Foundations of Sequence Analysis

Lecture notes for a course in the winter semester 2013/2014

Stefan Kurtz

October 14, 2013

Contents

CHAPTER 1

Overview

1.1 Application Areas

Sequences or equivalently texts, strings, or words are a natural way to represent information. We give a short list of areas where sequences to be analyzed come from:

- molecular biology

DNA	$\ldots a\ a\ c\ g\ a\ c\ g\ t \ldots$	4 nucleotides,	length: $\approx 10^3 - 10^9$
RNA	$\ldots a\ u\ c\ g\ g\ c\ u\ t \ldots$	4 nucleotides,	length: $\approx 10^2 - 10^3$
proteins	$\ldots L\ I\ S\ A\ I\ S\ T\ L\ I\ E\ B \ldots$	20 aminoacids,	length: $\approx 10^2 - 10^3$

  L = leucin
  I = isoleucin
  S = serin
  A = alanin etc.

- phonetic spelling:

english	40 phoneme
japanese	113 “morae” (syllables)

- spoken language, birdsong: discretized measurements, multidimensional (specifying frequency and energy) on a dynamic time scale

- graphics: (r, g, b) vectors with $r, g, b \in [0, 255]$ for the intensity of the red, green, and blue color of a pixel.

- text processing: sequences in ASCII format
  (comparison of files, search in index, spelling correction)

- information transmission: sequence of bits, blockcodes in noisy channels substitution/synchronization errors — decoding and correction

For more examples, see [KS83], a classical texbook on string comparisons.

Usually sequences encoding experimental or natural information are almost always inexact. Thus similar sequences have in a lot of cases the same or similar meaning or effect. Thus a main part of this lecture will be devoted to notions of similarity, and we will show how to handle these notions algorithmically.

1.2 Problems on Strings

Here is a short list of problems occurring on sequences:

1. sequence comparison: compare two sequences and show the similarities and differences.
2. sequences matching: find all positions in a sequence where a pattern sequence occurs.
3. regular expression matching: find all positions in a text where a regular expression matches.
4. multiple sequence matching: find all positions in a text where one of the sequences in a given set matches.
5. approximate sequence matching: find all positions in a text where a sequence matches, allowing for errors in the match.
6. dictionary matching: for a given word w find the word v in a given set of words with maximal similarity to w.
7. data compression: find long duplicated substrings in a given text.
8. data compression: sort all suffixes of a given sequence lexicographically.
9. structural pattern matching: find regularities in sequences, like repeats, tandems ww, palindromes, or unique subsequences.

1.3 Further Reading

There are now several text books on biological sequence analysis. The focus of the books are different. [Ste94] and [CR94] cover mostly exact and approximate string matching techniques. [Gus97] is more complete on the biologically relevant techniques. [Wat95] covers similar topics but it was written from a mathematician's viewpoint. [SM97] is also recommended. The most recent book [Pev00] covers some new topics.

CHAPTER 2

Basic Notions and Definitions

Let S be a set. $|S|$ denotes the number of elements in S and $\mathcal{P}(S)$ refers to the set of subsets of S.

$\mathbb{N}$ denotes the set of positive integers including 0. $\mathbb{R}^+$ denotes the set of positive reals including 0. The symbols $h, i, j, k, l, m, n, q, r$ refer to integers if not stated otherwise. $|i|$ is the absolute value of i and $i \cdot j$ denotes the product of i and j.

2.1 Notation for Sequences

Let $\mathcal{A}$ be a finite set, the *alphabet*. The elements of $\mathcal{A}$ are *characters*. Strings are written by juxtaposition of characters, i.e. we write characters composing a sequence without separators like commas or spaces.

We introduce a special notation the sequence containing no characters: ε denotes the *empty sequence*. If the empty sequence is concatenated with a sequence, then we omit the empty sequence. That is $\varepsilon w = w\varepsilon = w$ for all sequences w. The set $\mathcal{A}^*$ of *sequences over* $\mathcal{A}$ is defined by

$$\mathcal{A}^* = \bigcup_{i \geq 0} \mathcal{A}^i$$

where

$$\mathcal{A}^i = \begin{cases} \{\varepsilon\} & \text{if } i = 0 \\ \{aw \mid a \in \mathcal{A}, w \in \mathcal{A}^{i-1}\} & \text{if } i > 0 \end{cases}$$

That is, $\mathcal{A}^i$ is the set of sequences of length i. This set is defined recursively: The first case (for $i = 0$) states that the only sequence of length zero is the empty sequence. The second case (for $i > 0$) states that the sequences of length $i > 0$ are composed by some character from $\mathcal{A}$ prepended to some sequence of length $i - 1$ from $\mathcal{A}^{i-1}$.

$\mathcal{A}^+$ denotes $\mathcal{A}^* \setminus \{\varepsilon\}$. That is, $\mathcal{A}^+$ is the set of all non-empty sequences over $\mathcal{A}$. The symbols a, b, c, d refer to characters and $p, s, t, u, v, w, x, y, z$ to sequences, unless stated otherwise.

Example 1 1. ASCII: 8-bit characters, encoding as defined by the ASCII standard

2. $\{A, \ldots, Z, a, \ldots, z, 0, \ldots, 9, \}$: alphanumeric subset of the ASCII-set

3. $\{A, \ldots, Z\} \setminus \{B, J, O, U, X, Z\}$: letter code for 20 amino acids

4. $\{a, c, g, t\}$: DNA alphabet (adenine, cytosine, guanine, thymine)

5. $\{R, Y\}$: purine/pyrimidine-alphabet

6. $\{I, O\}$: hydrophilic/hydrophobic nucleotides/amino acids

7. $\{+, -\}$: positive/negative electrical charge □

These examples show that the size of the alphabets can be quite different. The alphabet size is an important parameter when determining the efficiency of several algorithms.

The *length* of a sequence s, denoted by $|s|$, is the number of characters in s. We make no distinction between a character and a sequence of length one.

Example 2 Let $\mathcal{A} = \{b, c\}$. Then ε is a sequence of length 0, and $bccb$ is a sequence of length 4. b and c are characters in $\mathcal{A}$ but also sequences of length 1. We can formally determine the set $\mathcal{A}^2$ by applying the above definitions as follows:

$$\begin{aligned}
\mathcal{A}^1 &= \{aw \mid a \in \mathcal{A}, w \in \mathcal{A}^0\} \\
&= \{aw \mid a \in \{b, c\}, w \in \{\varepsilon\}\} \\
&= \{a\varepsilon \mid a \in \{b, c\}\} \\
&= \{b\varepsilon, c\varepsilon\} \\
&= \{b, c\} \\
\mathcal{A}^2 &= \{aw \mid a \in \mathcal{A}, w \in \mathcal{A}^1\} \\
&= \{aw \mid a \in \{b, c\}, w \in \{b, c\}\} \\
&= \{bb, bc, cb, cc\}
\end{aligned}$$

If $s = uvw$ for some (possibly empty) sequences u, v and w, then

- u is a *prefix* of s,
- v is a *substring* of s, and
- w is a *suffix* of s.

Example 3 Let $s = acca$. s has the suffixes $acca$, cca, ca, a, and ε. s has the prefixes ε, a, ac, acc and $acca$. Besides the suffixes and prefixes s has the substrings c and cc.

A prefix or suffix of s is *proper* if it is different from s. A suffix of s is *nested* if it occurs more than once in s. A set S of sequences is *prefix-closed* if $u \in S$ whenever $ua \in S$. A set S of sequences is *suffix-closed* if $u \in S$ whenever $au \in S$. A substring v of s is *right-branching* if there are different characters a and b such that va and vb are substrings of s. Let $q > 0$. A *q-gram* of s is a substring of s of length q. q-grams are sometimes called q-tuples.

$s[i]$ is the i-th character of s. That is, if $|s| = n$, then $s = s[1 \ldots n]$ where $s[i] \in \mathcal{A}$. $s[n]s[n-1] \ldots s[1]$, denoted by s^{-1}, is the *reverse* of $s = s[1 \ldots n]$. If $i \leq j$, then $s[i \ldots j]$ is the substring of s beginning with the i-th character and ending with the j-th character. If $i > j$, then $s[i \ldots j]$ is the empty sequence. A sequence w begins at position i and ends at position j in s if $s[i \ldots j] = w$.

2.2 Algorithms, Efficiency and O-Notation

An algorithm is a procedure (a finite set of well-defined instructions) for accomplishing some task which, given an initial state, will terminate in a defined end-state. The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures.

Informally, the concept of an algorithm is often illustrated by the example of a recipe, although many algorithms are much more complex; algorithms often have steps that repeat (iterate) or require decisions (such as logic or comparison). Algorithms can be composed to create more complex algorithms.

In computer science, efficiency is used to describe several desirable properties of an algorithm or other construct, besides clean design, functionality, etc. Efficiency is generally contained in two properties: speed (the time it takes for an operation to complete), and space (maximum amount of the memory used up by the construct at any time of algorithm execution).

The speed and space of an algorithm is measured in various ways. In general, we estimate the running time and/or space of algorithms rather than count the exact number of machine instructions or machine words required by an algorithm. To obtain such information, two rules are necessary: The first is that the running time and space requirement is given for the worst case instance of the problem to be solved. The second is that we make the running time and space requirement depend on the size of the instance being solved. That is, instead of being a number, we measure the running time and space by a function that expresses the relation of the input size to the number of executed machine instruction/used machine words. For example, we could say that the running time of an algorithm is $5n^2 + 3n + 72$, where n is the size of the problem. We will however simplify running time and space results by dropping all constants and lower terms. Thus, in the term above we could drop $3n$ and 72, because they are lower-order terms with respect to n^2. We also drop the constant 5 and state that the running time of the algorithm is $O(n^2)$. The O-notation (Big-O) is used to indicate that we do not care for constants and lower-order terms. Using the O-notation, it is much simpler to compare algorithms concerning their running time and space requirements.

The O-notation, also asymptotic notation, is a mathematical notation used to describe the asymptotic behavior of functions. Its purpose is to characterize a function's behavior for very large (or very small) inputs in a simple but rigorous way that enables comparison to other functions. More precisely, the symbol O is used to describe an asymptotic upper bound for the magnitude of a function in terms of another, usually simpler, function. There are also symbols for lower bounds and tight bounds which are not discussed here.

Suppose f and g are two functions from integers to integers. We say that f is $O(g)$ if and only if there exists some constants n_0 and $c > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Example 4 Consider the functions f and g defined as follows:

$$\begin{aligned} f(n) &= 6n^4 - 2n^3 + 5 \\ g(n) &= n^4 \end{aligned}$$

Now for $n \geq 1$ the following inequality holds:

$$\begin{aligned} f(n) = 6n^4 - 2n^3 + 5 &\leq 6n^4 + 2n^3 + 5 \\ &\leq 6n^4 + 2n^4 + 5n^4 \\ &\leq 13n^4 \\ &= 13g(n) \end{aligned}$$

With $c = 13$ and $n_0 = 1$ we can conclude $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. That is, f is $O(g)$.

Note that by $O(g)$ we refer to the set of functions f such that f is $O(g)$. That is, $O(g)$ is the set of functions dominated by g.

When using the O-Notation, one does not always have to give explicit names to functions, like f and g as in the example above. Instead, one uses polynomials: For example, if we want to refer to the set $O(g)$ with g defined by $g(n) = n^2$ we simply write $O(n^2)$. Then we can, for example, state that the running time of some algorithm is $O(n^2)$. This means that the function f describing the relation of input size to running time of the algorithm is $O(n^2)$. Table 2.1 shows some commonly used functions for specifying asymptotic upper bounds.

Notation	Name	Example from this lecture notes
$O(1)$	constant	determining cost of an edit operation
$O(\log n)$	logarithmic	finding an item in a sorted array
$O(n)$	linear	determining the identity of two equal length sequences
$O(n \log n)$	quasilinear	determining an optimal chain
$O(n^2)$	quadratic	determining the edit distance of two sequences
$O(n^c)$, $c > 1$	polynomial	
$O(c^n)$	exponential	recursive counting of all alignments
$O(n!)$	factorial	

Table 2.1: Some commonly used functions for specifying asymptotic upper bounds.

CHAPTER 3

String Comparisons

The comparison of sequences is an important operation applied in several fields, such as molecular biology, speech recognition, computer science, and coding theory. The most important model for sequence comparison is the model of edit distance. It measures the distance between sequences in terms of edit operations, that is, deletions, insertions, and replacements of single characters. Two sequences are compared by determining a sequence of edit operations that converts one sequence into the other and minimizes the sum of the operations' costs. Such a sequence can be computed in time proportional to the product of the lengths of the two sequences, using the technique of dynamic programming.

The edit distance is a measure of local similarities in which matches between substrings are highly dependent on their relative positions in the sequences. There are situations where this property is not desired. Suppose one wants to consider sequences as similar which differ only by an exchange of large substrings. This occurs, for instance, if a text file has been created from another by a block move operation. In such a case, the edit distance model should not be used since it gives a large edit distance. There are two other sequence comparison models that are more appropriate for this case: The maximal matches model and the q-gram model.

The idea of the maximal matches model is to count the minimal number of occurrences of characters in one sequence such that if these characters are "crossed out", the remaining substrings are all substrings of the other sequence. Thus, sequences with long common substrings have a small distance. The idea of the q-gram model is to count the number of occurrences of different q-grams in the two sequences. Thus, sequences with many common q-grams have a small distance, independent of where they occur. A very interesting aspect is that the maximal matches distance and the q-gram distance of two sequences can be computed in time proportional to the sum of the lengths of the two sequences.

When comparing biological sequences, the edit distance computation is often to expen-

sive, while the order of the sequence characters is still important. Therefore heuristics have been developed, which approximate the edit distance model. Two of these heuristics are described in Sections 4.1 and 4.2.

In the following, we first consider the issue of sequence comparison in general. Then we describe the three models of sequence comparison in details and give algorithm to compute the respective distances. For the rest of this section let u and v be sequences of length m and n, respectively.

3.1 The Problem

The trivial method to compare two sequences is to compare them character by character: u and v are equal if and only if $|u| = |v|$ and $u[i] = v[i]$ for $i \in [1, n]$. However, this comparison model is too restrictive for several problems:

- searching for a name of which the spelling is not exactly known
- finding inflected forms of a word
- accounting for typing errors
- tolerating error prone experimental measurements
- allowing for ambiguities in the genetic code, e.g. gcu, gcc, gca, and gcg all code for alanin.
- searching for a protein with unknown function, a "similar" protein sequence, whose biological function is known.

To be more general one has to define a function $f : \mathcal{A}^* \times \mathcal{A}^* \to \mathbb{R}$, which delivers a qualitative measure of distance/similarity. Note that there is a duality in the notions "distance" and "similarity": the smaller the distance, the larger the similarity.

Let M be a set and $f : M \times M \to \mathbb{R}^+$ be a function. f is a *metric* on M if for all $x, y, z \in M$ the following properties hold:

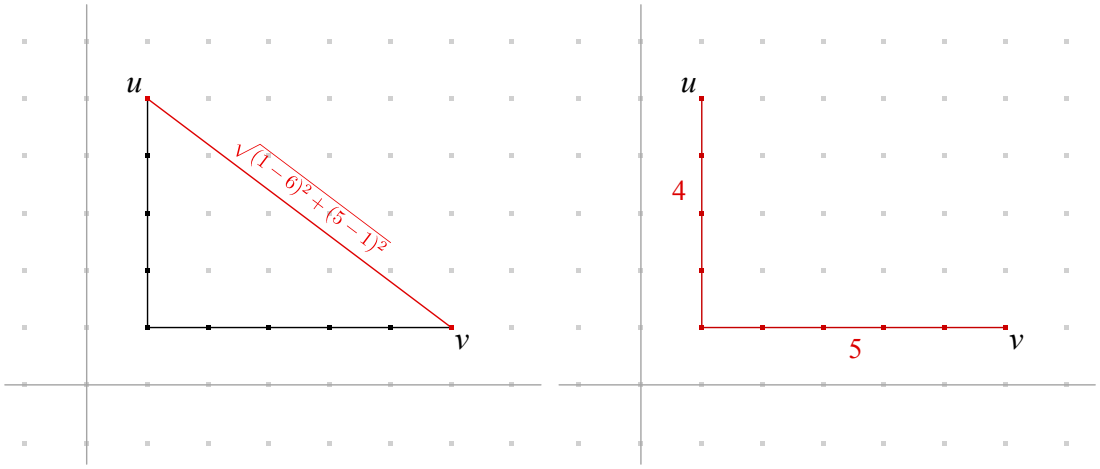
Zero Property	$f(x, y) = 0 \iff x = y$
Symmetry	$f(x, y) = f(y, x)$
Triangle Inequality	$f(x, y) \leq f(x, z) + f(z, y).$

If the symmetry and the triangle inequality hold, and also

$$x = y \Rightarrow f(x, y) = 0$$

then f is a *pseudo-metric* on M.

Figure 3.1: Illustration of the two-dimensional euclidean and block distance ($n = 2$). In this example, $u = (1, 5)$ and $v = (6, 1)$. (a) shows that $f_e(u, v) = \sqrt{(1-6)^2 + (5-1)^2} \approx 6.4$. Likewise, (b) shows that $f_b(u, v) = |1-6| + |5-1| = 5 + 4 = 9$.



(a) euclidean distance (b) block distance

Example 5 Let $\mathcal{A}$ be a finite subset of the integers. Suppose $n > 0$ and $M = \mathcal{A}^n$. Then we define the following distance notions:

$$\text{euclidean distance:} \quad f_e(u, v) = \sqrt{\sum_{i=1}^{n} (u[i] - v[i])^2}$$

$$\text{block distance:} \quad f_b(u, v) = \sum_{i=1}^{n} |u[i] - v[i]|$$

$$\text{hamming distance:} \quad f_h(u, v) = |\{i \mid 1 \leq i \leq n, u[i] \neq v[i]\}| \quad \square$$

These distance notions only make sense for sequences of the same length. The euclidean distance is the distance of the points (represented by strings of numbers) in euclidean space. Let $n = 2$. Suppose that $u[1] \geq v[1]$ and $u[2] \geq v[2]$ and let $a = u[1] - v[1]$, $b = u[2] - v[2]$, and c be the euclidean distance of u and v. Then, by the Pythagorean theorem, we have $a^2 + b^2 = c^2$ which implies $c = \sqrt{a^2 + b^2} = \sqrt{(u[1] - v[1])^2 + (u[2] - v[2])^2} = f_e(u, v)$.

For $n = 2$, the block-distance assumes a topology of the space in which one can only go left, right, up or down. So the block-distance is the number of vertical movements (left or right) plus the number of horizontal movements (up or down) to come from u to v.

The hamming distance counts the number of positions at which the sequences are different.

The distance notions we consider in the rest of this section are also defined for sequences of different length.

3.2 The Edit Distance Model

The notion of edit operations, introduced by Ulam [Ula72], is the key to the edit distance model.

Definition 1 An *edit operation* is a pair $(\alpha, \beta) \in (\mathcal{A}^1 \cup \{\varepsilon\}) \times (\mathcal{A}^1 \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$. □

α and β denote *sequences* of length ≤ 1. However, if $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$, then the edit operation (α, β) is identified with a pair of characters.

An edit operation (α, β) is usually written as $\alpha \to \beta$. This reflects the operational view which considers edit operations as rewrite rules transforming a source sequence into a target sequence, step by step. In particular, there are three kinds of edit operations:

- $a \to \varepsilon$ denotes the *deletion* of the character $a \in \mathcal{A}$,
- $\varepsilon \to b$ denotes the *insertion* of the character $b \in \mathcal{A}$,
- $a \to b$ denotes the *replacement* of the character $a \in \mathcal{A}$ by the character $b \in \mathcal{A}$.

Notice that $\varepsilon \to \varepsilon$ is not an edit operation. Insertions and deletions are sometimes referred to collectively as *indels*.

There is no uniform naming for an edit operation of the third type. Some authors use the term replacement [KS83, MM88], as we do. Others instead prefer the terms substitution [Wat89, Mye91] and change [WF74, Ukk93].

Sometimes sequence comparison just means to measure how different sequences are. Often it is additionally of interest to analyze the total difference between two sequences into a collection of individual elementary differences [KS83]. The most important mode of such analyses is an alignment of the sequences.

Definition 2 An *alignment* A of u and v is a sequence

$$(\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$$

of edit operations such that $u = \alpha_1 \ldots \alpha_h$ and $v = \beta_1 \ldots \beta_h$. □

Note that the unique alignment of ε and ε is the empty alignment, that is, the empty sequence of edit operations. An alignment is usually written by placing the characters of the two aligned sequences on different lines, with inserted dashes denoting ε. In such a representation, every column represents an edit operation.

Example 6 The alignment $A = (\varepsilon \to d, b \to b, c \to a, \varepsilon \to d, a \to a, c \to \varepsilon, d \to d)$ of the sequences $bcacd$ and $dbadad$ is written as follows:

$$\begin{pmatrix} - & b & c & - & a & c & d \\ d & b & a & d & a & - & d \end{pmatrix}$$

□

Example 7 Five alignments of $u = gabh$ and $v = gcdhb$:

$$A_1 = \begin{pmatrix} g & - & a & b & - & h & - \\ g & c & - & - & d & h & b \end{pmatrix} \quad A_2 = \begin{pmatrix} g & - & a & - & b & h & - \\ g & c & - & d & - & h & b \end{pmatrix} \quad A_3 = \begin{pmatrix} g & - & - & a & b & h & - \\ g & c & d & - & - & h & b \end{pmatrix}$$

$$A_4 = \begin{pmatrix} g & a & - & - & b & h \\ g & c & d & h & b & - \end{pmatrix} \qquad A_5 = \begin{pmatrix} g & a & b & h & - \\ g & c & d & h & b \end{pmatrix} \quad \square$$

Observation 1 Let u and v be sequences of length m and n, respectively. Let $A = (\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$ be an alignment of u and v. Then $m + n \geq h \geq \max\{m, n\}$.

Proof:

1. The alignment
$$\begin{pmatrix} u[1] & u[2] & \ldots & u[m] & - & - & \ldots & - \\ - & - & \ldots & - & v[1] & v[2] & \ldots & v[n] \end{pmatrix}$$
of u and v is of maximal length. Its length is $m + n$. Hence $m + n \geq h$.

2. Let $m \geq n$. Then
$$\begin{pmatrix} u[1] & u[2] & \ldots & u[n] & u[n+1] & \ldots & u[m] \\ v[1] & v[2] & \ldots & v[n] & - & \ldots & - \end{pmatrix}$$
is an alignment of u and v of minimal length. Hence $h \geq m = \max\{m, n\}$.

3. Let $m < n$. Then
$$\begin{pmatrix} u[1] & u[2] & \ldots & u[m] & - & \ldots & - \\ v[1] & v[2] & \ldots & v[m] & v[m+1] & \ldots & v[n] \end{pmatrix}$$
is an alignment of u and v of minimal length. Hence $h \geq n = \max\{m, n\}$.

3.2.1 The Number of Alignments

For all $m, n \geq 0$ let $Aligns(m, n)$ be the number of alignments of two fixed sequences, say u and v of length m and n. We will derive a recursive equation for $Aligns(m, n)$ by case distinction:

1. Let $m = 0$ or $n = 0$. According to Observation 1, an alignment of two sequences of length m and n has length h, where h satisifies $m + n \geq h \geq \max\{m, n\}$.
   - If $m = n = 0$ we conclude $0 \geq h \geq 0$, i.e. $h = 0$. That is, the alignment has length 0, and there is exactly one such alignment, namely the empty alignment.
   - If $m = 0$ and $n > 0$, then $m + n = n \geq h \geq n = \max\{m, n\}$, which implies $h = n$. That is, the alignment is of length n. Since it aligns an empty string u with a non-empty string v, it must consist of exactly n insertions. Of course, there is exactly one such alignment of u and v.

- If $m > 0$ and $n = 0$, then $m + n = m \geq h \geq m = \max\{m, n\}$, which implies $h = m$. That is, the alignment is of length m. Since it aligns a non-empty string u with an empty string v, and it must consist of exactly m deletions. Of course, there is exactly one such alignment.

Thus in this case we obtain $Aligns(m, n) = 1$.

2. Now let $m > 0$ and $n > 0$, i.e. u and v are both not empty. An alignment of u and v contains at least one edit operation. We make a case distinction about the kind of the last edit operation:
   - Consider all alignments of u and v ending with a deletion. They all delete the last character of u, i.e. they end with the same deletion. Thus the number of alignments ending with a deletion is the same as the number of all alignments of $u[1 \ldots m-1]$ and v. This is $Aligns(m-1, n)$. Thus there are $Aligns(m-1, n)$ alignments ending with a deletion.
   - Consider all alignments of u and v ending with an insertion. They all insert the last character of v, i.e. they end with the same insertion. Thus the number of alignments ending with an insertion is the same as the number of all alignments of u and $v[1 \ldots n-1]$. By definition, this is $Aligns(m, n-1)$. Thus there are $Aligns(m, n-1)$ alignments ending with an insertion.
   - Consider all alignments of u and v ending with a replacement. They all replace the last character of u with the last character of v, i.e. they end with the same replacement. Thus the number of alignments ending with a replacement is the same as the number of all alignments of $u[1 \ldots m-1]$ and $v[1 \ldots n-1]$. By definition, this is $Aligns(m-1, n-1)$. Thus there are $Aligns(m-1, n-1)$ alignments ending with a replacement.

   Combining all three cases, we conclude that the number of alignments $Aligns(m, n)$ is
   $$Aligns(m-1, n) + Aligns(m, n-1) + Aligns(m-1, n-1).$$

Altogether, we obtain the following recurrence:

$$Aligns(m, n) = \begin{cases} 1 & \text{if } m = 0 \text{ or } n = 0 \\ Aligns(m-1, n) + \\ Aligns(m-1, n-1) + \\ Aligns(m, n-1) & \text{otherwise} \end{cases}$$

$Aligns(n, n)$ can be approximated by the Stanton-Cowan-Numbers:

$$Aligns(n, n) \approx \left(1 + \sqrt{2}\right)^{2n+1} \cdot \sqrt{n}$$

For $n = 1000$ we have $Aligns(n, n) \approx \left(1 + \sqrt{2}\right)^{2001} \cdot \sqrt{1000} = 10^{767.4\ldots}$

In other words, the number of alignments is exponential in the length of the aligned sequences.

If one considers sets of alignments of the same sequence, one quickly recognizes that the order of insertions and deletions immediately following each other in an alignment is not important. For example, the two alignments

$$\begin{pmatrix} a & - \\ - & b \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} - & a \\ b & - \end{pmatrix} \tag{3.1}$$

should be considered equivalent. The idea is to ignore such difference and to restrict to the important parts of the alignment, namely those pairs of characters of u and v which appear in replacements. This results in the notion of subsequences:

Definition 3 A *subsequence* of u and v is a sequence of index pairs

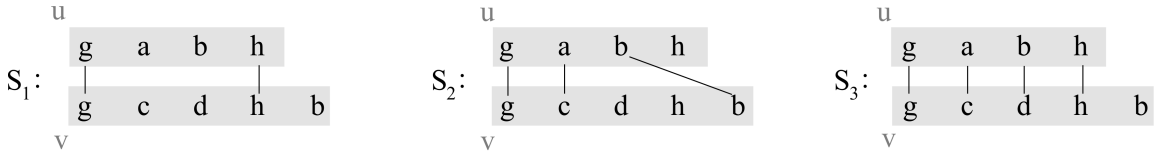
$$(i_1, j_1), \ldots, (i_r, j_r)$$

such that

$$1 \leq i_1 < \ldots < i_r \leq m \text{ and}$$
$$1 \leq j_1 < \ldots < j_r \leq n. \ \square$$

The index pair (i_h, j_h) stands for the replacement $u[i_h] \rightarrow v[j_h]$. All characters in u and v not occurring in a subsequence are considered to be deleted in u or v. For example, the empty subsequence stands for the alignments in (3.1). In a graphical representation, the index pairs of the subsequence appear as lines connecting the characters in the subsequence.

Example 8 The following subsequences of $u = gabh$ and $v = gcdhb$ represent the alignments of Example 7. In particular, $S_1 = (1,1), (4,4)$ represents A_1, A_2, and A_3, while $S_2 = (1,1), (2,2), (3,5)$ represents A_4, and $S_3 = (1,1), (2,2), (3,3), (4,4)$ represents A_5.



Observation 2 Let $Subseqs(m, n)$ be the number of subsequences of two fixed sequences of length m and n. Then

$$Subseqs(m, n) = \sum_{r=0}^{\min(m,n)} \binom{m}{r} \cdot \binom{n}{r} \tag{3.2}$$

Proof: Let r, $0 \leq r \leq \min\{m, n\}$ be arbitrary but fixed. For the ordered selection of the indices $i_1, \ldots, i_r$ there are $\binom{m}{r}$ possibilities; for the ordered selection of the indices $j_1, \ldots, j_r$ there are $\binom{n}{r}$ possibilities. All these possibilities have to be combined which gives $\binom{m}{r} \cdot \binom{n}{r}$ combinations. Summing over all possible r in the range from 0 to $\min\{m, n\}$ gives Equation (3.2). $\square$

$Subseqs(n,n)$ can be approximated by $2^{2n}\,(4\cdot\sqrt{n\pi})^{-1}$, e.g.

$$Subseqs(1000,1000) \approx 10^{600}$$

So we conclude that the number of alignments is larger than the number of subsequences (by a factor of about 10^{167}). As both numbers grow exponentially with the sequence length, this difference does not really matter in practice.

3.2.2 The Edit Distance Problem

The notion of optimal alignment requires some scoring or optimization criterion. This is given by a cost function.

Definition 4 A *cost function* δ assigns to each edit operation $\alpha\to\beta$, $\alpha\neq\beta$, a positive real cost $\delta(\alpha\to\beta)$. The cost $\delta(\alpha\to\alpha)$ of an edit operation $\alpha\to\alpha$ is 0. If $\delta(\alpha\to\beta)=\delta(\beta\to\alpha)$ for all edit operations $\alpha\to\beta$ and $\beta\to\alpha$, then δ is *symmetric*. If $\delta(\alpha\to\beta)=1$, for all edit operations $\alpha\to\beta$, $\alpha\neq\beta$, then δ is the *unit cost function*. δ is extended to alignments in a straightforward way: The cost $\delta(A)$ of an alignment $A=(\alpha_1\to\beta_1,\ldots,\alpha_h\to\beta_h)$ is the sum of the costs of the edit operations A consists of. More precisely,

$$\delta(A)=\sum_{i=1}^{h}\delta(\alpha_i\to\beta_i).\ \square$$

Example 9

1. Let
$$\delta(\alpha\to\beta)=\begin{cases}0 & \text{if } \alpha,\beta\in\mathcal{A} \text{ and } \alpha=\beta\\ 1 & \text{otherwise}\end{cases}$$
Then δ is the *unit cost*.

2. Let
$$\delta(\alpha\to\beta)=\begin{cases}0 & \text{if } \alpha,\beta\in\mathcal{A} \text{ and } \alpha=\beta\\ 1 & \text{else if } \alpha,\beta\in\mathcal{A} \text{ and } \alpha\neq\beta\\ \infty & \text{otherwise}\end{cases}$$
Then δ is the *hamming cost*.

3. Suppose δ is given by the following table:

δ	ε	A	C	G	T
ε		3	3	3	3
A	3	0	2	1	2
C	3	2	0	2	1
G	3	1	2	0	2
T	3	2	1	2	0

Then δ is the transversion/transition cost function. Bases A and G are called *purine*, and bases C and T are called *pyrimidine*. The transversion/transition cost function reflects the biological fact that a purine/purine and a pyrimidine/pyrimidine replacement is much more likely to occur than a purine/pyrimidine replacement. Moreover, it takes into account that indels occur less often than replacements and thus assigns cost 3 for an indel.

4. The following tables shows costs for replacements of amino acids, as suggested by Willy Taylor. For a cost function we would have to define the indel costs:

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	0	14	7	9	20	9	8	7	12	13	17	11	14	24	7	6	4	32	23	11
R	14	0	11	15	26	11	14	17	11	18	19	7	16	25	13	12	13	25	24	18
N	7	11	0	6	23	5	6	10	7	16	19	7	16	25	10	7	7	30	23	15
D	9	15	6	0	25	6	3	9	11	19	22	10	19	29	11	11	10	34	27	17
C	20	26	23	25	0	26	25	21	25	22	26	25	26	26	22	20	21	34	22	21
Q	9	11	5	6	26	0	5	12	7	17	20	8	16	27	10	11	10	32	26	16
E	8	14	6	3	25	5	0	9	10	17	21	10	17	28	11	10	9	34	26	16
G	7	17	10	9	21	12	9	0	15	17	21	13	18	27	10	9	9	33	26	15
H	12	11	7	11	25	7	10	15	0	17	19	10	17	24	13	11	11	30	21	16
I	13	18	16	19	22	17	17	17	17	0	9	17	8	17	16	14	12	31	18	4
L	17	19	19	22	26	20	21	21	19	9	0	19	7	14	20	19	17	27	18	10
K	11	7	7	10	25	8	10	13	10	17	19	0	15	26	11	10	10	29	25	16
M	14	16	16	19	26	16	17	18	17	8	7	15	0	18	17	16	13	29	20	8
F	24	25	25	29	26	27	28	27	24	17	14	26	18	0	27	24	23	24	8	19
P	7	13	10	11	22	10	11	10	13	16	20	11	17	27	0	9	8	32	26	14
S	6	12	7	11	20	11	10	9	11	14	19	10	16	24	9	0	5	29	22	13
T	4	13	7	10	21	10	9	9	11	12	17	10	13	23	8	5	0	31	22	10
W	32	25	30	34	34	32	34	33	30	31	27	29	29	24	32	29	31	0	25	32
Y	23	24	23	27	22	26	26	26	21	18	18	25	20	8	26	22	22	25	0	20
V	11	18	15	17	21	16	16	15	16	4	10	16	8	19	14	13	10	32	20	0

Definition 5 The *edit distance* of u and v, denoted by $edist_\delta(u, v)$, is the minimum possible cost of an alignment of u and v. That is,

$$edist_\delta(u, v) = \min\{\delta(A) \mid A \text{ is an alignment of } u \text{ and } v\}.$$

An alignment A of u and v is *optimal* if $\delta(A) = edist_\delta(u, v)$. If δ is the unit cost function, then $edist_\delta(u, v)$ is the *unit edit distance* between u and v. □

By definition, δ satisfies the zero property. If δ is symmetric and satisfies the triangle inequality, then $edist_\delta$ is a metric, as shown in [WF74]. Note that there can be more than one optimal alignment. The unit edit distance is sometimes called Levenshtein distance. The following observation states a simple property of the edit distance.

Observation 3 For any cost function δ and any two sequences $u, v \in \mathcal{A}^*$ the following equation holds:

$$edist_\delta(u, v) = edist_\delta(u^{-1}, v^{-1})$$

Proof: Let $A = (\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$ be an optimal alignment of u and v. Obviously, $A^{-1} = (\alpha_h \to \beta_h, \ldots, \alpha_1 \to \beta_1)$ is an alignment of u^{-1} and v^{-1}. Now suppose there is an alignment X of u^{-1} and v^{-1} such that $\delta(X) < \delta(A^{-1})$. That is, A^{-1} is not the optimal alignment of u^{-1} and v^{-1}. Now X^{-1} is an alignment of u and v and we have $\delta(X^{-1}) = \delta(X) < \delta(A^{-1}) = \delta(A)$. Thus A is not an optimal alignment. This is a contradiction. Hence our assumption above was wrong, i.e. there is no alignment X of u^{-1} and v^{-1} with $\delta(X) < \delta(A^{-1})$. As a consequence

$$edist_\delta(u, v) = \delta(A) = \delta(A^{-1}) = edist_\delta(u^{-1}, v^{-1}) \ \square$$

Definition 6 The *edit distance problem* is to compute the edit distance and all optimal alignments. □

By specifying the cost functions, we obtain special forms of edit distances:

Definition 7

- If δ is the unit cost, then $edist_\delta$ is the *unit edit distance* or *Levenshtein distance*.
- If δ is the hamming cost, then $edist_\delta$ is the *hamming distance*.

3.2.3 A Dynamic Programming Algorithm

A simple algorithm for solving this problem has been independently discovered by many different authors (see [KS83] for an account). In this section, we especially refer to the simplest version which is due to Wagner and Fischer [WF74]. All other differ somewhat. But, as remarked in [KS83], they can be seen as flowing out of the same concept.

Suppose a cost function δ is given and $u, v \in \mathcal{A}^*$ are fixed but arbitrary. We will now develop some recursive equation for We will develop the algorithm $edist_\delta(u, v)$ from which we derive a dynamic programming algorithm.

Consider an optimal alignment

$$A = \begin{pmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_h \\ \beta_1 & \beta_2 & \dots & \beta_h \end{pmatrix}$$

of $u = u[1 \dots m]$ and $v = v[1 \dots n]$. Then $\delta(A) = edist_\delta(u, v)$.

- Case (0): $u = \varepsilon$ and $v = \varepsilon$. Then A is the empty alignment and $\delta(A) = 0 = edist_\delta(u, v)$.
- Case (1): $u = \varepsilon$ and $v \neq \varepsilon$ Since $u = \alpha_1\alpha_2 \dots \alpha_h$, we conclude $\alpha_i = \varepsilon$ for all i, $1 \leq ih$. Hence $h = n$, and $\beta_j = v[j]$ for all j, $1 \leq j \leq h$. That is, A consists of insertions only and the cost of A is

$$\begin{aligned} \delta(A) &= \sum_{j=1}^{n} \delta(\varepsilon \to \beta_j) \\ &= \sum_{j=1}^{n} \delta(\varepsilon \to v[j]) \\ &= \sum_{j=1}^{n-1} \delta(\varepsilon \to v[j]) + \delta(\varepsilon \to v[n]) \end{aligned}$$

  With $v' = v[1 \dots n-1]$ and $b = v[n]$ we have $v = v'b$ and can conclude $edist_\delta(\varepsilon, v'b) = edist_\delta(\varepsilon, v') + \delta(\varepsilon \to b)$.

- Case (2): $v = \varepsilon$ and $u \neq \varepsilon$. Since $v = \beta_1\beta_2 \ldots \beta_h$, we conclude $\beta_j = \varepsilon$ for all j, $1 \leq j \leq h$. Hence $h = m$ and $\alpha_i = u[i]$ for all i, $1 \leq i \leq h$. That is, A consists of deletions only and the cost of A is

$$\begin{aligned}\delta(A) &= \sum_{i=1}^{m} \delta(\alpha_i \to \varepsilon) \\ &= \sum_{i=1}^{m} \delta(u[i] \to \varepsilon) \\ &= \sum_{i=1}^{m-1} \delta(u[i] \to \varepsilon) + \delta(u[m] \to \varepsilon)\end{aligned}$$

  With $u' = u[1 \ldots m-1]$ and $a = u[m]$ we have $u = u'a$ and can conclude $edist_\delta(u'a, \varepsilon) = edist_\delta(u', \varepsilon) + \delta(a \to \varepsilon)$.

- Case (3): $u \neq \varepsilon$ and $v \neq \varepsilon$. Then $u = u'a$ and $v = v'b$ for some $u', v' \in \mathcal{A}^*$ and some $a, b \in \mathcal{A}$. Now split A into an alignment A' consisting of the first $h-1$ edit operations and the hth edit operation:

$$A = \begin{pmatrix} A' & \alpha_h \\ & \beta_h \end{pmatrix}$$

  – Case (3a): $\alpha_h = a$ and $\beta_h = \varepsilon$. Then A' is an alignment of u' and v. Suppose that A' is not optimal. Let A'' be an optimal alignment of u' and v. Hence $\delta(A') > \delta(A'') = edist_\delta(u', v)$. Now $A''\alpha_h \to \beta_h$ is an alignment of u and v and we conclude

$$\begin{aligned}edist_\delta(u, v) &= \delta(A) \\ &= \delta(A') + \delta(\alpha_h \to \beta_h) \\ &> \delta(A'') + \delta(a \to \varepsilon) \\ &\geq edist_\delta(u, v)\end{aligned}$$

    This is a contradiction. Hence A' is an optimal alignment of u' and v, and $edist_\delta(u, v) = \delta(A) = \delta(A') + \delta(a \to \varepsilon) = edist_\delta(u', v) + \delta(a \to \varepsilon)$. The following case (3b) handles insertions and case (3c) handles replacements in an analogous way.

  – Case (3b): $\alpha_h = \varepsilon$ and $\beta_h = b$. Then A' is an alignment of u and v'. Suppose that A' is not optimal. Let A'' be an optimal alignment of u and v'. Hence $\delta(A') > \delta(A'') = edist_\delta(u, v')$. Now $A''\alpha_h \to \beta_h$ is an alignment of u and v and we conclude

$$\begin{aligned}edist_\delta(u, v) &= \delta(A) \\ &= \delta(A') + \delta(\alpha_h \to \beta_h) \\ &> \delta(A'') + \delta(\varepsilon \to b) \\ &\geq edist_\delta(u, v)\end{aligned}$$

This is a contradiction. Hence A' is an optimal alignment of u and v' and $edist_\delta(u,v) = \delta(A) = \delta(A') + \delta(\varepsilon \to b) = edist_\delta(u,v') + \delta(\varepsilon \to b)$.

– Case (3c): $\alpha_h = a$ and $\beta_h = b$. Then A' is an alignment of u' and v'. Suppose that A' is not optimal. Let A'' be an optimal alignment of u' and v'. Hence $\delta(A') > \delta(A'') = edist_\delta(u',v')$. Now $A''\alpha_h \to \beta_h$ is an alignment of u and v and we conclude

$$\begin{aligned} edist_\delta(u,v) &= \delta(A) \\ &= \delta(A') + \delta(\alpha_h \to \beta_h) \\ &> \delta(A'') + \delta(a \to b) \\ &\geq edist_\delta(u,v) \end{aligned}$$

This is a contradiction. Hence A' is an optimal alignment of u' and v' and $edist_\delta(u,v) = \delta(A) = \delta(A') + \delta(a \to b) = edist_\delta(u',v') + \delta(a \to b)$.

Since all three cases (3a), (3b), and (3c) may occur and we want to minimize costs, we have to compute the minimum over all three cases. Altogether, we get the following system of recursive equations:

$$\begin{aligned} edist_\delta(\varepsilon,\varepsilon) &= 0 && \text{(case (1) and (2))} \\ edist_\delta(\varepsilon,v'b) &= edist_\delta(\varepsilon,v') + \delta(\varepsilon \to b) && \text{(case (1))} \\ edist_\delta(u'a,\varepsilon) &= edist_\delta(u',\varepsilon) + \delta(a \to \varepsilon) && \text{(case (2))} \\ edist_\delta(u'a,v'b) &= \min \left\{ \begin{array}{l} edist_\delta(u',v'b) + \delta(a \to \varepsilon) \\ edist_\delta(u'a,v') + \delta(\varepsilon \to b) \\ edist_\delta(u',v') + \delta(a \to b) \end{array} \right\} && \text{(case (3))} \end{aligned}$$

Of course, the direct implementation of $edist_\delta$ as a recursive function would be inefficient, since the same function calls appear in different contexts. Analyzing the recurrence reveals that $edist_\delta(u',v')$ is evaluated for all pairs of prefixes u' of u and v' of v. So the idea is to tabulate these intermediate results, instead of recomputing them each time they are needed. That is, we compute an $(m+1) \times (n+1)$ table E_δ defined for all $i \in [0,m]$ and $j \in [0,n]$ as follows:

$$E_\delta(i,j) = edist_\delta(u[1 \ldots i], v[1 \ldots j])$$

The table thus has $m+1$ rows and $n+1$ columns. We use i to refer to the row number of the table (running from 0 to m) and j to refer to the column number of the table (running from 0 to n). This notation will be used consistently from now on. We now derive the equations for $E_\delta(i,j)$:

- Let $i = j = 0$. Then

$$\begin{aligned} E_\delta(0,0) &= edist_\delta(u[1 \ldots 0], v[1 \ldots 0]) \\ &= edist_\delta(\varepsilon,\varepsilon) \\ &= 0 \end{aligned}$$

- Let $i = 0$ and $j > 0$. Then

$$\begin{aligned} E_\delta(0,j) &= edist_\delta(u[1\ldots 0], v[1\ldots j]) \\ &= edist_\delta(\varepsilon, v[1\ldots j]) \\ &= edist_\delta(\varepsilon, v[1\ldots j-1]v[j]) \\ &= edist_\delta(\varepsilon, v[1\ldots j-1]) + \delta(\varepsilon \to v[j]) \\ &= E_\delta(0,j-1) + \delta(\varepsilon \to v[j]) \end{aligned}$$

- Let $i > 0$ and $j = 0$. Then

$$\begin{aligned} E_\delta(i,0) &= edist_\delta(u[1\ldots i], v[1\ldots 0]) \\ &= edist_\delta(u[1\ldots i], \varepsilon) \\ &= edist_\delta(u[1\ldots i-1]u[i], \varepsilon) \\ &= edist_\delta(u[1\ldots i-1], \varepsilon) + \delta(u[i] \to \varepsilon) \\ &= E_\delta(i-1,0) + \delta(u[i] \to \varepsilon) \end{aligned}$$

- Let $i > 0$ and $j > 0$. Then

$$\begin{aligned} E_\delta(i,j) &= edist_\delta(u[1\ldots i], v[1\ldots j]) \\ &= edist_\delta(u[1\ldots i-1]u[i], v[1\ldots j-1]v[j]) \\ &= \min \left\{ \begin{array}{l} edist_\delta(u[1\ldots i-1], v[1\ldots j-1]v[j]) + \delta(u[i] \to \varepsilon) \\ edist_\delta(u[1\ldots i-1]u[i], v[1\ldots j-1]) + \delta(\varepsilon \to v[j]) \\ edist_\delta(u[1\ldots i-1], v[1\ldots j-1]) + \delta(u[i] \to v[j]) \end{array} \right\} \\ &= \min \left\{ \begin{array}{l} E_\delta(i-1,j) + \delta(u[i] \to \varepsilon) \\ E_\delta(i,j-1) + \delta(\varepsilon \to v[j]) \\ E_\delta(i-1,j-1) + \delta(u[i] \to v[j]) \end{array} \right\} \end{aligned}$$

Combining these equations into one, we obtain the following recurrences for E_δ:

$$E_\delta(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ E_\delta(0,j-1) + \delta(\varepsilon \to v[j]) & \text{if } i = 0 \text{ and } j > 0 \\ E_\delta(i-1,0) + \delta(u[i] \to \varepsilon) & \text{if } i > 0 \text{ and } j = 0 \\ \min \left\{ \begin{array}{l} E_\delta(i-1,j) + \delta(u[i] \to \varepsilon) \\ E_\delta(i,j-1) + \delta(\varepsilon \to v[j]) \\ E_\delta(i-1,j-1) + \delta(u[i] \to v[j]) \end{array} \right\} & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

By definition, $E_\delta(m,n)$ gives the edit distance of u and v. The values in E_δ are computed in topological order, i.e. consistent with the data dependencies. That is, if $E_\delta(i,j)$ depends on $E_\delta(i',j')$, then $E_\delta(i',j')$ must be computed before $E_\delta(i,j)$. Analyzing the dependencies of the previous recurrence, one sees that $E_\delta(i,j)$ depends on the previous value

Algorithm 1 The DP Algorithm for the Edit Distance

Input: sequences $u = u[1 \ldots m]$ and $v = v[1 \ldots n]$
cost function δ
Output: $edist_\delta(u, v)$

$E_\delta(0,0) \leftarrow 0$
for $i \leftarrow 1$ **to** m **do**
    $E_\delta(i,0) \leftarrow E_\delta(i-1,0) + \delta(u[i] \rightarrow \varepsilon)$
for $j \leftarrow 1$ **to** n **do**
    $E_\delta(0,j) \leftarrow E_\delta(0,j-1) + \delta(\varepsilon \rightarrow v[j])$
    for $i \leftarrow 1$ **to** m **do**

$$E_\delta(i,j) \leftarrow \min \left\{ \begin{array}{l} E_\delta(i-1,j) + \delta(u[i] \rightarrow \varepsilon) \\ E_\delta(i,j-1) + \delta(\varepsilon \rightarrow v[j]) \\ E_\delta(i-1,j-1) + \delta(u[i] \rightarrow v[j]) \end{array} \right\}$$

print $E_\delta(m,n)$

in the current column j and on two values in the previous column. Hence one can compute the table E_δ column by column, and each column from top to bottom. This evaluation order is consistent with the data dependencies.

Example 10 Let $u = bcacd$, $v = dbadad$, and assume that δ is the unit cost function. Then E_δ is as follows:

			d	b	a	d	a	d
	$E_\delta(i,j)$	0	1	2	3	4	5	6
	0	0	1	2	3	4	5	6
b	1	1	1	1	2	3	4	5
c	2	2	2	2	2	3	4	5
a	3	3	3	3	2	3	3	4
c	4	4	4	4	3	3	4	4
d	5	5	4	5	4	3	4	4

Hence the edit distance of u and v is 4. □

Each entry in E_δ is computed in constant time. This leads to an $O(m\,n)$ time complexity. Note that the values in each column only depend on the values of the current and the previous column. Hence, if we only want to compute the edit distance, then it suffices to store only two columns in each step of the algorithm. In fact, it is possible to arrange the computation in such a way, that the space for only one column is required, plus two extra scalars (integers or floats, depending on the cost function). As a consequence, only $O(\min\{m,n\})$ space is required. The corresponding algorithm is then also termed "distance-only algorithm" for computing the edit distance.

In molecular biology, the above algorithm is usually called "the dynamic programming algorithm". However, dynamic programming (DP, for short) is a general programming paradigm. A problem can be solved by DP, if the following holds:

- optimal solutions to the problem can be derived from optimal solutions to subproblems.
- the optimal solutions can efficiently be determined, if a table of solutions for increasing subproblems are computed.

To completely solve the edit distance problem, we also have to compute the optimal alignments. An optimal alignment is recovered by tracing back from the entry $E_\delta(m,n)$ to an entry in its three-way minimum that yielded it, determining which entry gave rise to that entry, and so on back to the entry $E_\delta(0,0)$. This requires saving the entire table, giving an algorithm that takes $O(m\,n)$ space. This backtracking algorithm can best be explained by giving a graph theoretic formulation of the problem.

Definition 8 The *edit graph* $G(u,v)$ of u and v is an edge labeled graph. The nodes are the pairs $(i,j) \in [0,m] \times [0,n]$. The edges are given as follows:

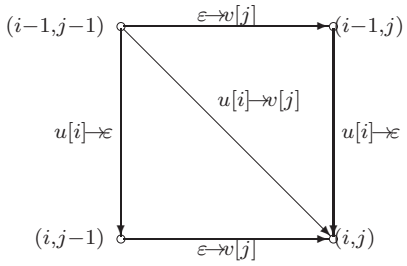
- For $1 \leq i \leq m, 0 \leq j \leq n$ there is a deletion edge
$$(i-1,j) \xrightarrow{u[i] \mapsto \varepsilon} (i,j)$$
- For $0 \leq i \leq m, 1 \leq j \leq n$ there is an insertion edge
$$(i,j-1) \xrightarrow{\varepsilon \mapsto v[j]} (i,j)$$
- For $1 \leq i \leq m, 1 \leq j \leq n$ there is a replacement edge
$$(i-1,j-1) \xrightarrow{u[i] \mapsto v[j]} (i,j)$$

□

This is illustrated in Figure 3.2.

The central feature of $G(u,v)$ is that each path from (i',j') to (i,j) is labeled by an alignment of $u[i'+1 \ldots i]$ and $v[j'+1 \ldots j]$, and a different path is labeled by a different alignment. An edge $(i',j') \xrightarrow{\alpha \mapsto \beta} (i,j)$ is *minimizing* if $E_\delta(i,j)$ equals $E_\delta(i',j') + \delta(\alpha \to \beta)$. A *minimizing path* is any path from $(0,0)$ to (m,n) that consists of minimizing edges only. In this framework, the edit distance problem means to enumerate the minimizing paths in $G(u,v)$. This is done by starting at node (m,n) and tracing the minimizing edges back to node $(0,0)$. The back tracing procedure can be organized in such a way that each optimal alignment A of u and v is computed in $O(|A|)$ time. To facilitate the backtracking, we store with each entry $E_\delta(i,j)$ three bits. Each of these bits tells us whether an incoming edge into (i,j) is minimizing. Thus we conclude:

Figure 3.2: A Part of the Edit Graph $G(u, v)$



Theorem 1 The edit distance problem for two sequences u of length m and v of length n can be solved in $O(m\,n + z)$ time, where z is the total length of all alignments of u and v. □

Example 11 Let $u = bcacd$ and $v = dbadad$. Suppose δ is the unit cost function. Then we have $edist_\delta(u, v) = 4$ and there are the following optimal alignments of u and v.

$-\,b\,c\,a\,c\,-\,d$ $d\,b\,-\,a\,d\,a\,d$	$b\,c\,a\,c\,-\,d$ $d\,b\,a\,d\,a\,d$	$-\,b\,c\,-\,a\,c\,d$ $d\,b\,a\,d\,a\,-\,d$	$-\,b\,-\,c\,a\,c\,d$ $d\,b\,a\,d\,a\,-\,d$	$-\,b\,c\,a\,-\,c\,d$ $d\,b\,-\,a\,d\,a\,d$	$b\,c\,a\,-\,c\,d$ $d\,b\,a\,d\,a\,d$	$-\,b\,c\,a\,c\,d$ $d\,b\,a\,d\,a\,d$

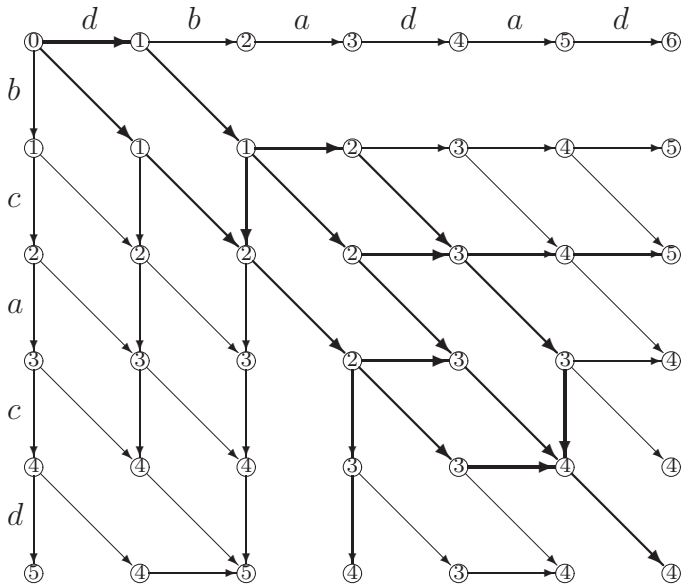
Figure 3.3 shows $G(u, v)$ with all minimizing edges. The minimizing paths are given by the thick edges. Each node is marked by the corresponding edit distance. It is straightforward to read the optimal alignments of u and v from the edit graph. □

The space requirement for the above procedure is $O(m\,n)$. Using a distance-only algorithm as a sub-procedure, there are divide and conquer algorithms that can determine each optimal alignment in $O(\min\{m, n\})$ space and $O(m\,n)$ time. These algorithms are very important, since space, not time, is often the limiting factor when computing optimal alignments between large sequences. We will consider these algorithms in Section 3.6.

3.3 Local Similarity

Up to this point we have focused on global comparison. That is, we have compared the complete sequence u with the complete sequence v. In biological sequences we often have long non-coding regions and small coding regions. Thus if two coding regions in two large sequences are similar, this does not imply that the sequences have a small edit distance, see Example 12. As a consequence, when comparing biological sequences it is sometimes important to perform local similarity comparisons: This means to find all pairs of substrings in u and v which are similar. It does not make sense to looks for pairs of substrings with some minimum distance, ε is a substring of any sequence and the pair $(\varepsilon, \varepsilon)$ has distance 0. To clarify the notion of sequence similarity, we introduce score functions.

Figure 3.3: The minimizing edges and paths in the edit graph $G(u, v)$ for $u = bcacd$ and $v = dbadad$. Each edge is a minimizing edge. The thick edges form the Minimizing Paths. Edge labels are not shown.



Example 12 The following global alignment shows that two sequences with similarities over the entire range of positions. Hence a global alignment is an appropriate means to represent the similarities.

```
--T--CC-C-AGT--TATGT-CAGGGGACACG--A-GCATGCAGA-AC
  |   || |   ||   | | | |||     || |   | |   |||||    |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG--TCAGAT--C
```

The middle line uses the symbol | to mark columns of identical characters.

Sometimes only segments of the sequences show similarities. Thus a global alignment does not make sense and one should look for local similarities, displayed in a local alignment:

```
CGTTTAGGCTCTTGCGCGTCCCAGTTAT-TCAGGGGACACGAGCATGCAGAGAC
                          ||||| | ||||
AATTGCCGCCGTCGTTTTCAGCAGTT-TGTCAGATCTTAGGTTCTTTAATTATATT
```

Definition 9 A *score function* σ assigns to each edit operation $\alpha \to \beta$ a *score* $\sigma(\alpha \to \beta) \in \mathbb{R}$. For each alignment $A = (\alpha_1 \to \beta_1, \ldots, \alpha_h \to \beta_h)$ we define the score $\sigma(A) = \sum_{i=1}^{h} \sigma(\alpha_i \to \beta_i)$. The similarity score of u' and v' is defined by

$$score_\sigma(u', v') = \max\{\sigma(A) \mid A \text{ is an alignment of } u' \text{ and } v'\}.\square$$

Figure 3.4: The BLOSUM62 similarity score matrix specifying a replacement score for each pair of amino acid

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

Table 3.4 shows the BLOSUM62 similarity matrix, which is widely used when comparing proteins. With some additional scores for insertions and deletions we would obtain a score function.

Definition 10 Let σ be a score function. We define

1. $loc_\sigma(u, v) = \max\{score_\sigma(u', v') \mid u' \text{ is substring of } u \text{ and } v' \text{ is substring of } v\}$

2. Let u' be a substring of u and v' be a substring of v such that

$$score_\sigma(u', v') = loc_\sigma(u, v)$$

   That is, $score(u', v')$ is the maximum score over all substrings of u and v. An alignment A of u' and v' satisfying $score_\sigma(u', v') = \sigma(A)$ is a *local optimal alignment* of u and v.

3. The *local optimal alignment problem* is to compute $loc_\sigma(u, v)$ and a local optimal alignment of u and v. □

A brute force solution to the local optimal alignment problem would be as follows:

compute for each pair (u', v') of substrings u' of u and v' of v the value $score_\sigma(u', v')$.

There are $O(n^2m^2)$ pairs (u', v') of substrings of u and v. Each computation of $score_\sigma(u', v')$ requires $O(mn)$ time. Thus, this method would require $O(n^3m^3)$ time, which is, of course, too expensive.

Now note that each substring u' of u is a suffix of a prefix of u and each substring v' of v is a suffix of a prefix of v. So the idea is to compute a matrix where each entry (i, j) contains the maximal score for all pairs of suffixes of prefixes ending at position i in u and position j in v. More precisely, we compute an $(m + 1) \times (n + 1)$-Matrix L_σ defined by

$$L_\sigma(i, j) = \max\{score_\sigma(x, y) \mid x \text{ is suffix of } u[1 \ldots i] \text{ and } y \text{ is suffix of } v[1 \ldots j]\}$$

Now we can conclude,

$$\begin{aligned} loc_\sigma(u, v) &= \max\{score_\sigma(u', v') \mid u' \text{ is substring of } u, v' \text{ is substring of } v\} \\ &= \max\{score_\sigma(x, y) \mid i \in [0, m], j \in [0, n], x \text{ is suffix of } u[1 \ldots i] \\ &\qquad\qquad y \text{ is suffix of } v[1 \ldots j]\} \\ &= \max\{\max\{score_\sigma(x, y) \mid x \text{ is suffix of } u[1 \ldots i] \\ &\qquad\qquad\qquad y \text{ is suffix of } v[1 \ldots j]\} \mid i \in [0, m], j \in [0, n]\} \\ &= \max\{L_\sigma(i, j) \mid i \in [0, m], j \in [0, n]\} \end{aligned}$$

That is, $loc_\sigma(u, v)$ can be computed by maximizing over all entries in table L_σ. Consider an edit graph representing all local alignments. Since we are interested in alignments of all pairs of substrings of u and v, we are interested in each path. The paths do not necessarily have to start at $(0, 0)$ or end at (m, n). Since a path can begin at any node, we have to allow the score 0 in any entry of the matrix. These considerations lead to the following result:

Theorem 2 Let σ be a score function satisfying

$$loc_\sigma(s, \varepsilon) = loc_\sigma(\varepsilon, s) = 0 \tag{3.3}$$

for any sequence $s \in \mathcal{A}^*$. Then the following holds:

- If $i = 0$ or $j = 0$, then $L_\sigma(i, j) = 0$.
- Otherwise,

$$L_\sigma(i, j) = \max \left\{ \begin{array}{l} 0 \\ L_\sigma(i - 1, j) + \sigma(u[i] \to \varepsilon) \\ L_\sigma(i, j - 1) + \sigma(\varepsilon \to v[j]) \\ L_\sigma(i - 1, j - 1) + \sigma(u[i] \to v[j]) \end{array} \right\}$$

Condition (3.3) is very important for the Theorem: prefixes with negative score are suppressed, and similar substrings occur as positive islands in a matrix dominated by 0-entries. In general, it is not easy to verify condition (3.3). However, one usually requires that σ is biased towards negative indel scores, that is, it holds: $\sigma(\alpha \to \beta) < 0$ for all insertions and deletions $\alpha \to \beta$. This condition then implies (3.3):

$$\begin{aligned} loc_\sigma(s, \varepsilon) &= \max\{score_\sigma(x, \varepsilon) \mid x \text{ is a substring of } s\} \\ &= \max\{score_\sigma(\varepsilon, \varepsilon)\} \cup \{score_\sigma(x, \varepsilon) \mid x \text{ is a substring of } s, x \neq \varepsilon\} \\ &= 0 \end{aligned}$$

Algorithm 2 The Smith-Waterman Algorithm

Input: sequences $u = u[1 \dots m]$ and $v = v[1 \dots n]$
score function σ satisfying (3.3)
Output: $loc_\sigma(u, v)$ and a local optimal alignment of u and v.

1. Compute Matrix L_σ according to Theorem 2.
2. Compute a maximal entry, say $L_\sigma(i, j)$, in L_σ.
3. Compute local optimal alignments by backtracking on a maximizing path starting at some entry (i', j') satisfying $L_\sigma(i', j') = 0$ and ending in (i, j).

One can similarly show $loc_\sigma(\varepsilon, s) = 0$.

Based on Theorem 2, we can derive Algorithm 2 which solves the local similarity search problem. The algorithm was first published in [SW81].

The Smith-Waterman Algorithm requires $O(mn)$ time and space.

Example 13 Consider the similarity score

$$\sigma(\alpha \to \beta) = \begin{cases} -1 & \text{if } \alpha = \varepsilon \text{ or } b = \varepsilon \\ -2 & \text{if } \alpha, \beta \in \mathcal{A}, \alpha \neq \beta \\ 2 & \text{if } \alpha, \beta \in \mathcal{A}, \alpha = \beta \end{cases}$$

and the sequences $u = xyaxbacsll$ and $v = pqraxabcstvq$. Then matrix L_σ is as follows:

		x	y	a	x	b	a	c	s	l	l
	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	0	0	0	0	0	0	0	0
q	0	0	0	0	0	0	0	0	0	0	0
r	0	0	**0**	0	0	0	0	0	0	0	0
a	0	0	0	**2**	1	0	2	1	0	0	0
x	0	2	1	1	**4**	**3**	2	1	0	0	0
a	0	1	0	3	3	2	**5**	4	3	2	1
b	0	0	0	2	2	5	**4**	3	2	1	0
c	0	0	0	1	1	4	3	**6**	5	4	3
s	0	0	0	0	0	3	2	5	**8**	7	6
t	0	0	0	0	0	2	1	4	7	6	5
v	0	0	0	0	0	1	0	3	6	5	4
q	0	0	0	0	0	0	0	2	5	4	3

The maximum value is 8. Tracing back the path along the bold face numbers gives a path representing the following local optimal alignment (with score 8)

$$\begin{array}{cccccccc} a & x & - & a & b & c & s \\ a & x & b & a & - & c & s \end{array}$$

Up until now we have seen how to compute optimal global alignments and optimal local alignments for a pair of sequences. In the former, both sequences are aligned completely, while in the latter, all substrings of both sequences are aligned to obtain an optimal alignment. As variations of global and local alignments, we present a solution to the approximate string searching problem. In the lecture notes for the module "Genome Informatics" we will also consider a variation computing overlaps between pairs of sequences.

3.4 The approximate string searching problem

Given a pattern $p \in \mathcal{A}^*$ of length m and an input string $t \in \mathcal{A}^*$ of length n, *the approximate string searching problem* consists of finding the positions in t where an approximate match ends. These positions are referred to as solutions of the approximate string searching problem. An *approximate match* is a substring w of t such that $edist_\delta(p, w) \leq k$, for a given threshold value $k \in \mathbb{R}^+$.

The approximate string searching problem is of special interest in biological sequence analysis. For instance, when searching a DNA database (the input string) for a pattern, a small but significant error must be allowed, to take into account experimental inaccuracies as well as small differences in DNA among individuals of the same or related species.

By a slight modification of the previous dynamic programming algorithms, one obtains a simple method (first published in [Sel80]) to solve the approximate string searching problem: Compute an $(m+1) \times (n+1)$-table D_δ such that for all $(i, j) \in [0, m] \times [0, n]$, the following holds:

$$D_\delta(i, j) = \min \{edist_\delta(p[1 \ldots i], w) \mid w \text{ is a suffix of } t[1 \ldots j]\}$$

By definition, there is an approximate match ending at position $j \in [0, n]$, if and only if $edist_\delta(p, w) \leq k$ for some suffix w of $t[1 \ldots j]$. Observe that the latter is equivalent to the condition $D_\delta(m, j) \leq k$. Thus, to solve the approximate string searching problem, one computes table D_δ and outputs all j satisfying $D_\delta(m, j) \leq k$. See Figure 3.5, for an example. As usual, the approximate matches w ending at position j (and an optimal alignment of p and w, if necessary) can be output, by backtracing from $D_\delta(m, j)$ to an entry $D_\delta(0, j')$, $j' \in [0, j-1]$ in the first row of table D_δ.

Recurrences for D_δ are easy to derive. Since we want to align p with all substrings of t, we have to allow that for each $j \in [0, n]$, the prefixes of $t[1 \ldots j]$ can be deleted at no cost. Thus, the only difference between table E_δ and D_δ is that the latter always has 0 in the first row:

$$D_\delta(i, j) = \begin{cases} 0 & \text{if } i = 0 \\ D_\delta(i-1, 0) + \delta(p[i] \to \varepsilon) & \text{if } i > 0 \text{ and } j = 0 \\ \min \left\{ \begin{array}{l} D_\delta(i-1, j) + \delta(p[i] \to \varepsilon) \\ D_\delta(i, j-1) + \delta(\varepsilon \to t[j]) \\ D_\delta(i-1, j-1) + \delta(p[i] \to t[j]) \end{array} \right\} & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

Obviously, each entry in table D_δ can be evaluated in constant time. Hence the algorithm runs in $O(mn)$ time. The space requirement is $O(mn)$ if we also want to compute approximate matches. If only the positions where an approximate match ends are to be found, then

						j					
				a	b	b	d	a	d	c	b	c
		D_δ	0	1	2	3	4	5	6	7	8	9
		0	0	0	0	0	0	0	0	0	0	0
	a	1	1	0	1	1	1	0	1	1	1	1
i	d	2	2	1	1	2	1	1	0	1	2	2
	b	3	3	2	1	1	2	2	1	1	1	2
	b	4	4	3	2	1	2	3	2	2	1	2
	c	5	5	4	3	2	2	3	3	2	2	1

Figure 3.5: Table D_δ for $p = adbbc$ and $t = abbdadcbc$. Let $k = 2$. Then in the last row we find values $\leq k$ for $j \in \{3, 4, 7, 8, 9\}$. Hence approximate matches end at these positions.

$O(m)$ space suffices. The sequence comparison methods described here always compute an $(m + 1) \times (n + 1)$-table in $O(mn)$ time, given sequences of length m and n, respectively. The local alignment method maximizes scores, while all other methods minimize distances. The main difference of the methods is the backtracing phase. This can be characterized by start entries and final entries in the different tables. The sought alignments always begin at start entries and they end at final entries, as depicted in a schematic way in Figure 3.6.

3.5 Variations of the Cost and Score Model

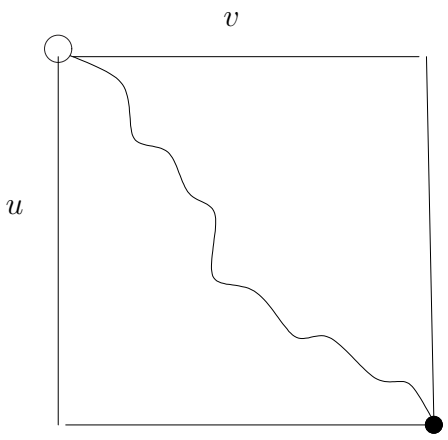
The usual evolutionary progression of biosequences involves individual point mutations, i.e. the insertion, deletion, or replacement of a single nucleotide or residue. Accordingly, the usual cost and score models assign costs and scores to individual edit operations. However, sometimes it is necessary to consider the deletion or insertion of entire segments as single units and to assign costs or scores to these. Thus we have to extend the alignment model by intervals of unaligned (i.e. inserted or deleted) characters, called *gaps* in the following. The notion of edit operations is generalized such that for each edit operation $\alpha \to \beta$ we have

$$(\alpha, \beta) \in (\mathcal{A} \times \mathcal{A}) \cup (\mathcal{A}^+ \times \{\varepsilon\}) \cup (\{\varepsilon\} \times \mathcal{A}^+)$$

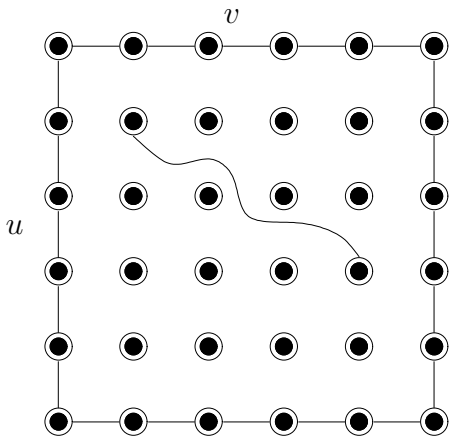
Figure 3.7 shows an example of an extended alignment with gaps. Consequently, we also have to extend the cost model to the new kinds of edit operations. In the simplest model, the *affine gap model*, each gap gets the sum of the costs for each symbol involved in the gap plus a *gap initiation cost* for the introduction of the gap. That is, for each $w \in \mathcal{A}^+$, define

$$\delta_{\text{affine}}(w \to \varepsilon) = g + \sum_{i=1}^{|w|} \delta(w[i] \to \varepsilon) \text{ and } \delta_{\text{affine}}(\varepsilon \to w) = g + \sum_{j=1}^{|w|} \delta(\varepsilon \to w[j])$$

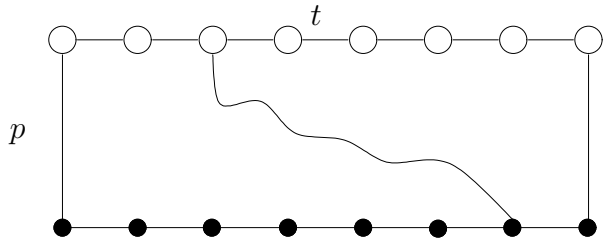
Figure 3.6: Backtracing strategies for different alignment problems in an edit graph. Each backtracing strategy is characterized by start entries (i.e. entries where the backtracing ends or an optimal alignment starts) and by final entries (i.e. entries where the backtracing starts or an alignment ends). Start entries are depicted as circles, while final entries are depicted as bullets.



global alignment (table E_δ)

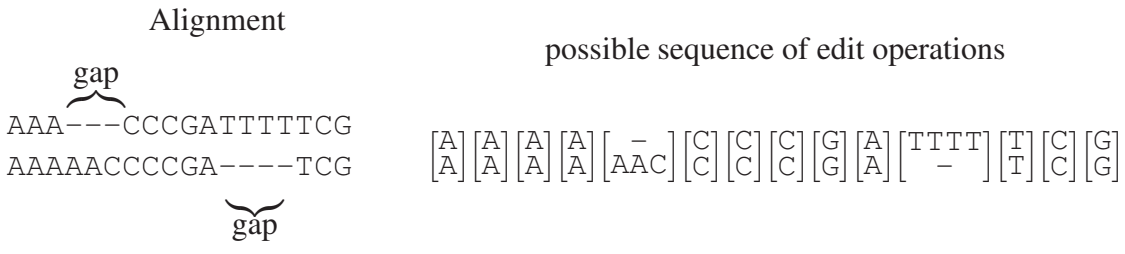


local alignment (table L_σ)



approximate string searching (table D_δ)

Figure 3.7: An alignment and the possible sequence of edit operations according to the extended alignment model.



where g is the gap initiation cost, $\delta(w[i] \to \varepsilon)$ is the cost of deleting $w[i]$ and $\delta(\varepsilon \to w[j])$ is the cost of inserting $w[j]$. Example 14 gives an example of the effect of an affine gap cost model.

Example 14 This example was adapted from `http://homepage.usask.ca/~ctl271/857/affine_gap_penalties.shtml`.

Consider the following alignment whose cost are determined as follows: cost 0 for a match, 1 for a mismatch, and 2 for an indel. The costs are given in the first line below the alignment.

```
G A A T T C C G T T A
|   | |   |   |     |
G G A T - C - G - - A

0 1 0 0 2 0 2 0 2 2 0 = cost 9 (linear gap costs)
0 1 0 0 2 0 2 0 2 1 0 = cost 8 (affine gap costs)
```

We can also make the first gap larger and align the second `C` in the first sequence with the first `C` in the second sequence to obtain the following alignment which also has cost 9.

```
G A A T T C C G T T A
|   | |     | |     |
G G A T - - C G - - A

0 1 0 0 2 2 0 0 2 2 0 = cost 9 (linear gap costs)
0 1 0 0 2 1 0 0 2 1 0 = cost 7 (affine gap costs)
```

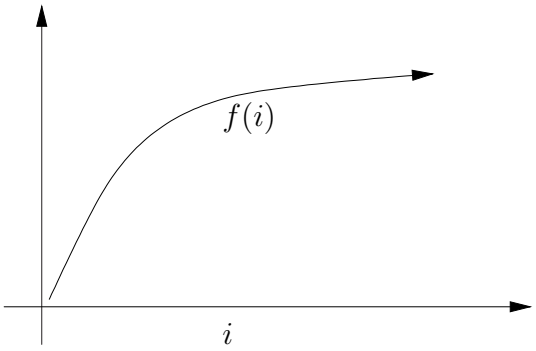
Now suppose an affine gap model with cost 0 for a match, 1 for a mismatch, 2 for a gap initiation and cost 1 for a gap extension. Then the first alignment gets score 8 (see second line below the alignment), due to the fact that there is one gap extension, reducing the cost by 1. In the second alignment there are two gap extensions, so the costs are reduced by 2 relative to the linear gap model, so that the alignment gets cost 7. Indeed the second alignment with smaller affine gap cost is the preferred alignment.

This example demonstrates the difference using affine gap costs rather than linear gap costs. Affine gap costs provide incentive for the alignment algorithm to keep sequence together where possible rather than inserting many small gaps. Generally this is the more desirable behavior, and so the use of affine gap costs makes sense.

A different scheme is to define costs where each additional symbol leads to smaller additional costs for the gap. This *concave model* is based on a function f_{concave} such that $f_{\text{concave}}(i+1) - f_{\text{concave}}(i) \leq f_{\text{concave}}(i) - f_{\text{concave}}(i-1)$ for all $i \geq 2$ (see Figure 3.8). For each $w \in \mathcal{A}^+$, it assigns costs as follows:

$$\delta_{\text{concave}}(w \to \varepsilon) = \delta_{\text{concave}}(\varepsilon \to w) = f_{\text{concave}}(|w|)$$

Figure 3.8: A concave function



The most general cost model assigns costs by an arbitrary computable function f^* on the length of the gap. That is, for each $w \in \mathcal{A}^+$, define

$$\delta^*(w \to \varepsilon) = \delta^*(\varepsilon \to w) = f^*(|w|)$$

The costs for an alignment A with respect to the different cost models is defined as the sum of the costs of the generalized edit operations, the alignment consists of. It is denoted by $\delta_{\text{affine}}(A)$, $\delta_{\text{concave}}(A)$, and $\delta^*(A)$, respectively.

In the following, we restrict our attention to the affine gap model and the general gap model. Algorithms for the latter also solve the concave model. There are more efficient algorithms for the concave model, but we will not discuss them here.

3.5.1 General Gap Costs

We start with the general gap cost model. The idea is to extend the edit graph $G(u, v)$ by additional edges, to accommodate all kinds of gaps. This leads to the Needleman and Wunsch Algorithm [NW70]. Let $i \in [0, m]$ and $i' \in [0, i-1]$. A gap $u[i'+1 \ldots i] \to \varepsilon$ is represented by an edge from (i', j) to (i, j) for some $j \in [0, n]$. Thus, for each $(i, j) \in [0, m] \times [0, n]$, the edit graph has to be extended by additional edges

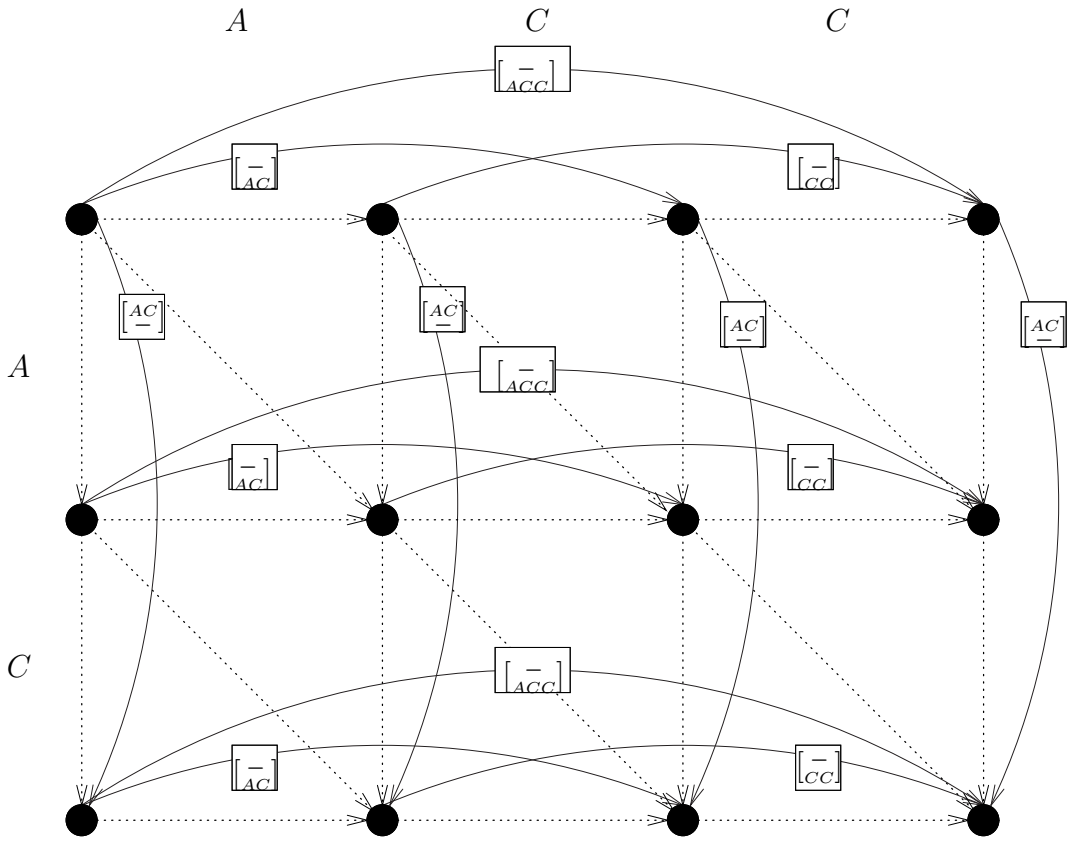
$$(i', j) \xrightarrow{u[i'+1 \ldots i] \to \varepsilon} (i, j)$$

for all $i' \in [0, i-1]$ and edges

$$(i, j') \xrightarrow{\varepsilon \to v[j'+1 \ldots j]} (i, j)$$

for all $j' \in [0, j-1]$. Thus, each node in the edit graph has on average $(n/2-1)+(m/2-1)$ additional edges, see also Figure 3.9. That is, there are $O(m+n)$ edges into each node and the total number of edges becomes $O(mn(m+n))$.

Figure 3.9: The General-Gap Edit Graph



To compute the edit distance according to the general gap cost model, we compute an $(m+1)\times(n+1)$-table GGC_{δ,f^*} such that $GGC_{\delta,f^*}(i,j)$ is the generalized edit distance of $u[1\ldots i]$ and $v[1\ldots j]$ with respect to the cost functions δ and f^*. Here δ is a cost function assigning costs to replacements only. From the considerations above, one easily derives the following recurrences:

$$GGC_{\delta,f^*}(i,j) = \min\begin{Bmatrix} 0 & \text{if } i=0 \text{ and } j=0 \\ \min\{GGC_{\delta,f^*}(i',j)+f^*(i-i') \mid i'\in[0,i-1]\} & \text{if } i>0 \\ \min\{GGC_{\delta,f^*}(i,j')+f^*(j-j') \mid j'\in[0,j-1]\} & \text{if } j>0 \\ GGC_{\delta,f^*}(i-1,j-1)+\delta(u[i]\to v[j]) & \text{if } i>0 \text{ and } j>0 \end{Bmatrix}$$

Note the compact description of the four different cases. The conditions about i and j determine whether the value to the left is included in the minimization. It is obvious that each entry in table GGC_{δ,f^*} is computed in $O(m+n)$ time. Thus we obtain an algorithm which runs in $O(mn(m+n))$ time.

3.5.2 Affine Gap Costs

Affine gap costs are a special case of the general gap cost model. However, they can be handled more efficiently. The idea, first presented in [Got82], is to extend the edit graph to an affine-cost edit graph to recognize which edge begins a gap and which edge extends a gap. Then we can assign appropriate costs to the two kinds of edges. That is, if an edge starts a gap, then we add the gap start costs g. Technically, we split each node (i,j) into three different nodes $(i,j,\mathtt{R})$, $(i,j,\mathtt{D})$, and $(i,j,\mathtt{I})$. The type of the node (R, D, and I) specifies that every path in the edit graph ending at that node, corresponds to an alignment that ends with the corresponding (single symbol) edit operation, that is:

- A replacement edge $(i-1,j-1)\xrightarrow{u[i]\to v[j]}(i,j)$ leads to three edges

  $$(i-1,j-1,x)\xrightarrow{u[i]\to v[j]}(i,j,\mathtt{R})$$

  for all $x\in\{\mathtt{R},\mathtt{D},\mathtt{I}\}$, in the affine-cost edit graph.

- A deletion edge $(i-1,j)\xrightarrow{u[i]\to\varepsilon}(i,j)$ leads to three edges

  $$(i-1,j,x)\xrightarrow{u[i]\to\varepsilon}(i,j,\mathtt{D})$$

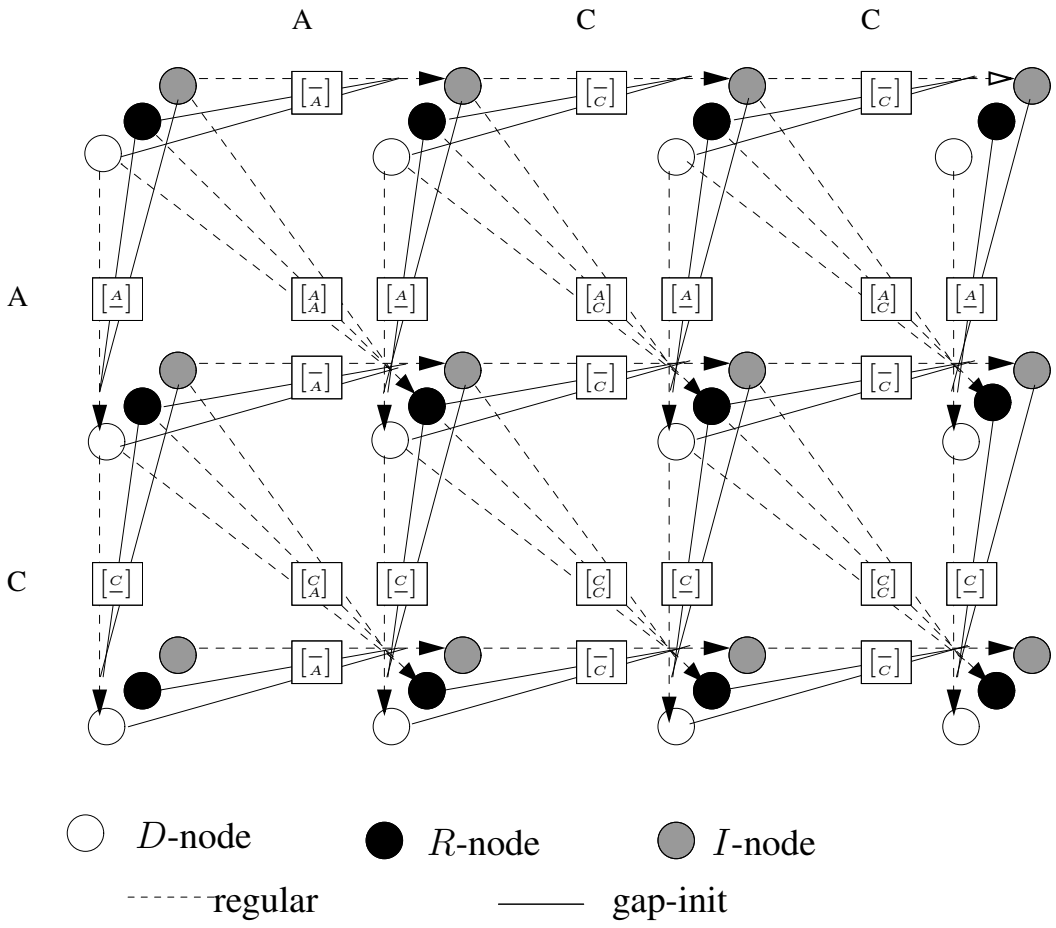
  for all $x\in\{\mathtt{R},\mathtt{D},\mathtt{I}\}$, in the affine-cost edit graph.

- An insertion edge $(i,j-1)\xrightarrow{\varepsilon\to v[j]}(i,j)$ leads to three edges

  $$(i,j-1,x)\xrightarrow{\varepsilon\to v[j]}(i,j,\mathtt{I}),$$

  for all $x\in\{\mathtt{R},\mathtt{D},\mathtt{I}\}$, in the affine-cost edit graph.

Figure 3.10: The Affine-Cost Edit Graph



See Figure 3.10 for an affine-cost edit graph.

The following table shows how to assign costs to the different kinds of edges in the affine-cost edit graph.

kind of edge	cost assignment
$(i-1, j-1, x) \xrightarrow{u[i] \to v[j]} (i, j, \mathtt{R})$	replacement, assign $\delta(u[i] \to v[j])$
$(i-1, j, \mathtt{D}) \xrightarrow{u[i] \to \varepsilon} (i, j, \mathtt{D})$	gap extension, assign $\delta(u[i] \to \varepsilon)$
$(i, j-1, \mathtt{I}) \xrightarrow{\varepsilon \to v[j]} (i, j, \mathtt{I})$	gap extension, assign $\delta(\varepsilon \to v[j])$
$(i-1, j, \mathtt{R}) \xrightarrow{u[i] \to \varepsilon} (i, j, \mathtt{D})$	gap start, assign $g + \delta(u[i] \to \varepsilon)$
$(i-1, j, \mathtt{I}) \xrightarrow{u[i] \to \varepsilon} (i, j, \mathtt{D})$	gap start, assign $g + \delta(u[i] \to \varepsilon)$
$(i, j-1, \mathtt{R}) \xrightarrow{\varepsilon \to v[j]} (i, j, \mathtt{I})$	gap start, assign $g + \delta(\varepsilon \to v[j])$
$(i, j-1, \mathtt{D}) \xrightarrow{\varepsilon \to v[j]} (i, j, \mathtt{I})$	gap start, assign $g + \delta(\varepsilon \to v[j])$

Node splitting does not affect the properties of the edit graph, i.e. a path from node (i', j', x) to (i, j, y) in the affine-cost edit graph corresponds to an alignment of $u[i'+1 \ldots i]$ and $v[j'+1 \ldots j]$, that ends with an edit operation according to the value y. The only difference is that the gaps in the alignments have costs according to the affine gap cost model. To compute optimal global alignments, we use $(0, 0, \mathtt{R})$, $(0, 0, \mathtt{D})$, and $(0, 0, \mathtt{I})$ as start entries and $(m, n, \mathtt{R})$, $(m, n, \mathtt{D})$, and $(m, n, \mathtt{I})$ as final entries. According to the considerations above, the edit graph has three times as much edges and nodes as before. But there are still $O(mn)$ nodes and edges. To compute the optimal alignment, one fills an $(m+1) \times (n+1) \times \{\mathtt{R}, \mathtt{D}, \mathtt{I}\}$-table A_{affine} such that the following holds:

- If $i = j = 0$, then $A_{\text{affine}}(i, j, \mathtt{R}) = 0$ and $A_{\text{affine}}(i, j, y) = g$ for $y \in \{\mathtt{D}, \mathtt{I}\}$.
- If $i > 0$ or $j > 0$, then

$$A_{\text{affine}}(i, j, y) = \min\ (\{\infty\} \cup \{\delta_{\text{affine}}(A) \mid A \text{ is an alignment of } u[1 \ldots i] \text{ and } v[1 \ldots j] \text{ and } A \text{ ends with an edit operation according to the value of } y\})$$

It is not difficult to derive the following recurrences for A_{affine}. We can restrict to the case that $i > 0$ or $j > 0$.

$$A_{\text{affine}}(i, j, \mathtt{R}) = \begin{cases} \infty & \text{if } i = 0 \text{ or } j = 0 \\ \min\{A_{\text{affine}}(i-1, j-1, y) + \delta(u[i] \to v[j]) \mid y \in \{\mathtt{R}, \mathtt{D}, \mathtt{I}\}\} & \text{otherwise} \end{cases}$$

$$A_{\text{affine}}(i, j, \mathtt{D}) = \begin{cases} \infty & \text{if } i = 0 \\ \min\ (\{A_{\text{affine}}(i-1, j, \mathtt{D}) + \delta(u[i] \to \varepsilon)\} \cup & \text{otherwise} \\ \quad \{A_{\text{affine}}(i-1, j, x) + g + \delta(u[i] \to \varepsilon) \mid x \in \{\mathtt{R}, \mathtt{I}\}\}) & \end{cases}$$

$$A_{\text{affine}}(i, j, \mathtt{I}) = \begin{cases} \infty & \text{if } j = 0 \\ \min\ (\{A_{\text{affine}}(i, j-1, \mathtt{I}) + \delta(\varepsilon \to v[j])\} \cup & \text{otherwise} \\ \quad \{A_{\text{affine}}(i, j-1, x) + g + \delta(\varepsilon \to v[j]) \mid x \in \{\mathtt{R}, \mathtt{D}\}\}) & \end{cases}$$

The sought cost of an optimal alignment according to the affine gap cost model is

$$\min\{A_{\text{affine}}(m, n, x) \mid x \in \{\mathtt{R}, \mathtt{D}, \mathtt{I}\}\}$$

Obviously, each entry in table A_{affine} can be evaluated in constant time. Thus the algorithm runs in $O(mn)$ time and space.

3.6 A Linear Space Alignment Algorithm

The classic algorithm to solve the edit distance problem for two sequences u and v of length m and n takes $O(mn)$ time and space. This is because it needs to store an $(m+1)\times(n+1)$-Matrix E_δ from which the alignments are computed by a backtracing procedure. While it is difficult to generally improve the time taken by the basic algorithm, in this section, we will derive an algorithm that only takes $O(\min\{m,n\})$ space to compute a single optimal alignment of u and v. In most cases, it suffices to compute only a single alignment, and so the linear space algorithm is very important in practice.

Recall that matrix E_δ satisfies

$$E_\delta(i,j) = edist_\delta(u[1\ldots i], v[1\ldots j])$$

for all $i \in [0,m]$ and $j \in [0,n]$.

It can be computed according to the recurrence of Figure 3.11

For each $j \in [0,n]$, let

$$E^j_{\text{col}} = E_\delta(0,j), \ldots, E_\delta(m,j)$$

denote the jth column of table E_δ and note that this is a vector of $m+1$ values. We will later refer to such columns as *distance columns*. As already observed earlier, if $j > 0$, then E^j_{col} can be computed from E^{j-1}_{col}, the sequence u, and the current character $v[j]$. This follows from a simple analysis of the data dependencies of the fundamental recurrences for E_δ. It then follows that one can compute the sequence of column vectors

$$E^0_{\text{col}}, E^1_{\text{col}}, \ldots, E^n_{\text{col}}$$

such that only two columns, the previous and the one being computed, need be extant at any time. This distance-only algorithm can compute the cost $E^n_{\text{col}}(m)$ of an optimal alignment in only $O(m)$ space. Realizing that one is free to compute a sequence of rows instead of columns, this further improves to $O(\min\{m,n\})$ space.

One may even non-asymptotically improve the computation of successive columns, so that only *one* $(m+1)$-element vector and two additional scalars are required to overwrite the new column values into the vector holding the previous colum values. We leave it as an exercise to develop such a function *nextEDtabcolumn* which takes three parameters: the vectors storing the previous column, the current character b, and the sequence u. This function is complemented by a function *firstEDtabcolumn* which computes E^0_{col}, given u.

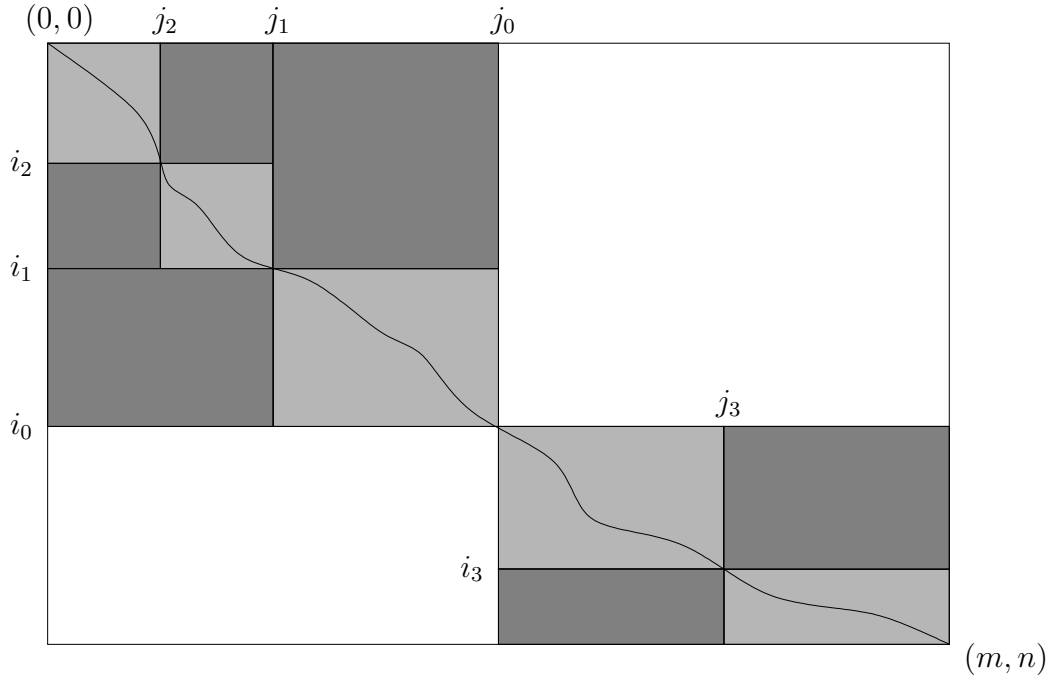
Now we turn to the delivery of an optimal alignment and describe the algorithm of [PAD99]. Some parts of the presentation follow [Mye98].

One immediately realizes that the distance-only algorithm has thrown away all the information needed for the traceback approach, as it only has available the last column of the table upon completion. The idea for the linear-space optimal alignment algorithm is to apply a divide-and-conquer strategy which works as follows: first find a node (i_0, j_0) that is on the middle of a minimizing path from node $(0,0)$ to node (m,n) in the edit-graph $G_\delta(u,v)$. Then recursively apply the same method to find a minimizing path from $(0,0)$ to (i_0,j_0) and a minimizing path from (i_0,j_0) to (m,n). This strategy is depicted in Figure 3.12.

Figure 3.11: The recurrence for table E_δ.

$$E_\delta(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ E_\delta(i-1,0) + \delta(u[i] \to \varepsilon) & \text{if } i > 0 \text{ and } j = 0 \\ E_\delta(0,j-1) + \delta(\varepsilon \to v[j]) & \text{if } i = 0 \text{ and } j > 0 \\ \min \left\{ \begin{array}{l} E_\delta(i-1,j) + \delta(u[i] \to \varepsilon) \\ E_\delta(i,j-1) + \delta(\varepsilon \to v[j]) \\ E_\delta(i-1,j-1) + \delta(u[i] \to v[j]) \end{array} \right\} & \text{if } i > 0 \text{ and } j > 0 \end{cases}$$

Figure 3.12: The Divide and Conquer Strategy of the Linear Space Alignment Algorithm



We say that a node $(\_, j_0)$ in the edit-graph $G_\delta(u, v)$ is on the middle of a path from $(0, 0)$ to (m, n) if and only if $j_0 = \lfloor n/2 \rfloor$. The divide-and-conquer strategy then becomes to find an index $i_0 \in [0, m]$ such that (i_0, j_0) is on a minimizing path from $(0, 0)$ to (m, n). We obtain i_0 by computing an additional $(m + 1) \times (n + 1)$-table R_{j_0} storing row indices. The columns of this table are referred to as *rowindex-columns*. For each $j \in [0, j_0 - 1]$ and each $i \in [0, m]$, $R_{j_0}(i, j)$ is undefined. For each $j \in [j_0, n]$ and each $i \in [0, m]$, $R_{j_0}(i, j)$ is defined and the following property holds: if $R_{j_0}(i, j) = i_0$, then there is a minimizing path from node (i_0, j_0) to node (i, j). It is easy to see that the recurrences of Figure 3.13 holds for R_{j_0}. The computation of $R_{j_0}(i, j)$ for all $j \in [j_0, n]$ follows the same scheme as the computation of table E_δ. In particular, one computes the jth column

$$R^j_{j_0} = R_{j_0}(0, j), \ldots, R_{j_0}(m, j)$$

along with E^j_{col}. With the same argumentation as above, one shows that this only requires one $(m + 1)$-element vector and two additional scalar values. It will be left as an exercise to derive functions *firstEDtabRtabcolumn* and *nextEDtabRtabcolumn*. The first function computes E^0_{col} and $R^0_{j_0}$ given u. The second function computes E^j_{col} and $R^j_{j_0}$ from E^{j-1}_{col} and $R^{j-1}_{j_0}$ given u, v, j, and j_0. Combining these functions one obtains a function $evaluateallcolumns$ which computes the final pair of columns of tables E_δ and R_{j_0}, given u, v, and a middle column index j_0

Suppose that we have determined E^n_{col} and $R^n_{j_0}$ as described above, where $j_0 = \lfloor n/2 \rfloor$. Then we know that $edist_\delta(u, v) = E^n_{\text{col}}(m)$ and (i_0, j_0) is on a minimizing path from $(0, 0)$ to (m, n), where $i_0 := R^n_{j_0}(m)$. We store i_0 at index j_0 of a table C which is indexed from 0 to n. C is the table of *crosspoints*. Each step of the linear space algorithm computes some value in table C by evaluating a distance column and a row-index column from the given substrings of u and v to align. Table C stores the crosspoints encoding an optimal alignment. In particular, $C(j) = i$ if and only if (i, j) is on a minimizing path from $(0, 0)$ to (m, n). Note that $C(0)) = 0$ and $C(n) = m$.

Suppose, that the two substrings of u and v to align, start at positions $i \in [0, m]$ and $j \in [0, n]$ and they are of length p and q. Algorithm 3 gives a recursive procedure $evaluatecrosspoints$, which employs the divide-and-conquer strategy to compute the crosspoints. The initial call to this procedure is

$$evaluatecrosspoints(C, 1, 1, m, n)$$

Note that a crosspoint is not computed if $n < 2$. In this case, v is of length 0 or 1 and it is trivial to compute an optimal alignment of u and v in linear space. Note that all recursive calls of $evaluatecrosspoints$ use the same globally-declared vectors E and R to compute the distance and row columns, and in turn the value i_0. This is correct, since once i_0 is determined, the values stored in E and R are no longer needed. This is essential to guarantee that the procedure only consumes $O(m)$ space.

We now analyze the space behavior of the presented algorithm. We previously had pointed out that all calls to the procedure $evaluatecrosspoints$ use the same two $(m + 1)$-element vectors. Additionally, $O(n)$ space for the table C is needed. Apart from that, only

a constant number of local variables in each call is required. Since the maximal depth of the recursion is $\log n$, the space requirement is $O(m + n + \log n)$.

To analyze the run time, suppose that the procedure requires $T(m, n)$ time for inputs of length m and n, respectively. The call to the procedure *evaluateallcolumns* requires $O(mn)$ time. Hence there is a constant h such that the dynamic programming (without the recursive calls) require hmn time. The recursive calls then solve subproblems of size $m_1 \times \lfloor n/2 \rfloor$ and $m_2 \times \lceil n/2 \rceil$ for some $m_1 > 0$, $m_2 > 0$ such that $m_1 + m_2 = m$. Since we do not know m_1 or m_2, we maximize the sum $T(m_1, \lfloor n/2 \rfloor) + T(m - m_1, \lceil n/2 \rceil)$ for all $m_1 \in [1, m]$. We thus obtain the following inequality:

$$T(m, n) \leq hmn + \max\{T(m_1, \lfloor n/2 \rfloor) + T(m - m_1, \lceil n/2 \rceil) \mid m_1 \in [1, m]\}$$

An easy exercise in induction shows that $T(m, n) \leq 2hnm$ for all m and n. Thus the improvement to linear space does not incur an asymptotic increase in the amount of time, as the complexity remains $O(mn)$. A simple approximation shows that the algorithm computes

$$mn + \frac{1}{2}mn + \frac{1}{4}mn + \ldots\ldots \approx 2mn$$

matrix entries over the whole computation. Empirical experience shows that an implementation of the linear space algorithm takes about twice as much time as the basic algorithm.

Algorithm 3 A recursive procedure to compute crosspoints. *evaluateallcolumns* computes the distance- and rowindex-columns for the substrings $u[i \ldots i+p-1]$ and $v[j \ldots j+q-1]$ given j_0. The latter is relative to the start index j, i.e. the implicit DP-tables are divided into two halves at column $j + j_0$. To store the columns, *evaluateallcolumns* uses the vectors E and R.

Input: $i \in [1, m], j \in [1, n], p \in [0, m - i + 1], q \in [0, n - j + 1]$, and uninitialized $(m + 1)$-element vectors E and R to store distance and row-index columns

Output: table C of crosspoints

1: **procedure** $evaluatecrosspoints(C, i, j, p, q)$
2: **if** $q \geq 2$ **then**
3: $j_0 := \lfloor q/2 \rfloor$
4: $evaluateallcolumns(E, R, j_0, u[i \ldots i + p - 1], v[j \ldots j + q - 1])$
5: $i_0 := R[p]$
6: $C(j + j_0 - 1) := i + i_0 - 1$
7: $evaluatecrosspoints(C, i, j, i_0, j_0)$
8: $evaluatecrosspoints(C, i + i_0, j + j_0, p - i_0, q - j_0)$
9: **end if**
10: create table C of $n + 1$ entries and set $C(0) := 0$ and $C(n) := m$
11: $evaluatecrosspoints(C, 1, 1, m, n)$

3.7 The Maximal Matches Model

The idea of this model, first described in [EH88] is to measure the distance between strings in terms of common substrings. Strings are considered similar if they have long common substrings, independent of where they occur. Technically, one counts the minimum number of occurrences of characters in one sequence such that if these characters are "crossed out", the remaining substrings are all substrings of the other sequence. The key to the model is the notion of partition. Recall that u and v are strings of length m and n, respectively.

Definition 11 A *partition* of v with respect to u is a sequence $(w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ of substrings $w_1, \ldots, w_r, w_{r+1}$ of u and characters $c_1, \ldots, c_r$ such that $v = w_1 c_1 \ldots w_r c_r w_{r+1}$. Let $\Psi = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ be a partition of v with respect to u. $w_1, \ldots, w_r, w_{r+1}$ are the *submatches* in Ψ. $c_1, \ldots, c_r$ are the *marked characters* in Ψ. The size of Ψ, denoted by $|\Psi|$, is r. *mmdist*(v, u) is the size of any minimal partition of v with respect to u. We call *mmdist*(v, u) *maximal matches distance* of v and u. □

Example 15 Let $v = cbaabdcb$ and $u = abcba$. $\Psi_1 = (cba, a, b, d, cb)$ is a partition of v with respect to u, since cba, b, and cb are substrings of u. $\Psi_2 = (cb, a, ab, d, cb)$ is a partition of v with respect to u, since cb and ab are substrings of u. It is clear that Ψ_1 and Ψ_2 are of minimal size. Hence, *mmdist*$(v, u) = 2$. □

There are two canonical partitions.

Definition 12 Let $\Psi = (w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ be a partition of v with respect to u. If for all $h \in [1, r]$, $w_h c_h$ is not a substring of u, then Ψ is the *left-to-right partition* of v with respect to u. If for all $h \in [1, r]$, $c_h w_{h+1}$ is not a substring of u, then Ψ is the *right-to-left partition* of v with respect to u. The left-to-right partition of v with respect to u is denoted by $\Psi_{lr}(v, u)$. The right-to-left partition of v with respect to u is denoted by $\Psi_{rl}(v, u)$. □

Example 16 For the strings $v = cbaabdcb$ and $u = abcba$ of Example 15 we have $\Psi_{lr}(v, u) = \Psi_1$ and $\Psi_{rl}(v, u) = \Psi_2$. □

One can show that $\Psi_{lr}(v, u)$ and $\Psi_{rl}(v, u)$ are of minimal size. Hence, we can conclude $|\Psi_{lr}(v, u)| =$ *mmdist*$(v, u) = |\Psi_{rl}(v, u)|$. This property leads to a simple algorithm for calculating the maximal matches distance. The partition $\Psi_{lr}(v, u)$ can be computed by scanning the characters of v from left to right, until a prefix wc of v is found such that w is a substring of u, but wc is not. w is the first submatch and c is the first marked character in $\Psi_{lr}(v, u)$. The remaining submatches and marked characters are obtained by repeating the process on the remaining suffix of v, until all of the characters of v have been scanned. Using the suffix tree of u, denoted by $ST(u)$ (see course "Genome Informatics", next Summersemester), the longest prefix w of v that is a substring of u, can be computed in $O(|\mathcal{A}| \cdot |w|)$ time. This gives an algorithm to calculate *mmdist*(v, u) in $O(|\mathcal{A}| \cdot (m + n))$ time and $O(m)$ space.

$\Psi_{rl}(v, u)$ can be computed in a similar way by scanning v from right to left. However, one has to be careful since the reversed scanning direction means to compute the longest

prefix of v^{-1} that occurs as substring of u^{-1}. This can, of course, be accomplished by using $ST(u^{-1})$ instead of $ST(u)$.

It is easily verified that *mmdist*$(u, v) = 1$ and *mmdist*$(v, u) = 2$ if v and u are as in Example 15. Hence, *mmdist* is not a metric on $\mathcal{A}^*$. However, one can obtain a metric as follows:

Theorem 3 Let $mmm(u, v) = \log_2((\textit{mmdist}(u, v) + 1) \cdot (\textit{mmdist}(v, u) + 1))$. mmm is a metric on $\mathcal{A}^*$. □

From the above it is clear that $mmm(u, v)$ can be computed in $O(|\mathcal{A}|\cdot(m+n))$ steps and $O(\max\{m, n\})$ space. Next we study the relation of the maximal matches distance and the unit edit distance. We first show an important relation of alignments and partitions.

Observation 4 Let A be an alignment of v and u. Then there is an $r \in [0, \delta(A)]$, and a partition $(w_1, c_1, \ldots, w_r, c_r, w_{r+1})$ of v with respect to u such that w_1 is a prefix and w_{r+1} is a suffix of u.

Proof: By structural induction on A. If A is the empty alignment, then $\delta(A) = 0$, $v = u = \varepsilon$, and the statement holds with $r = 0$ and $w_1 = \varepsilon$. If A is not the empty alignment, then A is of the form $(A', \beta \to \alpha)$ where A' is an alignment of some strings v' and u' and $\beta \to \alpha$ is an edit operation. Obviously, $v = v'\beta$ and $u = u'\alpha$. Assume the statement holds for A'. That is, there is an $r' \in [0, \delta(A')]$ and a partition $(w_1, c_1, \ldots w_{r'}, c_{r'}, w_{r'+1})$ of v' with respect to u' such that w_1 is a prefix and $w_{r'+1}$ is a suffix of u'. First note that w_1 is a prefix of u since it is prefix of u'. There are three cases to consider:

- If $\beta = \varepsilon$, then $\alpha \neq \varepsilon$ and $\delta(A) = \delta(A') + 1$. Hence, $v = v'\beta = w_1c_1 \ldots w_{r'}c_{r'}w_{r'+1}$. If $w_{r'+1}$ is the empty string, then it is a suffix of $u = u'\alpha$. If $w_{r'+1} = wc$ for some string w and some character c, then $v'\beta = w_1c_1 \ldots w_{r'}c_{r'}wcw'$ where $w' = \varepsilon$ is a suffix of $u = u'\alpha$. Thus, the statement holds with $r = r' + 1 \leq \delta(A)$.

- If $\beta \neq \varepsilon$ and $\alpha \neq \beta$, then $\delta(A) = \delta(A')+1$. Hence, $v = v'\beta = w_1c_1 \ldots w_{r'}c_{r'}w_{r'+1}\beta w$ where $w = \varepsilon$ is a suffix of $u = u'\alpha$. Thus, the statement holds with $r = r' + 1 \leq \delta(A)$.

- If $\beta \neq \varepsilon$ and $\alpha = \beta$, then $\delta(A) = \delta(A')$. Let $w = w_{r'+1}\beta$. Then $v = v'\beta = w_1c_1 \ldots w_{r'}c_{r'}w$, and w is a suffix of $u = u'\alpha$ since $w_{r'+1}$ is a suffix of u'. Thus, the statement holds with $r = r' \leq \delta(A)$. □

The following theorem shows that *mmdist*(v, u) is a lower bound for the unit edit distance of v and u.

Theorem 4 Suppose δ is the unit cost function. Then *mmdist*$(v, u) \leq edist_\delta(v, u)$.

Proof: Let A be an optimal alignment of v and u. Then by Observation 4 there is a partition Ψ of v with respect to u such that $|\Psi| \leq \delta(A)$. Hence, *mmdist*$(v, u) \leq |\Psi| \leq \delta(A) = edist_\delta(v, u)$. □

The relation between *mmdist* and $edist_\delta$ suggests to use *mmdist* as a filter in contexts where the unit edit distance is of interest only below some threshold k. In fact, there are algorithms for the approximate sequence searching problem using filtering techniques based on maximal matches.

3.8 The q-Gram Model

Like the maximal matches model, the q-gram model considers common substrings of the strings to be compared. Technically, one counts the number of occurrences of different q-grams in the two sequences. Thus, sequences with many common q-grams have a small distance, independent of where they occur. The q-gram model was first described in [Ukk92]. While the maximal matches model considers substrings of possibly different length, the q-gram model restricts to substrings of a fixed length q. In this section, let q be a positive integer. Recall that u and v are sequences of length m and n, respectively.

Definition 13 The *q-gram profile* of u is the function $G_q(u) : \mathcal{A}^q \to \mathbb{N}$, such that $G_q(u)(w)$ is the number of different positions in u where the sequence $w \in \mathcal{A}^q$ ends. □

The choice of the parameters q and $|\mathcal{A}|$ is very important for the q-gram distance. For example, if $q = 3$ and $|\mathcal{A}| = 4$, then $|\mathcal{A}|^q = 64$. That is, we can assume that in a short string, all q-gram occur. If $q = 4$ and $|\mathcal{A}| = 20$, then $|\mathcal{A}|^q = 160000$ and the string has to be very long to contain all q-grams. In general, one chooses $q \ll n$, e.g. $q \in [8, 11]$ when comparing entire genomes.

Definition 14 The *q-gram distance* $qgdist(u, v)$ of u and v is defined by

$$qgdist(u, v) = \sum_{w \in \mathcal{A}^q} |G_q(u)(w) - G_q(v)(w)|. \ \square$$

One can show that the symmetry and the triangle inequality hold for $qgdist$ (cf. [Ukk92]). The zero property does not hold as shown by the following example. Hence, $qgdist$ is not a metric.

Example 17 Let $q = 2$, $u = aaba$ and $v = abaa$. Then u and v have the same q-gram profile $\{aa \mapsto 1, ab \mapsto 1, ba \mapsto 1, bb \mapsto 0\}$. Hence, the q-gram distance of u and v is 0. □

The simplest method to compute the q-gram distance is to encode each q-gram into a number, and to use these numbers as indices into tables holding the counts for the corresponding q-gram.

Definition 15 Let $\mathcal{A} = \{a_1, \ldots, a_r\}$. Then

$$\overline{a_l} = l - 1$$

is the code of a_l and

$$\overline{w} = \sum_{i=1}^{q} \overline{w[i]} \cdot r^{q-i}$$

is the code of $w \in \mathcal{A}^q$.

If all characters in w have a minimum code 0, then

$$\begin{aligned}\overline{w} &= \sum_{i=1}^{q} \overline{w[i]} \cdot r^{q-i} \\ &= \sum_{i=1}^{q} 0 \cdot r^{q-i} \\ &= \sum_{i=1}^{q} 0 \\ &= 0\end{aligned}$$

That is, the minimum code of any w is 0. The maximum code is obtained when all characters in w have a maximum code $r-1$. Then

$$\begin{aligned}\overline{w} &= \sum_{i=1}^{q} \overline{w[i]} \cdot r^{q-i} \\ &= \sum_{i=1}^{q} (r-1) \cdot r^{q-i} \\ &= (r-1) \cdot \sum_{i=1}^{q} r^{q-i} \\ &= (r-1) \cdot (r^0 + r^1 + \cdots + r^{q-2} + r^{q-1}) \\ &= r \cdot (r^0 + r^1 + \cdots + r^{q-2} + r^{q-1}) - (r^0 + r^1 + \cdots + r^{q-2} + r^{q-1}) \\ &= r^1 + r^2 + \cdots + r^{q-1} + r^q - (r^0 + r^1 + \cdots + r^{q-2} + r^{q-1}) \\ &= r^1 + r^2 + \cdots + r^{q-1} - (r^1 + \cdots + r^{q-1}) + r^q - r^0 \\ &= r^q - 1\end{aligned}$$

That is, maximum code of any $w \in \mathcal{A}^m$ is $r^q - 1$. Moreover, for different $u, w \in \mathcal{A}^q$, we have $\overline{u} \neq \overline{w}$. As $|\mathcal{A}^q| = r^q$ and there are r^q numbers in the range from 0 to $r^q - 1$, we conclude: for each i, $0 \leq i \leq r^q - 1$, there is some $w \in \mathcal{A}^q$ such that $\overline{w} = i$. In other words, the mapping from q-grams to integer codes is bijective.

Example 18 Let $\mathcal{A} = \{A, C, G, T\}$ be the DNA-alphabet and define $\overline{A} = 0$, $\overline{C} = 1$, $\overline{G} = 2$, and $\overline{T} = 3$. For $q = 3$, there are $r^q = 4^3 = 64$ q-words. The smallest q-gram AAA in the lexicographic order of all q-grams gets integer code 0, the second smallest AAC gets integer code 1, etc. In general, the ith q-gram in lexicographic order of all q-gram gets code i. The latter holds for any q and any alphabet, if the encoding of single characters respects the alphabet order.

An important property is that the code of each q-gram can be computed incrementally in constant time, due to the fact that $\overline{xc} = (\overline{ax} - \overline{a} \cdot r^{q-1}) \cdot r + \overline{c}$ for any $x \in \mathcal{A}^{q-1}$ and any $a, c \in \mathcal{A}$.

The algorithm to compute the q-gram distance (see Algorithm 4) follows the following strategy:

1. Accumulate the q-gram profiles of u and v in two arrays τ_u and τ_v such that

$$\tau_u[\overline{w}] = G_q(u)(w) \text{ and } \tau_v[\overline{w}] = G_q(v)(w)$$

   for all $w \in \mathcal{A}^q$.

2. Compute the list $C = \{\overline{w} \mid w \text{ is } q\text{-gram of } u \text{ or } v\}$.

3. Compute $qgdist(u, v) := \sum_{c \in C} |\tau_u[c] - \tau_v[c]|$.

Let us consider the efficiency of the algorithm. The space for the arrays τ_u and τ_v is $O(r^q)$. The space for the set C is $O(m - q + 1 + n - q + 1) = O(m + n)$. Hence the total space requirement is $O(m + n + r^q)$. We need $O(r^q)$ time to initialize the arrays τ_u and τ_v. The computation of the codes requires $O(m + n)$ time. Each array lookup and update requires $O(1)$. Hence the total time requirement is $O(m+n+r^q)$. If $r^q \in O(n+m)$, then this method is optimal.

Like the maximal matches distance, the q-gram distance provides a lower bound for the unit edit distance.

Theorem 5 Let δ be the unit cost function. Then $qgdist(u, v)/(2 \cdot q) \leq edist_\delta(u, v)$.

Proof: See [JU91, Ukk92].

The relation between $qgdist$ and $edist_\delta$ suggests to use $qgdist$ as a filter in contexts where the unit edit distance is of interest only below some threshold k.

3.9 Significance of local alignments

When computing local alignments, the first question is often: Is this local alignment occurring by chance or not. More precisely, one is interested in quantifying the statistical significance of a local alignment, based on a simple model of random sequences. The statistical significance is usually expressed by an expectation value (E-value for short). The smaller the E-value, the more significant the local alignment.

In this section we want to show how to determine an E-value for a local alignment with a maximum number of mismatches. We exclude alignments with indels, because these are difficult to handle statistically. The cost of the local alignment is thus measured in terms of the hamming distance of the sequences involved in the matches. For two strings x and y of equal length, we define

$$\mathcal{H}(x, y) = |\{i \mid i \in [1, |x|], x[i] \neq y[i]\}|.$$

$\mathcal{H}(x, y)$ is the hamming distance of x and y.

Figure 3.13: The recurrence for table R_{j_0}

$$R_{j_0}(i,j) = \begin{cases} \textit{undefined} & \text{if } j \in [0, j_0 - 1] \\ i & \text{else if } j = j_0 \\ R_{j_0}(i, j-1) & \text{else if } E_\delta(i,j) = E_\delta(i,j-1) + \delta(\varepsilon \to v[j]) \\ R_{j_0}(i-1, j-1) & \text{else if } E_\delta(i,j) = E_\delta(i-1,j-1) + \delta(u[i] \to v[j]) \\ R_{j_0}(i-1, j) & \text{else if } E_\delta(i,j) = E_\delta(i-1,j) + \delta(u[i] \to \varepsilon) \end{cases}$$

Algorithm 4 Computing the q-gram distance

Input: sequences $u = u[1 \ldots m]$ and $v = v[1 \ldots n]$ both over alphabet $\mathcal{A}$
$q > 0$

Output: $qgdist(u, v)$

$r := |\mathcal{A}|$
for $c := 0$ **to** $r^q - 1$ **do**
  $\tau_u[c] := 0$
  $\tau_v[c] := 0$
$c := \sum_{i=1}^{q} \overline{u[i]} \cdot r^{q-i}$
$\tau_u[c] := 1$
$C := \{c\}$
for $i := 1$ **to** $m - q$ **do**
  $c := (c - \overline{u[i]} \cdot r^{q-1}) \cdot r + \overline{u[i+q]}$
  if $\tau_u[c] = 0$ **then** $C := C \cup \{c\}$
  $\tau_u[c] := \tau_u[c] + 1$
$c := \sum_{i=1}^{q} \overline{v[i]} \cdot r^{q-i}$
$\tau_v[c] := 1$
if $\tau_u[c] = 0$ **then** $C := C \cup \{c\}$
for $i := 1$ **to** $n - q$ **do**
  $c := (c - \overline{v[i]} \cdot r^{q-1}) \cdot r + \overline{v[i+q]}$
  if $\tau_u[c] = 0$ **and** $\tau_v[c] = 0$ **then** $C := C \cup \{c\}$
  $\tau_v[c] := \tau_v[c] + 1$
return $\sum_{c \in C} |\tau_u[c] - \tau_v[c]|$

Now consider two sequences u and v of length m and n, respectively. We want to compare these and allow a maximal number $k \geq 0$ of mismatches. That is, k is a *mismatch threshold*.

An *local k-mismatch alignment of u and v* (k-LMA, for short) of length ℓ is a triple (ℓ, i, j) such that the following holds:

- $k < \ell$, i.e. the number of mismatches is smaller than the length of the match.
- $i + \ell - 1 \leq m$ and $j + \ell - 1 \leq n$, i.e. $u[i \ldots i + \ell - 1]$ is a substring of u and $v[j \ldots j + \ell - 1]$ is a substring of v.
- $\mathcal{H}(u[i \ldots i + \ell - 1], v[j \ldots j + \ell - 1]) \leq k$, i.e. the number of mismatches between $u[i \ldots i + \ell - 1]$ and $v[j \ldots j + \ell - 1]$ is at most k.

It makes sense to extend each alignment maximally to both sides. Hence we define a notion of maximality: (ℓ, i, j) is *left-maximal* if either $i = 1$ or $j = 1$ or $u[i-1] \neq v[j-1]$. (ℓ, i, j) is *right maximal* if either $i + \ell = m + 1$ or $j + \ell = n + 1$ or $u[i + \ell] \neq v[j + \ell]$. (ℓ, i, j) is *maximal* if it is left-maximal and right maximal.

To assess the significance of an k-LMA, one determines its *E-value*, i.e., the number of local alignments of the same length or longer and with the same or a fewer number of mismatches, that one would expect to find between two random sequences of the same length as u and v.

As a model of random sequences, we assume the uniform Bernoulli model. That is, we assume that at each position of a random sequence, each of the r characters in $\mathcal{A}$ occurs with the same probability $p = \frac{1}{r}$.

3.9.1 Restricting to exact matches

Let us first restrict to the case that $k = 0$, in which case we have no mismatches in the local alignments (i.e. they represent *maximal exact matches*). We first show an important property of maximal exact matches of length $\geq \ell$.

Observation 5 For any pair of sequences u and v, the number of maximal exact matches of length $\geq \ell$ equals the number of left-maximal exact matches of length exactly ℓ.

Proof: Let $M_{max \geq \ell}$ be the set of maximal exact matches of length $\geq \ell$ and $M_{lmax=\ell}$ be the set of left-maximal exact matches of length exactly ℓ. We have to show that these two sets are of the same size. This property is shown by constructing a bijective mapping between the two sets. Bijective means one-by-one, that is, we can map each element in $M_{max \geq \ell}$ to a unique element in $M_{lmax=\ell}$ and vice versa. The mapping is as follows: Each maximal exact match (q, i, j) for $q \geq \ell$ (thus it is an element in $M_{max \geq \ell}$) is mapped to (ℓ, i, j) (which is an element in $M_{lmax=\ell}$). (ℓ, i, j) is left maximal, but not right-maximal whenever $q > \ell$. Let us denote this mapping by γ. To show that γ is bijective, one first needs to show it is injective, i.e. $\gamma((q, i, j)) \neq \gamma((q', i', j'))$ for all pairs of different maximal exact matches (q, i, j) and (q', i', j'). So let (q, i, j) and (q', i', j') be different maximal exact matches. Suppose that $(i, j) = (i', j')$. As $(q, i, j) \neq (q', i', j')$ we conclude $q \neq q'$. This is a

contradiction, since the length of a maximal exact match at a given pair of positions (i, j) is uniquely determined. Hence $(i, j) \neq (i', j')$ which implies $\gamma((q, i, j)) = (\ell, i, j) \neq (\ell, i', j') = \gamma((q', i', j'))$. γ is also surjective, since each left-maximal match (ℓ, i, j) can be extended on the right to a maximal match (q, i, j) of length $q \geq \ell$. The fact that γ is injective and surjective implies that it is bijective. As a consequence, $M_{max \geq \ell}$ and $M_{lmax=\ell}$ are of the same size.

Ignoring the effects of boundary cases (where $i = 1$ or $j = 1$ or $i + \ell - 1 \geq m$ or $j + \ell - 1 \geq n$), we obtain the following equations for the sought expected value:

$$
\begin{aligned}
&\mathbb{E}[\#\text{ of maximal exact substring matches of length} \geq \ell] \\
&= \mathbb{E}[\#\text{ of left-maximal exact substring matches of length } \ell] \\
&= \sum_{i \in [1,m], j \in [1,n]} Pr[(\ell, i, j) \text{ is a left-maximal exact substring match}] \\
&= \sum_{i \in [1,m], j \in [1,n]} Pr[u[i \ldots i + \ell - 1] = v[j \ldots j + \ell - 1] \text{ and } u[i-1] \neq v[j-1]] \\
&= \sum_{i \in [1,m], j \in [1,n]} Pr[u[i \ldots i + \ell - 1] = v[j \ldots j + \ell - 1]] \cdot Pr[u[i-1] \neq v[j-1]] \\
&= \sum_{i \in [1,m], j \in [1,n]} p^{\ell}(1-p) \\
&= m\, n\, p^{\ell}\, (1-p).
\end{aligned}
$$

Note that the second equality is due to the fact that we can count the number of left left-maximal exact substring matches of length ℓ by looking at each pair of positions i and j, considering the probability of a match at that position and summing up all such probabilities.

So we have finally shown that

$$\mathbb{E}[\#\text{ of maximal exact substring matches of length} \geq \ell] = m\, n\, p^{\ell}\, (1-p).$$

Example 19 shows some E-values of exact matches of length $\geq \ell$.

3.9.2 Allowing for mismatches

E-values for k-LMAs can be computed in a similar way. First, assume fixed values for ℓ and k. To determine the E-values in the general case, we have first compute the number of choices of k positions with mismatches from ℓ possible positions. We can apply a standard formula from combinatorics:

Given a set X of size ℓ, there are exactly

$$\binom{\ell}{k} = \frac{\ell!}{k! \quad (\ell - k)!} \tag{3.4}$$

subsets $Y \subseteq X$ such that Y has exactly k elements. To see this, consider the following: at first, we have ℓ choices for the first element of a subset, $\ell - 1$ choices for the second

element of a subset etc. In general, we have $(\ell-(q-1))$ choices for the qth element of the subset, for all q, $1 \leq q \leq \ell$. All combinations of choices are possible. Hence if we make the choices one after the other, we have

$$\prod_{q=1}^{k}(\ell-(q-1)) = \frac{\ell!}{(\ell-k)!} \tag{3.5}$$

choices altogether to obtain the subsets. However, many choices lead to the same sets. In particular, if we choose the same elements in a different order, we obtain the same subset. More precisely, each permutation of each subset of size k is generated. Now the number of permutations of a set of size k is $k!$. We however only want one subset for each such permutation of choices and thus we have to divide (3.5) by $k!$ to obtain (3.4).

Now we apply the above formula. At first note that there are $\binom{\ell}{k}$ choices for k positions out of ℓ possible positions in two strings of length ℓ. For each of the k positions, there is a mismatch with probability $1-p$. Hence, since the positions are independent, $(1-p)^k$ is the probability that there is a mismatch in all k positions. Moreover, p^{l-k} is the probability that there is a match in all remaining $l-k$ positions of the strings. Hence, the probability of two independent random sequences x and y, both of length ℓ, to have a hamming distance of exactly k is

$$Pr[\mathcal{H}(x,y)=k] \;=\; \binom{\ell}{k}p^{\ell-k}(1-p)^k.$$

To compute the expected number of LMAs of length ℓ or longer and with k or fewer mismatches, one has to sum over all possible $k' \leq k$ and over all lengths $\ell' \geq \ell$. The latter is necessary, in contrast to the case of exact substring matches, because for k-LMAs it is no longer true that the number of LMAs of length $\geq \ell$ equals the number of left-maximal LMAs of length exactly ℓ. Hence, if we ignore boundary cases, we obtain:

$$\begin{aligned}
&\mathbb{E}[\#\text{ of maximal} \leq k\text{-LMAs of length} \geq \ell] \\
&= \sum_{k'=0}^{k}\sum_{\ell'=\ell}^{\min(m,n)}\sum_{i\in[1,m],j\in[1,n]} Pr[(\ell',i,j)\text{ is a maximal }k'\text{-LMA}] \\
&= \sum_{k'=0}^{k}\sum_{\ell'=\ell}^{\min(m,n)}\sum_{i\in[1,m],j\in[1,n]} Pr[\mathcal{H}(u[i\ldots i+\ell'-1], \\
&\qquad\qquad v[j\ldots j+\ell'-1]) = k' \text{ and} \\
&\qquad\qquad u[i-1]\neq v[j-1] \text{ and } u[i+\ell']\neq v[j+\ell']] \\
&= \sum_{k'=0}^{k}\sum_{\ell'=\ell}^{\min(m,n)}\sum_{i\in[1,m],j\in[1,n]} Pr[\mathcal{H}(u[i\ldots i+\ell'-1], v[j\ldots j+\ell'-1]) = k']\cdot \\
&\qquad\qquad Pr[u[i-1]\neq v[j-1]]\cdot \\
&\qquad\qquad Pr[u[i+\ell']\neq v[j+\ell']] \\
&= \sum_{k'=0}^{k}\sum_{\ell'=\ell}^{\min(m,n)}\sum_{i\in[1,m],j\in[1,n]} \binom{\ell'}{k'}p^{\ell'-k'}(1-p)^{k'}(1-p)(1-p)
\end{aligned}$$

$$= \sum_{i\in[1,m],j\in[1,n]} \sum_{k'=0}^{k} \sum_{\ell'=\ell}^{\min(m,n)} \binom{\ell'}{k'} p^{\ell'-k'}(1-p)^{k'+2}$$
$$= m\,n \sum_{k'=0}^{k} \sum_{\ell'=\ell}^{\min(m,n)} \binom{\ell'}{k'} p^{\ell'-k'}(1-p)^{k'+2}.$$

Because the sums are largely dominated by the terms for $k' = k$ and $\ell' = \ell$, this can be approximated by

$$m\,n \binom{\ell}{k} p^{\ell-k}(1-p)^{k+2}.$$

Example 19 Consider two sequences of length $m = n = 10^6$. Let $p = \frac{1}{4}$. Then the E-values of an k-LMA of length $\geq \ell$ for different values of $\ell \in \{16, 32, 64, 128\}$ and $k \in \{0, 1, 2, 3\}$ are shown in the following table:

	k			
ℓ	0	1	2	3
16	1.3e+02	6.3e+03	1.4e+05	2.0e+06
32	3.0e-08	2.9e-06	1.4e-04	4.1e-03
64	1.7e-27	3.2e-25	3.0e-23	1.9e-21
128	4.9e-66	1.9e-63	3.6e-61	4.5e-59

CHAPTER 4

Database Search Methods

The amount of sequence information in today's data bases is growing very fast. This is especially true for the domain of genomics: The current and future projects to sequence large genomes (e.g. human, mouse, rice) produces gigabytes and will soon produce terabytes of sequence data, mainly DNA sequences and Protein sequences derived from the former. To make wealth out of these sequences, larger and larger instances of string processing problems have to be solved. While the standard methods for sequence comparison based on dynamic programming are well accepted, they are not fast enough to allow processing large search spaces. We have seen two alternative methods to perform sequence comparison. These provide valid, and formally defined models and corresponding algorithms compute the models in sub-quadratic time. However, in the real world other methods are used to detect interesting sequence similarities in large search spaces. These methods are subsumed under the name data base search methods. Table 4.1 gives an overview of the different database search methods. In this chapter we will describe two different programs for database searches, namely Fasta and Blast. We will also describe techniques to search position specific scoring matrices.

4.1 The Program Fasta

Fasta is a very popular tool for comparing biological sequences. It was introduced in [LP85, Pea90]. First consider the problem the Fasta-program was designed for: Let w be a *query sequence* (e.g. a novel DNA-sequence or an unknown protein). Let S be a set of sequences (the database). The problem is to find all sequences in S, which are similar to w.

We now need to define the similarity notion used by Fasta. There is no formally clear

Type of search	Method	Type of query data	Program examples	Results of database search
Sequence similarity search with query sequence	search for database sequence that can be aligned with query sequence	single sequence, e.g. DAHQSNGA	SSEARCH; BlastP; WU-BLAST	list of database sequences having the most significant similarity scores
Alignment search with profile (scoring matrix with gap penalties)	prepare profile from a multiple sequence alignment (Profilemake) and align profile with database sequence	profile representing gapped multiple sequence alignment	PROFILESEARCH	list of database sequences that can be aligned with the profile
Search with position specific scoring matrix (PSSM) representing ungapped sequence alignment (BLOCK)	prepare PSSM from ungapped region of multiple sequence alignment or search for patterns of same length in unaligned sequences, the use for database search	PSSM representing ungapped alignment	MAST	list of database sequences with one or more patterns represented by PSSM but not necessarily in the same order
Iterative alignment search for similar sequences that starts with a query sequence, builds a gapped multiple alignment, and then uses the alignment to augment the search	uses initial matches to query sequence to build a type of scoring matrix and searches for additional matches to the matrix by an iterative search method	builds matches to query sequence, use results in further iterations	PSI-BLAST	PSI-BLAST finds a set of sequences related to each other by the presence of common patterns (not every sequence may have same patterns)
Search query sequence for patterns representative of protein families	search for patterns represented by scoring matrix or Hidden Markov Model (profile HMM)	single sequence, e.g. DAHQSNGA	PROSITE; INTERPRO; PFAM; CDD/IMPALA	list of sequence patterns found in query sequence

Table 4.1: Overview of Database Search Methods as adapted from the textbook of Mount.

Table 4.2 lists the URLs of the mentioned software tools.

Tool	Web-Address
PROFILESEARCH	`ftp.sdsc.edu/pub/sdsc/biology/`
SSEARCH	`http://fasta.bioch.virginia.edu/fasta/`
BlastP	`http://www.ncbi.nlm.nih.gov/BLAST/`
WU-BLAST	`http://blast.wustl.edu/`
MAST	`http://meme.sdsc.edu/meme/website/mast.html`
PSI-BLAST	`http://www.ncbi.nlm.nih.gov/BLAST`
PROSITE	`http://www.expasy.ch/prosite`
INTERPRO	`http://www.ebi.ac.uk/interpro`
PFAM	`http://pfam.sanger.ac.uk`
CDD/IMPALA	`http://www.ncbi.nlm.nih.gov/Structure/cdd/cdd.shtml`

Table 4.2: The URLs of the software tools listed in Table 4.1.

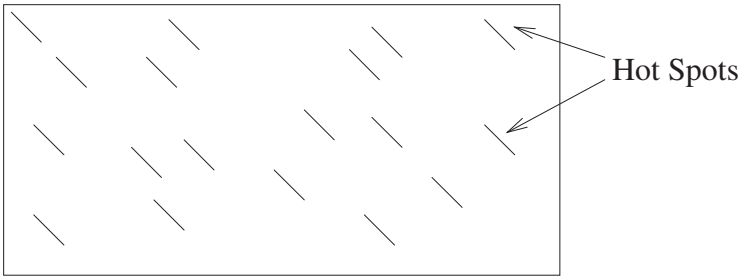


Figure 4.1: Hot Spots in the Fasta Algorithm

model of what Fasta computes, but a heuristic stepwise procedure defined next.

4.1.1 Finding Hot Spots

Consider an arbitrary but fixed $u \in S$. Let E_δ be the corresponding distance matrix with w, defined by the equation $E_\delta(i,j) = edist_\delta(u[1\ldots i], w[1\ldots j])$ where δ is the unit cost function. Let q be a fixed constant. One chooses $q = 6$ for DNA and $q = 2$ for Proteins. The idea is to count for each diagonal the number of minimizing subpaths of length q on this diagonal. Each such minimizing subpath stands for a common q-gram in u and w, also called "hot-spot", see Figure 4.1 for illustration. The number of hot spots on each diagonal gives a score, according to the following definition:

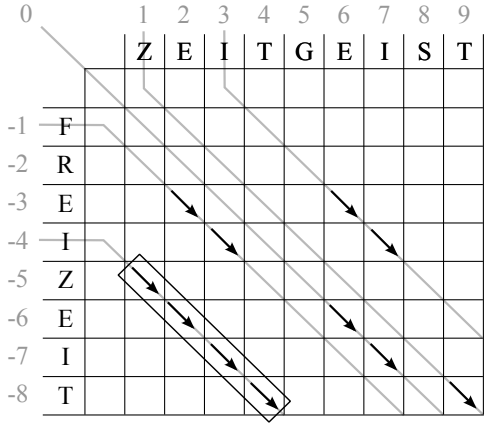
Definition 16 Let $m = |u|$ and $n = |w|$. For $d \in [-m, n]$ let

$$\textit{count}(u,w,d) = |\{(i,j) \mid i \in [1, |u|-q+1], j \in [1,|w|-q+1], j-i = d, \\ u[i\ldots i+q-1] = w[j\ldots j+q-1]\}|$$

The Fasta score is defined by $score_{\text{fasta}}(u,w) = \max\{\textit{count}(u,w,d) \mid d \in [-m,n]\}$. □

Example 20 Let $u =$ `FREIZEIT`, $w =$ `ZEITGEIST`, and $q = 2$. Then $\textit{count}(u,w,-4) =$

Figure 4.2: The matching diagonals for `freizeit` and `zeitgeist` as determined by the Fasta Algorithm. The numbers are diagonal numbers.



3, $\mathit{count}(u, w, -1) = 1$, $\mathit{count}(u, w, 0) = 1$, $\mathit{count}(u, w, 3) = 1$ and $\mathit{count}(u, w, d) = 0$ for $d \notin \{-4, -1, 0, 3\}$. This can be easily verified in Figure 4.2. □

Note that only the subpaths on the same diagonal are counted. In other words, the matching q-grams have to be at the same distance in both u and w. This is the main difference to the q-gram distance model, where the order of the q-grams is not important.

Algorithm 5 provides a method to compute the Fasta-score. The running time of this algorithm is clearly $O(r^q + m + n + \sum_{d=-m}^{n} \mathit{count}(u, w, d))$ for one database sequence u. That is, the more similar w and u, the more time the algorithm requires.

4.1.2 Combining Hot Spots to Diagonal Runs

If the Fasta-score for some database sequence w is smaller than some minimum threshold, then it is discarded. Otherwise, w is processed further by looking for diagonal runs in the Edit-matrix (without computing the matrix, of course). Diagonal runs are hot spots appearing on the same diagonal, with small gaps in between, see Figure 4.3 for an illustration. To score diagonal runs, one assigns positive weights to the hot spots and negative penalties to gaps. Note that not necessarily all hot spots on the same diagonal are put into a single diagonal run. Instead a diagonal can contain more than one diagonal run.

4.1.3 Constructing a Directed Graph from Diagonal Runs

In the next step, a directed graph is constructed. The nodes of the graph are the diagonal runs from the previous step with corresponding weights assigned. Let us denote a diagonal run d by the upper left corner $(l_1(d), l_2(d))$ and the lower right corner $(r_1(d), r_2(d))$. The

Algorithm 5 An algorithm to compute $score_{\text{fasta}}(u, w)$.

1. Encode each q-gram as an integer $c \in [0, r^q - 1]$, where $r = |\mathcal{A}|$. The details of this encoding are described in Section 3.8.

2. The query sequence w is preprocessed into a function $h_w : [0, r^q - 1] \to \mathcal{P}(\mathbb{N})$ defined by

$$h_w(c) := \{i \in [1, |w| - q + 1] \mid c = \overline{w[i \ldots i + q - 1]}\}$$

   That is, each "bucket" $h_w(c)$ holds the positions in w where the q-gram with code c occurs.

3. In the final phase, the data base is processed.

foreach $u \in S$
  $m := |u|$
  $n := |w|$
  for $d := -m$ **to** n **do** $\mathit{count}(u, w, d) = 0$
  for $j := 1$ **to** $m - q + 1$ **do**
    $c := \overline{u[j \ldots j + q - 1]}$
    foreach $i \in h_w(c)$
      $\mathit{count}(u, w, j - i) := \mathit{count}(u, w, j - i) + 1$
  $score_{\text{fasta}}(u, w) := \max\{\mathit{count}(u, w, d) \mid d \in [-m, n]\}$

nodes for diagonal d and d' are connected if $r_1(d) \leq l_1(d')$ and $r_2(d) \leq l_2(d')$. The edges get a negative weight. The graph is obviously acyclic and therefore we can efficiently compute a path of maximal total weight.

Suppose that all paths of maximal weight are computed. From each path we pick the first and the last node. The upper left corner of the first node and the lower right corner of the last node define a pair of substrings of w and u. These are aligned using standard global alignment algorithms, see Figure 4.4 for Illustration.

4.2 The Program BLAST

BLAST is the most popular program to perform sequence database searches. As in [AGM+90], we will describe the program for the case where the input sequences are proteins. Therefore the corresponding subprogram of BLAST is BLASTP. Nevertheless we will use the term BLAST and not BLASTP here. Our exposition very closely follows [CWC04].

The BLAST algorithm is a five-stage process that is efficient and effective for searching genomic databases. The steps progressively reduce the search space, but each is more fine-grain and takes longer to process each sequence. Table 4.3 shows the average time spent performing each stage of the algorithm.

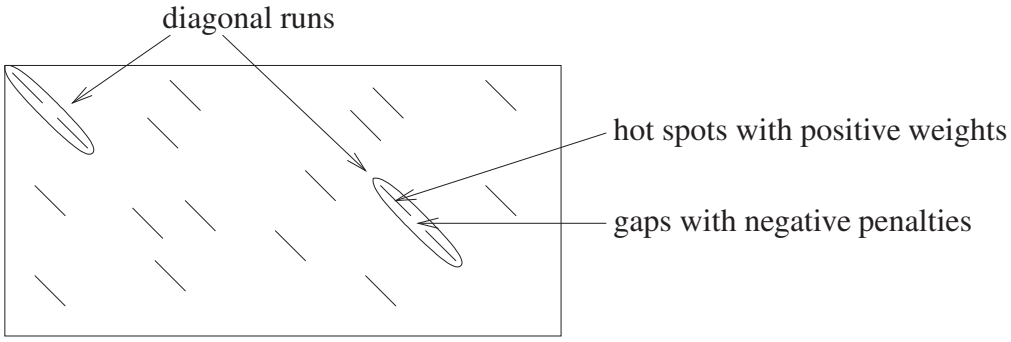


Figure 4.3: Diagonal Runs in the Fasta Algorithm

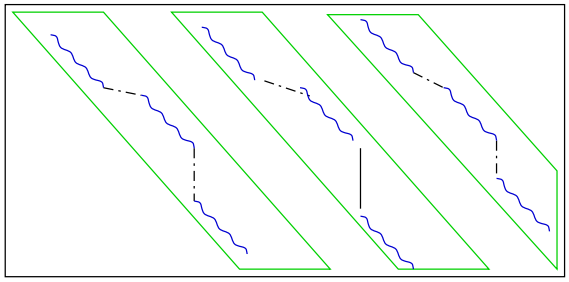


Figure 4.4: Optimal Paths from Diagonal Runs

Stage	Task	relative overall time
1	find blast hits	37%
2	identify pairs of hits on the same diagonal	18%
3	perform ungapped extension	13%
4	perform gapped extension	30%
5	collect traceback information and display alignments	2%

Table 4.3: Average running time of each stage of the BLAST algorithm according to the evaluation of [CWC04].

4.2.1 Stage 1: find blast hits

In the first stage, each database sequence is compared against the query sequence to compute blast hits. To explain what this is, suppose we want to search a protein sequence database, given a score function σ, satisfying $\sigma(\alpha \to \beta) = -\infty$ for any deletion or insertion operation $\alpha \to \beta$. That is, the model does not allow for insertions and deletions. Again we suppose a query sequence w and an arbitrary but fixed database sequence u.

Definition 17 Let $q \in \mathbb{N}$ and $k \geq 0$ be a threshold. A blast-hit is a pair (i, j) of indices such that $score_\sigma(u[i \ldots i+q-1], w[j \ldots j+q-1]) \geq k$. □

We now sketch an algorithm to find blast-hits. This algorithm is iterated over all database sequences u.

1. In the first step, we construct the k-environment for the query sequence w, defined by

   $$Env_k(w) = \{(s, j) \mid s \in \mathcal{A}^q, j \in [1, |w| - q + 1], score_\sigma(s, w[j \ldots j+q-1]) \geq k\}$$

   This can be done by first enumerating all strings of length q occurring in w. For each such substring, say s, one generates a trie[1] representing $\mathcal{A}^q$. On each node α of the trie representing the sequence y of length q' for some $q' \leq q$, one maintains $score_\sigma(y, s[1 \ldots q'])$. The nodes of the trie at depth q with a score $\geq k$ represent a sequence from the k-environment of w. To speed up the computation, one can apply methods to prune the trie. This means that once a node is reached which cannot be on a path representing a sequence in the environment, then the entire subtree below the node need not be computed.

2. Note that all strings $s \in \mathcal{A}^q$ satisfying $score_\sigma(s, w[j \ldots j+q-1]) \geq k$ are of the same length q. We can therefore map each $s \in \mathcal{A}^q$ to a unique integer $\overline{s}$ in the range $[0, r^q - 1]$ where $r = |\mathcal{A}|$.

3. $Env_k(w)$ is represented as follows:
   - by a vector V of r^q bits such that $V[\overline{s}]$ is 1 iff $(s, j) \in Env_k(w)$ for some j.
   - for each $s \in \mathcal{A}^q$ satisfying $V[\overline{s}] = 1$, we store the set $\mathit{Pos}(s) := \{j \mid (s, j) \in Env_k(w)\}$.

4. Each database sequence u is then processed by shifting a window of length q over it. Suppose that $i + q$ characters have been processed and let s be the current substring of length q in the window, i.e. $s = u[i \ldots i+q-1]$. Then we compute $\overline{s}$ in constant amortized time and look up if $V[\overline{s}]$ is 1. If this is the case, then for all $j \in \mathit{Pos}(s)$, (i, j) is a hit.

[1] A trie is a tree whose edges are labeled by single characters, such that for each node α and each character a there is at most one edge from α labeled a. A trie represents a sequence set, say S, if the sequences in S can be read from the paths of the trie.

The first three steps only depend on the query sequence w. Hence they only have to be performed once for the database.

Example 21 Suppose that σ is the BLOSUM62-score function, see Figure 3.4. Let $q = 4$, $k = 22$ and $w = \texttt{MGHLPLAWLSQ}$. Then we obtain the following k-environment $Env_k(w)$:

(MGHL,0)	(GHLP,1)	(GHIP,1)	(GHMP,1)	(GHVP,1)
(HLPL,2)	(PLAW,4)	(PLSW,4)	(PLTW,4)	(PLVW,4)
(PLGW,4)	(PLXW,4)	(PLCW,4)	(PIAW,4)	(PMAW,4)
(PVAW,4)	(PFAW,4)	(LAWL,5)	(AWLS,6)	(WLSQ,7)
(WISQ,7)	(WMSQ,7)			

The size of the vector and the sets $Pos(s)$ grow exponentially with q and $1/k$. So these parameters should be selected carefully. For protein sequences, $q = 4$ and $k = 22$ (if σ is the BLOSUM62 score function) seem to be a reasonable choice.

Once w has been preprocessed, u is processed in $O(|u| + z)$ time, where z is the number of hits. In an older version of BLAST used until about 1998, each BLAST hit triggered the processing of the next stage, namely the ungapped extensions. The newer version of BLAST (BLAST 2.0 and later), an intermediate step was introduced, which is described next.

4.2.2 Stage 2: Combining blast hits at close distance

In this stage, BLAST looks for pairs of blast hits on the same diagonal within some maximal distance, say δ, see the left side of Figure 4.5. Consider two hits (i, j) and (i', j') on the same diagonal d, i.e. $j - i = d = j' - i'$ and suppose that $i < i'$. As the blast hits are computed in increasing order of the positions in the database, (i, j) is computed before (i', j'). From the assumption we can conclude $i = j - j' + i'$ and as $i < i'$, $j - j' < 0$. Hence $j < j'$. Thus, with respect to the same diagonal, blast hits are computed in increasing order of the positions in both sequences. Hence, for each diagonal d it suffices to keep track of the last index $f(d) = i$ such that there is a blast hit (i, j) satisfying $d = j - i$. For any new blast hit (i', j') one computes the diagonal index $d' = j' - i'$. If $f(d')$ is defined, then $f(d') < i'$ and one checks $i' \leq f(j' - i') + \delta$. If this is the case, then the distance of the two blast hits $(f(d'), d' + f(d'))$ and (i', j') on the diagonal d' is at most δ and one has found a pair of relevant blast hits which is further processed. In any case, one updates $f(d') := i'$.

4.2.3 Stage 3: Ungapped extensions

Suppose we have a pair of blast hits on the same diagonal within some maximal distance δ. To determine if the pair of blast hits occurs in a region of pairwise similarity, one computes the mid point between the two hits and uses this as the starting point for the ungapped left-extension and the right-extension method described below. The sensitivity of the extensions depends on a "drop-off" parameter $X_d > 0$. An appropriate choice of parameters q, k, δ, X_d leads to a method which is fast and sensitive.

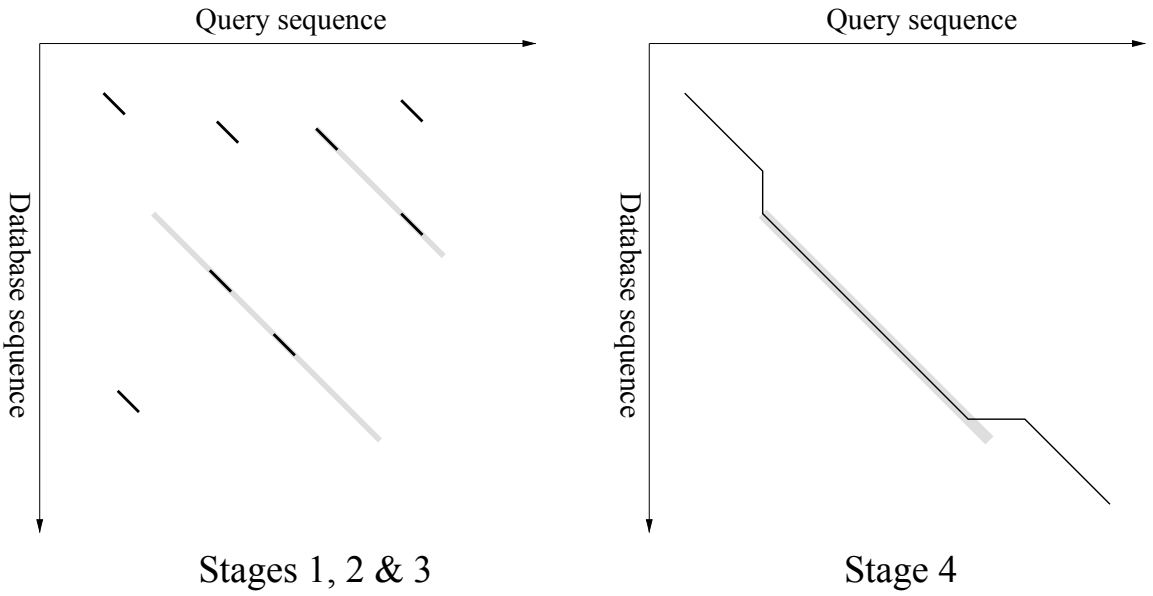


Figure 4.5: Illustration of the first four stages of the BLAST algorithm. In stage 1 BLAST hits are identified, shown as short black lines. In stage 2 one identifies pairs of hits occurring near each other and on the same diagonal. These are subject to ungapped extensions in stage 3, with the result shown as a longer gray line. In this example, the longer of the two ungapped extensions scores above some threshold and is passed on to stage 4, where it is used as a starting point for constructing a higher-scoring gapped alignment.

We here describe the ungapped extension for a mid-point (i, j) between two blast hits. Let $m = |u|$ and $n = |w|$. For the extension to the left, the sequences $u[1 \ldots i-1]$ and $w[1 \ldots j-1]$ are compared from right to left. For the extension to the right, the sequences $u[i \ldots m]$ and $w[j \ldots n]$ are compared from left to right. Each pair of characters $(u[i-l], w[j-l])$, $l \in [1, \min\{i-1, j-1\}]$, and $(u[i+r], w[j+r])$, $r \in [0, \min\{m-i, n-j\}]$, delivers a score according to the score function σ. For both extensions, the scores are accumulated and the maximum value $X_{\max}$ reached during the extension is kept track of. As soon as a score smaller than $X_{\max} - X_d$ is reached, the extension is stopped, see Algorithm 6 for a function computing the length of the extension to the right. The pair of sequences delivered by the extensions to the left and to the right is called maximum segment pair (MSP, see Fig. 4.6). For any such MSP, a significance score is computed. If this is better than some predefined significance threshold, then the MSP is processed by the next stage.

Algorithm 6 A function extending a blast hit (i, j) for the sequences database sequence u and the query sequence w to the right. σ is the score function and $X_d > 0$ is drop-off parameter.

```
1:  function extendhitright(u, w, σ, X_d, i, j)
2:  X_max := 0
3:  scoresum := 0
4:  r := 0
5:  while i + r ≤ |u| and j + r ≤ |w| do
6:      scoresum := scoresum + σ(u[i + r], w[j + r])
7:      if scoresum < X_max − X_d then
8:          break
9:      end if
10:     if scoresum > X_max then
11:         X_max := scoresum
12:     end if
13:     r := r + 1
14: end while
15: return r − 1
```

4.2.4 Stage 4: Gapped extension

In the fourth stage, a gapped alignment is performed to determine if the high scoring ungapped region forms part of a larger, higher scoring alignment. The right side of Figure 4.5 illustrates an example where this is the case: The single, high-scoring ungapped extension identified in Stage 2 is considered as the basis of a gapped alignment, and the black line shows the alignment identified through this process.

The gapped alignment algorithm used by BLAST differs from Smith-Waterman local alignment. Rather than exhaustively computing all possible paths between the sequences,

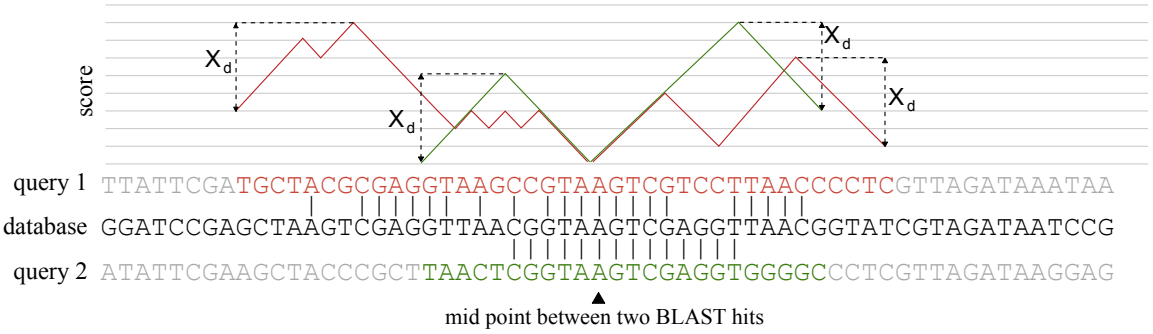


Figure 4.6: Illustration of the BLAST hit extension strategy, each beginning at the position marked by the triangle. Two queries are shown above and below the database sequence, respectively. We assume a match score of 1 and a mismatch score of -1. At the top of the figure, the accumulated score obtained when comparing the red and green part of the query sequence against the corresponding part of the database sequence is shown in red and green respectively. The extension stops when the accumulated score X drops below $X_{\max} - X_d$, which is signified by showing the corresponding query positions in light gray.

the gapped scheme explores only insertions and deletions that augment the high-scoring ungapped alignment. Therefore, this step begins by identifying a seed point that lies within a high-scoring portion of the ungapped region. After this, a gapped alignment is attempted, using affine gap scores. The method is very similar to the computation of optimal global alignment for affine gap costs, see Section 3.5.2.

A gapped alignment stops when the score falls below a value determined by another dropoff parameter, Y_d. This parameter controls the sensitivity and speed trade off: the higher the value of Y_d, the greater the alignment sensitivity but the slower the search process. If the resulting gapped alignment scores more than a minimum value (which is determined from an external E-value cutoff parameter) it is passed on to the fourth and final stage of BLAST. On average, according to [CWC04], less than 0.01 percent of the gapped alignments performed during the third stage score above the default E-value cutoff of 10, and that this stage consumes on average 30 percent of the total search time.

4.2.5 Stage 4: Collect traceback information and display alignments

BLAST uses Karlin-Altschul statistics [SW96, KA90] to report the statistical significance of each gapped alignment to the user as an E-value. The E-value represents the chance that an alignment with at least the same score would be found if a randomly constructed query with typical amino-acid composition is searched against a randomly generated database. The length of the query sequence and the size of the collection are taken into consideration. There are three steps to determine the statistical significance of a gapped alignment:

1. A nominal score S is determined for each alignment using a mutation data scoring

matrix, typically either a BLOSUM [HH93] or PAM [DSO78] matrix and a function for penalizing gaps.

2. The nominal score is converted to a normalized score S' defined as follows:

$$S' = \frac{\lambda S - \ln K}{\ln 2}$$

where λ and K are precomputed by random simulation for each scoring matrix and gap penalty combination. Scores in this normalized form are expressed in bits and are comparable across different scoring schemes.

3. The normalized score is converted into an E-value E as follows:

$$E = \frac{Q}{2^{S'}}$$

where Q is the size of the search space, that is, $Q \approx mn$, where m and n are the total number of residues in the query and the database, respectively.

The resulting value of E is reported to the user. As the equations above are invertible, one can determine the minimum nomimal score required to achieve a specific E-value. This approach is used by BLAST to determine the cutoff parameter from the user-specified E-value.

4.3 Searching Position Specific Scoring Matrices

Position specific scoring matrices (PSSMs) have a long history in sequence analysis. A high PSSM-score in some region of a sequence often indicates a possible biological relationship of this sequence to the family or motif characterized by the PSSM. There are several databases incorporating PSSMs, e.g. PROSITE, PRINTS, BLOCKS, or TRANSFAC. In this section we describe what PSSMs are and we develop methods to search PSSMs.

A PSSM is a representation of a multiple alignment of related sequences. We define it as a function $M : [1, m] \times \mathcal{A} \to \mathbb{R}$, where m is the length of M and $\mathcal{A}$ is a finite alphabet. Usually M is represented by an $m \times |\mathcal{A}|$ matrix, see Table 4.4 for an example. Each row of the matrix reflects the frequency of occurrence of an amino acid or nucleotide at the corresponding position of the alignment. From now on, let M be a PSSM of length m. We define $score(w, M) = \sum_{h=1}^{m} M(h, w[h])$ for a sequence $w \in \mathcal{A}^m$ of length m. Given a sequence S of length n over alphabet $\mathcal{A}$ and a threshold value th, the *PSSM searching problem* is to find all positions $j \in [1, n - m + 1]$ in S such that $score(S[j \ldots j + m - 1], M) \geq th$.

A simple algorithm for the PSSM searching problem slides along the sequence and computes $score(w, M)$ for each $w = S[j \ldots j + m - 1]$, $j \in [1, n - m + 1]$. The running time of this algorithm is $O(mn)$. It is used e.g. in the programs *FingerPrintScan* [SFA99], *BLIMPS* [HGPH00], *MatInspector* [QFWW95], and *MATCH* [KGR+03].

The technique of lookahead scoring, introduced in [WNMB00], gives an improvement over the simple algorithm. Lookahead scoring allows to stop the calculation of $score(w, M)$

when it is clear that the given overall score threshold th cannot be achieved. To explain the method, we define

$$\begin{aligned} \mathit{pfxscore}_d(w, M) &= \sum_{h=1}^{d} M(h, w[h]), \\ \max_d &= \max\{M(d, a) \mid a \in \mathcal{A}\}, \\ \sigma_d &= \sum_{h=d+1}^{m} \max_h \end{aligned}$$

for any $d \in [1, m]$. That is, $\mathit{pfxscore}_d(w, M)$ is the score for the prefix $w[1 \ldots d]$ of length d, and σ_d is the maximal score that can be achieved in the last $m-d$ positions of the PSSM. Let $th_d = th - \sigma_d$ be the *intermediate threshold* at position d. The next lemma reveals an important property of prefix scores:

Lemma 1 The following statements are equivalent:

(1) $\mathit{pfxscore}_d(w, M) \geq th_d$ for all $d \in [1, m]$.

(2) $\mathit{score}(w, M) \geq th$.

Proof: (1)⇒(2): Suppose that (1) holds. Then $\sigma_m = \sum_{h=m+1}^{m} \max_h = 0$ and

$$\begin{aligned} \mathit{score}(w, M) &= \sum_{h=1}^{m} M(h, w[h]) \\ &= \mathit{pfxscore}_m(w, M) \\ &\geq th_m \\ &= th - \sigma_m \\ &= th. \end{aligned}$$

(2)⇒(1): Suppose that (2) holds. Let $d \in [1, m]$. Then

$$\begin{aligned} \mathit{score}(w, M) &= \sum_{h=1}^{m} M(h, w[h]) \\ &= \sum_{h=1}^{d} M(h, w[h]) + \sum_{h=d+1}^{m} M(h, w[h]) \\ &= \mathit{pfxscore}_d(w, M) + \sum_{h=d+1}^{m} M(h, w[h]) \end{aligned}$$

Hence $\mathit{score}(w, M) \geq th$ implies $\mathit{pfxscore}_d(w, M) + \sum_{h=d+1}^{m} M(h, w[h]) \geq th$. Since $M(h, w[h]) \leq \max_h$ for $h \in [1, m]$, we conclude

$$\sum_{h=d+1}^{m} M(h, w[h]) \leq \sum_{h=d+1}^{m} \max_h = \sigma_d$$

and hence

$$\begin{aligned} pfxscore_d(w, M) &\geq th - \sum_{h=d+1}^{m} M(h, w[h]) \\ &\geq th - \sigma_d \\ &= th_d. \end{aligned}$$

Lemma 1 gives a necessary condition for a PSSM-match, which can easily be exploited: When computing $score(w, M)$ by scanning w from left to right, one checks for $d = 1, 2, \ldots$, if the intermediate threshold th_d is achieved. If not, the computation can be stopped. See Table 4.4 for an example and Algorithm 7 for pseudo-code. The lookahead scoring algorithm runs in $O(kn)$ time, where k is the average number of PSSM-positions per sequence position actually evaluated. In the worst case, $k \in O(m)$, which leads to the worst case running time of $O(mn)$, not better than the simple algorithm. However, k is expected to be much smaller than m, leading to considerable speedups in practice.

Algorithm 7 The lookahead scoring algorithm

Input: Sequence S of length n, PSSM M of length m, threshold th
Output: all positions in S matching M.

```
for j := 1 to n − m + 1 do
    score := 0
    for d := 1 to m do
        score := score + M(d, S[j + d − 1])
        if score < th_d then
            break
    if score ≥ th then
        print match at position j with score score
```

A	-19	5	7	-29	-14	-25	7	-34	7	-7
C	**92**	-17	-8	**99**	-22	-34	-8	-27	**40**	**43**
D	-45	17	-29	-55	14	-25	-25	-44	-16	16
E	-49	**22**	-28	-61	22	-16	-24	-43	-14	-7
F	-30	-28	2	-42	-28	-37	-19	-60	-9	-27
G	-36	-15	-25	-45	9	-30	-23	-41	-14	-15
H	-38	-7	-10	-47	-8	-15	-22	-8	-6	-9
I	-12	-23	25	-31	-26	-36	4	-16	-17	-24
K	-41	-8	-23	-52	15	45	-15	-38	14	-5
L	-21	-27	-4	-34	-27	-34	-10	-14	-20	-26
M	-22	-21	-5	-36	-20	-26	-8	-17	-15	-18
N	-40	-26	-25	-49	-7	-18	-19	-39	-10	-6
P	-46	18	-32	-56	-26	-35	-29	-51	-24	-25
Q	-44	-7	-26	-55	-3	-9	-21	-40	-11	25
R	-44	-13	-25	-55	**31**	**49**	11	-36	12	13
S	-30	-9	-18	-38	-13	-25	-13	-39	15	25
T	-25	9	13	-35	5	-26	**31**	-35	9	-8
V	16	-19	22	-29	-23	-33	31	-21	-13	-21
W	-35	-33	-11	-44	-30	-39	-31	-1	-16	-30
Y	-34	-25	**36**	-46	-24	-31	-22	**56**	20	-24
σ_d	407	385	349	250	219	170	139	83	43	0
th_d	-7	15	51	150	181	230	261	317	357	400

Table 4.4: PSSM of length $m = 10$ of a zinc-finger motif. If the score threshold is $th = 400$, then only substrings beginning with C or V can match the PSSM, since all other amino acids score below the intermediate threshold $th_0 = -7$. That is, lookahead scoring will skip over all substrings beginning with amino acids different from C and V.

CHAPTER 5

Multiple Sequence Alignment

Up until now we always have considered comparing two sequences. This section is devoted to the comparison of $k \geq 2$ sequences to obtain multiple alignments. For example, given the following protein sequences (small sections of six tyrosine kinase sequences),

```
>SRC_RSVP
FPIKWTAPEAALYGRFTIKSDVWSFGILLTELTTKGRVPYPGMVNREVLDQVERG
>YES_AVISY
FPIKWTAPEAALYGRFTIKSDVWSFGILLTELVTKGRVPYPGMVNREVLEQVERG
>ABL_MLVAB
FPIKWTAPESLAYNKFSIKSDVWAFGVLLWEIATYGMSPYPGIDLSQVYELLEKD
>FES_FSVGA
QVPVKWTAPEALNYGRYSSESDVWSFGILLWETFSLGASPYPNLSNQQTREFVEKG
>FPS_FUJSV
QIPVKWTAPEALNYGWYSSESDVWSFGILLWEAFSLGAVPYANLSNQQTREAIEQG
>KRAF_MSV36
TGSVLWMAPEVIRMQDDNPFSFQSDVYSYGIVLYELMAGELPYAHINNRDQIIFMVGRG
```

we would like to obtain a multiple alignment like this:

```
SRC_RSVP     -FPIKWTAPEAALY---GRFTIKSDVWSFGILLTELTTKGRVPYPGMVNR-EVLDQVERG
YES_AVISY    -FPIKWTAPEAALY---GRFTIKSDVWSFGILLTELVTKGRVPYPGMVNR-EVLEQVERG
ABL_MLVAB    -FPIKWTAPESLAY---NKFSIKSDVWAFGVLLWEIATYGMSPYPGIDLS-QVYELLEKD
FES_FSVGA    QVPVKWTAPEALNY---GRYSSESDVWSFGILLWETFSLGASPYPNLSNQ-QTREFVEKG
FPS_FUJSV    QIPVKWTAPEALNY---GWYSSESDVWSFGILLWEAFSLGAVPYANLSNQ-QTREAIEQG
KRAF_MSV36   TGSVLWMAPEVIRMQDDNPFSFQSDVYSYGIVLYELMA-GELPYAHINNRDQIIFMVGRG
```

Multiple sequence alignments are usually constructed for a set of sequences that are assumed to be homologous (i.e., they have a common ancestor sequence) and the goal is

usually to detect homologous residues or bases and place them in the same column of the multiple alignment.

In the following, we will describe two approaches for scoring multiple alignments, and several methods which compute multiple alignments with optimal or near optimal score. The description follows [SM97].

Suppose that we are given a set S of k sequences $s_1, \ldots, s_k$ over some alphabet $\mathcal{A}$. Let $n_i := |s_i|$ for each $i \in [1,k]$. Let $- \notin \mathcal{A}$ be a gap-symbol, and $\mathcal{A}' = \mathcal{A} \cup \{-\}$. To get rid of gaps in sequences we define a function $delgap : (\mathcal{A}')^* \to \mathcal{A}^*$ by

$$\begin{aligned} delgap(a) &= a \text{ for each } a \in \mathcal{A} \\ delgap(-) &= \varepsilon \\ delgap(w[1 \ldots n]) &= delgap(w[1]) \ldots delgap(w[n]) \text{ for each sequence } w \in (\mathcal{A}')^n \end{aligned}$$

A *multiple sequence alignment* (*MSA*, for short) of S is a sequence of k strings $(s'_1, \ldots, s'_k)$ satisfying the following conditions:

(1) There is some $l \geq 1$ such that $s'_i \in (\mathcal{A}')^l$ for all $i \in [1,k]$,

(2) $delgap(s'_i) = s_i$ for all $i \in [1,k]$, i.e. deleting all gap symbols in s'_i gives s_i

(3) for all $j \in [1,l]$, there is at least one $i \in [1,k]$ such that $s'_i[j] \neq -$, i.e. there is no column in the *MSA* which consists of gaps only.

$l = |s'_1|$ is the length of the *MSA* $(s'_1, \ldots, s'_k)$. Note that l satisfies the following condition:

$$\max\{n_i \mid i \in [1,k]\} \leq l \leq \sum_{i=1}^{k} n_i$$

As with pairwise alignments we show an *MSA* by placing the s'_i on top of each other, see the example at the beginning of this section.

5.1 Scoring MSAs by consensus distance

We consider two basic approaches to score *MSA*s. The first approach is based on the notion of consensus. The consensus of a column is a non-gap symbol that occurs most often in the column. The consensus sequence is the sequence of consensi of the columns. Counting in all columns the number of symbols which are not identical to the consensus symbol of the column leads to a distance notion.

Consider an *MSA* $(s'_1, \ldots, s'_k)$ of the sequences $s_1, \ldots, s_k$. Let l be the length of the *MSA*. For each $j \in [1,l]$ and each $a \in \mathcal{A}$ we define $count(j,a) = |\{i \in [1,k] \mid s'_i[j] = a\}|$, i.e. $count(j,a)$ is the number of occurrences of character a in the jth column of the *MSA*. A sequence $cons$ of length l is a *consensus* of $(s'_1, \ldots, s'_k)$ if for each $j \in [1,l]$, $count(j, cons[j]) \geq count(j,a)$ for all characters $a \in \mathcal{A}$.

$$dist(cons, (s'_1, \ldots, s'_k)) = \sum_{j=1}^{l} (k - count(j, cons[j]))$$

is the *distance* of the consensus $cons$ and the *MSA*.

It does not matter which consensus we take, the distance to the given *MSA* is invariant:

Lemma 2 Let $cons_1$ and $cons_2$ be two different consensi of the same *MSA* $(s'_1, \ldots, s'_k)$. Then

$$dist(cons_1, (s'_1, \ldots, s'_k)) = dist(cons_2, (s'_1, \ldots, s'_k)).$$

Proof: Lets consider a particular column of the *MSA*, i.e. lets fix $j \in [1, l]$. First note that neither $count(j, cons_1[j]) > count(j, cons_2[j])$ nor $count(j, cons_1[j]) < count(j, cons_2[j])$. Hence $count(j, cons_1[j]) = count(j, cons_2[j])$, i.e. $cons_1[j]$ and $cons_2[j]$ occur the same number of times. As a consequence

$$k - count(j, cons_1[j]) = k - count(j, cons_2[j])$$

Since this holds for each $j \in [1, l]$, the sum of the sizes over all j and thus the distance must be identical. □

Since the distance is not influenced by the choice of the consensus, it makes sense to evaluate an *MSA* according to the distance to a consensus. That is, we define the consensus costs by

$$conscost(s'_1, \ldots, s'_k) = dist(cons, (s'_1, \ldots, s'_k))$$

where $cons$ is a consensus. The smaller the distance, the better the multiple alignment. Small distances mean that the columns of the alignment mostly contain a few different characters. Let us consider an example:

Example 22 Let $s_1 =$ `AATGCT`, $s_2 =$ `ATTC`, $s_3 =$ `TCC`. An *MSA* of these sequences is as follows, with a consensus shown below the bottom line:

s'_1	A A T G C T
s'_2	A − T T C −
s'_3	− − − T C C
$cons$	A A T T C T

We have $dist(cons, (s'_1, s'_2, s'_3)) = 1 + 2 + 1 + 1 + 0 + 2 = 7$.

The notion of consensus distance leads to the Multi-Alignment Consensus problem, which consists of finding the *MSA* $(s'_1, \ldots, s'_k)$ of the given sequences $s_1, \ldots, s_k$ such that the consensus cost $conscost(s'_1, \ldots, s'_k)$ is minimal.

Note that the notion of consensus, as defined above, is often called *majority voting*. Often the notion consensus is defined in a more general way as *some* sequence derived from a multiple alignment (without stating how it is derived).

5.2 Scoring MSAs by sums of pairwise scores

The second optimization criterion for *MSA*s is based on pairwise distances. Let δ be a function assigning a value $\delta(\alpha \to \beta) \geq 0$ to each edit operation $\alpha \to \beta$, extended by defining $\delta(- \to -)$ as some non-negative value.

The score is determined by computing the sum of all pairs of the characters and gaps in each column of the *MSA*. Hence it is called *sum-of-pairs scoring* (SP-scoring, for short). Consider an *MSA* $(s'_1, \ldots, s'_k)$ of length l. For each $j \in [1, l]$ we define the SP-column score

$$\delta_{\mathrm{SP}}(s'_1[j], \ldots, s'_k[j]) = \sum_{i=1}^{k-1} \sum_{r=i+1}^{k} \delta(s'_i[j] \to s'_r[j])$$

The SP-alignment score is then defined as the sum of all SP-column scores:

$$\delta_{\mathrm{SP}}(s'_1, \ldots, s'_k) = \sum_{j=1}^{l} \delta_{\mathrm{SP}}(s'_1[j], \ldots, s'_k[j])$$

Example 23 Let $s_1 = \mathtt{AATGCT}$, $s_2 = \mathtt{ATTC}$, $s_3 = \mathtt{TCC}$ be the sequences from Example 22. Consider the following *MSA* of s_1, s_2, s_3:

$$\begin{array}{ll} s'_1 & \mathtt{A\ A\ T\ G\ C\ T} \\ s'_2 & \mathtt{A\ -\ T\ T\ C\ -} \\ s'_3 & \mathtt{-\ -\ -\ T\ C\ C} \end{array}$$

Define the distance function δ such that for all $\alpha, \beta \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, -\}$ we have $\delta(\alpha \to \beta) = 0$, if $\alpha = \beta$ and $\delta(\alpha \to \beta) = 1$, otherwise. Then we obtain the following SP-score for the alignment above:

$$\begin{aligned} \delta_{\mathrm{SP}}(s'_1, s'_2, s'_3) &= \sum_{j=1}^{6} \sum_{i=1}^{2} \sum_{r=i+1}^{3} \delta(s'_i[j], s'_r[j]) \\ &= \sum_{j=1}^{6} (\delta(s'_1[j], s'_2[j]) + \delta(s'_1[j], s'_3[j]) + \delta(s'_2[j], s'_3[j])) \\ &= (0+1+1) + (1+1+0) + (0+1+1) + \\ &\quad (1+1+0) + (0+0+0) + (1+1+1) \\ &= 2+2+2+2+0+3 \\ &= 11 \end{aligned}$$

The *MSA*-problem with SP-scoring is to find the *MSA* of the given sequences with minimum SP-alignment score. The minimum score is denoted by $\delta_{\mathrm{opt-SP}}(s_1, \ldots, s_k)$.

Note that for both scoring models we consider here, minimization is the optimization criterion. Despite this fact, we often use the notion "similarity", which is usually used in combination with maximizing scores.

5.3 Computing Optimal Multiple Sequence Alignments

The dynamic programming techniques for pairwise sequence alignments can be generalized to *MSA*s in a straightforward way: One computes an $(n_1+1) \times \cdots \times (n_k+1)$-table M such that for each $(q_1, \ldots, q_k) \in [0, n_1] \times \cdots \times [0, n_k]$, $M(q_1, \ldots, q_k)$ is the score of the optimal *MSA* for the prefixes $s_1[1 \ldots q_1], \ldots, s_k[1 \ldots q_k]$. Now lets develop a recurrence for M. In analogy to the pairwise case, we consider all possible last columns of an *MSA* of $s_1[1 \ldots q_1], \ldots, s_k[1 \ldots q_k]$. The last column consists of any combination of gaps and characters $s_i[q_i]$, $i \in [1, k]$. If the ith row of the last column is character $s_i[q_i]$, then in dimension i, one has to refer to the entries in $M(\ldots, q_i - 1, \ldots)$. If the ith row of the last column is a gap, then in dimension i, one has to refer to the entries in $M(\ldots, q_i, \ldots)$. Assuming that the minimum of an empty set is 0, we thus obtain the following recurrence for M:

$$\begin{aligned} M(q_1, \ldots, q_k) = \min\{ & M(q_1 - d_1, \ldots, q_k - d_k) + \\ & colcost(\varphi(q_1, d_1), \ldots, \varphi(q_k, d_k)) \mid (d_1, \ldots, d_k) \in \{0,1\} \times \cdots \times \{0,1\}, \\ & (d_1, \ldots, d_k) \neq (0, \ldots, 0), \\ & q_1 \geq d_1, \ldots, q_k \geq d_k \} \end{aligned}$$

where

$$\varphi(q_i, d_i) = \begin{cases} s_i[q_i] & \text{if } d_i = 1 \\ - & \text{otherwise} \end{cases}$$

and $colcost$ is a function assigning costs to a column of k elements from $\mathcal{A}'$, depending on the chosen model. If the model is based on the best consensus, then $colcost = conscost$. In the SP-scoring model, $colcost = \delta_{\text{SP}}$.

Note that table M is of size $z = \prod_{i=1}^{k}(n_i + 1)$. Each of the z entries in table M requires to compute the minimum of up to $2^k - 1$ values. Each such value is the sum of another entry in table M plus the cost for the corresponding column. In case of the best consensus model, the column cost can be evaluated in $O(k)$ time. In case of the SP-scoring model, the column cost can be evaluated in $O(k^2)$ time, since each of $k(k-1)/2$ pairwise scores have to be added up.

Example 24 Consider the problem for $k = 3$ sequences. Then there are $2^k - 1$ neighboring edges along which the minimum is to be computed, see Figure 5.1.

Table M can be organized as a one dimensional array of size z. Each entry stores a distance value and a bitvector with k bits. The latter encodes a k-tuple $(d_1, \ldots, d_k)$ which gave rise to the minimum distance value. That is, bit i of the bitvector is set if and only if $d_i = 1$. Given $(q_1, \ldots, q_k)$, one computes the address of the entry in the array in $O(k)$ time, and accesses it in constant time. As a consequence, each entry is computed in $O(f(k) \cdot 2^k)$ time where $f(k) = k$ for the consensus model and $f(k) = k^2$ for the SP-scoring model. Hence the dynamic programming algorithm computes the distance of an optimal multiple alignment in $O(z \cdot f(k) \cdot 2^k)$ time. Having stored a bitvector in each entry, one can perform a traceback starting at entry $M(n_1, \ldots, n_k)$, computing each column of an optimal alignment

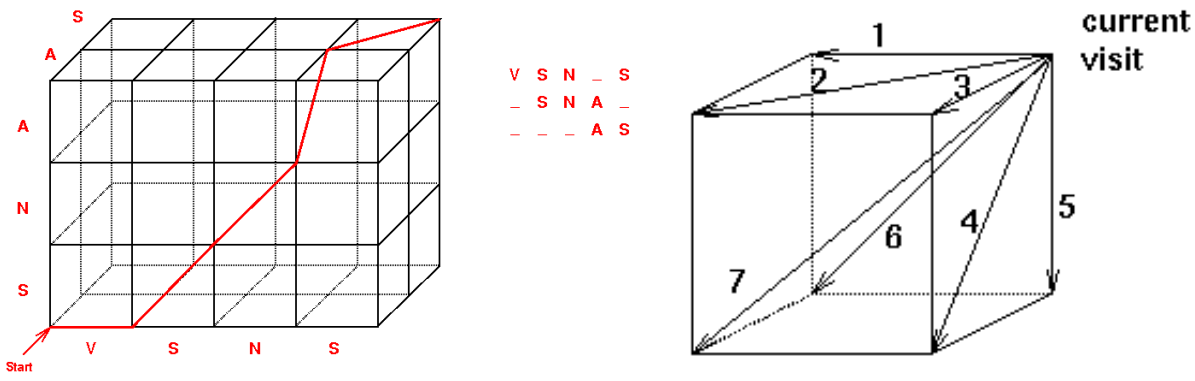


Figure 5.1: The 3 sequences case: alignment path and neighboring edges for a given node.

in $O(k)$ steps. Hence the traceback phase requires $O(k \cdot l)$ steps, where l is the length of the *MSA*. The space requirement of the algorithm is $O(z)$.

Given these results, it is clear that the method is only applicable, if k is small. Consider the SP-scoring model and assume that each computation step requires 1 μs. Then, computing the alignment for 2 sequences of length 100 requires about $2^2 \cdot 100^2 \approx 40$ ms. For 4 sequences of length 100, we need $2^4 \dot{1}00^4 \approx 1600$ s, and for 8 sequences of length 100, we need $2^8 \dot{1}00^8 \approx 83$ years. Also the space requirement is large: $\approx 10\ kB$ for two sequences, $10^8 \approx 95\ MB$ for 4 sequences, and $10^{16} \cdot 2$ byte $\approx 18TB$ for 8 sequences, each of length 100.

5.4 Combining Pairwise Alignments

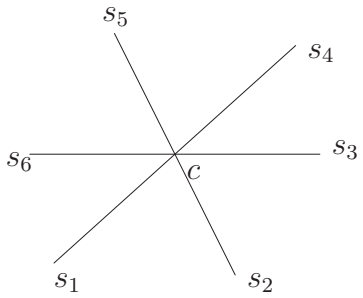
Since the exact computation of optimal *MSA*s is very costly in practice, we will discuss methods which deliver approximations of the optimal solutions. The idea is to construct an *MSA* from a set of pairwise alignments of the given sequences. In order to do so, we have to define the following notion:

Let $S = \{s_1, \ldots, s_k\}$ be a set of sequences, and S' be a subset of S. Suppose an *MSA* $Al_S = (s'_1, \ldots, s'_k)$ of S. The *projection of* Al_S *with respect to* S' is the *MSA* $proj(Al_S, S')$ produced as follows:

- delete all rows in Al_S which do not correspond to a sequence in S'.
- in the remaining rows delete all columns which only consist of gaps.

The alignment Al_S is said to be *compatible* with the projection $Al_{S'} = proj(Al_S, S')$.

Figure 5.2: A star alignment tree of 7 sequences $c, s_1, \ldots, s_6$ with the center sequence c and the remaining sequences $s_1, \ldots, s_6$ at the leaves.



Example 25 Let $S = \{\texttt{ACGG}, \texttt{ATG}, \texttt{ATCGG}\}$, $S' = \{\texttt{ACGG}, \texttt{ATG}\}$, $S'' = \{\texttt{ATG}, \texttt{ATCGG}\}$ and consider the following *MSA* Al_S of S:

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

Then $proj(Al_S, S')$ is as follows:

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

And $proj(Al_S, S'')$ is as follows:

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

The relation between pairs of sequences are represented in form of an alignment tree:

Let S be a set of sequences. An *alignment tree* for S is a labeled tree where the node set is S and each edge (s, t) in the tree is labeled by an optimal pairwise alignment of the sequences s and t.

There are methods which take an alignment tree, and construct a multiple alignment of S that is compatible with the optimal pairwise alignments in the tree. We will not discuss the general method to do this, but we will consider a special case where the alignment tree is a star, i.e. there is a center node c and edges (c, s) for all $s \in S \setminus \{c\}$. In this case, one often uses the notion *star alignment*. See also Figure 5.2 for a star alignment tree. The star alignment method first finds the sequence in the center, and then constructs from the optimal pairwise alignments on the edges a multiple alignment that is compatible with the pairwise alignments. We also state that the pairwise alignment is compatible with the star alignment. Whenever we have constructed a multiple alignment from the sequences $c, s_1, \ldots, s_i$, and we want to add the pairwise optimal alignment of c and s_{i+1}, we adhere

Algorithm 8 The star alignment algorithm.

Input: A set S of sequences
Output: Star alignment Al of S

1. Compute the center sequence c of the star alignment

   foreach $s \in S$ **do**
   foreach $t \in S \setminus \{s\}$ **do**
   compute an optimal alignment of s and t with distance $\delta_{\text{opt}}(s,t)$
   let $totalscore(t) = \sum_{s \in S \setminus \{t\}} \delta_{\text{opt}}(s,t)$
   let $c \in S$ be a sequence s.t. $totalscore(c) \leq totalscore(s)$ for all $s \in S$
   let T be the star alignment tree with center c and leaves from $S \setminus \{c\}$

2. Determine a compatible multiple alignment

   choose an arbitrary $s \in S$
   let Al be an optimal alignment of c and s
   $S' := \{c, s\}$
   while $S' \neq S$ **do**
   choose an arbitrary $s' \in S \setminus S'$
   $S' := S' \cup \{s'\}$
   Determine a compatible *MSA* Al' of S' by combining Al with an optimal alignment of c and s', adhering to the principle "once a gap always a gap"
   $Al := Al'$

to the principle "once a gap always a gap". That is, the gaps in the pairwise alignment of c and s_{i+1} are inherited to the multiple alignment, see Algorithm 8. Let us consider an example:

Example 26 Consider the sequences $c = \texttt{ATGCATT}$, $s_1 = \texttt{AGTCAAT}$, $s_2 = \texttt{TCTCA}$, $s_3 = \texttt{ACTGTAATT}$ and the alignments

$$\begin{aligned} c' &= \texttt{A T G - C A T T} \\ s_1' &= \texttt{A - G T C A A T} \\ s_2' &= \texttt{- T C T C A - -} \end{aligned}$$

and

$$\begin{aligned} c'' &= \texttt{A - T G C - A T T} \\ s_3'' &= \texttt{A C T G T A A T T} \end{aligned}$$

By combining the gaps from c' and c'', we obtain the sequence $c''' = \texttt{A-TG-C-ATT}$. Furthermore, retaining the columns from both alignments, the resulting alignment is as follows:

$$\begin{aligned} c''' &= \texttt{A - T G - C - A T T} \\ s_1''' &= \texttt{A - - G T C - A A T} \\ s_2''' &= \texttt{- - T C T C - A - -} \\ s_3''' &= \texttt{A C T G - T A A T T} \end{aligned}$$

Note that the resulting *MSA* is not optimal, since it would probably decrease the distance, if we would replace s_3''' by $\texttt{ACTGT-AATT}$.

Next we show that Algorithm 8 delivers a good approximation of the optimal SP-alignment, if the pairwise scoring function satisfies some additional properties.

We say that the cost function $\delta : \mathcal{A}' \to \mathbb{R}_{\geq 0}$ is *good* if the following holds:

1. For each $a \in \mathcal{A}'$, we have $\delta(a, a) = 0$
2. For each $a, b, c \in \mathcal{A}'$, the triangle equality $\delta(a, c) \leq \delta(a, b) + \delta(b, c)$ holds.

Lemma 3 Let δ be a good cost function, let $S = \{c, s_1, \ldots, s_k\}$ be a set of sequences. Suppose T is a star alignment tree with center c, and let $(c', s_1', \ldots, s_k')$ be an *MSA* of S of length l which is compatible with T. Then for all $i, j \in [1, k]$:

$$\delta(s_i', s_j') \leq \delta(s_i', c') + \delta(c', s_j') = \delta_{\text{opt}}(s_i, c) + \delta_{\text{opt}}(c, s_j) \tag{5.1}$$

Proof: Consider column q of $(c', s_1', \ldots, s_k')$. Then, by assumption

$$\delta(s_i'[q], s_j'[q]) \leq \delta(s_i'[q], b) + \delta(b, s_j'[q])$$

for any character $b \in \mathcal{A}'$. Hence, with $b = c'[q]$ we obtain $\delta(s_i'[q], s_j'[q]) \leq \delta(s_i'[q], c'[q]) + \delta(c'[q], s_j'[q])$. Since this triangle inequality holds for any column, it holds for the sum over all columns, and therefore

$$\begin{aligned}
\delta(s_i', s_j') &= \sum_{q\in[1,l]} \delta(s_i'[q], s_j'[q]) \\
&\leq \sum_{q\in[1,l]} (\delta(s_i'[q], c'[q]) + \delta(c'[q], s_j'[q]) \\
&= \sum_{q\in[1,l]} \delta(s_i'[q], c'[q]) + \sum_{q\in[1,l]} \delta(c'[q], s_j'[q]) \\
&= \delta(s_i', c') + \delta(c', s_j')
\end{aligned}$$

It remains to show the equality in (5.1). By assumption, the *MSA* is compatible with T, i.e. the projection of the *MSA* to $\{s_i, c\}$ is an optimal alignment of s_i and c. Since $\delta(-,-) = 0$, deleting a column consisting of gaps only does not change the score of the pairwise alignment. Therefore, $\delta(s_i', c') = \delta_{\text{opt}}(s_i, c)$. With an analogous argumentation, one shows $\delta(c', s_j') = \delta_{\text{opt}}(c, s_j)$ which completes the proof. □

Now we can prove the central theorem of this section:

Theorem 6 Let δ be a good cost function, let δ_{SP} be the function which evaluates the SP-score for a given *MSA*. Consider a set $S = \{s_1, \ldots, s_k\}$ of sequences, and the *MSA* $Al = (s_1', \ldots, s_k')$ of S, delivered by Algorithm 8. Then the following property holds:

$$\delta_{\text{SP}}(s_1', \ldots, s_k') \leq \left(2 - \frac{2}{k}\right) \cdot \delta_{\text{opt-SP}}(s_1, \ldots, s_k)$$

Proof: Consider an *MSA* $(s_1'', \ldots, s_k'')$ of S such that $\delta_{\text{SP}}(s_1'', \ldots, s_k'') = \delta_{\text{opt-SP}}(s_1, \ldots, s_k)$, i.e. $(s_1'', \ldots, s_k'')$ is an optimal alignment of S. Define two terms $sum' = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s_i', s_j')$ and $sum'' = \sum_{i=1}^{k} \sum_{j=1}^{k} \delta(s_i'', s_j'')$. Furthermore, without loss of generality, we can assume that $c = s_k$ is the central sequence of the star and define $c_{\text{score}} = \sum_{i=1}^{k-1} \delta(s_i, c)$. Now first observe, that

$$\begin{aligned}
sum' &\leq \sum_{i=1}^{k} \sum_{j=1}^{k} (\delta_{\text{opt}}(s_i, c) + \delta_{\text{opt}}(c, s_j)) \\
&= \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} (\delta_{\text{opt}}(s_i, c) + \delta_{\text{opt}}(c, s_j)) \\
&= \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \delta_{\text{opt}}(s_i, c) + \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \delta_{\text{opt}}(c, s_j) \\
&= \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \delta_{\text{opt}}(s_i, c) + \sum_{i=1}^{k-1} \sum_{j=1}^{k-1} \delta_{\text{opt}}(s_j, c)
\end{aligned}$$

$$
\begin{aligned}
&= \sum_{j=1}^{k-1}\sum_{i=1}^{k-1}\delta_{\text{opt}}(s_i,c) + \sum_{j=1}^{k-1}\sum_{i=1}^{k-1}\delta_{\text{opt}}(s_i,c)\\
&= 2\cdot(k-1)\cdot\sum_{i=1}^{k-1}\delta_{\text{opt}}(s_i,c)\\
&= 2\cdot(k-1)\cdot c_{\text{score}}
\end{aligned}
$$

Moreover, we derive another inequality for sum'':

$$
\begin{aligned}
sum'' &\geq \sum_{i=1}^{k}\sum_{j=1}^{k}\delta_{\text{opt}}(s_i,s_j)\\
&= \sum_{i=1}^{k}\left(\sum_{j=1}^{k}\delta_{\text{opt}}(s_i,s_j)\right)\\
&\geq \sum_{i=1}^{k}\left(\sum_{j=1}^{k}\delta_{\text{opt}}(c,s_j)\right)\\
&= k\cdot\sum_{j=1}^{k}\delta_{\text{opt}}(c,s_j)\\
&= k\cdot\sum_{i=1}^{k}\delta_{\text{opt}}(s_i,c)\\
&= k\cdot c_{\text{score}}
\end{aligned}
$$

From these two inequalities, we derive the following chain of inequalities:

$$\frac{sum'}{sum''} \leq \frac{2\cdot(k-1)\cdot c_{\text{score}}}{sum''} \leq \frac{2\cdot(k-1)\cdot c_{\text{score}}}{k\cdot c_{\text{score}}} = \frac{2\cdot(k-1)}{k} = \frac{2k-2}{k} = \frac{2k}{k}-\frac{2}{k} = 2-\frac{2}{k}$$

which is equivalent to $sum' \leq \left(2-\frac{2}{k}\right)\cdot sum''$. Finally, we obtain

$$
\begin{aligned}
\delta_{\text{SP}}(s'_1,\ldots,s'_k) &= \sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\delta(s'_i,s'_j)\\
&= \frac{1}{2}sum'\\
&\leq \frac{1}{2}\cdot\left(2-\frac{2}{k}\right)\cdot sum''\\
&= \left(2-\frac{2}{k}\right)\cdot\frac{1}{2}\cdot sum''\\
&= \left(2-\frac{2}{k}\right)\cdot\frac{1}{2}\cdot\sum_{i=1}^{k}\sum_{j=1}^{k}\delta(s''_i,s''_j)\\
&= \left(2-\frac{2}{k}\right)\cdot\delta_{\text{opt-SP}}(s_1,\ldots,s_k)\ \square
\end{aligned}
$$

This result shows that the star alignment gives a good approximation of the optimal SP alignment. By definition we also have

$$\delta_{\mathrm{opt-SP}}(s_1, \ldots, s_k) \leq \delta_{\mathrm{SP}}(s'_1, \ldots, s'_k)$$

i.e. δ_{SP} is an upper bound for the optimal SP-alignment score. This can be exploited to reduce the search space for an optimal SP-alignment. To understand this, first note that we can efficiently compute $\delta_{\mathrm{SP}}(s'_1, \ldots, s'_k)$ using Algorithm 8. Now, if a value $d = M(q_1, \ldots, q_k)$ satisfies

$$d > \delta_{\mathrm{SP}}(s'_1, \ldots, s'_k)$$

then $d > \delta_{\mathrm{opt-SP}}(s_1, \ldots, s_k)$ and hence an optimal *MSA* represented by a path in M cannot cross $M(q_1, \ldots, q_k)$. Hence, we do not have to compute any matrix entry larger than

$$\delta_{\mathrm{SP}}(s'_1, \ldots, s'_k)$$

This concept of reducing the search space is described by Carillo & Lipman [CL88] and it is implemented in the program MSA.

5.5 Multiple Alignment in Practice

5.5.1 Iterative Multiple Alignment

In order to avoid exponential time complexity, one often reduces the multiple alignment problem to the iterated application of pairwise alignment. Therefore, one first generalizes the pairwise alignment of two sequences to the pairwise alignment of two multiple alignments. This is an easy adaptation where the score function is generalized to a score of two alignment columns rather than two single characters.

A popular protocol for iterative multiple alignment is the following (Fig. 5.3):

1. Compute all pairwise alignment scores.
2. From these scores compute a tree that is used in order to decide about the order of the pairwise alignments, the so-called *guide tree*.
3. Successively align pairs of sequences or alignments guided by the branching order of the guide tree until one large multiple alignment remains.

Suppose we have k sequences $s_1, \ldots, s_k$ and a distance $d(s_i, s_j)$ between each pair of sequences. A guide tree can be computed by clustering the given sequences, at each stage merging two sequences, and at the same time creating a new node in the tree. The tree is assembled "upwards", first clustering pairs of leaves, then pairs of clustered leaves etc. Each node is given a height and the edge lengths are obtained as the difference of heights of its two end nodes.

The most striking advantages of the iterative multiple alignment method as compared to SP-optimal alignment are:

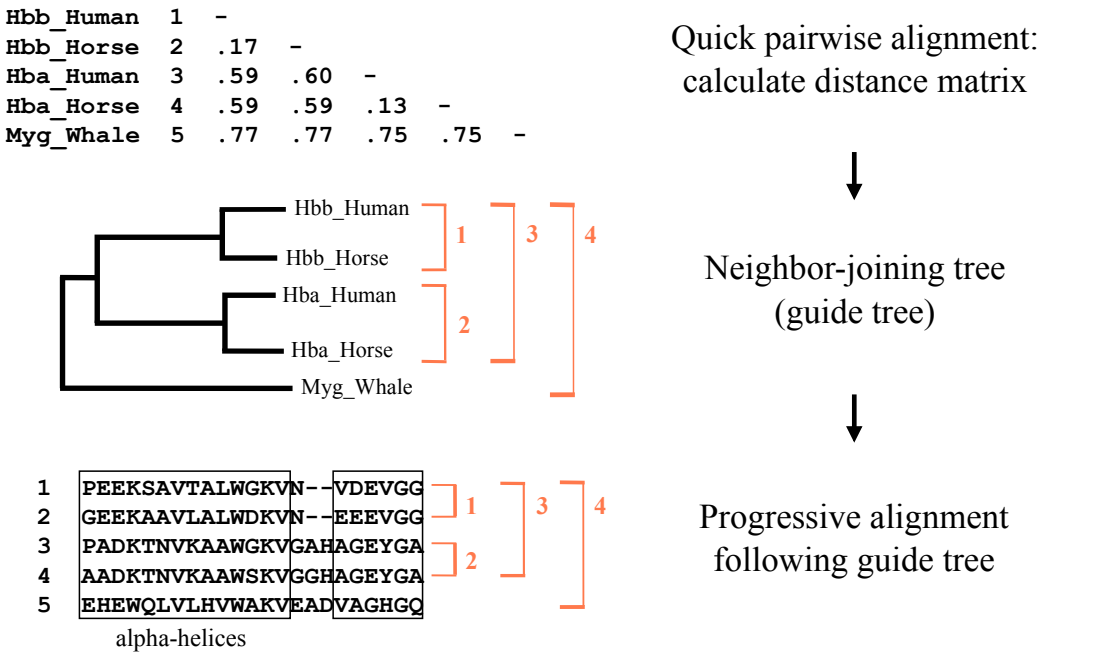


Figure 5.3: Overview of ClustalW procedure. Figure adapted from [http://www.slideshare.net/jomcinerney/alignments]

- This approach leads to fast algorithms that are applicable to many and long sequences.
- Conservations in subfamilies that are aligned before other, less closely related sequences are added, lead to good alignments of subfamily motifs.

On the other hand there are a few disadvantages:

- Errors in an early step can not be eliminated later, i.e. “once a gap, always a gap”.
- In iterative multiple alignment there is no well-defined objective function which must be optimized during the algorithm. Thus it cannot be used to evaluate the performance of this approach in comparison to other heuristic methods.
- A (possibly wrong) guide tree remains represented in the alignment and might bias the result if the alignment is used as basis for reconstructing an evolutionary tree.

The *de-facto* standard for iterative multiple alignment is the ClustalW program,[1] and its graphical user interface ClustalX.[2]

5.5.2 Divide-and-Conquer Alignment

To circumvent the interdependence of tree and alignment is to simply avoid the use of a tree when computing the alignment. Obviously, sum-of-pairs multiple alignment does not

[1] http://www-igbmc.u-strasbg.fr/BioInfo/ClustalW/Top.html
[2] http://www-igbmc.u-strasbg.fr/BioInfo/ClustalX/Top.html

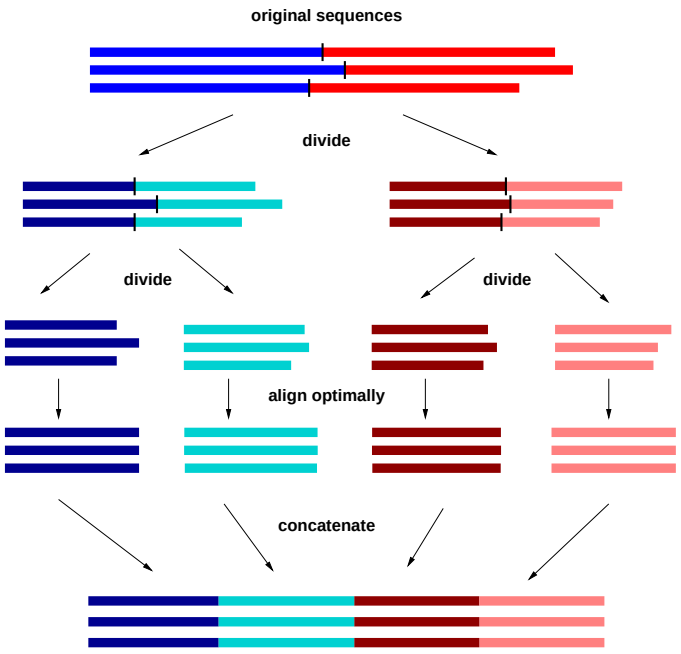


Figure 5.4: The divide-and-conquer alignment procedure (DCA).

require a tree, but we have seen that this problem is very expensive to solve. An approximation to SP-optimal alignment is the divide-and-conquer multiple alignment algorithm (DCA, for short), described in [SMD97].

The general idea of DCA is rather simple: Each sequence is cut in two after a suitable cut position somewhere close to its midpoint. This way, the problem of aligning one family of (long) sequences is divided into the two problems of aligning two families of (shorter) sequences, the prefix and the suffix sequences. This procedure is re-iterated until the sequences are sufficiently short—say, shorter than a user defined size Z which is a parameter of DCA—so that they can be aligned optimally by MSA. Finally, the resulting short alignments are concatenated, yielding a multiple alignment of the original sequences. Figure 5.4 sketches the method.

Computing Cut Positions

Of course, the main difficulty with this approach is how to identify those cut-position combinations which lead to an optimal or - at least - close to optimal concatenated alignment. Here, a heuristic based on so-called additional-cost matrices which are used for quantifying the compatibility of cut positions in distinct sequences proved to be successful.

More precisely, given a sequence s of length n and a cut position c, $0 \leq c \leq n$, we denote by $s(\leq c)$ the prefix sequence $s[1 \ldots c]$ and by $s(> c)$ the suffix sequence $s[c+1 \ldots n]$ and we use the dynamic programming procedure to compute, for all pairs of sequences s_p, s_q and for all cut positions c_p of s_p and c_q of s_q, the additional cost $C_{s_p,s_q}[c_p, c_q]$ defined by

$$C_{s_p,s_q}[c_p, c_q] = \delta(s_p(\leq c_p), s_q(\leq c_q)) + \delta(s_p(> c_p), s_q(> c_q)) - \delta(s_p, s_q)$$

which quantifies the additional charge imposed by forcing the alignment of s_p and s_q to optimally align the two prefix sequences $s_p(\leq c_p)$ and $s_q(\leq c_q)$ as well as the two suffix sequences $s_q(> c_q)$ and $s_q(> c_q)$, rather than aligning s_p and s_q optimally.

Now for given cut positions $c_1, c_2, \ldots, c_k$ we define

$$C(c_1, c_2, \ldots, c_k) := \sum_{1 \leq p < k} \sum_{p < q \leq k} C_{s_p, s_q}[c_p, c_q]$$

This is the additional cost for choosing the cut positions $c_1, c_2, \ldots, c_k$. There are heuristic algorithms which try to compute cut positions $c_1, c_2, \ldots, c_k$ minimizing $C(c_1, c_2, \ldots, c_k)$.

Assume a function $calccut$ which computes cut positions for a given set of sequences. Let Z be the cutoff value for the sequence lengths, that is, if the set of sequences to align contains a sequence of length $\leq Z$, then one applies the program MSA to the sequences to obtain an optimal multiple sequence alignment. The DCA algorithm can be described by the following recursive-function:

$$\begin{aligned} DCA(s_1, s_2, \ldots, s_k) = \; & \textbf{if} |s_i| \leq Z \text{for some } i, 1 \leq i \leq k \\ & \textbf{then return } MSA(s_1, s_2, \ldots, s_k) \\ & \textbf{else return } DCA(s_1(\leq c_1), s_2(\leq c_2), \ldots, s_k(\leq c_k) \cdot \\ & \qquad DCA(s_1(> c_1), s_2(> c_2), \ldots, s_k(> c_k)) \\ & \qquad \textbf{where } (c_1, c_2, \ldots, c_k) := calccut(s_1, s_2, \ldots, s_k) \end{aligned}$$

An implementation of the divide-and-conquer alignment procedure is the program DCA.[3] It allows to compute near SP-optimal multiple alignments of up to about 15 sequences.

Iterative Improvement of the Upper Bound

Remember the method for the computation of SP-optimal alignments sketched at the end of Section 5.4. This method, implemented in the software tool MSA, requires an upper bound μ on the optimal multiple alignment sum-of-pair score, to prune the search space. It delivers an optimal multiple alignment computing only a fraction of the overall search space, i.e. the k dimensional dynamic programming table. The method becomes faster, the better the upper bound μ. Let $MSA(\mu, s_1, s_2, \ldots, s_k)$ be the function computing an optimal multiple alignment for a given upper bound μ. The upper bound delivered by the star algorithm is provably good. In practice, the upper bound delivered by the DCA algorithm is even better.

DCA called with a small value of Z allows to quickly compute an upper bound μ for the pruning algorithm delivering the optimal alignment. However, the following observation gets us even further. DCA has a time versus quality tradeoff; the larger one chooses the parameter Z, the (provably) better is the multiple alignment one gets, while the computation time increases due to the larger optimal alignments to be computed. This motivates an iterative combination of both DCA and the optimal alignment procedure: Successively, we call DCA with increasing values of Z where, at each step, we can use the values of the

[3] `http://bibiserv.techfak.uni-bielefeld.de/dca/`

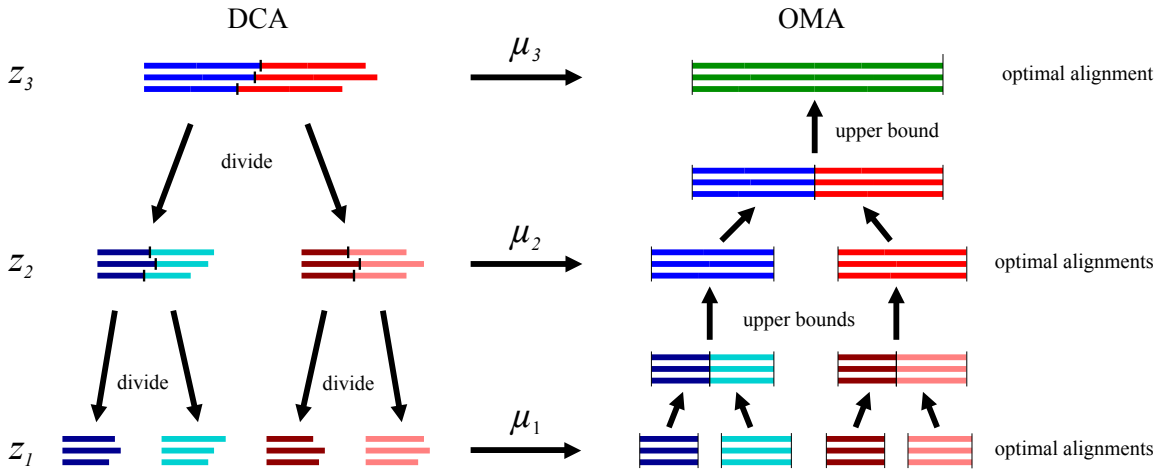


Figure 5.5: Combination of DCA and the branch-and-bound method to compute optimal alignments (OMA), where $z_1 < z_2 < z_3$ and $\mu_1 > \mu_2 > \mu_3$.

corresponding partial alignments from the previous step to compute an upper bound μ for the computation of an optimal alignment using the MSA algorithm.

This procedure has some nice properties:

- we can stop at any point of this procedure and have a heuristic alignment.
- The longer we wait, the better the alignment.
- If we are lucky, the procedure stops and delivers an optimal multiple alignment according to the sum-of-pairs model.

This procedure is the first iterative multiple alignment algorithm that provably converges to the optimal alignment.

OMA[4] is an implementation of this iterative method. More explanations can be found in [RSW99].

[4] `http://bibiserv.techfak.uni-bielefeld.de/oma/`

Bibliography

[AGM+90] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. A Basic Local Alignment Search Tool. *J. Mol. Biol.*, **215**:403–410, 1990.

[CL88] H. Carillo and D. Lipman. The Multiple Sequence Alignment Problem in Biology. *SIAM Journal of Applied Mathematics*, **48**(5):1073–1082, 1988.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, Oxford, 1994.

[CWC04] M. Cameron, H. E. Williams, and A. Cannane. Improved gapped alignment in blast. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 1(3):116–129, 2004.

[DSO78] M.O. Dayhoff, R.M. Schwartz, and B.C. Orcutt. A Model of Evolutionary Change in Proteins. Matrices for Detecting Distant Relationships. *Atlas of Protein Sequence and Structure*, **5**:345–358, 1978.

[EH88] A. Ehrenfeucht and D. Haussler. A New Distance Metric on Strings Computable in Linear Time. *Discrete Applied Mathematics*, **20**:191–203, 1988.

[Got82] O. Gotoh. An Improved Algorithm for Matching Biological Sequences. *J. Mol. Biol.*, **162**:705–708, 1982.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

[HGPH00] J.G. Henikoff, E.A. Greene, S. Pietrokovski, and S. Henikoff. Increased Coverage of Protein Families with the Blocks Database Servers. *Nucleic Acids Res.*, **28**:228–230, 2000.

[HH93] S. Henikoff and J.G. Henikoff. Performance Evaluation of Amino Acid Substitution Matrices. *Proteins: Structure, Function, and Genetics*, **17**:49–61, 1993.

[JU91] P. Jokinen and E. Ukkonen. Two Algorithms for Approximate String Matching in Static Texts. In *Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science*, pages 240–248. Lecture Notes in Computer Science **520**, Springer Verlag, 1991.

[KA90] S. Karlin and S.F. Altschul. Methods for Assessing the Statistical Significance of Molecular Sequence Features by using General Scoring Schemes. *Proc. Nat. Acad. Sci. U.S.A.*, **87**:2264–2268, 1990.

[KGR+03] A.E. Kel, E. Gößling, I. Reuter, E. Cheremushkin, O.V. Kel-Margoulis, and E. Wingender. MATCH: a tool for searching transcription factor bindi ng sites in DNA sequences. *Nucleic Acids Res.*, **31**:3576–3579, 2003.

[KS83] J.B. Kruskal and D. Sankoff. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, MA, 1983.

[LP85] D.J. Lipman and W.R. Pearson. Rapid and Sensitive Protein Similarity Search. *Science*, **227**:1435–1441, 1985.

[MM88] E.W. Myers and W. Miller. Sequence Comparison with Concave Weighting Functions. *Bulletin of Mathematical Biology*, **50**(2):97–120, 1988.

[Mye91] E.W. Myers. An Overview of Sequence Comparison Algorithms in Molecular Biology. Technical Report TR 91-29, University of Arizona, Tucson, Department of Computer Science, 1991.

[Mye98] G. Myers. The Algorithmic Foundations of Molecular Computational Biology. Unpublished Manuscript, 1998.

[NW70] S.B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino-Acid Sequence of Two Proteins. *J. Mol. Biol.*, **48**:443–453, 1970.

[PAD99] D.R. Powell, L. Allison, and T.I. Dix. A Versatile Divide and Conquer Technique for Optimal String Alignment. *Information Processing Lett.*, **70**:127–139, 1999.

[Pea90] W.R. Pearson. Rapid and Sensitive Sequence Comparison with FASTP and FASTA. In Doolittle, R., editor, *Methods in Enzymology*, volume **183**, pages 63–98. Academic Press, San Diego, CA, 1990.

[Pev00] P.A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, Cambridge, MA, 2000.

[QFWW95] K. Quandt, K. Frech, E. Wingender, and T. Werner. MatInd and MatInspector: new fast and versatile tools for detection of consensus matches in nucleotide data. *Nucleic Acids Res.*, 23(23):4878–4884, 1995.

[RSW99] K. Reinert, J. Stoye, and T. Will. Combining Divide-and-Conquer, the $\mathcal{A}^*$-Algorithm, and Successive Realignment Approaches to Speed up Multiple Sequence Alignment. In *Proceedings of GCB 1999*, pages 17–24, 1999.

[Sel80] P.H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, **1**:359–373, 1980.

[SFA99] P. Scordis, D.R. Flower, and T.K. Attwood. FingerPRINTScan: intelligent searching of the PRINTS motif database. *Bioinformatics*, 15(10):799–806, 1999.

[SM97] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing, Boston, M.A., 1997.

[SMD97] J. Stoye, V. Moulton, and A.W.M. Dress. DCA: An Efficient Implementation of the Divide-and-Conquer Approach to Simultaneous Multiple Sequence Alignment. *Comp. Appl. Biosci.*, **13**(6), 1997.

[Ste94] G.A. Stephen. *String Searching Algorithms*. World Scientific, Singapore, 1994.

[SW81] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Biol.*, **147**:195–197, 1981.

[SW96] Altschul SF and Gish W. Local alignment statistics. *Methods Enzymol*, 266:460–80, 1996.

[Ukk92] E. Ukkonen. Approximate String-Matching with q-Grams and Maximal Matches. *Theoretical Computer Science*, **92**(1):191–211, 1992.

[Ukk93] E. Ukkonen. Approximate String-Matching over Suffix Trees. In *Proc. of* CPM 93*,* LNCS 684, pages 229–242, 1993.

[Ula72] S.M. Ulam. Some Combinatorial Problems Studied Experimentally on Computing Machines. In Zaremba, S.K., editor, *Applications of Number Theory to Numerical Analysis*, pages 1–3. Academic Press, 1972.

[Wat89] M.S. Waterman. Sequence Alignments. In M.S. Waterman, editor, *Mathematical Methods for DNA Sequences*, pages 53–92. CRC-Press, Boca Raton, FL, 1989.

[Wat95] M.S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman Hall, 1995.

[WF74] R.A. Wagner and M.J. Fischer. The String to String Correction Problem. *Journal of the ACM*, **21**(1):168–173, 1974.

[WNMB00] T.D. Wu, C.G. Nevill-Manning, and D.L. Brutlag. Fast Probabilistic Analysis of Sequence Function using Scoring Matrices. *Bioinformatics*, **16**(3):233–244, 2000.