    return align
```

Figure 2.10. *Algorithm for optimal alignment in d-strip.*

time complexity of the algorithm, denoted $T(n, d)$, can then be estimated with the following recurrence:

$$T(n, d) \;=\; c\, dn + p\, T(n/p, d),$$

for some constants c and p. In this expression, the term $c\ dn$ is the time spent for computing the matrix of total scores and $p\ T(n/p, d)$ is the cost of p recursive calls with sequences of length n/p. We claim that $T(n, d) \leq c\ dn \log_p n$. The proof is developed by the induction on n. For $n = 1$, the statement is obviously true, as no maximum computations occurs. For $n > 1$, we have the following equations:

$$\begin{aligned} T(n,d) &\leq c\,dn + p\,T(n/p,d) \\ &\leq c\,dn + c\,p\,d(n/p)\log_p(n/p) \\ &= c\,dn + c\,dn\log_p n - c\,dn \\ &= c\,dn\log_p n \end{aligned}$$

for some constant c. By setting p to $n^{1/c}$, we get rid of the log factor, leaving $T(n,d) \leq c\,dn$. This is asymptotically the same time complexity as that of the *DBand* algorithm.

The algorithm uses $\mathcal{O}(p\,d)$ space to find the first p-tuple of optimal cut positions. This linear buffer may be reused in the recursion, so the total space complexity, considering also the size of the input sequences, remains linear in m and n, given that p is set to $n^{1/c}$. Our findings are summed up in the following theorem.

Theorem 2.5 *An optimal alignment of sequences s and t within a specified band of width $2d+1$ can be computed in time $\mathcal{O}(dn)$ and space $\mathcal{O}(p\,d)$, where n is the common length of the sequences and $p = n^{1/c}$ for some constant c.*

Chapter 3

Multiple sequence alignment

The objective of this chapter is to generalize the alignment algorithms from Chapter 2 to work with multiple sequences. In Chapter 3, we study algorithms for simultaneous aligning of more than two sequences. We start out by giving the formal definition of the multiple alignment problem and proceed with a review of the dynamic programming algorithm in k dimensions. A generalization of Hirschberg's space-saving algorithm for optimal alignments of k sequences is discussed next. Furthermore, we extend the two-sequence algorithm for aligning within a specified band to k sequences. Lastly, we combine the banding approach with Hirschberg's divide-and-conquer technique to derive an algorithm for multi-alignments within a band that uses less space than the standard dynamic programming.

3.1 Motivation

In sequence analysis one frequently compares several protein sequences that have similar function. Finding their optimal multi-alignment might be useful in the following applications:

- Discovering similarities in conserved regions;
- Grouping proteins into families;
- Reconstructing evolutionary trees for various organisms.

3.2 *SP*-alignment

To formalize the concept of the optimal multi-alignment and discuss its scoring, we need to introduce some terminology.

Definition 3.1 (Multi-alignment) Let $s = \{s_1, s_2, \ldots, s_k\}$ be a set of sequences over an alphabet Σ. An *alignment* between sequences in s is a set of extended sequences $s' = \{s'_1, \ldots, s'_k\}$ over the alphabet $\Sigma^+ = \Sigma \cup \{-\}$ with the following three properties, for all $i, j : 1 \leq i \leq k$, $1 \leq j \leq n$:

1. $|s'_i| = n$, where n is the length of the alignment s'.
2. After removal of all spaces from s'_i, we have $s'_i = s_i$.
3. At least one $s'_i[j]$ is not a space.

An example of a multi-alignment is given in Figure 3.1. There exist a large number of possible multi-alignments for a given set of k sequences. Our goal is to find an alignment that maximizes

s'_1 :	A	C	T	–	G
s'_2 :	A	–	T	C	G
s'_3 :	G	C	C	–	A
s'_4 :	G	T	T	–	–

Figure 3.1. *A multi-alignment of four short DNA sequences*

an objective function. A simple approach to scoring multi-alignments is to evaluate scores column by column. To score an alignment columns we use an *additive* function known as the *sum-of-pairs* (SP) score, which is based on the pairwise score function defined in Section 2.1.

Definition 3.2 (*SP*-score of alignment) Let α be an alignment of k sequences. Also, let w be a score function for aligning a pair of characters and define $w(-,-) = 0$. The score of the column $column_j$ in α is computed as:

$$SP\text{-score}(column_j) = \sum_{x,y \in column_j} w(x,y) \tag{3.1}$$

The *score* of the alignment α is the sum of its column scores:

$$SP\text{-score}(\alpha) = \sum_{j=1}^{|s'_1|} SP\text{-score}(column_j), \tag{3.2}$$

Alternatively, we can score an multi-alignment based on pairwise alignments that can be extracted from it.

Definition 3.3 (Induced pairwise alignment) Let α be an alignment of k sequences. A pairwise alignment obtained from α by selecting any two sequences and deleting columns consisting of two spaces is called the *induced* pairwise alignment or the *projection* of α on the two selected sequences.

The following lemma connects the SP-score of a multi-alignment to its pairwise scores. It states that the total score of an alignment is the sum of the scores of all its projections.

Lemma 3.1 *If $w(-,-) = 0$, then for any alignment α of $s_1, s_2, \ldots, s_k$, we have*

$$SP\text{-}score(\alpha) = \sum_{1 \le i < j \le k} score(\alpha_{i,j}) \tag{3.3}$$

where $\alpha_{i,j}$ is the pairwise alignment induced by α on s_i and s_j.

Definition 3.4 (Multi-similarity) The *multi-similarity* between k sequences is the maximum score of any multi-alignment according to a score function w, i.e.

$$sim(s_1, s_2, \ldots, s_k) = \max \{SP\text{-score}(\alpha) : \ \alpha = (s'_1, \ldots, s'_k),\ \alpha \in \mathcal{A}(s_1, s_2, \ldots, s_k)\},$$

where $\mathcal{A}(s_1, s_2, \ldots, s_k)$ is the set of all possible multi-alignments of $s_1, s_2, \ldots, s_k$.

Definition 3.5 (Optimal multi-alignment) An *optimal* multi-alignment is a multi-alignment that has the maximum SP-score.

Notice that Equations 3.2 and 3.3 do the same thing, only in different orders. In each case, we are computing the score $w(s'_i[k], s'_j[k])$ for each column k and for each pair (i, j). We are now ready to give a precise formulation of the multi-alignment problem.

Problem 3.1 (Multi-similarity problem) Given k sequences and a score function w, find the maximum score of their multi-alignment.

Problem 3.2 (Optimal multi-alignment problem) Given k sequences and a score function w, find their optimal multi-alignment, i.e. an alignment that achieves the maximum score.

3.3 Dynamic programming in k dimensions

The dynamic programming technique can easily be extended to more than two sequences. Let $s_1, s_2, \ldots, s_k$ be k sequences we wish to align and assume, for simplicity, that they have the same size n. Instead of a two-dimensional matrix, we use a k-dimensional matrix A to hold the optimal alignment scores between arbitrary prefixes of k sequences. Specifically, $A[i_1, i_2, \ldots, i_k]$ contains the optimal score for $s_1[1..i_1], s_2[1..i_2] \ldots, s_k[1..i_k]$. Using boldface letters to denote k-tuples, the two-sequence recurrence relation in Equation 2.2 for k sequences becomes:

$$A[\mathbf{i}] = \max\left\{A[\mathbf{i} - \mathbf{b}] + SP\text{-score}(Column(\mathbf{s}, \mathbf{i}, \mathbf{b})) : \ \mathbf{b} \in \{0,1\}^k \setminus \{0, \ldots, 0\}\right\}, \tag{3.4}$$

where $Column(\mathbf{s}, \mathbf{i}, \mathbf{b}) = (c_j)_{1 \le j \le k}$ with

$$c_j = \begin{cases} s_j[i_j] & \text{if } b_j = 1 \\ - & \text{if } b_j = 0. \end{cases}$$

Term SP-score$(Column(\mathbf{s}, \mathbf{i}, \mathbf{b}))$, which is the score of current column in the alignment, is computed according to Equation 3.1. After initializing A with the statement $A[\mathbf{0}] = 0$, the algorithm computes the remaining cells following the order of the above recurrence. The operator max is taken over at most $2^k - 1$ values, as this is the number of all possible configurations for an alignment column. A cell $A[\mathbf{i}]$ is a *border cell* if at least one of its indices i_j is 0. Border cells, which depend on less than $2^k - 1$ other cells, are calculated with the same recurrence except that b_j must be zero when i_j is zero. When k is fixed, k nested **for** loops can be used to fill in A.

The alignment itself can be recovered in $\mathcal{O}\left(n^k\right)$ space by using a backtracking procedure similar to the one described for the pairwise case. The backtracking algorithm uses $\mathcal{O}\left(n^k\right)$ space since the entire matrix is required to be present in memory. Alternatively, the generalized Hirschberg's algorithm can be used to deliver the alignment in $\mathcal{O}\left(n^{k-1}\right)$ space, which will be discussed in the next section.

Complexity analysis

The SP method requires $\mathcal{O}(k^2)$ time for computing column scores as there are $k(k-1)/2$ pairwise scores to add up for each column. The computation of each cell depends on $2^k - 1$ other neighbors. Since there are $(n+1)^k$ cells in total to compute, the time complexity can be estimated to $\mathcal{O}\left(k^2 2^k n^k\right)$. This confirms the well-known fact [11]: the multi-alignment problem is NP-hard under the SP-score.

Theorem 3.1 *An optimal alignment of k sequences, under the SP-score, can be computed in time $\mathcal{O}\left(k^2 2^k n^k\right)$ and space $\mathcal{O}\left(n^k\right)$, given that all sequences have the same size n.*

3.4 Generalized Hirschberg's algorithm

We describe a generalization of Hirschberg's algorithm to k dimensions that reduces the space complexity of the dynamic programming with one order of magnitude, at the expense of roughly doubling the computation time.

The basic divide-and-conquer idea for multi-alignments goes as follows. Given sequences $s_1, s_2, \ldots, s_k$, cut s_1 in half and determine a $(k-1)$-tuple of optimal cut positions in remaining sequences to allow the problem decomposition. The problem of finding an optimal alignment of $s_1, s_2, \ldots, s_k$ is this way reduced to the two subproblems: the *prefix problem* of optimally aligning $s_1[1..n_1/2], s_2[1..i_2], \ldots, s_k[1..i_k]$ and the *suffix problem* of optimally aligning $s_1[n_1/2+1..n_1], s_2[i_2+1..n_2], \ldots, s_k[i_k+1..n_k]$. Each subproblem is subdivided recursively until the sequences are short enough to be aligned within the available memory using dynamic programming. The concatenation of the optimal subalignments produces an overall optimal alignment of k sequences.

It is easy to see that optimal cut positions always exist — we could select them explicitly if we were given an optimal alignment. Finding optimal i_j's without a pre-given optimal alignment is a more difficult task; however the strategy is the same as in the two-sequence case:

- Compute the optimal scores of aligning $s_1[1..n_1/2]$ with prefixes $s_2[1..n_2] \ldots, s_k[1..n_k]$;
- Compute the optimal scores of aligning $s_1[n_1/2+1..n_1]$ with suffixes $s_2[1..n_2] \ldots, s_k[1..n_k]$;
- Choose the best among the total scores.

To save the above scores the algorithm uses two matrices A and B of prefix scores and suffix scores, respectively. It computes the first halves of the matrices in a row fashion, saving only the last rows, that is, two $(k-1)$-dimensional hyperplanes given by $\lfloor n_1/2 \rfloor$ and $\lfloor n_1/2+1 \rfloor$ as the first coordinates. As in the pairwise case, adding A and B with their contents results in the the matrix C of *total scores*:

$$\begin{aligned} C[n_1/2, i_2 \ldots, i_k] &= sim(s_1[1..n_1/2], s_2[1..i_2], \ldots, s_k[1..i_k]) \quad + \\ & \quad sim(s_1[n_1/2+1..n_1], s_2[i_2+1..n_2], \ldots, s_k[i_k+1..n_k]), \end{aligned}$$

for each input sequence s_j with $|s_j| = n_i,\ 0 \le i_j \le n_i,\ 1 \le j \le k$. The algorithm computes the midrow in C, which is the optimal scores of all alignments for each possible choice of i_j's. To retrieve optimal i_j's, it then chooses the best among the total scores, i.e.

$$C[n_1/2, i_2 \ldots, i_k] = \max \{ C[n_1/2, l_2, \ldots, l_k] \ : \ 0 \le l_j \le n_j,\ 2 \le j \le k \},$$

since this score is equal to the multi-similarity $sim(s_1, s_2, \ldots, s_k)$.

The complete pseudocode of the algorithm appears in Figure 3.2. In this code, the matrix A is computed in space $\mathcal{O}\left(n^{k-1}\right)$ using *BestScore*. This procedure is implemented analogously to its pairwise counterpart. A similar procedure *BestScoreRev* is used for B. The stopping criterion for recursion is given by a threshold M, the available memory. Parameter M is set $\prod_{i=2}^{k} (|s_i| + 1)$, which is the minimal amount of memory at the first level of recursion required for searching $(k-1)$-dimensional space for the optimal i_j's. Without this amount of memory available the algorithm cannot run, in which case a further space reduction must be considered — a possible topic for future work.

Complexity analysis

The lower bound for the space complexity is proportional to the space for one $(k-1)$-dimensional row, which is $\mathcal{O}\left(n^{k-1}\right)$. This storage can be reused in the recursion so the total space complexity

Algorithm *DCMA*

input: set of sequences $s = \{s_1, s_2, \ldots, s_k\}$, indices $c_1, \ldots c_k,\ d_1, \ldots d_k$
output: optimal alignment of s

$n_j := |s_j|, \quad j \in \{1, \ldots, k\}$
$n := \{n_1, n_2, \ldots, n_k\}$

// Base case: construct short alignments with dynamic programming
if $n_1 * n_2 * \ldots * n_k < M$ **then** *// M is the available memory*
  return *Malign*(s)
else
  // General case: divide and solve recursively
  $i_1 := \lfloor n_1/2 \rfloor$
  BestScore $(\{s_1[1..i_1], s_2[1..i_2], \ldots, s_k[1..n_k]\}, \textit{pref-scores})$
  BestScoreRev $(\{s_1[i_1 + 1..n_1], s_2[i_2 + 1..n_2], \ldots, s_k[1..n_k]\}, \textit{suff-scores})$

  // To retrieve optimal cuts, find maximum among total scores
  $\{i_2, i_3, \ldots, i_k\} :=$ *BestCut* $(n, \textit{pref-scores}, \textit{suff-scores})$

  // Two recursive calls solves two subproblems
  DCMA $(\{s_1[c_1..i_1], s_2[c_2..i_2], \ldots, s_k[c_k..i_k]\}, \textit{pref-align})$
  DCMA $(\{s_1[i_1 + 1..d_1], s_1[i_2 + 1..d_2] \ldots, s_k[i_k + 1..d_k]\}, \textit{suff-align})$
  return *Concat* $(\textit{pref-align}, \textit{suff-align})$

BestCut (m, A, B)
  $\textit{maxscore} := -\infty$
  $\textit{bestcuts} := \emptyset$
  $i_1 := \lfloor m_1/2 \rfloor$
  for $i_2 := 0$ **to** m_2 **do**
    for $i_3 := 0$ **to** m_3 **do**
      $\ddots$
      for $i_k := 0$ **to** m_k **do**
        $C[i_1, i_2, \ldots, i_k] := A[i_1, i_2, \ldots, i_k] + B[i_1, i_2, \ldots, i_k]$
        if $C[i_1, i_2, \ldots, i_k] > \textit{maxscore}$ **then**
          $\textit{maxscore} := C[i_1, i_2, \ldots, i_k]$
          $\textit{bestcuts} := \{i_2, i_3, \ldots, i_k\}$
  return *bestcuts*

Figure 3.2. *Hirschberg's algorithm in k dimensions.*

remains the same. We conclude that the reduction in the space complexity from $\mathcal{O}\left(n^k\right)$ to $\mathcal{O}\left(n^{k-1}\right)$ is not so impressive as that from quadratic to linear space in the pairwise case.

In decomposition of problems, the size of the subproblems is roughly half that of the main problem, so the total slow-down in the algorithm caused by the recursion. Next we analyze the time complexity in a more formal way. To keep the analysis simple, we assume that all

sequences have the same length $n = \ell \cdot 2^t$ and optimal cut positions are the middle positions in the sequences. The time complexity of the algorithm, $T(n)$, can be estimated with the following recurrence relation:

$$T(n) \;\leq\; \begin{cases} c_1 k^2 2^k n^k & \text{if } n \leq \ell \\ c_1 k^2 2^k n^k \;+\; c_2 n^{k-1} \;+\; 2T\left(n/2\right) & \text{otherwise,} \end{cases} \tag{3.5}$$

for some constants c_1 and c_2. Here, $2T\left(n/2\right)$ is the amount of work for solving the two subproblems of size $n/2$ each and the remaining part is the time required for dividing the problem. Precisely, $c_1 k^2 2^k n^k$ is the time for computing matrices A and B and $c_2 n^{k-1}$ is the number of comparison required for searching for maximum element among $(n+1)^{k-1}$ elements in the midrow of C. Unfolding the recurrence, we obtain a summation for T with $t = \log n - \log \ell$ terms plus the amount of work spent at the last recursion level for aligning n/ℓ sequences of length ℓ each:

$$\begin{aligned} T(n) \;&\leq\; \sum_{i=0}^{t-1} 2^i \left[c_1 k^2 2^k \left(\frac{n}{2^i}\right)^k \;+\; c_2 \left(\frac{n}{2^i}\right)^{k-1} \right] \;+\; \frac{n}{\ell}\left(c_1 k^2 2^k \ell^k\right) \\ &=\; c_1 k^2 2^k n^k \sum_{i=0}^{t-1} \left(\frac{1}{2^{k-1}}\right)^i \;+\; c_2\, n^{k-1} \sum_{i=0}^{t-1} \left(\frac{1}{2^{k-2}}\right)^i \;+\; c_1\, k^2 2^k n \ell^{k-1} \\ &=\; c_1 k^2 2^k n^k \,\frac{1-\left(\frac{1}{2^{k-1}}\right)^t}{1-\frac{1}{2^{k-1}}} \;+\; c_2\, n^{k-1}\, \frac{1-\left(\frac{1}{2^{k-2}}\right)^t}{1-\frac{1}{2^{k-2}}} \;+\; c_1\, k^2 2^k n \ell^{k-1} \qquad (3.6) \\ &=\; \mathcal{O}\left(k^2 2^k n^k \;+\; n^{k-1} \;+\; k^2 2^k n \ell^{k-1}\right), \quad \text{for } t > 0 \text{ and } k > 1. \qquad (3.7) \end{aligned}$$

In the extreme cases, we obtain the following results:

1. For $t = 1$ or, equivalently, $\ell = n$, the first two terms in Equation (3.6) disappear, while the third term becomes $\mathcal{O}\left(k^2 2^k n^k\right)$, the time bound of dynamic programming running on original sequences.
2. For $t = \log_2 n$ (or $\ell = 1$), the first two terms remain the same as in Equation (3.7), while the third term reduces to $\mathcal{O}(k^2 2^k n)$. Hence, T is $\mathcal{O}\left(k^2 2^k n^k \;+\; n^{k-1} \;+\; k^2 2^k n\right)$, which is, again, asymptotically the same time as for dynamic programming.

An inductive proof for time bound T is given in Appendix A.

Theorem 3.2 *An optimal alignment of k sequences, under the SP measure, can be computed in time $\mathcal{O}\left(k^2 2^k\, n^k \;+\; n^{k-1}\right)$ and space $\mathcal{O}\left(n^{k-1}\right)$, given that all sequences have the same size n.*

3.5 *DBand* in k dimensions

We have seen how to reduce the space complexity of the dynamic programming algorithm from $\mathcal{O}\left(n^k\right)$ to $\mathcal{O}\left(n^{k-1}\right)$; our goal here is to speed up the computations of alignment scores. The time bound can be improved by excluding those cells from computations hat are not likely to be part of an optimal alignment. This is a generalization of *DBand* algorithm from Section 2.4.

Suppose we want to align k fairly similar sequence of length n. The algorithm exploits the fact that optimal alignments of similar sequences have their paths near the main diagonal in the matrix. It establishes a band or a *polyhedron* of radius d around the main diagonal and look for optimal alignments only inside this area. A cell $\mathbf{i} = (i_1, \ldots, i_k)$ belongs to the main diagonal

if $i_l = i_{l+1}$ for $1 \leq l < k$. Any cell $A[\mathbf{i}]$ is computed according to the standard recurrence in Equation (3.4) except for border cells, in which case neighboring cells that fall outside the d-strip are not consulted in maximum computations. To test whether a position $\mathbf{i}$ falls inside the d-strip, we use the following statement:

$$InsideStrip\ (\mathbf{i}, d)\ \equiv\ (-d \leq i_1 - i_j \leq d\ :\ \ 1 < j \leq k).$$

The complete *DBand* algorithm in k dimensions is given in Figure 3.3. It accepts k sequences and a parameter d and returns the optimal score of their multi-alignment confined to the d-strip. The procedure that performs fill-in the d-strip of A is implemented as k nested **for** loops. The algorithm can be extended to compute the optimal score in the d-strip around a given heuristic alignment rather than around the main diagonal. We can use the algorithm to improve the score of a given heuristic alignment.

Algorithm *DBand* in k dimensions
input: sequences $s = \{s_1, s_2, \ldots, s_k\}$ of length n, integer d
output: optimal score of an alignment confined to d-strip

$n := |s_1|$
for $i_1 := 1$ **to** n **do**
  for $i_2 :=$ max$(0, i_1 - d)$ **to** min$(n, i_1 + d)$ **do**
    $\ddots$
    for $i_k :=$ max$(0, i_1 - d)$ **to** min$(n_1, i_1 + d)$ **do**
      $a[\mathbf{i}] :=$ max$\{a[\mathbf{i} - \mathbf{1}] + SP\text{-}score\big(Column(\mathbf{s}, \mathbf{i}, \mathbf{b})\big)\}$
      if $InsideStrip\ (\mathbf{i} - \mathbf{b}, d)$ **then**
        $a[\mathbf{i}] :=$ max$\{a[\mathbf{i} - \mathbf{b}] + SP\text{-}score\big(Column(\mathbf{s}, \mathbf{i}, \mathbf{b})\big)\}$
return $a[n_1, n_2, \ldots, n_k]$

Figure 3.3. *DBand algorithm in k dimensions.*

As in the two-sequence case, the main concern here is how to choose a parameter d so the computed score is optimal even without the band constraints. A value of d is best found by computing a lower bound L for the alignment score and then setting d to $S_{max} - L$, where S_{max} is the maximum possible score of aligning k identical sequences. A quick estimate for L can be obtained as follows:

$$L\ =\ \sum_{1 \leq x < y \leq k} sim(s_x, s_y).$$

Complexity analysis

The lower bound for the computational time is proportional to the number of cells in the d-strip, that is, $\mathcal{O}(nd^{k+1})$. Each cell can be computed in time $\mathcal{O}(2^k k^2)$, given that the column scores take $\mathcal{O}(2^k)$ time. The total time complexity of *DBand* is thus $\mathcal{O}\left(k^2 2^k\ nd^{k-1}\right)$, which is a big advantage over dynamic programming when $d \ll n$, since the d-strip is in that case only a small subset of the set containing all the matrix cells. For similar sequences, spaces will be evenly distributed in the sequences so *DBand* is likely to find an optimal alignment score. The space complexity is $\mathcal{O}\left(nd^{k-1}\right)$ as the algorithm saves the entire d-strip in the memory.

Theorem 3.3 *An optimal alignment of k sequences in the d-strip, under the SP score, can be computed in time* $\mathcal{O}\left(k^2 2^k\ nd^{k-1}\right)$ *and space* $\mathcal{O}\left(nd^{k-1}\right)$*, given that all sequences have the size n.*

3.6 *BandDCMA* algorithm: A faster multi-aligning in less space

Hirschberg divide-and-conquer algorithm can be extended to handle the band constraints. An outline of the algorithm *BandDCMA* for optimal multi-alignments within a specified band follows.

1. Set $i_1 = \lfloor n_1/2 \rfloor$.
2. In row fashion, compute the first halves of the matrices A and B of prefix and suffix scores, respectively, subject to constraint that $i_j \in [i_1 - d, i_1 + d]$ for $2 \leq j \leq k$, saving only two midrows:

$$\begin{aligned} A[i_1, i_2, \ldots, i_k] &= sim\left(s_1[1..i_1], s_2[1..i_2], \ldots, s_k[1..i_k]\right) \\ B[i_1, i_2, \ldots, i_k] &= sim\left(s_1[i_1 + 1..n_1], s_2[i_2 + 1..n_2], \ldots, s_k[i_k + 1..n_k]\right) \end{aligned}$$

3. Choose i_j's $(2 \leq j \leq k)$ maximizing the total scores, i.e.

$$(i_1, i_2, \ldots, i_k) = \underset{i_j \in [i_1 - d, i_1 + d]}{\arg\max} \left\{A[i_1, i_2, \ldots, i_k] + B[i_1, i_2, \ldots, i_k]\right\},$$

4. Construct optimal alignments between $s_1[1..n_1/2], s_2[1..i_2], \ldots, s_k[1..i_k]$ and those between $s_1[n_1/2 + 1..n_1], s_2[i_2 + 1..n_2], \ldots, s_k[i_k + 1..n_k]$ in a recursive manner. The subproblems are decomposed until they are small enough to be solved within the available memory using *DBand* in k dimensions.
5. Combine produced optimal subalignments to obtain an overall optimal alignment constrained to the d-strip.

The pseudocode of the algorithm is given in Figure 3.4. In this code, the function *BestCut*, implemented as k nested **for** loops, finds optimal i_j's with respect to i_1 by searching in the band region of radius d around the main diagonal. In other words, it searches for optimal i_j's that are at most d positions away from the middle position in s_1, in their optimal alignment. For given k sequences, the algorithm finds their best multi-alignment that has at most kd spaces.

Complexity analysis

Let $T(n, d)$ be the time complexity of the algorithm running on k sequences. Then T can be estimated from the following recurrence:

$$T(n, d) \leq \begin{cases} c_1\, k^2 2^k\, n\, d^{k-1} & \text{if } n \leq \ell \\ 2T\left(n/2, d\right) + c_1\, k^2 2^k\, n\, d^{k-1} + c_2\, d^{k-1} & \text{otherwise} \end{cases} \tag{3.8}$$

for constants c_1 and c_2. After unfolding, we get a summation for T with $t = \log n - \log \ell$ terms:

$$\begin{aligned} T(n, d) &\leq \sum_{i=0}^{t-1} 2^i \left[c_1 k^2 2^k \left(\frac{n}{2^i}\right) d^{k-1} + c_2 \left(\frac{n}{2^i}\right) d^{k-2}\right] + \frac{n}{\ell}\left(c_1 k^2 2^k \ell^k\right) \\ &= c_1 k^2 2^k n d^{k-1} \sum_{i=0}^{t-1} 1^i + c_2\, d^{k-2} \sum_{i=0}^{t-1} 1^i + c_1\, k^2 2^k n \ell^{k-1} \qquad (3.9) \\ &= \mathcal{O}\left(k^2 2^k\, d^{k-1} n \log_2 n + d^{k-1} \log_2 n + k^2 2^k n \ell^{k-1}\right) \qquad (3.10) \end{aligned}$$

for $t > 0$ and $k > 1$. In the extreme cases, we have the following results:

Algorithm *BandDCMA*

input: set $s = \{s_1, s_2, \ldots, s_k\}$, integer d, indices $c_1, \ldots c_k$, $d_1, \ldots d_k$
output: optimal alignment of s

$n_j := |s_j|, \quad j \in \{1, \ldots, k\}$
$n := \{n_1, n_2, \ldots, n_k\}$
// *Base case: construct alignments in available memory with* $BandMalign$
if $n_1 * n_2 * \ldots * n_k < M$ **then** // *available memory* M, *set to* d^{k-1}
  return $BandMalign(s)$ // *takes d.p.matrix precomputed with DBand*
else
  // *General case: divide and solve recursively*
  $i_1 := \lfloor n_1/2 \rfloor$
  BandScore $(\{s_1[1..i_1], s_2[1..i_2], \ldots, s_k[1..n_k]\}, d,$ *pref-scores*$)$
  BandScoreRev $(\{s_1[i_1 + 1..n_1], s_2[i_2 + 1..n_2], \ldots, s_k[1..n_k]\}, d,$ *suff-scores*$)$
  // *To retrieve optimal cuts, find maximum among total scores*
  // *subject to constraints* $i_j \in \{i_1 - d, \ldots, i_1 + d\}$ *for* $j \in \{1, \ldots, k\}$
  $\{i_2, i_3, \ldots, i_k\} :=$ *BandCut* $(n, d,$ *pref-scores*, *suff-scores*$)$
  // *Two recursive calls solves two subproblems*
  BandDCMA $(\{s_1[c_1..i_1], s_2[c_2..i_2], \ldots, s_k[c_k..i_k]\}, d,$ *pref-align*$)$
  BandDCMA $(\{s_1[i_1 + 1..d_1], s_1[i_2 + 1..d_2] \ldots, s_k[i_k + 1..d_k]\}, d,$ *suff-align*$)$
  return *Concat* (*pref-align*, *suff-align*)

BandCut (m, d, A, B)
  $maxscore := -\infty$
  $bestbandcuts := \emptyset$
  $i_1 := \lfloor m_1/2 \rfloor$
  for $i_2 := 1$ **to** m_2 **do**
    for $i_3 :=$ $\max(0, i_2 - d)$ **to** $\min(m_3, i_2 + d)$ **do**
      $\ddots$
      for $i_k :=$ $\max(0, i_2 - d)$ **to** $\min(m_k, i_2 + d)$ **do**
        $C[i_1, i_2, \ldots, i_k] := A[i_1, i_2, \ldots, i_k] \; + \; B[i_1, i_2, \ldots, i_k]$
        if $C[i_1, i_2, \ldots, i_k] > maxscore$ **then**
          $maxscore := C[i_1, i_2, \ldots, i_k]$
  return *bestbandcuts*

Figure 3.4. *DC-algorithm for aligning within d-strip.*

- For $t = 0 \quad (\ell = n)$, the first two terms in Equation (3.9) disappear, while the third term becomes $\mathcal{O}\left(k^2 2^k \, n d^{k-1}\right)$, the run-time of the *DBand*.
- For $t = \log_2 n \quad (\ell = 1)$, the two summands in Equation (3.9) are computed to $\log_2 n$, while the third term reduces to $\mathcal{O}(k^2 2^k n)$. Hence, the expression for T in Equation (3.10) is reduced to $\mathcal{O}\left(k^2 2^k \, n \, d^{k-1} \log_2 n \; + \; d^{k-1} \log_2 n + k^2 2^k n\right)$. Since this is the asymptotically

same time as $\mathcal{O}\left(k^2 2^k\, nd^{k-1} \log_2 n\right)$, we conclude that T exceeds the time bound for *DBand* in k dimensions by a $\log_2 n$ factor.

Notice that we can get rid of the $\log_2 n$ factor in the time complexity expression T by decomposing the problem directly into $p > 2$ subproblems, which are then solved recursively. A generalization of *FastDCA* to k dimensions does exactly that. The cost to pay is in the space complexity because the generalized *FastDCA* must keep p $(k-1)$-dimensional rows in memory at the same time. Hence, the algorithm is a trade-off between the computational time and memory requirement. We skip the implementations details as it is straightforward extension of its pairwise counterpart.

The space complexity is $\mathcal{O}\left(kn + d^{k-1}\right)$ because the algorithm computes only those cells in the midrow that falls inside the d-strip. We sum up our results in the following theorem.

Theorem 3.4 *An optimal alignment of k sequences confined to the d-strip, can be computed in time $\mathcal{O}\left(k^2 2^k\, nd^{k-1} \log_2 n\right)$ and space $\mathcal{O}\left(kn + d^{k-1}\right)$, assuming that all sequences have the same size n and the SP measure is used.*

Chapter 4

Discussion

In our theoretical study of the sequence alignment problem, we have combined various algorithmic techniques including dynamic programming, divide-and-conquer and branch-and-bound. By combining recursive and iterative methods, we were able to improve the computation time and reduce space requirement of some standard alignment algorithms. We have also generalized the two-sequence methods to work with k sequences.

Our main result is the *BandDCMA* algorithm for multi-alignments within a specified diagonal band. It computes an optimal alignment between k sequences within a constant band of radius d considerably faster than standard dynamic programming, given that $d \ll n$ where n is the length of the shortest sequences. The quality of the computed alignment depends greatly on how we choose parameter d. Choosing d too small may result in a low-quality alignment. On the other hand, high values of d would make the algorithm as impractical as standard dynamic programming as computation time of our algorithm grows exponentially with d. As we have not implemented the algorithm, it is difficult to say which values of d are appropriate for aligning a set of rather similar sequences. One way to obtain a reasonable value for d is by computing the alignment score using a fast heuristic algorithm. Our algorithm can be then used to search for best alignments in the band of radius d around this heuristic alignment. It should also be mentioned that the stopping criterion for the recursion could be chosen differently. Exploring this option might be a possible topic for further studies. A natural extension of this work is to implement the algorithm.

Although multi-alignment algorithms in this thesis are straightforward generalizations of their pairwise counterparts, we hope that they will contribute to a better understanding of algorithmic techniques for the multi-alignment problem and lead to developing new algorithms.

References

[1] Alberts, B; Johnson A; Lewis, J; Raff, M; Roberts, K; Walter, P: *Molecular biology of the cell.* 4th edition. New York: Garland; 2002.

[2] Arvestad, L: *Algorithms for biological sequence alignment*, PhD thesis, Dec. 1999.

[3] Carrillo, H; Lipman, D: *The multiple sequence alignment problem in biology.* SIAM J. Appl. Math., 48(5):1073–1082, 1988.

[4] Chao, K; Pearson, W R; Miller, W: *Aligning two sequences within a specified diagonal band*, CABIOS, 8(1992), 481–487.

[5] Edmiston, E W, Core, N G; Saltz, J H; Smith, R M *Parallel processing of biological sequence comparison algorithms*, Journal of Parallel Program. 17 259–275 (1988).

[6] Hirschberg, D S: *A linear space algorithm for computing maximal common subsequences.* Commun. ACM, 18(6):341–343, 1975.

[7] Iyengar, A K: *Parallel characteristics of sequence alignment Algorithms.* ACM, 1989.

[8] Myers, E W; Miller, W: *Optimal alignments in linear space.* Comp. Appl. Biosci. 4:11–17 (1988).

[9] Setubal, J; Meidanis, J: *Introduction to computational molecular biology*. Boston: PWS Publishing, 1997.

[10] Stoye, J: *Divide-and-conquer multiple sequence alignment* Dissertation Thesis. Universität Bielefeld, Forschungsbericht der Technischen Fakultät, Abteilung Informationstechnik, Report 97–02, 1997.

[11] Wang, L; Jiang, T: *On the complexity of multiple sequence alignment.* J. Comput. Bio., 1:337–348, 1994.

[12] Wang, L; Jiang, T: *Algorithmic methods for multiple sequence alignments*, City U of Hong Kong, UC Riverside.

Appendix A: Time analysis of Hirschberg's algorithm in k dimensions

Let sequences $\mathbf{s} = (s_1, s_2, \ldots, s_k)$ have lengths $\mathbf{n} = (n_1, n_2, \ldots, n_k)$ and let $T(\mathbf{n})$ denote the time complexity of the algorithm. Then T can be estimated with the following recursion:

$$T(\mathbf{n}) \;\leq\; k^2\, 2^k \prod_{j=1}^{k} n_j \;+\; \prod_{j=2}^{k} n_j \;+\; T(n_1/2, \ldots, i_k) \;+\; T(n_1/2, \ldots, n_k - i_k).$$

Lemma 1 *The time bound for the algorithm DCMA is as follows:*

$$T(\mathbf{n}) \;\leq\; 2\, k^2\, 2^k \prod_{j=1}^{k} n_j \;+\; 2 \prod_{j=2}^{k} n_j.$$

Proof. We proceed by induction on n_1.

For $n_1 = 1$, no maximum computations occur so the hypothesis is obviously true:

$$T(\mathbf{n}) \;=\; k^2\, 2^k \prod_{j=2}^{k} n_j \;<\; 2 \left(k^2\, 2^k + 1 \right) \prod_{j=2}^{k} n_j$$

For $n_1 > 1$, we have

$$\begin{aligned}
T(\mathbf{n}) \;&\leq\; k^2\, 2^k \prod_{j=1}^{k} n_j \;+\; \prod_{j=2}^{k} n_j \;+\; T(n_1/2, \ldots, i_k) \;+\; T(n_1/2, \ldots, n_k - i_k) \\
&=\; \left(k^2\, 2^k\, n_1 + 1\right) \prod_{j=2}^{k} n_j \;+\; \left(k^2\, 2^k\, n_1 + 1\right) \prod_{j=2}^{k} i_j \;+\; \left(k^2\, 2^k\, n_1 + 1\right) \prod_{j=2}^{k} (n_j - i_j) \\
&=\; \left(k^2\, 2^k\, n_1 + 1\right) \left[\prod_{j=2}^{k} n_j \;+\; \prod_{j=2}^{k} i_j \;+\; \prod_{j=2}^{k} (n_j - i_j) \right] \\
&\leq\; 2\, k^2\, 2^k \prod_{j=1}^{k} n_j \;+\; 2 \prod_{j=2}^{k} n_j.
\end{aligned}$$

The last inequality holds because the inequality

$$\prod_{i=1}^{k} a_i \;+\; \prod_{i=1}^{k} b_i \;\leq\; \prod_{i=1}^{k} (a_i + b_i)$$

is always true for $a_i, b_i \geq 0$, where i is a positive integer. ∎